II2

Systèmes d'exploitation et Programmation Concurrente

Chap. II

GESTION des PROCESSUS et des THREADS

Partie 1. Les PROCESSUS

Introduction

✓ Un SE *tourne en permanence* après le démarrage de l'ordinateur (ou tout autre dispositif/unité informatique).

- Il permet le développement et l'exécution de nouveaux programmes (sur processeur)
- Décrire le fonctionnement d'un SE ? → **Trop compliqué**
  - ***Concept fondamental : la décomposition !***

Notion de Processus (1/3)

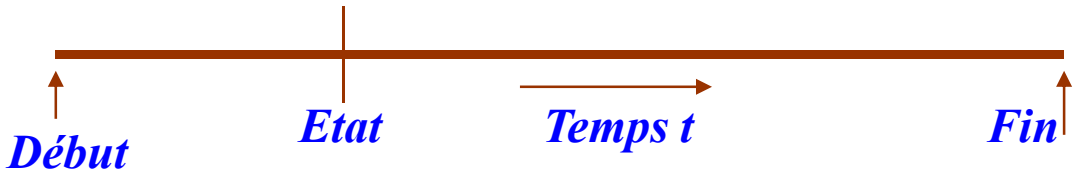
- ✓ ***Abstraction du processeur?*** → *Notion de* ***Processus?***
- ✓ ***Qu'est-ce qu'un programme?***
  - ❑ Ensemble de modules sources/objets
  - ❑ Résultat de l'édition de liens (actions manipulant des données)
    - ⇨ *Description* **Statique (Code + données)**
- ✓ ***Que désigne le terme processeur?***
  - ❑ Entité matérielle capable d'exécuter des instructions
  - ❑ ''parfois '' aussi entité logicielle (interpréteur, …)

Notion de Processus (2/3)

✓ *Qu'est-ce qu'un processus ?*

❑ *Entité **dynamique** représentant l'exécution d'un programme sur un processeur*

⇨ *créée à un instant donné, a un état qui évolue au cours du temps et qui disparaît, en général, au bout d'un temps fini*



Notion de Processus (3/3)

✓ **Définitions technique de Processus:**

- **Unité de structuration**
- Un processus a besoin de ressources (mémoire, CPU, données, unités d'E/S) pour s'exécuter → **Abstraction du SE pour la**
  - ⇨ allocation de la **MC** (code, données, pile) → **Espace d'adressage**
  - ⇨ allocation du processeur (PC, …)

✓ ***Exemples de processus:***

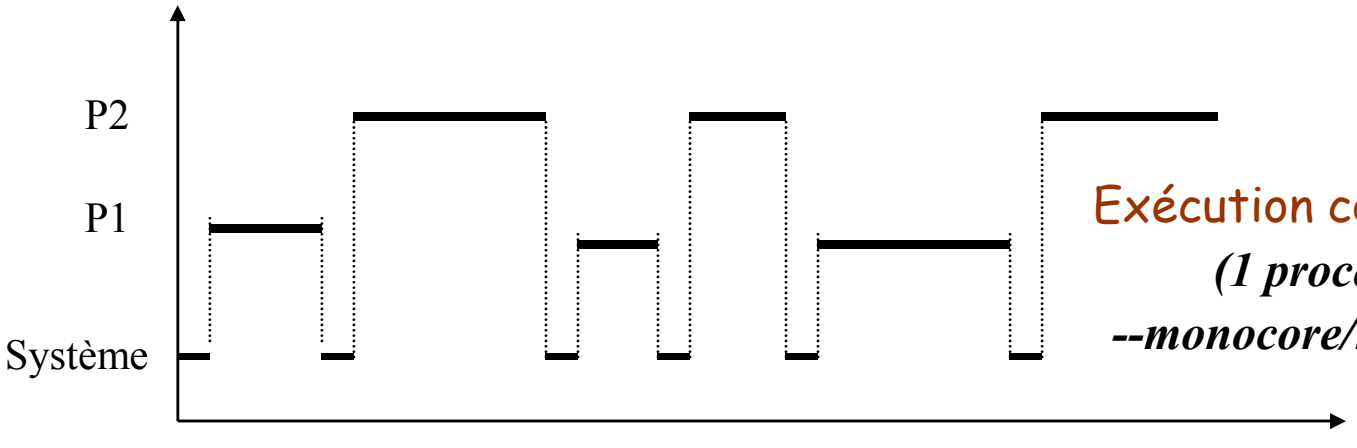
- echo $PATH // Exécution d'une commande
- gcc -o TP TP.c // Compilation d'un programme c
- java TP_sys // Exécution d'un programme
- firefox //Navigateur web

Parallélisme et Concurrence

- ✓ Deux processus P1 et P2 en mémoire centrale (prêts à s'exécuter)
- ✓ **Comment mettre en œuvre l'exécution (concrète) de P1 et P2 ?**
  - ❑ **2 possibilités (extrêmes) + 1 (intermédiaire)**

Exécution séquentielle
(1 processeur)

Exécution parallèle
(2 processeurs)



Exécution concurrente
(1 processeur
--monocore/multicores)

Parallélisme et Concurrence (suite)

- ✓ Parallélisme explicite → structuration d'applications en tâches concurrentes et communicantes

- ✓ Les SEs supportent différents types de tâches; les plus communs:
  - ❑ Processus
  - ❑ Threads

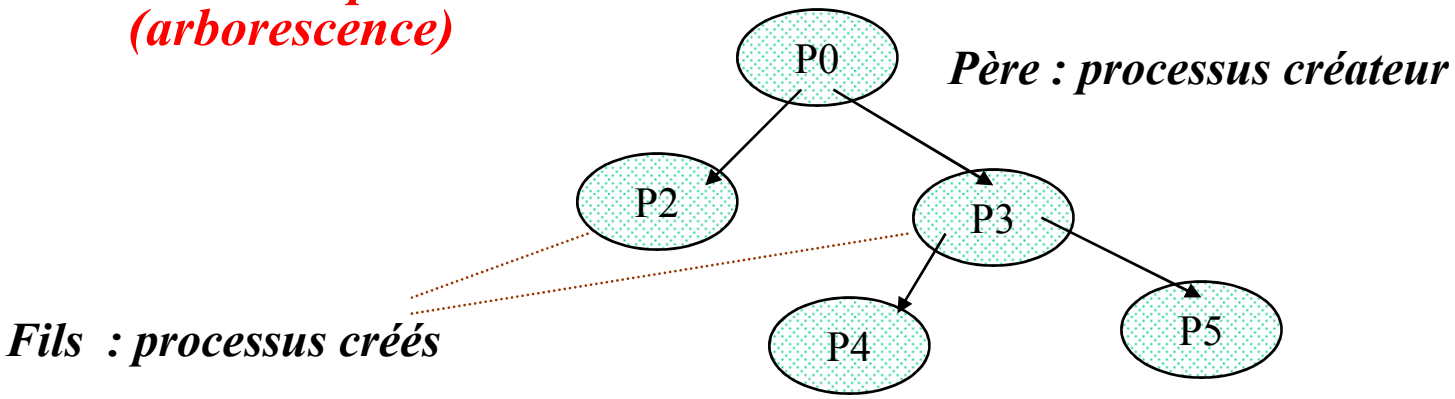
- ✓ Les SEs implémentent le multitâche selon les caractéristiques du processeur:
  - ❑ Monoprocesseur (mono core)
  - ❑ Mono core avec support de multithtread
  - ❑ Multicore et multithread

Concurrence?

