**Chapitre 1 :Gestion des Processus**

**Un processus est un programme en exécution. Il définit un objet dynamique tandis**

**que le programme est un objet statique.**

**Un thread ou encore processus léger (lightweight process) est une unité d’exécution de code. Il est issu d’un processus mais ne contenant que la pile d’exécution.**

**Les processus sont organisés sous forme d’une arborescence ou chaque processus a un seul père et peut avoir plusieurs fils.**

**Un processus est identifié par un PID (Process IDentifier) et un PPID (Parent Process IDentifier).**

**Modèle des processus: États**

**Lorsqu’un processus s’exécute; il change d’état. Il peut se trouver dans l’un des trois états principaux suivants: Prêt ,** **Élu,** **Bloqué**

**Attributs d’un PCB:** **(Process Control Bloc: PCB).**

**PID et PPID, État, Priorité, Compteur ordinal, Fichiers ouverts, Pointeurs: seg. code, seg. données, seg.**

**Les primitives de gestion de processus:**

**Création d'un processus int fork()**

**(fork () = n ) => si n = 0 c est le processus père**

**sin n >0 fils**

**Si n <0 erreur de création de fils**

**Identification des processus int pid = getpid(); int ppid = getppid()**

**Identification de propriétaires int pid = getuid(); int ppid = getgid()**

**Mise en sommeil d’un processus int sleep(int n);**

**Terminaison d’un processus void exit(int statut)**

**Un bon algorithme d’ordonnancement doit être capable de: équité. Efficacité. Temps de réponse**

**Deux types d’algorithmes d’ordonnancement:**

**1)Ordonnancement sans réquisition (sans préemption)**

**2)Ordonnancement avec réquisition**

**Quand invoquer l’ordonnanceur : Chaque fois que le processus exécutant est interrompu**

**Les algorithmes d’ordonnancement:**

**FCFS: First Come First Served (Gestion des processus à l'aide d'un FIFO, Ordonnancement non-préemptif, Facile à implémenter, Sous-optimal)**

Temps traitement moyen = temps fin d'execution – temps Arrivé = (24-0)+ (27-0)+ (30-0) /3 = 27ms

Temps Attente moyen = (0-0)+ (24-0)+ (27-0) / 3 = 17ms

**Plus Court d’Abord (Short Job First SJF):**

**SJF sans réquisition** ayant le plus petit temps d’exécution**.**

**SJF avec réquisition** ayant le plus petit temps d’exécution restant.

**Algorithmes Ordonnancement**

* **Tourniquet (Round Robin, l’algorithme circulaire)**
* **Il s'agit d'un FCFS avec l'introduction d’un Quantum Q**
* **Ordonnancement préemptif**
* **Très utilisé**

