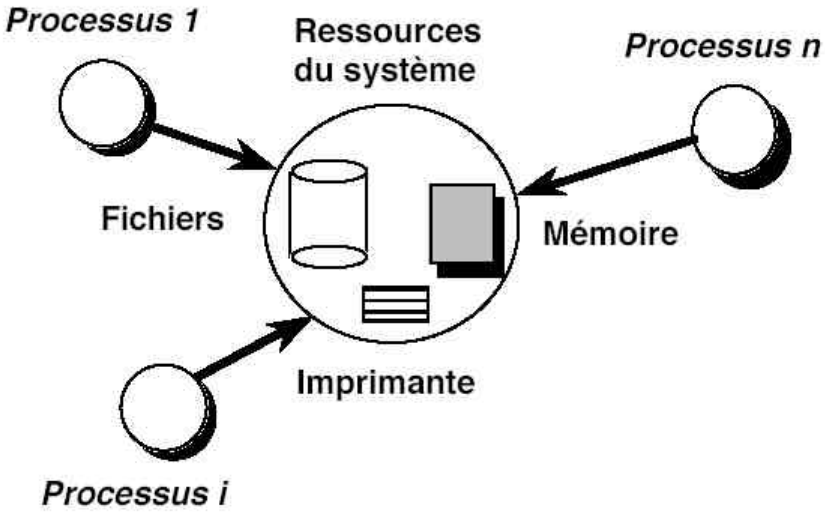
La synchronisation des processus

Partie 2a

La synchronisation

- Un système d'exploitation dispose de ressources (imprimantes, disques,mémoire, fichiers, base de données, ...), que les processus peuvent vouloir partager.



- Ils sont alors en situation de concurrence vis à vis des ressources.Il faut synchroniser leurs actions sur les ressources partagées.

La synchronisation

- Nombre de ressources d'un Système d'Exploitation est limité
- Problème **d'accès concurrent** vis-à-vis des ressources partagées entre plusieurs processus
- Problème de la synchronisation des actions sur ces ressources en cas de modification
- La partie de programme dans laquelle se font des accès à une ressource partagée s'appelle **une *section critique*.** (Ex Base de Donnée)
- L'accès à cette section critique devra se faire en ***exclusion mutuelle*** (ce qui implique de rendre indivisible la séquence d'instructions composant la section critique).

La synchronisation

- Interdire la lecture et l'écriture des données partagées à plus d'un processus à la fois : **mécanisme d'exclusion mutuelle**

- Quatre conditions doivent être réunis pour qu'une coopération entre processus soient corrects et efficaces :

  - Exclusion Mutuelle : Deux processus ne peuvent être **en même temps** en section critique

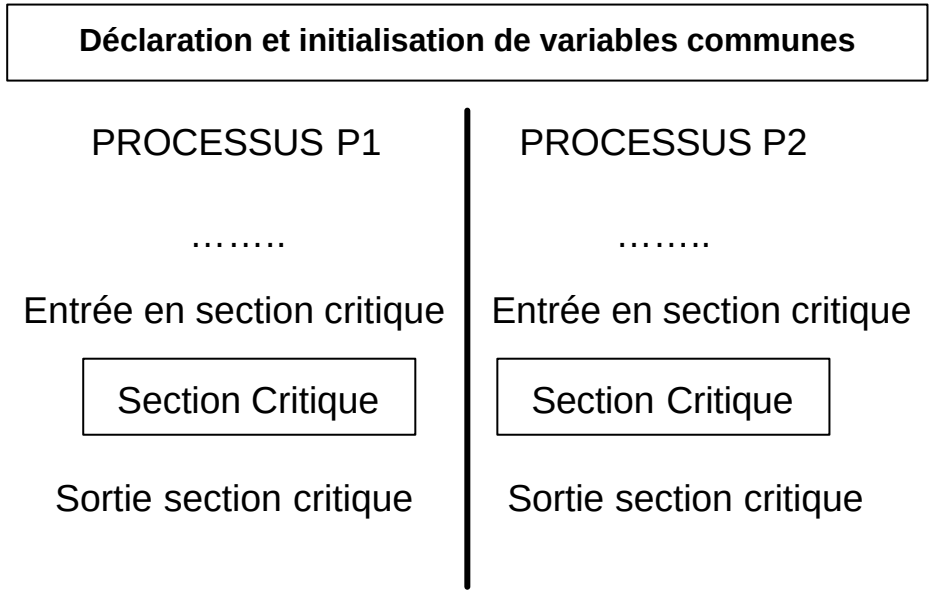
  - Aucune hypothèse ne doit être faite sur les **vitesses relatives** des processus et sur **le nombre** de processeurs.

