

Résumé global

Question 1 (processus)

Définissez le concept de processus, décrivez les mécanismes de communication entre les processus et détaillez les algorithmes de CPU scheduling

Définition du concept de processus

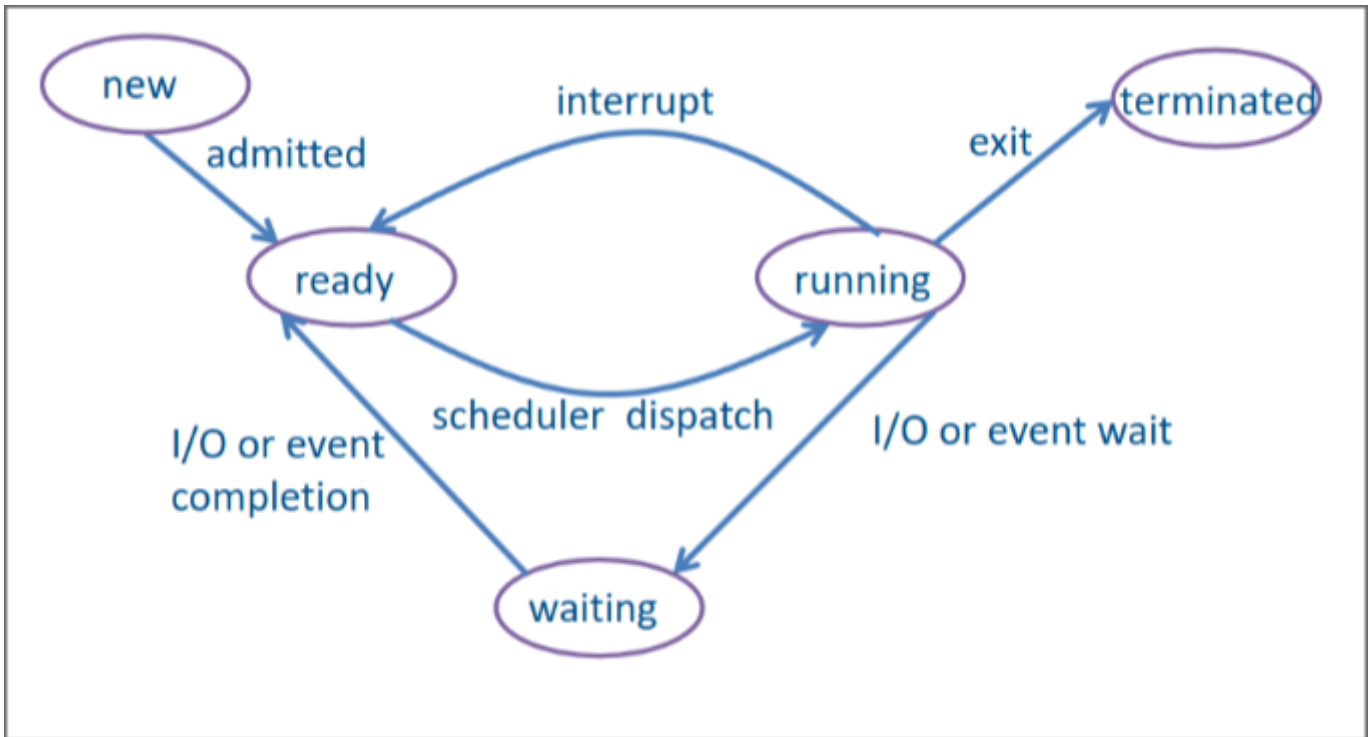
Un processus est un **programme en cours d'exécution**. Il comporte **le code du programme**, le **program counter** (qui indique à quel instruction du programme, on est et quelle est la suivante), **la pile** et **les données** du programme.

La création d'un nouveau processus est faite via l'appel système `fork`

Un processus est caractérisé par :

- Un **PID**, identifiant du processus
- Un **PPID** (l'identifiant du processus père qui a créé le processus actuel)
- La **priorité** du processus
- Le **temps** consommé au CPU
- La **table des fichiers ouverts**
- L'**état** du processus

Voici les différents états des processus et leurs relations :



- **new**, correspond à un programme qui a été sélectionné pour être démarré. Ses instructions ont été copiées en mémoire, mais 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 s'exécuter jusqu'à écoulement du temps imparti ou qu'un processus de plus haute priorité arrive, dans quel cas il retourne en **ready**. S'il demande des ressources, il ira en état **waiting** et s'il a fini son exécution ou qu'il est tué, il ira en état **terminated**.
- **waiting**, le processus est en état d'attente d'un évènement (par exemple, appui d'un bouton) ou de ressources (par exemple, lecture d'un fichier).
- **terminated**, une fois le processus terminé ou tué, il libère la totalité des ressources qu'il a détenues.

Pour pouvoir exécuter plusieurs processus et donner l'impression que les processus s'exécute simultanément, le **scheduler** (un élément du système) va alterner très vite entre les différents états de chaque processus.

Le scheduler va classer les processus selon s'ils sont des processus de calcul (CPU) ou des processus d'entrée-sortie. Il va toujours privilégier les processus d'entrée sortie, car ce sont eux qui donnent l'interactivité du système et qui donne l'illusion que tout s'exécute en même temps.

Pour changer de processus, le système va sauvegarder **le contexte** (les données) du processus en cours. Puis charger le contexte du processus à exécuter.

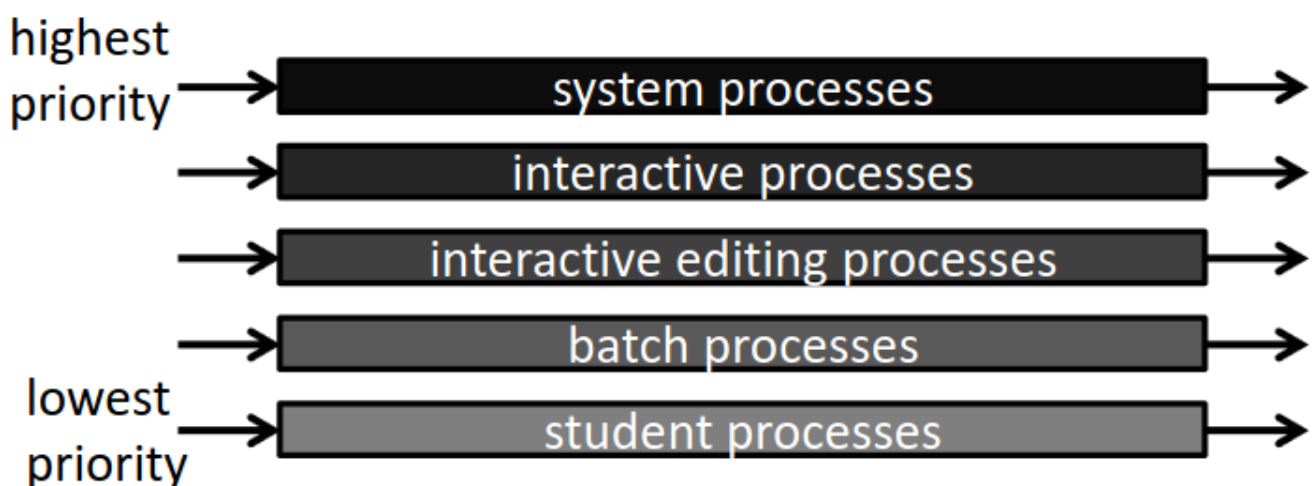
Différents algorithmes de CPU scheduling

Le scheduler a pour but de maximiser l'utilisation du CPU pour avoir un débit (nombre de processus terminé par unité de temps) le plus important possible.

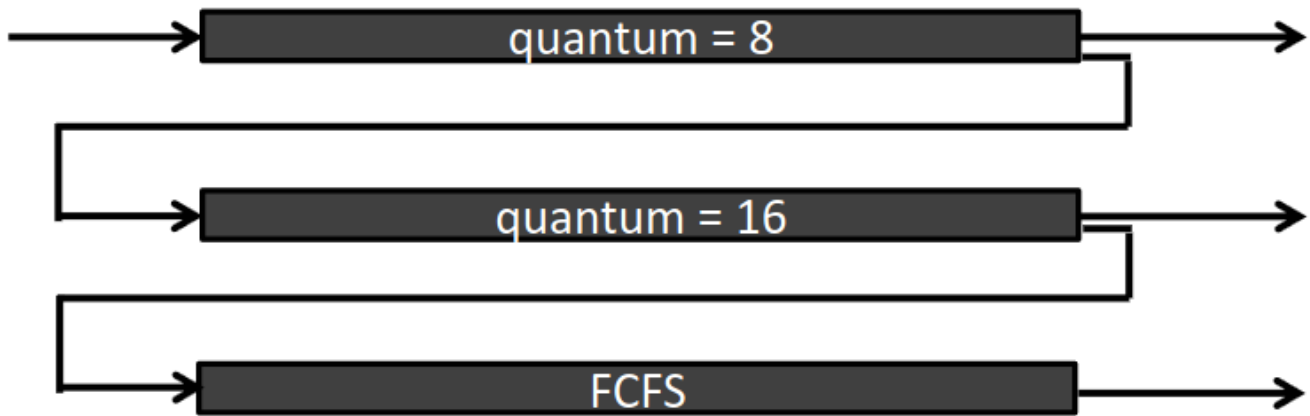
C'est pour cela que le scheduler doit utiliser un algorithme pour pouvoir choisir quels processus lancer afin de maximiser l'utilisation CPU.

Comme algorithmes, on a par exemple,

- **FCFS, First Come First Served**, premier arrivé, premier servi. Simple à implémenter, mais si un des premiers processus est lent, toute la file est ralentie.
- **SJF, Shortest-Job First**, ordonne les processus selon leur durée pour mettre les processus les plus rapides au début. Cet algorithme est seulement possible si on sait à l'avance la durée des processus, ce qui est rarement le cas aujourd'hui.
- **Priorité**, trie les processus par ordre de priorité, ainsi les processus de plus haute priorité seront exécutés avant. De plus, cet algorithme peut aussi être **préemptif**, donc si un processus de plus haute priorité arrive, le processus en cours peut être mis en état ready (mis en pause) pour lui laisser place. Cependant, cela peut mener à de la famine parmi les basses-priorité si des processus de plus haute priorité arrivent tout le temps.
 - Le problème de famine peut être résolu en combinant l'age et la priorité, ainsi les processus ayant trop attendu passe devant
- **Round-Robin Scheduling, Tourniquet**, les processus sont servis dans l'ordre d'arrivée, chaque processus reçoit le CPU pour un temps déterminé (**quantum**). Lorsque qu'un processus épuise son quantum, il est remplacé (c'est donc **préemptif**).
 - Il faut toute fois faire attention, si le quantum est **trop grand** cela donnera une impression de lenteur et de lag à l'utilisateur
 - En revanche, si le quantum est **trop petit**, cela diminue l'efficacité du CPU à cause des changements de processus à répétition
- **Multilevel Queue Scheduling**, avoir des files différentes suivant la nature du processus. Une priorité et un mécanisme de scheduling propre est attaché à chaque file.



- **Multilevel Feedback Queue Scheduling**, comme le précédent, sauf que les processus n'appartiennent plus à une file et peuvent migrer vers d'autres files. Chaque file possède des caractéristiques spéciales (quantum, algorithme, etc).



