Synthèse complète d'OS

Par Snow (q220244), CC-BY-SA

Contents

1) Les processus

Un processus est un programme en cours d'exécution.

Un programme est donc un **élément passif** (un ensemble d'octets sur le disque) tandis qu'un processus est un **élément actif** (un programme en cours d'exécution).

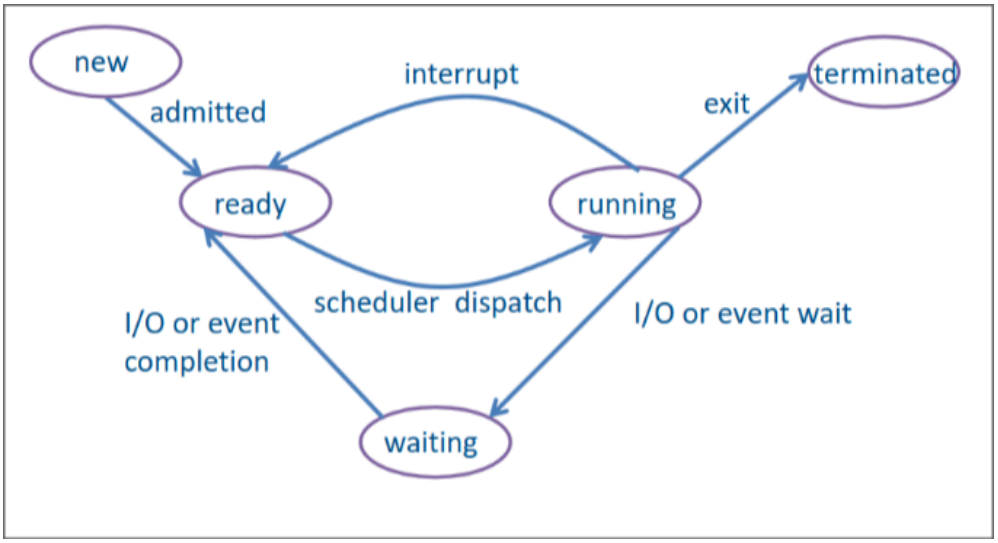
1.1) Que comporte un processus ?

- Le code du programme
- Le program counter (à quel instruction on est dans le programme, qui permet de savoir quelle sera la suivante) et les registres
- La pile (stack) et les données du programme

1.2) Informations concernant le processus

- PID (process ID) qui est l'identifiant du processus
- PPID (parent process id) qui est l'identifiant du processus parent
- Priorité du processus
- Temps CPU : temps consommé au CPU
- Tables des fichiers
- État du processus

1.2.1) État



- **new** correspond à un programme qui a été sélectionné pour être démarré, ses instructions ont été recopiées en mémoire par l'OS et un nouveau processus y a été attaché, mais pas encore exécuté, son contexte d'exécution et ses détails n'ont pas encore été préparés.
- **ready** le processus a été créé et dispose de toutes les ressources pour effectuer ses opérations
- **running** le processus a été choisi par le scheduler pour tourner, il va donc exécuter ses instructions jusqu'à écoulement du temps imparti. S'il a besoin de plus de ressource, il passe dans l'état *waiting*, s'il a terminé son exécution, il passe en état *terminated* sinon il peut encore passer en *ready* si un processus de plus haute priorité arrive.

- **waiting** le processus est en attente d'un évènement (exemple appui d'un bouton ou écoulement d'un certain temps) ou de ressources (exemple lecture de disque). Le processus ne peut rien faire pour l'instant.
- **terminated** une fois que le processus est terminé (ou a été tué), il libère la totalité des ressources qu'il a détenues.

Vous pouvez avoir plus d'information sur ce sujet en consultant ce site[1].

1.3) Pour exécuter plusieurs processus

Le système alterne très vite entre les différents états pour donner l'illusion que plusieurs processus s'exécutent en même temps.

En somme on garde en mémoire les processus, le **scheduler** va choisir les processus à exécuter; lorsqu'un processus est en attente un autre processus va être sélectionné pour être exécuté. Le but du scheduler est de maximiser l'utilisation du CPU.

1.4) Le scheduler

Le scheduler va sélectionner le processus à exécuter, c'est lui qui va alterner entre les différents états de chaques processus.

Le scheduler utilise un algorithme précis et il doit être le plus rapide possible.

Le scheduler classifie les processus selon leur type :

- Processus CPU (calculs)
- Processus E/S (I/O, entrée sortie)

On va toujours vouloir privilégier les processus entrée-sorties, qui sont ceux qui dialogues avec l'utilisateur et qui vont donner l'illusion que les choses d'exécutent en même temps.

1.4.1) Changement de contexte

Pour changer de processus on doit pouvoir sauvegarder le contexte (les données) du processus précédent.

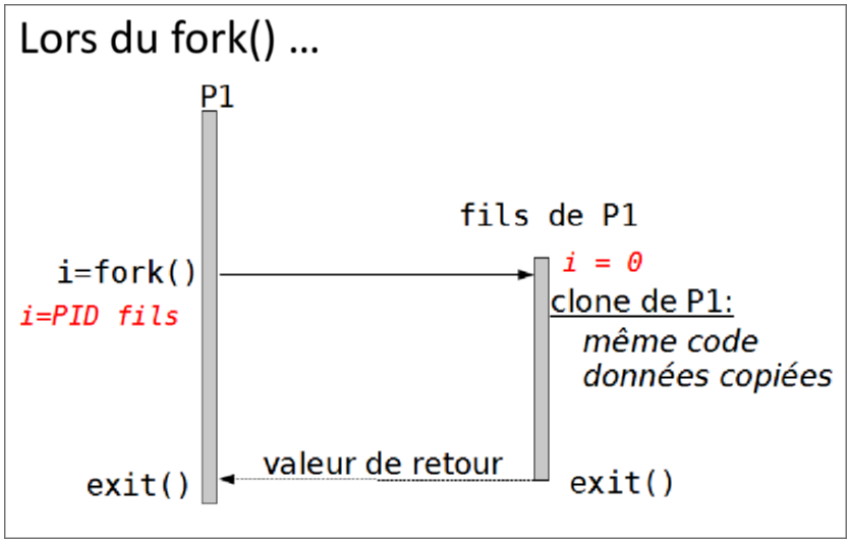
Le système va donc sauvegarder toutes les informations du processus pour pouvoir le redémarrer plus tard.

Ensuite le scheduler va sélectionner un autre processus et en charger les informations/contexte pour le démarrer.

Il va ainsi faire cela tout le temps pour alterner entre tous les processus en attente, prêts et en cours pour maximiser l'utilisation du CPU et donner l'illusion que tout fonctionne en même temps.

[1] https://eskool.gitlab.io/tnsi/processus/etats/

1.5) Création d'un processus (fork)



Pour créer un processus on utilise l'appel système *fork*. Le processus créé par un fork est appelé le processus *fils*, et le processus qui a créé le *fils* est appelé le *père*.

Le processus *fils* est un clone de son *père*, toutes les données du premier sont recopiées dans le fils.

La fonction `fork()` en C va retourner un entier :

- `-1` si une erreur est survenue (comme souvent en C, une valeur négative veut dire qu'une merde s'est passée)
- `0` pour le processus fils
- Le PID du fils pour le processus père

1.5.1) Exemples en C

1.5.1.1) Exemple simple

Voici un autre exemple :

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main (void)
{
    /* Variable pour stoquer la réponse du fork */
    pid_t pid;

    /* Fork et mise du résultat dans la variable */
    reponse_fork = fork();

    /* Si le reponse_fork est 0, alors c'est le fils qui lit l'info */
    if (reponse_fork == 0) {
```

```
      printf("Je suis le processus fils\n");

    /* Si le reponse_fork est autre chose, alors c'est le père qui lit l'info */
    } else {
      printf("Je suis le processus père et mon fils est le : %d\n", reponse_fork);
    }

    /* Fin des deux processus */
    return EXIT_SUCCESS;
}
```

Va retourner quelque chose comme :

```
Je suis le processus père et mon fils est le : 243328
Je suis le processus fils
```

1.5.1.2) Exemple plus complexe

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main (void) {
  /* La valeur de i par défault est 5 */
  int i=5;
  pid_t ret;

  /* Ce code sera exécuté uniquement sur le père */
  printf("Avant le fork() ... \n");

  /* La valeur de retour sera 0 sur le processus fils, et le pid du fils sur le processus père
*/
  ret = fork();

  /* Le code à partir d'ici sera exécuté sur les deux processus */
  printf("Après le fork() ... \n");

  /* Sur le processus fils, i sera multiplié par 5 */
  if(ret == 0) {
    i*=5;

  /* Sur le processus père, i sera additioné de 5 */
  } else {
    i+=5;
  }

  /* Le code ici sera exécuté sur les deux processus */
  printf("La valeur de i est: %d\n", i);

  /* On retourne la valeur de succès d'exécution ce qui va tuer les deux processus */
```

```
  return EXIT_SUCCESS;
}
```

Va retourner :

```
Avant le fork() ...
Après le fork() ...
La valeur de i est: 10
Après le fork() ...
La valeur de i est: 25
```

1.6) Fin d'un processus

Un processus se termine quand il n'y a plus aucune instruction à exécuter ou lorsque l'appel système `exit(int)` est appellé (cette fonction permet de renvoyer une valeur entière au processus père).

1.6.1) wait et waidpid

Un processus père peut attendre la mort de son fils à l'aide des fonctions `wait()` et `waitpid()` et peut ainsi récupérer l'entier retourné par le `exit(int)` du fils.

La fonction `wait()` va simplement attendre la mort d'un fils (peu importe lequel) tandis que la méthode `waitpid()` va attendre la mort d'un processus fils déterminé.

Les fonctions `wait` et `waitpid` retourne le pid du fils, il faut donc passer le pointeur d'une variable en argument pour récupérer les valeurs. Voici un exemple :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

int main(void) {
  char chaine[100+1];
  int compteur = 0;
  pid_t pid_fils;

  /* On crée un nouveau processus */
  switch (fork()){
    /* Si le résultat est -1 c'est qu'il y a eu un problème */
    case -1:
      printf("Le processus n'a pas été créé.");
      exit(-1);

    /* Si on est le processus fils, on demande d'entrer une chaine de caractères */
    case 0:
      printf("Entrez une chaine de caractères : ");
      scanf("%100[^\n]%*c", chaine);

      /* On retourne la longueur de la dite chaine en exit */
```

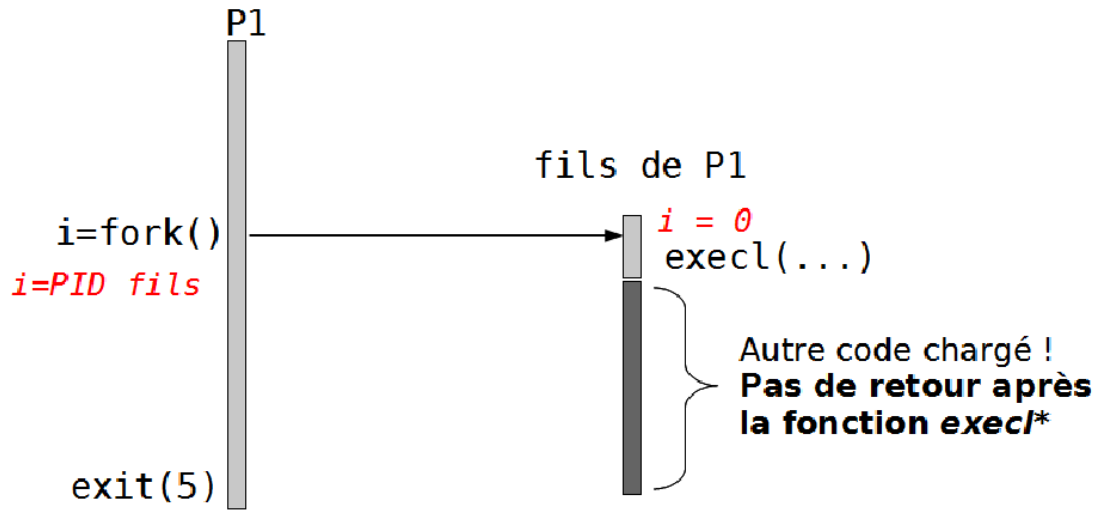
```
      exit(strlen(chaine));

    /* Si on est le processus père, on attends la mort du fils et on récupère la sortie du exit
dans une variable */
    default:
      /* On stoque le retour du exit dans une variable ainsi que le PID du fils */
      pid_fils = wait(&compteur);
      /* On extrait la longueur de la chaine depuis la sortie du wait avec WEXITSTATUS */
      printf("Enfant %d est mort. Compteur = %d", pid_fils, WEXITSTATUS(compteur));
  }

  return EXIT_SUCCESS;
}
```

1.6.2) execl

`execl` permet d'avoir de charger un autre dans le processus, une fois cette fonction execl exécuté le code du processus remplacé est perdu.



La fonction prends en paramètre, deux choses :

- Le **chemin vers le programme**
- Les **arguments du programme** ce qui commence par le chemin du programme (une deuxième fois) et qui termine par un NULL

Voici un exemple d'execl :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(void) {
```

```c
  /* On crée un nouveau processus avec fork() */
  switch(fork()) {
    /* Si fork retourne -1 c'est qu'il y a eu un problème */
    case -1: printf("Erreur fork()\n");
             exit(-1);

    /* Si fork retourne 0 c'est que c'est le processus fils, on va donc exécuter la commande ls avec execl */
    case 0: printf("Je suis le fils\n");
            /* Execl va lançer la commande "ls -l" */
            /* Le premier paramètre est le chemin vers le programme */
              /* Le deuxième paramètre est le chemin vers le programme qui va être passé en argument */
            /* Le troisième paramètre est le flag "-l" qui sera passé en argument */
            /* Le NULL termine la liste des arguments */
             if(execl("/run/current-system/sw/bin/ls", "/run/current-system/sw/bin/ls", "-l", NULL)) {
              /* Si le execl retourne -1, c'est qu'il y a eu une merde */
              printf("Erreur execl()\n");
              exit(-2);
            }
            printf("Ce code ne sera jamais exécuté car il est après le execl");

    /* Pour le processus père, on va simplement attendre que le fils ai terminé */
    default: wait(NULL);
             printf("Mon fils a terminé\n");
  }
  return EXIT_SUCCESS;

  /* Le switch n'a pas besoin de break car dans tous les cas, cela se fini par un exit, il ne peut donc rien y avoir après */
}
```

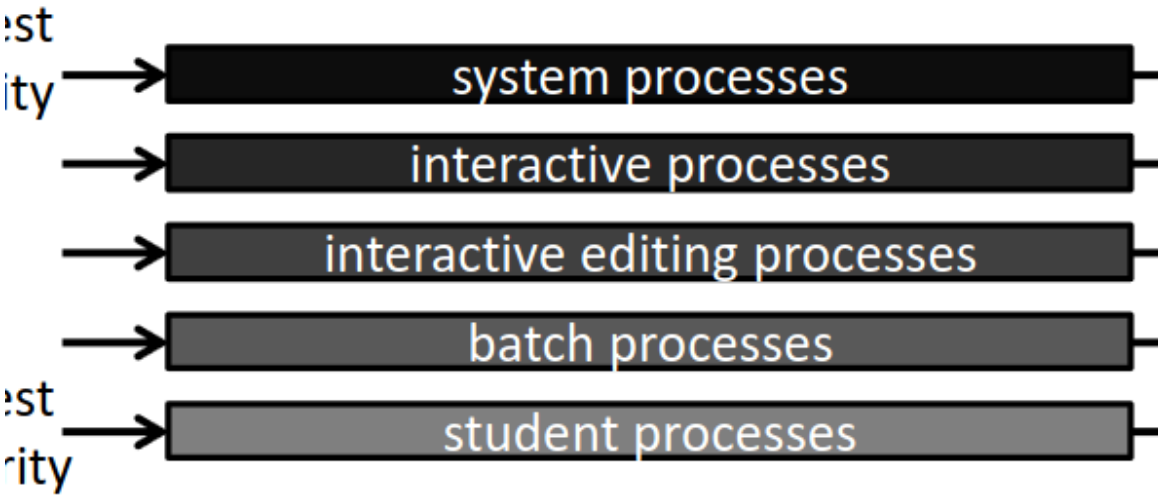
1.7) Choix des processus

Le **scheduler** du système d'exploitation doit sélectionner les processus à démarrer pour maximiser l'utilisation du CPU (généralement entre 40% et 90%) pour avoir un débit (le nmobre de processus terminés par unité de temps) important (si les processus sont trop long, le début sera faible).

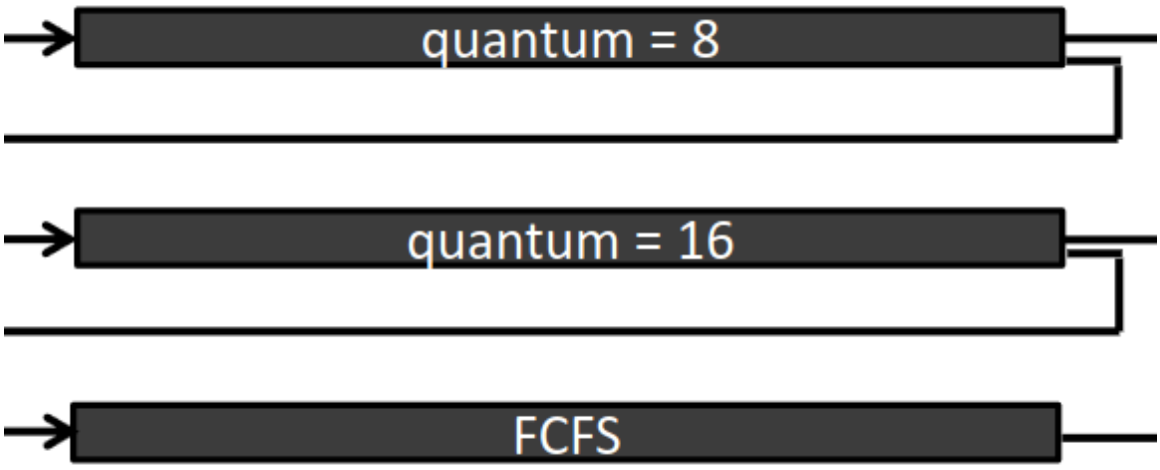
1.7.1) Algorithmes

- **FCFS (First-Come, First-Served)**, la file ici est une FIFO (first in, first out), c'est l'algorithme le plus simple à implémenter mais il peut être très long, car si le premier processus est long, il ralenti tous les processus qui suivent
- **SJF (Shortest-Job First Scheduling)**, est une amélioration du précédent, il ordonne les processus selon leur durée, ainsi les processus les plus rapides viennent au début et les plus lents à la fin. Cet algorithme est seulement possible si on sait à l'avance la durée du processus, mais aujourd'hui c'est rarement le cas.

- **Priorité**, on tient compte de la priorité d'un processus, ainsi les processus avec la priorité la plus élevée (nombre le plus petit) sont exécutés avant.
  - Cet algorithme peut être préemptif ce qui signifie qu'un processus qui tourne (running) peut être mis sur pause (en état ready) si un processus de plus haute priorité arrivé.
  - Cependant cela peut mener à de la famine car les si il y a continuellement des processus de plus haute priorité qui arrive.
    - Ce problème peut être résolu en combinant l'age et la priorité (ainsi les processus ayant attendu trop longtemps passe avant)
- **Round-Robin Scheduling (Tourniquet)**, les processus sont servi dans l'ordre d'arrivée et chaque processus reçoit le CPU pour un temps déterminé (appelé quantum), ainsi on va alterner entre chaque processus avec un temps donné (c'est donc un algorithme préemptif)
  - Si le quantum est trop grand, l'utilisateur·ice aura l'impression que le système lag car rien ne pourra être fait tant que le processus en cours est n'a pas fini son quantum
  - Si le quantum est trop petit, alors on va perdre en efficacité du CPU car beaucoup de l'énergie de calcul sera mise dans le fait d'échanger tous les processus tout le temps.
- **Multilevel Queue Scheduling**, qui s'agit d'avoir de files différentes suivant la nature du processus, une priorité et un mécanisme de scheduling propre est attaché à chaque file, il est ainsi possible d'avoir FCFS et Round-Robin sur des files différentes.



- **Multilevel Feedback Queue Scheduling**, les files sont plus dynamique (un processus n'appartient pas à une file et migrent d'une file à l'autre), chaque file a des caractéristiques précises (quantum, algorithme scheduling, etc).
  - Par exemple on peut dire qu'un processus va commencer dans une RR de quantum 8, si il n'a pas fini à la fin de son quantum il passe dans une autre file de priorité mois élevée avec un quantum de 16 et si il n'a toujours pas fini il passe dans une priorité encore mois élevée en FCFS.



1.7.2) Choix de l'algorithme

Il n'y a pas un seul bon algorithme car chaque algorithme sert à remplir un but précis.

On peut évaluer ces algorithmes selon une certaine utilisation en utilisant des modèles mathématiques, des simulations, des implémentations et des tests.

1.7.3) Quel algorithme utilisé dans l'OS ?

Sous Windows, c'est un système à 32 niveaux de priorités (préemptif).

Linux en revanche utilise un autre système de scheduling appelé CFS, vous pouvez en apprendre plus dans cette vidéo[2].

1.8) Communication IPC

Il est nécessaire que les processus communiquent entre-eux (pour le partage d'information, la répartition des calculs, la modularité et la facilité). La communication inter-process sont très courant sous UNIX et servent à résoudre ce problème.

1.8.1) Différentes options

- Fichiers, cependant c'est très lent et difficile à synchroniser
- Tube nommé ou non-nommé
- Files de messages
- Mémoire partagée, qui a l'avantage d'être très rapide
- Socket (échanges via le réseau) qui est universel

1.8.2) Les tubes

Les tubes sont des petits fichiers géré en file circulaire, ils sont si petit qu'ils sont souvent en cache (ce qui est donc très efficace). Si le message devient trop grand, il sera alors découpé en blocs.

[2]https://www.youtube.com/watch?v=MkJfuI5_hjc&t=875

1.8.2.1) Tubes non-nommés

Les tubes non-nommés sont des tubes temporaires, ils sont alloué via l'appel système `pipe()`

Il existe différents tubes standards :

- `stdin` tube de lecture (via le clavier, genre `scanf`)
- `stdout` tube de sortie (affichage à l'écran, genre `printf`)
- `stderr` est un tube de sortie pour les messages d'erreurs

Il est ainsi possible de rediriger ces tubes.

1.8.2.1.1) Opérations

- Ecriture dans le tube avec appel système `write(int h, char* b, int s)` (h étant le tube, s les premiers octets, et b le buffer)
- Lecture dans le tube avec appel système `read(int h, char* b, int s)`
- Fermeture du tube via `close(int h)`

Note les fonctions `read` et `write` retournent 0 si on tente d'écrire ou de lire un tube sans qu'il n'y a pas de processus à l'autre bout du tube.

1.8.2.1.2) Exemple

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
  int tube[2];
  char buffer[255];

  /* On crée le tube et on note les identifiant entrée et sortie dans le tableau  */
  pipe(tube);

  /* On crée un nouveau processus */
  switch(fork()) {
    case -1:
      printf("Erreur fork()\n");
      exit(-1);

    /* Pour le processus fils */
    /* Le processus fils va lire le processus tube[0] pour avoir la lecture en écriture */
    /* Le buffer va être la variable où les données vont être écrites */
    /* Et enfin 's' est la taille que l'on va récupérer */
    case 0:
      /* Si le tube est vide, read va attendre que le tube soit rempli */
      read(tube[0], buffer, 254);
      printf("Message: %s\n", buffer);
```

```
      break;

    /* Pour le processus père : */
    /* Ici on écrit "salut à toi" dans le tube en écriture (tube[1]), le buffer va donc contenir le message */
    /* Le 's' va contenir la longueur du buffer */
    /* Ainsi le message va être envoyé au fils */
    default:
      strncpy(buffer, "salut a toi", 12);
      write(tube[1], buffer, strlen(buffer));

      /* Ici on attends que le processus fils meurt, sinon le read du fils retournera 0 car il n'y aura plus le processus à l'autre bout car le programme sera terminé */
      wait(NULL);
  }
  return EXIT_SUCCESS;
}
```

1.8.2.1.3) Redirections

Par défault les 3 tubes standard sont dirigé vers le stdout (ou stderr si configuré autremenet).

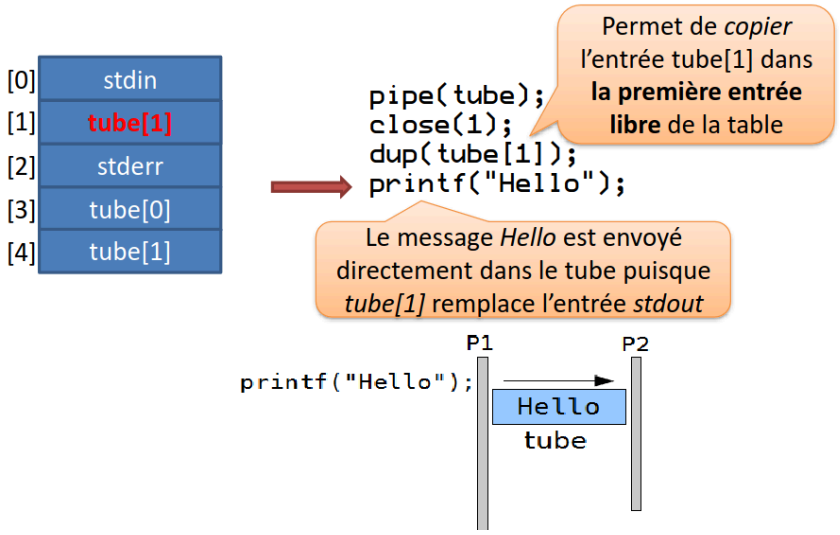
On peut également rediriger ces tubes, ainis ce qui était affiché à l'écran est alors dirigé automatiquement dans le tube ou peut être lu à partir d'un tube.

1.8.2.1.3.1) Utilisation en shell

```
# On liste les fichiers et on récupère toutes les lignes contenant "dia"
# grep prends comme entrée le résultat du ls
# C'est le shell qui va automatiquement rediriger le stdout du ls comme le stdin du grep
ls | grep "dia"
```

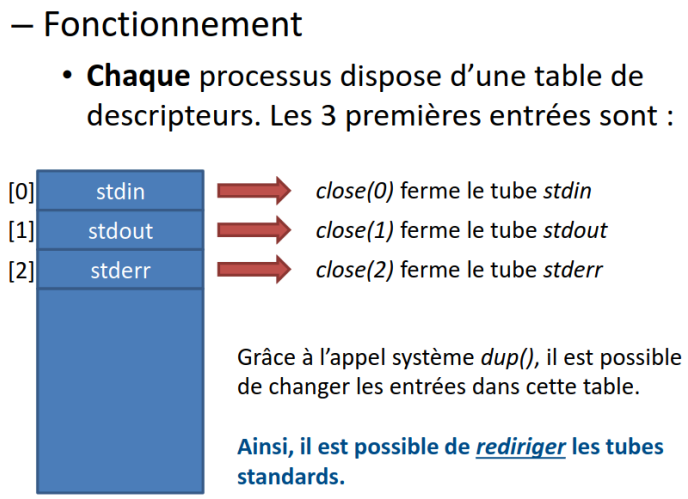
1.8.2.1.3.2) Fonctionnement

Voici un exemple de redirection :



Dans cet exemple :

1. On crée un tube
2. On ferme le stdout
3. On copie notre sortie de tube comme étant le stdout
4. On écrit dans le stdout ⟶ donc dans notre tube



1.8.2.1.3.3) Exemple en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
  int tube[2];
  char buffer[255];

  /* On crée notre nouveau tube */
  pipe(tube);

  switch(fork()) {
    case -1:
      printf("Erreur fork()\n");
      exit(-1);

    /* Pour le processus fils */
    case 0:
      /* On ferme le stdin */
      close(0);
      /* On copie l'entrée du nouveau tube pour remplacer le stdin */
```

```
      dup(tube[0]);
      /* On lit depuis le stdin (on lit donc depuis le tube) */
      scanf("%[^\n]%*c", buffer);
      /* On affiche le message stdout */
      printf("Message: %s\n", buffer);
      break;

    /* Pour le processus père */
    default:
      /* On ferme le stdout */
      close(1);
      /* On copie la sortie du tube dans le stdout */
      dup(tube[1]);
      /* On print un message vers le stdout, qui a été redirigé vers le nouveau tube */
      printf("salut a toi\n");
      /* On force le printf a se faire maintenant */
      fflush(stdout);
      /* On attends que le processus fils meurre pour éviter de causer une erreur de lecture
du tube */
      wait(NULL);
  }
  return EXIT_SUCCESS;
}
```