- ✓ **Multiprogrammation / Concurrence (Pseudo-parallélisme) :**
  - ❑ Un SE doit, en général, traiter plusieurs processus en même temps
  - ❑ ***Entrelacement des exécutions*** (simuler une exécution parallèle)
    - ⇨ A tout moment le SE ne traite qu'*un seul processus à la fois*, il s'interrompt et passe au suivant.
    - ⇨ La *commutation étant très rapide* → Illusion d'un traitement simultané
- ✓ *Les processus utilisateurs sont lancés par un interprèteur de commandes.*
- ✓ Ils peuvent eux-mêmes lancer ensuite d'autres processus

Structuration des processus

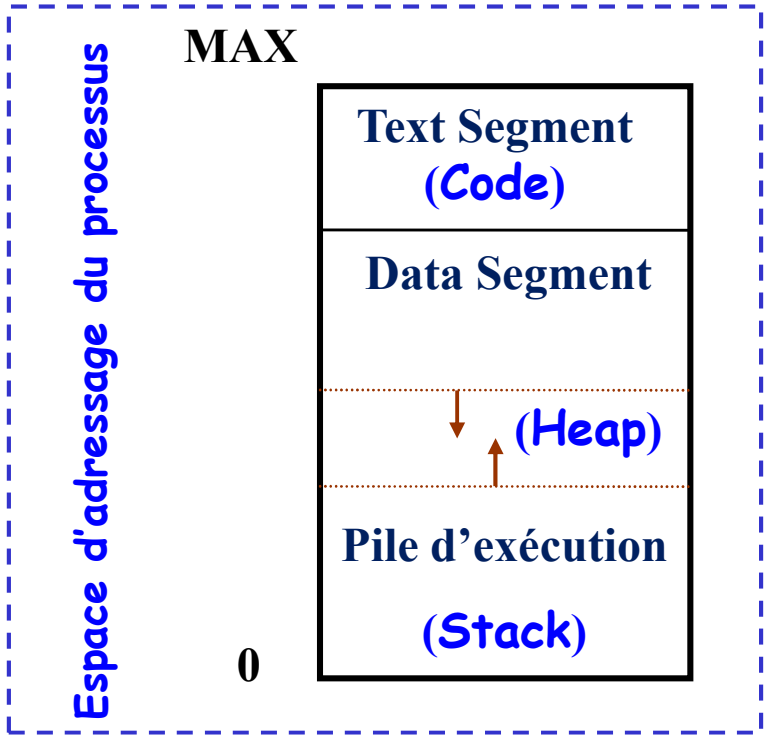
Hiérarchie des processus (arborescence)



✓ Au lancement du système, il n'existe qu'un seul processus (P0): *INIT*

Processus en Mémoire Centrale

✓ Défini par *3* ***segments*** *code, pile, données* et d'un ***contexte***.



Segments d'un processus

Processus en Mémoire Centrale (suite)

- ***Segment code (Text) -- Lecture seulement***
  - Contient les instructions
  - Invariant (toute la durée d'exécution du processus).
- ***Segment données (Data) -- Lecture/Ecriture***
  - Contient les variables globales + données statiques (Constantes), qui sont initialisées à la compilation
- ***Pile d'exécution (Stack) -- Lecture/Ecriture***
  - Un programme modulaire (fonctions/procédures)
  - *Contient les variables échangées + variables locales*
  - *File d'attente gérée selon LIFO*

- **Note importante:** La protection de la mémoire centrale se fait via les limitations de chaque segment (voir plus loin chap. gestion de la mémoire centrale)

Notion de Ressources

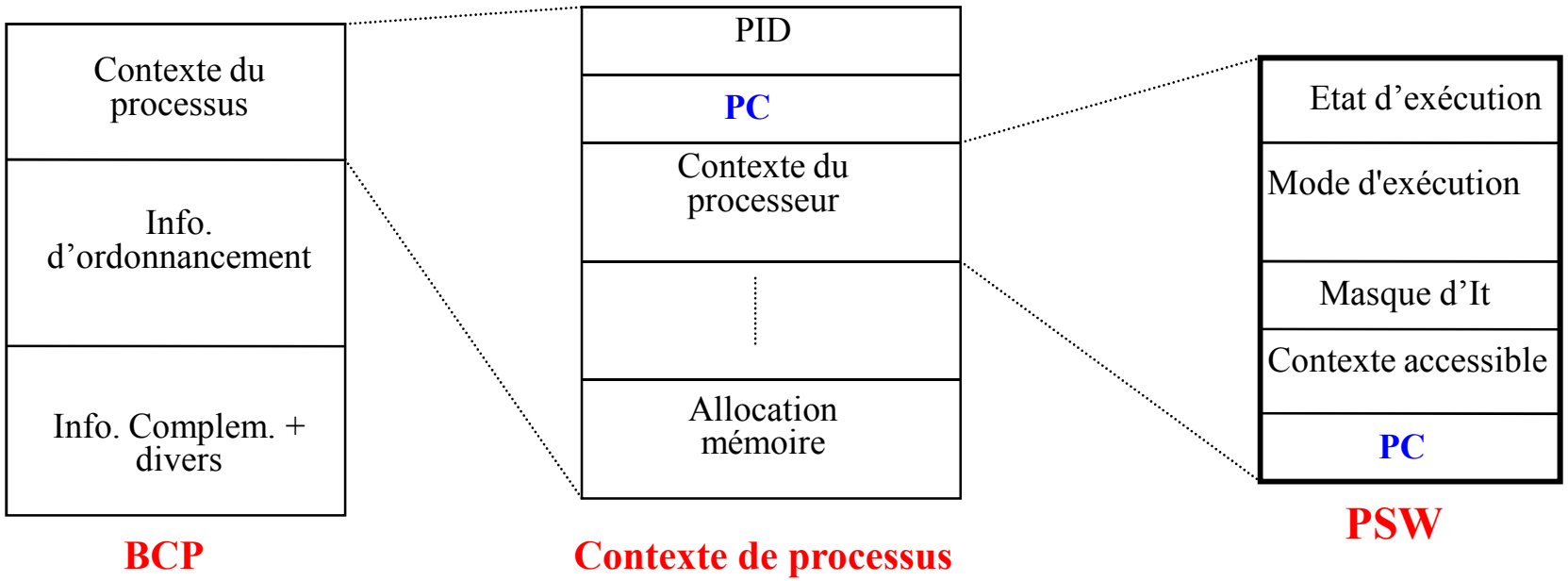
- **Ressource** = *toute entité dont a besoin un processus pour s'exécuter*
  - Processeur physique, mémoire centrale, périphériques, données momentanément indisponible, événement, etc. …
- Caractéristique: **Nombre de points d'accès**
  - Nombre de processus possédant la ressource à un instant donné
  - Si un seul point d'accès, la ressource est dite critique, processus en exclusion mutuelle.