Description des mécanismes de communication entre les processus

Il est nécessaire que les processus puissent communiquer entre eux pour se partager des informations, répartir les calculs, etc.

La **communication inter-process (IPC)** représente les mécanismes de communication entre les processus.

Voici différentes options,

- Par **fichiers**, cependant, cela est très lent et difficile à synchroniser
- Par **tubes**, qui sont des petits fichiers gérés en file circulaire qui sont souvent mis en cache (ce qui les rend donc très efficaces), si le fichier devient trop grand, on peut alors le découper en blocs.
 - Les **tubes non-nommés** sont des tubes temporaires tels que `stdin`, `stdout` ou `stderr`. Il est possible de rediriger les différents tubes. Les tubes non-nommés sont liés entre père et fils.
 - Les **tubes nommés** sont permanents via des fichiers spéciaux sur le système. Les tubes nommés ne sont pas liés entre père et fils et peuvent être utilisé par des processus complètement indépendants.
- La **mémoire partagée** est un moyen très commun pour partager des informations. Il s'agit d'une zone mémoire commune à plusieurs processus, la taille est entièrement configurable (comme avec `malloc`) et après un `fork` le processus fils hérite de la zone mémoire partagée.
 - Lorsque l'on alloue une zone de mémoire partagée via `shmget`, on lui indique une clé (identifiant de la zone), une taille en octet et les permissions de la zone.
 - Une fois allouée, on peut récupérer un pointeur vers la zone de mémoire avec `shmat` auquel on passe l'identifiant retourné par `shmget`, l'adresse dans la zone (généralement 0), et des flags tel que read only (souvent NULL).

- Une fois qu'on a terminé, on peut ensuite détacher la zone mémoire avec `shmdt` et éventuellement la supprimer avec `shmctl`

Question 2 (processus)

Détaillez les problèmes de synchronisation entre processus, définissez et détaillez le fonctionnement des threads et le problème et les solutions à un interblocage

Les problèmes de synchronisation entre processus

Lorsque plusieurs processus coopèrent, ils doivent fréquemment interagir entre eux et ils doivent parfois attendre qu'une opération soit effectuée par un autre processus pour travailler. Lorsqu'un processus attend un autre, on parle de **point de synchronisation**.

C'est pourquoi il faut avoir des mécanismes pour envoyer et attendre des événements, des processus.

Sous UNIX les deux mécanismes les plus importants sont les **signaux** (événements capturés par un processus permettant de signaler aux processus une erreur tel que `SIGINT` pour un CTRL+C par l'utilisateur dans le terminal) et les **sémaphores**.

Un sémaphore est une variable entière en mémoire qui n'est accédée qu'au moyen de fonction atomiques `p()` et `v()`. La fonction p va attendre que la valeur soit plus grande que 0 pour la remettre à zéro et continuer. Et la fonction v va incrémenter la variable, ce qui va réveiller tous les processus qui l'attendait.

Section critique

La communication entre processus pose également d'autres problèmes, par exemple pour la modification d'une donnée commune comme de la mémoire partagée par exemple. Pour résoudre cela, il faut mettre en place des **sections critiques**, il s'agit d'un ensemble d'instructions qui doivent être exécutées du début à la fin sans interruption. Cela assure donc que la donnée n'est pas modifiée en cours de route et que les données ne deviennent pas incohérentes.

Pour faire une section critique, on peut,

- Utiliser une **variable partagée** ainsi, on attend que la variable soit à 0, lorsqu'elle est à 0, on la met à 1, on fait la section critique et on la remet à 0.
 - Le problème avec cette méthode est qu'elle n'est pas fiable, dans le cas où le processus se ferait mettre en pause entre le `while` (attente) et la mise de la variable à 1. De plus cette méthode utilise du CPU pour rien pour faire un `while`
- Utiliser une méthode **par alternance**, on associe chaque processus à un tour, ainsi le processus un va attendre que le tour soit à 0, et l'autre va attendre que le tour soit à 1. Ensuite, ils font leur section critique et inverse la variable.
 - Cette méthode ne souffre pas du problème de fiabilité précédent, cependant elle est beaucoup plus compliquée à gérer, surtout s'il y a plus de deux processus à synchroniser. Aussi, elle consomme également du temps CPU pour juste un `while`
- Synchronisation **par fichier**, consiste à créer un fichier (lock file) en mode exclusif (donc un seul processus peut y accéder), donc on tente de créer le fichier (jusqu'à ce que ça fonctionne), on fait la section critique et on supprime le fichier.
 - Cela a cependant l'inconvénient d'être fortement ralenti par le système de fichier, car pour chaque tentative d'accès au fichier, le processus doit être mis en état waiting, ready puis de nouveau running.
- Synchronisation **hardware**, consiste à utiliser des instructions assembleurs pour protéger une section critique. Lorsque l'on veut entrer dans une section critique, on inverse une variable booléenne "lock", si elle était initialement à true, on attend et on recommence l'opération en boucle. Si elle était initialement à false, on entre en section critique. Une fois la section critique finie, on remet à false.
 - Cette technique est utilisée par le système d'exploitation pour implémenter certains mécanismes de protection comme les sémaphores. Cependant, elle a quand même le désavantage du `while` cité précédemment.
- Synchronisation **par sémaphore**, cette méthode de synchronisation est celle à **priviléger**, avant d'entrer en section critique, on attend sur le sémaphore correspondant à l'autre processus. Ensuite, on fait la section critique et une fois fini, on met notre propre sémaphore à 1 pour indiquer à l'autre processus qu'il peut y aller.

Fonctionnement des threads

Un **thread** est une sorte de mini-processus, chaque processus peut avoir plusieurs files d'exécutions (il est alors **multithreads**). L'avantage d'un thread est que c'est beaucoup plus rapide et léger à créer qu'un processus complet. On peut par exemple avoir un thread pour la saisie de texte, un thread pour l'affichage et un thread pour la vérification.

De plus, les threads d'un même processus partagent les informations et un système à plusieurs cœurs peut exécuter les threads d'un même processus sur plusieurs cœurs différents pour améliorer les performances.

Les threads peuvent être implémentés à deux niveaux, au niveau **noyau** (kernel) où il est alors pris nativement en charge par le SE, ou au niveau **utilisateur** où il doit alors être supporté au travers de libraires externes.

Les threads peuvent donc être implémentés selon plusieurs modèles :

- **many-to-one**, soit entièrement en espace utilisateur avec des libraires externes. Le SE ne voit alors que le processus. La création de thread sous ce modèle est rapide, mais le scheduler du système ne peut pas bien optimiser les performances. Donc si un thread fait une activité bloquante, cela risque d'empêcher tous les autres threads de travailler. Ce modèle n'est plus adapté aux CPU multicœurs.
- **one-to-one**, soit tout en espace kernel. Les threads sont complètement gérés au niveau du SE. Le scheduling est donc plus avantageux et est tout à fait compatible avec les CPU multicœurs. Cependant, ce modèle est couteux pour le SE qui doit tout gérer. C'est par exemple ce modèle qui est utilisé sous Linux.
- **many-to-many**, soit l'utilisation d'un pool de threads auquel les threads utilisateurs sont assignés à la volée. Cela combine les avantages des deux modèles précédents, cependant il est assez complexe et le SE doit avoir été construit autour de ce modèle.

Il y a également quelques problèmes à considérer avec les threads. Par exemple, que se passe-t-il lors d'un fork ou d'un execl ? faut-il dupliquer tous les threads ? Faut-il écraser les threads ?

Ou encore pour terminer un thread, faut-il qu'un thread demande la terminaison d'un autre (libération **asynchrone**) ? Ou alors chaque thread vérifie lui-même s'il doit continuer (libération **différée**) ?

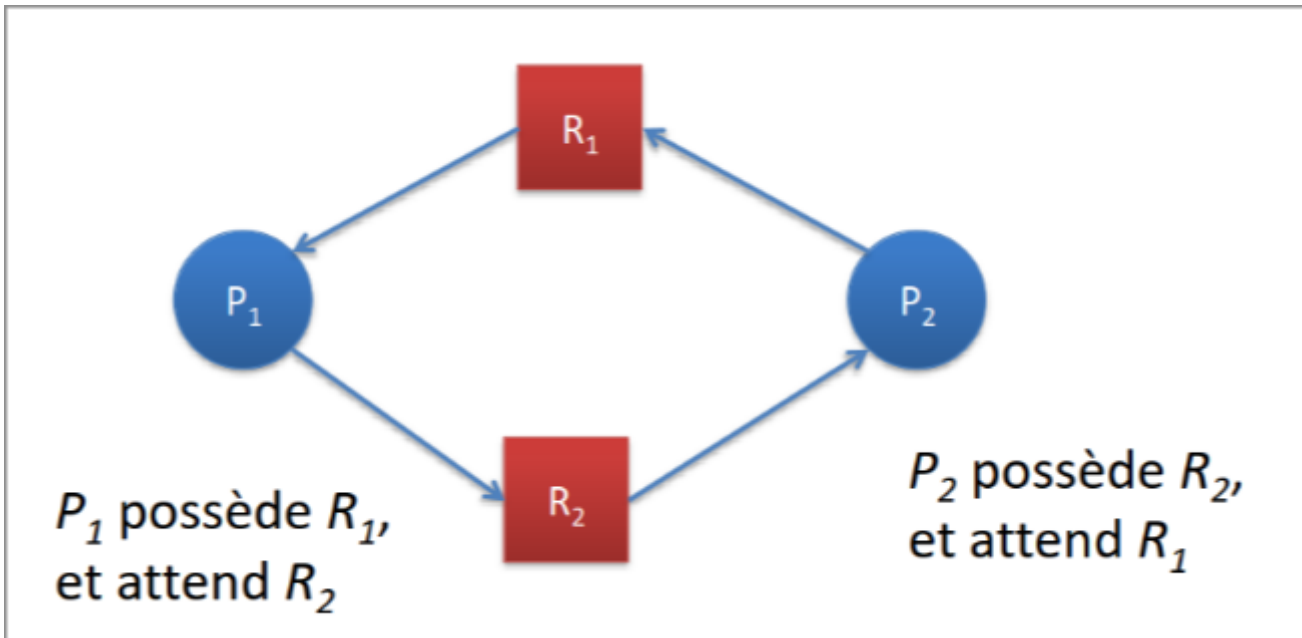
Lorsqu'un signal est envoyé au processus, quel thread doit le recevoir ? Tous certains ou un seul ?

Toutes ces décisions dépendent beaucoup de l'OS utilisé.

Problème et solutions d'interblocages

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

Cependant, 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.



Pour empêcher un interblocage, il faut pouvoir empêcher au moins une de ses conditions :

- L'**exclusion mutuelle**, lorsque les processus utilisent des ressources non partageables. On ne peut pas empêcher cela.
- La **détention et l'attente**, lorsqu'un processus détient une ressource et demande une autre ressource en même temps. Pour empêcher cela, il faudrait
 - Soit, que les processus demandent toutes les ressources dès le départ. Seulement, on ne connaît généralement pas cette information.
 - Soit, dès qu'un processus demande une nouvelle ressource, il doit libérer toutes les autres puis toutes les récupérer en plus de la ressource demandée. Cependant, cela peut mener à l'accumulation de ressources qui peut mener à une famine parmi les autres processus.
- L'**impossibilité de réquisitionner une ressource**, il est impossible de réquisitionner une ressource, car on ne peut pas savoir si elle sera dans un état cohérent quand elle sera réquisitionnée.
- L'**attente circulaire**, on peut détecter un cycle d'attente et si un cycle survient faire certaines actions.