1.8.2.1.3.4) Autre exemple (avec execl)

Lorsque l'on redirige un pipe, le pipe reste redirigé si on exécute un autre programme par après avec `execl`, on peut donc passer l'output d'un programme dans un autre programme. Voici un exemple de pipe qui prends le résultat du ls et compte le nombre de lignes, c'est l'équivalent de `ls | wc -l`. Notez cependant que les path de ls et wc sont **très certainement** différent sur votre système, pour connaitre le PATH réel faites la commande `whereis ls` et `whereis wc`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(void) {
  int tube[2];

  /* On crée un nouveau tube */
  pipe(tube);

  /* On crée un premier enfant */
  if (fork() == 0) {
      /* On ferme le stdout */
      close(1);
      /* On redirige la sortie du tube dans le stdout */
      dup(tube[1]);
```

```
        /* On ferme les tubes pour laisser uniquement le stdin et stdout */
        close(tube[0]);
        close(tube[1]);

        /* On exécute le ls */
        execl("/run/current-system/sw/bin/ls", "/run/current-system/sw/bin/ls", NULL);
        /* Puis ce que rien n'arrive après un execl le reste du code ne s'exécutera pas */
    }

    /* On crée un deuxième enfant */
    if (fork() == 0) {
        /* On ferme le stdin */
        close(0);
        /* On remplace le stdin par le tube[0] */
        dup(tube[0]);
        /* On ferme les tubes pour laisser uniquement les stdin et stdout */
        close(tube[0]);
        close(tube[1]);

        /* On exécute wc -l ça récupère le stdin du ls */
        execl("/run/current-system/sw/bin/wc", "/run/current-system/sw/bin/wc", "-l", NULL);
    }

    /* On ferme le tube[0] et tube[1] pour laisser uniquement le stdin et stdout */
    close(tube[0]);
    close(tube[1]);

    /* On attends la mort des fils pour mourrir aussi */
    wait(NULL);
    wait(NULL);
    return EXIT_SUCCESS;
}
```

1.8.2.2) Tubes nommés

Les tubes nommés sont permanent via des fichiers spéciaux dans le filesystem.

On peut en créer un en utilisant `mkfifo(const char* nom, mode_t mode)` (le nom préise le nom du tube et le mode précise les permissions).

Les processus non-només sont liés entre père et fils, tandis qu'ici les processus nommés peuvent être utilisé par des processus qui bien que sont complètement indépendant l'un de l'autre.

Un processus peut ouvrir un tube en utilisant `open(const char* nom, int flags)` (qui est bloquant par défaut tant que le tube n'est pas ouvert des deux cotés), les flags définissent le mode d'ouverture (écriture, lecture ou les deux bien que cela ne soit pas recommandé).

On peut écrire dans un pipe avec `write(int fd, char* buf, int size)` et lire avec `read(int fd, char* buf, int size)`

On peut enfin fermer un tube avec `close(int fd)`

1.8.2.2.1) TODO Exemple

1.8.3) Mémoire partagée

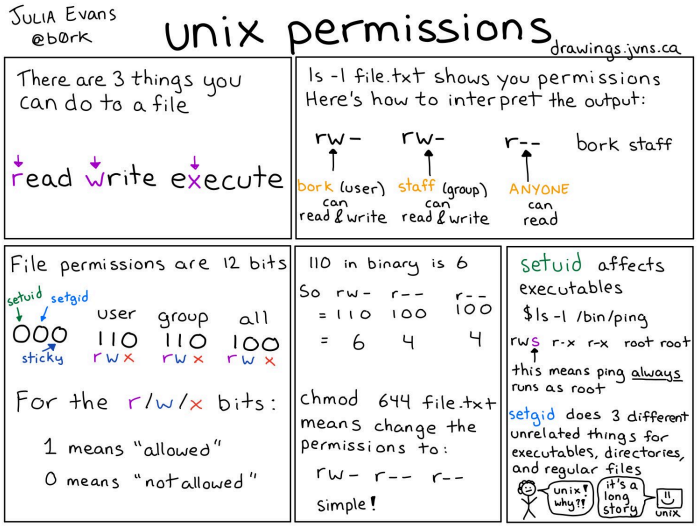
La mémoire partagée est un moyen très commun pour partager des informations entre processus, la zone de mémoire est commune à plusieurs processus. La taille est complètement configurable (comme avec malloc) et après un fork, le processus fils hérite de la mémoire partagée.

1.8.3.1) Shmget - Allocation

L'allocation se fait via `int shmget(key_t key, int s, int fl)` où

- La clé est l'identifiant de la mémoire partagée
- `s` est la taille en octets
- `fl` est le flag de permission sur la zone

1.8.3.1.1) Petite note sur les permissions



Les permissions se font via un code tel que `0664` :

- Le premier `0` indique que le nombre est en octal et non pas en décimal. Ainsi `0644` c'est `110 110 100` en binaire, et `777` est `1 100 001 001` en binaire.
- Premier `6` ⟶ est le propriétaire signifie que le propriétaire peut lire et écrire(read (1) write (1) execute (0) = 110 = 6)
- Deuxème `6` ⟶ est le groupe qui peut lire et écrire également (read (1) write (1) execute (0) = 110 = 6)
- Enfin le `4` ⟶ les autres utilisateurs peuvent seulement lire (read (1) write (0) execute (0) = 100 = 4)

1.8.3.2) Shmat - Récupération de pointeur

L'appel `shmat` permet de récupérer un pointeur vers la zone mémoire partagée. Sa signature de méthode est la suivante : `char* shmat(int shmid, char* addr, int flags)` où

- `char*` est le pointeur retourné
- `int shmid` est l'identifiant retourné par shmget
- `char* addr` est l'addresse souhaitée (généralement positionnée à 0 pour laisser le système choisir)
- `int flags` pour les paramètres de restriction (par exemple SHM_{RDONLY} donne un pointeur en lecture seule)

1.8.3.3) Shmdt - Détacher la zone

L'appel `shmdt` (qui prends en argument le pointeur) va détacher la zone mémoire sans pour autant la libérer.

1.8.3.4) Shmctl - Gérer la zone

L'appel `int shmctl(int shmid, int cmd, struct shmid_ds* ds)` permet de gérer la zone de mémoire.

- `shmid` est le descripteur de la zone retourné par shmget
- `cmd` détermine l'opération souhaitée (pour supprimer on utilise `IPC_RMID` mais il existe également IPC_{STAT} pour avoir des informations, IPC_{SET} pour modifier les valeurs associées, etc)
- `ds` contient les données en rapport avec les commandes STAT et SET

1.8.3.5) Exemple

Disons que l'on veut faire 2 programme, 1 premier écrit dans la zone mémoire et le deuxième la lit :

- Premier programme :

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

#define SHM_KEY 2324
#define K 1024

int main(void) {
  int shmid;
  char* ptr;

  /* On alloue une zone de mémoire partagée avec l'identifiant 2324, une taille de 1024 octets,
et une permission totale pour tout le monde */
  shmid = shmget(SHM_KEY, K, 0777|IPC_CREAT);

  /* Récupère un pointeur vers la zone de mémoire partagée */
  ptr = shmat(shmid,NULL,0);

  /* On copie une chaine de caractère dans la mémoire partagée */
  strcpy(ptr, "Hello !\n");

  /* On détache la zone mémoire (ce qui ne la libère pas mais permet qu'un autre processus
```

```
l'utilise) */
  shmdt(ptr);

  /* On ferme le programme */
  return EXIT_SUCCESS;
}
```

- Deuxième programme :

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>
#include <stdlib.h>
#include <sys/msg.h>
#include <stdio.h>

#define SHM_KEY 2324
#define K 1024

int main(void) {
  int shmid;
  char *ptr;

  /* On récupère la zone mémoire avec l'identifiant, la taille et le flag */
  shmid = shmget(SHM_KEY, K, 0777);

  /* Si le shmid retourné est < 0 alors c'est que la zone n'a pas été trouvée */
  if (shmid < 0) {
    printf("Erreur SHM\n");
    exit(-1);
  }

  /* On récupère le pointeur de la mémoire partagée */
  ptr = shmat(shmid, NULL, 0);

  /* On print le contenu de la mémoire partagée */
  printf("sa %d", IPC_CREAT);
  printf("Contenu : %s\n", ptr);

  /* On détache la mémoire du programme */
  shmdt(ptr);

  /* Le shmctl IPC_RMID va détruire la zone mémoire */
  shmctl(shmid, IPC_RMID, NULL);

  return EXIT_SUCCESS;
}
```

1.8.3.6) Commande ipcs pour lister les mémoires partagées

Si vous souhaitez voir la liste des zones partagées on peut utiliser la commande `ipcs`.

```
[snowcode@snowcode:~]$ gcc mempar.c

[snowcode@snowcode:~]$ ./a.out

[snowcode@snowcode:~]$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000914 4          snowcode   777        1024       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

1.9) Synchronisation

Lorsque plusieurs processus coopèrent, ils doivent souvent intéragir entre eux, ils doivent parfois attendre qu'une opération soit effectuée par un autre processus pour travailler.

Il faut donc avoir des mécanismes qui permettent d'envoyer des événements aux processus (un processus doit pouvoir attendre l'évènement).

1.9.1) Types de synchronisation

Sous UNIX, les mécanismes suivants sont mis en oeuvre pour la synchronisation :

- Les signaux
- Les sémaphores

On parlera de **point de synchronisation** lorsqu'un processus attend un autre.

1.9.2) Les signaux

Un signal est un événement capturé par un processus, c'est aussi un mécanisme simple utilisé par le système d'exploitation pour signaler aux processus une erreur (SIGILL, SIGFPE, SIGUSR1, SIGUSR2, etc).

1.9.2.1) Exemple

Voici par exemple un programme dont la fonction `sighandler` est appellée lorsque le signal SIGUSR1 est déclenché :

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void sighandler(int signum);
```

```c
/*
  Ce programme va lier la fonction sighandler au signal SIGUSR1
  Ce qui signifie que lorsque l'on lance le programme (qui contient une boucle infinie), lorsque
l'on lance le signal via "pkill -SIGUSR1 a.out" (par exemple)
  La fonction sighandler va être appellée et "SIGUSR1 reçu" va donc s'afficher à l'écran.
*/
int main(void) {
  /* Si on remplace ici SIGUSR1 par SIGINT et que l'on fait CTRL+C, on va appeller la commande
sighandler */
  if(signal(SIGUSR1, sighandler) == SIG_ERR) {
    printf("Erreur sur la gestion du signal\n");
    exit(-1);
  }

  while(1) {
    sleep(1);
    printf("Hello\n");
  }

  return EXIT_SUCCESS;
}

void sighandler(int signum) {
  printf("SIGUSR1 reçu\n");
}
```

1.9.2.2) Opérations

Il existe plusieurs opérations différentes sur les signaux :

- `signal` et `sigset` qui lient un signal à une fonction. `signal` la lie une seule fois, tandis que `sigset` la lie continuellement.
- `alarm` déclenche le signal SIGALARM au processus courrant.
- `pause` suspend le processus jusqu'a la réception d'un signal
- `kill` envoie un certain signal au processus dont le PID est donné.

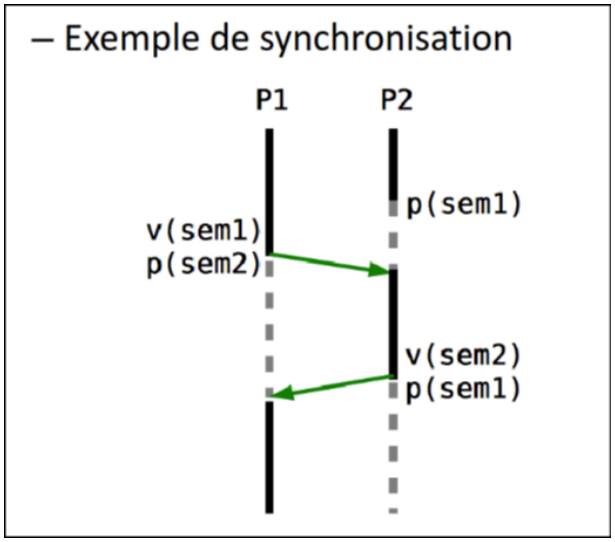
1.9.3) Les sémaphores

Un sémaphore est une variable entière en mémoire qui *excepté pour son initialisation* est accédée uniquement au moyen de fonction atomiques (ne pouvant pas être décomposée) `p()` et `v()`.

La fonction `p(sem)` va vérifier que la valeur est plus grande que zero, si c'est le cas, la variable est décrementée et l'exécution continue, si ce n'est pas le cas, alors il attend que ce soit le cas.

La fonction `v(sem)` va simplement incrémenter la variable de 1, et va ainsi réveiller tous les processus qui attendrait ce sémaphore.

Ces foncitons ne sont pas présente dans C de base il faut importer les fichiers `semadd.h` et `semadd.c` depuis l'espace de cours.



1.9.3.1) Exemple

Voici un exemple d'un programme qui communique avec un processus fils via 2 sémaphores. Il est intéressant de noter que généralement un processus ne va faire qu'une seule opération par sémaphore (par exemple que des `p()` sur sem1 et que des `v()` sur sem2 ou inversément)

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "semadd.h"

#define SEM1 12345
#define SEM2 23456

/*
  Ce programme va créer 2 sémaphores et 2 processus (un père et un fils).
  Le fils et le père vont tous les deux exécuter une boucle sauf qu'a chaque itération ils vont s'attendre l'un l'autre.
   Ainsi le père attends le sémaphore du fils (sem2) qui est émit lorsque le fils a fini son itération
  De même le fils va ensuite attendre le sémaphore du père (sem1) qui est émit lorsque le père a fini son itération

  Si on exécute ipcs -s lors de l'exécution du programme, on peut voir la liste des sémaphores créés.

  Contrairement aux signaux, on peut créer nos propres sémaphores tandis que les signaux eux sont défini par le système d'exploitation.
*/
int main(void) {
  int sem1, sem2, i;
```

```
  /* Création des sémaphores */
  sem1=sem_transf(SEM1);
  sem2=sem_transf(SEM2);

  /* Création des deux processus */
  switch(fork()) {
    case -1:
      printf("Erreur fork()\n");
      exit(-1);

    /* Boucle du fils */
    case 0:
      printf("Je suis le fils %d\n", getpid());
      for(i=0;i<5;++i) {
        printf("[FILS] Valeur de i : %d\n",i);
        sleep(5);
        v(sem2); /* Envois du sémaphore (2) au père */
        p(sem1); /* Attente du sémaphore (1) du père */
      }

    /* Boucle du père */
    default:
      for(i=0;i<5;++i) {
        p(sem2); /* Attente du sémaphore (2) du fils */
        printf("[PERE] Je suis le père\n");
        sleep(5);
        v(sem1); /* Envois du sémaphore (1) au fils */
      }
  }

  return EXIT_SUCCESS;
}
```

1.9.3.2) Semget - Allocation de sémaphores

L'allocation se fait via `int semget(int key, int nb, int flag)`, où

- La valeur retournée est un descripteur "semid"
- La clé est la valuer qui identifie le sémaphore
- Les flags définissent les permissions, comme pour les mémoires partagées `IPC_CREAT` permet de demander la création des sémaphores

On peut aussi simplifier l'allocation à partir d'une clé en utilisant `int sem_transf(int key)`, cette fonction n'est **pas officielle** mais le fichier est disponible sur HELMo Learn.

1.9.3.3) Semctl - Gestion de sémaphores

On peut gérer les sémaphores (nottament pour libérer la mémoire) en utilisant `int semctl(int semid, int semnum, int cmd, union semun attr)` où

- semid est le descripteur du sémaphore

- semnum identifie le sémaphore (généralement c'est 0 si il n'y en a qu'un)
- cmd identifie la commande (IPC_{SET}, GETVAL, SETVAL, IPC_{RMID} ou IPC_{STAT}).
- `union semun attr` est une "union" (un type de structure où chaqun des éléments partagent la même zone mémoire, ainsi ce ne peut être qu'un seul élément à la fois, un peu comme une enum en Rust). Il faut généralement définir cette structure soi-même en revanche.

1.9.3.4) Semop - Faire les opérations sur les sémaphores

`int semop(int semid, struct sembuf* sops, unsigned nsops)` est la fonction qui est derrière les fonctions `p()` et `v()`.

- semid est le descripteur du sémaphore
- sops est un tableau de structures semfus (contenant l'opération)
- nsops est le nombre d'éléménts du tableau sops

1.9.4) Section critique

C'est bien beau la synchronisation sauf que la coopération entre plusieurs processus pose également des problème si deux processus concurrents souhaite modifier les même données au même moment.

1.9.4.1) Définition section critique

On peut donc mettre en place une **section critique**, c'est un ensemble d'instructions qui devraient être exécutées du début à la fin sans interruption.

Une section critique est indispensable lorsque l'on traite des données partagée afin qu'elle soit protégée et que ces données partagées ne deviennent pas incohérente.

Par exemple, si on fait par exemple une liste chainée, elle risque de ne plus être cohérente après plusieurs modifications.

On ne peut cependant pas empêcher la concurrence entre les processus. Pour cela on va mettre en place des protections avant toute modification pour s'assurer qu'un autre processus n'est pas déjà en train de modifier la zone partagée.

1.9.4.2) Variable partagée

Celle ci consiste à partager une variable entre plusieurs processus, qui est initiallement définie à 0. Avant d'entrer dans le processus, on boucle sur la valeur de cette variable.

Si la variable est différente de 0 on boucle (et on attends). Ensuite on place la variable à 1 avant de commencer la section critique puis on la remet à 0 une fois que cela est fini.

```
while (i != 0);
i = 1;
/* Section critique ici */
i = 0;
```

1.9.4.2.1) Problème

Un gros problème peut survenir si un processus reviens dans l'état ready (par exemple avec la fin de son quantum de temps) entre l'instruction while et l'instruction de `i = 1`.

Ainsi l'autre processus peut lui aussi entrer en section critique et peut lui aussi avoir son quantum de temps qui expire durant celui ci.

Ainsi on peut donc arriver dans une situation ou plusieurs processus sont dans une section critique en même temps (ce qui est justement la chose à éviter).

Ainsi, cette méthode de protection n'est pas fiable.

En plus de cela, utiliser une boucle while comme ceci consomme inutilement du CPU.

Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020[3] à 2:25:50.

1.9.4.3) Par alternance

La protection par alternance consiste de manière similaire à la méthode précédente à avoir une variable partagée mais ou chaque processus attends une valeur différente.

Ainsi, par exemple un programme 1 pourrait avoir le code suivant :

```
while (tour != 0);
/* Section critique ici */
tour = 1;
```

Et un programme 2 pourrait avoir le code suivant :

```
while (tour != 1);
/* Section critique ici */
tour = 0;
```

Ainsi lorsque tour est à 0, le programme 1 peut exécuter sa section critique, une fois qu'elle a fini le programme 2 peut exécuter la sienne, et une fois que le progrmame 2 à fini, le programme 1 peut recommencer.

1.9.4.3.1) Problèmes

Cette méthode de protection est fiable, contrairement à la précédente. Cependant elle souffre tout de même d'assez gros problèmes...

Premièrement, elle est assez difficile à gérer, surtout si il y a plus de deux processus à synchroniser.

Et deuximèment, comme la précédente, elle est assez peu efficace car utiliser une boucle while ainsi consomme inutilement du CPU.

Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020[4] à 2:33:20.

1.9.4.4) Par fichier

La protection par fichier consiste à ouvrir et créer un fichier (appellé "lock file") en mode exclusif (c'est à dire qu'un seul processus peut accéder au fichier à la fois) pour annoncer que la section critique commence.

Puis enfin à supprimer le fichier une fois que la section critique est terminée.

[3] https://cours.swilabus.be/content/seinf20/C3
[4] https://cours.swilabus.be/content/seinf20/C3

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define FIN_SECTION_CRITIQUE 1
#define DEBUT_SECTION_CRITIQUE -1

int quid(int op, char* nom, int essais) {
  int i;

  /*
   * Quand on débute la section critique, on crée un fichier dit "lock file" en mode exclusif,
   * si cela n'est pas possible c'est qu'une section critique est déjà en cours
   */
  if(op == DEBUT_SECTION_CRITIQUE) {
    for(i=0;i<essais;++i) {
      /* Tenter d'écrire un fichier en mode exclusif (un seul processus a accès au fichier à
la fois) et renvoyer 0 en cas de succès */
      if(open(nom,O_WRONLY|O_CREAT|O_EXCL) >=0) {
        return 0;
      }

      /* Si cela n'a pas fonctionné, réessayer dans une seconde */
      else if(i<essais) {
        sleep(1);
      }
    }
  }

  /*
   * A la fin d'une section critique on supprime le lock file
   */
  if(op == FIN_SECTION_CRITIQUE) {
    /* Suppression du fichier et retourne 0 en cas de succès */
    if(unlink(nom) == 0) {
      return 0;
    }
  }

  /* Retourne -1 en cas d'erreur ou dans le cas où tous les essais ont échoués */
  return -1;
}


int main(void) {
  printf("Attente section critique\n");
  quid(DEBUT_SECTION_CRITIQUE, "program.lock", 5);

  /* Section critique ici */
```

```
  printf("Début section critique\n");
  sleep(5);

  printf("Fin section critique\n");
  quid(FIN_SECTION_CRITIQUE, "program.lock", 5);

  return EXIT_SUCCESS;
}
```

1.9.4.4.1) Problèmes

Cette solution est tout à fait fonctionnelle et fiable, cependant le fait de devoir gérer un fichier peut rendre les choses un peu compliquée, de plus cela ralenti les choses. Car pour chaque accès au fichier, le processus devra passer en état **waiting**, puis **ready**, puis de nouveau **running**.

Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020[5] à 2:37:50.

1.9.4.5) Synchronisation hardware

La synchronisation hardware consiste à utiliser des instructions assembleurs pour protéger une section critique.

Voici un pseudo-code de démonstration :

```
boolean TestAndSet (boolean target) {
    /* On copie la valeur de target */
    boolean rv = target;

    /* On met target à true */
    target = true;

    /* On retourne la copie de la valeur initiale */
    return rv;
}
```

Ainsi pour l'utiliser il suffirait de faire ceci :

```
/* On attends que le lock (variable partagée initialement à false) soit mis à false pour
continuer */
while (TestAndSet(lock));

/* Section critique ici */

/* On met le lock à false une fois terminé */
lock = false;
```

Ainsi lorsque lock est à false, TestAndSet va la mettre à true et retourner false ce qui va donc faire passer la boucle et entrer en section critique. Une fois cette dernière terminée, le lock retourne à false.

[5] https://cours.swilabus.be/content/seinf20/C3

En revanche si lock est à true, TestAndSet va retourner true et par conséquent rester dans le while, en attente jusqu'a ce que la variable soit à false.

1.9.4.5.1) Problèmes

Cette méthode est fiable mais le problème avec celle ci c'est l'utilisation du `while` qui va une fois de plus consomer du CPU pour simplement attendre.

Il est toute fois bon de noter que cette méthode est utilisée par le système d'exploitation pour gérer d'autes systèmes de protection tel que les sémaphores.

> *Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020[6] à 2:47:00.*

1.9.4.6) Sémaphore

Les sémaphores[7] permettent de très simplement protéger une section critique, voici un exemple :

```
#include "semadd.h"
#include "sys/sem.h"
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define KEY_SEM1 12345
#define KEY_SEM2 12346

int main(void) {

  int sem1, sem2;

  /* On crée 2 sémaphores */
  sem1 = sem_transf(KEY_SEM1);
  sem2 = sem_transf(KEY_SEM2);

  /* On crée un nouveau processus */
  switch (fork()) {
    case -1:
      printf("Quelque chose s'est mal passé lors de la création du processus...\n");
      return EXIT_FAILURE;

    /* Pour le fils */
    case 0:
      /* Attente du père */
      printf("En attente du père\n");
      p(sem1);

      /* Section critique */
```

[6] https://cours.swilabus.be/content/seinf20/C3
[7] id:afbc9842-6ff9-4077-aaf6-4bf191706403

```
      printf("Section critique du fils commence\n");
      sleep(3);

      /* Annonce au père qu'il a fini */
      printf("Section critique du fils se termine\n");
      v(sem2);

      break;


    /* Pour le père */
    default:
      /* Section critique */
      printf("Début de la section critique du père\n");
      sleep(3);

      /* Annonce au fils qu'il a fini */
      printf("Fin de la section critique du père\n");
      v(sem1);

      /* Attends le fils avant de supprimer les sémaphores */
      p(sem2);
      semctl(sem1, IPC_RMID, 0);
      semctl(sem2, IPC_RMID, 0);
  }