  - Interblocage : Aucun **processus suspendu** en dehors d'une section critique ne doit **bloquer** les autres processus.

  - Famine : Aucun processus ne doit **attendre trop longtemps** avant d'entrer en section critique.

Réalisation d’une section critique

- Schéma Général



Exclusion mutuelle

- Il existe plusieurs méthodes pour réaliser **l'exclusion mutuelle** :
  - Avec attente Active
    - Les variables de verrouillages
    - l'Alternance
  - Sans attente Active
    - Les sémaphores
    - Et d'autres approches …..
- Le système garantit l 'absence d'interférence entre les exécutions par plusieurs processus

Attente Active

- Un processus désirant entrer dans une section critique doit être mis en attente si la section critique devient libre
- Un processus quittant la section critique doit le signaler aux autres processus
- Protocole d'accès à une section critique :

Processus 1

while(pos_verr(var)==echec);
section_critique();
lève_ver(var);

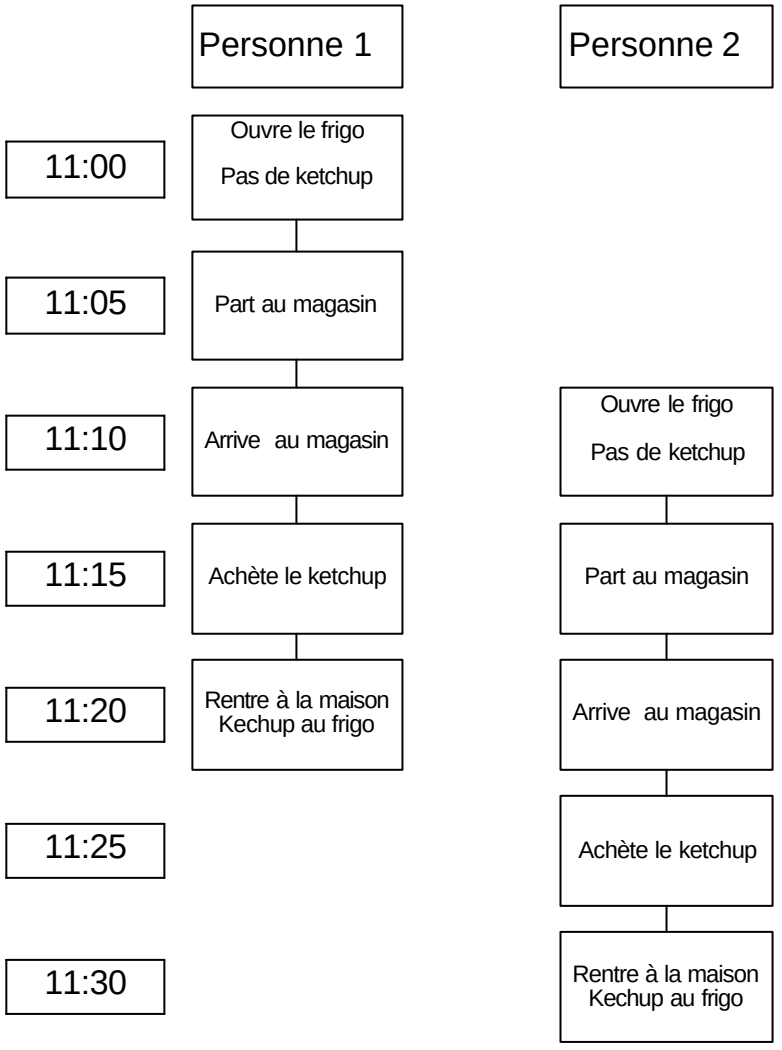
Processus 2

while(pos_verr(var)==echec);
section_critique();
lève_ver(var);

- *Teste si un processus peut rentrer en section critique, le processus **exécutant une boucle infinie** jusqu'à ce qu'il soit autorisé à rentrer.*