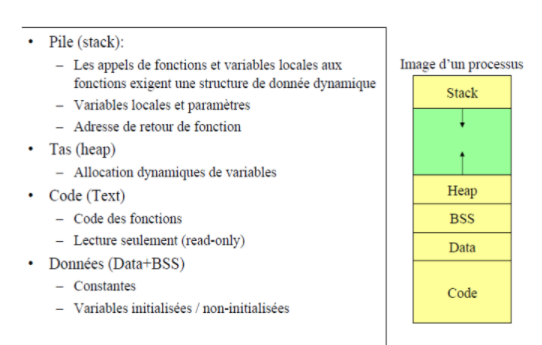
**Ordonnancement avec priorité**

**Priorité dynamique**

**Ordonnancement : non-préemptif ou préemptif**

**Chapitre 2 : Communication Interprocessus**

**Processus en mémoire**

****

**PID = visibles par les commandes top et ps**

**pid = 1 c est le processus père qui a lancé par le boot**

**Orphelin son père est mort et il est récupérer par l init**

**Etat de processus**

**Une image contenant texte

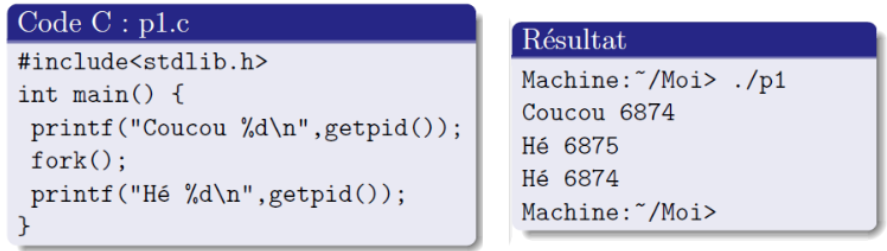
Description générée automatiquement**

**Une image contenant table

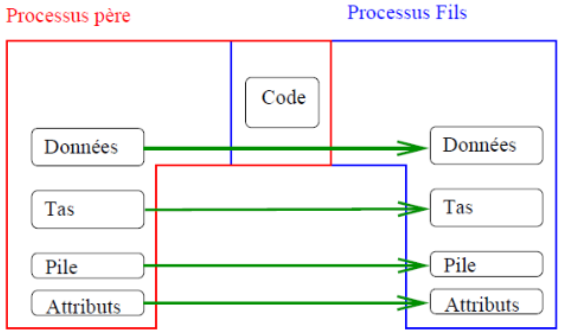
Description générée automatiquement**

**Création de processus**

**Fork() création du processus fils recopie totale (données + attributs) du processus père vers le processus fils (nouveau pid)**

****

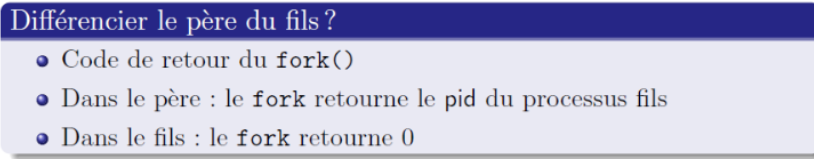
**Création de processus Clonage**

****

**Une image contenant texte

Description générée automatiquement**

**Le fils n’hérite pas l’ID du père les instructions initialisé à 0 les verrou de son père , priorité , des signaux pendant du père .**

