IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Processus & Threads

Enseignante

Jocelyne Elias

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Concept de processus

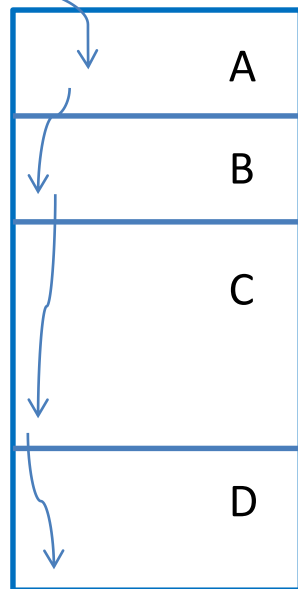
Définition : ***un processus est l'unité de travail dans un système en temps partagé; un processus est un programme en exécution***

- processus ≠ programme
- **Programme :** un programme est une entité passive (un fichier qui contient une liste d'instructions [section de textes] stockée sur le disque)
- **Processus :** concept central de tout système d'exploitation
  - Entité active
  - Section de données
  - Segment de mémoire dynamiquement alloué durant l'exécution …

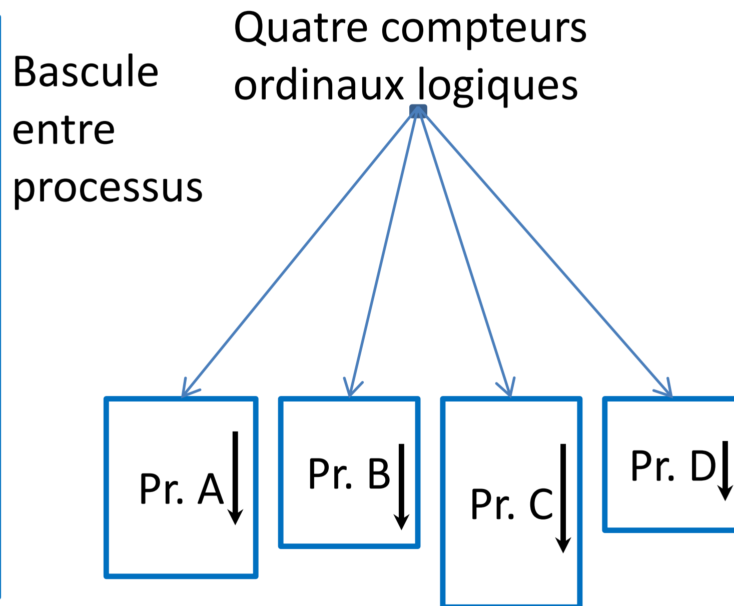
IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Le modèle de processus

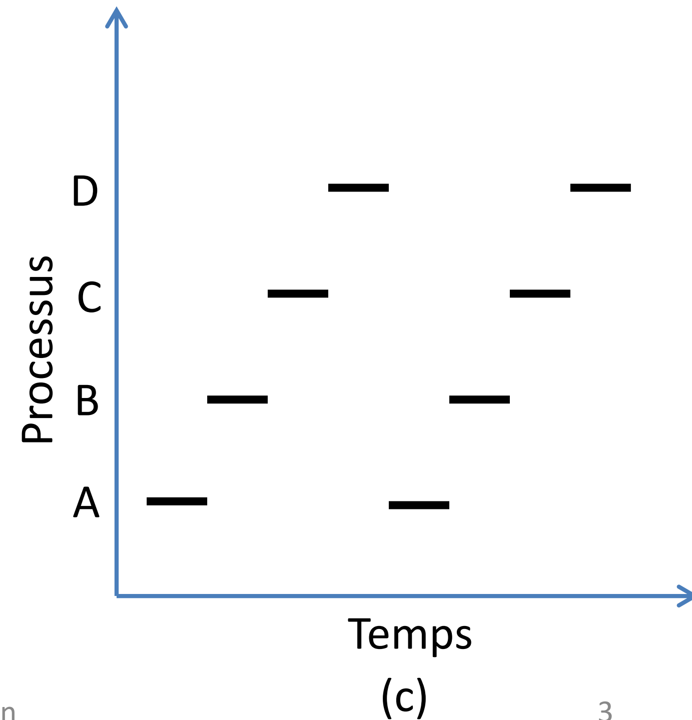
- Figure (a) : Multiprogrammation de 4 programmes
- Figure (b) : Modèle conceptuel de 4 processus séquentiels indépendants
- Figure (c) : Un seul programme est actif à un instant donné.



(a)



(b)



(c)

La création d'un processus

- Il existe essentiellement 4 évènements provoquant la création de processus :
  - Initialisation du système (lors de l'amorçage du SE).
  - Exécution d'un appel système de création de processus par un processus en cours d'exécution (appel système ***fork***, …).
  - Requête utilisateur sollicitant la création d'un nouveau processus (saisir une commande ou cliquer sur une icône)
  - Lancement d'un travail en traitement par lots (gros mainframes).

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

La fin d'un processus

- Rien n'est éternel, pas même un processus. Tôt ou tard, le processus s'arrête pour diverses raisons :
  - Arrêt normal (volontaire) : appel système ***exit***
  - Arrêt pour erreur (volontaire) : exécution d'une instruction illégale, d'une référence mémoire inexistante ou division par zéro.
  - Arrêt pour erreur fatale (involontaire) : compilation d'un programme inexistant (*gcc foo.c -o foo*).
  - Le processus est arrêté par un autre processus (involontaire) : appel système ***kill***

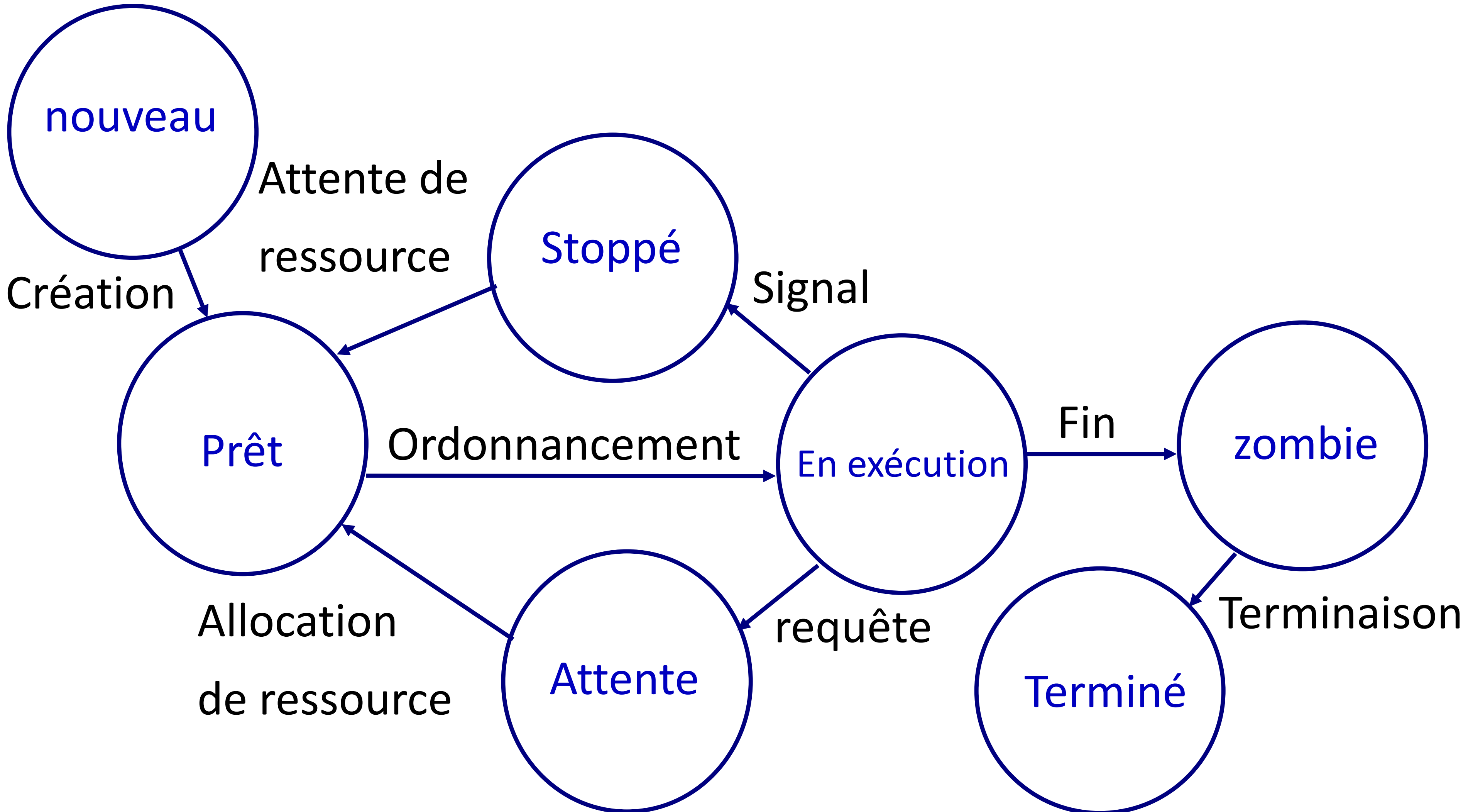
IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

L'état des processus

- L'état d'un processus est défini en partie par l'activité courante de ce processus.
- Un processus peut se trouver dans l'un des états suivants :
  - nouveau (new) : le processus est en cours de création
  - en exécution (running) : les instructions sont exécutées
  - en attente (waiting) : le processus attend qu'un évènement se produise (par exemple, saisie d'une donnée)
  - prêt (ready) : le processus est prêt pour être affecté à un processeur
  - terminé (terminated) : le processus a fini son exécution

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

L'état des processus



Concept de thread

- un thread est une unité de base de l'utilisation d'un processus.
- Le concept de thread a pour objet de permettre à plusieurs threads d'exécution de partager un jeu de ressources pour travailler ensemble, afin d'accomplir une tâche donnée.
- Les threads autorisent les exécutions multiples dans le même environnement de processus.