La synchronisation

Exemple : Trop de Ketchup

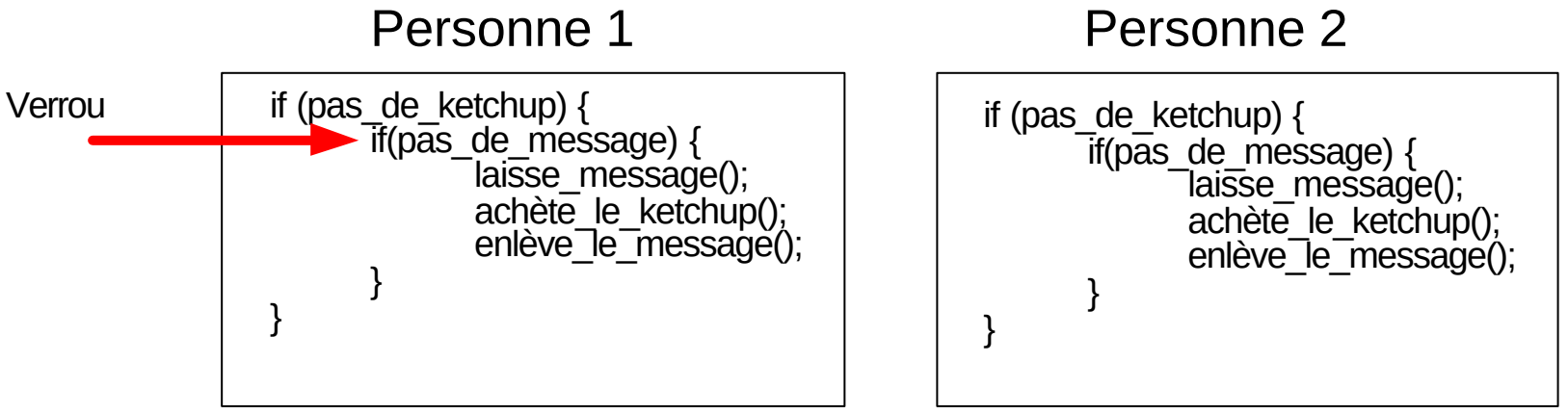


Verrouillage

- Variable de verrouillage : variable unique dont le contenu permettra de savoir s'il y a déjà un processus en section critique

  - Ex :Trop de Ketchup :
    - Pas plus d'une personne achète du ketchup
    - Quelqu'un achète du ketchup si nécessaire

  1- On laisse un message (verrou)
  2- On enlève le message (retrait du verrou)
  3- On n'achète pas s'il y a un message (attente)



Alternance

- Alternance : Tour à tour les processus accèdent à la section critique
  - Ex :Trop de Ketchup : on laisse un message ***signé***

Personne 1

```
écrit_message_A
if(pas_de_message_B) {
      if (pas_de_ketchup){
            achète_le_ketchup();
      }
}
enlève_le_message_A();
```

Personne 2

```
écrit_message_B
if(pas_de_message_A) {
      if (pas_de_ketchup){
            achète_le_ketchup();
      }
}
enlève_le_message_B();
```

  - Inconvénient : si A est interrompu après avoir laissé son message, mais avant d'avoir lu le message de B, et que B est interrompu de la même, façon alors **<u>blocage</u>**

Exclusion Mutuelle

- Ex :Trop de Ketchup : une solution

Personne 1

```
écrit_message_A();
while(message_B); /* Point X */
if(pas_de_ketchup) {

     achète_le_ketchup();

}
enlève_message_A();
```

Personne 2

```
écrit_message_B();
if(pas_de_message_A) { /*Point Y */

     if(pas_de_ketchup) {

          achète_le_ketchup;}

}
enlève_message_B();
```

Au point Y:

- s'il n'y a pas de message de A, B peut acheter sans crainte
- s'il y a un message de A, soit A est en train d'acheter, ou bien A attend que B s'en aille. B peut donc s'en aller sans crainte

Au point X:

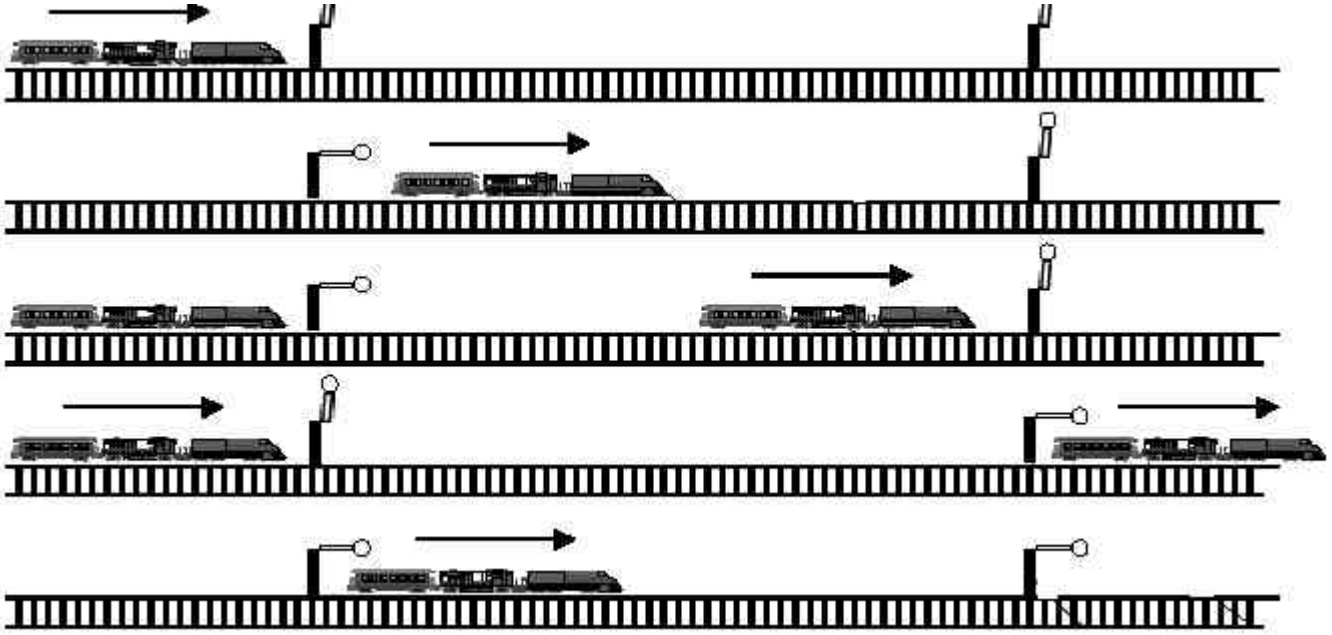
- s'il y a un message de B, on ne sait pas. Alors, on attend. Si B achète, A n'aura rien à faire à la fin. Si B n'achète pas, A le fera.

Les sémaphores

• Le concept de sémaphore est issu de la signalisation ferroviaire

• Il est utilisé pour contrôler l'accès à des tronçons de voie ferrée par des trains circulant les uns derrière les autres :

ressource non partageable



Les sémaphores

- La relation d'exclusion mutuelle n'est pas assez fine pour décrire la synchronisation entre processus: on peut utiliser le sémaphore

- Un sémaphore est composé :

  - d'**un marqueur** utilisé pour synchroniser plusieurs processus. Il agit comme une clef qu'un processus doit obtenir afin de continuer son exécution.
  - d'une **file d'attente des tâches** qui sont éventuellement bloqué devant ce sémaphore

- Distributeur de jetons :
  - Nombre de jetons fixé à la création du sémaphore, non renouvelable rendu après utilisation
  - 1 jeton = 1 verrou
  - Le nombre de jeton décroît quand il est acquis (« verrouillé »)
  - Le nombre de jeton croit quand il est libéré (« déverrouillé »)
  - Le nombre de jeton est toujours positif ou nul

Les sémaphores

- Ressource à n points d'accès ⟶ Sémaphore à n jetons

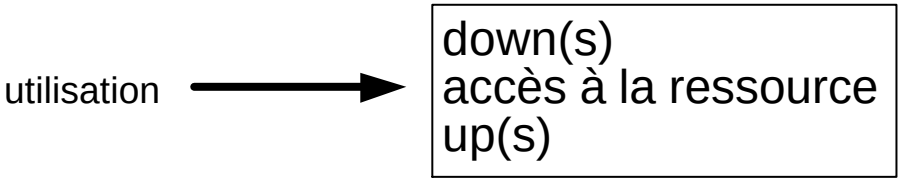
- Distributeur de jetons :
  - Nombre de jetons fixe, non renouvelable rendu après utilisation

P(s) = down(s)

S'il y a un jeton disponible, le processus l'obtient
S'il n'y en a pas, le jeton est rendu disponible

V(s) = up(s)

S'il y a des processus en attente, un seul obtient le jeton
S'il n'y en a pas, le jeton est rendu disponible