Contexte du Processeur

- ✓ **Hypothèses :** 1 seul processeur + plusieurs processus le processeur est réservé à l'usage d'un seul processus
- ✓ ***Contexte*** = Contenu des registres adressables/spécialisés --**Mot d'état (PSW - Program Statuts Word)**
  - ⇨ Etat d'exécution : Actif/Attente
  - ⇨ **Mode d'exécution (Protection matérielle):** Système/utilisateur
  - ⇨ Masque d'interruption
  - ⇨ Contexte accessible en mémoire

Contexte d'un Processus (1/2)

- **Contexte d'un processus:** *ensemble d'informations nécessaires à la gestion d'un programme en cours d'exécution (code + données)*



BCP --Contenu

- ✓ **BCP –Bloc de contrôle de Processus (PCB –Process control bloc):**
  - ⇨ **Structure de données associée à l'exécution de chaque programme**

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

BCP --Contenu

- **Etat du processus**: … (voir plus loin)
- **Compteur ordinal** –*CO* (Program counter –PC): indique l'adresse de l'instruction suivante à exécuter dans le processus.
- **Les registres CPU**: Les registres varient en nombre et en type, en fonction de l'architecture informatique.
  - ⇨ dont les accumulateurs, les registres d'index, pointeurs de pile et les registres à usage général
  - ⇨ ces registres doivent être sauvegardés si une interruption se produit, afin d'assurer au processus de se poursuivre correctement par la suite

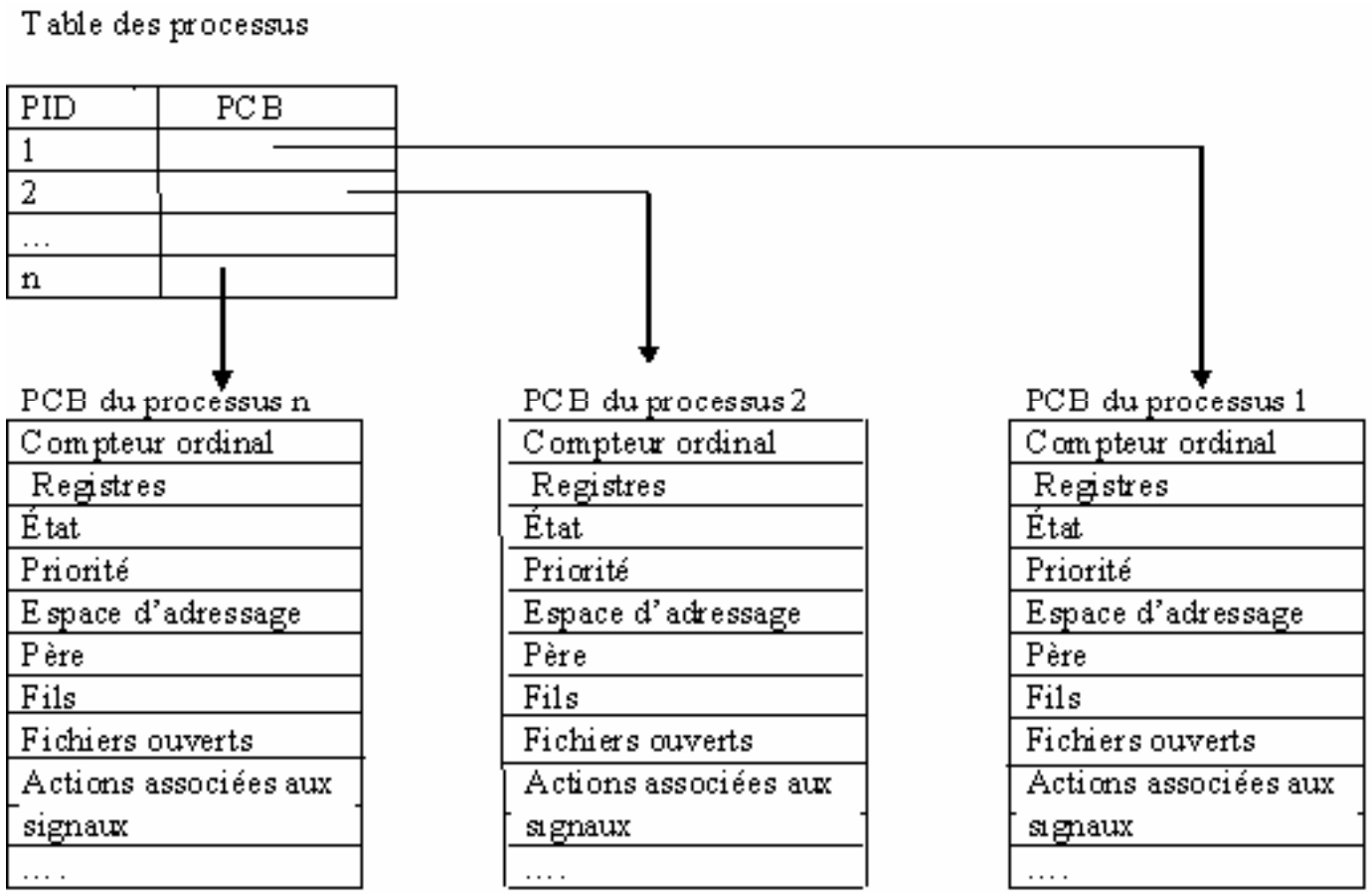
BCP –Contenu (suite)

- **Informations relatives à l'ordonnancement du CPU:** Comprennent la priorité du processus, les pointeurs vers des files d'attente d'ordonnancement, et les autres paramètres d'ordonnancement.