Pour éviter complètement l'attente circulaire, il faut 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. Tant que le système évolue d'état sûr en état sûr, on est sûr qu'un interblocage ne surviendra.

Utilisation de l'algorithme du banquier

Pour vérifier si on est en état sûr, on peut utiliser l'**algorithme du banquier**. On doit calculer la matrice des allocations courantes (C) correspondant aux besoins - les demandes, des demandes (R) correspondant aux besoins - allocations courantes et des ressources disponibles (A) correspondant aux ressources existantes - allocations courantes.

Pour chaque processus, on regarde si on peut remplir sa demande à partir des ressources disponibles. Si oui, on marque le processus comme terminé et on ajoute ses ressources aux ressources disponibles.

On répète cela en boucle jusqu'à ce que tous les processus soient terminés, si cela n'est pas possible, alors l'état n'est pas sûr.

Pour allouer depuis un état sûr, lorsqu'un processus demande une ressource, on va hypothétiquement diminuer les ressources disponibles (et les besoins), et on va augmenter les ressources allouées. On effectue ensuite l'algorithme de vérification pour voir si l'état qui arriverait après l'allocation est sûr.

S'il l'est alors, on peut allouer, s'il ne l'est pas, il faut attendre.

Détection d'un interblocage

Le problème d'éviter un cycle d'attente, c'est qu'il faut connaître à l'avance les besoins des processus, ce qui est souvent inconnu.

Alors si on veut simplement pouvoir détecter et traiter un interblocage lorsqu'il survient. On peut savoir qu'un interblocage arrive si l'utilisation du CPU est trop faible et que les processus sont en état waiting.

Correction ou non d'un interblocage

Pour corriger l'interblocage, il faut alors tuer un processus qui pose un problème et tenter de maintenir les ressources dans un état cohérent.

Si possible, il faut faire un *rollback* vers le contexte où le système était avant pour s'assurer que les ressources ne sont pas incohérentes. Cela nécessite toute fois que le contexte ait été sauvegardé.

Certains systèmes cependant ne corrigent pas les interblocages (par exemple UNIX) car les interblocages ne surviennent pas fréquemment, c'est alors à l'administrateur·ice de le corriger.

Question 3 (mémoire)

Détaillez le fonctionnement de la mémoire, expliquez l'allocation, la pagination et la segmentation en concluant par une description de l'utilisation conjointe de ces deux mécanismes.

Fonctionnement de la mémoire

La mémoire est une ressource indispensable et est partagée entre tous les processus. C'est une suite non structurée d'octets.

Un processus doit se trouver entièrement en mémoire afin de s'exécuter, cela nécessite donc de trouver une gestion efficace de la mémoire pour permettre à tous les processus de s'exécuter.

Chaque processus doit disposer de sa propre zone mémoire pour éviter des problèmes de sécurité ou d'intégrité des données. Si un processus essaye d'accéder à quelque chose en dehors de sa zone mémoire, on a une "segmentation fault" qui fait crash le programme. La mémoire peut cependant être partagée entre les processus si cela est explicitement demandé.

Types de mémoires

Il existe différents types de mémoire, les **registres** qui sont des zones mémoires attachées au processeur, leur taille est généralement très réduite (quelques Ko) mais sont très rapides, car pouvant être accédés en un seul cycle d'horloge du processeur.

La **mémoire vive** est la mémoire principale du système, souvent présente dans des quantités de l'ordre de quelques Go, elle est toute fois plus lente par rapport au CPU, mais reste beaucoup plus rapide qu'un disque.

La **mémoire cache** a pour but d'améliorer les performances du système en mettant en mémoire plus rapides les données fréquemment utilisée. Il existe trois niveaux de mémoire cache, le niveau 1 stocke temporairement les instructions et données tandis que les niveaux 2 et 3 sont présentes entre la mémoire vive et celle de premier niveau.

Enfin, il y a la **mémoire virtuelle** qui est une mémoire simulée par le système et qui utilise le disque dur comme mémoire supplémentaire. Elle peut être de l'ordre de plusieurs To si on le souhaite, mais elle est très lente. Il faut donc l'utiliser pour stocker des données peu utilisées.

Translation d'adresse

Lorsqu'un programme est chargé, il est placé entièrement placé en mémoire à son adresse de départ. Les instructions du programme font référence à l'adresse `0` qui est l'**adresse logique** du début du programme.

Il faut donc traduire l'adresse logique en **adresse physique**, c'est-à-dire en une position particulière dans la mémoire RAM.

Cette conversion peut être faite, soit à la **compilation**, l'adresse physique est alors hard-codée dans le programme, ce qui n'est plus du tout d'actualité aujourd'hui, au **chargement** où l'adresse de départ est choisie au moment du lancement du processus ou à l'**exécution** cependant cela demande un hardware précis est utilisé par la segmentation.

Cette conversion se fait par le **Memory Management Unit (MMU)** qui est un composant matériel spécialisé dans cette opération.

Allocation de la mémoire

Pour allouer la mémoire à chaque processus, il y a différentes méthodes dépendant du type de système.

En **mono-programmation**, où un seul processus s'exécute en même temps que le système d'exploitation (genre MS-DOS), le processus a toute la mémoire pour lui, sauf pour une petite partie réservée au SE. Le BIOS réalise une gestion des périphériques.

En **multiprogrammation**, on peut soit diviser la mémoire en **partitions fixes** de taille variable pour ensuite associer chaque processus dans la plus petite zone qui peut la contenir. Cela a cependant le problème de gaspiller beaucoup de ressources (par exemple, un processus de quatre unités qui est dans une partition de sept unités).

Ou alors, on peut utiliser des **partitions variables** qui sont allouées selon les besoins des processus. Pour pouvoir allouer, il suffit donc de créer une partition de la taille adaptée.

Cela peut être fait avec une **table de bits** qui est une cartographie de la mémoire divisée en blocs (petits blocs = plus précis, mais plus lent à parcourir), on note un 1 si le bloc est occupé ou un 0 s'il est libre. Il suffit alors juste de trouver un espace avec suffisamment de zéro consécutif pour allouer une zone. Cela a cependant le désavantage que la table peut prendre beaucoup de place et donc être longue à parcourir.

Sinon, on peut utiliser une **liste chaînée** où chaque élément de la liste correspond à une partition et sait si la partition est libre, la position de début de la partition, sa longueur et le lien avec l'élément suivant de la liste. Cela a l'avantage de pouvoir être parcouru beaucoup plus rapidement et de faciliter la fusion de blocs libres lorsqu'un programme libère une partition. Cela a cependant le désavantage de mener à de la **fragmentation interne** (bouts de mémoire non utilisé parmi la zone allouée à un processus).

Pour choisir quelle partition allouer au processus, il y a le choix entre plusieurs algorithmes.

- Le **first-fit**, qui consiste à prendre la première zone assez grande. C'est souvent cette méthode qui est utilisée, car elle est très rapide.
- Le **best-fit** qui consiste à parcourir toute la mémoire jusqu'à trouver la meilleure partition, cela est cependant une mauvaise idée parce que c'est lent et ça crée beaucoup de **fragmentation externe** (bouts de mémoire non utilisable puisque trop petit en dehors des zones allouées) en créant des bouts de partitions trop petites pour être utilisées.
- Le **worst-fit** qui consiste à parcourir la mémoire pour prendre la partition la plus grande possible pour éviter d'avoir trop de fragmentation externe. Cet algorithme est très rapide pour les listes triées par taille.

Pour résoudre les problèmes de fragmentation, il faut pouvoir rassembler les zones mémoires, pour cela peut utiliser le **compactage** qui consiste à rassembler les zones occupées et les zones libres

ensemble. Cela est cependant coûteux en ressource et lent. Une autre solution est d'utiliser la **pagination**.

Pagination

La pagination permet d'avoir un espace d'adressage en mémoire physique non contigu tout en ayant un espace d'adressage en mémoire logique contigu.

Pour cela, la mémoire logique est divisée en blocs de taille fixe (les **pages**), on fait de même avec la mémoire physique où les blocs sont appelés des **frames**. Chaque page correspond à une frame via la **table des pages**.

Lorsqu'un processus démarre, on va donc calculer le nombre de pages nécessaires pour son exécution, ensuite, on regarde le nombre de frames disponible. S'il y a suffisamment de frames disponibles, on peut lier les pages aux frames et les allouer au processus, sinon il faut attendre.

Grâce à ce système, la fragmentation externe n'existe plus et la fragmentation interne sera en moyenne d'une demi-page par processus.

Segmentation

L'utilisateur·ice voit un programme comme un ensemble de blocs distinct dont les données ne sont pas mélangées avec les autres (genre, code du programme, données, pile, etc). C'est pourquoi on va diviser l'espace d'adressage logique du processus en segments. Chaque segment a un nom, un numéro et une taille.

On peut par exemple avoir un segment pour les données du programme, un segment pour la pile, un segment pour le code.

Cette division de la mémoire permet au système d'offrir une protection adaptée pour ne pas mélanger les différentes données, on ne peut donc pas écrire sur le code du programme en modifiant les données.

Cela permet aussi de pouvoir partager certains segments entre plusieurs processus (ex. libraires systèmes) pour économiser l'espace mémoire.

Utiliser uniquement de la segmentation peut mener à un retour à la fragmentation externe, car les segments sont de taille variable. C'est pourquoi il faut la combiner avec de la pagination.

Pagination avec segmentation

On va prendre pour exemple ici l'architecture des processeurs Intel 32 bits.

Pour convertir l'adresse logique en adresse physique, il faut d'abord faire passer l'adresse logique à l'**unité de segmentation** qui va extraire le numéro de segment, le type de segment (privé ou partagé), les informations de protection et le décalage.

À partir de ces informations, elle va construire l'**adresse linéaire**, qui est composée du directory (table des pages à utiliser parmi le répertoire des tables de pages), la page et le déplacement.