  return EXIT_SUCCESS;
}
```

Comme vu précédemment, les p et v des sémaphores sont des actions unitaires, il n'y a donc pas de risque que le processus soit arreter au millieu. L'utilisation des sémaphores est la manière recommandée de gérer des sections critiques.

Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020[8] à 2:52:00.

1.10) Les threads

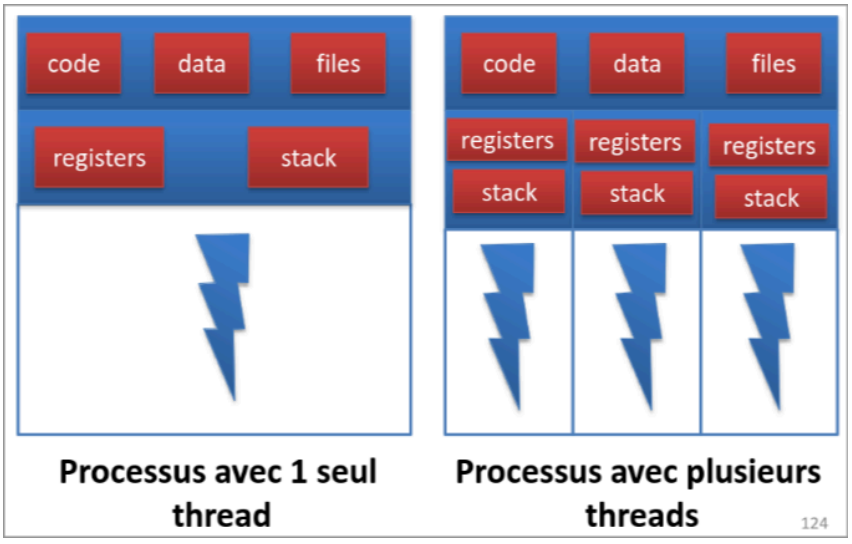
Les processus que l'on a vu n'avait qu'un seul fil d'exécution (monothread) mais il est possible d'avoir un processus avec plusieurs fils d'exécutions (multithread).

Les threads sont en somme des sortes de "mini processus".

1.10.1) Avantages

Contrairement aux processus il est beaucoup plus rapide d'en créer un nouveau, également les threads d'un même processus partagent les informations. En plus sur un système avec plusieurs coeurs l'exécution des threads d'un même processus peut se faire en parallèle ce qui offre une performance intéressante.

[8] https://cours.swilabus.be/content/seinf20/C3



1.10.2) Exemple

Par exemple on pourrait avoir un thread utilisé pour une saisie de texte, un autre thread pour l'affichage et encore un dernier thread pour vérifier les informations reçues.

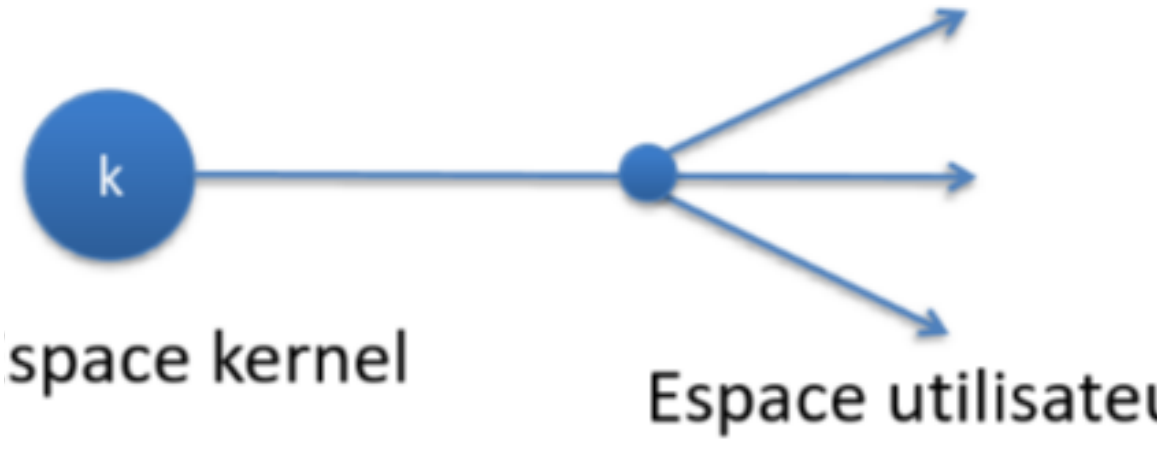
1.10.3) Modèles d'implémentations

Les threads peuvent être implémentés à deux niveaux :

- Dans **l'espace kernel**, il est alors pris en charge nativement par le système d'exploitation au même titre que les processus
- Dans **l'espace utilisateur**, il est alors supporté au travers de libraries externe

Les threads peuvent être implémentés selon plusiuers modèles :

1.10.3.1) Plusieurs à un

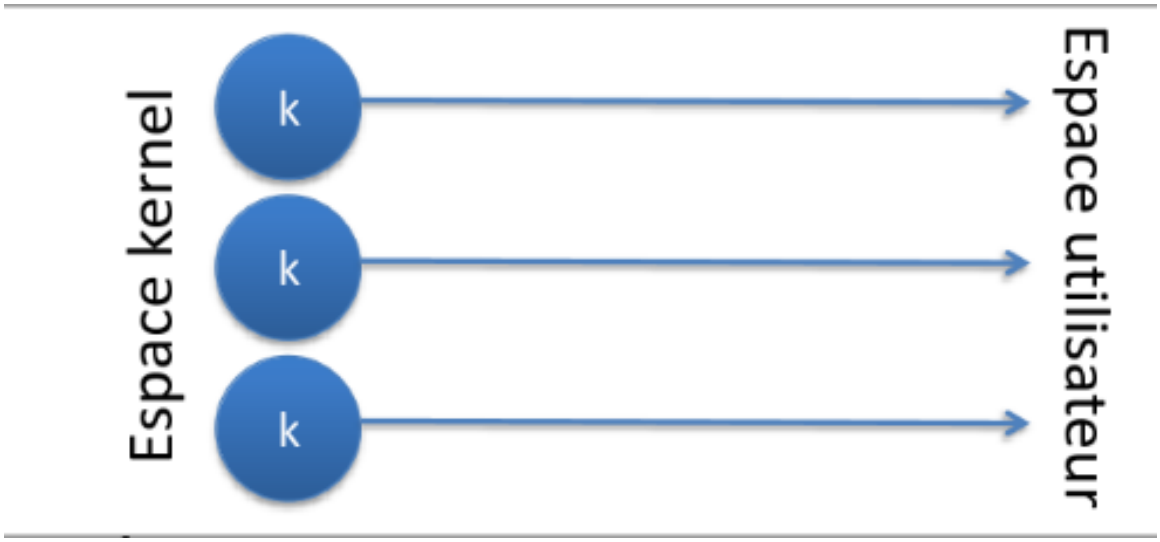


Dans ce modèle les threads sont supporté par une librarie externe, le système d'exploitation n'en a donc aucune connaissance et ne vois que le processus.

L'avantage est que sa création est rapide, cependant les inconvénients sont que un seul thread (le processus) est vu par le système, le scheduler du système n'est donc pas adapté. Si un thread réalse une opération bloquante, cela risque d'empécher tous les autres threads de travailler.

Enfin cette implémentation n'est plus vraiment courrante car elle n'est pas adaptées aux CPU multi-coeurs.

1.10.3.2) Un à un



Dans ce modèle chaque thread utilisateur est attaché à un thread kernel. Ainsi les threads sont complètement géré au niveau du système d'exploitation.

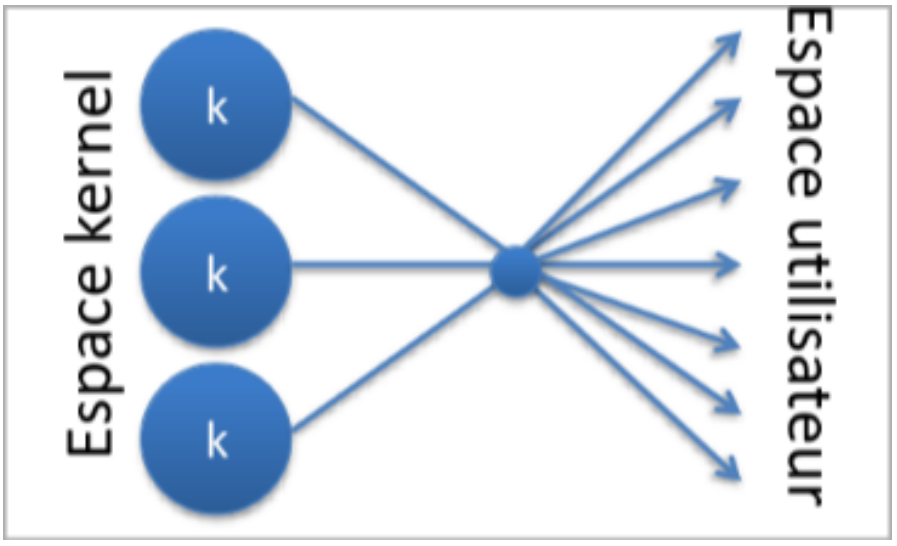
Cela a l'avantage de créer un scheduling plus avantageux et d'être compatible avec les processeurs multi-coeurs.

Cependant ce modèle est couteux pour le système car c'est lui qui doit tout gérer.

C'est ce modèle qui est nottament utilisé dans Linux. Voici par exemple la liste des threads associés au processus Firefox sur mon système.

```
[snowcode@snowcode:~]$ ps -T -p 608449
    PID    SPID TTY          TIME CMD
608449  608449 tty1     00:00:00 Web Content
608449  608453 tty1     00:00:00 IPC I/O Child
608449  608455 tty1     00:00:00 Socket Thread
608449  608456 tty1     00:00:00 HTML5 Parser
608449  608457 tty1     00:00:00 JS Watchdog
608449  608458 tty1     00:00:00 Backgro~Pool #1
608449  608459 tty1     00:00:00 Timer
608449  608463 tty1     00:00:00 RemVidChild
608449  608464 tty1     00:00:00 ImageIO
608449  608465 tty1     00:00:00 ImageBridgeChld
608449  608466 tty1     00:00:00 RemoteLzyStream
608449  608467 tty1     00:00:00 ProcessHangMon
608449  608468 tty1     00:00:00 ProfilerChild
```

1.10.3.3) Plusieurs à plusieurs



L'idée du plusieurs à plusieurs est de créer un *pool* de thread au quel les threads utilisateurs vont être assigné à la volée au cours de l'exécution.

De cette façon cela combine les avantages des deux modèles précédents. Cependant ce modèle est assez peu courrant car il nécessite que le sysètme d'exploitation soit construit autour de ce modèle car il est plus complexe à gérer que les autres.

1.10.4) Problèmes

Il y a quelques difficultés à considérer pour les threads. Par exemple :

- Que se passe-t-il en cas de `fork()` dans un thread ? Certains OS vont dupliquer tous les threads, d'autres ne vont pas le faire.

- Et avec `execl()` ? L'appel execl remplace le code du processus pour charger celui d'un autre. Ainsi le code remplace tous les threads du processus
- Pour terminer l'exécution d'un thread il y a deux possibilités, dans tous les cas il faut faire très attention pour la libération des ressources
  - Le faire de manière **asynchrone**, un thread demande la terminaison d'un autre (cela est cependant rare)
  - Le faire de manière **différée**, chaque thread vérifie régulièrement s'il doit continuer ou s'arrêter
- Quand un signal est envoyé à un processus, quels threads recoivent le signal ? Tous, certains ou un en particulier ? Cela dépends du type de signal et cela est encore une fois pas comment dans tous les OS.

1.10.5) Librarie

Pour créer des threads dans les systèmes UNIX il existe la librarie standard `pthread` dont voici quelques fonctions intéressantes :

- `pthread_attr_init` qui permet de fixer certains attributs, mais pas utile dans le cours
- `pthread_create` pour créer et démarrer un nouveau thread
- `pthread_join` pour attendre la mort d'un thread
- `pthread_exit` pour terminer l'exécution d'un thread, cette fonction permet aussi de retourner une valeur de retour à pthread join via un pointeur générique `void*`

1.10.5.1) Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* thread1(void* args) {
  int i;

  /* On trasforme le void* args en pointeur de int avec un cast */
  int* resultat;
  resultat = (int*)args;

  for (i = 0; i < 10; i++) {
    printf("[THREAD 1] %d\n", i);
    /* On ajoute le nombre courrant au résultat, attention à ne pas oublier de déréferencer le
pointeur */
    *resultat += i;
  }

  return resultat;
}

void* thread2(void* args) {
  int i;
```

```c
  /* On trasforme le void* args en pointeur de int avec un cast */
  int* resultat;
  resultat = (int*)args;

  for (i = 0; i < 24; i++) {
    printf("[THREAD 2] %d\n", i);
    /* On ajoute le nombre courrant au résultat, attention à ne pas oublier de déréferencer le pointeur */
    *resultat += i;
  }

  return resultat;
}

int main(void) {
  pthread_t tid1, tid2;

  /* On initialise les résultats à 0 */
  int resultat1 = 0;
  int resultat2 = 0;

   /* Création des threads auquels on passe les pointeurs vers les variables resultat1 et resultat2 */
  pthread_create(&tid1, NULL, *thread1, &resultat1);
  pthread_create(&tid2, NULL, *thread2, &resultat2);

  /* On attends que tous les tests se finissent */
   /* Nous n'avons pas besoin ici de récupérer la valeur de retour car on a toujours accès aux variables dont on a passé les pointeurs plus tôt, surtout que cela rends les choses très compliquées de manipuler des void** (pointeur de pointeur de valeur de type inconnue) */
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  /* Nous pouvons ensuite simplement récupérer les valeurs des résultats */
  printf("Résultat du thread 1 = %d\n", resultat1);
  printf("Résultat du thread 2 = %d\n", resultat2);

  return EXIT_SUCCESS;
}
```

1.11) Interblocages

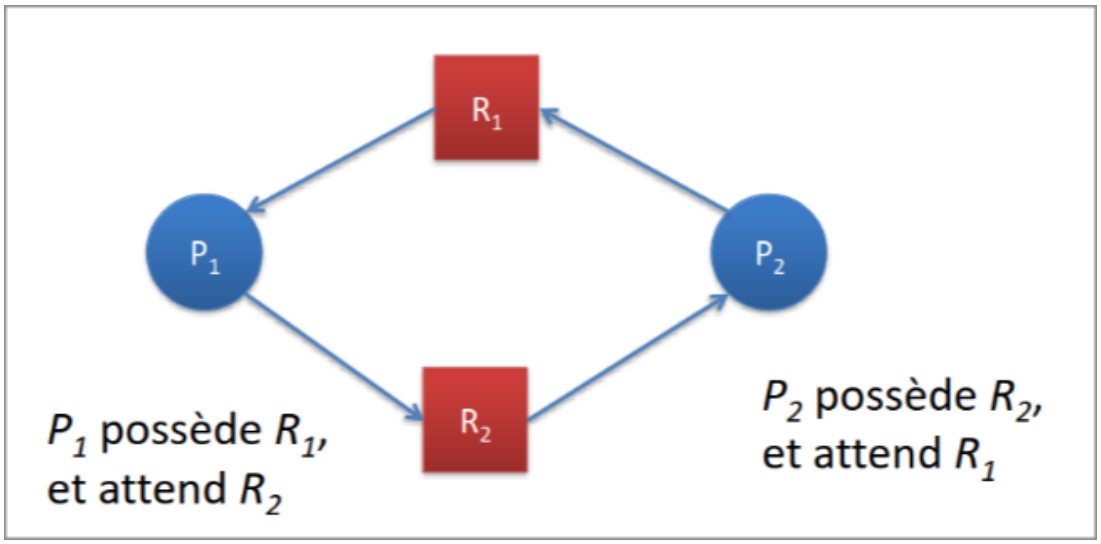
Les ressources (la mémoire, CPU, périphériques, etc) sont limitées, il faut donc gérer les ressources de manière efficace pour permettre au plus grand nombre de processus de s'exécuter.

Un interblocage peut survenir si un processus détient une ressource A qui est demandée par un autre processus détenant une ressource B qui est elle-même demandée par le premier processus.

1.11.1) Conditions d'un interblocage

Un interblocage survient lorsque ces 4 conditions sont réunies simultanément :

- L'**exclusion mutuelle**, c'est lorsque les processus utilisent des ressources qui ne peuvent pas être partagées.
- La **détention et l'attente**, les processus doivent à la fois détenir une ressource et attendre une autre ressource.
- L'**impossibilité de réquisitionner une ressource**, car c'est dégueulasse et que l'on ne peut pas savoir l'état de la ressource.
- L'**attente circulaire**, voir plus bas



1.11.2) Empécher un interblocage

Pour empécher un interblocage il faut empécher l'une des conditions d'arriver.

- L'exclusion mutuelle ? on ne peux pas empécher un processus de détenir des ressources non partageable
- La détetention et l'attente ? Il y a deux solutions pour faire en sorte que la détention et l'attente n'arrive pas en même temps :
  - Un processus pourrait demander toutes les ressources dont il pourrait avoir besoin dès le départ de son exécution
  - Lorsqu'un processus demande une nouvelle ressource, il doit libérer toutes les autres puis récupérer toutes ces ressources, plus la ressource demandée. Ce qui signifie que le processus accumule toujours plus de ressources ce qui peut créer une famine parmis les autres.
- Impossibiilité de réquisitionner une ressource ? Il n'est pas possible de s'assurer que les ressources seront dans un bon état lorsqu'elle sont réquisitionnées
- L'attente circulaire ? On peut essayer de détecter un cycle et si un cycle arrive, on peut par exemple numéroté chaque ressource et imposer aux ressources de demander les ressrouces dans l'ordre croissant de leur numéro.

1.11.2.1) Eviter l'attente circulaire

Pour éviter l'attente circulaire il faut donc savoir la quantité de ressources disponibles et occupées ainsi que les besoins de chaque processus.

Le système est dit en **état sûr** s'il est capable de satisfaire tous les processus. Et tant que le système évolue d'état sûr en état sûr, aucun interblocage ne peut survenir. Ce pendant un état non sûr ne conduit pas nécessairement à un interblocage.

1.11.2.1.1) Algorithme du banquier

Vidéo d'explication de l'algorithme du banquier[9]

1.11.2.1.1.1) Compléter les informations que l'on a

Au total pour pouvoir appliquer l'algorithme du banquier il nous faut :

- La matrice des ressources existantes (E)
- La matrice des besoins des processus (B)
- La matrice des allocations courrantes (C)
- La matrice des ressources disponibles (A), qui correspond aux ressources existantes - les allocations courrantes (E-C)
- La matrice des demandes des processus (R), qui correspond aux besoins des processus - les allocations courrantes (B-C)

Les matrices que l'on va vraiment utiliser pour l'algorithmes sont celles des allocations courrantes (C), des demandes (R) et des ressources disponibles (A).

1.11.2.1.1.2) Vérfier si le système est dans un état sûr

- Pour chaque processus on va regarder si on peut remplir sa demande (R) à partir des ressources disponibles (A).
  - Si c'est possible, alors on marque le processus comme terminé et on ajoute aux resources disponibles (A) toutes les allocations du processus terminé (C).
- On fait cela en boucle jusqu'a arriver à un résultat où tous les processus (C) sont terminés. Si à la fin tous les processus ne sont pas terminé, alors l'état n'est pas sûr.

1.11.2.1.1.3) Pour allouer depuis un état sûr

- Pour un processus qui demande une ressource, on va *hypothétiquement* diminuer les ressources disponible de la demande, on va augmenter ses ressources allouées et diminuer ses besoins. C'est à dire que l'on va faire :
  - ressources disponibles -= demande
  - besoins -= demande
  - ressources allouées += demande
- On va ensuite effectuer l'algorithme précédent pour vérifier si ce système hypothétique est en état sûr, si c'est le cas, alors on peut allouer, sinon il faut attendre

1.11.3) Détecter un interblocage

Le problème avec la première solution est que l'on ne sait pas en avance ce dont les processus ont besoins. Et il est plus efficace de simplement détecter et corriger un interblocage que d'empécher un interblocage car les interblocages restent peu fréquent.

[9] https://youtu.be/pmt1VimA0-o?feature=shared

Cet algorithme de détection et de correction va se lancer lorsque le CPU n'est plus utilisé, ce qui signifie que les processus sont en état "waiting".

1.11.3.1) Détection d'un cycle d'attente dans l'allocation

Pour détecter un interblocage il suffit de simplement connaitre les ressources disponibles, les allocations courrantes et les demandes actuelles.

Pour chaque processus en cours on va vérifer si ses demandes actuelles peuvent être satisfaite avec les ressources disponibles. Pour chaque processus trouvé, on va incrémenter les ressources disponibles des allocations courrantes et on va définir le processus comme terminé.

Si à la fin il reste des demandes non satisfaite, il y a un interblocage.

1.11.3.2) Correction d'un interblocage

Pour corriger un interblocage on va tuer un processus qui pose problème et tenter de maintenir les ressources dans un état cohérent.

On peut donc faire un rollback vers le contexte où le système était avant pour s'assurer que les ressources ne sont pas dans un état dégueulasse, du moins si on sauvegarde le contexte du processus régulièrement.

1.11.4) La politique de l'autruche

Sur certains systèmes (tel que les systèmes UNIX), c'est à l'administrateur·ice de s'occuper de gérer un interblocage et le système d'exploitation s'en fout.

2) La mémoire

La mémoire a toujours été une ressource indispensable d'un système. Elle est partagée entre tous les processus. La mémoire est une suite non structurée d'octets, le système d'exploitation ne connait donc pas la structure des informations en mémoire (qui dépendent de chaque processus).

2.1) Importance de la mémoire

Les instructions d'un programme peuvent contenir des adresses mémoires en argument, il est ainsi nécessaire qu'un processus se trouve entièrement en mémoire pour pouvoir s'exécuter. Une gestion efficace de cette mémoire est alors primordiale.

2.2) Types de mémoires

2.2.1) Registres

Les registres sont des zones mémoires attachées au processus, leur taille est généralement très réduite (de l'ordre de quelques Ko) mais sont extrêmement rapides, pouvant généralement être accédé en un seul cycle d'horloge du processeur.

2.2.2) Mémoire vive (RAM)

La mémoire vive est la mémoire principale du système. Elle est la ressource importante à gérer et est souvent présente en quantité (entre deux et 64 Go). Elle est standardisée aux normes DDR3 et DDR4. Elle est plus lente par rapport au CPU (il faut parfois plusieurs cycles d'horloge pour

récupérer une information), le CPU doit donc attendre ou mettre la donnée en cache. Cependant, cette ressource reste beaucoup plus rapide qu'un disque (SSD ou HDD).

2.2.3) Mémoire cache

La mémoire cache a pour but d'améliorer les performances générales du système en planquant en mémoire plus rapide les données fréquemment demandées.

Il existe trois niveaux de mémoire cache, le premier niveau sert à stocker temporairement des instructions et données. Les deux autres niveaux sont présents entre la mémoire RAM et la mémoire cache de premier niveau.

2.2.4) Mémoire virtuelle

La mémoire virtuelle est une mémoire simulée par le système. Elle utilise le disque dur comme RAM additionnelle. Elle est seulement limitée par la taille du disque dur, mais elle est très lente.

2.3) Isolation de la mémoire

Il est essentiel que chaque processus ait une zone mémoire réservée pour éviter des problèmes de sécurité et d'intégrité des données.

Un processus ne peut donc pas écrire dans l'espace mémoire d'un autre. S'il essaye, on aura une "Segmentation Fault". La mémoire n'est de ce fait pas partagée entre les processus sauf lorsque cela est explicitement demandé. (Voir Mémoire partagée[10] pour plus d'informations).

2.4) Translation d'adresse

Lorsqu'un programme est chargé, il est placé entièrement en mémoire à une adresse de départ. Son espace d'adressage est défini et les instructions du programme font référence à l'adresse 0.

La translation peut être effectuée à 3 moments :

- À la **compilation** (inexistant aujourd'hui), on met les adresses physiques dans le programme compilé ;

– Au **chargement** (l'adresse de départ est choisie lors du chargement du processus par le système d'exploitation) ;

- A l'**exécution**, lors de l'exécution des instructions, cela demande un hardware précis et est utilisé par la segmentation.

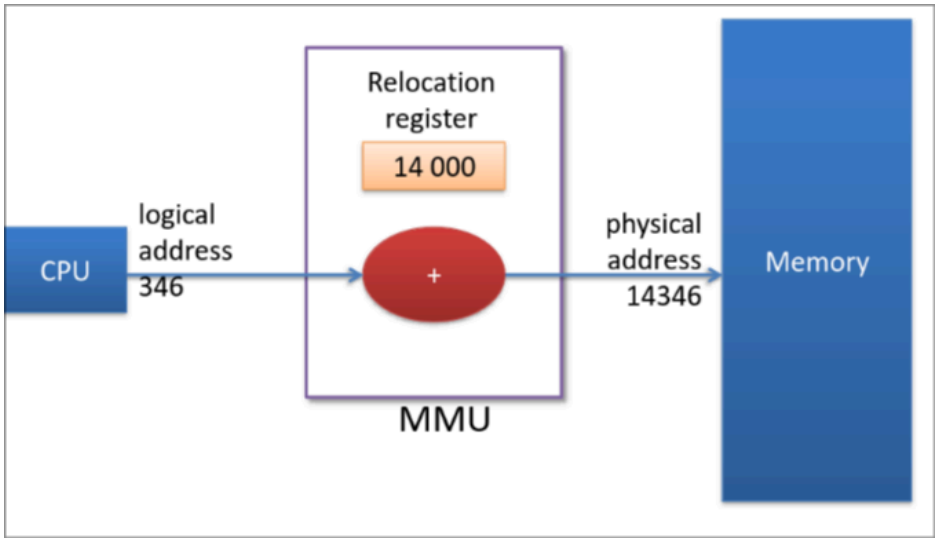
2.5) Types d'adresses

On parle d'**adresse logique** pour parler de l'adresse présente dans les programmes (qui se réfèrent à l'adresse 0).

On parle en revanche d'**adresse physique** pour parler d'une case mémoire adressable de la mémoire RAM.

La conversion entre l'adresse **logique** et l'adresse **physique** est faite par le **Memory Management Unit (MMU)** qui est un composant matériel spécialisé dans l'opération de translation.

[10] id:07344adc-b189-4a4d-8d03-b4da4433305f



2.6) Allocation de la mémoire

Le système d'exploitation, les processus systèmes et utilisateurs se trouvent dans la mémoire, il faut donc un mécanisme de protection pour isoler les processus. Ce mécanisme, c'est le MMU vu plus tôt.

2.6.1) Mono-programmation

Lorsqu'un seul processus s'exécute à la fois en même temps que le système d'exploitation, c'est le cas sur les systèmes très rudimentaires comme MS-DOS.

Étant donné qu'il n'y a qu'un seul processus à la fois, le système d'exploitation n'a pas beaucoup à faire, car il n'y a pas besoin d'isoler la mémoire (sauf pour la petite partie réservée au système d'exploitation).

Le BIOS réalise une gestion des périphériques.

2.6.2) Multi-programmation

2.6.2.1) Partitions fixes

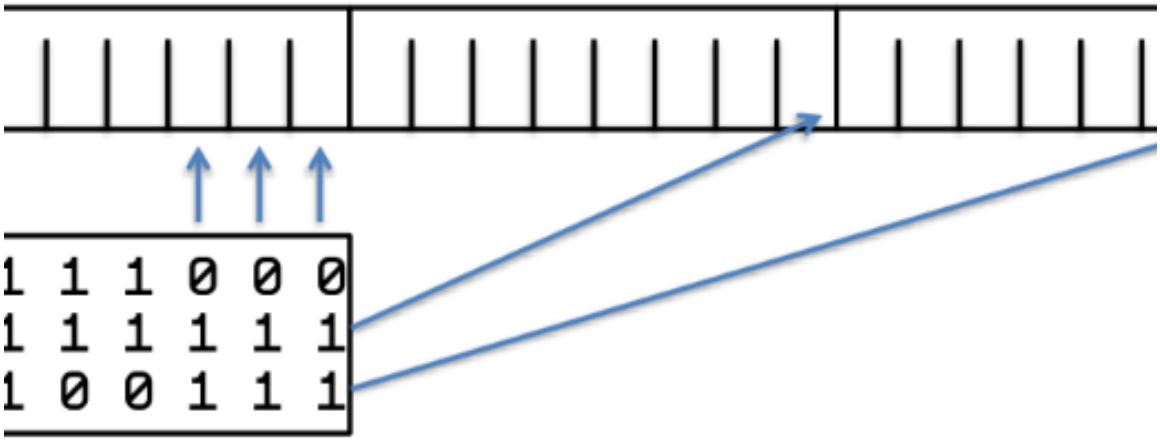
On peut pré-découper la mémoire en morceaux de taille variable (les partitions). On va donc tenter de placer chaque processus dans la plus petite partition qui peut la contenir.

L'un des problèmes de ce système est qu'il va y avoir beaucoup de restes (fragmentation interne) car si un a besoin de quatre unités, mais que la seule partition que l'on peut prendre en contient 8, on a quatre unités non utilisées.

2.6.2.2) Partitions variables

On alloue l'espace **selon les besoins des processus**, on va créer des partitions en fonction des demandes faites par les processus.

2.6.2.2.1) Table de bits (gestion de mémoire)



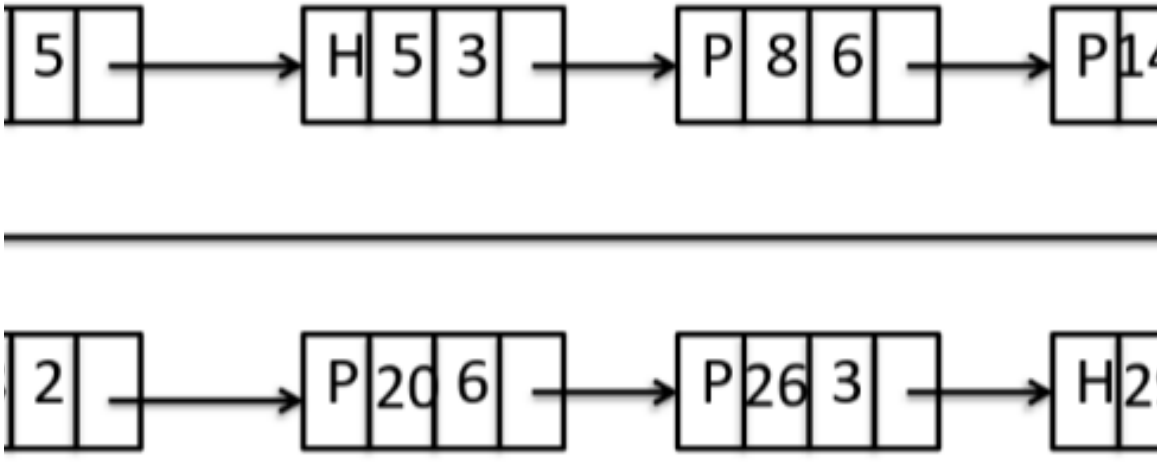
La table de bits correspond à une cartographie de la mémoire découpe en blocs d'allocations de taille fixe. Pour chaque bloc, on va noter un 1 si c'est occupé, ou un 0 si le bloc est libre dans la table de bits.

Ainsi, il suffit de seulement stocker 1 bit par unité d'allocation. Cependant, le problème est que si les unités d'allocations sont petites, alors il la table sera grande, mais ce sera plus précis.

À l'inverse, si les unités d'allocations sont grandes, la table sera plus petite, mais sera moins précise (ce qui implique donc un plus grand gaspillage de mémoire).

Un autre problème est que plus la taille est grande, moins l'allocation sera rapide, car il faudra parcourir toute la table jusqu'à trouver le segment d'unités libre recherché.

2.6.2.2.2) Liste chainée (gestion de mémoire)



Une autre méthode correspond à conserver une liste chainée des partitions mémoire. Cette liste est généralement triée suivant les adresses des partitions libres, ainsi les partitions qui se suivent dans la mémoire se suivront dans la liste.

Chaque élément de la liste chainée n'a alors qu'à stocker quatre informations : si le bloc est libre ou pas, la position du début de la partition, la longueur de la partition et le lien avec la partition suivante. Par exemple `P86` signifie qu'il y a un processus sur la partition en position huit d'une longueur de six unités.

Ce qui implique également de facilité la fusion de blocs libre, si un programme libère une partition, cette partition pourra être facilement fusionnée avec les éventuelles partitions libres adjacentes pour former une partition plus grande.

L'avantage est de rendre l'allocation beaucoup plus rapide, le désavantage est que cela crée de la fragmentation.

> *Pour en savoir plus, vous pouvez regarder sur ce site[11].*

2.6.2.2.3) Choix de la partition

Lorsque l'on recherche la partition de mémoire qui pourra accueillir un programme, il y a plusieurs algorithmes possibles.