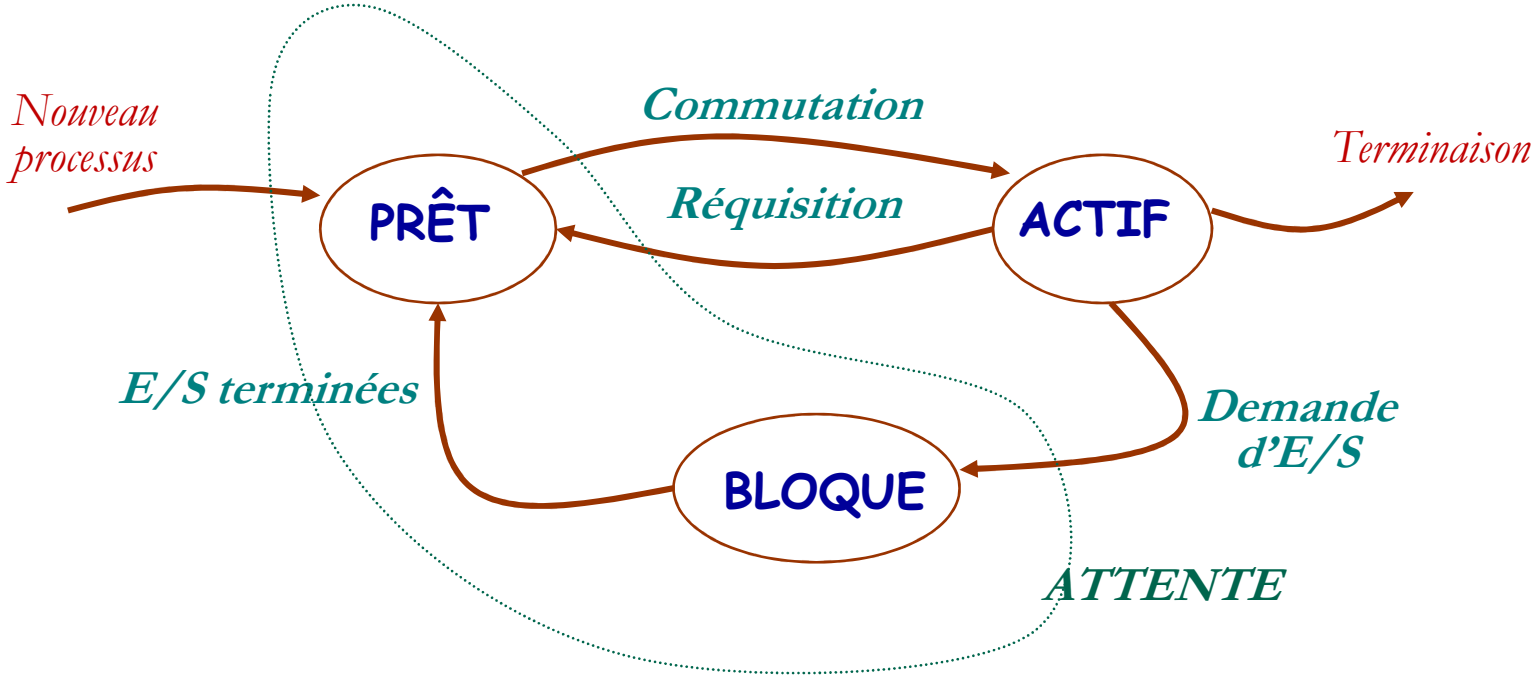
- **Informations relatives à la gestion de la mémoire:** peuvent inclure les tables de page, les tables de segments... (à voir chap. MC)

- **Informations d'état des E/S:** Comprennent la liste des périphériques E/S alloués au processus, la liste de fichiers ouverts, ...

Table des processus et PCBs



Etats d'un Processus -Trois états



- ✓ **Commutation**: effectuée par le scheduler et dispatcher

Etats d'un Processus (suite)

✓ Etat Actif: 2 modes d'exécution (user/kernel)

✓ 3-4 transitions

- **Actif → Bloqué** action volontaire de demande de ressource(s)
  - ⇨ Freq. E/S
- **Bloque → Prêt** action extérieure au processus (ressource disponible)
- **Prêt ⇔ Actif** Décision de l'allocateur du processeur

INTERRUPTION (It) --Définitions

1. Une It est un signal envoyé, de manière asynchrone, au processeur, qui le force à suspendre son activité en cours au profit d'une autre.
2. A chaque It correspond un vecteur d'Its, un emplacement mémoire, contenant une @. L'arrivée de l'It provoque le branchement à cette @.
3. La suspension d'un processus se fait au 1er ***point observable/interruptible***.

✓ **Une It peut être :**

- **Externe** à l'activité en cours: réaction du SE à des événements asynchrones externes.
- **Interne (Trap)** : liée à l'exécution de l'instruction en cours
  - ***Exception (déroutement)*** : exécution d'une opération illicite
  - ***Appel Système (SC/SVC)*** : nécessite l'appel d'une fonction du SE

Gestion des Interruptions

- **Linux:** chaque interruption (matérielle (IRQ) ou logicielle (trappe)) est repérée par un entier sur 8 bits, le vecteur d'interruption.
  - ⇨ **0 --31:** interruptions non masquables, exceptions;
  - ⇨ **32 -- 47:** interruptions masquables levées par les périphériques (IRQ)
  - ⇨ **128:** appels système
  - ⇨ **autres:** utilisables pour les trappes autres que celles émises par le système
- ⇨ Une table des vecteurs d'interruption est placée en mémoire centrale et associe à chaque valeur l'adresse d'une routine de gestion.
- ✓ La CPU vérifie à chaque instruction si une interruption a été reçue.

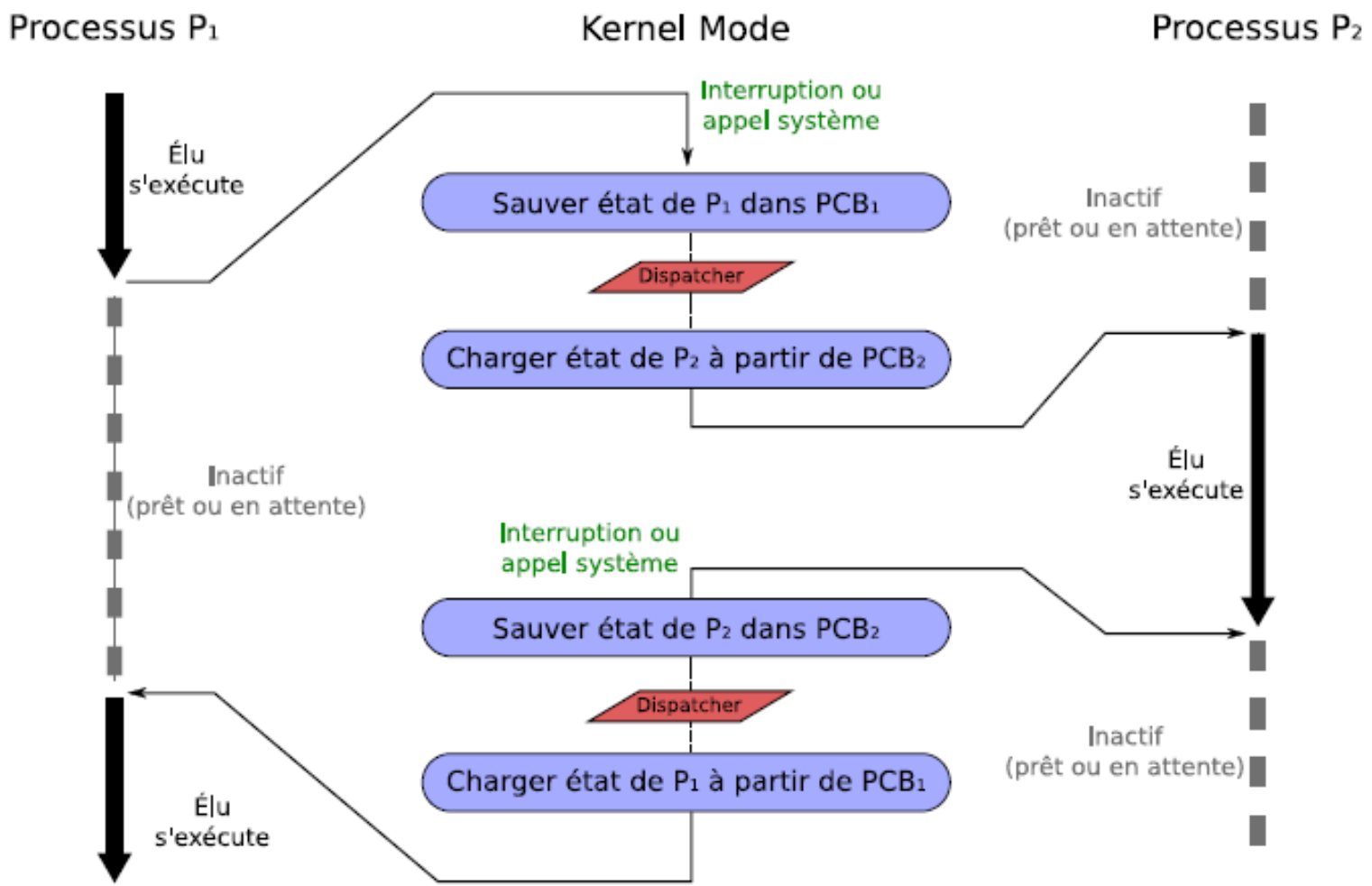
Mécanisme de Commutation (1)