****

**Indéterminisme du déroulement**

**Une image contenant texte

Description générée automatiquement**

**Appels système wait et exit :**

**Ces deux primitives permettent l’e=élimination de l’état zombie et permettent la synchronisation entre père et fils**

**Une image contenant texte

Description générée automatiquement**

**Recouvrement d’un processus : code et donnée remplacé , pointeur des instructions réinitialisés**

**Une image contenant texte

Description générée automatiquement**

**Création puis Recouvrement et synchronisation**

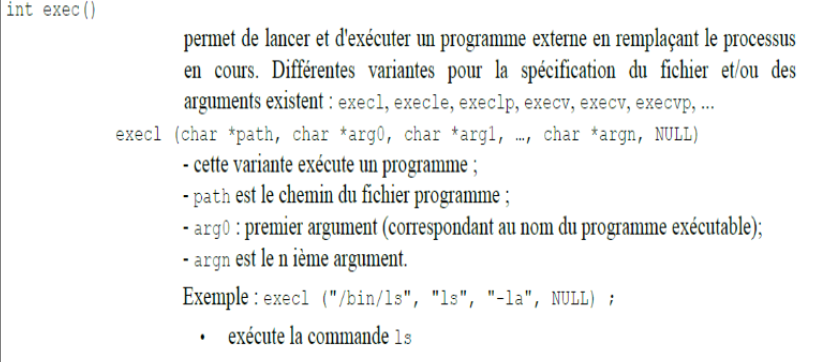
**Les paramètres : argc et argv**

**Argc : tableau de pointeur chaque pointeur pointent sur des chaines de caractères**

**Argv : nombre de chaines de caractères sur lesquelles pointe argc**

**Une image contenant table

Description générée automatiquement**

****

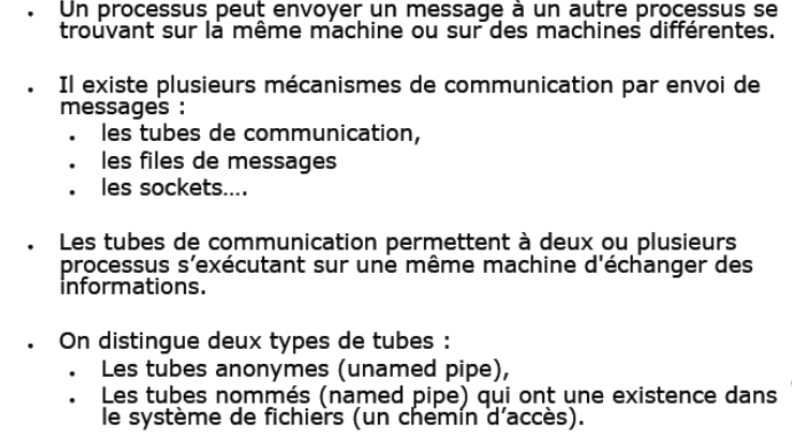
**Chapitre 3 :**

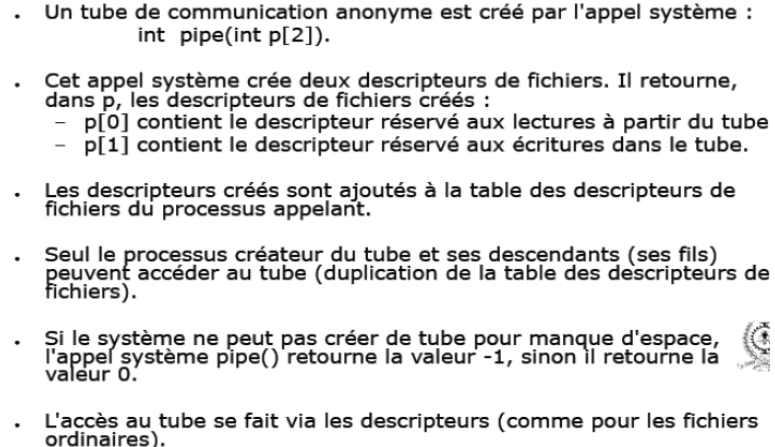
**Communication interprocessus**

**Une image contenant texte

Description générée automatiquement**

**Les tubes anonymes**

****

****

**Une image contenant texte

Description générée automatiquement**

**Une image contenant texte, personne, document

Description générée automatiquement**

**Les tubes anonymes :**

**Redirection de stdin et stdout**

**Une image contenant texte

Description générée automatiquement**

**Chapitre 4 Communication interprocessus**

**Le problème précédent est dû aux conflits d’accès à la même ressource.**

**La partie du programme à partir de laquelle on accède à la ressource partagée est appelée section (région) critique.**

**Solution: L’exclusion mutuelle est une méthode qui assure qu’un seul processus est autorisé d’accéder à une ressource partagée; les autres processus seront exclus de la même activité.**

**Quatre conditions doivent être vérifier pour assurer l’exclusion**

**mutuelle:**

**Deux processus ne doivent pas se trouver simultanément dans leurs sections critiques.**

**Aucun processus à l’extérieur de sa section critique ne doit bloquer les autres processus.**

**Aucun processus ne doit attendre indéfiniment pour entrer dans sa section critique.**

**Il ne faut pas faire d’hypothèse quant à la vitesse ou le nombre de processeurs**

**Algorithme d’accès à une section critique :**

**Entrer\_Section\_Critique () /\* attente si SC non libre \*/**

**Section\_Critique() /\* un seul processus en SC \*/**

**Quitter\_Section\_Critique()**

**L’attente peut être :**

**Active : la procédure Entrer\_Section\_Critique est une boucle dont la condition est un test qui porte sur des variables indiquant la présence ou non d’un processus en Section critique.**