- L'algorithme **first-fit** qui correspond à prendre la première partition libre trouvée étant suffisamment grande pour contenir le programme. C'est souvent cet algorithme qui est utilisé, car il est très rapide.
- L'algorithme **best-fit** qui correspond à parcourir toute la mémoire à la recherche de la meilleure partition. Cependant, cette méthode est assez peu efficace et crée beaucoup de fragmentation externe en créant des bouts de partitions trop petites pour être utilisées
- L'algorithme **worst-fit** qui correspond à de nouveau parcourir toute la mémoire, mais cette fois-ci à la recherche de la plus grande zone disponible afin d'éviter de créer trop de *fragmentation externe* comme le best-fit. Cet algorithme est particulièrement rapide pour les listes triées par taille.

2.6.3) Fragmentations

- **La fragmentation interne** survient lorsque la mémoire est divisée en unité d'allocation de taille fixe. Elle provient de la mémoire allouée par le SE, mais pas demandée au processus. Le SE alloue donc un surplus de mémoire par rapport à la demande du processus.
- **La fragmentation externe** survient suite aux allocations et libérations successives, la mémoire ressemble alors à un gruyère.

2.6.3.1) Solutions à la fragmentation

Pour rassembler les zones mémoires et résoudre ce problème de fragmentation, on peut utiliser le **compactage**. Cela consiste à rassembler les zones mémoires occupées et les zones libres ensembles de façon à créer de grands espaces libres.

[11] https://zestedesavoir.com/tutoriels/607/les-systemes-dexploitation/allocation-memoire/

Cela est cependant uniquement possible dans le cas de la translation à l'exécution et c'est un mécanisme assez couteux en ressource.

Une autre solution est d'utiliser de la **pagination** que nous allons voir en détail dans la section suivante.

2.6.4) Swapping

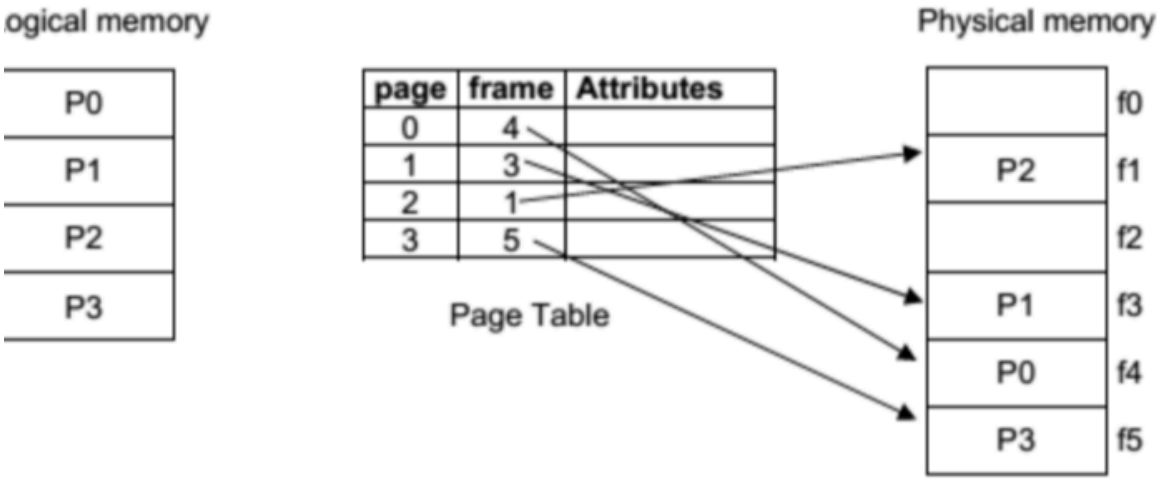
Pour pouvoir s'exécuter un processus doit se trouver entièrement en mémoire, alors lorsqu'il y a beaucoup de processus en mémoire et peu de mémoire disponible, on peut mettre un processus qui ne s'exécute pas (état ready) sur le disque dur pour libérer de la mémoire (via une partition sur le disque dur ou via un fichier "swap file").

Un processus peut être swappé lorsque son quantum de temps a expiré (retour de running à ready), si le quantum est petit il y aura beaucoup de changement de processus et donc beaucoup d'accès au disque en revanche si le quantum est grand, il y aura moins de changements de processus et moins d'accès au disque.

Les performances de ce mécanisme sont de ce fait déterminées par le temps de transfert mémoire <-> disque.

On ne peut swap que les processus auquel on n'a pas besoin d'accéder à la mémoire, c'est-à-dire les processus en état **ready** car pour les processus en état **waiting**, le système d'exploitation doit pouvoir accéder à la mémoire du processus pour y faire les entrées-sorties.

2.7) Pagination



La pagination permet d'avoir un espace adressage en mémoire physique non contigu. Elle nécessite toute fois une modification du MMU pour intégrer la *table des pages*. Elle va avoir une "table des pages" pour savoir où sont les morceaux. Cette méthode est utilisée par tous les systèmes d'exploitations actuels.

2.7.1) Fonctionnement

On va donc diviser la mémoire logique en blocs de taille fixe (les **pages**), on va faire de même avec la mémoire physique (que l'on va appeler les **frames**). Ainsi une page correspond à une frame. De ce fait, la **table des pages** va lier chaque page à une frame. L'espace de stockage pour le swapping est également découpé en blocs de même taille que les frames.

Lorsqu'un processus démarre, on va donc calculer la taille nécessaire en nombre de pages, ensuite, on va regarder le nombre de frames disponibles. S'il y a suffisamment de frames disponibles le processus peut alors démarrer, sinon le processus ne peut pas démarrer. Par exemple, si on a un processus nécessitant 10 octets et que chaque page fait quatre octets, cela nécessitera 3 pages. Sauf que s'il n'y a que 2 frames disponibles le processus ne pourra pas démarrer.

Les frames sont gérées par le système d'exploitation, ce dernier va garder la liste des frames libres et occupées et le nombre de frames disponibles. Le SE doit aussi mettre en place une protection pour empêcher un processus de dépasser son espace d'adressage.

2.7.2) Effet sur la fragmentation

Ainsi, la fragmentation externe n'existe plus, car bien que l'adressage soit vu comme contigu au niveau du programme, elle ne l'est pas au niveau physique.

La fragmentation interne, elle sera d'en moyenne 1/2 page par processus, comme on ne peut pas assigner des demi-frames, donc si un processus nécessite 10 octets, mais qu'une frame fait 4 octets et que chaque page correspond à une frame (en conséquence chaque page fait 4 octets également), on devra alors assigner 3 pages pour le processus de 10 octets, ce qui laisse ensuite deux octets (soit une demi-page) non utilisée.

2.8) Segmentation

Un·e utilisateur·ice voit un programme comme un ensemble d'instructions, de données, de fonctions, etc. Bref, un ensemble de blocs distinct dont les données ne sont pas mélangées avec les autres.

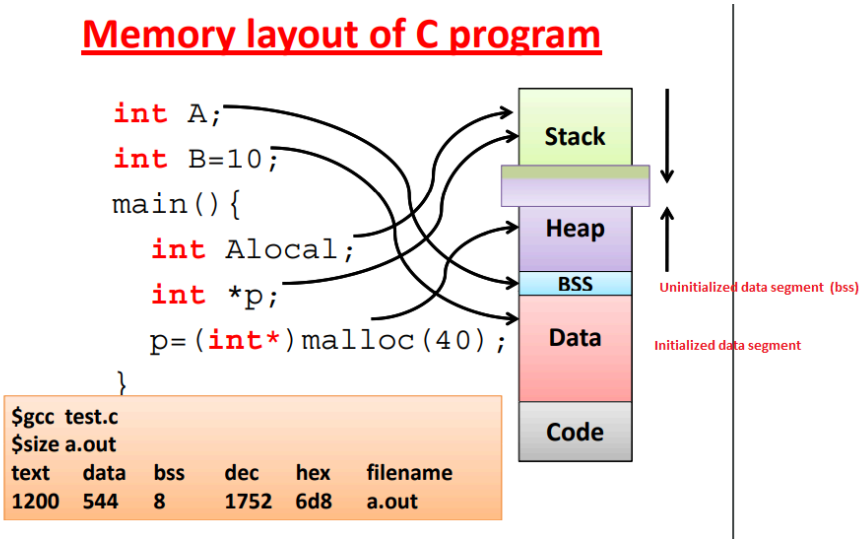
Il convient donc de traduire cette vue à l'intérieur du système d'exploitation. Pour cela, on va diviser l'espace d'adressage du processus en segments. Chaque segment a un nom, un numéro et une taille.

2.8.1) Les différents segments

Ainsi dans une architecture Intel x86, on peut avoir :

- Le **Code Segment (CS)** qui désigne le segment qui contient les instructions du programme (lecture seule)
- Le **Data Segment (DS)** qui contient les données du programme (variables globales, variables `static`)
- Le **Stack Segment (SS)** qui contient la pile du programme (variables locales, appels, etc)
- Le **Extra Segment (ES)** qui est un segment supplémentaire défini par le·a développeur·euse du programme

De manière plus descriptive, on peut définir la mémoire du point de vue d'un programme comme ceci. À noter cependant que, cela est surtout comme ça que fonctionnaient les anciens systèmes, mais comme on va le voir dans la suite ce n'est plus exactement comme ça que cela fonctionne.



Avec la segmentation comme ceci, une adresse logique devient ainsi `<segment-number, offset>`.

2.8.2) Avantages

En utilisant la segmentation, le système offre ainsi une protection adaptée car chaque segment contient des données d'une même nature et ne peuvent donc pas intervenir dans d'autres données. On ne peut pas par exemple accéder aux instructions du programme depuis le *data segment*.

Également, les segments peuvent être partagés entre plusieurs processus (par exemple avec les libraires systèmes) et ainsi faire une économie d'espace.

2.8.3) Désavantages

Cela fait un retour à la fragmentation, étant donné que les segments sont de taille variables, on retrouve de la segmentation externe. C'est pour cela qu'il faut combiner la segmentation avec la pagination.

2.9) Segmentation et pagination

Pour combiner les avantages de la segmentation[12] avec les avantages de la pagination[13], on va regarder à l'exemple du fonctionnement de l'architecture des processeurs Intel 32 bits.

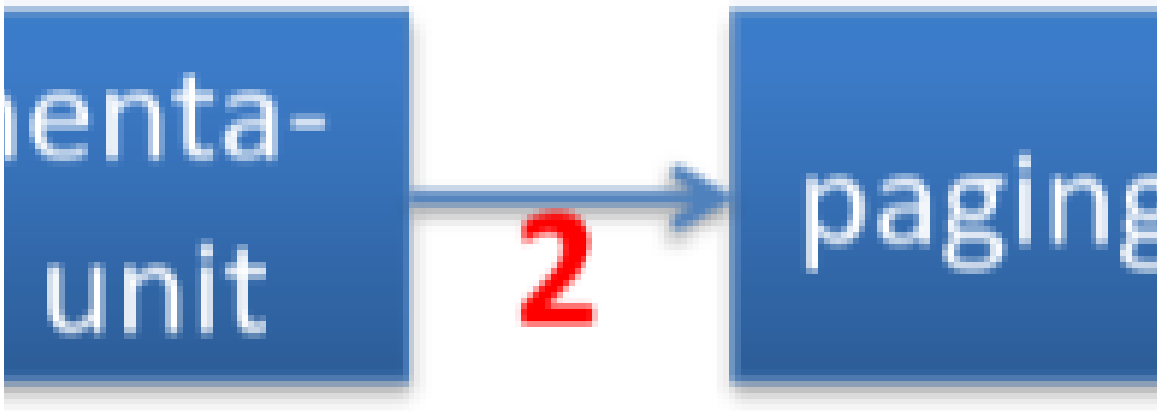
Ce processeur peut gérer un maximum de 16 384 segments de maximum 4 Go. L'espace d'adressage est découpé en deux parts égales. Une partition pour les segments privés (par exemple, processus) et une autre pour les segments partagés (par exemple, librairie partagée).

Il y a également deux tables, la LDT (Logical Descriptor Table) pour la partition des segments privés, et la GDT (Global Descriptor Table) pour les segments partagés.

[12] id:9177b160-40cb-4891-86d8-e8b21eda9c05
[13] id:62171119-bb1b-46fa-a227-a04ec4a48a1c

2.9.1) Conversion vers adresse physique

Pour passer de l'adresse logique à l'adresse physique, il va falloir passer par deux intermédiaires :



- L'**unité de segmentation** qui transforme l'adresse logique en adresse linéaire
- L'**unité de pagination** qui transforme l'adresse linéaire en adresse physique.

2.9.1.1) Unité de segmentation

L'adresse logique est composée d'un sélecteur et d'un déplacement. Le sélecteur est lui-même composé de 3 informations :

- Le numéro du segment (noté *s*)
- Si c'est un segment privé ou partagé (LDT ou GDT) (noté *g*)
- Des informations de protection (noté *p*)

À partir de cette adresse logique, l'unité de segmentation construit l'adresse linéaire.

2.9.1.2) Unité de pagination

L'adresse linéaire est composée de 3 informations,

- Le **directory**, qui indique la table des pages à utiliser parmi le répertoire des tables de pages du processus courant
- La **page**, qui indique la page dans la table des pages
- Le **offset** (déplacement), qui est le même que celui cité dans l'adresse logique.

À partir de cette adresse linéaire, l'unité de pagination construit l'adresse physique comme vu précédemment dans le chapitre sur la Pagination[14].

[14] id:62171119-bb1b-46fa-a227-a04ec4a48a1c

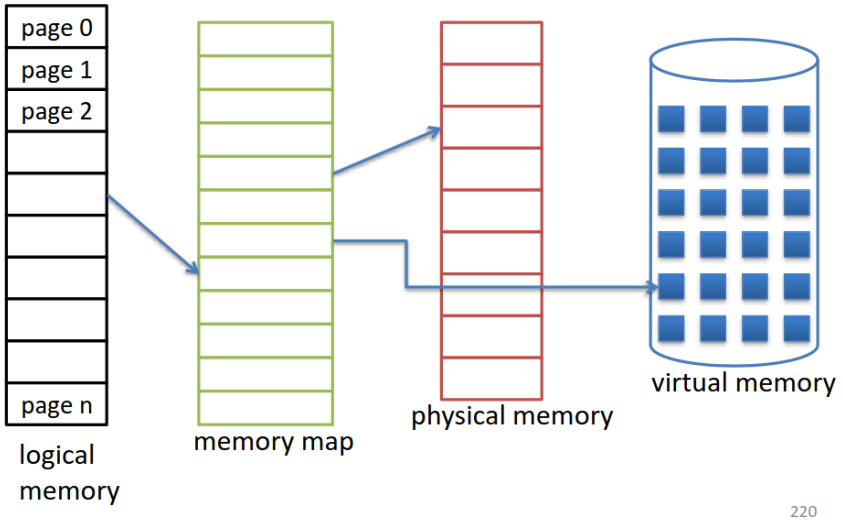
2.10) Mémoire virtuelle

Il a été dit précédemment qu'il faut qu'un programme soit entièrement en mémoire pour pouvoir s'exécuter comme processus, alors comment faire lorsque le programme est plus gros que la taille de mémoire vive ?

Ce qui est utilisé par les systèmes aujourd'hui c'est la mémoire virtuelle, il s'agit d'utiliser le disque dur comme mémoire supplémentaire à la RAM.

Cela est bien sûr plus lent, surtout si le stockage est un disque dur (et pas un SSD), mais il permet de faire tourner de très gros programmes (ou juste énormément de processus) avec peu de RAM.

2.10.1) Utilisation de la pagination



Cela tombe bien, car la pagination[15] est justement très adaptée à cela, car il suffit de lier de l'espace sur le disque à la table des pages (*memory map* sur le schéma). Ainsi la mémoire virtuelle sur le disque va être divisée en pages.

La table des pages doit bien sûr être adaptée pour pouvoir disposer d'un bit indiquant si la page se trouve sur le disque ou sur la mémoire physique.

Et le stockage/disque doit avoir une partie prévue pour contenir la mémoire virtuelle, celle-ci est appelée "swap device". Il s'agit d'un fichier sous Windows (Pagefile.sys), et d'un fichier ("swapfile") ou d'une partition ("swap partition") sous Linux.

2.10.2) Fonctionnement

Avec la mémoire virtuelle, les pages du processus à exécuter se trouvent sur le disque dur. Lors du chargement du programme, **seules les pages nécessaires** sont placées en mémoire physique.

Le changement entre mémoire virtuelle et mémoire physique est effectué lorsqu'une page non chargée est demandée. Lors du chargement, le système alloue quelques pages et les charge depuis le disque.

[15] id:62171119-bb1b-46fa-a227-a04ec4a48a1c

Ainsi lorsqu'une page est demandée, on regarde dans la table des pages pour voir si elle est présente en mémoire (bit valide), si oui tout va bien et tout se passe comme d'habitude. En revanche si ce n'est pas le cas (bit invalide) alors, il s'agit d'un **défaut de page**.

2.10.2.1) Traitement d'un défaut de page

Quand un défaut de page survient, il faut le résoudre le plus vite possible pour ne pas impacter les performances du système :

1. On vérifie si la page demandée est présente dans l'adressage du programme, si ce n'est pas le cas alors c'est un *memory overflow* ce qui provoque un arrêt du processus.
2. Si la page existe, ainsi on trouve une frame libre en mémoire physique et récupère la page depuis le disque
3. On ajuste la table des pages pour indiquer la page comme valide et indiquer sa position dans la mémoire physique
4. On redémarre le processus à l'instruction qui a causé l'erreur

Les défauts de pages faisant des appels d'entrée-sortie au disque, il est important d'en avoir le moins possible afin de ne pas trop impacter les performances du système.

2.10.2.2) Que faire quand il n'y a plus de frames ?

Lorsqu'un défaut de page survient, mais qu'il n'y a plus de frame disponible sur la mémoire physique, il faut alors que le système en libère une.

Pour cela, il faut que le système choisisse quelle frame remplacer, l'écrivent sur le disque et mettent à jour la table afin de libérer une frame pour résoudre le défaut de page.

2.10.2.2.1) Algorithmes de choix des pages victimes

Il est important d'utiliser un bon algorithme pour choisir les pages qui seront mises en mémoire virtuelle, car sinon cela risque d'augmenter le nombre de défauts de page et donc de considérablement ralentir le système.

Voici une comparaison de plusieurs algorithmes :

- **FIFO** (First In, First Out) la page qui est sélectionnée est la page la plus ancienne. Cependant, la performance de cette méthode n'est pas très bonne, car ce n'est pas parce qu'une page est vieille qu'il ne faudra pas y accéder souvent.
- **OPT** (Remplacement Optimal), cela consiste à essayer d'avoir le taux de faut de page minimal en éliminant la page qui ne sera plus nécessaire avant longtemps. Cependant, ce mécanisme est dur à implémenter, car on ne peut pas savoir en avance ce qui sera nécessaire. Ce mécanisme sert uniquement de base théorique aux autres systèmes.
- **LRU** (Least Recently Used) consiste à choisir la page la moins récemment utilisée. Pour implémenter cela, il faut soit y ajouter une timestamp (ce qui rendra la chose moins efficace), ou alors mieux utiliser un système de pile, ainsi dès qu'une page est utilisée, on la met en haut de la pile, comme ceci, on sait que la page la moins utilisée sera forcément la dernière de la pile.
  - **LRU approximé**[16] est une des variantes du LRU. L'idée est de définir un bit de référence pour chaque page. Initialement le bit est à 0 pour tous, dès qu'une page est accédée, le bit est mis

à 1. Lorsque tous les bits sont à 1, on met tout à 0 sauf la dernière. Ainsi lorsqu'un défaut de page survient, la page victime sera la première page avec un 0 (vous pouvez trouver plus d'information dans cette vidéo[17]).
- **LRU avec octet de référence**, l'idée ici est d'utiliser la méthode précédente, mais à intervalle régulier collecter le bit de référence pour le placer dans un octet de référence. Ainsi lorsqu'il faut choisir quel page sera la victime, il suffit de prendre celle qui a l'octet de référence le plus bas. Et dans le cas où il y en a plusieurs avec la même valeur, prendre le premier.
- **Algorithme de la seconde chance**, l'idée est de stocker 2 bits par page, une pour le bit de référence (voir LRU approximé) et l'autre pour un bit de modification. Le bit de modification est mis à un si la page a été modifiée sur la mémoire physique, mais pas encore sur la mémoire virtuelle. À l'inverse, elle est mise à 0 si elle est égale à la mémoire virtuelle. Ainsi, on peut savoir si une page a été utilisée et si elle nécessite une écriture sur le disque. Le meilleur choix étant de choisir une page qui n'a pas été utilisée récemment ET qui n'a pas été modifiée.

- **LFU** (Least Frequently Used) consiste à compter le nombre de fois qu'une page est utilisée et à choisir la page avec le plus petit compteur. Le problème est que ce n'est pas par ce qu'une page a été très utilisée qu'elle le sera toujours. À l'inverse ce n'est pas par ce qu'une page n'a pas beaucoup été utilisée qu'elle ne va pas le devenir.
- **MFU** (Most Frequently Used) consiste, elle aussi, à compter le nombre de fois qu'une page est utilisée et à choisir la page avec le plus *grand* compteur en réponse à l'observation statistique du LFU. Cependant, ces deux algorithmes sont peu utilisés, car ils sont peu efficaces.

2.10.3) Amélioration des performances

2.10.3.1) Écriture retardée des pages

Pour améliorer les performances en cas de défaut de page, on peut garder un ensemble de page (appelée **pool**) qui sont toujours immédiatement disponibles. Ainsi, lorsqu'une page victime nécessite d'être sauvegardée (bit de modification à 1 dans l'algorithme de la seconde chance), il suffit de donner une page du pool au processus qui est donc utilisable directement. Ensuite, on sauvegarde la page victime dans la mémoire virtuelle et cette page intègre ensuite le pool.

2.10.3.2) Allocation des frames

Il faut avoir une allocation raisonnable de la mémoire physique (frames), si trop de frames sont alloués au processus, on risque de tomber vite à court de frame et ainsi faire beaucoup de défauts de pages. Si trop peu de mémoire frames sont allouées, alors on tombe vite dans des défauts de page.

Alors il faut trouver un moyen d'allouer les frames de manière à satisfaire le plus possible tous les processus.

- La première méthode est celle de l'**allocation équitable**, on divise le nombre de frames disponible par le nombre de processus actifs. Cette méthode n'est pas intéressante, car cela créera un gaspillage pour les petits processus et une mauvaise performance pour les plus gros.

[16] https://github.com/karlmcguire/plru
[17] https://www.youtube.com/watch?v=8CjifA2yw7s

- La deuxième méthode est celle de l'**allocation proportionnelle**, les frames sont allouées proportionnellement à la taille du processus. Il faut toute fois tenir compte de la multiprogrammation, si de nouveaux processus sont créé le nombre de frames par processus va diminuer et si des processus se terminent les frames sont de nouveau disponibles. Il faut aussi tenir compte de la priorité des processus, un processus de haute priorité doit s'exécuter vite, plus il a de ressources plus, il va s'exécuter vite et donc plus vite ces frames seront libérées.

2.10.3.2.1) Trashing

L'allocation des frames est ainsi un problème compliqué, si on n'alloue pas assez de frame à un processus, le processus va passer beaucoup de temps à transférer des informations. Le **trashing** c'est lorsqu'un processus n'a pas suffisamment de frames allouées pour s'exécuter correctement et passe ainsi plus de temps à récupérer des pages qu'à s'exécuter.

La cause de ce problème est que, comme on a vu avec les processus, pour utiliser au mieux le CPU lorsque ce dernier est peu utilisé, le système va augmenter le degré de multi-programmation[18], ce qui va diminuer le nombre de frames disponibles et donc augmenter le nombre de défauts de pages. Sauf que pour résoudre le défaut de page les processus concernés devront être mis dans l'état **waiting** en attente de l'entrée-sortie, ce qui va ainsi de nouveau réduire l'utilisation du CPU. Ainsi le cycle vicieux se répète.

2.10.3.2.1.1) Modèle de la localité

Une manière de résoudre ce problème est d'utiliser le **modèle de la localité**.

Une localité, c'est un ensemble de pages qui sont activement utilisées ensemble. Un programme est ainsi composé de plusieurs localités différentes qui peuvent se chevaucher. Lorsqu'un processus s'exécute, ce dernier va de localité en localité.

Par exemple, lorsqu'une fonction est appelée, cela définit une nouvelle localité et quand l'on quitte cette fonction, on entre dans une autre localité.

Ainsi si on arrive à identifier la localité en cours, on peut alors charger suffisamment de frames pour les accommoder, cela va créer des défauts de pages jusqu'à ce que toutes les pages de la localité jusqu'à ce qu'elle soit toutes en mémoire, ensuite elles ne feront plus aucun défaut de page jusqu'à ce qu'on change de localité.

Si on n'alloue pas suffisamment de frames par rapport à la localité courante le système va faire du trashing.

3) Systèmes de fichiers

3.1) Introduction

3.1.1) Définition d'un fichier

Un **fichier** est une collection nommée d'information en relation. Pour l'utilisateur·ice c'est un moyen de conserver des informations.

[18] id:6fe2589e-6ea3-44ec-8fb8-9ceabc15c24f

Le contenu d'un fichier est défini par son type (et donc sa structure), le programme qui l'a créé et les informations placées par l'utilisateur·ice.

Il existe plusieurs médias qui peuvent être utilisés pour stocker des fichiers de manière "définitive". La qualité de ce dernier est importante, par exemple le stockage à long terme sur les disques optique est difficile à garantir.

Le système d'exploitations propose, grâce aux fichiers un moyen d'utiliser l'espace de stockage. Le fichier est une abstraction nécessaire qui permet la représentation informatique de concept existants, et cela, de manière uniforme et simple d'accès.

Un fichier a plusieurs **attributs** tel que, le nom, l'identifiant, le type, les permissions, la localisation, la taille, la date de création et le propriétaire.

3.1.2) Opérations sur les fichiers

Les opérations sur les fichiers sont,

- La **création** qui alloue de l'espace de stockage et stocke les informations concernant le fichier.
- La **lecture** qui est un appel système qui permet de lire le contenu d'un fichier et de stocker le résultat en mémoire.
- L'**écriture** qui est un autre appel système qui permet d'écrire dans un fichier des informations présente en mémoire.
- Le **positionnement** qui est un déplacement du pointeur de position centrale (pour pouvoir lire ou écrire à un certain endroit).
- La **suppression** qui correspond à supprimer le fichier ou une partie de celui-ci

3.1.3) Ouverture d'un fichier

L'**ouverture** d'un fichier est l'opération indiquant au système d'exploitation que le programme souhaite accéder à un fichier. Lors de cette ouverture le système mémorise les informations sur le fichier, vérifie si l'accès est autorisé et initialise les structures internes et place le pointeur de position au début.

Une fois que le traitement est fini, le fichier est **fermé**, ce qui va supprimer les structures internes.

Ce système d'ouverture et de fermeture soulage fortement le système d'exploitation, car sans ça il faudrait répéter toutes les procédures d'ouverture et fermeture à chaque opération sur le fichier, tandis qu'ici, on le fait qu'une fois.

Il est aussi bon de noter que plusieurs utilisateurs doivent pouvoir ouvrir les , il faut donc mettre une structure efficace en mémoire pour éviter les informations en double.

3.1.4) Informations à retenir sur les fichiers utilisés

Les informations à retenir sur le fichier sont, le mode d'ouverture (lecture seule, lecture-écriture, écriture seule, etc), l'emplacement vers le premier bloc du fichier, la position courante, et les permissions d'accès.

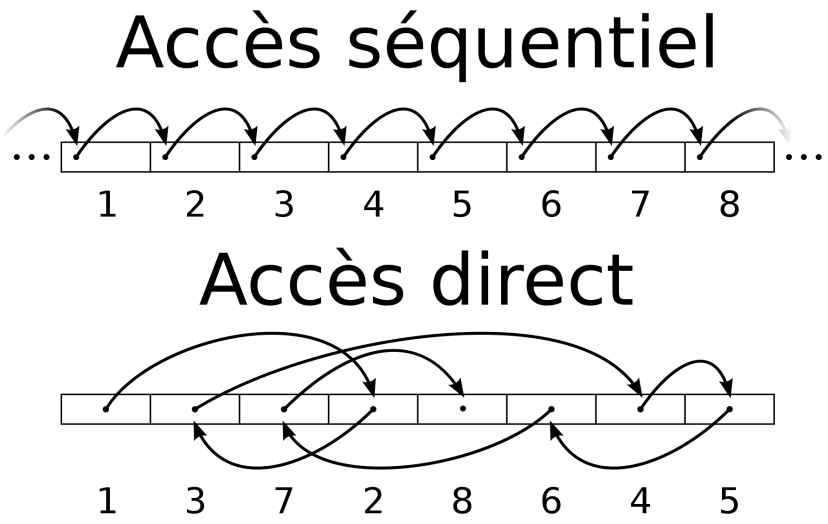
3.1.5) Type de fichier

Pour traiter le fichier correctement, il faut connaitre sa structure interne, et pour cela, il faut connaitre son type.