- ***Commutation de processus :***
  - Le passage d'un processus à un autre
  - Elle est réalisée par un ordonnanceur (**scheduler**) en collaboration avec le **dispatcher** du système (à voir chap. ordonnancement)
  - Cet ordonnanceur est activé par des interruptions (de l'horloge, de disques, ou de terminaux).

Mécanisme de Commutation (2)

- Pour pouvoir *retarder*, et dans certains cas *annuler, la prise en compte d 'It* on utilise le *masquage* et le *désarmement* :
  - Protéger un processus contre les Its les moins prioritaires
  - Retarder la commutation jusqu'à ce que le processus le plus prioritaire ait terminé son exécution ou soit lui même interrompu par un autre plus prioritaire.
- ***Commutation de processus = commutation des contextes de processus***
- Enchaînement indivisible des opérations suivantes :
  - *Sauvegarde (au min) du mot d'état* (à emplacement particulier de la mémoire)
  - *Chargement d'un autre mot d'état* (depuis un emplacement spécifique de la mémoire vers le processeur).
- Le nouveau processus peut alors être exécuté à partir de l'état où il se trouvait lorsqu'il a été lui même interrompu.
- Cette commutation de contexte ne peut s'effectuer que lorsque le processeur se trouve dans un état interruptible

Mécanisme de Commutation (3) -- Schéma

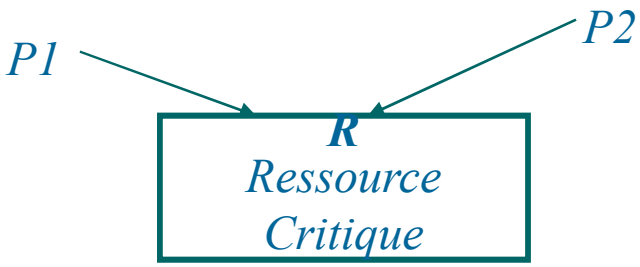


Interaction entre Processus (1)

- ✓ Il existe entre les processus un certain nombre de relations, appelées **INTERACTIONS**;
  - ❑ ces interactions peuvent être de compétition ou de coopération
- ✓ **Compétition**
  - ❑ Situation dans laquelle plusieurs processus doivent utiliser simultanément une ressource à accès exclusif (1 seul processus à la fois), encore appelée ***ressource critique***.
  - ❑ **Exp.**
    - ⇨ L'usage du processeur (pseudo-parallélisme)
    - ⇨ Accès à un périphérique (imprimante)

Interaction entre Processus (2)
--Compétition: exclusion mutuelle

- ✓ 2 processus en compétition sont dits en **exclusion mutuelle** pour cette ressource critique .
  - ❑ ***Une solution possible :*** Faire attendre les processeurs demandeurs que l 'occupant actuel ait fini (FIFO)

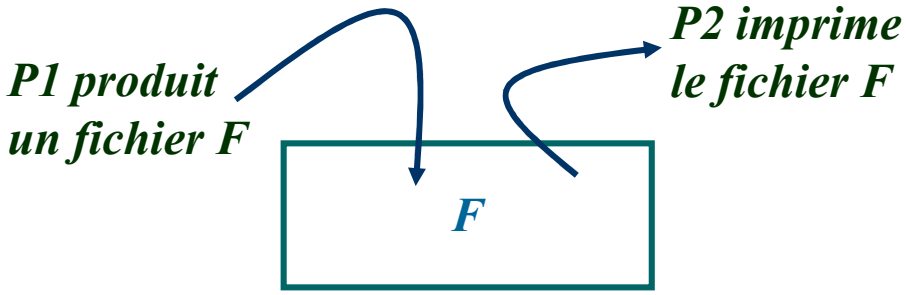


Ordre d'utilisation : Indifférent
P1; P2 ou P2; P1

Interaction entre Processus (3)
--Coopération

✓ **Coopération**

- Situation dans laquelle plusieurs processus ***collaborent à une tâche commune*** et doivent se ***synchroniser*** pour réaliser cette ***tâche*** .
- Deux processus qui coopèrent peuvent également se trouver en exclusion mutuelle pour une ressource commune.



P2 ne peut s'exécuter que si:
P1 a terminé son exécution
P1 < P2

Interaction entre Processus (4) --Synchronisation

- Un processus doit attendre qu'un autre processus ait franchi un certain point de son exécution point de synchronisation
- Imposer des **contraintes de synchronisation** aux processus
  - ⇨Précédence des processus
  - ⇨Conditions de franchissement de certains points critiques

Interactions entre Processus (5) -- Attente

✓ Relations entre deux processus ⇨ Faire attendre un processus

- Comment réaliser cette attente ?

✓ **Solution 1: ATTENTE ACTIVE**

P1	**P2**
while (ressouce occupee)	*Ressource occupee = True;*
{ };	*Utiliser Ressource;*
Ressource occupee = True;	*Ressource occupee = False;*
….	

- Très peu économique si pseudo-parallélisme
- Difficulté d'une solution correcte (chap. IPC)

Interactions entre Processus (6) -- Attente (suite)

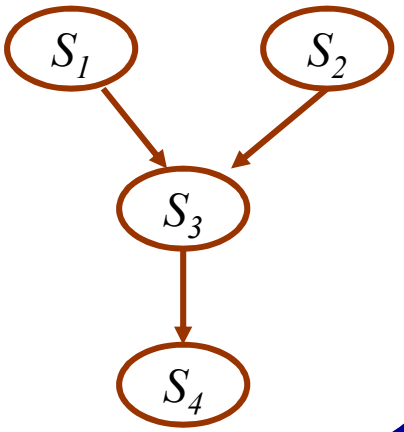
✓ **Solution 2 : ATTENTE PASSIVE –BLOCAGE de processus**

❑ Processus bloqué : attente d'une ressource non disponible jusqu'a son réveil *explicite* par un autre processus

Conditions d'Exécution Concurrente (1)