Concept de thread

Un thread possède les éléments suivants :

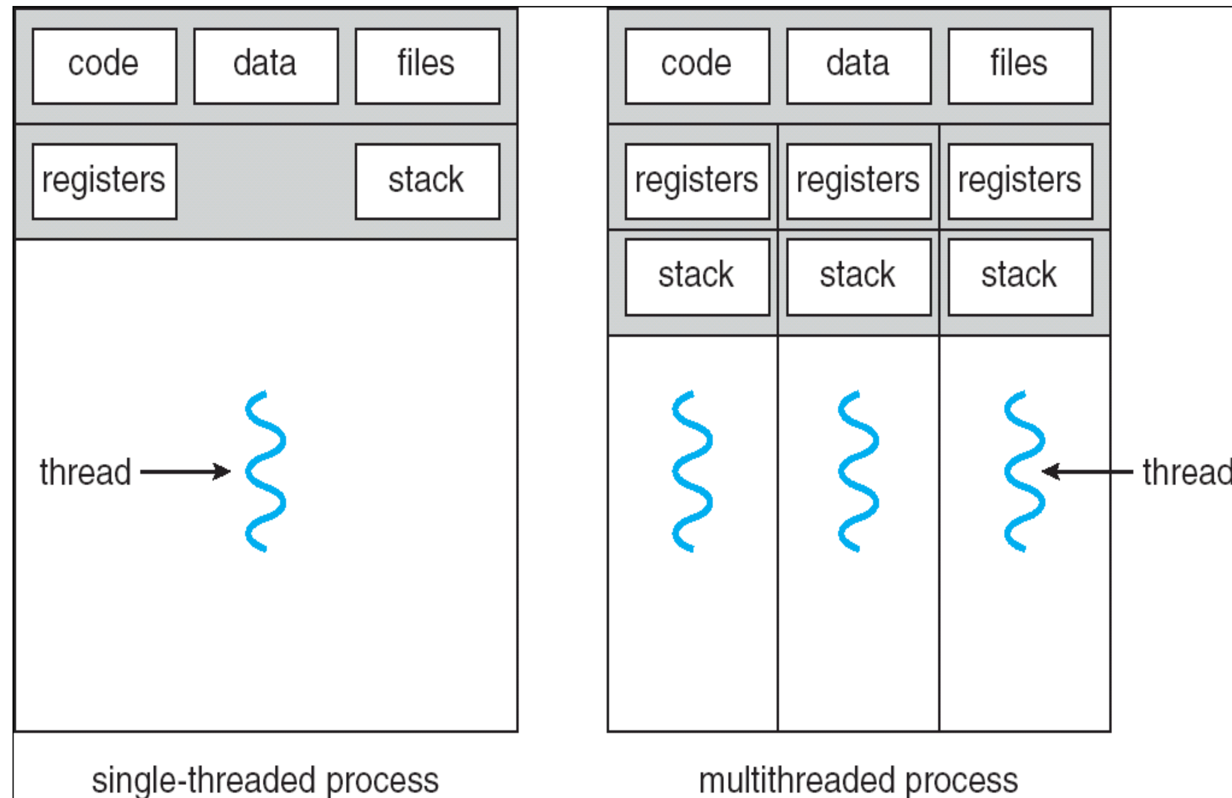
- un identificateur et l'état.
- un compteur ordinal : effectue le suivi des instructions à exécuter.
- un ensemble de registres : détiennent ses variables de travail en cours.
- une pile : historique de l'exécution + ens. de blocs d'activation (frames) correspondants aux procédures appelées mais n'ayant encore rien retourné.
  - Bloc d'activation : variables locales de la procédure et adresse de retour à employer une fois l'appel de procédure est exécuté.

Concept de thread

- Un thread partage avec d'autres threads d'un même processus :
  - L'espace d'adressage du processus
  - Variables globales
  - Fichiers ouverts
  - Processus enfant
  - Alertes en attente
  - Signaux et gestionnaires de signaux
  - Informations de décompte

Processus multithread

Un processus multithread : les threads font véritablement partie de la même tâche, et ils coopèrent étroitement et activement les uns avec les autres.



IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Un serveur Web multithread

- Le **thread dispatcher** lit les requêtes entrantes à traiter et active un **thread worker**
- Le **thread worker** vérifie si la requête peut être satisfaite dans le cache des pages Web. Sinon, il entame une opération ***read*** sur le disque