Le type est identifié par l'extension (sous Windows), par un Uniform Text Identifier (sous Mac OS X), par un "nombre magique" qui est la valeur des premiers octets (sous Unix) et avec des "attributs étendus" sous OS/2.

3.1.6) Méthode d'accès des fichiers

La méthode d'accès aux fichiers dépend du support physique, ainsi une bande magnétique aura un accès **séquentiel** alors qu'une clé USB ou un disque dur aura un accès **aléatoire**.



L'accès séquentiel correspond à un enregistrement en séquence, ainsi pour pouvoir lire le 10e élément, il faut d'abord passer sur les neuf premiers.

L'autre type d'accès est l'accès **aléatoire** ou **direct** qui correspond à des enregistrements de taille fixe, ainsi, on peut calculer directement l'adresse de l'élément auquel on veut aller sans avoir besoin de lire les précédents.

Il est aussi possible d'avoir un **accès séquentiel indexé** qui va créer un index permettant à la fois un accès séquentiel et un accès direct aux fichiers.

3.2) Structure du système de fichiers

Pour organiser le système de fichier, il y a deux structures principales,

La **partition** qui est une partie du disque dur. Ainsi, on peut découper le disque en plusieurs partitions (exemple, partition système, partition de données), ainsi cela peut permettre d'isoler des données du reste ou encore d'avoir plusieurs systèmes d'exploitations sur un même disque (dual-boot).

La deuxième structure est le **répertoire** (ou **dossier**), qui contient les informations sur les fichiers qu'il contient (nom, emplacement, taille, pointeur vers le premier bloc).

Ainsi, la structure interne d'un répertoire doit permettre de localiser, créer, supprimer, renommer, visualiser des fichiers ou encore aller dans un autre répertoire.

3.2.1) Organisation des répertoires

Les répertoires peuvent être à un niveau, deux niveaux ou sous forme d'arbre (c'est ce qui est utilisé aujourd'hui). Le répertoire de départ est appelé le **répertoire racine** (ou **root directory**), ainsi, on va représenter ce répertoire racine par un `/` ou un `\`.

Ainsi, un fichier peut être défini soit par un chemin d'accès **absolu**, c'est le chemin d'accès partant du répertoire racine (exemple `/home/snowcode/image.png`).

On peut aussi définir un fichier par un chemin d'accès **relatif**, c'est un chemin d'accès partant d'un autre répertoire. Voici par exemple un chemin d'accès relatif au dossier `/home` : `snowcode/image.png` ou encore `./snowcode/image.png`.

La manière de supprimer un dossier qui contient des fichiers ou d'autres dossiers dépendent du système d'exploitation. Par exemple, sous Linux, pour supprimer un dossier, il faut faire `rm -r mondossier` le `-r` signifiant qu'il va supprimer de manière récursive.

3.2.2) Fonctionnement des liens

Sous Windows, on utilise des **raccourcis**, les raccourcis sont simplement des fichiers `.lnk` désignant un autre emplacement.

En revanche, sous Unix, on utilise des **liens symboliques**, c'est une entrée particulière (comme un fichier ou un dossier) qui désignent un emplacement différent.

À la différence de Windows, si un programme ouvre un lien symbolique, il va automatiquement ouvrir le fichier (ou le dossier) qui est pointé par ce dernier. C'est donc très intéressant pour faire apparaître un même fichier à plusieurs endroits.

> *Sous Linux, un lien symbolique peut être créé avec la commande `ln -s <chemin de fichier vers lequel pointer> <position du lien>`*

3.2.3) Opération de montage

L'opération de montage permet de rendre accessible un système de fichier. Il peut s'agir d'une autre partition, d'un autre média (DVD, USB, etc), et le format peut être différent de la partition actuelle (NTFS, FAT, ext4, etc).

> *Le format de systèmes de fichiers **FAT**, bien qu'ancien et un peu limité, a l'avantage d'être supporté par tous les systèmes d'exploitations. En revanche, **NTFS** est un système Windows, **ext4** est un système Linux et **APFS** est un format de système macOS.*

Durant cette opération de montage, le système vérifie la cohérence et donne accès aux informations.

Cette opération de montage peut être **implicite** (le système le fait automatiquement) ou **explicite** (l'utilisateur·ice doit lui demander spécifiquement de monter le système).

Ainsi dans Linux, si je connecte une clé USB et que je la monte dans un dossier, je pourrais aller dans le dossier et interagir avec les fichiers comme si de rien était alors qu'en vérité ces fichiers sont sur la clé USB.

3.3) Implémentation du système de fichiers

Maintenant, on va voir comment la structure[19] vue précédemment est implémentée dans le système d'exploitation.

3.3.1) Informations stockées

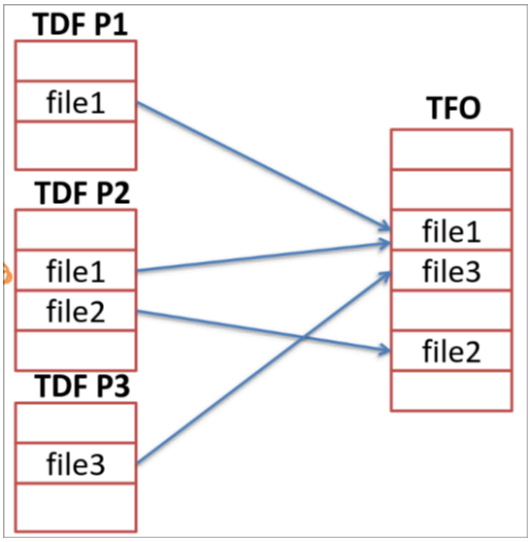
Sur le disque, on va stocker :

- Le **boot control block**, qui sont les informations nécessaires pour démarrer le système d'exploitation
- Le **partition control block**, qui contient les informations détallant les partitions en cours (nombre de blocs, taille, etc), le système le plus utilisé aujourd'hui est GPT, autrefois, c'était MBR.
- Le **root directory** qui est le répertoire principal pour le stockage des dossiers et fichiers
- Le **file control block** qui contient les informations (propriétaire, permissions, etc) sur les fichiers (sous Unix, on va parler d'**i-nodes**)

En mémoire, on va retenir :

- La table des partitions montée et des chemins d'accès (vous pouvez aussi la voir dans `/proc/mounts`)
- Les informations sur les répertoires récemment visités, car s'ils sont récemment visités, ils ont de grandes chances de l'être souvent.
- La **table générale des fichiers ouverts** (TFO) qui décrit tous les fichiers ouverts sur le système. À savoir que si un fichier est ouvert par deux processus différents, il n'apparaîtra qu'une fois dans cette table.
- La **table des fichiers par processus** (TDF, table des descripteurs de fichiers), décrit les fichiers ouverts pour le processus courant en incluant des informations supplémentaires par rapport à la TFO tel que la position courante et le mode d'ouverture.

[19] id:d8da3bd7-f5de-4cfc-a304-a13acbfb936d



Ainsi, lorsqu'un fichier est créé, on va stocker un FCB (file control block ou i-node) pour inclure les permissions, la date, le propriétaire, le groupe, la taille, le premier bloc, etc.

Ensuite le fichier va être enregistré dans la TFO (qui comprend les informations du FCB ainsi que le nombre de processus lié à ce fichier).

Le fichier va aussi être enregistré dans la TDF du processus pour inclure le mode d'accès et le pointeur de position courante.

Lorsque le fichier est fermé, on supprime l'entrée de la TDF du processus et on met à jour la TFO. On ne supprimera ce fichier de la TFO que si plus aucun processus n'utilise le fichier.

3.3.2) Implémentation des répertoires

Maintenant, il faut trouver une manière d'implémenter les répertoires de manière que leur gestion puisse être rapide et qu'il soit facile de retrouver l'information.

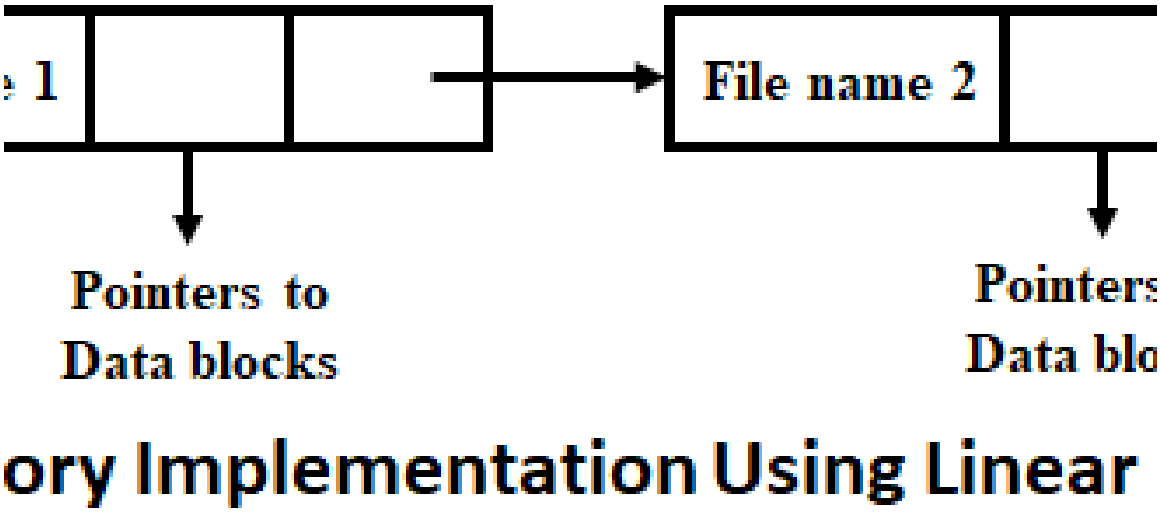
Plusieurs méthodes sont possibles, mais le choix dépendra de si on souhaite favoriser la rapidité ou l'espace disque.

Il faut donc pouvoir gérer le fait qu'un dossier peut contenir un autre dossier aussi bien que des fichiers.

> *Pour en savoir plus sur les deux solutions vu ci-dessous, vous pouvez consulter cet article*[20].

[20] https://www.electronicsmind.com/directory-implementation-in-operating-system/

3.3.2.1) Répertoires sous forme de liste



ory Implementation Using Linear

La première solution est simplement de considérer chaque répertoire comme une liste contenant les noms des fichiers avec un pointeur vers le début de ces fichiers.

Cela implique donc qu'il faudra faire une recherche séquentielle pour trouver un fichier dans un répertoire, de même cela implique qu'il faudra faire cette même recherche séquentielle pour créer le fichier (pour vérifier si le nom existe) ainsi que pour le marqué comme libéré.

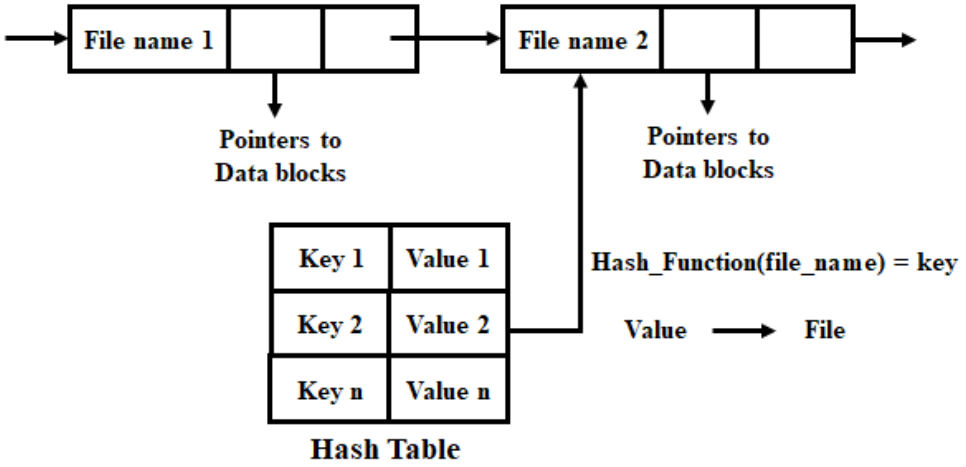
Les inconvénients de cette méthode sont que le parcours séquentiel ralenti considérablement l'utilisation. Si on décide de faire une liste triée, le problème sera toujours là, car il faudra toujours maintenir la liste triée.

3.3.2.1.1) Note sur la libération de l'espace

Marquer l'espace comme libéré consiste à marquer l'entrée du répertoire comme inutilisée.

> *Il est important de se rappeler que lorsqu'on supprime un fichier ou un dossier, ce dernier n'est pas réellement supprimé, l'espace est simplement marqué comme étant libre. Il est donc tout à fait faisable de récupérer ces informations si on lit séquentiellement les espaces libres. Pour supprimer définitivement un fichier, on peut utiliser la commande* `shred`

3.3.2.2) Répertoires sous forme de table hashée



Directory Implementation Using Hash Table

L'idée de cette deuxième solution est d'avoir une liste d'éléments, mais avec en plus une table de hash donnant directement le pointeur du fichier.

Ainsi, lorsque l'on cherche un fichier avec un certain nom, on va hasher le nom du fichier, ce qui va nous donner la clé dans la table. Ensuite, on va pouvoir trouver le pointeur vers le fichier directement en la récupérant depuis la table avec la clé.

Cette méthode a l'avantage d'être très rapide, car il n'y a pas besoin de parcourir quoi que ce soit pour trouver les fichiers. Il est toute fois important de noter qu'il faut trouver des solutions pour traiter les cas où plusieurs noms arriveraient au même hash (**collision**).

Un désavantage de cette méthode est que les tables hashées ont généralement une table fixe et que la performance de la fonction de hash est aussi dépendante de la taille de la table. C'est pourquoi il faut soit limiter la taille du dossier, ou bien étendre la table lorsqu'elle est pleine.

3.3.3) Allocation des fichiers

Maintenant, il va falloir trouver un moyen d'allouer efficacement les fichiers le plus rapidement possible.

On va parler ici de trois méthodes couramment utilisées, la méthode utilisée dépend du système d'exploitation.

3.3.3.1) Allocation contiguë

L'allocation contiguë signifie que chaque fichier occupe des blocs qui se suivent sur le disque. Ainsi, si quand un bloc est en cours de lecture, la lecture du bloc suivant ne requiert pas de mouvement de la tête de lecture, ce qui offre donc une performance intéressante.

Également, si on sait combien de blocs occupe le fichier et que l'on sait la position de son premier bloc, on peut calculer la position du dernier bloc du fichier, ainsi cela fait un accès séquentiel et direct, facile et rapide.

3.3.3.1.1) Problèmes

Le souci avec l'allocation contiguë est que l'on va se retrouver avec des problèmes de fragmentation externe[21] à cause des allocations et libérations successibles. Pour résoudre ce problème, il faut donc utiliser le compactage (la défragmentation). C'est une opération qui peut être dangereuse si elle est interrompue ainsi que chronophage.

Un deuxième problème est de déterminer la taille initiale du fichier, car les fichiers ont tendance à croitre avec le temps. Si un fichier est suivi directement par un autre fichier, que faire si on veut faire croitre le fichier A ?

Si on veut copier le fichier à un autre endroit, cela va rendre le système beaucoup plus lent.

Si on prévoit un "buffer" pour permettre au fichier de croire, alors ça nous mène à de la fragmentation interne **et** externe.

3.3.3.1.1.1) Résolution du problème de la fragmentation sous Linux

Pour régler ces problèmes, cependant Linux (avec ext4) utilise une variante. À la place de placer chaque fichier à la suite des autres. Les fichiers sont éparpillés sur le disque de manière à maximiser les espace entre eux.

De cette manière, on s'assure que les fichiers ont toujours suffisamment de place pour grandir et quand les fichiers sont trop rapprochés, le système d'exploitation va les réarranger pour les espacer à nouveau.

Grâce à cette technique, la défragmentation est presque entièrement inutile sous Linux, car le taux de fragmentation reste toujours très bas.

> *Si vous voulez en savoir plus, vous pouvez consulter cet article Wikipedia sur ext4[22] ou cet article comparant la fragmentation sous Linux et Windows[23].*

3.3.3.2) Allocation chainée

L'allocation chainée consiste à ce que chaque fichier soit une liste de blocs chainés entre eux. Ainsi, le répertoire contient un pointeur vers le premier bloc de la liste et chaque bloc contient un lien vers le bloc suivant.

Cela a l'avantage de ne pas avoir de fragmentation externe et le fichier peut croitre sans problème puis ce que le bloc peut être écrit n'importe où.

3.3.3.2.1) Problème

Le problème de cette méthode est que le contenu des fichiers va être éparpillé partout, la tête de lecture va donc devoir beaucoup voyager et donc ralentir la lecture et l'écriture.

Pour résoudre ce problème, le système FAT alloue des groupes de blocs (cluster) plus tôt que des blocs. Le problème toute fois avec cette méthode, c'est que cela crée de fragmentation interne si les clusters ne sont pas utilisés complètement.

---

[21] id:829e422d-b290-4b7b-b79d-cb046cf56acc

[22] https://fr.wikipedia.org/wiki/Ext4

[23] https://www.howtogeek.com/115229/htg-explains-why-linux-doesnt-need-defragmenting/

3.3.3.3) Allocation indexée

Le principe de l'allocation indexée est de garder un bloc "index" pour chaque fichier. Le bloc va contenir les informations des positions de tous les autres blocs du fichier.

L'avantage principal de l'allocation indexée est qu'elle permet un accès direct aux différents blocs utilisés dans le fichier. Cela permet donc un accès au fichier beaucoup plus rapide si on veut accéder à un point précis.

3.3.3.3.1) Problème

Un problème avec cette méthode est que l'on ne sait pas d'avance la taille nécessaire de l'index. Ainsi, on peut soit lier les index entre eux (via une liste chainée) ou alors utiliser des index d'index (index à plusieurs niveaux).

Un autre problème est que cela empire le problème de tête de lecture de l'allocation chainée, car il faudra, en plus de devoir passer sur tous les blocs éparpillés, passer sur tous les blocs d'index.

Également, pour les très petits fichiers (2 ou 3 blocs), l'allocation indexée garde un bloc complet pour l'index, ce qui est donc beaucoup moins efficace au niveau de l'utilisation du disque.

3.3.3.4) Quelle méthode d'allocation choisir ?

La méthode d'allocation à choisir dépends donc de la façon dont le système sera utilisé, car chaque méthode montre des différences (niveau temps, gaspillage, etc) comme nous l'avons vu juste avant.

Une idée est donc de permettre différents types d'allocations différentes ou de mêler plusieurs méthodes.

Par exemple, certains systèmes utilisent l'allocation contiguë pour les petits fichiers et l'allocation indexée pour les fichiers plus gros ou grandissent.

3.3.4) Gestion de l'espace libre

Nous avons vu précédemment le fonctionnement de la gestion de l'allocation, cependant pour faire cela, il faut prendre un bloc libre et l'utiliser. Il faut donc avant tout trouver un moyen de trouver un bloc libre.

On peut donc utiliser les mêmes méthodes que celles utilisées pour la gestion de mémoire. C'est-à-dire l'utilisation de table de bits[24] ou de liste chaînées[25].

En utilisant une table de bits (ou bitmap en anglais), on va garder une table qui mentionne pour chaque bloc s'il est libre ou occupé. L'avantage est que cette méthode est assez simple et efficace. En revanche, le désavantage est que pour des gros volumes cette table va prendre beaucoup de place. Par conséquent, cette méthode est intéressante pour des volumes faibles, mais devient embêtante pour des volumes plus grands.

L'autre solution est d'utiliser une liste chainée, ainsi chaque bloc libre connait le bloc libre suivant. On peut également faire en sorte de grouper les blocs libre de manière à pouvoir les allouer plus rapidement sans devoir tous les parcourir un à un.

[24] id:8eaf97a6-aec8-421b-9870-66e7b2c5d244

[25] id:e44bb7ec-dbba-40a8-87b1-90a0c5534225

`ext2` (précurseur de ext3 et ext4) utilisait une liste chainée[26] pour stocker les blocs libres. Cependant, cela menait à plus de fragmentation, car les fichiers étaient collés les un aux autres (voir Résolution du problème de la fragmentation sous Linux[27] pour comprendre pourquoi).

Ensuite `ext3` a commencé à utiliser une table de bit[28]. Cependant, ce système faisait les choses bloc par bloc, alors `ext4` a remplacé ce système par un nouveau[29] qui fait plusieurs blocs d'un coup ce qui améliore drastiquement les performances. De plus, `ext4` est optimisé de manière à séparer les allocations pour éviter la fragmentation[30].

3.3.5) Restauration des données

3.3.5.1) Vérification et correction

Le système de fichier peut devenir incohérent et des erreurs peuvent apparaitre à cause d'arrêt inattendu du système ou encore de problèmes matériels. C'est pour ces raisons que le système d'exploitation doit mettre en place des mécanismes pour vérifier la cohérence du système et corriger les erreurs.

La vérification consiste à parcourir les différents blocs des différents fichiers et à résoudre les problèmes qui pourraient survenir, par exemple :

- Un bloc est défini 2 fois comme libre, dans ce cas, il suffit de supprimer l'une des deux occurrences (ou lien si c'est une liste chainée)
- Un bloc est défini comme à la fois libre et à la fois occupé. Alors, on considère qu'il est occupé.
- Un bloc n'est défini ni comme libre, ni comme occupé. Alors, on considère qu'il est libre.
- Un bloc (ou une séquence de blocs) est défini comme occupé par deux fichiers différents. Alors, on duplique ces blocs communs dans les deux fichiers.

3.3.5.2) Sauvegarde et restauration

La **sauvegarde** consiste souvent en la copie de donnée, ailleurs. Elle est prévue pour une restauration rapide.

L'**archivage** consiste à sauvegarder des données plus longtemps, dans quel cas il faut également faire attention au média utilisé pour qu'il soit fiable, éprouvé, robuste et durable.

Il y a deux types de sauvegardes, les sauvegardes **incrémentales** et **différentielle**.

La sauvegarde incrémentale consiste à seulement sauvegarder les changements. Par exemple, la première fois, on fait une sauvegarde complète, la deuxième fois, on fait une sauvegarde de ce qui a changé depuis la première fois, et la troisième fois, on fait une sauvegarde de ce qui a changé depuis la deuxième fois.

---

[26] https://ext4.wiki.kernel.org/index.php/Frequently_Asked_Questions#Why_was_ext2_created.3F

[27] id:e39c37ed-4757-4f51-b5a5-e5255e15fd57

[28] https://ext4.wiki.kernel.org/index.php/Frequently_Asked_Questions#What_is_the_bitmap_allocator.3F

[29] https://ext4.wiki.kernel.org/index.php/Frequently_Asked_Questions#What_is_multiblock_allocation_.28mballoc.29.3F

[30] https://ext4.wiki.kernel.org/index.php/Frequently_Asked_Questions#What_is_delayed_allocation_.28delalloc.29.3F_What_are_its_advantages_in_Ext4.3F

La sauvegarde différentielle consiste à ne copier que les modifications ayant changé depuis la dernière sauvegarde **complète**. Par exemple, la première fois, on fait une sauvegarde complète, la deuxième fois, on fait une sauvegarde de ce qui a changé depuis la première fois, la troisième fois, on fait une sauvegarde de ce qui a changé depuis la première fois (ce qui inclut donc une redondance de ce qui était dans la deuxième sauvegarde), etc.

La taille de la sauvegarde différentielle va donc beaucoup plus augmenter que celle de la sauvegarde incrémentale, jusqu'à la prochaine sauvegarde complète.

La sauvegarde incrémentale est donc plus légère et plus rapide, mais sera plus complexe à restaurer (car s'il y a eu 17 sauvegarde, il faudra restaurer les 17 éléments) tandis qu'avec la sauvegarde différentielle, il suffira de restaurer la dernière sauvegarde complète et la dernière sauvegarde différentielle.

3.3.5.3) Système de fichiers journalisé

Le système de fichier est quelque chose de complexe et sa manipulation nécessite beaucoup d'opérations. Un problème peut arriver n'importe quand et si l'opération en cours ne peut pas se terminer, alors cela peut mener à un état incohérent dont la correction pourrait engendrer une perte de donnée.

Le **système de fichier journalisé** permet de limiter les dégâts en gardant un historique des opérations en cours, de cette manière le système peut *défaire* les opérations non terminées en cas de panne.

C'est un système qui est assez similaire à celui des transactions en base de données, il faut donc garantir l'atomicité (le fait qu'une action ne puisse pas être décomposée), la cohérence, l'isolation et la durabilité (ACID) pour chaque action du journal.

4) Entrées sorties

4.1) Introduction

Un système sans entrée ou sortie n'est pas très utile. Car s'il n'y a pas d'entrée-sortie, ça veut dire, pas de réseau, pas d'écran, pas de clavier, etc.

Il est donc très important pour le système d'exploitation de contrôler les périphériques. Par exemple, le SE doit pouvoir gérer les interruptions (évènements envoyés par les périphériques) ainsi que traiter les erreurs ou autres évènements qui pourraient survenir (exemple, on débranche la clé USB pendant une action de lecture).

Le SE doit aussi fournir une interface vers ces périphériques de la manière la plus uniforme possible pour rendre les périphériques faciles d'accès et universel.

4.2) Coté matériel

Au niveau matériel, on distingue plusieurs éléments :

- Le **périphérique d'entrée-sortie**, par exemple clavier, souris, écran, etc. On pourra le caractériser par sa nature ainsi que sa vitesse de transmission. Ces périphériques vont faire de la **signalisation** (ou **interruption**) pour signaler au système que sa tâche est terminée.
- Le **contrôleur** qui est l'interface entre le périphérique et le système d'exploitation. Le contrôleur permet au SE de contrôler le périphérique.

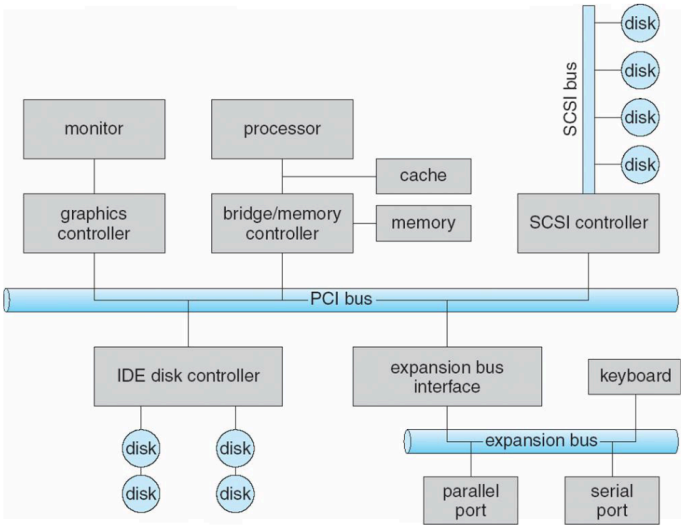
4.2.1) Périphériques E/S

Il existe plusieurs types de périphériques d'E/S, ici, on va parler de deux types principaux.

Il y a les périphériques **blocs** où l'information est stockée dans des blocs de taille fixes et adressables. Un exemple de tels périphériques est un disque dur.

Il y a aussi les périphériques **caractères** où l'information est un flux de caractère sans tenir compte d'une structure quelconque, elle est donc non adressable. Par exemple, la carte réseau, clavier, souris, etc.

4.2.2) Controlleur de périphérique



Le contrôleur est la partie électronique qui contrôle le matériel. Souvent, le périphérique dispose d'une partie mécanique et d'une partie électronique (exemple, un disque dur). Cependant, ce n'est pas toujours le cas, par exemple la carte graphique n'est pas attachée à l'écran.

Certaines cartes d'extensions sont même dissociées de la carte mère, c'est par exemple le cas avec les périphériques USB.

4.2.2.1) Contrôle/commandes

Le système d'exploitation **commande** le périphérique au moyen du contrôleur, il n'interagit jamais directement avec le périphérique lui-même.

Le contrôleur dispose fréquemment d'un microprocesseur hautement adapté au périphérique, de registres qui permettent de commander le contrôleur et de mémoire pour faire les échanges (par exemple pour faire des buffers).

4.2.2.2) Interruptions

Le matériel envoie des interruptions pour signaler que l'opération en cours est terminée. Le gestionnaire d'interruption du système d'exploitation va gérer l'interruption.

S'il n'y a qu'une seule interruption, le système gère l'interruption et choisit ensuite le processus suivant à démarrer.

S'il y a plusieurs interruptions, le système traite d'abord les interruptions urgentes en ignorant temporairement les autres.