- ✓ Augmenter le degré de multi-programmation et donc le taux d'utilisation du processeur ?
  - ❑ Etablir les contraintes de précédence (relation <)
  - ❑ Construire un graphe de précédence : graphe acyclique où
    - ⇨ chaque nœud, S_i, représente une instruction
    - ⇨ chaque arc S_iS_j signifie que l'instruction S_j ne pourra s'exécuter que si S_i ait terminée son exécution.
  - ❑ **Exemple :** Soit la séquence d 'instructions suivante :

S1	lire(x)
S2	lire(y)
S3	z = x * y
S4	écrire(z)



Conditions d'Exécution Concurrente (2)

- **Notation :** Soient les deux ensembles suivants :
  - $R(S_i)$ -- ReadSet = Ensemble de toutes variables de S_i qui ne changent pas après exécution de S_i.
  - $W(S_i)$ -- WriteSet = Ensemble de toutes les variables référencées dans S_i qui changent de valeurs après exécution de S_i)
- **Illustration** (exemple précédent) :
  - $R(S_1) = \varnothing$ $W(S_1) = \{x\}$ $R(S_2) = \varnothing$ $W(S_2) = \{y\}$
  - $R(S_3) = \{x, y\}$ $W(S_3) = \{z\}$ $R(S_4) = \{z\}$ $W(S_4) = \varnothing$
- **Conditions de Bernstein :**
  - 2 instructions (processus) S_1, S_2 ; ils peuvent s'exécuter en concurrence si :
  - $\boldsymbol{R(S_1) \cap W(S_2) = R(S_2) \cap W(S_1) = W(S_1) \cap W(S_2) = \varnothing}$
  - **N.B.** $R(S_i) \cap W(S_j)$ n'est pas forcément vide; exp. x = x +1;

Les processus sous Unix
Principes généraux (1)

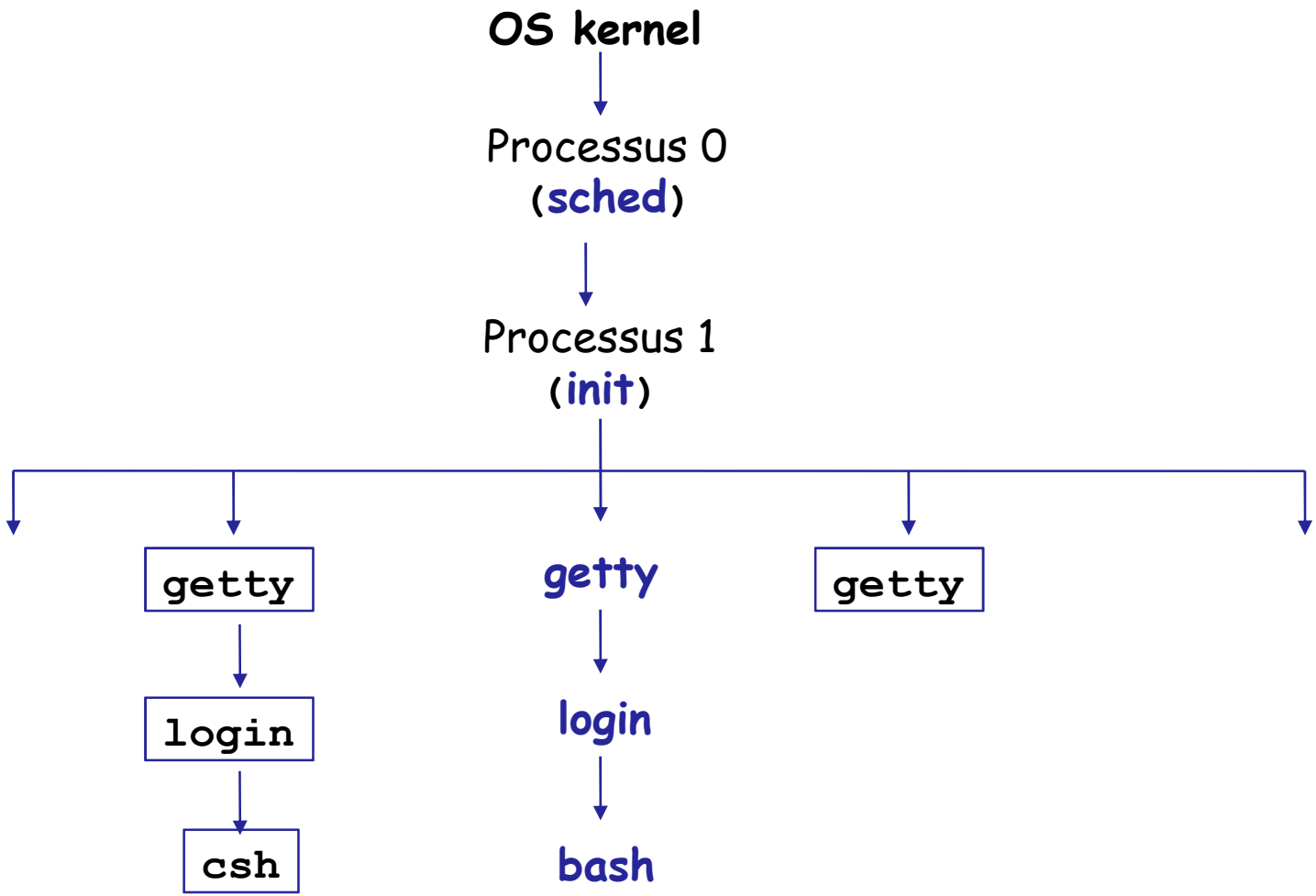
- ***Définition technique de processus:***
  - Abstraction du SE pour l'allocation mémoire et l'allocation du processeur
- ***Processus dans Unix:***
  - Taper une commande, exécuter un programme (sous le shell):
    - La création sans le savoir d'un nouveau processus dont la durée de vie est celle de la fonction demandée.
  - Appel système (au niveau de l'API)
    - Création par appel à une fonction spéciale ***fork*** (voir plus loin)

Les processus sous Unix --Principes généraux(2)

Identification de processus

- *Numéro de processus unique:* ***PID*** *(Process IDentification)*
  - *La fonction* ***int getpid(void)****: indique le numéro du processus qui l'exécute*
  - *La commande* ***ps****: donne la liste des processus (voir options avec man)*
  - *La commende* ***top****: montre l'activité du processeur*
- *Numéro du processus parent:* ***PPID*** *à partir duquel ce processus est lancé*
  - *La fonction* ***getppid(void)****: obtenir l'id du processus père*
- *Numéro user: réel ou effectif (propriétaire --qui a écrit le programme)*
  - *Les fonctions* ***int getuid(void)*** *ou* ***int geteuid(void)***
- *Numéro groupe (GID): la fonction fetpgrp(void)*