La communication interprocessus

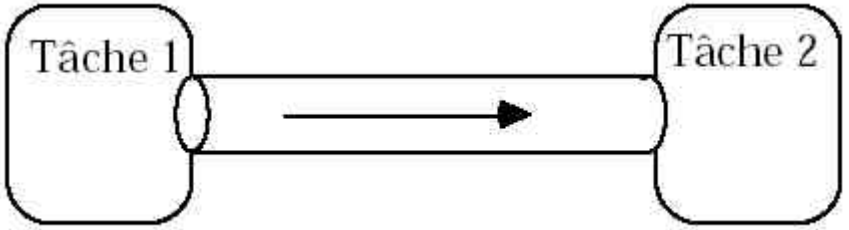
Partie 2b

La communication interprocessus

- Moyens pour les processus de communiquer:
  - **Les tubes ordinaires ou nommés,**
  - **Les files de messages**
  - **La mémoire partagée**
  - **Les signaux**

Les tubes

- Deux grands types : *les tubes nommés* et *les tubes ordinaires*
- Caractéristiques communes :
  - Assimilés à des fichiers dans le système
    - *Ces objets sont désignés localement dans les processus les utilisant par des descripteurs de fichiers*
    - *Les tubes peuvent être associés aux E/S standard des processus*
  - Mécanismes unidirectionnels
    - *Un processus est soit lecteur soit écrivain*
    - *Opération de lecture dans un tube est destructrice*
  - Transfert de données
    - *Mode Stream*
    - *Mode FIFO*



Les tubes ordinaires

- Caractéristiques des tubes ordinaires :
  - Assimilable à un fichier sans nom
  - Aucune entrée dans les répertoires
  - Impossible d'obtenir une entrée avec « Open »
  - L'écrivain et le lecteur sont de la même famille
  - Détruit lorsque le processus écrivain et le processus lecteur ont fermé tous les descripteurs
  - L'appel de la fonction du noyau ***pipe()*** crée un tampon de donnée dans lequel deux processus pourront venir respectivement ***Lire et Ecrire***
  - ***int pipe (int fd[2])*** ***fd[0] : Lecture , fd[1] : Ecriture***
  - Le système réalise **la synchronisation** en bloquant un processus qui essaye de lire dans un tube vide jusqu'à l'arrivée de données

Les tubes nommés

- Caractéristiques des tubes nommés :
  - Utilisation d'un fichier pour le transfert d'information
  - Possibilité de les partager entre processus sans liens de parenté
  - L'appel de la fonction du noyau mknod() crée un tube nommé

    int mkfifo (const char *ref, mode_t mode);
    ref : définit le chemin d'accès au tube
    mode : les droits d'accès à cet objet

    Shell : ***mkfifo – p [m mode) ref ;***

  - L'ouverture du tube se fait par le moyen de la primitive open()

Les files de message

- Les messages forment un moyen de communication privilégié
- Un message est constitué **d'un entier** (N°de message) et **d'une chaîne de longueur variable ou fixe** (le corps du message)
- Leur mode de fonctionnement est voisin des échanges de données sur un réseau
- Les communications se font à travers deux opérations :
  - **Send (Destinataire,message)** envoie un message au destinataire spécifié
  - **Receive (Source,message)** reçoit un message d'un processus source spécifié ou non
- Les opérations d'envoi et de réception entre processus peuvent être directes sans utilisation de mémoire partagée ou indirectes par l'intermédiaire d'une boite aux lettres.

Les files de message

- Les communications se font de manière synchrone ou asynchrone.

- Le synchronisme peut se représenter par la capacité d'un tampon de réception. Si le tampon n'a pas de capacité l'émetteur doit attendre que le récepteur lise le message pour pouvoir continuer. Les deux processus se synchronisent sur ce transfert et on parle alors d'un « **rendez-vous** ».

- Deux processus asynchrones : P et Q, peuvent aussi communiquer de cette manière en mettant en œuvre un mécanisme d'acquittement

P	Q
envoie(Q, message)	
	reçois(P, message)
	envoie(P, acquittement)
reçois(Q, acquittement)	

Les segments de mémoire partagée

- Les différents mécanismes de communication se base sur :
  - D'une part la recopie de données de l'espace d'adressage de l'émetteur vers le noyau et d'autre part la recopie de données du noyau vers l'espace d'adressage du processus destinataire.

- Le procédé le plus rapide pour transférer des données entre deux processus est de **ne rien transférer du tout**.

- Les processus **partagent des pages physiques** par l'intermédiaire de leur espace d'adressage. Il n'y a donc plus de recopie des informations.

- Les pages partagées par les processus sont des ressources critiques dont les accès doivent être synchronisés (au moyen de sémaphore par exemple).

Les segments de mémoire partagée

- Le partage d'une zone mémoire entre un nombre arbitraire de processus est offerte par le système UNIX

- La zone mémoire partagée est appelée segment de mémoire partagée.
- Il peut y avoir plusieurs segments partagés, chacun étant partagé par un ensemble de processus actifs.