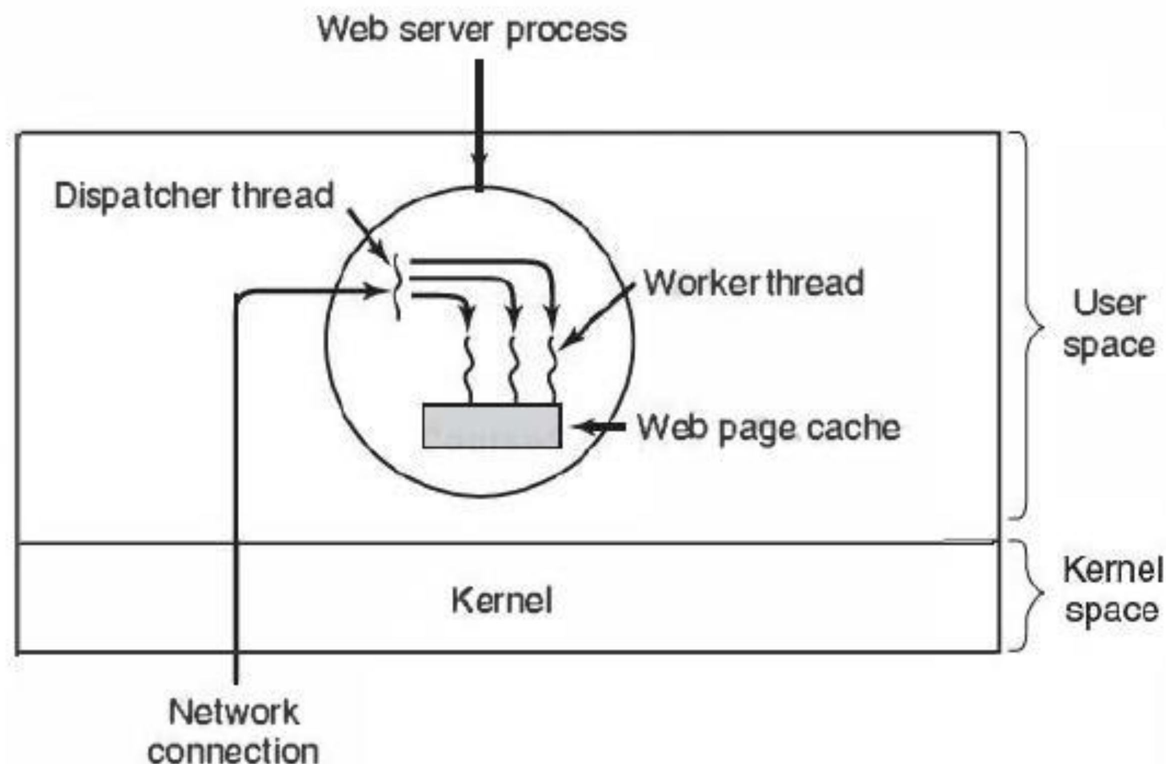


Figure 2. A multithreaded Web server

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Un serveur Web multithread

```
While (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

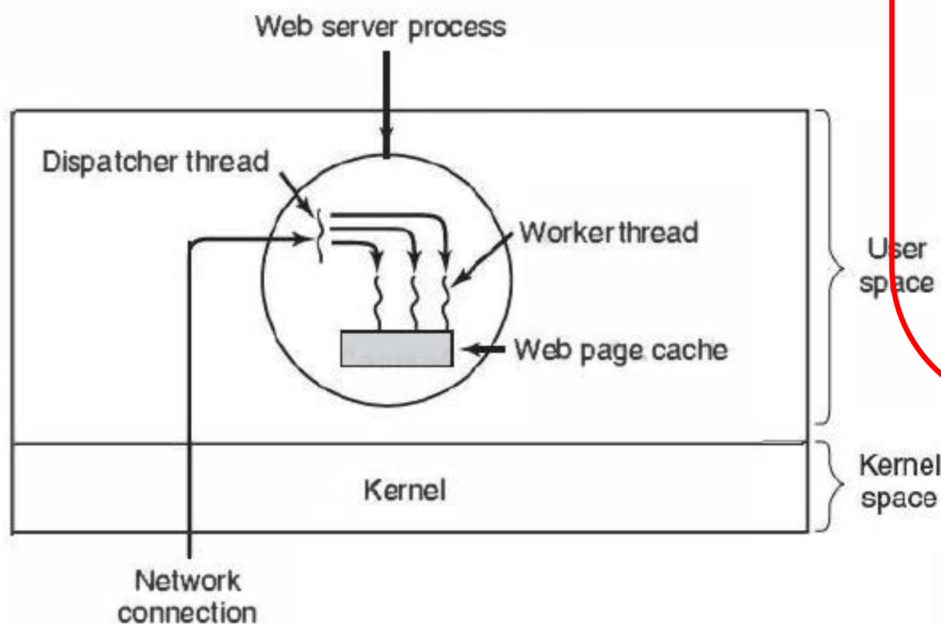


Figure 2. A multithreaded Web server

```
While (TRUE) {
    wait_for_work(&buf);
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page));
    read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Avantages des threads

- La création d'un thread est une opération 100 fois plus rapide que celle d'un processus.
- Les threads permettent de pouvoir superposer 2 ou plusieurs activités (traitement proprement dit et les E/S).
- La communication entre les threads est plus simple et plus rapide.
- Le partage des ressources est naturelle : par défaut, les threads partagent la mémoire et les ressources du même processus.
- La programmation multithread est plus efficace. Très utiles sur les systèmes multiprocesseurs => véritable parallélisme est possible.
- …

Les threads de POSIX

- Threads portables définis par IEEE (norme IEEE 1003.1c)
- Le paquetage **Pthreads** définit ce type de threads et est implémenté dans la plupart des systèmes Unix
- Ces threads possèdent un ensemble de propriétés : un **identificateur**, un **ens de registres** (dont un **compteur ordinal**) et des **attributs** (taille de pile, paramètres d'ordonnancement …) stockés dans une structure.
- Exemples d'appels de fonctions :
  - **pthread_create** : crée un nouveau thread
  - **pthread_exit** : termine le thread appelant
  - **pthread_join** : attend la fin d'un thread

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Les threads de POSIX

- Exemples d'appels de fonctions :
  - pthread_create : crée un nouveau thread
  - pthread_exit : termine le thread appelant
  - pthread_join : attend la fin d'un thread
  - **pthread_yield** : libère l'Unité Centrale (UC) pour laisser un autre thread s'exécuter
  - **pthread_attr_init** : crée et initialise une structure attribut du thread appelant
  - **pthread_attr_destroy** : supprime la structure attribut du thread

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Parallélisme

Objet : exécution de plusieurs activités

Définition :

Parallélisme (ou simultanéité) d'exécutions de 2 ou plusieurs activités <=> on peut observer à un instant *t* donné la prise en charge par la ''machine'' ou par le ''système informatique'' de l'exécution de 2 ou plusieurs activités.

Référentiel : le niveau d'observation où l'on se place :

- instruction machine
- tâche ou processus
- travail, session ou transaction (ensemble de tâches)

Parallélisme

- Architecture matérielle et niveau de parallélisme :
  - monoprocesseur : niveau de la tâche
  - multiprocesseur : niveau de l'instruction
  - Interactions entre tâches parallèles
    - concurrence : tentative d'accès multiple à une ressource non partageable
    - coopération : échange de données de manière cohérente

- Contrôle d'un ensemble de tâches parallèles :
  - centralisé : utilisation d' un ‘‘arbitre’’ ou ‘‘chef d'orchestre’’
  - distribué : règles de comportement mutuel

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Ressource et partage de ressource

Ressource : élément nécessaire à la progression d'un processus

Partage :

- **Ressource commune** : accessible par plusieurs processus, simultanément ou non
  - **Ressource commune partageable** (simultanément) : possibilité d'utilisation (ou d'allocation) simultanée à 2 ou plusieurs processus. Exemple : fichier en lecture, code exécutable, etc...
  - **Ressource commune non partageable** : l'utilisation de la ressource par plusieurs processus ne peut se faire que séquentiellement, un seul à la fois.

Exemple – ressource non partageable

Problème posé par une manipulation non atomique d’une ressource non partageable :

Soit une variable **F** non partageable, et
2 processus **A** et **B** s’exécutant en parallèle,
on suppose que, par un moyen quelconque du système (matériel ou logiciel), l’accès à **F** se fait en exclusion mutuelle entre **A** et **B**

Exemple – ressource non partageable

Processus **A**

1) -- Lire **A** à partir de **F**
2) -- **A** <-- **A** + 1
3) -- Ecrire **A** sur **F**

Processus **B**

4) -- Lire **B** à partir de **F**
5) -- **B** <-- **B** -1
6) -- Ecrire **B** sur **F**

L'exécution de ces processus (séquentiels) doit donc respecter l'ordre local à chacun de ces processus:

A : 1 --> 2 --> 3

B : 4 --> 5 --> 6

et l'exclusion mutuelle d'accès à **F** interdit le parallélisme d'exécution entre 1, 4, 3 et 6.

Exemple – ressource non partageable

Toute exécution respectant ces contraintes parait donc correcte et devrait donc donner le même résultat.

Soient les exécutions suivantes en prenant à chaque fois 10 pour valeur initiale de **F** :

1 --> 2 --> 3 --> 4 --> 5 --> 6

1 --> 2 --> 4 --> 5 --> 6 --> 3

4 --> 1 --> 2 --> 5 --> 3 --> 6

Les conditions de concurrence (race conditions)

Conclusion ?

Exclusion mutuelle

Unicité d'accès à une ressource : problème de l'exclusion mutuelle

Définition (exclusion mutuelle d'accès à une ressource pour plusieurs processus) : a tout instant *t* au plus un processus peut accéder à la ressource. Tout processus peut accéder à la ressource au bout d'un temps fini.

Mise en œuvre :

- par matériel :
    - accès à une case mémoire via un accès unique
    - accès à un monoprocesseur
- par logiciel centralisé : appels système (***flock()*** ou autre)
- par logiciel distribué : protocole d'accès

Exclusion mutuelle

Caractéristiques :

- L'exclusion mutuelle engendre une **restriction du parallélisme**
- Les processus concernés utilisent **séquentiellement** (mais dans un ordre aléatoire) la ressource
- Atomicité d'action
- Séquence indissociable :
  - instruction machine,
  - écriture d'un fichier sur imprimante
  - …

Section critique

- Séquence d'opérations nécessitant pendant tout leur déroulement et pour leur cohérence l'usage exclusif d'une ressource commune non partageable et non requérable.
- La section critique doit être considérée vis-à-vis de la ressource commune non partageable qu'elle utilise, dite alors ''ressource critique'' comme <u>une opération atomique</u>.
- La section critique doit alors être exécutée en exclusion mutuelle par rapport à toutes les autres sections critiques concernant une même ressource critique

Section critique - Exemples

Section critique
pour le Processus **A** :

1) -- Lire **A** à partir de **F**
2) -- **A** <-- **A** + 1
3) -- Ecrire **A** sur **F**

Section critique
pour le Processus **B** :

1) -- Lire **B** à partir de **F**
2) -- **B** <-- **B** -1
3) -- Ecrire **B** sur **F**

Un exemple d’utilisation des sémaphores

- Plus tard dans le cours …

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Gestion des processus

Appels Système

Appels système

Les *appels système* sont des fonctions fournies par le noyau du système d'exploitation et utilisées par les programmes utilisateurs. Quelques exemples :

- *fork* : création de processus
- *wait-exit* : synchronisation entre processus
- *execve, …, execl* : lancement d'un nouveau programme
- *flock et lockf* : protocole d'exclusion mutuelle
- *pipe* : création de tube de communication
- *kill-signal* : envoi et réception de signal
- *socket* : interface de communication (à distance)
- *autres ipc* : pour sémaphores, files de messages, mémoire partagée …

fork()

- <u>But</u> : Créer un nouveau processus.