4.2.2.2.1) Fonctionnement

Le CPU sauvegarde le contexte du processus en cours et lance une routine (une fonction indépendante du reste du système) adaptée à l'interruption reçue. La routine à lancer est renseignée dans le **vecteur des interruptions** où pour chaque interruption un pointeur vers la routine à exécuter est renseignée.

Dès que l'interruption est traitée, l'état du contrôleur du périphérique est modifié. Le gestionnaire des interruptions est alors prêt à traiter les interruptions suivantes.

Une fois toutes les interruptions traitées, le système d'exploitation choisit le processus suivant à lancer.

4.2.2.3) Dialogue via port d'E/S

Ce type de dialogue entre le système et le contrôleur consiste à échanger des données en écrivant ou lisant des données dans les registres du contrôleur.

Cette méthode n'est plus trop utilisée aujourd'hui.

4.2.2.4) Dialogue via E/S mappé en mémoire

Dans cette méthode, on va faire correspondre une zone mémoire (de la mémoire centrale) aux registres du contrôleur. Ainsi, lorsque l'on lit dans la zone mémoire, on lit et/ou écrit dans les registres du contrôleur.

L'avantage de cette méthode est que l'on a juste à utiliser des instructions habituelles d'accès à la mémoire pour contrôler les périphériques.

Grâce à cette méthode, il n'y a donc pas besoin d'instructions assembleur particulières et il n'y a pas non plus besoin de mettre en place des mécanismes de protection particuliers, car il suffit d'interdire ou d'autoriser l'accès à la zone mémoire.

4.2.2.5) Amélioration des performances via DMA

Si on utilise simplement le contrôleur d'un disque dur, lorsque l'on veut lire sur ce dernier, le contrôleur va lire un bloc à partir du périphérique, et le placer dans sa mémoire, ensuite, il va vérifier l'information, puis faire une interruption au système d'exploitation. Enfin, ce dernier va exécuter la routine adaptée et copier l'information dans sa mémoire centrale.

Le problème avec ce système est que le CPU est fort sollicité pour faire le transfert et ne peut donc rien faire d'autre. Une solution à cela est d'utiliser un contrôleur **DMA** (Direct Memory

Access[31]). C'est un microprocesseur spécialisé dans le transfert d'informations entre un contrôleur quelconque et la mémoire centrale.

Ce contrôleur DMA peut aussi bien se trouver sur certains périphériques que sur la carte mère.

Le contrôleur DMA possède, comme tout contrôleur, plusieurs registres. Les siens sont, l'adresse (pour indiquer où les informations doivent être écrites en mémoire), le compteur d'octets (pour indiquer quelle quantité doit y être écrite), le registre de contrôle (qui spécifie de quel périphérique il s'agit, si c'est une lecture ou une écriture et de l'unité du transfert (octet, mot, etc)).

Ainsi, en utilisant un contrôleur DMA, lorsque l'on veut lire sur un disque dur, le système d'exploitation va indiquer au DMA les données dont il a besoin, puis va demander au disque une lecture. Ensuite le contrôleur disque va lire et vérifier les blocs et c'est le DMA qui va se charger de transférer directement les informations dans la mémoire. Une fois le transfert terminé, le DMA fait une interruption au système d'exploitation.

De cette manière, le CPU peut faire d'autres choses durant le transfert

4.2.2.5.1) Vol de cycle

Attention toutefois, lorsque le DMA travaille, il se peut qu'il entre en conflit avec le processus, car les mémoires ne peuvent faire qu'un accès par cycle.

Le DMA ne pouvant pas attendre aussi longtemps que le processeur, parce qu'il risque de perdre des informations, il a donc priorité d'accès à la mémoire. On dit alors qu'il y a un **vol de cycle** du processeur.

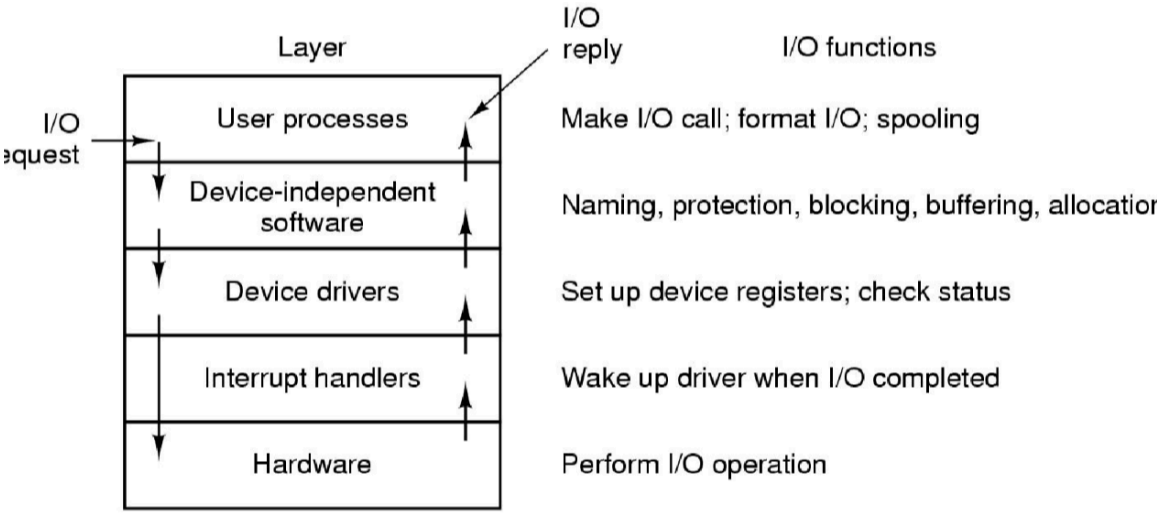
4.3) Coté logiciel

La partie logicielle de l'entrée-sortie est une partie du système d'exploitation qui a pour but de fournir une interface vers les périphériques tout en assurant une indépendance par rapport au type de périphérique.

Par exemple, l'accès à une clé USB doit être, du point de vue des programmes, identique à l'accès sur un disque dur.

Cette partie du système d'exploitation est divisée en quatre couches, que l'on va adresser en partant du plus proche du matériel vers le plus haut niveau.

[31] https://fr.wikipedia.org/wiki/Acc%C3%A8s_direct_%C3%A0_la_m%C3%A9moire#R%C3%A9solution_des_conflits



4.3.1) Le gestionnaire d'interruption

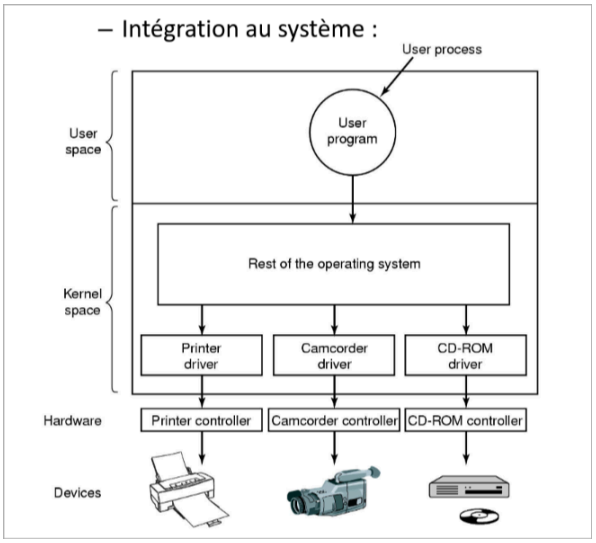
Le gestionnaire de périphérique (le pilote) qui lance l'opération d'entrée-sortie, va attendre une interruption.

Lorsque l'interruption survient, **gestionnaire d'interruptions** va avertir le gestionnaire de périphérique et celui-ci va terminer l'opération voulue.

Lorsqu'une interruption survient, le gestionnaire d'interruptions va sauvegarder le contexte du processus en cours, créer le contexte pour le traitement de l'interruption, se place en état "disponible" pour traiter les interruptions suivantes, exécuter la procédure de traitement (la routine dont on a parlé dans la section précédente).

Une fois les interruptions gérées, le scheduler choisit le processus à démarrer.

4.3.2) Gestionnaire de périphérique (pilote ou driver)



Le **gestionnaire de périphérique**, aussi appelé **pilote** ou **driver**, dépends de la nature du périphérique, ainsi un gestionnaire de disque fonctionne très différemment d'un gestionnaire de souris.

Le gestionnaire de périphérique, afin d'être utilisable, doit être chargé au cœur du système, dans le noyau, en mode protégé. C'est pour cela qu'un mauvais driver pour conduire à des plantages du système d'exploitation.

Mais en même temps, ces gestionnaires de périphériques, écrits par les fabricants, doivent pouvoir être inséré facilement dans le noyau (kernel) pour être utilisé.

4.3.2.1) Fonctionnement

Le système appelle des fonctions internes (open, read, write, initialize, etc) des pilotes afin de commander le périphérique. Les pilotes vont ensuite vérifier si les paramètres reçus sont corrects et valide et vont traduire certains paramètres en données physiques.

Par exemple, un pilote de disque dur va traduire un numéro de bloc en cylindre, piste, secteur et tête.

Ensuite, le pilote vérifie si le périphérique est disponible (sinon il attend), il vérifie également l'état du périphérique et le commande.

Pendant le travail du périphérique, le pilote s'endort, car il ne peut plus rien faire. Une fois qu'il est réveillé par le gestionnaire d'interruption, il vérifie que tout s'est bien passé et transmet les informations.

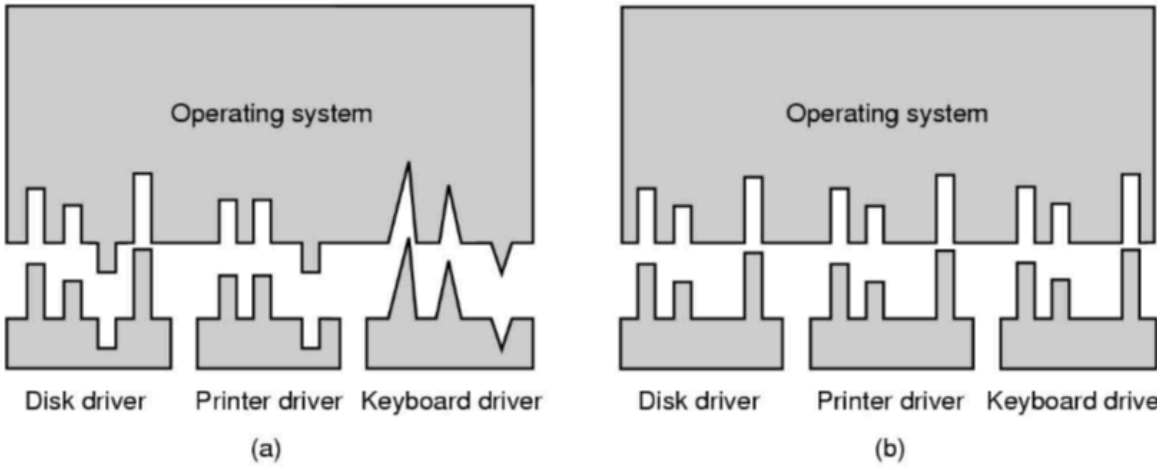
4.3.3) La couche logicielle indépendante

La couche de **logicielle indépendante** (aussi appelée **interface standard**)

Cette couche fournit une interface uniforme pour les drivers (la fameuse API dont on a parlé plus tôt), ainsi que du buffering, de la gestion d'erreur, de l'allocation et libération des périphériques.

4.3.3.1) Interface standard

Grâce à cette couche, on s'assure que les drivers sont appelés de manière uniforme.



A gauche, un système sans interface standard, à droite un système avec une interface standard

4.3.3.2) Uniformisation des noms

La désignation d'un périphérique doit être non ambiguë, sous Linux le répertoire `/dev` contient des entrées pour chaque périphérique. Un périphérique est donc traité au même titre qu'un fichier, il ne peut donc pas en avoir deux avec le même nom.

Par exemple, un disque dur sera appelé `/dev/sda`, si on met un autre disque dur celui-ci sera appelé `/dev/sdb`.

4.3.3.3) Buffering

Afin d'améliorer les performances du système et d'éviter de faire beaucoup d'accès inutilement, on garde des buffers (tampons) dans le système d'exploitation et dans les contrôleurs. L'objectif étant de placer en mémoire les informations qui sont souvent accédées.

4.3.3.4) Gestion d'erreurs

C'est aussi à cette couche que revient la gestion des erreurs. L'utilisation de périphériques induit un certain nombre d'erreurs (temporaire ou permanentes).

Il y a deux types d'erreurs, les **erreurs de programmation** qui surviennent lorsque l'opération demandée est impossible (exemple, écriture sur un périphérique d'entrée), dans quel cas on rapporte l'erreur et on s'arrête.

Et les **erreurs d'entrées sorties** qui surviennent lorsqu'une opération se termine de façon anormale (par exemple, on déconnecte le disque, ou le périphérique est défectueux). Dans ce cas, le driver décide de soit tenter à nouveau l'opération ou alors de rapporter l'erreur.

4.3.3.5) Allocations et libérations des périphériques

Certains périphériques ne sont pas partageables, par exemple avec une imprimante, il est impossible d'imprimer plusieurs choses en même temps.

Il revient alors au système d'exploitation de vérifier si le périphérique est libre et s'il peut être alloué au processus.

La fonction `open` permet à un processus d'informer le système d'exploitation qu'il souhaite utiliser un périphérique, le système vérifie alors sa disponibilité.

4.3.4) Couche d'entrée-sortie applicative

Cette partie de la gestion de l'entrée sortie est faite au niveau de l'application, par exemple via les libraires systèmes liées aux programmes.

C'est là que l'on va par exemple trouver les appels systèmes de `stdio` tel qu'open, printf, read, write, etc.

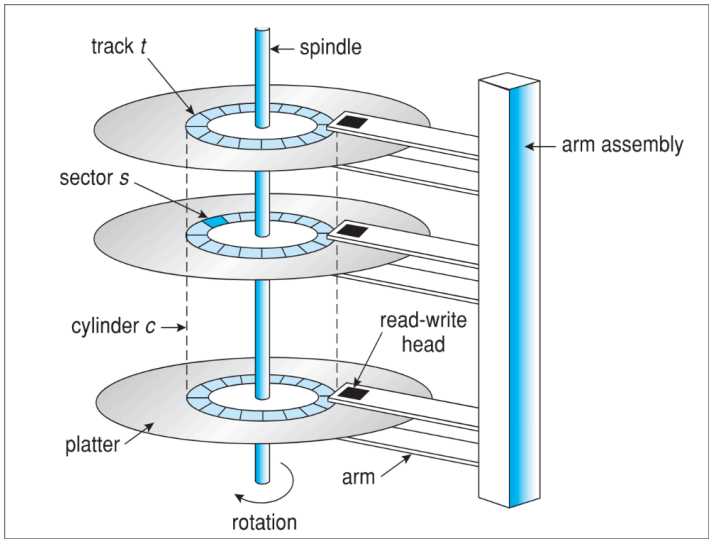
Cette couche va également fournir un système de **spooling** qui permet de créer une fille d'attente pour l'accès aux périphériques non partageables. Par exemple via une liste de jobs d'impression pour une imprimante.

4.4) Disque

4.4.1) Matériel

Un disque magnétique est composé de plusieurs disques physiques appelés les **plateaux**.

Le disque est découpé en cylindres, pistes et secteurs. Il y a autant de pistes que de positions différentes de la tête de lecture.



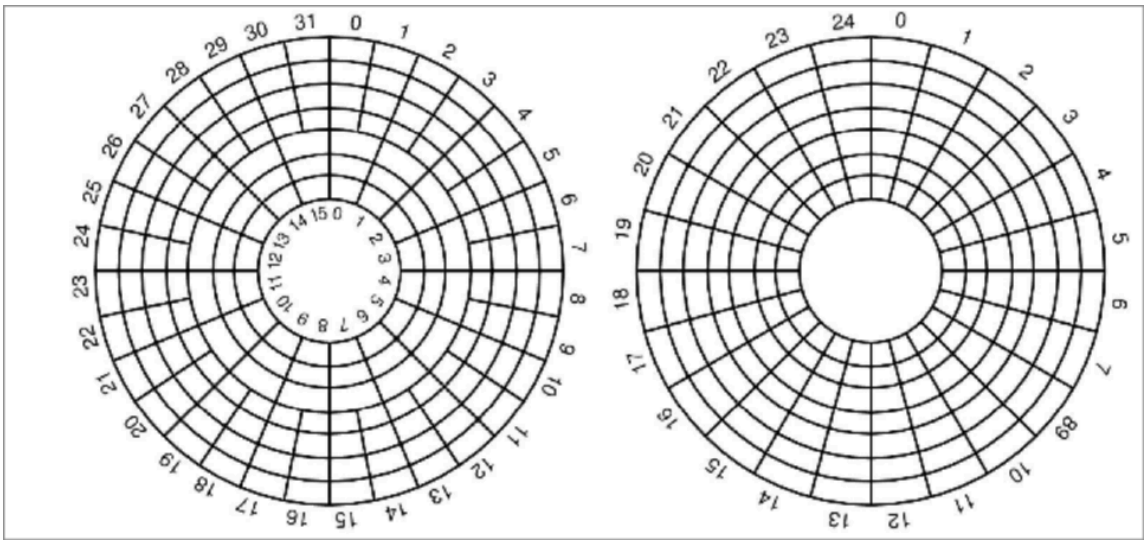
Un **cylindre**, c'est l'ensemble de pistes à une position donnée de la tête de lecture.

Une **piste** correspond à un cercle sur un plateau à une certaine position de la tête de lecture. Chaque piste est elle-même découpée en secteur.

Les **secteurs** sont des blocs de tailles fixes qui compose chaque piste.

Pour en savoir plus sur le fonctionnement des disques dur magnétiques, vous pouvez consulter cette page Wikibooks[32].

Les disques dur ont une interface qui permet au contrôleur d'utiliser des commandes de haut niveau. La géométrie du disque dur n'est pas nécessairement celle qui est annoncée, car le nombre de secteurs par piste n'est pas constant.



A gauche, véritable géométrie du disque. À droite, géométrie du disque simplifie pour rendre les accès plus simple

4.4.2) RAID

RAID est une technologie qui consiste à combiner plusieurs disques afin d'améliorer les performances (si on a plusieurs disques, on peut par exemple écrire sur plusieurs disques en même temps) et/ou la fiabilité (si on a plusieurs disques, on peut par exemple dupliquer les informations sur tous les disques, comme ça si un disque tombe en panne, on peut restaurer les données).

Le **mirroring** consiste à dupliquer toutes les informations sur un second. Ainsi, lors d'un accès en écriture, le contrôleur effectue la modification sur les deux disques simultanément et indique qu'il a terminé lorsque l'opération est achevée sur les deux disques.

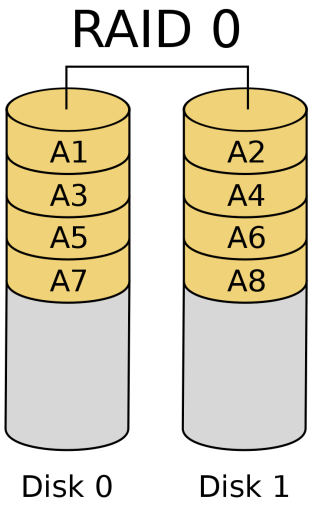
Le **stripping** consiste à répartir l'information sur plusieurs disques. De cette manière, si on a huit disques, on peut distribuer les informations sur chaque disque de façon à écrire huit fois plus rapidement qu'avec un seul.

Il existe plusieurs niveaux de RAID, certains utilisent du mirroring, d'autre du stripping et d'autres les deux de manière à favoriser les performances et la fiabilité.

Si vous souhaitez en savoir plus sur les niveaux de RAID, vous pouvez consulter cet article Wikipédia[33] et celui-ci[34].

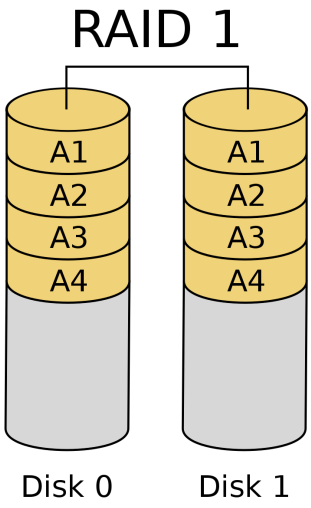
[32]https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/Les_disques_durs
[33]https://en.wikipedia.org/wiki/Standard_RAID_levels

4.4.2.1) RAID 0



Le RAID 0 fait uniquement du stripping. Il va donc construire un grand espace disque en combinant les disques. Ainsi, les données sont réparties entre des différents disques de manière à améliorer les performances.

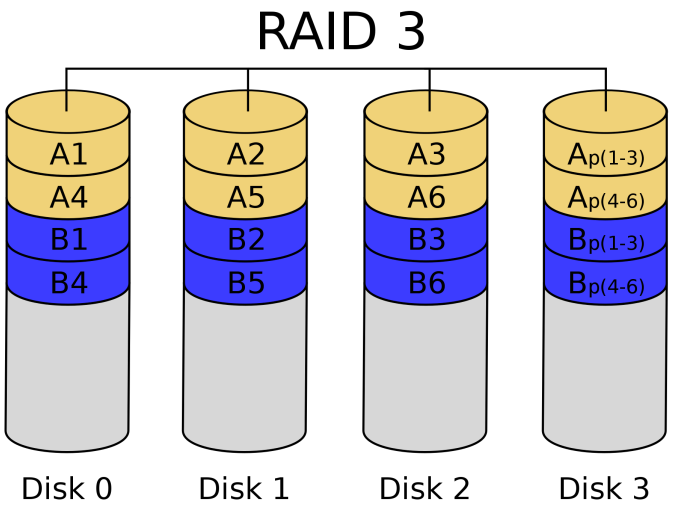
4.4.2.2) RAID 1



Le RAID 1 fait exclusivement du mirroring, il faut donc doubler le nombre de disques. Cela améliore beaucoup la fiabilité de l'information, mais il est assez couteux.

[34]https://en.wikipedia.org/wiki/Nested_RAID_levels

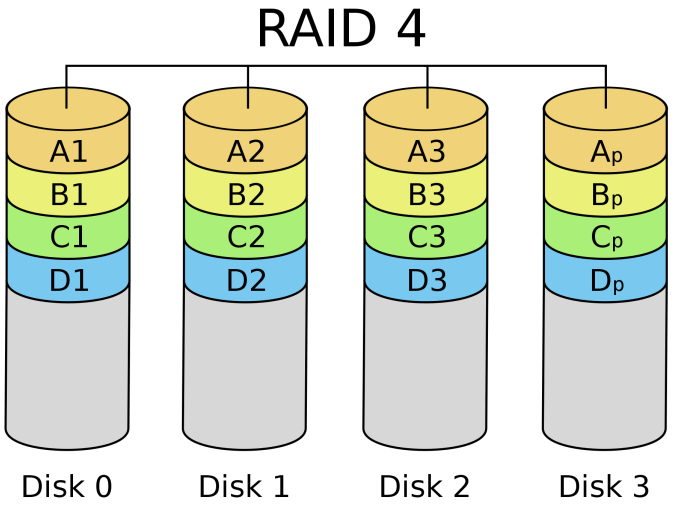
4.4.2.3) RAID 3



Le RAID 3 fait du stripping au niveau des octets. Un bit de parité est gardé sur un disque séparé et les octets vont être réparti sur les différents disques.

L'avantage du RAID 3 est que si on perd un disque, on peut reconstruire l'information à la volée en utilisant le disque de parité.

4.4.2.4) RAID 4

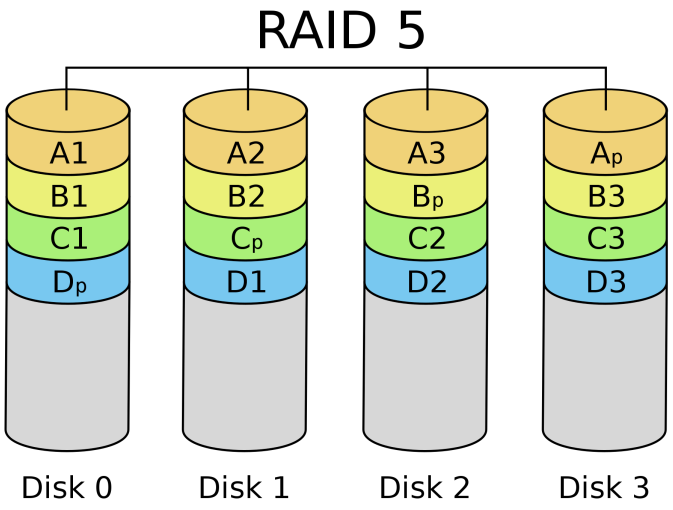


Le RAID 4 fait du stripping au niveau des blocs. Les blocs de parités sont gardés sur un disque séparé.

Un avantage du RAID 4 est que la lecture d'un bloc ne nécessite l'accès qu'à un seul disque (contrairement au RAID 3), on peut donc également satisfaire la lecture de plusieurs blocs simultanément si ceux-ci ne sont pas localisés sur le même disque.

Il y a cependant un problème à l'écriture, étant donné que tous les bits de parité sont sur le disque de parité, on ne peut pas écrire des blocs en parallèle, car on ne peut avoir qu'un seul accès au disque de parité à la fois.

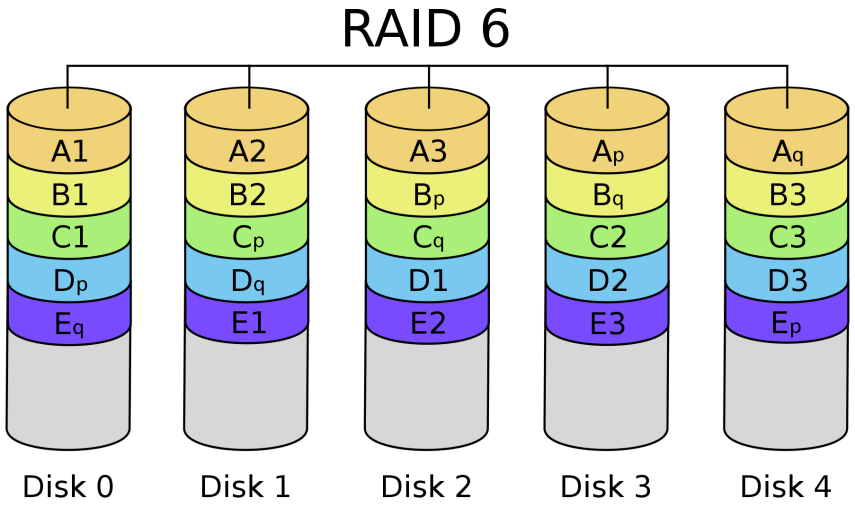
4.4.2.5) RAID 5



Le RAID 5 est une amélioration du RAID 4 qui va distribuer les informations sur la parité, de cette manière chaque disque contient des blocs de données et de parité. À chaque bloc de donnée correspond un bloc de parité stocké sur un disque.

Lors d'une écriture, il y a alors moins de problème, car plusieurs requêtes d'écriture peuvent être faites en même temps parce que tout dépend de la position de la donnée à écrire et de la localisation de l'information de parité.

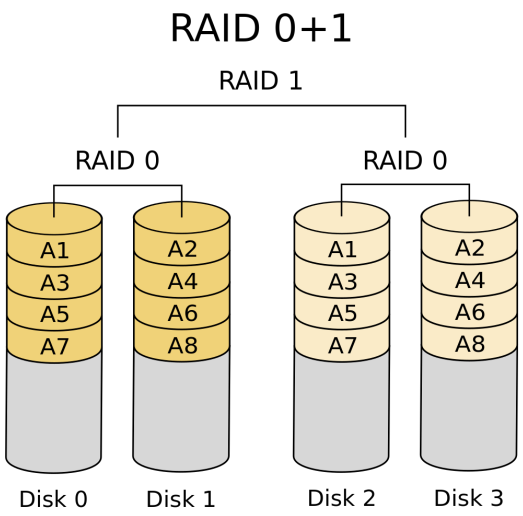
4.4.2.6) RAID 6



Le RAID 6 permet de protéger de la perte de deux disques dur, elle utilise un algorithme de parité plus complexe et nécessite un CPU plus important pour le contrôleur.

Plus il y a de données, plus le RAID 6 est préférable au RAID 5.

4.4.2.7) RAID 0+1



Ce niveau utilise RAID 0 pour le stripping et RAID 1 pour le mirroring de tous les disques en stripping. Il est très utile si l'efficacité et la protection sont importantes, cependant il est assez cher.

4.4.2.8) Lequel choisir ?

Généralement, c'est le RAID 5 qui est choisi s'il y a plusieurs disques ou alors le RAID 1 s'il n'y a que deux disques. Sauf dans des cas où il y a des demandes plus particulières au niveau de la performance et/ou de la fiabilité du système.