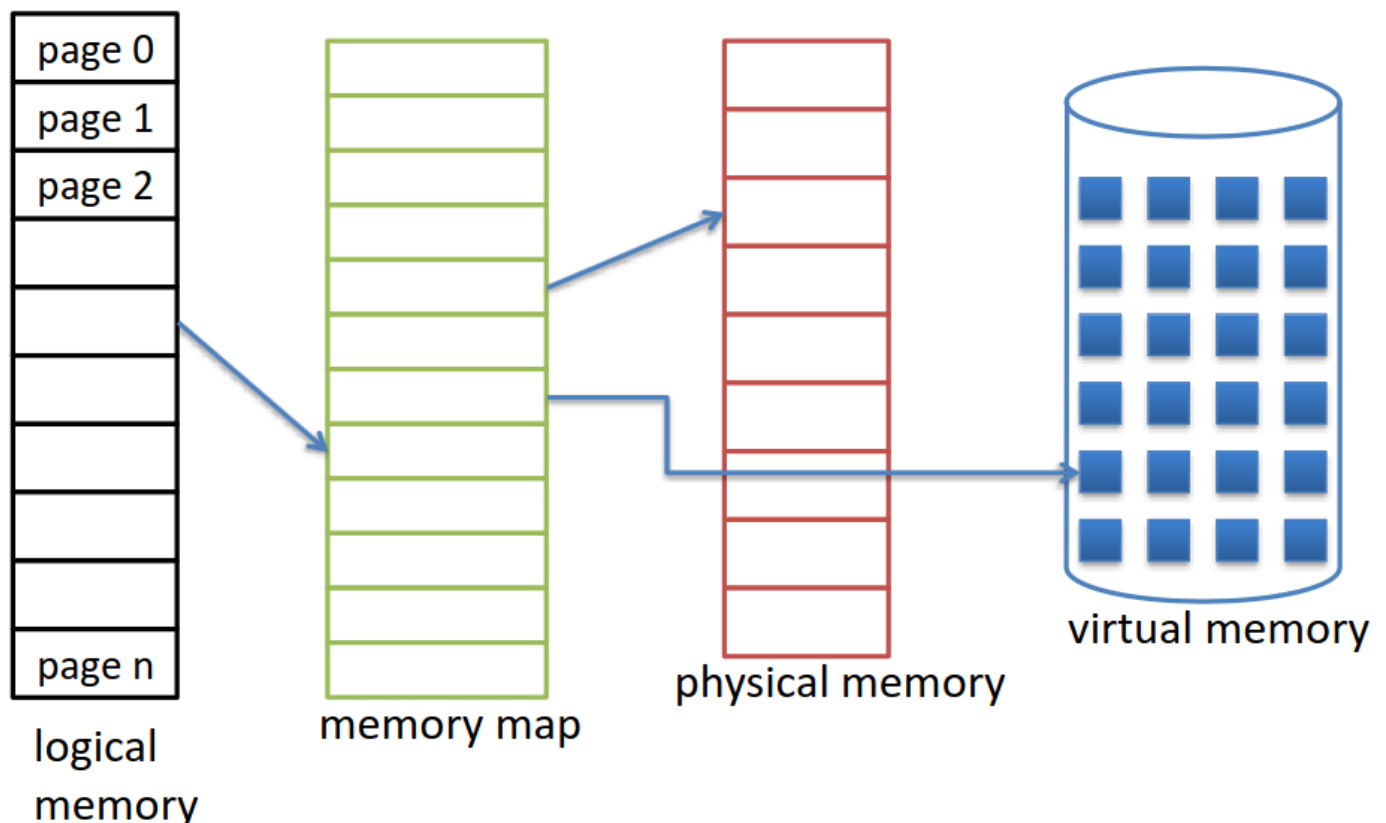
Enfin, l'**unité de pagination** n'a plus qu'à prendre les informations de l'adresse linéaire pour trouver la bonne frame.

Question 4 (mémoire)

Détaillez le fonctionnement de la mémoire virtuelle (y compris les algorithmes de remplacement des pages).

Puis ce qu'il faut qu'un programme se trouve entièrement en mémoire pour s'exécuter, il faut trouver un moyen d'exécuter de très gros programmes (plusieurs Go), ou beaucoup de petits programmes.

Pour cela, on peut utiliser la **mémoire virtuelle** qui est une mémoire sur le disque, qui bien que peu rapide est très abondante.



Pour l'implémenter, on peut utiliser la pagination, dans lequel on va lier des espaces sur le disque à la table des pages, la mémoire virtuelle est donc elle-même divisée en pages. La table des pages est adaptée pour inclure un bit indiquant si la page se trouve sur le disque ou sur la mémoire physique.

Il faut donc également que le disque ait un endroit réservé pour avoir la mémoire virtuelle, cela peut être un fichier ou une partition.

Fonctionnement

Seules les pages nécessaires sont placées en mémoire physique. Le changement (swap) entre mémoire virtuelle et mémoire physique est effectué lorsqu'une page non chargée est demandée.

Lorsqu'une page est demandée, on va voir dans la table des pages si elle est présente en mémoire physique (bit valide), si ce n'est pas le cas alors, il s'agit d'un défaut de page.

Il est important de minimiser les défauts de pages pour ne pas trop impacter le système.

Tout d'abord, on vérifie si la page demandée est bien présente dans l'adressage du programme, si oui, on trouve une frame libre et on récupère la page depuis le disque. Ensuite, on ajuste la table des pages pour indiquer la page comme valide et indiquer sa position en mémoire physique. Enfin, on peut redémarrer le processus à l'instruction qui a causé le défaut de page.

S'il n'y a plus de frames disponible, il faut trouver une page victime qui sera remplacée dans la mémoire virtuelle pour libérer une frame.

Algorithmes de choix des pages victimes

Il est important de choisir un bon algorithme afin de minimiser les défauts de pages.

- **FIFO** (First In, First Out), on sélectionne la page la plus ancienne. La performance est mauvaise, car ce n'est pas parce qu'une page est vieille qu'elle n'est plus utilisée.
- **OPT** (Remplacement Optimal), éliminer les pages qui ne seront plus nécessaires avant longtemps. Cela est cependant seulement une base théorique et est difficile à implémenter puisqu'on ne peut pas savoir quand une page sera utile à nouveau.
- **LRU** (Least Recently Used), éliminer la page qui a été utilisée la moins récemment. Pour cela, il faudrait ajouter une timestamp pour chaque page, ce qui serait assez peu efficace. Sinon, on peut aussi utiliser une pile ou l'un des autres algorithmes dérivés du LRU.
 - **LRU approximé**, chaque page possède un bit de référence (par défaut à 0), dès qu'on accède à la page, on la met à 1, lorsque tout est à 1, on met tout à zéro sauf la dernière. La page choisie en cas de défaut de page est la première page avec un bit 0.
 - **LRU avec octet de référence**, même chose que LRU approximé, mais à intervalles réguliers, on collecte les bits de références pour les mettre dans des octets de

référence. La page victime sera la page avec la valeur d'octet la plus petite.

- **Algorithme de la seconde chance**, on stocke 2 bits par page, un bit de référence (comme LRU approximé) et un bit de modification. Le bit de modification est mis à un si la page est modifiée sur la mémoire physique, mais pas encore sur la mémoire virtuelle. La page victime idéale sera la page avec un bit de modification à 0 et un bit de référence à 0, car pas besoin de E/S.
- **LFU** (Least Frequently Used), éliminer la page utilisée la moins souvent, on ajoute donc un compteur pour chaque page et on sélectionne celui qui a le plus bas. Pas un bon algorithme parce que ce n'est pas parce qu'une page n'a pas encore beaucoup été utilisée qu'elle ne le sera pas.
- **MFU** (Most Frequently Used), inverse du LFU pour palier à la critique précédente. Cependant, cela reste un mauvais algorithme.

Amélioration des performances

Pour améliorer les performances, on peut garder un pool (ensemble) de page qui sont toujours immédiatement disponibles. Ainsi, lorsqu'un défaut de page survient, on alloue une page du pool immédiatement, on sauvegarde la page victime un peu plus tard et on ajoute la page victime dans le pool.

Allocation des frames

L'allocation des frames au processus est quelque chose de complexe, s'il n'y en a pas assez, beaucoup de défauts de pages. S'il y en a trop, plus assez de frames disponibles, donc beaucoup de défauts de pages.

Si on fait simplement le nombre de frames divisé par le nombre de processus (**allocation équitable**), il y aura beaucoup de gaspillage, car des petits processus auront beaucoup de mémoire et les gros pas assez.

Si on fait une **allocation proportionnelle** à la taille des processus, il faut tenir compte que les processus à plus **haute priorité** doivent avoir plus de frames pour s'exécuter plus vite. Il faut aussi tenir compte de la **multiprogrammation**, soit beaucoup de processus = moins de frames par processus.

Si on n'alloue pas assez de frames au processus, le processus va passer beaucoup de temps à transférer des informations. Le **trashing** c'est quand un processus n'a pas suffisamment de frames allouées et passe plus de temps à récupérer ses pages qu'à s'exécuter. Sauf que puisque l'utilisation CPU diminue, le système va augmenter la multiprogrammation, donc augmenter le nombre de processus, ce qui va diminuer encore plus le nombre de frames et donc causer encore plus de défauts de pages.

Pour résoudre le problème de trashing, il faut utiliser le **modèle de la localité**, il faut arriver à identifier les pages qui sont utilisées ensemble (localité), cela assure donc que tant que le programme utilise le même groupe de page, il n'y aura pas de défaut de page.

Question 5 (fichiers)

Expliquez le support des fichiers dans le SE, la structure du système de fichier et évoquez les problèmes d'allocation

Support des fichiers dans le SE

Un fichier est une **collection nommée d'informations en relation**. C'est pour l'utilisateur·ice, un moyen de conserver des informations. Le système doit permettre aux programmes de **créer, lire, écrire, changer la position du curseur et supprimer un fichier**.

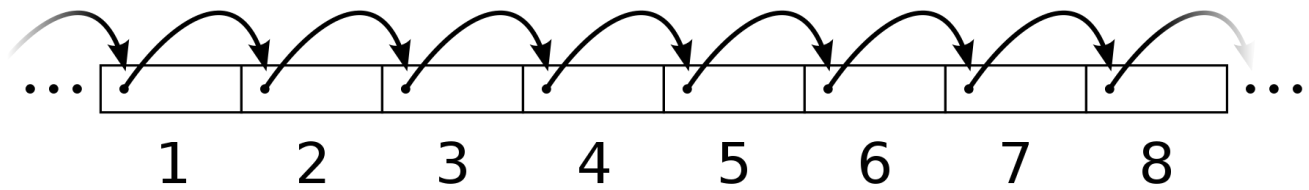
Lorsqu'un processus souhaite accéder à un fichier, il demande une **ouverture** du fichier, le système va alors mémoriser les informations sur le fichier (mode d'ouverture, emplacement du premier bloc, position courante et permissions d'accès), vérifier si l'accès est autorisé, initialiser les structures internes et placer le curseur de position au début.

Une fois le traitement fini, le fichier est **fermé**, le système peut donc supprimer les structures internes. Ce système d'ouverture et de fermeture soulage beaucoup le système d'exploitation.

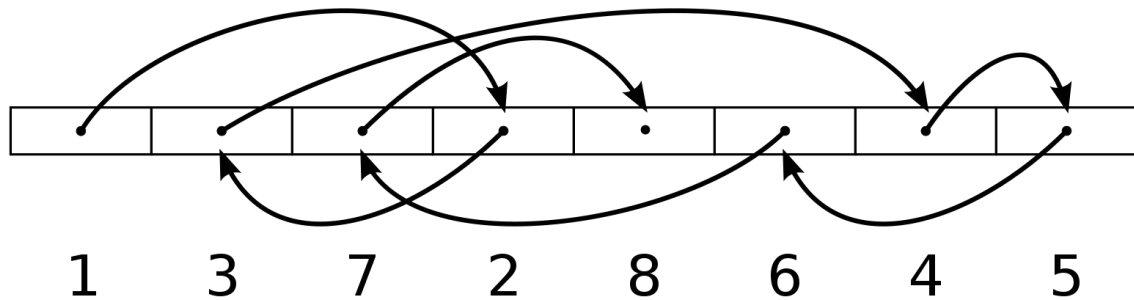
Le **type** du fichier (qui indique sa structure interne) est identifié par l'extension (sous Windows), un "uniform text identifier" (sous macOS), par les valeurs des premiers octets (sous Linux) ou avec des attributs étendus (sous OS/2).