**Non active : le processus passe dans l’état endormi et ne sera réveillé que lorsqu’il sera autorisé à entrer en section critique**

**Active :**

**.Solution 1: Masquage des interruptions**

**Lorsqu’un processus entre en section critique il doit masquer les interruptions.**

**Solutions de l’exclusion mutuelle par attente active**

**Pas de commutation de processus**

**Lorsqu’ il quitte sa section critique il doit restaurer les interruptions.**

**C’est une solution matérielle qui permet de résoudre complètement le problème. Mais elle est dangereuse en mode utilisateur s’il oublie de restaurer les interruptions.**

**Solution 2: Variables de verrouillage**

**Un verrou est variable binaire partagée qui indique la présence d’un processus en section critique.**

**si verrou=0 alors section critique libre**

**si verrou=1 alors section critique occupée**

**Cette solution ne garantie pas l’exclusion mutuelle car le verrou est une variable partagée qui peut constituer aussi une section critique.**

**Solution 3: Alternance stricte**

**Tour est une variable partagée qui indique le numéro de processus autorisé a entrer en section critique.**

**L’alternance stricte est une solution simple et facile a implémenter.**

**Mais, un processus qui possède Tour peut ne pas être intéressé immédiatement par la section critique et en même temps il bloque un**

**autre processus qui demande. Problème de famine**

**Solution 4: Solution de Peterson**

**Cette solution assure complètement l’exclusion mutuelle.**

**Mais, le processus qui attend sa section critique consomme du temps processeur inutilement**

**(attente active).**

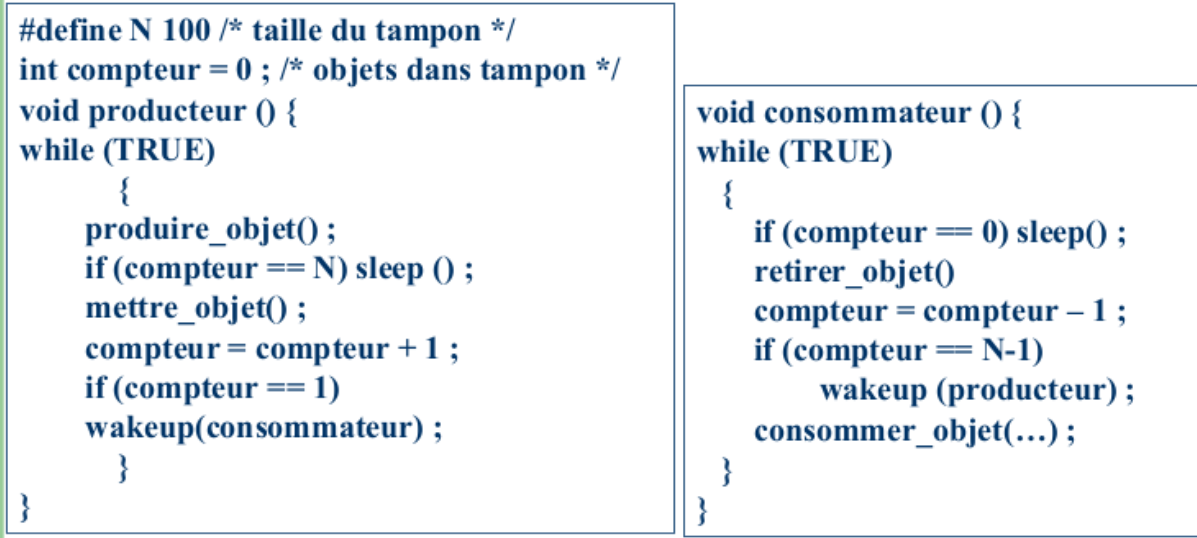
**Sans Active :**

**Les primitives Sleep et Wakeup:**

**Le système d’exploitation offre deux appels système:**

**1. Sleep (dormir) qui bloque le processus appelant.**

**2. Wakeup (réveiller) qui réveille le processus donné en argument.**

****

**Analyse de cette solution :**

**L’accès à la variable compteur n’est pas protégé, ce qui peut entraîner des incohérences dans les valeurs prises par cette Variable.**