4.4.2.9) Options particulières

Il y a des options particulières sur les différents systèmes, notamment la possibilité de faire du **hot-swap** ou d'avoir des disques **spare**.

Le hot-swap consiste à pouvoir retirer des disques pendant que ceux-ci sont connectés (du moment qu'il n'est pas actuellement utilisé).

Le spare consiste à avoir un disque présent inutilisé, mais prêt à l'emploi. Ainsi, si un disque dur tombe en panne, le disque spare peut alors être utilisé.

4.4.2.10) Autres considérations

Il est important de bien considérer que le système ait un débit important pour avoir les meilleures performances.

Mais surtout, il est important d'avoir une certaine diversité entre les disques dur. Il ne faut donc pas prendre tous les disques dur de la même marque au même moment, de la même génération.

Car alors, il y a de grandes chances que les disques se fatiguent de la même manièrent et tombe en panne plus ou moins en même temps.

4.4.3) Formatage

Avant qu'un disque ne puisse être utilisé, il doit être formaté (**formatage de bas niveau**), cette opération consiste à écrire la géométrie du disque (secteur, cylindre, piste, etc).

Cette opération doit être réalisée par le fabricant. De cette manière, chaque secteur contient un préambule (informations décrivant le secteur tel que le numéro ou le cylindre), les données et un code ECC pour la gestion des erreurs.

Une fois l'espace créé, il est possible de mettre en place des partitions, c'est la première structure logique du disque.

Le disque est séparé en plusieurs partitions, et chaque partition est vue par le système comme un espace séparé. Par exemple, un système Linux va voir les partitions `/dev/sda1` et `/dev/sda2` comme deux espaces complètements différents.

Le secteur zéro du disque contient le partiion control block et le boot control block[35] qui mentionnent comment le disque et découpé et comment le système peut démarrer.

Le **formatage de bas-niveau** est une opération de formatage réalisée par le système d'exploitation, elle consiste à donner une structure à la partition (un format de système de fichier[36]).

4.4.4) Scheduling

Le disk arm scheduling est une opération faite par le système d'exploitation pour planifier les requêtes d'entrées-sorties.

En effet, plusieurs requêtes d'E/S peuvent arriver depuis plusieurs processus pendant que le disque est toujours en train de traiter une requête, donc les autres requêtes doivent attendre.

L'importance du scheduling (avec les HDD) est de minimiser le **seek-time**, c'est-à-dire minimiser les mouvements de la tête de lecture afin de pouvoir lire les informations plus vite.

Nous allons donc, voire plusieurs algorithmes de scheduling différents,

- Le **FCFS** (First Come First Served), soit les requêtes sont servies dans l'ordre dans lesquelles elles arrivent. Cet algorithme a l'avantage d'être équitable, mais a de mauvaises performances, car il ne fait rien pour améliorer le temps de réponse du disque.
- Le **SSTF** (Shortest Seek-Time First), autrement dit de servir toujours la requête la plus proche de la position courante. Cet algorithme a l'avantage de faire diminuer le temps de réponse du disque. Mais il a le désavantage de devoir calculer les positions et risque également de causer de la famine parmi les requêtes les plus éloignées.
- Le **SCAN** (ou algorithme de l'ascenseur), où le bras de lecture va toujours dans la même direction jusqu'à arriver à la fin du disque avant de retourner dans le sens inverse, aussi, il n'y a plus de famine, car quoi qu'il arrive, on avance.

[35] id:f7486013-ee04-4ec2-aca1-a75f59674ebd

[36] id:c0c8c166-63a0-428e-8338-e5b754011606

- Le **C-SCAN** (SCAN circulaire), fonctionne comme le SCAN sauf qu'au lieu de partir dans l'autre sens lorsque la fin du disque est atteinte, elle retourne à l'autre extrémité du disque.
- Le **LOOK** est une variante du SCAN qui a la place de continuer jusqu'aux extrémités du disque, va s'arrêter lorsqu'il n'y a plus de requête dans la direction.
- Le **C-LOOK** (LOOK circuliare), comme le LOOK, excepté qu'à la place d'aller dans la direction inverse, il va directement à la première requête.

> *Vous pouvez découvrir d'autres algorithmes ou avoir plus d'explication et d'illustrations sur les algorithmes décrit ici en allant lire cette page*[37]*.*

4.4.4.1) Choix de l'algorithme

Le SSTF est courant et fournit un bon remplacement à FCFS.

Si le disque est très chargé, C-SCAN et C-LOOK sont intéressants.

Le plus souvent on va trouver du SSTF ou une variante de LOOK.

Aujourd'hui, le scheduling est réalisé à plusieurs niveaux, par exemple au niveau du système d'exploitation, mais également au niveau des contrôleurs.

4.4.5) Gestion des erreurs

Il peut y avoir beaucoup d'erreurs différentes sur un disque.

Par exemple, des secteurs défectueux peuvent survenir lors de la construction du disque dur, notamment lors du formatage de bas niveau, ou même survenir durant l'utilisation du disque.

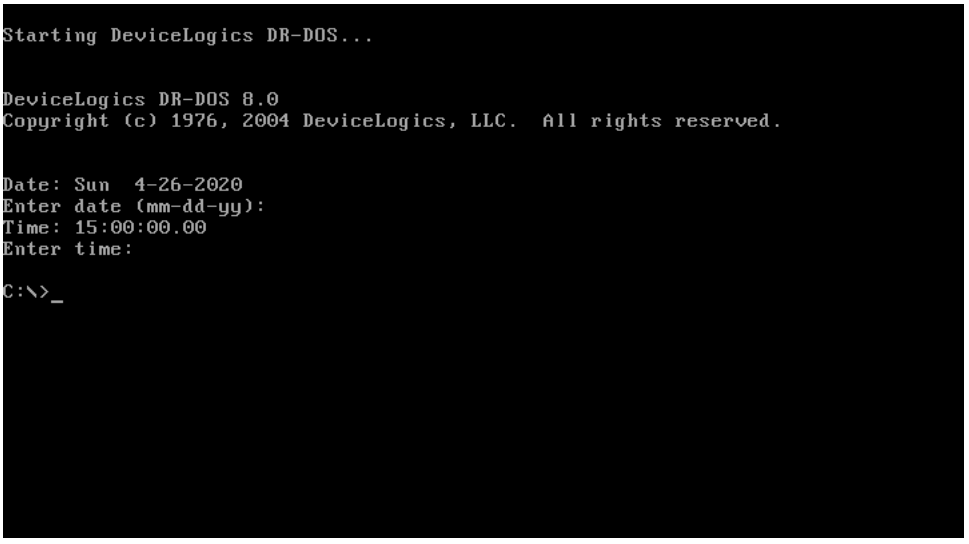
C'est pourquoi le système d'exploitation doit pouvoir se souvenir des blocs défectueux afin de ne plus les utiliser dans le futur.

4.5) Horloge

L'horloge est un périphérique spécial et essentiel. Il sert simplement à déterminer la date et l'heure.

C'est une fonction essentielle, car c'est nécessaire pour maintenir des statistiques, calculer le quantum de temps des processus, etc.

[37] https://www.geeksforgeeks.org/difference-between-scan-and-look-disk-scheduling-algorithms/



Fun fact : la Raspberry Pi n'a pas d'horloge, la date et l'heure est synchronisés via le réseau[38] au démarrage. Et s'il n'y a pas de connexion, l'horloge va commencer à compter à partir de l'heure à laquelle il s'est arrêté.

Les systèmes MS-DOS n'avaient pas non plus d'horloge, il fallait donc indiquer la date et l'heure manuellement au démarrage.

L'horloge fonctionne avec un quartz, un compteur et un registre. Le quartz, lorsqu'il est sous tension, génère un signal périodique très précis, ce signal peut ensuite être utilisé pour la synchronisation des composants. Le signal est ensuite fourni au compteur qui va décompter jusqu'à arriver à zéro.

Une fois arrivé à zéro, une interruption est émise vers le CPU et le gestionnaire d'horloge prend la main.

Ensuite, ici, il y a deux modes de fonctionnement. Il y a le mode **one-shot**, le système envoie l'interruption et attends une réaction. Sinon, il y a le mode **square wave** qui, une fois que le compteur arrive à 0, il recommence. On parle ici de **ticks d'horloge**.

Le système d'exploitation va ensuite maintenir l'heure en comptant le nombre de secondes depuis une date et heure de référence (par exemple, sous Linux/UNIX, on compte le nombre de secondes depuis le 1ier janvier 1970 00:00 UTC).

Pour synchroniser l'heure, il suffit donc de savoir le nombre de secondes depuis un point de référence.

Le logiciel d'horloge a ensuite pour but de maintenir l'heure, vérifier qu'un processus ne dépasse pas son quantum, compter l'utilisation du CPU, gérer l'appel système alarm, maintenir différentes statistiques, etc.

[38] https://www.youtube.com/watch?v=WX5E8x3pYqg

Par exemple, pour vérifier qu'un processus ne dépasse pas son quantum, le compteur d'horloge est initialisé par le scheduler en fonction du quantum. À chaque interruption, il est diminué et une fois arrivé à zéro le processus est remplacé.

Pour pouvoir gérer plusieurs évènements temporels en même temps, le système met en place une **file d'évènements**, qui agit comme une sorte d'agenda pour planifier des évènements.

5) Sécurité

5.1) Protection, domaine et matrice d'accès

Lorsque l'on parle de **protection**, on parle de l'ensemble des mécanismes mis en place pour l'accès, la gestion des ressources systèmes par les processus, etc. Cette protection doit être fournie autant par le système que par les applications.

La **sécurité** en revanche concerne un spectre plus large incluant les virus, les attaques et la cryptographie.

La protection a pour but de prévenir les violations d'accès et d'améliorer la fiabilité (en détectant des erreurs humaines par exemple).

Si une ressource n'est pas protégée, elle peut être mal utilisée par des utilisateur·ice·s autorisé·e·s (ou non).

Une **politique** de protection correspond à la définition de ce qui doit être protégé. Tandis qu'un **mécanisme** de protection indique comment protéger.

5.1.1) Zones de protections

Sur l'ordinateur, il faut pouvoir protéger les processus et les **objets**.

Les objets peuvent être autant matériel (CPU, mémoire, imprimante, disque, etc) que logiciel (fichiers, programmes, sémaphore, etc).

Pour ce qui est des processus, il faut s'assurer que chaque processus ne peut accéder qu'aux ressources auxquelles il a l'autorisation. Il ne doit accéder qu'aux ressources qui sont nécessaires pour terminer sa tâche.

5.1.2) Domaine de protection

Un **domaine** définit un ensemble d'objet et de type d'opération qui peut être exécutée sur les objets (la possibilité d'exécuter une action sur un objet s'appelle un **droit d'accès**).

Un domaine est donc finalement une collection de droits d'accès.

Chaque processus opère à l'intérieur d'un domaine de protection. Les mêmes objets peuvent être référencés dans plusieurs domaines.

Un domaine peut être créé de plusieurs manières (par utilisateur, processus, procédure, etc). Et des opérations pour changer de domaine sont prévues.

L'association entre un processus et un domaine peut être **statique** (ne change pas, les droits d'accès restent donc tout le temps les mêmes), ou **dynamique** (les droits d'accès peuvent changer).

5.1.3) Matrice d'accès

La matrice d'accès est une représentation des domaines et de leurs droits d'accès.

C'est un tableau à deux dimensions où chaque ligne représente un domaine (aussi appelé rôle ou sujet), et chaque colonne représente un objet (aussi appelé asset).

Voici un exemple de matrice d'accès :

Sujets		Objets: Fichier 1	Segment 1	Segment 2	Processus 2	Editeur
Processus	1	Lire	Executer	Lire Ecrire		Entrer
Processus	2	Lire Ecrire				Entrer
Processus	3		Lire Ecrire Executer		Entrer	Entrer

La matrice d'accès permet de vérifier les politiques de protection voulues. Il faut donc définir les politiques pour chaque objet, assigner chaque domaine à l'exécution d'un processus.

Voyons maintenant quelques droits spécifiques,

5.1.3.1) Droit au changement de domaine

object / main	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	sw cor
D_3		read	execute					
D_4	write		write		switch			

Pour représenter la permission de changer vers un autre domaine, il suffit de considérer les domaines aussi comme des objets. Ensuite pour permettre à D1 de pouvoir avoir les droits accès de D2, il suffit de mettre **switch** dans l'intersection de D1 et D2.

5.1.3.2) Droit à la copie de son droit

Le droit **copy** permet à un domaine de copier son droit pour un objet donné à un autre domaine.

Ce droit est représenté par une $*$.

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

Par exemple, ici, D2 peut copier le droit lecture de F2 sur un autre domaine que le sien.

Il existe aussi une variante du droit copy qui est le droit **transfert** (ou copie limitée). Si dans l'exemple précédent, il s'agit d'un droit de transfert, alors lorsque D2 passe son droit de lecture de F2 à un autre domaine, il perd le droit de lecture.

5.1.3.3) Droit à la modification des droits d'un objet

Le droit **owner** permet à un domaine de modifier n'importe quel droit pour un certain objet.

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

Dans cet exemple, D1 possède un accès total à F1. De cette manière, D1 peut par exemple s'accorder un droit d'écriture sur F1 ou encore accordé un droit de lecture de F1 à D2.

5.1.3.4) Droit au contrôle des droits d'un domaine

Le droit **control** permet à un domaine de supprimer des droits d'accès à un autre domaine.

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D
D_1	read		read			switch		
D_2				print			switch	swi cor
D_3		read	execute					
D_4	write		write		switch			

Dans cet exemple, D2 peut par exemple supprimer le droit d'accès en écriture (F1) de D4.

5.1.4) Sous UNIX (setuid)

Sous UNIX, chaque domaine correspond à un utilisateur·ice, et lorsque qu'un exécutable est lancé, le processus prend le domaine de l'utilisateur·ice qui l'a lancé.

Sauf lorsque le bit `setuid` est mis. Ce bit est représenté par un `s` lorsque l'on regarde les permissions d'un fichier. Lorsque c'est le cas, l'exécutable va se lancer en tant que l'utilisateur·ice qui possède le fichier, à la place de s'exécuter en tant qu'utilisateur·ice qui l'a lancé.

Par exemple, avec le fichier sudo, lorsque l'on fait un `ls -l` dessus, on peut voir les permissions à gauche.

```
-r-s--x--x 1 root root 63432 Jan  6 09:22 /run/wrappers/bin/sudo
```

On peut voir qu'il y a un `s`, cela signifie que lorsque j'exécute ce fichier en tant qu'utilisateur `snowcode`, le fichier est réellement exécuté en tant que `root`.

5.1.5) Implémentation

Utiliser une matrice d'accès dans le système ne serait pas très pratique, car prendrait beaucoup de place et ne pourrait pas être mise en mémoire centrale.

C'est pourquoi, on utilise des **access list** à la place. L'access list est une liste associée à chaque objet. Elle contient des pairs de domaines et de droits. Un objet dont le domaine n'a pas de droit n'est pas présent sur la liste.

Il est également possible d'étendre la liste avec des droits par défaut (dans quel cas, pour vérifier les droits d'une opération, on vérifie d'abord les droits par défaut avant de rechercher la paire correspondante au domaine).

5.1.5.1) Révocation des droits

Ensuite, chaque système doit aussi prévoir comment révoquer des droits. Par exemple pour définir si c'est immédiat ou décalé, pour tous les utilisateurs ou seulement certains, si cela est temporaire ou permanent, etc.

Chaque système définit les stratégies possibles et laisse le choix à l'utilisateur·ice.

5.2) Sécurité

Nous avons vu comment protéger le système et déterminer les droits d'accès. Maintenant, il faut trouver des mécanismes pour protéger le système contre le vol d'information, la modification/destruction non autorisée d'information et surveiller l'utilisation du système.

Les mécanismes de sécurité ont pour but d'empêcher ou de ralentir le plus possible toute violation du système, il faut que le cout pour pénétrer un système soit plus important que le gain que l'on peut en retirer.

5.2.1) Niveaux de protections

Il y a 4 niveaux de protections différents :

1. Physique, protéger le matériel dans des endroits protégés
2. Humains, surveillance pour l'accès au matériel
3. Réseau, protection adéquate, firewalls, etc
4. Système d'exploitation

5.2.2) Dangers d'une attaque

La sécurité est évidemment primordiale, les entreprises stockent toutes leurs informations sur des systèmes informatiques et les informations personnelles valent de l'or. Des informations perdues ou volées peuvent conduire une entreprise à la faillite.

5.2.3) Identification des utilisateur·ice·s

Les systèmes actuels ne fonctionnent que lorsque l'utilisateur·ice est authentifié·e.

Nous allons parler ici de plusieurs types d'authentification différents et de quelques bonnes pratiques pour chacun d'entre eux.

5.2.3.1) Identification par mot de passe

Le mot de passe est une méthode courante d'authentification, cependant elle n'est pas forcément très sûre, notamment, car elle dépend beaucoup de l'utilisateur·ice.

5.2.3.1.1) Comment un mot de passe peut être découvert

Un mot de passe peut être volé de plusieurs manières, soit en **connaissant l'utilisateur** et en devinant le mot de passe (c'est pourquoi il faut que les mots de passes soient aléatoires), ou de façon **brutale** (par dictionnaire ou énumération).

Il est aussi possible de voler un mot de passe en utilisant un **keylogger**, c'est-à-dire un programme ou appareil qui va enregistrer toutes les entrées clavier de l'utilisateur·ice.

Un mot de passe peut aussi être **sniffé** en écoutant tout ce qui se transmet sur le réseau, si le mot de passe n'est pas transféré de manière chiffrée, alors on peut récupérer le mot de passe.

Un mot de passe peut encore être découvert en analysant des bases de données d'anciens mots de passe découvert, car mal stocké par d'autres entreprises. En effet, comme beaucoup d'utilisateur·ice·s ne changent pas de mot de passe, il y a de grandes chances que ce dernier n'ait pas changé.

Enfin, le mot de passe peut encore être découvert à cause d'autres maladresses de l'utilisateur·ice·s, tel que tomber dans une attaque de **fishing** ou encore l'écrire sur un post-it sur son bureau.

5.2.3.1.2) Stockage d'un mot de passe

> *Si vous voulez en savoir plus sur les dangers et les différentes manières de stocker des mots de passes, regardez cette vidéo[39].*

5.2.3.1.2.1) Plain text

La pire manière de stocker des mots de passes est de juste les stocker en base de donnée. Car lorsque l'on fait cela, ça signifie que toute personne ayant accès à la base de donnée a accès à tous les utilisateurs et mots de passes.

Pire encore, puis ce que les utilisateur·ice·s réutilisent souvent les mêmes mots de passes, pour beaucoup d'entre eux, cela donne accès à l'entièreté de leur vie en ligne.

5.2.3.1.2.2) Chiffré

Une autre manière est de chiffrer les mots de passes. Cela signifie que si quelqu'un a accès à la base de donnée, il ne pourra rien en faire. Cependant, cela signifie aussi que si quelqu'un a accès à la clé de déchiffrement, cette personne a toujours accès à tous les mots de passes. C'est donc également une chose à éviter.

[39] https://www.youtube.com/watch?v=8ZtInClXe1Q&t=23

Un autre problème avec cette méthode est que si certains mots de passes sont identiques, on pourra le reconnaitre, car on verra le même mot de passe chiffré dans la base de donnée plusieurs fois.

5.2.3.1.2.3) Hashing

Cette manière consiste à hasher (faire passer à travers une fonction de hashage) les mots de passes.

Une fonction de hashage est une fonction à sens unique, ainsi, on fait passer le mot de passe à l'intérieur, cela génère un texte, mais il est impossible de retrouver le mot de passe à partir du texte.

Ainsi, il suffit de stocker le hash dans la base de donnée, ainsi lorsque l'utilisateur·ice veut se reconnecter, on hash le mot de psase entré et on compare le hash dans la base de donnée avec celui qui vient d'être généré.

Le problème avec cette méthode est que plusieurs utilisateur·ice·s ayant le même mot de passe vont avoir le même hash. Par conséquent, on le saura directement dans la base de donnée.

Il y a notamment une attaque en particulier qui utilise cette faiblesse, c'est la **rainbow table attack**, à la place d'avoir une liste de mots de passes courants, les hasher et les essayer un par un contre chaque hash (ce qui est une attaque par **dictionaire**), on va avoir une liste de mots de passes pré-hashé et simplement comparer les hash. Ce qui fait que l'attaque est beaucoup plus rapide, surtout que beaucoup d'algorithmes de hashage sont volontairement lents afin de décourager les attaquants.

5.2.3.1.2.4) Hashing avec salt

Cette méthode est l'une des meilleures méthodes aujourd'hui. Elle consiste à utiliser un **salt** avec le hash pour se protéger contre les **rainbow table attacks**.

Un **salt** est une chaine de caractère aléatoire différente pour chaque utilisateur qui va être ajouté à chaque mot de passe. Le salt est ensuite présent juste à côté du hash en clair.

Donc, lorsqu'un·e utilisateur·ice se connecte, on va aller chercher le salt, l'ajouter à son mot de passe et le hasher. Ensuite, on compare ce hash avec celui dans la base de donnée.

Grâce à cette chaine aléatoire (le salt), les rainbow table attacks sont inutiles, car elles ne peuvent plus comparer les hash.

Cette méthode est d'autant plus puissante lorsqu'elle fonctionne avec des algorithmes de hashage lent, parce que l'attaquant n'a pas d'autre choix que de tester les mots de passe un par un et va perdre beaucoup de temps.

5.2.3.1.2.5) Ne pas stocker de mot de passe

Stocker des mots de passes est une grande responsabilité, il est possible que ce soit plus sûr de ne tout simplement pas les stocker et d'utiliser des mécanismes tel que OAuth pour à la place demander à des tiers de confiance de le gérer pour nous. Par exemple par Google ou Microsoft.

Ou encore d'utiliser des appareils physiques tels qu'une carte électronique ou un appareil FIDO2 (on va en reparler plus tard) à la place ou en plus du mot de passe.

5.2.3.2) Identification à plusieurs facteurs

Une première manière de faire de l'authentification à plusieurs facteurs est d'utiliser un système de code à usage unique tel que le TOTP. À savoir que certains de ces mécanismes peuvent aussi être utilisés seul, pas seulement en conjonction avec un mot de passe (par exemple une clé FIDO2 pour authentification SSH), on va en reparler dans l'identification password-less.

L'avantage d'utiliser l'identification à plusieurs facteurs est de palier la faiblesse du système de mot de passe en demandant à un autre système de générer un code ou de résoudre un problème (par exemple, envois de SMS, TOTP, hash chain, clé FIDO2, etc).

5.2.3.2.1) Hash Chain

Une autre manière de faire est d'utiliser à répétition la fonction de hashage.

Par exemple, on génère une valeur aléatoire de départ (le seed).

Ensuite, on exécute la fonction de hashage un certain nombre de fois (par exemple 1000 fois) sur ce seed, ce qui donne donc un hash de hash de hash de hash de … du seed. Le serveur fait de même et sauvegarde ce hash.

Lors de la première authentification, l'utilisateur·ice génère un code à usage unique en dérivant une fois de mois qu'avant le seed. Dans cet exemple, l'utilisateur va donc hasher le seed 999 fois. Le serveur peut alors vérifier que lorsqu'il hash le seed, il obtient bien le hash de départ, l'authentification est donc réussie. Le serveur défini alors ce nouveau hash (de 999 fois) à la place de l'ancien (de 1000 fois).

Ainsi le processus se répète jusqu'à arriver à zéro où dans quel cas il faut générer un nouveau seed.

5.2.3.2.2) TOTP (Time-based One Time Password)

> *Cette section est en bonus et n'est pas présente dans le cours*

En somme, l'utilisateur·ice a un appareil (smartphone, digipass ou autre), qui partage un secret avec le serveur (token) ainsi que le temps (le temps doit être le même que sur le serveur).

Ainsi, après que l'utilisateur·ice a entré son mot de passe, on lui demande le code à usage unique. L'utilisateur·ice peut ensuite demander à l'appareil de générer ce code.

Ce code est généré en hashant le secret partagé (token) et le temps actuel. Ainsi, le mot de passe n'est valable que pour une certaine durée de temps.

En somme, en plus d'ajouter son mot de passe, l'utilisateur·ice va également

5.2.3.2.3) FIDO2

> *Cette section est en bonus et n'est pas présente dans le cours. Vous pouvez en apprendre plus sur FIDO2 en regardant cette vidéo[40].*

Lorsque l'on veut s'enregistrer sur un site internet, la clé FIDO2 va créer une paire de clé (clé privée et clé publique, on va y revenir plus tard).

[40] https://www.youtube.com/watch?v=ze2i9V1_aIc&t=234

Une fois la paire de clé générée, la clé publique est envoyée au site avec un "handle", cet handle contient des informations uniques sur le site qui a demandé l'authentification.

Ainsi, lorsque l'on souhaite se connecter, le site va envoyer un message aléatoire à signer à la clé, la clé va ensuite vérifier que le handle correspond bien (que c'est le bon site, donc protection contre le fishing, le bon compte, sur la bonne clé).

Si tout correspond, la clé va signer le challenge envoyé par le site. Le site pourra ensuite vérifier que la signature est correcte avec la clé publique.

Ce mécanisme est donc très simple (uniquement un bouton) et très sécurisé (protection contre le fishing, aucune information personnelle à stocker).

5.2.3.3) Identification password-less

L'identification password-less repose sur l'idée que les mots de passes sont compliqués à gérer, assez faible et que l'authentification à plusieurs facteurs peut être compliquée.

L'identification password-less repose donc sur d'autres principes tels que la biométrie (empreinte digitale, reconnaissance faciale, etc), sur un code unique (par SMS ou application tierce, voir TOTP[41] plus tôt), par lien magique (envois d'un lien de connexion par mail), par notification push, ou par clé FIDO2[42] par exemple.

Si vous voulez en savoir plus sur l'authentification password-less, vous pouvez regarder cette vidéo[43].

5.2.3.3.1) Identification biométrique

L'identification biométrique a pour but d'authentifier sur base d'une propriété de l'utilisateur·ice (sa tête, ses doigts, ou autre).

Le scanner converti et numérise la propriété, la donnée numérisée est alors traitée comme un mot de passe et est hashée et salée.

Le problème de cette méthode est que la protection de ces données devient vraiment très critique, contrairement aux mots de passes, on ne peut pas changer sa tête ou ses empreintes digitales.

De plus, ces informations sont aussi des informations très confidentielles qui peuvent aussi avoir un impact politique important[44].

5.2.4) Sécurité des applications

Écrire du code exempt d'erreur est difficile, et les erreurs peuvent conduire à des vulnérabilités qui permettent d'attaquer le programme.

L'attaque peut permettre d'obtenir des droits non accordés au départ, faire planter l'application, introduire des données incorrectes, etc.

5.2.4.1) Attaques courantes

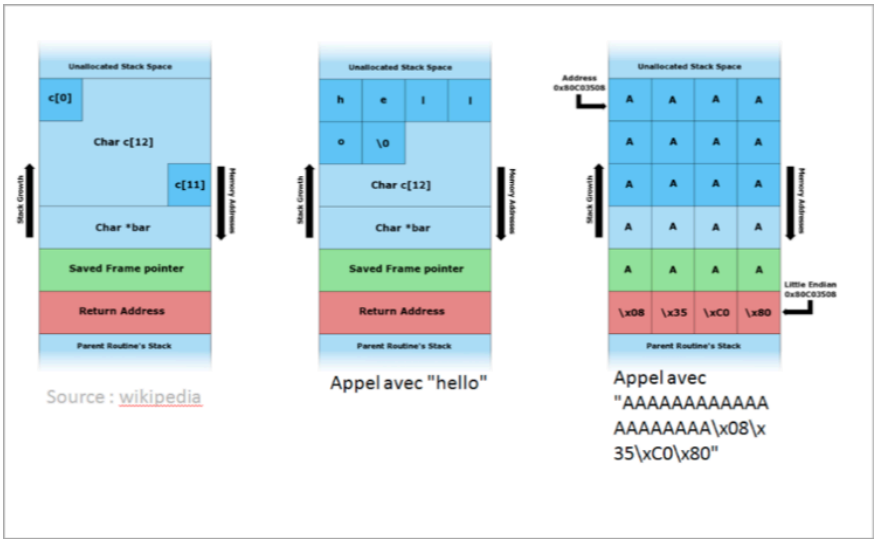
[41] id:e34a2b20-9658-44e0-a934-64c8e5ea56fc

[42] id:cd67b68e-d47e-403e-8e8c-0c5e38edb92b

[43] https://www.youtube.com/watch?v=ze2i9V1_aIc&t=234

[44] https://www.laquadrature.net/2019/06/21/le-vrai-visage-de-la-reconnaissance-faciale/

- **Remote Code Execution** (RCE), exécution de code à distance en soumettant une donnée précise à l'application
  - Un exemple qui a fait beaucoup de bruit est celui de la vulnérabilité log4shell[45] dans le système de log Java "log4j" qui faisait qu'il était possible d'exécuter du code sur toute application utilisant la librairie. Cette vulnérabilité était si dangereuse qu'elle fut considérée par certains gouvernements comme l'un des problèmes de sécurité informatique la plus sérieuse des 20 dernières années.
- **SQL Injection**, injection de code SQL dans la base de donnée en soumettant des données précises.
- **Format String vulnerabilities** qui consiste à soumettre du code qui est compris comme une commande par l'application. Pour en savoir plus, des exemples de code en C sont donnés dans cet article[46].
- **Cross-Site Scripting** (XSS), qui est encore dû à une non-vérification des soumissions de l'utilisateur·ice qui peut mener à intégrer du code HTML dans une page. Ce qui signifie que l'on peut aussi injecter du code JavaScript dans la page qui seront exécutés par tous les visiteurs de celle-ci.
- **Username enumeration**, si le système mentionne que le nom d'utilisateur est introuvable, il est possible de tester plusieurs noms d'utilisateurs pour savoir lesquels sont présents sur le système. Il est parfois aussi possible de faire cela via l'API du site (genre en regardant `/api/user/01` et les énumérer comme ça).
- **Stack/buffer overflow**, l'attaquant soumet une chaine de caractère spécifique qui provoque un débordement de la pile. Grâce à ce débordement, le code exécuté devient celui de l'attaquant, cela peut donc lui faire prendre le contrôle du code de l'application, et si l'application s'exécute dans le domaine administrateur, lui faire devenir administrateur (**privilege escalation**).