Diagramme de démarrage des processus Unix



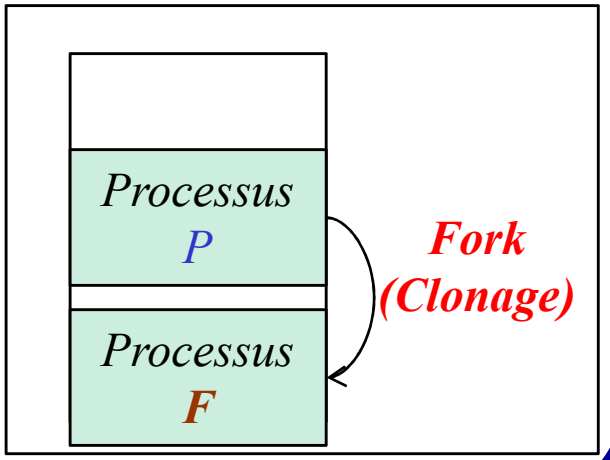
Les processus Unix les plus importants

- **init**: racine de tous les processus, lancé au démarrage de la machine (invoqué après le bootstrap) et il est le responsable de démarrage de tous les autres processus
  - /sbin/init
  - **Init** utilise le fichier **inittab** et les répertoires **/etc/rc?.d**
- **getty**: processus de login qui gère les logins de sessions

Création d'un processus Unix

La création d'un nouveau processus est un mécanisme fondamental d'Unix

- *Appel système* ***pid_t fork(void):*** *dupliquer le processus appelant en retournant*
  - *0 dans le processus fils (→ opération réussie)*
  - *Numéro du processus fils dans le père*
  - *-1 échec de création*
- *Exécution parallèle/concurrente des deux processus*
  - *Seules différences: pid, ppid, @*

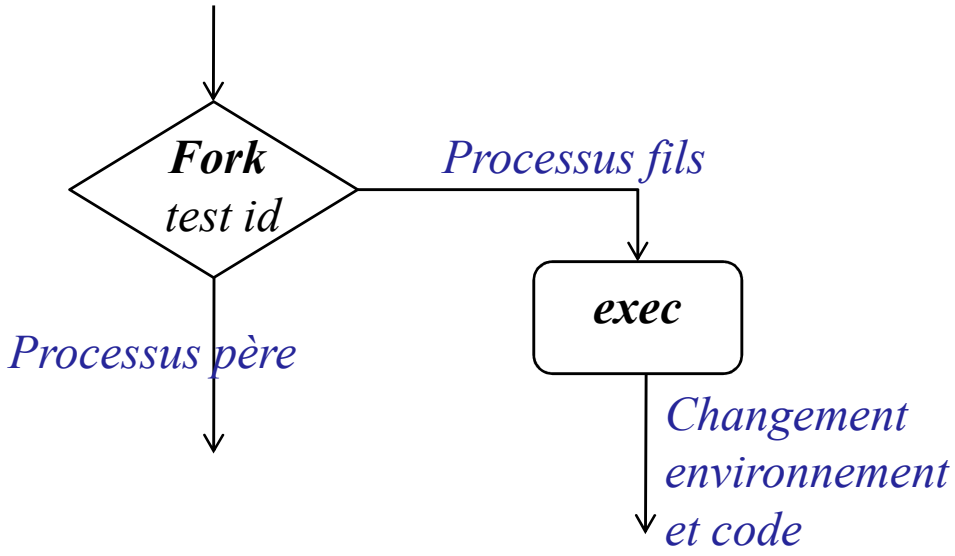


Création d'un processus Unix (2)

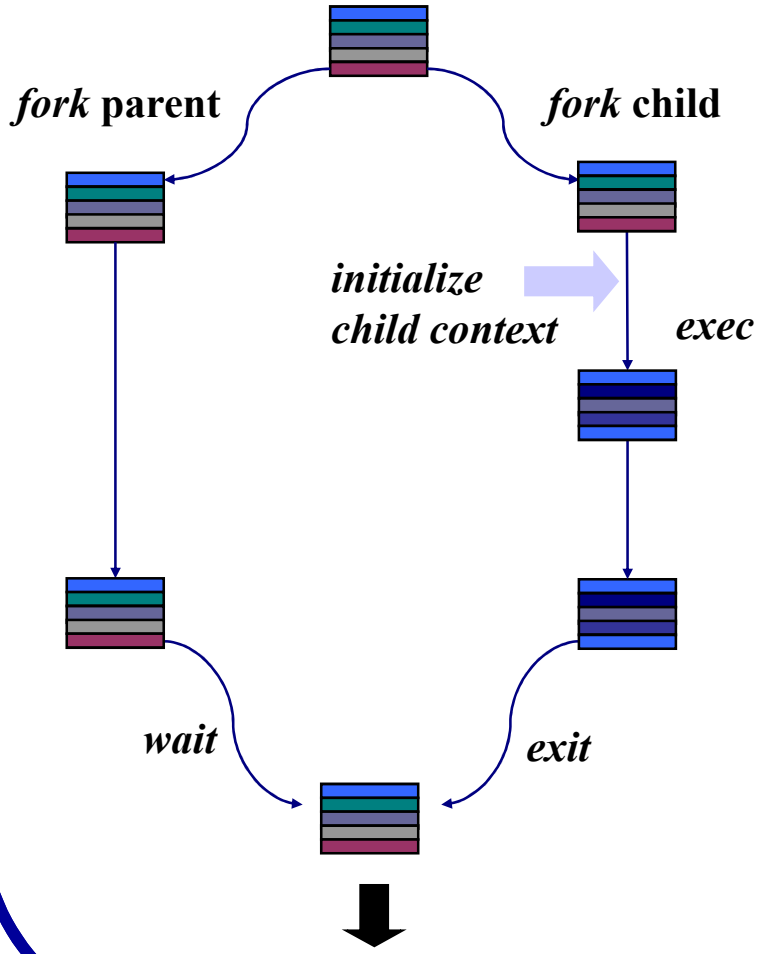
```
#include<errno.h>
#include<stdio.h>
#include<unistd.h>

int main(void) {
    int pid = fork();   /* création d'un nouveau processus fils */
    if  (pid == -1) perror("Echec sur le lancement de fork");
    else if  (pid == 0)    /* Processus fils */
            printf(" Je suis le fils, mon PID est %d\n", getpid());
        else /* Processus pere */
            printf(" Je suis le pere,  de PID %d, et mon fils est de PID %d\n", getpid(), pid);
     printf(" FIN!, je suis %d \n", getpid());
    return 0;
}
```

Création d'un processus Unix (3)
Fork/exec

- ***Le démarrage d'un nouveau processus passe par***
  - *Création du nouveau processus (fork)*
  - *Test pour identifier les deux processus et distinguer le père du fils*
  - *Décider de faire exécuter un nouveau code différent du père:*
    - *Substituer le code du fils par le code qu'on désire exécuter au moyen de l'appel système* ***exec***



Fork/exec (suite)



int pid = fork();

Create a new process that is a clone of its parent.

exec("program" [, argvp, envp]);*

Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);

Exit with status, destroying the process.