- <u>Syntaxe</u> :

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

pid_t ***fork*** *(void)*

- Valeur de retour (si succès) = pid (identificateur) du processus fils pour le père et 0 pour le fils.
- Valeur de retour (si erreur) = -1 au père sans créer le fils.

getpid(), getppid()

- But : Connaître son propre identifiant (getpid()) et celui du père (getppid()) du processus appelant.

- Syntaxe :

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

*pid_t **getpid** (void) et pid_t **getppid** (void)*

- Valeur de retour (si succès) = pid (identificateur) du processus appelant (getpid()) et celui du père (getppid()).
- Valeur de retour (si erreur) = -1.

fork (Création de processus)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(void) {
        int np;
        system(“ps”);  ← Etat des processus en cours d’exécution (process status)
        if( (np=fork() ) < 0)  ← Création du fils
        { perror (“erreur de création”); exit(errno);}
        if (np==0) {
                printf (“\n Je suis le fils  num %d de père %d \n”, getpid(),getppid());
                system(“ps”);
                printf (“\n Je suis le fils  num %d, et je me termine \n”, getpid());
        }
        else {
                printf (“\n Je suis le père num %d créateur de %d \n”, getpid(),np);
                system(“ps”);
                printf (“\n Je suis le père num %d, et je me termine \n”, getpid());
        }
        exit(0);
}
```

wait et waitpid

- But : ***wait*** permet
  - l'élimination des processus zombis,
  - la synchronisation d'un processus sur la terminaison de ses descendants (ses processus fils) avec récupération des informations relatives à cette terminaison.
- ***wait*** provoque la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.
- Syntaxe :

```
#include <sys/types.h>
#include <sys/wait.h>
```

pid_t **wait** (int *pointeur_status)

  - Valeur de retour (si succès) = pid du processus fils qui vient de se terminer,
  - pointeur_status : état de terminaison du fils.

waitpid

- Syntaxe :

```
#include <sys/types.h>
#include <sys/wait.h>
```

pid_t **waitpid** (pid_t pid, int *status, int options)

- Valeur de retour = , si succès, pid du processus fils qui vient de terminer, sinon -1
- **pid** : pid du fils attendu à terminer.
- **status** : pointeur sur l'endroit où est sauvegardé l'état de terminaison du fils.
- **options** (***WNOHANG***, ***WUNTRACED***, …) : par exemple, blocage ou non du processus jusqu'à la terminaison d'un fils ou réception d'un signal, …

waitpid

pid_t **waitpid** (pid_t pid, int *status, int options)

Plus de détails sur la valeur de **pid** :

- Si **pid** > 0 : la fonction attend la fin du processus fils dont l'identifiant est pid.
- Si **pid** = 0 : la fct attend la fin de n'importe quel pr. fils appartenant au même groupe que le pr. appelant.
- Si **pid** = -1 : la fct attend la fin de n'importe quel fils, comme avec la fct ***wait()***
- Si **pid** < -1 : la fct attend la fin de n'importe quel pr. fils appartenant au groupe de pr. dont le numéro est ***-pid***.

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

fork + wait

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(void) {
        int np, np2, stat;
        system("ps");
        if( (np=fork() ) < 0)
        { perror ("erreur de création"); exit(errno);}
        if (np==0) {
                printf ("Je suis le fils  num %d de père %d \n", getpid(),getppid());
                system("ps");
                printf ("Je suis le fils  num %d, et je me termine \n", getpid());
                exit(1);
        }
        else {
                printf ("Je suis le père num %d créateur de %d \n", getpid(),np);
                printf("J'attends la fin du fils %d \n", np);
                np2 = wait(&stat);
                printf ("Le fils %d s'est terminé avec la valeur de retour %d \n", np2, stat>>8);
                system("ps");
        }
        exit(0);
}
```

exit(1); ← Terminaison normale du fils avec une valeur de retour égale à 1

np2 = wait(&stat); ← Attente de la terminaison du fils; **np2** contient le pid du fils et **stat** contient le statut du terminaison du fils

Identification de l'utilisateur et du groupe du processus

- uid_t **getuid()** ⟹ UID (User IDdentifier) réel : est celui de l'utilisateur ayant lancé le programme.
- uid_t **geteuid()** ⟹ UID effectif : est celui qui correspond aux privilèges accordés au processus.
- UID effectif peut être différent de l'UID réel si le fichier (exécutable) dispose de l'attribut Set-UID (droits spéciaux !!).
- int **setuid**(uid_t uid_effectif)
- int **seteuid**(uid_t uid_effectif)
- gid_t **getgid**()
- gid_t **getegid**()
- Idem, GID effectif peut être différent du GID réel si le fichier (exécutable) dispose de l'attribut Set-GID.
- …

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

main(..., ...) - rappel

main (int nbarg, char *tbarg[])

nbarg : nombre d’arguments passés sur la ligne de commande
tbarg : liste des arguments
tbarg[0] : nom du programme en exécution
tbarg[1] : premier argument
tbarg[2] : deuxième argument
...
tbarg[nbarg-1] : dernier argument
tbarg[nbarg] : NULL

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

main(…, …) - rappel

main (int nbarg, char *tbarg[])

Exemple :

./montest fichier1.txt 100 fichier2.txt chaine1 chaine2

nbarg = 6

tbarg[0] = montest

tbarg[1] = fichier1.txt

tbarg[2] = 100

tbarg[3] = fichier2.txt

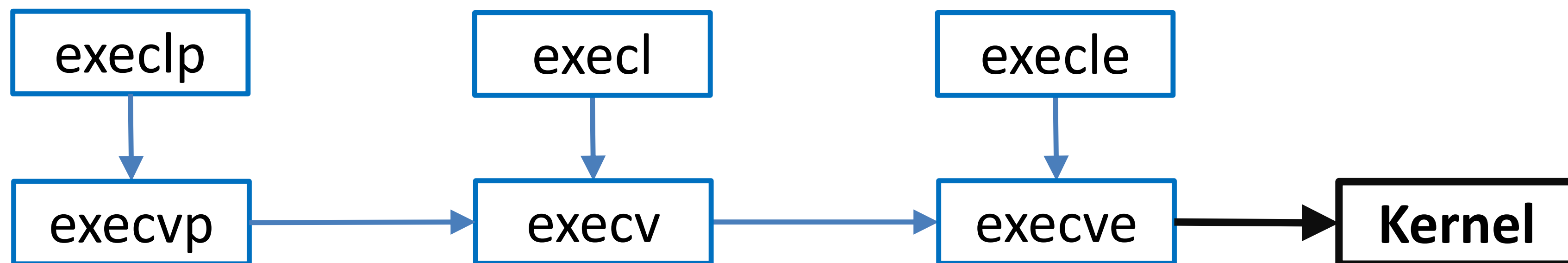
tbarg[4] = chaine1

tbarg[5] = chaine2

tbarg[6] = NULL

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Fonctions de la famille exec()



- Exécuter un nouveau programme => appeler l'une des fonctions de la famille ***exec()***.
- L'appel de l'une des fonctions de ***exec()*** permet de remplacer l'espace mémoire du processus appelant par le code et les données de la nouvelle application.
- Ces fonctions ne reviennent qu'en cas d'erreur, sinon le processus appelant est entièrement remplacé.
- Sous Linux, existe la fct. ***execve()***. Toutes les autres fonctions sont implémentées dans la bibliothèque C à partir de ***execve()***.