- Un processus peut accéder à plusieurs segments de mémoire partagée.

- Ces segments ont une existence indépendante des processus.

- Un segment, une fois créé, pourra continuer d'exister même si aucun processus ne l'utilise.

- Un processus aura la possibilité de demander le rattachement d'un segment à son espace d'adressage, après quoi il pourra accéder aux données qu'il contient comme il le fait pour les autres données, c'est-à-dire via leur adresse (et ceci sans passer par l'intermédiaire d'un appel système).

Les signaux

- Le signal est **une interruption logicielle qui** diffère des messages sur les points suivants :

  - Un signal peut être émis à tout moment, occasionnellement à partir d'un autre processus,mais plus souvent à partir du noyau comme conséquence d'un événement exceptionnel
  - Un signal n'est pas nécessairement reçu ni pris en compte.
  - Par défaut, la plupart des signaux entraînent la terminaison du processus récepteur.
  - Un processus peut ignorer les signaux d'un type donné.
  - Les signaux n'ont pas de contenu informatif.
  - Le récepteur ne peut déterminer l'identité de l'émetteur.
  - Les signaux ne peuvent être envoyés qu'à des processus.
  - Les signaux ne doivent être utilisés que pour des événements exceptionnels, et jamais pour des communications ordinaires.

Les signaux

- La commande kill permet, depuis un processus shell, d'envoyer un signal de numéro donné à un processus (ou un groupe de processus).
- La primitive générale d'envoi d'un signal à un processus est la suivante : ***int kill (int pid, int signal) ;***
  - *L'ensemble des processus destinataires du signal sera fonction du pid passé en argument :*
    - *>**0 processus d'identité pid***
    - ***0 tous les processus dans le même groupe que le processus émetteur***
    - ***<-1 tous les processus du groupe |pid|***
  - *kill() rend 0 en cas de succès et -1 en cas d'échec**.***

- Par défaut, un signal provoque la destruction du processus récepteur, à condition bien sûr,que le processus émetteur possède ce droit de destruction.

Les signaux

- A chaque type de signal est associé, dans le système un handler
- Ce handler définit le comportement par défaut d'un processus auquel un exemplaire d'un signal de ce type est délivré.

- Les différents comportements par défaut possibles sont :
  - Terminaison du processus.
  - Terminaison du processus avec image mémoire (core).
  - Signal ignoré.
  - Suspension du processus.
  - Reprise d'un processus stoppé.

  - La primitive **pause** permet de se mettre en attente de l'arrivée de signaux.

  - Pause() est de l'attente pure : elle ne fait rien de particulier et n'attend rien de particulier. Cependant, puisque l'arrivée d'un signal interrompt toute primitive bloquée, on peut tout aussi bien dire que pause attend un signal. Le plus souvent le signal que pause attend est le signal d'alarme (SIGALARM) qui aura été au préalablement armé à l'aide de la primitive alarm ().

Les signaux

Signal	N°	Commentaires
SIGHUP	1	Signal émis lors d'une déconnexion
SIGINT	2	Il est émis à tout processus associé à un terminal de contrôle quand on appuie sur la touche d'interruption 'Ctrl-C'.
SIGQUIT	3	Semblable à SIGINT mais le signal est émis quand on appuie sur la touche d'abandon (normalement 'Ctl-\').
SIGILL	4	Instruction illégale.
SIGPFE	8	Erreur de calcul flottant.
SIGKILL	9	Signal d'interruption « radicale ». C'est la seule manière absolument sûre de détruire un processus, puisque ce signal est toujours fatal. Il ne peut être ni ignoré ni intercepté. A n'utiliser qu'en cas d'urgence. SIGTERM est préférable.
SIGALRM	14	Signal émis par alarm(int sec) au bout de sec secondes.
SIGTERM	15	Terminaison logicielle. Il s'agit du signal standard de terminaison. C'est le signal par défaut émis par la commande kill qui est également utilisé lors d'un arrêt du système pour mettre fin à tous les processus actifs. Tout programme devrait intercepter ce signal pour procéder à une mise en ordre rapide (par exemple supprimer les fichiers temporaires, libérer telle ou telle ressource,...) avant de se terminer par la fonction exit().
SIGUSR1	16	Premier signal à la disposition de l'utilisateur. Il peut être utilisé pour les programmes d'applications pour la communication inter-processus même si cela n'est pas très conseillé.
SIGUSR2	17	Deuxième signal à la disposition de l'utilisateur.
SIGCLD	18	Ce signal est envoyé au père à la terminaison d'un processus fils. Il agit différemment des autres signaux. En effet, si celui-ci est ignoré par le processus père, il lui permet d'ignorer le retour de ses fils sans que ces derniers encombrent la table des processus à l'état defunct.