Un fichier peut être accédé via plusieurs méthodes. De manière **séquentielle**, c'est-à-dire que l'on lit tout du début à jusqu'à la position voulue (par exemple sur une bande magnétique), **direct** où on va directement à la position voulue ou **séquentielle indexée** qui permet un accès séquentiel et un accès direct.

Accès séquentiel



Accès direct



Structure du système de fichiers

Structures

Il y a deux structures principales du système de fichiers, la **partition** qui est une partie du disque dur qui permet d'isoler des données du reste.

Le **répertoire** qui contient des fichiers ou d'autres répertoires. Il faut donc que cette structure puisse localiser, créer, supprimer, renommer, visualiser des fichiers et se déplacer dans un autre répertoire.

Le répertoire contenant tout sur le système est le répertoire **racine** (root directory). Un chemin d'accès est dit **absolu** s'il part de la racine et **relatif** s'il part d'un autre dossier.

Pour supprimer un dossier, cela dépend des systèmes, sous Linux les fichiers sont supprimés récursivement.

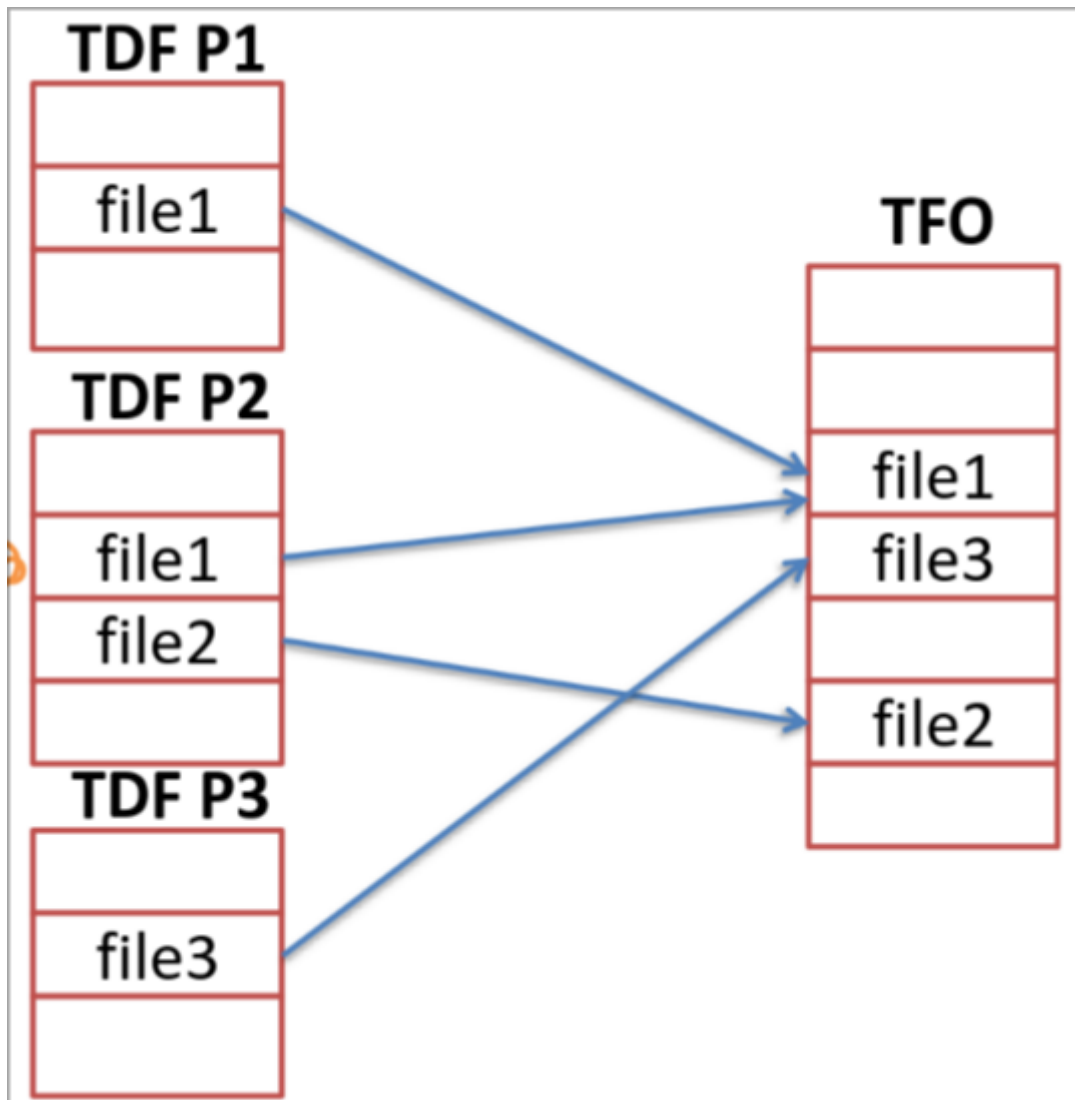
Il est possible de créer des **liens** vers des fichiers à d'autres emplacements, sous Windows, c'est simplement un fichier binaire `.lnk`, sous Linux, c'est un fichier spécial qui pointe un autre fichier. Accéder à un lien sous Linux revient donc exactement à accéder au fichier. Cela permet de faire des économies de place.

Il est aussi possible de **monter** d'autres systèmes de fichiers d'une autre partition ou d'un autre média (dvd, usb, etc), le format peut aussi être différent du format actuel (ntfs, fat, ext4, etc).

Lors d'un montage, le système vérifie la cohérence et donne accès aux informations en le faisant apparaître dans le système de fichier. Le montage peut être **implicite** (automatique) ou **explicite** (demandé manuellement).

Implémentation

Le système retient la liste des partitions montées, les informations sur les répertoires récemment visités, la **table générale des fichiers ouverts** (qui décrit tous les fichiers ouverts sur le système) et la **table de fichiers par processus** (qui décrit les fichiers ouverts pour le processus courant, en incluant des informations supplémentaires par rapport à la TFO tel que le mode d'ouverture ou la position courante).



Pour implémenter les répertoires, le système peut le faire sous forme de liste (le problème est qu'il faut alors un accès séquentiel). Ou alors sous forme de table hashée en plus de la liste (hashmap), cela permet alors un accès direct et séquentiel aux fichiers. Il faut tout de fois faire attention aux **collisions** (deux fichiers pour un seul hash) et à la **grandeur variable de la table**, il faut donc pouvoir la faire grandir.

Problèmes d'allocation des fichiers

Pour allouer l'espace, il faut trouver un moyen de savoir où écrire les fichiers sur le disque.

La première méthode est l'**allocation contiguë**, chaque fichier occupent des blocs qui se suivent sur le disque. Ce qui rend donc la lecture très simple. Cela peut aussi permettre un accès direct aux positions du fichier si on connaît la position et le premier bloc. Le problème est que cela amène de la **fragmentation externe** à cause des libérations successives, il faut alors compacter (**défragmenter**) ce qui est dangereux et chronophage. Également il faut pouvoir faire le fichier, si on y crée un buffer pour le permettre de grandir, alors on se retrouve avec de la **fragmentation externe et interne**.

La deuxième méthode est l'**allocation chaînée**, chaque fichier est une liste de blocs chaînés entre eux. Cela a l'avantage de ne pas avoir de fragmentation externe et de pouvoir faire croître le fichier, car les blocs peuvent se trouver n'importe où. Cela crée toutefois **beaucoup de voyage de la tête de lecture**. Pour résoudre ce problème, certains systèmes allouent des groupes de blocs (cluster), plus tôt que des blocs, ce qui crée donc de la **fragmentation interne**.

La troisième méthode est l'**allocation indexée**, on garde un ou plusieurs bloc(s) "index" pour chaque fichier qui pointe les blocs du fichier. Cela permet un accès direct aux différents blocs, tout en permettant aux blocs de se trouver n'importe où. Cependant l'index peut devenir très grand, le problème de voyage de la tête de lecture est toujours là et maintenant, il va en plus falloir passer sur les blocs d'index.

La méthode d'allocation à choisir dépend de la façon dont le système est utilisé, certains systèmes utilisent plusieurs méthodes. Par exemple allocation contiguë pour les petits fichiers et allocation indexée pour les plus gros.

Pour pouvoir allouer, il faut aussi pouvoir trouver les **blocs marqués libres** pour cela, on peut utiliser une **table de bit** (cependant elle peut grandir beaucoup) ou une **liste chaînée** (qui a aussi l'avantage de pouvoir fusionner les espaces libres).

Le système de fichier peut devenir incohérent, il faut donc que le SE mette en place des systèmes pour vérifier la cohérence et corriger des erreurs (par exemple vérification qu'un bloc ne soit pas deux fois comme libre, deux fois comme occupé, les deux en même temps ou rien du tout).

Pour restaurer les données, on peut aussi utiliser la **sauvegarde** (copie pour but de restauration rapide) et l'**archivage** (copie dans le but de les garder longtemps).

Il y a deux types de **sauvegarde**, la sauvegarde **incrémentale** (on enregistre que ce qui a changé depuis la dernière sauvegarde) ou **différentielle** (on enregistre que ce qui a changé depuis la dernière sauvegarde *complète*). La différentielle prend donc plus d'espace et est plus lente à faire, mais est plus simple à restaurer.

Si un problème survient durant des opérations sur le système de fichier, le système peut utiliser le **journal** (une historique de toutes les opérations en cours) pour défaire des opérations non terminées ou en panne. Ce système est similaire à celui des transactions en base de données et les opérations respectent le principe **ACID** (Atomicité, cohérence, isolation, durabilité).

Question 6 (E/S)

Présentez la gestion des E/S dans le SE (matériel, logiciel ainsi que les différentes couches).

Gestion des E/S au niveau matériel

Au niveau matériel, on distingue les **périphériques d'entrées-sorties** (clavier, souris, écran, etc) des **contrôleurs** (interface entre le SE et le périphérique), permet au SE de contrôler le périphérique et permet d'envoyer des **interruptions** au SE.

Il existe deux types principaux de périphériques d'E/S, les périphériques **blocs**, les données sont adressables et dans des blocs de taille fixes (exemple disque dur), et les périphériques **caractères** où l'information est un flux sans structure (carte réseau, clavier, souris, etc).

Un contrôleur est la partie électronique qui contrôle le matériel, soit sur la carte mère, soit externe (clé USB par exemple).

Le SE commande le périphérique via les **registres du microprocesseur** du contrôleur du périphérique.

Le dialogue entre le SE et le contrôleur être soit fait via les **ports d'E/S** en écrivant ou lisant des données dans les registres (plus d'actualité aujourd'hui). Ou alors en mappant une **zone mémoire mappée aux registres du contrôleur**, cela a donc l'avantage de devoir simplement utiliser les mécanismes de protection de la mémoire et des opérations de base (pas besoin d'instructions assembleur particulières).