Fonctions de la famille exec()

execlp → execvp
execl → execv
execle → execve
execvp → execv → execve → **Kernel**

execl

- But :
  - remplacer un processus par un nouveau programme; le ***pid*** du processus ne change pas.
  - execl ne crée pas un nouveau processus.
  - execl simplement remplace la pile (stack), les données et le texte du processus courant par ceux du nouveau programme lu à partir du disque.
  - Cette fonction est complémentaire à la fonction *fork()*

- Syntaxe :

```
#include <unistd.h>

int execl (const char *path, const char *arg, ...)
```

  - ***execl*** retourne uniquement en cas d’erreur avec la valeur -1.

execl – un exemple

execl

```
#include <stdio.h>        #include <stdlib.h>       #include <unistd.h>        #include <errno.h>
#include <sys/types.h>    #include <sys/stat.h>
#define  Tmaxnom 30
int main(int nbarg, char * tbarg[]) {
	int idpro, df;
	char chaine[Tmaxnom];
	struct stat descfich;
	system("ps");
	if( (idpro=fork() ) < 0)
	{ perror ("erreur de création"); exit(errno);}
	if (idpro==0) {
		printf ("Je suis le fils  num %d de père %d \n", getpid(),getppid());
		df = open("ajouter", O_RDWR|O_CREAT, 0600);
		printf ("Index du fichier ajouter %d \n", df);
		fstat(df, &descfich);
		printf ("\t\t Num d'Inode:  %d \n", descfich.st_ino);
	}
	else {
		printf ("Je suis le père num %d créateur de %d \n", getpid(),idpro);
		system("ps");
		sleep(3);
		printf ("Je suis le père, num %d, et je me termine \n", getpid());
	}
	exit(0);
}
```

Nombre d'arguments (nbarg)

Tableau de pointeurs sur des chaines de caractère (* tbarg[])

execl (suite)

```
#include <stdio.h>        #include <stdlib.h>        #include <unistd.h>        #include <errno.h>
#include <sys/types.h>    #include <sys/stat.h>
#define  Tmaxnom 30

int main(int nbarg, char * tbarg[]) {
        int idpro, df;
        char chaine[Tmaxnom];
        struct stat descfich;
        system("ps");
        if( (idpro=fork() ) < 0)
        { perror ("erreur de création"); exit(errno);}
        if (idpro==0) {
                printf ("Je suis le fils  num %d de père %d \n", getpid(),getppid());
                df = open("ajouter", O_RDWR|O_CREAT, 0600);
                printf ("Index du fichier ajouter %d \n", df);
                // fstat(df, &descfich);
                // printf ("\t\t Num d'Inode:  %d \n", descfich.st_ino);
                sprintf(chaine, "%d", df);
                if ( execl("./compl", "compl", chaine, NULL) == -1)
                        { perror ("erreur dans compl"); exit(errno);}
                else // On ne peut pas passer ici !!!
                        printf ("Je suis le fantôme du fils \n");
        }
        else { … }
        exit(0);
}
```

sprintf : Identique à printf(…) mais le résultat est inséré dans chaine

execl : ← Lancer un nouveau programme ("compl") qui substitue le processus appelant, avec 2 arguments: l'identificateur du fichier et NULL. Le nouveau programme exécuté garde le même identificateur (pid)

execl (suite)

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

#define  Tmaxnom 30

int main(int nbarg, char * tbarg[]) {
        int df;
        char chaine[Tmaxnom];
        struct stat descfich;

        printf (“Je suis le fils  num %d de père %d, et nv. programme: %s \n”, getpid(),getppid(),tbarg[0]);
        system(“ps”);
        printf (“Index du fichier ajouter reçu %s \n”, tbarg[1]);
        sscanf(tbarg[1], “%d”, &df);
        fstat(df, &descfich);
        printf (“\t\t Num d’Inode par %s:  %d \n”, tbarg[0] , descfich.st_ino);
        printf (“Le fils  num %d se termine\n”, getpid());
        exit(0);
}
```

Verrouillage d'un fichier

- Il y a deux manières pour faire le verrouillage :
  - La fonction ***flock()*** *: file lock,* un appel système hérité de l'univers Berkeley Software Distribution (BSD)
  - Les commandes *F_GETLK*, *F_SETLK* et *F_SETLKW* de la fonction ***fcntl()***.
  - ***fcntl()*** est une fonction qui permet de consulter ou de paramétrer plus aspects d'un descripteur de fichier.
    - Cette fonction est déclarée dans <fcntl.h>
    - Syntaxe : int ***fcntl*** (int descripteur, int commande, …)

flock()

- Verrouillage d'un fichier : l'appel système ***flock()*** (*file lock*, héritée de l'univers Berkeley Software Distribution (BSD))
  - Porte sur un *descripteur* de fichier
  - Verrouille le fichier tout entier
  - Un seul verrou sur le fichier ⟹ verrou exclusif
  - Plusieurs verrous sur le même fichier ⟹ verrou partagé
  - Le verrou est consultatif, c.-à-d. qu'il n'empêche aucune autre opération que celle de la pose de verrou
  - Le verrou peut être bloquant ou passant (***LOCK_NB***)

flock() (suite)

- int ***flock*** (int fd, int operation) : verrouiller/déverrouiller le fichier dont le descripteur est *fd*. Valeur de retour = 0 (succès) ou -1 (si erreur).
  - *fd* : descripteur du fichier
  - Operation :
    - LOCK_SH : verrous partagés (ou plusieurs verrous) sur le même fichier *fd*.
    - LOCK_EX : verrou exclusif (ou un seul verrou) sur le fichier *fd*.
    - LOCK_UN (**unlock**) : pour déverrouiller le fichier *fd*.
    - LOCK_NB : ne pas bloquer la fonction lors de la demande de verrouillage du fichier *fd*. Dans ce cas là, elle retourne avec l'erreur *EWOULDBLOCK*.

- int ***lockf*** (int fd, int cmd, off_t len) : verrouiller/déverrouiller des parties d'un fichier

IUT DE PARIS

Programme ajout.c

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

```
#define ID_PROC         Ajout
#define N_Boucles       10
#define ATTENTE         100
#define NB_INST         5
static int num_inst=0 ;
void inst_suiv (void) {
        int cpte;
        for(cpte=0; cpte  < ATTENTE*100000; cpte++);
        num_inst= (num_inst<NB_INST)?num_inst+1:1 ;
        printf(" AJOUT instruction %d \n", num_inst);
}
void main (int nbarg, char *tbarg[])
{
        int df_n, nombre, num_boucle, nbo_lus, nbo_ecrits ;
        off_t lret ;
        if ((df_n= open ("NOMBRE",O_RDWR|O_CREAT,0644)) < 0) {
                perror("ouverture de NOMBRE");
                exit(errno);
        }
        for (num_boucle=0;num_boucle<N_Boucles;num_boucle++) {
                inst_suiv();

                inst_suiv();
                if ((lret=lseek(df_n,(off_t)0,SEEK_SET)) < 0)  { perror("1er lseek sur NOMBRE"); exit(errno);  }
                if ((nbo_lus=read(df_n,&nombre,sizeof(int))) < sizeof(int)) {
                        perror("lecture NOMBRE");
                        exit(errno);
                }
```

fonction de ralentissement permettant un entrelacement des programmes concurrents

Lire sizeof(int) du fichier df_n dans nombre,
nbo_lus : nombre d'octets lus si succès

Programme ajout.c (suite)

```
                inst_suiv();
                nombre++;
                inst_suiv();
                if ((lret=lseek(df_n,(off_t)0,SEEK_SET)) < 0) {
                        perror("2ème lseek sur NOMBRE");
                        exit(errno);
                }
                if ((nbo_ecrits=write(df_n,&nombre,sizeof(int))) < sizeof(int))
                {       perror("écriture NOMBRE");
                        exit(errno);
                }
                inst_suiv();

        }/* end for */

        if ((lret=lseek(df_n,(off_t)0,SEEK_SET)) < 0) {
                perror("dernier lseek sur NOMBRE");
                exit(errno);
        }
        if ((nbo_lus=read(df_n,&nombre,sizeof(int))) < sizeof(int)) {
                perror("lecture NOMBRE");
                exit(errno);
        }
        printf("Valeur finale (PLUS) nombre: %d \n",nombre);
        close(df_n);
}
```

Aller au début du fichier df_n

SEEK_CUR : position courante
SEEK_END : fin du fichier

Ecrire sizeof(int) du nombre sur le fichier df_n,
nbo_ecrits : nombre d’octets écrits si succès

flock() – un exemple

programme de démonstration de l'appel flock

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

#define ID_PROC         Ajout
#define N_Boucles       10
#define ATTENTE         100
#define NB_INST         5
static int num_inst=0 ;
void inst_suiv (void) {
        int cpte;
        for(cpte=0; cpte  < ATTENTE*100000; cpte++);
        num_inst= (num_inst<NB_INST)?num_inst+1:1 ;
        printf(" AJOUT instruction %d \n", num_inst);
}
void main (int nbarg, char *tbarg[])
{
        int df_n, nombre, num_boucle, nbo_lus, nbo_ecrits ;
        off_t lret ;
        if ((df_n= open ("NOMBRE",O_RDWR|O_CREAT,0644)) < 0) {
                perror("ouverture de NOMBRE");
                exit(errno);
        }
        for (num_boucle=0;num_boucle<N_Boucles;num_boucle++) {
                inst_suiv();
                if (flock(df_n,LOCK_EX) < 0 )  { perror("verrouillage "); exit(errno); }
                inst_suiv();
                if ((lret=lseek(df_n,(off_t)0,SEEK_SET)) < 0)  { perror("1er lseek sur NOMBRE"); exit(errno);  }
                if ((nbo_lus=read(df_n,&nombre,sizeof(int))) < sizeof(int)) {
                        perror("lecture NOMBRE");
                        exit(errno);
                }
```

fonction de ralentissement permettant un entrelacement des programmes concurrents