La communication & la synchronisation interprocessus

Partie 2c

Les IPC

- Les mécanismes d'IPC permettent de faire communiquer et/ou de synchroniser n'importe quel couple de processus locaux (de la même machine) sans liens de parenté.

- Les trois mécanismes d'IPC sont :
  - Files de messages (msg),
  - Segments de mémoire partagée(msh),
  - Sémaphores(sem)

- Ces mécanismes ne sont plus désignés localement dans les processus par des descripteurs standards, et de ce fait il n'est plus possible d'utiliser les mécanismes de lecture et d'écriture standards sur de tels objets.

- Le système prend en charge la gestion de ces objets. C'est lui qui tient à jour **les tables systèmes** par type d’objet qui les contiennent.

Références IPC

- Les objets IPC sont référencés par deux noms :
  - **le numéro d'identification** (Id) dans le processus qui est retourné par les fonctions d'accès à la ressource get : msgget(), semget(), shmget().
  - **une clé** qui est utilisée dans l'appel de la fonction get pour identifier l'objet IPC du système auquel on cherche a accéder.
  - La commande ftok() permet de créer la clé
  - Pour partager un objet IPC, les processus partagent la clé externe qui lui est associée et utilisent les méthodes propres à chaque type d'objet IPC

Consultation des tables IPC

- La commande **ipcs** permet de consulter les tables systèmes, alors que la commande **ipcrm** supprime une entrée de la table :

```
$ ipcs
IPC status from /dev/kmem as of Tue Oct 20 08:56:30 1998
T     ID          KEY         MODE        OWNER      GROUP
Message Queues:
q     100         0x00000000  --rw-------  root       info
q     51          0x00000000  --rw-------  root       info
q     2           0x49179e95  --rw-rw-rw-  root       root
q     155         0x00000000  --rw-rw----  jmr        ens
Shared Memory:
m     0           0x41440014  --rw-rw-rw-  root       root
m     1           0x41442041  --rw-rw-rw-  root       root
Semaphores:
s     0           0x41442041  --ra-ra-ra-  root       root
s     1           0x4144314d  --ra-ra-ra-  root       root
$
```

T est le type de l'objet (***q*** pour **file de message**, ***m*** pour mémoire partagée et ***s*** pour sémaphore), **ID** est l'identification interne de l'objet, **KEY** (valeur hexadécimale) est la clé de l'objet (identification externe) qui identifie l'objet de manière unique au niveau système,**MODE** représente les droits d'accès à l'objet,**OWNER** et **GROUP** représentent respectivement l'identité du propriétaire de l'objet et l'identité du groupe propriétaire de l'objet.

Exemple : Les files de Message IPC

- C’est une implantation UNIX du concept de boîte aux lettres, qui permet la communication indirecte entre des processus.
- Les messages sont enregistrés et prélevés dans l’ordre FIFO
- Utilisation du principe des boites aux lettres : on dépose dans la boite un message que d'autres processus pourront lire.
- Le mode de lecture/écriture se fait de manière groupée par une structure de taille donnée. Chaque instruction de lecture ou d‘écriture se fait sur un message entier (toute la structure de message). Pour que les lectures soient compatibles avec les écritures, les messages sont typées.
- Règle d'or : le type d'un message est un entier strictement positif.
- Le type du message permet aux processus d'effectuer les bons ordres de lecture, mais aussi permet de sélectionner le ou les messages dans la file d'attente.

Exemple : Les files de Message IPC

- Le fichier de message <sys/msg.h>
- Quelques primitives permettant de paramétrer les appels :
  - MSG NOERROR l'extraction d'un message trop long n'entraîne pas d'erreur (le message est tronquée).
  - MSG R autorisation de lire dans la file.
  - MSG W autorisation d‘écrire dans la file.
  - MSG RWAIT indication qu'un processus est bloquée en lecture.
  - MSG WWAIT indication qu'un processus est bloquée en écriture.