**Réveils perdus : c’est le principal défaut de ce mécanisme. Un signal wakeup envoyé à un processus qui ne dort pas (encore)est perdu.**

**Sémaphore**

**Pour remédier au problème des réveils en attente (les wakeup perdus), l’idée est d’employé une variable entière appelée:**

**Sémaphore à laquelle est associée une file d’attente des processus bloqués.**

* **sémaphore=0 🡪 aucun réveil n’est mémorisé**
* **sémaphore>0 🡪 un ou plusieurs réveils sont en attente**

**Les sémaphores**

**Un sémaphore s est manipulé par les opérations :**

**down(s) : - décrémente la valeur de s si s>0,**

**- si s=0, alors le processus est mis en attente.**

**- si un ou plusieurs processus sont en attente sur ce**

**sémaphore, l'un d'entre eux est réveillé,**

**down(s) : -La vérification du compteur La décrémentation du compteur/Le passage en sommeil dans le cas d’un compteur égal à zéro.**

**up(s) :- La vérification du compteur ; -L’incrémentation du compteur/Réveil d’un processus\***

**Initialisation d'un sémaphore**

**Une image contenant texte

Description générée automatiquement**

**Une image contenant texte

Description générée automatiquement**

**Une image contenant texte

Description générée automatiquement**

**Chapitre 5Les Threads POSIX**

**Threads : De nombreux systèmes d’exploitation modernes offrent la possibilité d’associer à un même processus plusieurs chemins d’exécution (multithreading, multiflot d’exécution).**

**Un thread est une unité d’exécution rattachée à un processus, chargée d’exécuter une partie du processus.**

**Chaque thread a :**

**– un identificateur unique**

**– une pile d'exécution**

**– des registres (un compteur ordinal)**

**– un état…**

**Le multithreading permet l’exécution simultanée ou en pseudo-parallèle de plusieurs parties d’un même processus.**

**Les avantages de thread :**

**Améliorer la rapidité des applications (pas de blocages pour des tâches qui peuvent être effectuée**

**en parallèle)**

**Exploiter de manière efficace les machines multiprocesseur**

**Améliorer la structure de votre programme**

**Utiliser moins de ressources système**

**Les différents type de thread**

**Thread utilisateur :**

**Les threads utilisateur sont implantés dans une bibliothèque (niveau utilisateur) qui fournit un**

**support pour les gérer.**

**Les threads noyau**

**Les threads noyau sont directement supportés par le système d’exploitation. Le système d’exploitation se charge de leur gestion.**

**Du temps CPU est alloué à chaque thread. (modèle un-à-un)**

**Si un thread d’un processus est bloqué, un autre thread du même processus peut être élu par le noyau**

**Cette implémentation est plus intéressante pour les systèmes multiprocesseurs.**

**Création d’un thread :**

**int pthread\_create(pthread\_t \*tid, const pthread\_attr\_t \*tattr,**

**void\*(\*start\_routine)(void \*), void \*arg);**

**// renvoie 0 si l’appel réussit, sinon !=0 : identifiant de l’erreur**

**Attendre qu’un thread se termine :**

**int pthread\_join(thread\_t tid, void \*\*status);**

**Explication :**

**Suspend l'activité du thread appelant tant que le thread tid n'est pas terminée.**

**status contient la valeur de retour du thread tid lors de sa terminaison.**

**Terminer une thread:**

**void pthread\_exit(void \*status);**

**Explication :**

**Termine l‘exécution du thread courant avec une valeur de retour particulière**

**Avoir le TID d’un thread :**

**pthread\_t pthread\_self(void);**

**Thread et mutex :**

**int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);**

**int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);**

**int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);**