Lire sizeof(int) du fichier df_n dans nombre, nbo_lus : nombre d'octets lus si succès

flock() – un exemple

```
            inst_suiv();
            nombre++;
            inst_suiv();
            if ((lret=lseek(df_n,(off_t)0,SEEK_SET)) < 0) {
                        perror("2ème lseek sur NOMBRE");
                        exit(errno);
            }
            if ((nbo_ecrits=write(df_n,&nombre,sizeof(int))) < sizeof(int))
            {           perror("écriture NOMBRE");
                        exit(errno);
            }
            inst_suiv();
            if (flock(df_n, LOCK_UN) < 0 ) {  perror("déverrouillage ");  exit(errno); }
      }/* end for */

      if ((lret=lseek(df_n,(off_t)0,SEEK_SET)) < 0) {
                  perror("dernier lseek sur NOMBRE");
                  exit(errno);
      }
      if ((nbo_lus=read(df_n,&nombre,sizeof(int))) < sizeof(int)) {
                  perror("lecture NOMBRE");
                  exit(errno);
      }
      printf("Valeur finale (PLUS) nombre: %d \n",nombre);
      close(df_n);
}
```

Aller au début du fichier df_n

SEEK_CUR : position courante
SEEK_END : fin du fichier

Ecrire sizeof(int) du nombre sur le fichier df_n,
nbo_ecrits : nombre d'octets écrits si succès

Enlever le verrou

fcntl()

- Les verrous sont représentés par des structures *flock*
- La structure *flock* comprend les 5 champs suivants :
    - l_type (short int) : type du verrou (F_**RD**LCK, F_**WR**LCK, F_**UN**LCK)
    - l_whence (short int) : la portion du fichier concernée par le verrouillage (peut prendre les valeurs SEEK_SET, SEEK_CUR et SEEK_END)
    - l_start (off_t) : début de la portion verrouillée (p/r à l_whence)
    - l_len (off_t) : longueur de la partie à verrouiller dans le fichier, mesurée en octets.
    - l_pid (off_t) : le pid du processus détenteur du verrou.

- F_RDLCK : le processus demande un accès en lecture
- F_WRLCK : accès en écriture

IUT DE PARIS

fcntl() - exemple

```
struct flock   my_lock;
char    chaine[2];

my_lock.l_type = F_WRLCK;
my_lock.l_whence = SEEK_SET;
my_lock.l_start = 0;
my_lock.l_pid = 0;

while( fcntl(fd, F_SETLK, & my_lock) < 0 ) {
        fprintf( stdout, '' Fichier verrouille, reessayer ? '');
        fgets(chaine, 2, stdin);
        if (toupper(chaine[0]) == 'O')
                continue;
        return -1;
}

// Ici l'accès est autorisé, on peut faire les modifications et puis libérer le verrou.
fcntl(fd, F_UNLCK, & my_lock);
return 0;
```

Communications entre processus

Inter-Process Communication

Inter-Process Communication

Les *IPC* (Inter-Process Communication) regroupent un ensemble de mécanismes permettant à des processus concurrents (ou distants) de communiquer :

- *Fichiers (+ verrous) : diapositives précédentes …*
- ***Tubes*** *(pipes en anglais) et* ***tubes nommés*** *(named pipes)*
- *Files de messages :* échange de messages entre processus *(send et receive)*.
- *Mémoires partagées:* Une modification apportée par un processus est perçue par les autres processus.
  - ***Sémaphores***
- *Signaux …*

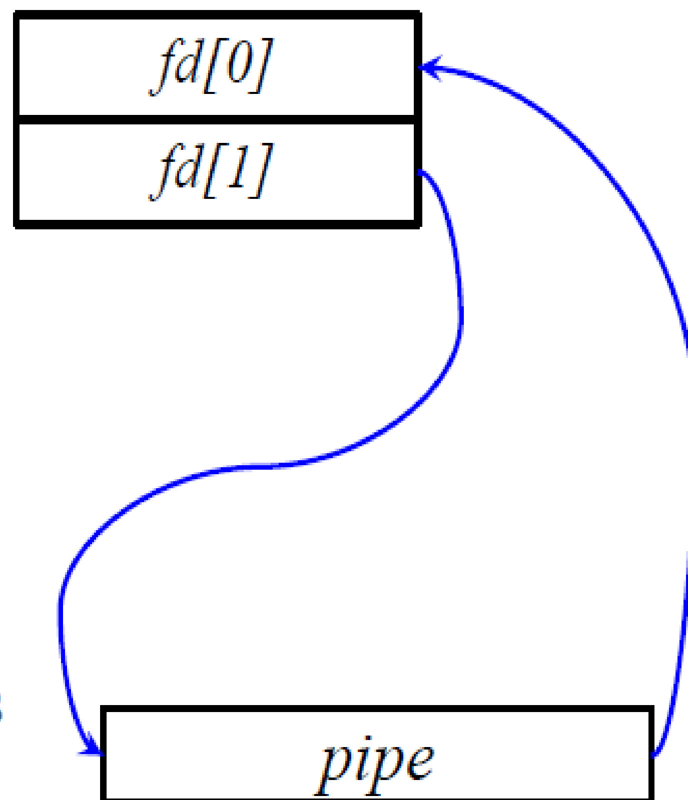
Les Tubes

- **But :** ***pour permettre la communication entre processus***
- Le premier mécanisme introduit dans *Unix* pour la communication entre processus
- Assimilé à un fichier (***virtuel***) ouvert en lecture et ouvert en écriture
- Pas de représentation disque
- Durée de vie limitée à celle des processus qui l'utilisent
- Partageable entre processus d'une même famille

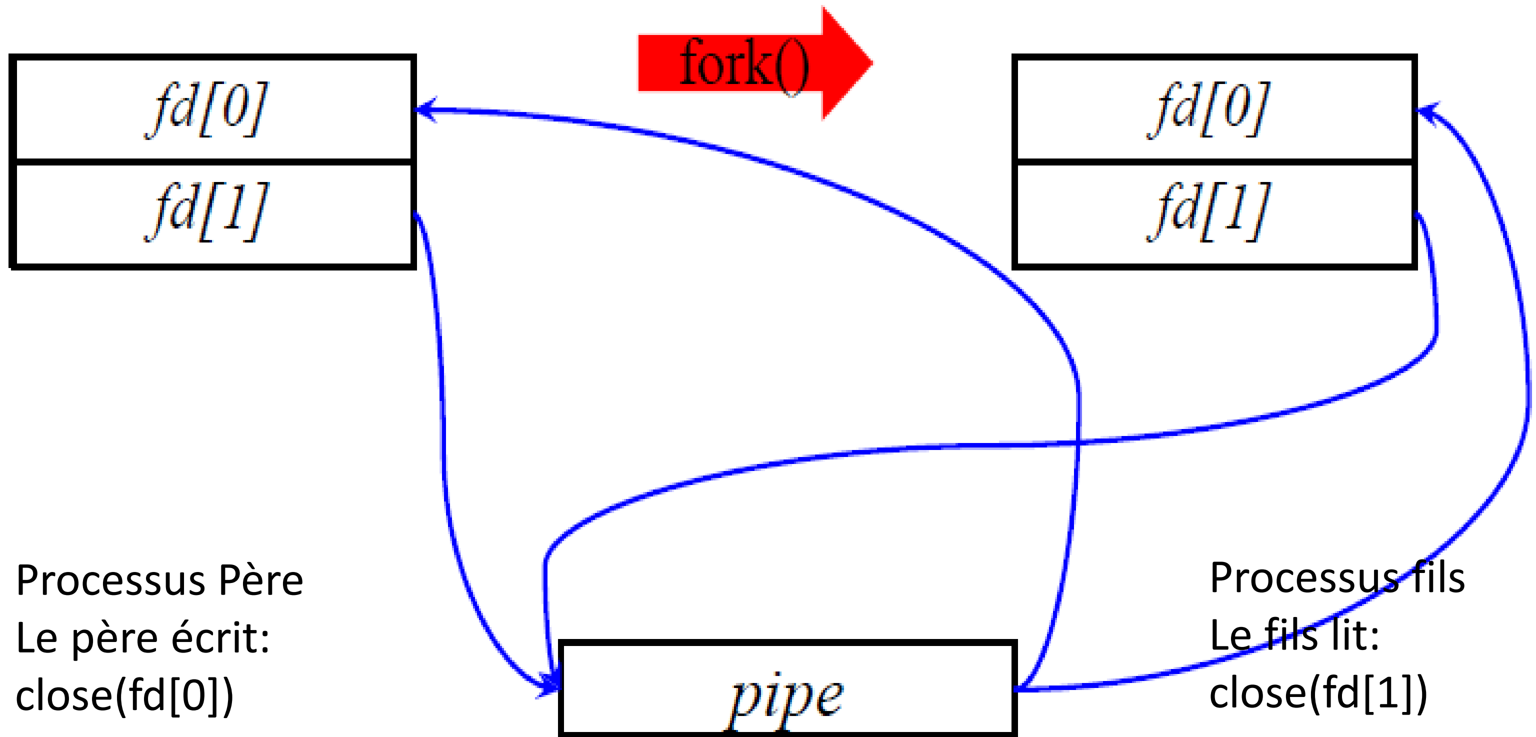
Les Tubes

- #include <unistd.h>
- int pipe(int fd[2]) :

  fd[0] est ouvert en lecture et fd[1] en écriture
- Tout ce qui est écrit dans **fd[1]** est transmis à travers le **tube** (ou **pipe**) à **fd[0]**
- Les descripteurs de fichier (fd[0] , fd[1]) ne sont pas liés à aucun fichier réel, mais à un tampon dans le noyau, dont la taille est spécifiée par le paramètre système PIPE_BUF