  - Structure générique d’un message :

```
Struct msgbuf {
long mtype; /* type du message */
char mtext[NBCAR]; /* texte du message */
} message;

message.mtype = 3
strcpy(message.mtext, ‘’ j’ai compris les IPC ‘’);
msgsnd(id,&message,strlen(message.mtext),0);
```

Exemple : Les files de Message IPC

- **Envoi de message :**

#include <sys/msg.h>
int msgsnd (int dipc, const void *p_msg, int lg, int options);

Envoie dans la file de l'ipc le message pointée par p msg.

- lg taille du message égale à sizeof(struct msgbuf) - sizeof(long), le type du message n'étant pas compté dans cette longueur.
- Valeur de retour (0) succès (-1) échec.
  Un appel de msgsnd bloquée peut être interrompu par un signal ou par la destruction de la file de message. Dans ce cas, elle renvoie (-1)

- **Extraction de message**

int msgrcv(int dipc, void *p_msg, int taille, long type, int options);
est une demande de lecture dans la file de l'ipc d'un message de longueur inférieure ou égale à taille,
Le paramètre type permet de spécifier le type du message à extraire:
- si type > 0, le plus vieux message de ce type est extrait;
- si type == 0, le plus vieux message est extrait;
- si type < 0, le message le plus vieux du type le plus petit, mais inférieur ou égal à type est extrait. Ceci permet de définir des priorités entre les messages. Dans tous les cas, l'appel est bloquant si il n'y a pas de message du type voulu en attente.

Les processus Légers

Partie 2c

Les processus légers

- Dans les systèmes UNIX classiques, une abstraction fondamentale a été définie : **le processus**

- Cette abstraction recouvre à la fois:
  - un ensemble de ressources (essentiellement de l'espace d'adressage, mais aussi des fichiers ouverts et autres)
  - une unité d'exécution unique (l'activité sur ces ressources).

- Une autre approche multi processus / client serveur a été de séparer les deux composantes essentielles du processus, c'est-à-dire d'une part les ressources physiques, d'autre part les activités (unités d'exécution) pour former :
  - Un environnement d'exécution pour les ressources partagées : **les tâches**
  - Une activité : **les threads**

Les processus légers

- Les threads peuvent être vus comme des sous processus dans un processus
- Les threads :
  - partagent des ressources communes.
  - utilisent le même espace mémoire
  - exécutent le même code ou presque
  - accèdent aux même variables ou structures globales
  - utilisent les mêmes fichiers
  - **Correspondent à différents points d'exécution dans le programme : un contexte d'exécution différent**

- Il est constitué essentiellement d'un sous ensemble du P.C.B. (Processus Control Block d'UNIX) : de registres, une position dans la pile, un segment de pile propre, un pointeur dans le programme.

- Les ressources communes sont définies au niveau du processus et appartient au processus, elles peuvent être utilisées par tous les threads
- Chaque thread possède ses ressources spécifiques

Les processus légers

- La pile (stack) est un élément important du contexte d'exécution d'un thread car elle est utilisée pour :
  - Contenir les variables locales aux fonctions
  - Contrôler la navigation dans les appels imbriqués des fonctions
  - Le contexte n'est donc pas seulement défini par la valeur des registres du processeur mais également par l' état de la pile.
  - **Chaque thread possède sa propre pile**

- Les processus multi thread permettent à plusieurs thread de partager le même espace mémoire: ce partage peut engendrer des problèmes de synchronisation

Les processus légers

Processus classique (lourd)

espace d 'adressage protégé à un fil d 'exécution

Commutation de contexte
toujours changement d 'espace d 'adressage

Communication
outils entre espace d 'adressage (tubes, messages queues)

Pas de parallélisme dans un espace d 'adressage

Processus à threads (léger)

espace d 'adressage protégé à n fils d 'exécution (n ≥ 1)

Commutation de contexte : allégée
pas de changement d 'espace d 'adressage entre fils d 'un même processus

Communication : allégée
les fils d 'exécution d 'un même processus partagent les données, mais attention à la synchronisation

Parallélisme dans un espace d 'adressage

Les processus légers

- **Primitives**

  - **Création**

```
int pthread_create (pthread_t  *thread, const pthread_attr_t  *attr, void *
(*start_routine)(void *),  void   *arg);
```

  - **synchronisation entre threads**

```
int   pthread_join ( pthread_t    thread, void **value_ptr);
```

  - **terminaison de threads**

```
int   pthread_exit (void **value_ptr);
```

Les processus légers

• Les données de contrôles des ressources sont partagées par tous, mais chaque activité disposera de ses propres données de contrôle d'exécution.

•Chaque thread dispose de sa propre pile d'exécution.

•Un processus est un programme où une seule thread s'exécute.