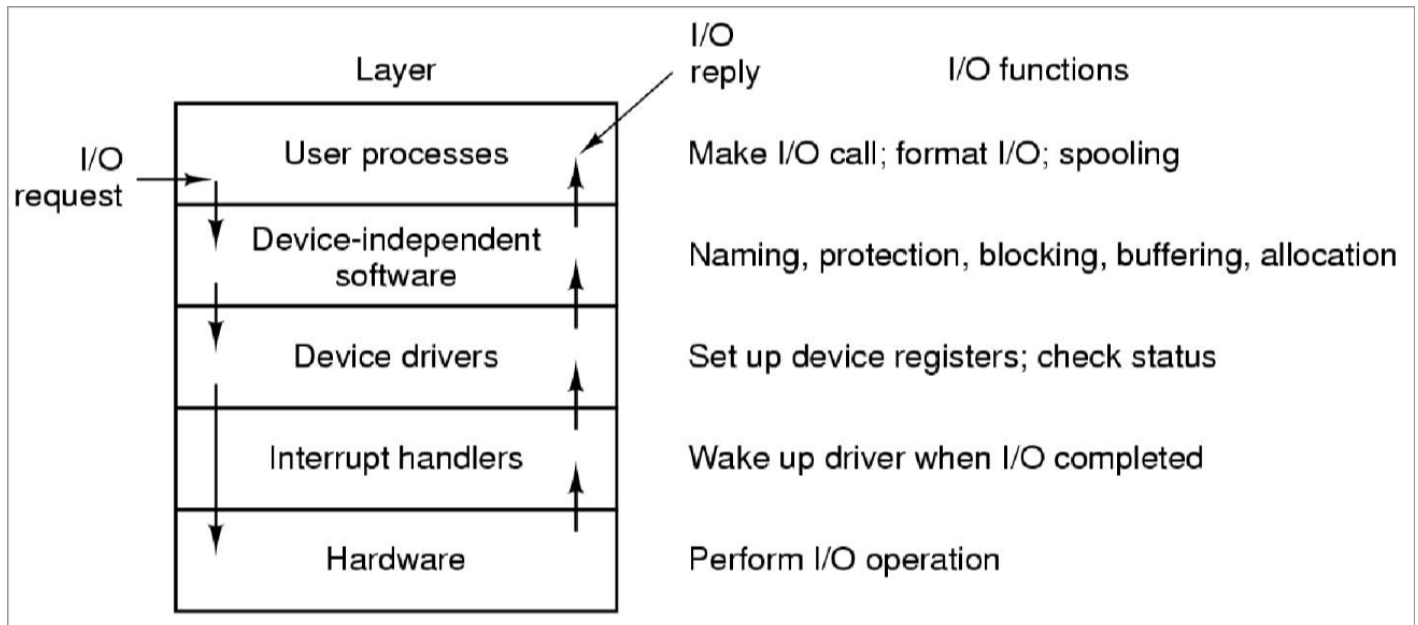
Lors d'une lecture sur le disque, le contrôleur va enregistrer un bloc dans sa mémoire, le vérifier et faire une interruption au SE. Le SE va ensuite exécuter une routine adaptée pour copier l'info en mémoire.

Cela signifie que le SE est constamment sollicité durant le transfert. Pour permettre le soulager le SE, on peut introduire un contrôleur DMA qui va directement copier les informations en mémoire. Ainsi, lorsque l'on veut lire un bloc, on dit au DMA ou stocker les infos en mémoire, et on dit au contrôleur du disque de lire le bloc. Le bloc est ensuite passé au DMA qui va directement mettre les infos en mémoire. Une fois que toutes les infos sont en mémoire, il fait une interruption.

Cependant, lorsque le DMA travaille, il ne peut pas attendre pour écrire les données, car sinon il risque d'en perdre. C'est pourquoi le DMA passe en priorité par rapport au CPU en cas d'accès à la mémoire. C'est ce qu'on appelle un **vol de cycle**.

Gestion des E/S au niveau logiciel

La partie logicielle a pour but de fournir une interface unifiée vers les périphériques, aux programmes. L'accès à une clé USB ou un disque doit donc être vue comme la même chose du point de vue du programme. Cette partie du système est divisée en plusieurs couches.



Le **gestionnaire d'interruption**, dès qu'une interruption survient depuis le périphérique, le gestionnaire d'interruption va sauvegarder le contexte du processus en cours, créer le contexte pour le traitement de l'interruption, se placer en état disponible pour traiter les interruptions suivantes, exécuter la routine (procédure de traitement) adaptée à l'interruption. Une fois toutes les interruptions gérées, le scheduler choisit le processus à démarrer.

Le **gestionnaire de périphérique** (ou pilote ou driver), dépend de la nature du périphérique. Les pilotes doivent être chargés au cœur du système (dans le noyau/kernel en mode protégé) pour pouvoir agir directement sur les périphériques (c'est pourquoi un mauvais driver peut faire crasher le système). Lorsqu'il doit faire une action, le gestionnaire de périphérique traduit les informations données en informations physiques, ensuite, il s'endort jusqu'à être réveillé par le gestionnaire d'interruption.

La **couche logicielle indépendante** (ou interface standard) sert à uniformiser l'accès aux périphériques. Il possède donc une API qui doit être respectée pour tous les pilotes pour pouvoir accéder aux pilotes de la même manière partout. Cette couche va également uniformiser les noms des périphériques, mettre en place des buffers (tampons) pour améliorer les performances, gérer les erreurs (de programmation ou entrée-sortie, si c'est le deuxième cas, c'est le pilote qui réessaye).

C'est aussi cette couche qui s'occupe de l'allocation et la libération des périphériques non partageables (genre une imprimante). Lors d'une ouverture, cette couche vérifie la disponibilité.

La **couche d'entrée-sortie applicative** gère les entrées sorties au niveau de l'application, par exemple via des librairies systèmes liées aux programmes (genre `stdio`) et fournir un système de **spooling** qui est une file d'attente pour l'accès aux périphériques non partageables (exemple, liste de jobs d'impression).

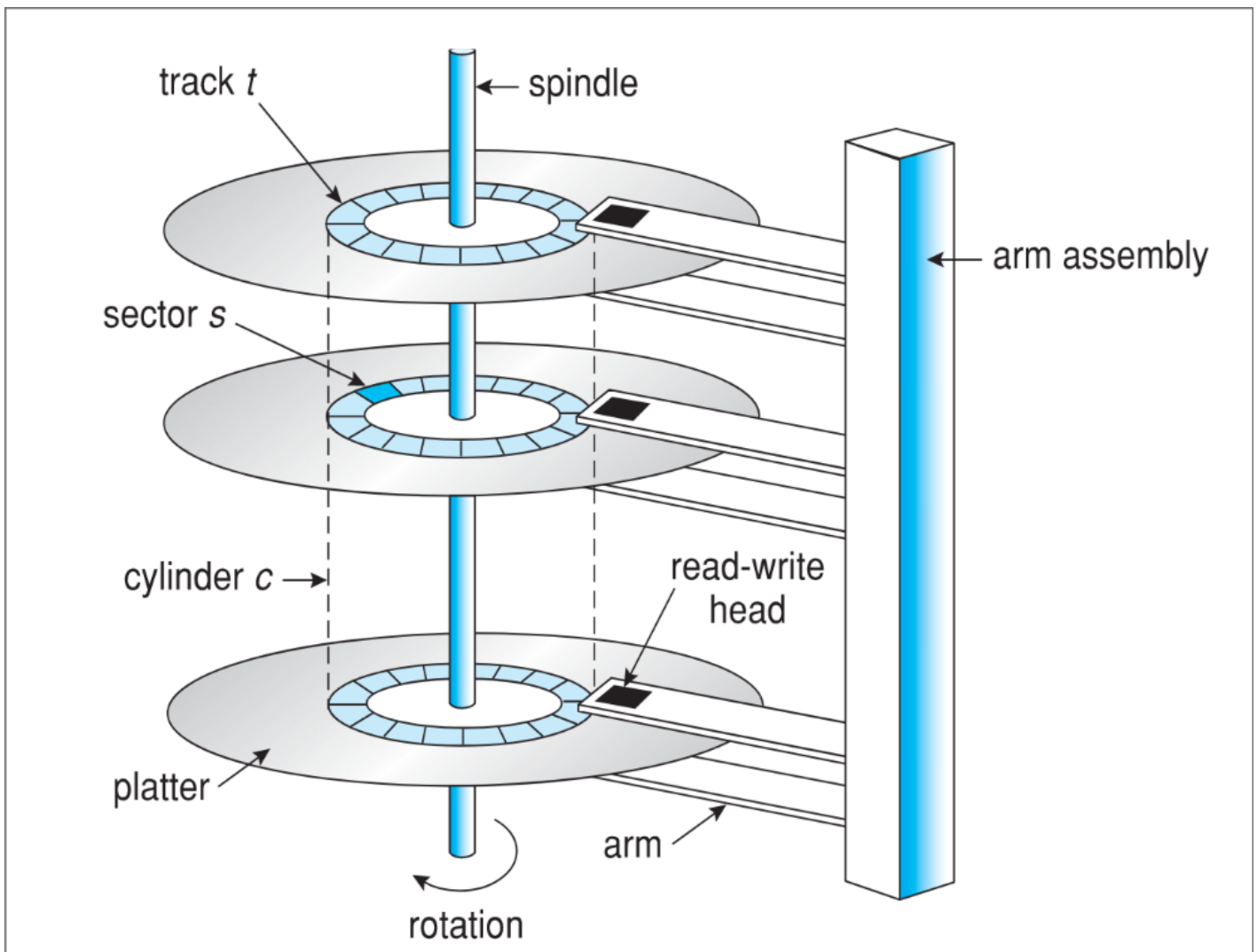
Question 7 (E/S)

Illustrez la gestion des E/S dans le SE en détaillant le fonctionnement du disque et de l'horloge

Disque

Un **disque magnétique** est composé de plusieurs disques physiques (**plateaux**), le disque est découpé en cylindres, pistes et secteurs.

- Un **cylindre** correspond à l'ensemble des pistes qui correspondent à une certaine position de la tête de lecture.
- Une **piste** correspond à un cercle sur le plateau qui correspond à une certaine position de la tête de lecture. Chaque piste est elle-même découpée en secteurs.
- Les **secteurs** sont des blocs de tailles fixes qui composent chaque piste.



La géométrie du disque dur n'est pas forcément celle annoncée, car le nombre de secteurs par piste n'est pas constant. La géométrie du disque du point de vue du SE est donc une simplification.

Afin de pouvoir être utilisé, un disque doit être formaté (**formatage de bas niveau** par le fabricant) qui consiste à écrire la géométrie du disque (chaque secteur contient un préambule concernant le numéro du secteur, du cylindre, etc) avec des données telles qu'un code ECC pour la gestion d'erreur.

Ensuite, on peut mettre en place des partitions qui seront vues comme des espaces séparés par les systèmes d'exploitations.

Il peut y avoir beaucoup d'erreurs différentes sur un disque, notamment des erreurs qui peuvent survenir lors de la construction ou du formatage de bas niveau. Il faut donc que le système d'exploitation stocke les blocs défectueux afin de ne pas les utiliser par la suite.

Le **disk arm scheduling** est une opération faite par le SE pour minimiser le **seek time** (déplacement de la tête de lecture) lorsqu'il faut traiter plusieurs requêtes. Il y a donc plusieurs algorithmes différents :

- Le **FCFS**, premier arrivé, premier servi. Simple, mais pas efficace.