Les Tubes



- Le père et le fils partagent les descripteurs de fichier associés au tube
- Quand le père (fils) écrit à une extrémité du tube, le fils (père) peut lire

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Les Tubes - exemple

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

#define    mod 5
```

```
// Initialisation
// Récupération d'une chaîne de car. en
paramètre
int main (int nbarg, char *tbarg[]) {
    int idpro, tube[2],nbcarlus, attente;
    char chaine[19], lgmes, lgrec;

    if(nbarg <= 1) {
        printf("utilisation %s <suite de chaines
de car> \n", tbarg[0]);
        exit(-1);
    }
    srand(getpid());
    pipe(tube);
```

Création d'un tube qui est associé à un couple de descripteurs de fichier *tube[0]* et *tube[1]*

Les Tubes - exemple

```
if (( idpro = fork ()) < 0) {
    perror("creation de processus");
    exit(errno);
}

if ( idpro == 0) { // le fils consommateur
    close(tube[1]); // permet la détection de fin de fichier
    nbcarlus = read(tube[0],&lgrec,1);

    while (nbcarlus > 0) {
        nbcarlus = read(tube[0],chaine,lgrec);
        chaine[lgrec] = 0;
        printf( "\t\t\t\t\tchaine lue : %s\n",chaine);
        nbcarlus= read(tube[0],&lgrec,1);
    }

    close(tube[0]);
    printf("\t\t\t\tje suis le recepteur, num %d et je me termine
\n", getpid());

}
```

Les Tubes - exemple

```
 else { // le pere producteur
   int i, np, etat;

   close(tube[0]); // extrémité de lecture

   for (i=1; i< nbarg ; i++) {
      lgmes = strlen(tbarg[i]);
      write(tube[1],&lgmes,1);
      write(tube[1],tbarg[i],strlen(tbarg[i]));
      printf( "chaine emise : %d eme message\n",i);
      attente = rand()% mod;
      sleep(attente);
   }
   close(tube[1]); // pour déclencher la fin de fichier

   printf (" je suis l'emetteur, num %d et je me termine \n",
getpid());
   np = wait(&etat);
   printf(" Fin du programme\n");
 }/* else */
}/* fin main */
```

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Les Tubes - exemple

```
 ./tube-b bonjour chez vous
chaine emise : 1 eme message
chaine emise : 2 eme message
                           chaine lue : bonjour
                           chaine lue : chez
chaine emise : 3 eme message
 je suis l'emetteur, num  5754 et je me termine
                           chaine lue : vous
                           je suis le recepteur, num  5755
et je me termine
 Fin du programme
```

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Les sémaphores

Définition : un sémaphore est un mécanisme pour la synchronisation ou la protection des sections critiques des processus.

Le **sémaphore** contrôle l'accès à une **ressource partagée** par plusieurs processus, en associant cette ressource avec un **feu** qui permet de s'assurer que pas plus d'**un** processus à un moment peut l'utiliser.

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Les sémaphores à compteur

Un **sémaphore** est une structure de données de type objet, et est constitué :

- une variable entière positive/négative (la valeur du sémaphore)
- une file d'attente
- 2 méthodes :
  - **P(sem)** Proberen → décrémentation de la valeur du sémaphore et blocage éventuel du processus appelant
  - **V(sem)** Verhegen → incrémentation de la valeur du sémaphore et éveil d'un processus (action atomique)
- Instantiation : Init (sem,valeur_initiale)

Les sémaphores à compteur

P(sem) :

-sem.cpt <-- sem.cpt -1
-Si sem.cpt < 0
-alors bloquer le processus appelant
-insérer dans sem.fa le descripteur du processus
-fsi

V(sem) :

-sem.cpt <-- sem.cpt + 1
-Si sem.cpt > 0
-alors extraire un descripteur de sem.fa
-éveiller le processus correspondant
-fsi

Action atomique

Les sémaphores binaires

Un **sémaphore binaire** (ou ***mutex***) contrôle l’accès à la section critique d’un processus ou d’un thread, et sa valeur ne peut être que 0 ou 1. Le sémaphore binaire est un cas particulier d’un sémaphore à compteur.

On suppose que le sémaphore est initialisé à 1 (val_init = 1).

La stratégie générale d’utilisation d’un sémaphore binaire pour contrôler l’accès à une section critique est la suivante :

Init (exmu, val_init)

P(exmu)
SC(R)
V(exmu)
1er passage : exmu.cpt > 0 (val_init = 1)
2ème passage : exmu.cpt = 0

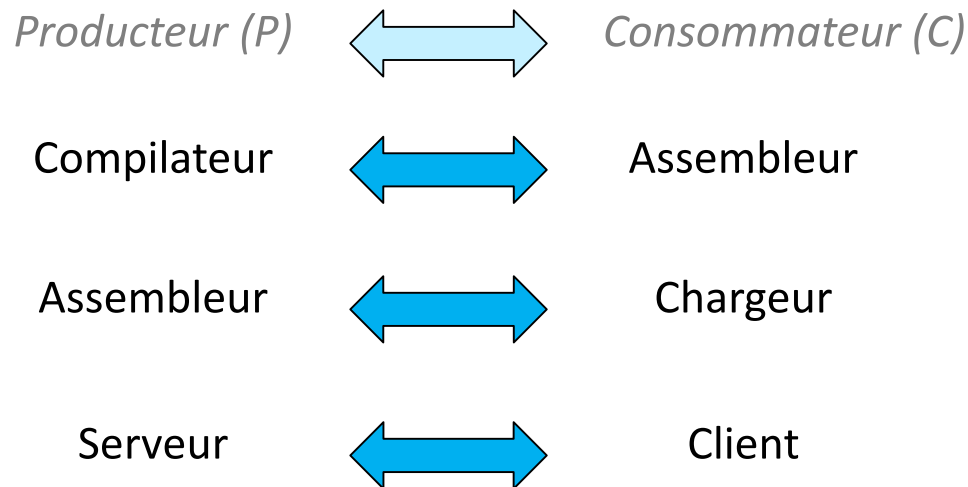
Exemple :

Le problème du producteur-consommateur

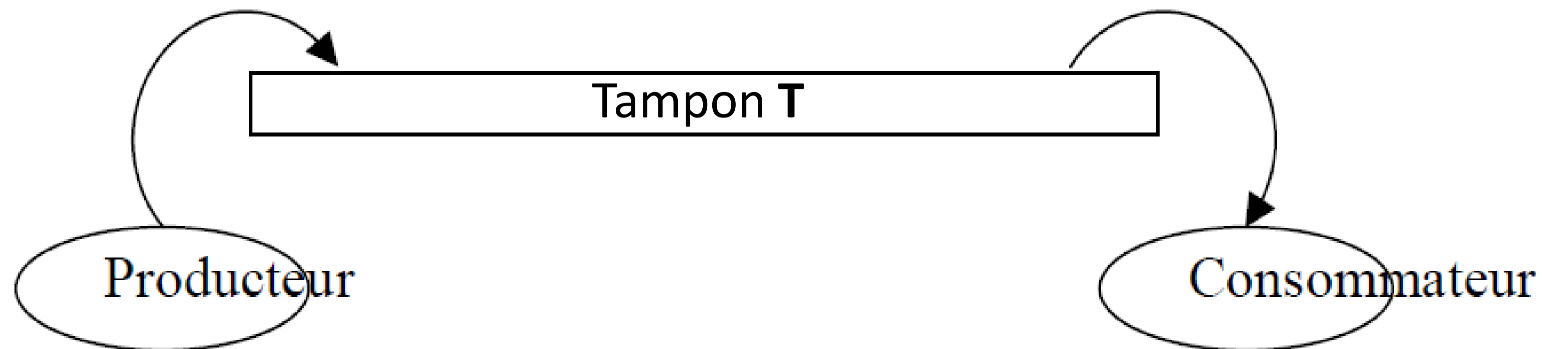
Le problème du producteur-consommateur

Un processus **producteur (P)** produit des informations qui sont consommées par un processus **consommateur (C)**.

Exemples :



Le problème du producteur-consommateur



- Tampon (**T**) borné de taille fixe ***n***
- **T** peut être rempli par **P** et vidé par **C**
- **P** peut produire un élément pendant que **C** consomme un autre
- **P** et **C** doivent être ***synchronisés :***
  - **C** ne tente pas de consommer un élément qui n'a pas encore été produit !
  - **P** ne peut pas insérer un élément dans le tampon s'il est plein !

Le problème du producteur-consommateur

- ***compteur*** est le nombre d'éléments dans le tampon **T**
- **P** doit attendre (s'endormir) si **T** est plein
- **C** doit attendre (s'endormir) si **T** est vide
- Quand **P** voit que **T** était vide et vient d'ajouter un élément dans **T**, il va éveiller **C**
- Quand **C** voit que **T** était plein et vient de retirer un élément, il va éveiller **P**

Producteur :