[45] https://en.wikipedia.org/wiki/Log4Shell
[46] https://owasp.org/www-community/attacks/Format_string_attack

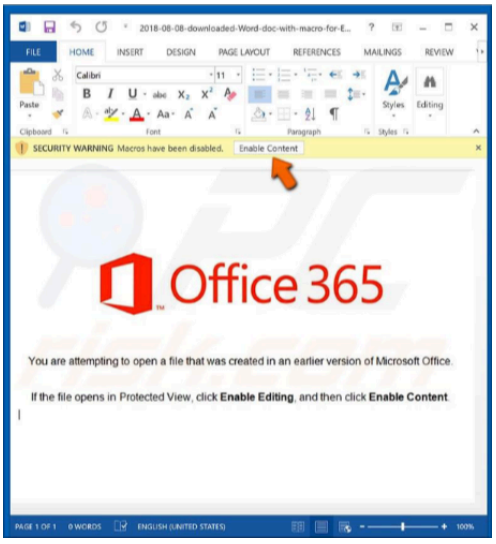
- **Déni de service** (DoS), le fait de rendre inaccessible un système en le bombardant de requêtes. Le système devient ainsi inaccessible durant le temps de l'attaque, bien qu'aucun dommage ne soit opéré, cela cause un problème en terme l'image pour l'entreprise et peut lui faire perdre de l'argent à cause de l'inaccessibilité du site.
  - Si une attaque DoS est fait depuis plein d'ordinateurs différents (ce qui fait qu'elle est plus compliquée à arrêter), on parle d'attaque de **déni de service distribuée** (DDoS). Chaque ordinateur est appelé un **zombie** et l'ensemble des ordinateurs infectés utilisé est appelé un **botnet**.

5.2.4.2) Programmes malveillants

Maintenant, on va parler de types de programmes malicieux,

- **Cheval de troie** (ou **trojan**), est un programme qui se fait passer pour ce qu'il n'est pas pour déclencher une action hostile. Cela peut par exemple être un faux anti-virus.
- **Backdoor** (porte dérobée), le programme installe une porte d'accès à l'utilisateur, il est ainsi possible de faire exécuter des commandes à la machine à distance.
  - C'est une méthode assez utilisée pour constituer des attaques DDoS, car la machine devient un zombie qui fait partie d'un botnet auquel l'attaquant demande de faire des requêtes à répétition sur un site.
  - Cette méthode peut aussi être utilisée pour installer un cryptomineur qui va miner des cryptomonnaies pour l'attaquant sur toutes les machines victimes
- Les **vers informatiques** (ou **WORMS**) sont des programmes qui se propagent par les réseaux informatiques. Les vers utilisent des vulnérabilités d'autres systèmes pour se propager, ainsi chaque machine infectée infecte les autres machines du réseau à son tour.
  - Un exemple très connu de WORM est celui de WannaCry[47], un ransomware qui a infecté plus de 300000 ordinateurs sur plus de 150 pays différents et qui aurait causé des pertes de l'ordre de plusieurs centaines de millions de dollars. Ce WORM se propageait via une vulnérabilité dans un port de communication du système Windows.
- Les **virus informatiques** sont des ensembles d'instructions qui utilisent un programme pour se reproduire. Ainsi, comme un virus biologique, un virus informatique a pour principal but de se reproduire en infectant un programme hôte. Certains virus ont également une charge active qui attaque le programme à un moment déterminé. Certains virus utilisent des techniques avancées pour se cacher des systèmes de protection.
  - Les **virus script** sont des virus écrit dans un langage interprété (par exemple le VBA, langage de programmation de Microsoft Office), le virus utilise un environnement tiers pour s'exécuter et se reproduire. Cela peut par exemple être sous la forme d'une macro d'un document Microsoft Office.

[47] https://en.wikipedia.org/wiki/WannaCry_ransomware_attack



5.2.5) Protection contre les attaques

Pour se protéger contre des attaques, il est important de protéger le **périmètre** (tout ce qui est vers l'extérieur, tel que le réseau), en installant un firewall au niveau du réseau.

Ensuite, il est aussi important de protéger les **machines** en installant un firewall personnel et un anti-virus.

D'autres choses sont importantes telles que ne jamais travailler avec un compte administrateur.

5.2.5.1) Sécurité d'un parc informatique

Pour assurer la sécurité d'un parc informatique, on peut utiliser des **scanners de vulnérabilités**, ce sont des programmes qui vont regarder si le système est vulnérable à certaines attaques afin de pouvoir mettre à jour les composants qui en ont besoin.

Il faut aussi pouvoir assurer l'**intégrité des programmes**, on peut donc garder de manière sécurisée les empreintes de tous les programmes et vérifier celles-ci lors de leur lancement. Le seul souci, c'est qu'il faut tenir compte des mises à jour.

On peut aussi utiliser un **système de détection d'intrusion** (IDS), c'est un programme qui tente de repérer des activités anormales sur un système en se basant sur des plans d'attaque connus, en observant l'activité et les journaux systèmes.

Par exemple, `fail2ban` est un IDS qui bannit les adresses IP qui réalisent un certain nombre de tentatives de connexions illicites.

Un IDS peut aussi analyser le comportement de l'utilisateur pour détecter des comportements anormaux (exemple un secrétaire qui compile un logiciel). Il faut cependant faire attention à configurer l'IDS correctement pour éviter les faux positifs. Un autre exemple d'IDS est `Snort`

Enfin, un denier élément important est l'analyse de fichiers journaux. Tous les systèmes d'exploitation consignent des informations sur ce qu'il se passe sur le système, on peut donc utiliser

des outils pour analyser automatiquement ces fichiers journaux tel que Splunk, Grafana Loki ou encore Crowdsec.

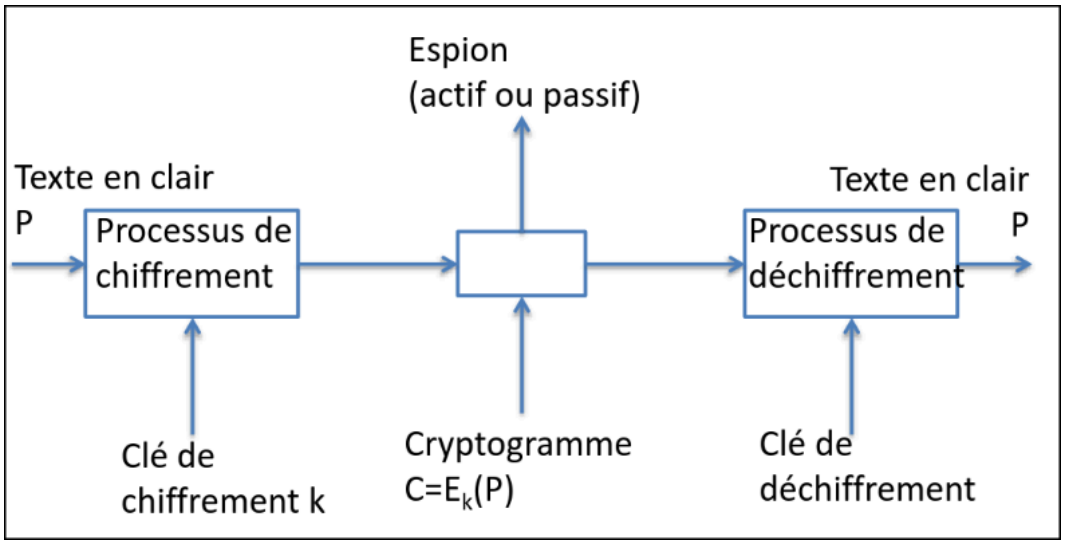
5.3) Cryptographie

5.3.1) Introduction à la cryptographie

La **cryptographie** consiste à cacher des informations en utilisant des algorithmes. Il ne faut pas le confondre avec la **stéganographie** qui consiste à cacher des messages dans d'autres messages.

Par exemple, si on prend un message tel que `BONJOUR` (**texte en clair**), et que pour chaque lettre, on la décale d'un certain nombre de positions dans l'alphabet (**processus de chiffrement**), ici, on va choisir 13 positions, (13 sera ici la **clé de chiffrement**), on obtient `OBAWBHE` qui est notre **cryptogramme** (message chiffré).

Pour déchiffrer le message, il suffit alors de faire le processus inverse, ce processus inverse est donc le **processus de déchiffrement** qui retournera `BONJOUR` (notre texte en clair).



5.3.1.1) Histoire de la cryptographie

La cryptographie est utilisée depuis très longtemps par les militaires, diplomates et amants.

Avant l'avènement de l'informatique, la cryptographie était limitée aux capacités du cerveau humain, il fallait pouvoir changer de méthode de chiffrement si quelqu'un se faisait prendre.

Avec l'avènement de l'informatique est venu la capacité de calculer des résultats beaucoup plus complexe.

5.3.1.1.1) Substitution et transposition

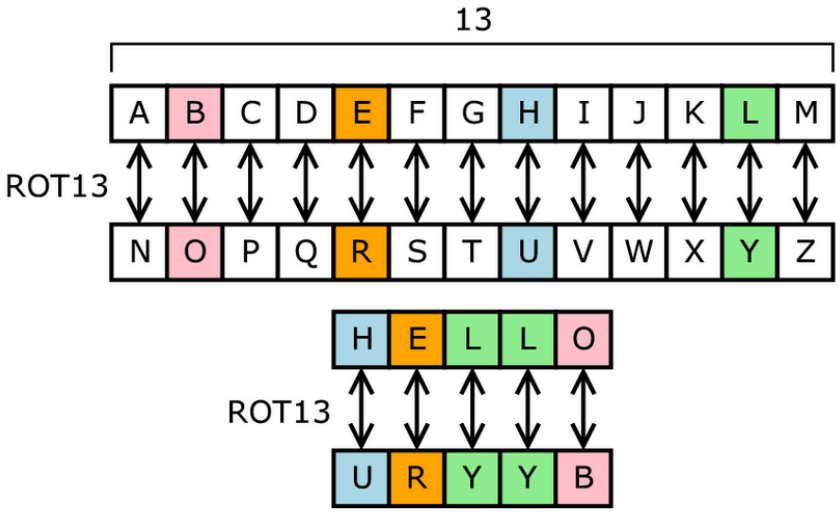
La substitution et la transposition sont deux méthodes historiques de sécurisation des données. Ces méthodes ne devraient plus être utilisées aujourd'hui pour protéger des données.

> ***En sécurité, il n'y a rien de pire que d'avoir l'illusion qu'on est protégé***

5.3.1.1.1.1) Substitution

La **substitution** est un mécanisme par lequel chaque caractère du texte clair est remplacé par un autre caractère dans le texte chiffré. Cela signifie qu'on a une table de correspondance de chiffres, sons, mots ou autre à quelque chose. C'est par exemple le cas du code de césar dont nous avons parlé juste avant.

Voici, par exemple, la table de substitution du ROT13, qui est un code de César auquel la clé est à 13.

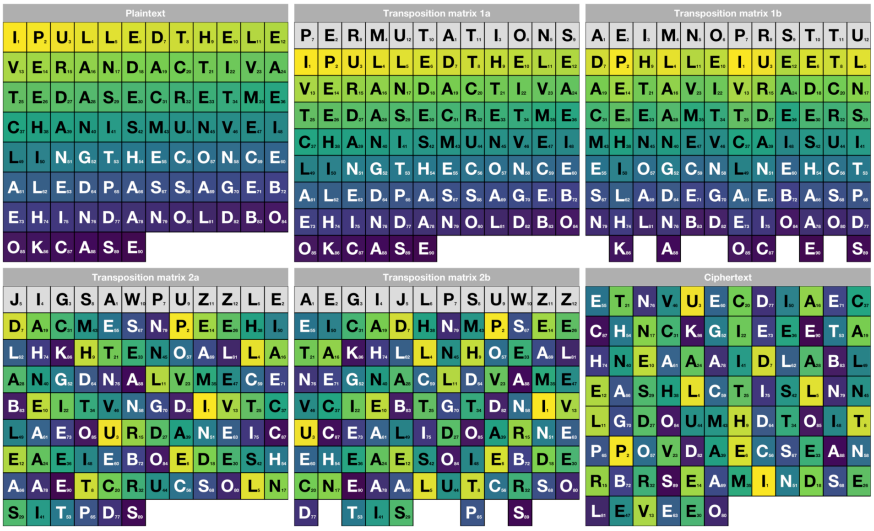


Le problème avec ce système est que l'on peut utiliser la linguistique et un contexte pour supposer la présence de certains mots ou la fréquence de certaines lettres.

Par exemple, si on sait que le texte est rédigé en français, on peut compter la lettre qui revient le plus souvent, supposer que c'est un E et compter la différence entre la lettre du cryptogramme et la lettre E pour obtenir la clé.

5.3.1.1.1.2) Transposition

La **transposition** consiste à mélanger les lettres d'une certaine manière.



Ici par exemple, on commence par mettre le texte dans une grille, puis on ajoute un premier mot clé au-dessus et on trie les colonnes par ordre alphabétique de la clé.

Ensuite, on transforme les colonnes en lignes et on met un deuxième mot clé au-dessus. On peut ensuite procéder de la même manière en triant les colonnes par ordre alphabétique de la clé. Le résultat de la grille donne donc le cryptogramme final.

Le problème avec cette méthode est similaire à celui de la substitution, on peut faire des inversions et tenter d'identifier des mots.

5.3.1.1.2) XOR (ou exclusif)

Le XOR est une opération de base du CPU, ce qui la rend très rapide.

L'idée ici est de faire une opération XOR entre un texte en clair et une clé de chiffrement. De la même manière, on peut déchiffrer le texte en faisant le cryptogramme XOR la clé.

Par exemple, si le message en clair est la lettre `A`, on convertit cela en binaire, ce qui donne `00001010`. Si notre clé est la lettre `H` que l'on convertit en binaire `01001000`. On peut faire un XOR dessus pour obtenir le cryptogramme :

```
    00001010 => A (texte en clair)
XOR 01001000 => H (clé)
    --------
    01000010 => B (cryptogramme)
```

Maintenant pour déchiffrer, il suffit de prendre le cryptogramme et la clé et de refaire un XOR

```
    01000010 => B (cryptogramme)
XOR 01001000 => H (clé)
    --------
    00001010 => A (texte en clair)
```

Le problème avec cette méthode est que si on connait un exemple dans lequel on a à la fois le texte clair et le cryptogramme, on peut retrouver la clé en utilisant le même mécanisme.

De la même manière, on peut facilement trouver la clé en faisant de l'analyse par fréquence sur base de la longueur de la clé.

Cependant, si la clé est complètement aléatoire et de la même longueur que le message, il s'agit alors d'un "one-time pad" ou "masque jetable" et c'est théoriquement un code incassable.

5.3.1.1.3) Masque jetable

Un masque jetable est une technique de chiffrement se basant sur une longue liste non répétitive et aléatoire de lettres (le masque). Chaque lettre est utilisée pour code une lettre du texte clair.

Par exemple, on peut additionner le rang de la lettre du masque avec celui du texte clair modulo 26 pour obtenir le rang de la lettre du texte chiffré.

Ou alors, on peut procéder en utilisant un XOR comme précédemment.

Puis ce que la clé est de la même longueur que le message et qu'elle est entièrement aléatoire, il est impossible de la déchiffrer. Cependant, cette technique a le gros désavantage d'avoir des clés très longues et peu pratiques.

5.3.1.2) Niveaux de chiffrements

Le niveau de chiffrement sera établi en fonction des groupes de personne contre lesquels on désire se protéger. Le niveau sera différent si on souhaite se protéger contre ses concurrents ou de la NSA.

C'est également une question de longueur de clé. Plus la clé est longue, plus ce sera sécurisé. Par exemple, un cadenas à trois chiffres aura 1 000 combinaisons possibles tandis qu'un cadenas à six chiffres aura 1 000 000 combinaisons possibles.

5.3.1.3) Algorithme comme garant de la sécurité

L'algorithme qui est utilisé pour faire de la cryptographie est le garant (avec la ou les clés) de la sécurité des messages.

Un bon algorithme doit être public (pour être vérifiable), sûr (éprouvé pendant plusieurs années et par des experts) et indépendant (sans coopération avec des organismes ayant des intérêts contradictoires tels que la NSA).

Dans un algorithme sûr, que l'espion ne soit qu'avec du texte chiffré, avec des correspondances texte en clair et texte chiffré ou même avec du texte clair choisi, l'espion ne peut pas trouver la clé.

5.3.2) Cryptographie symétrique et asymétrique

Entre les deux extrêmes que nous venons de voir (code de césar d'un côté et le one-time pad). Divers mécanismes ont été développés.

5.3.2.1) Crypto système à clé secrète (cryptographie symétrique)

Une clé secrète est partagée entre toutes les personnes qui doivent communiquer.

Les systèmes historiques correspondent à cette catégorie, mais aujourd'hui, on a également des manières plus sécurisées telles que AES.

Bien que ce crypto système ait été le standard pendant plusieurs siècles, il a quelques problèmes.

Par exemple, les clés doivent être distribuées et rester sûr, si la clé est compromise, tous les messages sont compromis. Et si une clé différente est utilisée par paire d'utilisateur, le nombre de clés nécessaires pour rester sûre devient très élevé.

5.3.2.2) Crypto système à clé publique (cryptographie asymétrique)

À la place d'avoir une clé secrète partagée par tout le monde, on va avoir une clé qui ne peut faire que du chiffrement (la clé publique), et une clé de déchiffrement (la clé privée).

La clé publique, comme son nom l'indique, peut être partagée partout. De cette manière, si on veut communiquer avec 100 personnes, à la place d'avoir 100 clés, on va avoir une seule clé publique partagée partout.

N'importe qui peut chiffrer un message avec cette clé publique, mais seule la clé privée pourra permettre de la déchiffrer.

Cela règle donc les problèmes de la cryptographie asymétrique, cependant ce système a le désavantage d'être beaucoup plus lourd et de demander beaucoup de calcul (ce qui est la raison pour laquelle il est si récent).

Il reste tout de même un problème, celui de la confiance. Comment pouvons-nous être sûrs que la personne qui donne la clé privée est bien qui elle prétend être ?

Pour cela, on peut utiliser des tiers de confiance déjà connus à l'avance qui pourront certifier des clés (c'est par exemple le cas de l'entreprise Let's Encrypt qui permet de certifier les clés TLS pour le HTTPS).

5.3.2.2.1) RSA

> *Pour apprendre et comprendre le fonctionnement de RSA, allez voir cette playlist[48], pour avoir des détails sur le calcul uniquement, vous pouvez consulter cette vidéo[49] et si vous voulez utiliser quelque chose de plus simple que l'algorithme d'Euclide étendu, vous pouvez utiliser le théorème de Bachet-Bézout[50] à la place.*

Voici comment calculer les clés publiques et privées en RSA,

1. Tout d'abord, on choisit deux nombres premiers, que l'on va appeler p et q. Par exemple $p = 3$ et $q = 5$.
2. Ensuite, on fait le produit de ces deux nombres, que l'on va appeler n, soit $n = pq = 3*5 = 15$
3. Ensuite, on calcule la fonction phi tel que $\Phi(n) = (p-1)(q-1) = (3-1)(5-1) = 2*4 = 8$
4. Ensuite, on choisit un entier e dont le PGCD avec $\Phi(n)$ vaut 1; autrement dit, il faut trouver un nombre e tel que e et $\Phi(n)$ soient premiers entre eux (aucun facteurs premiers communs)
5. Enfin, il faut trouver le nombre de déchiffrement d tel que $ed \mod \Phi(n) = 1$, soit $ed - kn = 1$.
   - On peut ici utiliser le théorème de Bachet-Bézout[51] qui dit que si deux nombres (a et b) sont premiers entre eux, alors on peut trouver des entiers x et y tel que $ax + by = 1$. Ici, on a

[48] https://www.youtube.com/playlist?list=PLjgrsP5Vg40mVUj2cmzUyb6Ik1IiCj8P9
[49] https://www.youtube.com/watch?v=kYasb426Yjk
[50] https://www.youtube.com/watch?v=9nDQzF4jmBs

e et $\Phi(n)$ qui sont premiers entre eux, par conséquent on peut appliquer l'algorithme. En considérant x comme étant d et y comme étant k.

6. La clé publique est $\{e, n\}$ et la clé privée est $\{d, n\}$.

On peut donc maintenant chiffrer un message m (qui doit être inférieur à n) en faisant $m^e \bmod n = c$. Et on peut déchiffrer un message en faisant $c^d \bmod n = m$. $

De même on peut signer un message en "déchiffrant" un texte en clair, $m^d \bmod n = s$ et on peut le vérifier en "chiffrant" la signature, $s^e \bmod n = m$.

Les chapitres ci-dessous sont optionnels pour le cours, mais aident à mieux comprendre le fonctionnement de RSA.

5.3.2.2.1.1) Exponentiation modulaire

Pour pouvoir avoir une clé pour déchiffrer et une clé pour chiffrer, il faut pouvoir trouver un moyen de faire une opération facilement (chiffrement avec clé secrète) mais de rendre l'opération inverse très compliquée (déchiffrement) si on ne connait pas une valeur supplémentaire (clé privée).

Cette fonction pour RSA c'est l'exponentiation modulaire, l'idée est que si on fait $m^e \bmod n = c$, cela demande beaucoup d'essai-erreur pour pouvoir en partant de e, n et c revenir à m.

Cependant, si on a un autre exposant (d) on peut l'inverser simplement en faisant $c^d \bmod n = m$.

Si on applique e et d ne même temps, le message ne change donc pas, ainsi $m^{ed} \bmod n = m$, cela sera important pour plus tard.

5.3.2.2.1.2) Factorisation de nombres premiers

Maintenant, il faut trouver un moyen de trouver e, d et n de manière à rendre tout cela possible. Pour cela, il faut trouver une autre fonction qui simple à faire dans un sens et compliquée à faire dans l'autre.

Cette fonction dans RSA c'est la factorisation de nombres premiers (pour rappel, un nombre premier est un nombre qui ne peut être divisé entièrement que par un ou lui-même). On sait que tous les nombres ont exactement une factorisation de nombres premiers, cependant, cette factorisation de plus en plus compliquée en fonction de la grandeur du nombre.

Cette propriété fait que la factorisation est un très bon candidat, car si on utilise des nombres premiers assez grands, il sera impossible de le factoriser avec nos moyens actuels.

Ainsi, on peut trouver deux nombres premiers très grands et les multiplier ensemble. Le produit de ces deux nombres premier sera très simple à calculer, mais très difficile à inverser parce que la multiplication est simple, mais la factorisation est elle très complexe.

Maintenant, il faut trouver une fonction qui dépend de la connaissance de la factorisation de n.

[51] https://www.youtube.com/watch?v=9nDQzF4jmBs

5.3.2.2.1.3) Indicatrice d'Euler, fonction Phi

Cette fonction, c'est indicatrice d'Euler que l'on va ici appeler phi. Ainsi la fonction $\Phi(n)$ donne le nombre d'entiers positifs plus petit que n qui ne partagent pas de facteurs premiers avec n. Par exemple $\Phi(8) = 4$ car huit ne partagent pas de facteurs communs avec 1, 3, 5 et 7, mais partagent des facteurs communs avec 2, 4 et 6.

Cette fonction est donc très compliquée à calculer pour des grands nombres, mais vraiment simple à calculer pour des nombres premiers. Puisqu'un nombre premier ne peut être divisé que par 1 ou lui-même, la fonction phi revient à dire $\Phi(n) = n - 1$.

De même la fonction est multiplicative, donc si a et b sont premiers, $\Phi(a*b) = (a - 1)(a - 1)$.

Il faut maintenant trouver un moyen de lier la fonction phi à l'exponentiation modulaire.

5.3.3) Signatures cryptographiques

Une signature permet d'identifier que quelqu'un a bien écrit quelque chose. Une signature doit être authentique, non falsifiable, non réutilisable. De même, un document signé ne peut pas être modifié et une signature ne peut pas être reniée.

Il faut pouvoir faire toutes ces propriétés dans les signatures numériques (cryptographiques).

5.3.3.1) Avec de la cryptographie symétrique

Pour faire un système de signature avec une seule clé secrète, il faut trois acteurs, le signataire, le destinataire et un tiers de confiance. C'est le tiers de confiance qui donne toute sa sécurité à la structure.

Chaque acteur partage une clé avec le tiers de confiance. Ainsi lorsque le signataire va partager le document avec sa clé au tiers de confiance.

Le tiers de confiance peut donc confirmer la signature puisque qu'il connait la clé secrète. Le tiers de confiance peut donc ajouter une certification sur le document et le signer en utilisant la clé partagée avec le destinataire.

Le destinataire peut donc ensuite valider que le tiers de confiance a authentifié la signature et donc considérer la signature comme correcte, car le tiers de confiance a utilisé sa clé.

Le problème ici est que toute la sécurité du système réside dans le tiers de confiance, donc si le tiers de confiance est compromis, toutes les signatures sont compromises.

5.3.3.2) Avec de la cryptographie asymétrique

Avec de la cryptographie asymétrique, on a uniquement besoin du signataire et du destinataire. Le destinataire connait la clé publique du signataire. Le rôle d'un potentiel tiers de confiance ici est seulement d'authentifier le propriétaire de la clé (cela n'a donc besoin d'être fait qu'une seule fois).

Nous avons vu que lorsque quelque chose est chiffré avec une clé publique, seule la clé privée peut la déchiffrer. Mais à l'inverse, lorsque quelque chose est signé avec une clé privée, elle peut être validée par les clés publiques.

La procédure de signature équivaut globalement à appliquer l'algorithme de déchiffrement sur le texte à signer. Ainsi, il suffira au destinataire de chiffrer le résultat pour retrouver le texte d'origine.

5.3.3.3) Systèmes de confiances

5.3.3.3.1) Certificats numériques X509

Les certificats numériques x509 lient une clé publique à une identité (nom de domaine, adresse email, etc) en utilisant une signature cryptographique.

Toute personne faisant confiance au tiers connait la clé publique de ce dernier afin de pouvoir vérifier les certificats. C'est notamment cela qui est utilisé sur internet pour vérifier les connexions HTTPS. Le navigateur possède une liste de clés publiques de tiers de confiances pour vérifier les certificats.

Certains tiers de confiances offrent leurs services de vérification gratuitement, c'est par exemple le cas de "Let's Encrypt", cependant la vaste majorité sont payants, mais ont l'avantage d'offrir une vérification beaucoup plus rigoureuse.

Let's Encrypt ne fait que vérifier que la demande de certificat est bien faite depuis une machine sous le nom de domaine demandé, alors que d'autres tiers de confiance vont aller se renseigner sur l'entreprise et l'appeler pour avoir une confirmation de l'authenticité de la demande.

5.3.3.3.2) Système de PGP (Pretty Goog Privacy)

C'est un système à clé publique sans tiers de confiance. L'identité des personnes est garantie de manière transitive.

Ainsi, si Alice connait Bob, elle peut certifier sa clé publique en la signant. Bob peut alors partager la clé signée par Alice. De cette manière, toute personne faisant confiance aux fréquentations d'Alice pourra faire confiance à Bob en vérifiant la signature.