- Le **SSTF**, toujours servir les requêtes les plus proches. Fait diminuer le temps de réponse, mais nécessite de calculer les positions tout le temps et risque de créer de la famine parmi les requêtes les plus éloignées.
- Le **SCAN**, le bras de lecture va toujours dans la même direction jusqu'à arriver à la fin du disque et retourner dans le sens inverse. Plus de famine, car on avance toujours.
- Le **C-SCAN**, comme le SCAN sauf qu'à la place de revenir dans l'autre sens, il va directement à l'autre extrémité et recommence
- Le **LOOK**, comme le SCAN sauf qu'à la place d'aller à l'autre bout du disque, il s'arrête à la dernière requête.
- Le **C-LOOK**, comme le C-SCAN, mais s'arrête à la dernière requête.

Le C-SCAN et C-LOOK sont intéressants pour les disques très chargés. SSTF est un bon remplacement à FCFS. Et c'est souvent un SSTF ou une variante du LOOK que l'on va trouver sur les systèmes.

Aujourd'hui le scheduling est aussi réalisé au niveau des contrôleurs en plus du SE.

RAID

RAID est une technologie qui consiste à combiner plusieurs disques afin d'améliorer les performances et/ou la fiabilité. Elle fonctionne à l'aide du **mirroring** et du **stripping**.

Le **mirroring** consiste à dupliquer les informations d'un disque sur un autre. Lors d'un accès en écriture, on effectue la modification sur les deux disques en même temps.

Le **stripping** consiste à répartir les informations sur plusieurs disques. Cela peut permettre de lire huit fois plus vite, car on peut lire sur tous les disques en même temps.

Les différents RAIDs sont répartis en niveaux,

Le **RAID 0** fait simplement du stripping, cela permet d'avoir un grand espace en combinant les deux disques.

Le **RAID 1** fait uniquement du mirroring, cela permet d'avoir une plus grande fiabilité en dupliquant les données.

Le **RAID 3** fait du stripping au niveau des octets et garde un bit de parité sur un disque séparé. Si on perd un disque, on peut ainsi reconstruire l'information à la volée avec le bit de parité.

Le **RAID 4** fait du stripping au niveau des blocs, les blocs de partiés sont gardés sur un disque séparé. Contrairement au RAID 3, une lecture de bloc ne nécessite l'accès qu'à un seul disque, ce qui est donc plus rapide. On ne peut cependant pas écrire en parallèle, car tous les bits de parités sont sur le même disque (pas d'accès concurrent au disque de parité).

Le **RAID 5** est une amélioration du RAID 4 qui consiste à distribuer les bits de parités sur tous les disques afin de permettre un accès concurrent en écriture. Généralement, c'est ce modèle qui est

choisi.

Le **RAID 6** est une amélioration du RAID 5 qui ajoute un bit de parité dans le but de protéger contre la perte de deux disques dur. Ce modèle est cependant plus complexe et nécessite que le contrôleur RAID ait un CPU plus important.

Le **RAID 0+1** consiste à utiliser deux RAID 0 (stripping) qui sont dupliqués par un RAID 1 (mirroring). Cela est cependant assez cher.

RAID permet aussi des options particulières comme le **hot-swap**, qui consiste à permettre d'enlever des disques pendant que le système tourne, ainsi que l'option **spare** qui consiste à avoir un disque présent inutilisé, mais prêt à l'emploi en cas de perte d'un disque.

Pour faire un RAID réussi, il faut pouvoir maximiser le débit de lecture et écriture ainsi que de favoriser une diversité parmi les disques dur afin qu'ils ne tombent pas tous en panne en même temps.

Horloge

L'horloge est un périphérique spécial qui sert simplement à déterminer la date et l'heure du système. Cette fonction est essentielle pour maintenir des statistiques, calculer le quantum de temps des processus, etc. Certains systèmes ne disposent pas d'horloge et synchronisent leur temps via le réseau ou en le demandant à l'utilisateur·ice au lancement du système.

L'horloge fonctionne avec un quartz, un compteur et un registre. Le quartz lorsqu'il est sous tension génère un signal périodique très précis qui peut être utilisé pour la synchronisation des composants. Le signal est ensuite fourni au compteur qui va décompter jusqu'à arriver à zéro. Une fois à zéro une interruption est émise vers le CPU et le gestionnaire l'horloge prend la main.

Ensuite, ici, il y a deux modes de fonctionnement. Soit le mode **one-shot**, une interruption est envoyée et on attend une réaction. Ou alors le mode **square wave** qui, une fois que le compteur arrive à zéro, recommence. On parle ici de **ticks d'horloge**.

Le système va ensuite maintenir la date et l'heure en comptant le nombre de secondes depuis une référence (exemple, sous UNIX 1er Janvier 1970 00:00:00 UTC). Pour synchroniser l'heure, il suffit donc de savoir le nombre de secondes depuis un point de référence.

Ensuite, le logiciel d'horloge va maintenir l'heure, vérifier qu'un processus ne dépasse pas son quantum, compter l'utilisation CPU, gérer l'appel système `alarm`, etc.

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 jusqu'à arriver à zéro ou il est alors remplacé.

Pour pouvoir cependant gérer plusieurs évènements temporels sans pour autant avoir plusieurs horloges dans le système, le système va créer une **file d'évènement** qui agit comme une sorte d'agenda pour planifier des évènements.

Question 8 (sécurité)

Définissez la protection (y compris la matrice d'accès). Présentez la sécurité et les mécanismes pour l'assurer à l'intérieur du SE.

Définition de la protection

La **protection** est un mécanisme mis en place pour l'accès à la gestion de ressources du système par les processus. La **sécurité** est 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). Une **politique de protection** définit ce qui doit être protégé et les **mécanismes de protection** sont les moyens de protéger.

Il faut pouvoir protéger les processus et les **objets**, les objets peuvent être ressources matérielles (CPU, mémoire, etc) ou logicielles (fichiers, sémaphores, programmes, etc).

Il faut s'assurer que chaque processus ne peut accéder qu'aux ressources auxquelles il a l'autorisation.

Un **domaine de protection** définit l'ensemble d'objet et d'opération qui peuvent être exécutés sur ceux-ci. La possibilité d'exécuter quelque chose s'appelle un **droit d'accès**, un domaine est une collection de droits d'accès et chaque processus opère à l'intérieur d'un domaine de protection.

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

La matrice d'accès

La matrice d'accès est une représentation des différents domaines, objets et droits d'accès. Chaque ligne correspond à un domaine et chaque colonne représente un objet. Chaque case représente les droits d'accès (lire, écrire, exécuter, etc) pour un domaine par rapport à un objet précis.

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

Pour pouvoir changer de domaine, il suffit de représenter les domaines comme des colonnes également et d'utiliser le mot **switch** dans la colonne du domaine vers lequel on peut aller.

object domain	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	switch control
D_3		read	execute					
D_4	write		write		switch			

Une autre permission spéciale est celle de pouvoir copier son droit à un autre domaine. Ce droit est représenté par une *. Il y a aussi la permission de **transfert** qui est comme celui de la **copie** sauf que lorsqu'un transfert se fait, le domaine perd le droit d'accès.

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

Il y a aussi le droit à la modification des droits d'accès d'un objet, c'est le droit **owner** qui permet de modifier n'importe quel droit pour un objet donné.

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

Enfin, il y a le droit de contrôle d'un domaine qui permet de supprimer des droits d'accès à un autre domaine. Ce droit est le droit **control**.

object domain	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	switch control
D_3		read	execute					
D_4	write		write		switch			

Sous UNIX, chaque domaine correspond à un utilisateur·ice, lorsqu'un exécutable est lancé, le processus prend le domaine de l'utilisateur·ice courant·e. Excepté lorsque le bit `setuid` est mis, dans quel cas le processus est lancé dans le domaine du propriétaire de l'exécutable.

Utiliser une matrice d'accès comme implémentation de la protection ne serait pas très pratique, car prendrait beaucoup de place. C'est pourquoi on utilise des **access list** à la place, pour chaque objet, elle contient des paires de domaines et de droits (si un domaine n'est pas lié à un certain objet, alors il n'a pas de droit sur l'objet).

Le système doit aussi prévoir comment révoquer des droits, si c'est immédiat ou décalé, pour quels utilisateur·ice·s, temporaire ou permanent, etc.

Sécurité et mécanismes dans le SE

Les mécanismes de sécurité consistent à protéger le système contre le vol d'information, la modification ou destruction non autorisée d'information et la surveillance de l'utilisation du

système. Le but est de ralentir les violations du système le plus possible pour rendre le coût de pénétrer un système plus important que le gain à en retirer.

Il y a plusieurs niveaux de protection différents, le niveau **physique** (protection du matériel), **humain** (surveillance de l'accès au matériel), **réseau** (via des firewalls), et **système d'exploitation**.

Des informations volées ou perdues peuvent conduire à la faillite d'une entreprise.

Identifications des utilisateur·ice·s

Les systèmes actuels ne fonctionnent que lorsque les utilisateur·ice·s sont authentifié·e·s. Il y a différentes méthodes d'identifier les personnes,

Identification par mot de passe

L'identification par **mot de passe** qui est l'une des plus courantes, mais qui a beaucoup de problèmes. Les mots de passes peuvent être découverts par espionnage (keylogger, network sniffing), force brute (dictionnaire, rainbow table, brute force), ou encore par déduction (le nom du chien, date de naissance, etc) ou encore par arnaque (fishing). Beaucoup de vulnérabilités des mots de passes sont dans le fait que beaucoup de gens utilisent des mots de passes faibles et les réutilisent partout.

Les mots de passes peuvent être stockés de plusieurs manières, mais seules certaines sont sûres,

- Si on stocke les mots de passes **en clair**, cela signifie que quiconque a accès à la base de données a accès à la vie en ligne de presque tous les utilisateur·ice·s
- Si on **chiffre** les mots de passes, cela pose un problème si quelqu'un découvre la clé, ou si plusieurs personnes utilisent le même mot de passe, on pourra également le voir, car les mots de passes chiffrés seront les mêmes.
- Si on les **hash** (faire passer les mots de passe dans une fonction à sens unique pour créer une "empreinte" des mots de passes), plusieurs personnes avec le même mot de passe auront le même hash. Par ailleurs, cette méthode est vulnérable au **rainbow table attack** qui consiste à utiliser des tables de hash pré calculées pour ne pas avoir à calculer les hash un à un, ce qui rend l'attaque plus rapide qu'une attaque par dictionnaire.
- Si on les **hash avec salt** qui consiste à utiliser la méthode précédente, mais en ajoutant une chaîne de caractère aléatoire au mot de passe (le salt est ensuite simplement mis à côté du mot de passe). Cela règle tous les problèmes cités précédemment.

La méthode de stockage des mots de passes recommandée est le hash avec salt en utilisant une fonction de hashage lente pour ralentir le plus possible les attaques.

Ne pas stocker les mots de passes et le gérer l'identification par un tiers de confiance peut être une bonne idée si le stockage des mots de passes est une trop lourde responsabilité.

Identification a plusieurs facteurs

L'identification à plusieurs facteurs consiste à utiliser d'autres systèmes en conjonction avec les mots de passe afin de sécuriser le mécanisme. On peut par exemple utiliser des codes à usages uniques (tel que le TOTP) qui peuvent être générés sur base de **chaînes de hash** (hashage n fois d'une valeur de départ, à chaque utilisation, on fait un hash en moins = codes à usage unique) ou sur base du **temps** (hashage du temps actuel + d'un token partagé avec le serveur = code à usage unique). L'identification peut aussi être faite via des appareils externes tels qu'une clé FIDO2.