```
Tque en_vie

produire (R)
insérer(R,tampon)

ftque
```

Consommateur :

```
Tque en_vie

R=extraire(tampon)
      consommer(R)

ftque
```

Le problème du producteur-consommateur

- Comment éviter l'attente active ?
- 2 fonctions :
  - *sleep()*
  - *wakeup()*
- Idée
  - Quand un processus appelle sleep(), il s'endort
  - Si un autre processus appelle wakeup(), il réveille le processus endormi

Le problème du producteur-consommateur

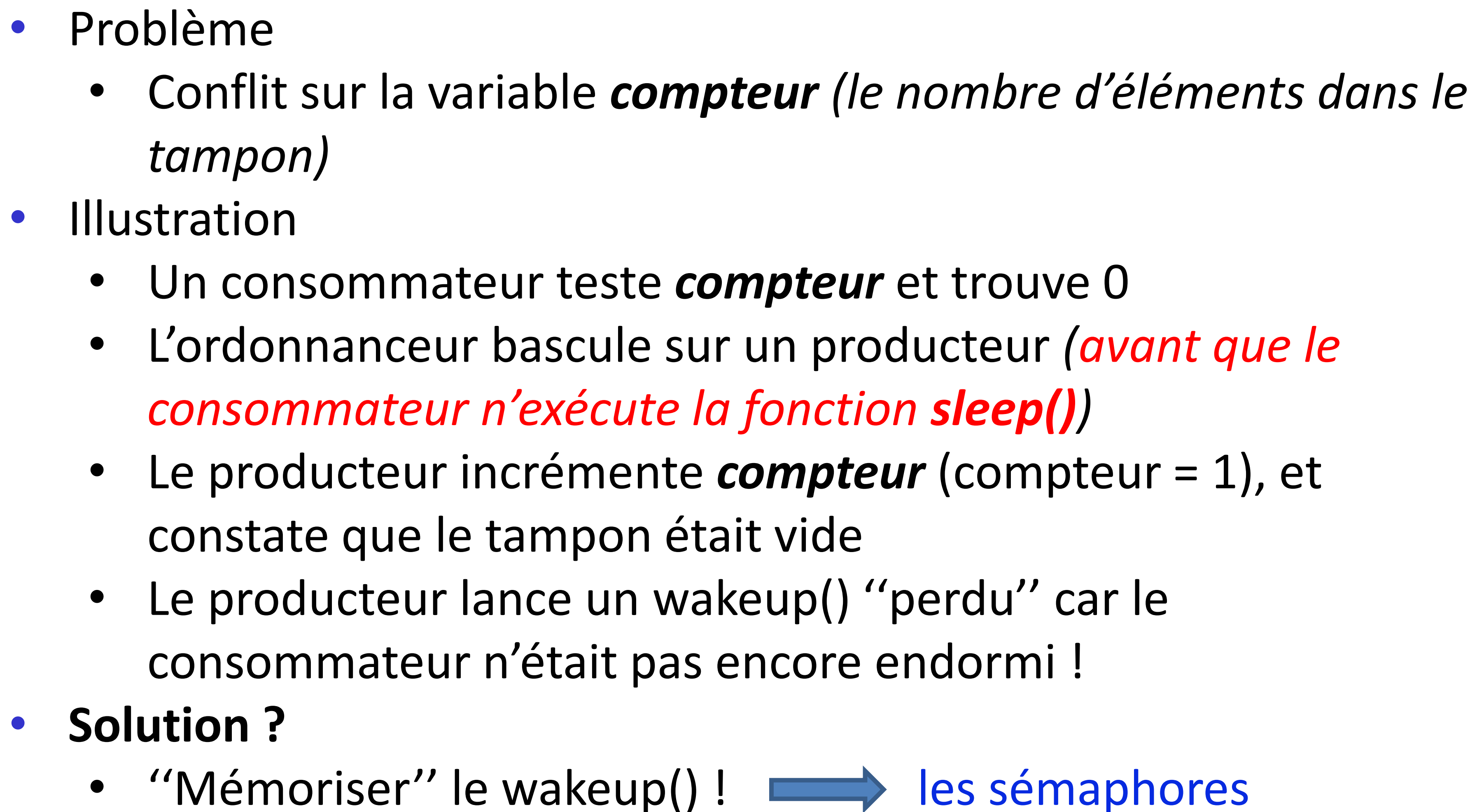
```
#define N 100          /* nbre d'emplacement ds tampon */

int compteur = 0 ;     /* nbre d'objets ds tampon */

void producteur () {
while (VRAI) {
        produire_objet(…) ;
        if (compteur == N) sleep () ;
        mettre_objet(…) ;
        compteur = compteur + 1 ;
        if (compteur == 1)
                wakeup(consommateur) ;
        }
}
```

Le problème du producteur-consommateur

```
void consommateur () {
while (TRUE) {
        if (compteur == 0) sleep() ;
        retirer_objet(…)
        compteur = compteur – 1 ;
        if (compteur == N-1)
                wakeup (producteur) ;
        consommer_objet(…) ;
        }
}
```

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Le problème du producteur-consommateur



Le problème du producteur-consommateur

- Le producteur et le consommateur ne peuvent pas accéder en même temps au tampon (une ressource critique)
- Donc, besoin d’un sémaphore binaire

→ **Résoudre le problème du producteur-consommateur avec les sémaphores**

→ **Gérer l’accès au tampon en utilisant un sémaphore binaire ou une exclusion mutuelle**

Le problème du producteur-consommateur

- Solution avec 3 sémaphores :
  - **full**
    - Un sem. à compteur initialisé à 0 avec une liste vide
    - Contrôle le nombre d'éléments insérés dans le tampon
    - La liste contient les processus consommateurs bloqués
  - **empty**
    - Un sem. à compteur initialisé à *n* avec une liste vide
    - Contrôle le nombre de places vides dans le tampon
    - La liste contient les processus producteurs bloqués
  - **exmu**
    - Un sémaphore binaire initialisé à 1
    - Contrôle l’entrée et la sortie en Section Critique

Le problème du producteur-consommateur

init(full, 0), init(empty, n)

Producteur :

```
Tque en_vie
  produire (R)
  P(empty)

  insérer(R,tampon)

  V(full)
ftque
```

Consommateur :

```
Tque en_vie
  P(full)

  R=extraire(tampon)

  V(empty)
  consommer(R)
ftque
```

Le problème du producteur-consommateur

init(full, 0), init(empty, n), init(exmu,1)

Producteur :

```
Tque en_vie
  produire (R)
  P(empty)
  P(exmu)
  insérer(R,tampon)
  V(exmu)
  V(full)
ftque
```

Consommateur :

```
Tque en_vie
  P(full)
  P(exmu)
  R=extraire(tampon)
  V(exmu)
  V(empty)
  consommer(R)
ftque
```

IUT DE PARIS
UNIVERSITÉ PARIS DESCARTES

Le problème du producteur-consommateur

```
#define BUFFER_SIZE 100
int buffer[BUFFER_SIZE] ;
/* Les sémaphores doivent être initialisées avant usage */
sema_t mutex, full, empty ;
sema_init(&mutex, 1) ; sema_init(&full, 0) ; sema_init(&empty, BUFFER_SIZE) ;
```

```
void producteur() {
  int item ;
  while(TRUE) {
    item = produire_objet() ;
    down(&empty) ;
    down(&mutex) ;
    deposer_objet(item, buffer) ;
    up(&mutex) ;
    up(&full) ;
  }
}
```

```
void consommateur() {
  int item ;
  while(TRUE) {
    down(&full) ;
    down(&mutex) ;
    item = retirer_objet(buffer) ;
    up(&mutex) ;
    up(&empty) ;
    consommer_objet(item) ;
  }
}
```