Ce mécanisme (TOTP) est par exemple utilisé pour des digipass de banques.

Identification password-less (+ biométrie)

On peut aussi utiliser des mécanismes vu précédemment, tel que celui de clé FIDO2 qui peut être utilisé pour SSH, permettant de se connecter sans utiliser de mot de passe.

Sinon, on peut aussi utiliser l'**identification biométrique** qui se fait en scannant des propriétés de l'utilisateur·ice·s tel qu'une empreinte digitale ou la forme du visage. Ces informations scannées deviennent ensuite une sorte de mot de passe.

Le problème avec cela cependant est que ces données sont très personnelles et confidentielles, le stockage est donc primordial, car on ne peut pas changer notre empreinte digitale.

Sécurité des applications

Écrire un code sans erreurs 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.

Comme attaques courantes, on trouve :

- La **Remote Code Execution** (RCE), l'exécution de code à distance en soumettant des données précises à l'application (exemple log4shell)
- La **SQL Injection**, injection de code SQL dans la base de données en injectant des données précises.
- Les **Format String vulnerabilities** qui consiste à soumettre du code qui sera compris comme des commandes par l'application (par exemple via des `%s` ou `%d`).
- **Cross-Site Scripting** (XSS), ajout de code HTML en soumettant des données incorrectes (et donc potentiellement Javascript), exécuté par tous les visiteurs de la page.
- **Username enumeration** consiste à tester plusieurs noms d'utilisateur·ice·s pour savoir lesquels sont présents sur le système.
- **Stack/Buffer Overflow**, soumet des caractères spécifiques de façon à faire déborder la pile du programme, le code exécuté devient alors celui de l'attaquant qui peut prendre le contrôle de l'application.

- **Déni de Service** (DoS), rendre inaccessible un système en le bombardant de requêtes.
 - **Déni de Service Distribué** (DDoS), un DoS exécuté depuis pleins de machines différentes (machines **zombies**) qui font partie d'un ensemble de machine appelé **botnet**.

Comme programmes malveillants, on trouve :

- Le **cheval de troie** (trojan) qui est un programme qui se fait passer pour un autre afin de déclencher des actions hostiles (exemple, faux anti-virus)
- La **porte dérobée** (backdoor), le programme installe une porte d'accès à l'utilisateur qui permet d'exécuter des commandes à distance (ce qui peut transformer la machine en **zombie** faisant partie d'un **botnet**)
- Les **vers informatiques** (WORMS) se propagent par le réseau via des vulnérabilités des machines qui en font partie.
- Les **virus informatiques** sont des ensembles d'instructions qui utilisent un programme pour se reproduire. Utilisent des techniques pour se cacher, ont pour principal but de se reproduire et peuvent avoir une charge active qui attaque le système à un moment déterminé.
 - Les **virus script** sont des virus écrit dans un langage interprété qui utilise un environnement tiers (ex Microsoft Word) pour s'exécuter (par exemple via une macro).

Pour se protéger contre les attaques, il est important de protéger le **périmètre** (tout ce qui est autour des machines, par exemple le réseau), en installant un firewall réseau. Ainsi que de protéger les **machines** en installant un firewall sur les machines et un anti-virus. Il faut également faire attention à ne jamais travailler en compte administrateur.

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

Ou encore une sauvegarde des **empreintes** de tous les programmes dans le but de vérifier celles-ci lors du lancement pour vérifier que des programmes n'aient pas de virus. Il faut cependant faire attention à les garder à jour.

Utiliser un **système de détection d'intrusion** (IDS) qui va repérer les activités anormales sur un système en se basant sur des plans d'attaques connus, comportements anormaux, en observant l'activité et les journaux systèmes (par exemple `fail2ban` qui banni les IP qui font des tentatives trop fréquentes)

Enfin, on peut utiliser des systèmes **d'analyse de fichiers journaux** tels que Splunk ou Grafana Loki.

Question 9 (sécurité)

Décrivez le fonctionnement de la cryptographie (historique et domestique) et le fonctionnement de la signature numérique.

Fonctionnement de la cryptographie

La cryptographie consiste à cacher des informations en utilisant des algorithmes. Le texte non chiffré est appelé **texte en clair**, le texte chiffré est appelé **cryptogramme**.

Le niveau de chiffrement utilisé dépend des personnes contre lesquels on désire se protéger. On ne va pas utiliser le même niveau de chiffrement si on souhaite se protéger contre la NSA ou contre ses concurrents.

C'est aussi une question de longueur de clé, plus la clé est longue, plus ce sera sécurisé.

L'algorithme qu'on utilise est également déterminant, un bon algorithme doit être public (vérifiable), sûr (éprouvé et approuvé) et indépendant (pas de lien avec des groupes tel que la NSA).

Un algorithme est sûr s'il est impossible qu'un espion ayant un texte chiffré, un texte chiffré et un texte en clair ou une infinité de textes chiffrés et en clair ne puisse déterminer la clé de l'algorithme ou déchiffrer les éléments.

Histoire de la cryptographie

La cryptographie existe depuis des siècles, mais était avant pratiquée par des militaires, des diplomates et des amants. Avant l'informatique, la cryptographie était limitée aux capacités du cerveau humain.

Deux méthodes catégories d'anciens chiffrements (datés et qui ne doivent plus être utilisés aujourd'hui) sont la **substitution** (remplacement de symboles par d'autres, comme le code de César) et la **transposition** (mélange de symboles).

On peut aussi utiliser l'opération XOR comme base, car c'est une opération très simple à effectuer pour le CPU. On fait donc texte en clair XOR clé pour obtenir le cryptogramme, puis on fait cryptogramme XOR clé pour obtenir le texte en clair.

Un premier problème est que l'on peut retrouver la clé en faisant cryptogramme XOR texte en clair et donc déchiffrer toutes les autres communications.

Un autre problème est que si la clé se répète, on peut arriver à identifier la clé en faisant de l'analyse de fréquence.

Si la clé en revanche est complètement aléatoire et de la même longueur que le message, il s'agit alors d'un **masque jetable** qui est théoriquement un code incassable.

Le **masque jetable** est une technique de chiffrement se basant sur une longue liste non répétitive et aléatoire de lettres, chaque lettre est utilisée pour coder une lettre du texte clair. On peut par exemple additionner le rang de la lettre du masque avec celui de la lettre du texte clair. Cette technique est donc théoriquement incassable, mais peu pratique, car les clés sont très longues.

Cryptographie à clé secrète (symétrique)

Dans les crypto systèmes à clé secrète, une clé sert à la fois à déchiffrer et à chiffrer (on dit qu'elle est symétrique). Les systèmes historiques sont dans cette catégorie, mais il y en a aujourd'hui des versions plus sécurisées telles que AES (Advanced Encryption Standard).

Le problème avec ce crypto système est qu'il faut garder en mémoire énormément de clés différentes par paire d'utilisateur, ce qui fait beaucoup de clés. De plus, ces clés doivent être distribuées et gardées de manière sûre, car si elle est compromise tous les messages sont compromis.

Cryptographie à clé publique/privée (asymétrique)

À la place d'avoir une clé, on va avoir deux clés, une clé pour chiffrer et une clé pour déchiffrer. Ainsi, la clé pour chiffrer peut-être distribuée publiquement tandis que la clé pour déchiffrer reste secrète.

Cela fait qu'il n'y a maintenant plus besoin que d'une seule clé pour qu'un nombre infini d'utilisateurs puissent chiffrer des messages.

Également, on peut partager les clés facilement et sans risque puis ce qu'elles sont publiques.

Pour pouvoir assurer qu'une clé appartient bien à quelqu'un cependant, on peut utiliser des **tiers de confiance** tel que Let's Encrypt qui vont certifier les clés.

RSA

La sécurité de RSA se base sur la **factorisation de grands nombres** composés de grands nombres premiers, car la factorisation est un problème qui devient toujours plus complexe quand les nombres grandissent.

Et pour avoir un processus qui fonctionne dans un sens, mais pas dans l'autre sans avoir une information supplémentaire, RSA utilise l'**exponentiation modulaire**, c'est-à-dire que si on a $m^e \bmod n = c$, et que l'on connaît, n , e , et c , il est très complexe de trouver m sauf si on a un

autre exposant qui permettrait de renverser l'opération. On peut ensuite utiliser le **théorème d'Euler** pour trouver des exposants e (pour chiffrer) et d (pour déchiffrer).

1. Trouver deux nombres premiers (par exemple, $p = 11$ et $q = 17$)
2. Multiplier les deux nombres pour donner n (soit ici, $n = p * q = 187$)
3. Trouver l'indicateur d'Euler (fonction ϕ) en faisant $\phi(n) = (p - 1) * (q - 1) = 160$
4. Trouver un entier e qui soit premier avec la fonction ϕ (indicateur d'Euler), par exemple 13
5. Trouver d dans $ed \bmod \phi(n) = 1$, soit $ed + k \phi(n) = 1$. En appliquant le théorème d'Euclide étendu ou celui de Bachet-Bézout (e et $\phi(n)$ étant premiers entre eux)
6. Clé publique : $\{e, n\}$, clé privée $\{d, n\}$

Pour chiffrer (ou vérifier une signature) on peut alors faire $m^e \bmod n = c$ et pour déchiffrer (ou signer) on peut faire $c^d \bmod n = m$. À savoir que m doit être plus petit que n .

Signatures cryptographiques

Une signature permet d'identifier que quelqu'un a bien écrit quelque chose, une signature doit être authentique, non falsifiable, non réutilisable. Un document signé ne peut pas être modifié et une signature ne peut pas être reniée. Le but des signatures cryptographiques est d'avoir toutes ces propriétés.

Avec un **crypto système à clé secrète**, il faut avoir un **tiers de confiance** qui connaît deux clés secrètes (une pour le signataire, et une pour le destinataire) permettant d'assurer que le document est authentique. Toute la sécurité est donc dans le tiers de confiance.

Avec un **crypto système à clé publique**, le tiers de confiance est facultatif (il peut toutefois être utilisé pour valider une clé). Le signataire signe ("décrypte un message en clair") avec sa clé privée, ensuite le destinataire peut vérifier ("chiffrer la signature" en quelque sorte) pour retrouver le message.

Les systèmes de confiances tels que les certificats TLS utilisés pour le HTTPS (certificats X509) lient une clé publique à une identité en utilisant une signature cryptographique d'un tiers de confiance. Les navigateurs ont ainsi une liste de tiers de confiance qui leur permet de vérifier les certificats.

Certains tiers de confiances sont gratuits (Let's Encrypt) cependant ils n'offrent pas le même niveau d'assurance et de confiance que des tiers de confiances payant (qui vont aller manuellement faire la vérification).

La confiance dans le système **PGP** (Pretty Good Privacy) en revanche fonctionne très différemment. L'identité des personnes est garantie de manière transitive ; Alice peut signer la clé de Bob pour annoncer à tous ses amis que Bob est de confiance. Les amis d'Alice pourront ainsi vérifier cela en vérifiant la signature d'Alice.