int pid = wait(&status);*

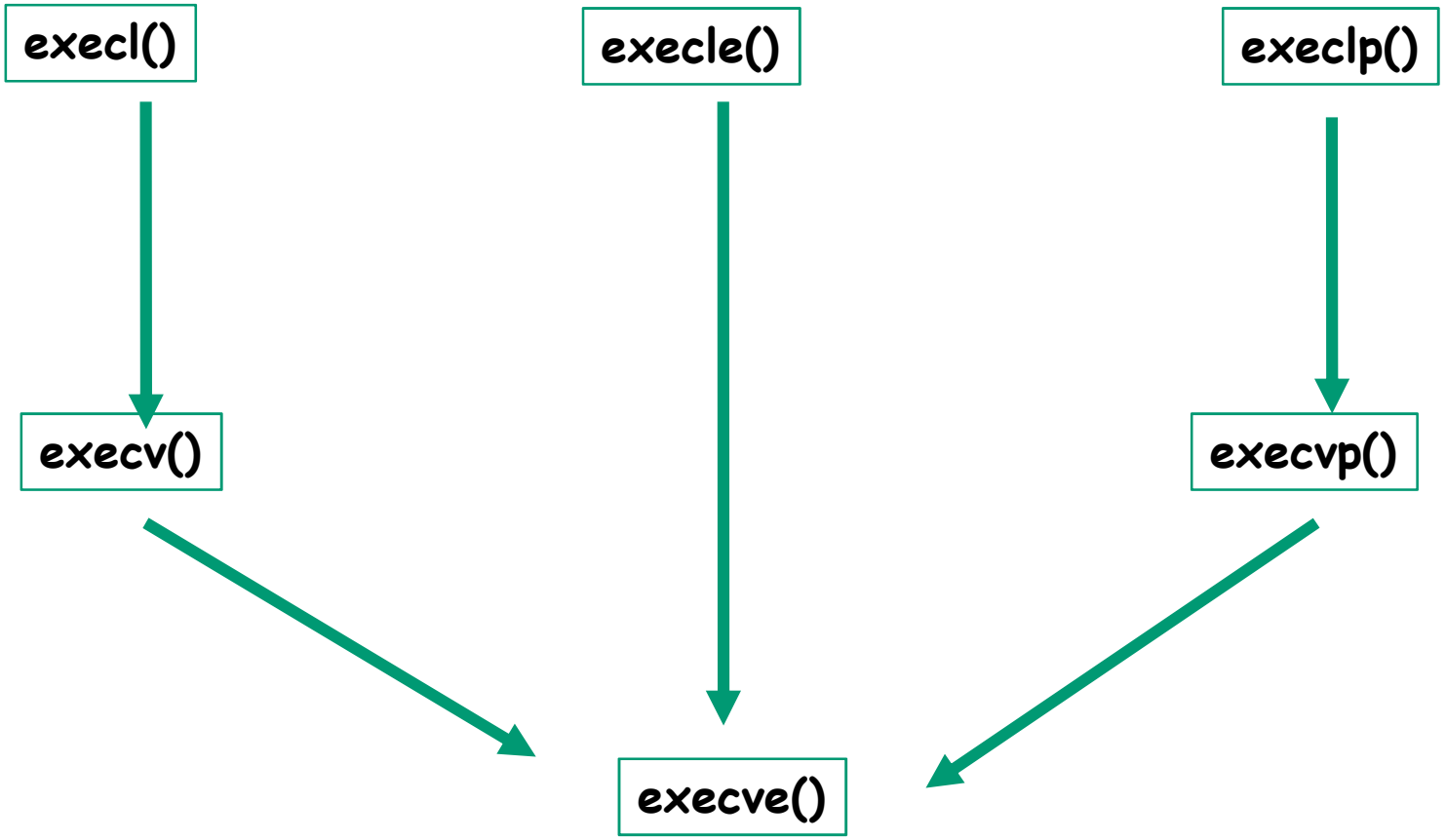
Wait for exit (or other status change) of a child.

Variantes exec(..)

- Il existe 6 variantes de la fonction exec, qui diffèrent sur les paramètres passés.

```
int execl (const char *path, const char *arg, ... );
int execlp (const char *file, const char *arg, ... );
int execle (const char *path, const char *arg
          , ..., char *const envp[] );
int execv (const char *path, char *const argv[] );
int execvp (const char *file, char *const argv[] );
int execve (const char *filename, char *const argv [],
       char *const envp[] );
```

Variantes exec(..)



Fork/exec –Variante d'*exec*

- **int execve (char *prog, char **argv, char **envp);**
  - prog – full pathname of program to run
  - argv – argument vector that gets passed to main
  - envp – environment variables, e.g., PATH, HOME
- Generally called through a wrapper functions
  - **int execvp (char *prog, char **argv);**
    - Search PATH for prog, use current environment
  - **int execlp (char *prog, char *arg, ...);**
    - List arguments one at a time, finish with NULL

Exemple de excelp

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

int main()
{       pid_t pid;
        /* fork a child process */
        pid =fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");   return 1;
        }
        else if (pid == 0) { /* child process */
                printf("Child Process says Hello !\n");
                execlp("/bin/ls","ls",NULL);
                 }
                 else { /* parent process */
                    printf("Parent Process says Hello !\n");
                    wait (NULL) ; //waiting his child to complete
                    printf("Child Complete");
                 }
        return 0;}
```

Exercice en classe

Soit le code suivant : test.c

```
main() {
    int c = 5;
    int child = fork();
    if (child == 0) c += 5;
    else {
        child = fork();
        c += 10;
        if(child)  c += 5;  }
    }
```

1. Combien y a t'il de copies de c.
2. Quelles sont leurs valeurs à l'exécution du programme.
3. On remplace if(child) c+=5; par

   If (child)

   execlp(../test.,../test.,NULL;

   Que se passe-t-il ?

Terminaison d'un processus Unix (1)

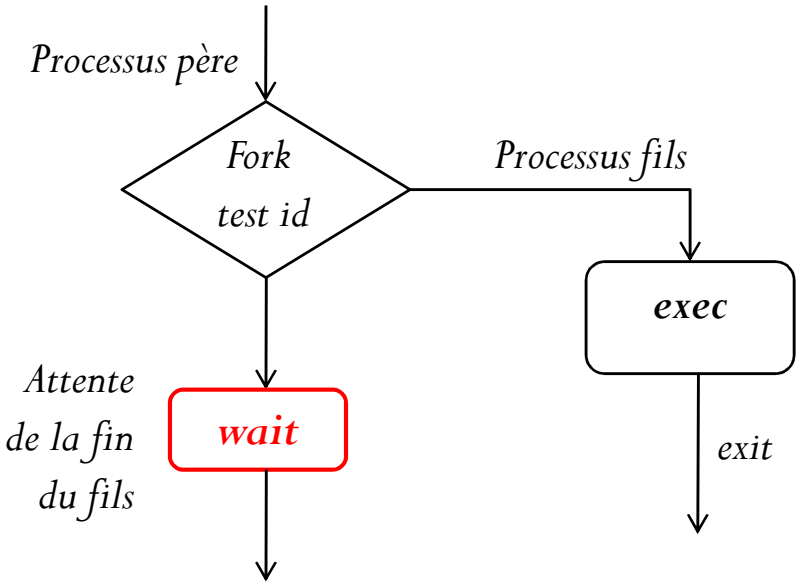
- Un processus se termine lorsque:
  - **exit normal:** dernière instruction (volontaire) ***void exit(int status)***
  - **exit d'erreur** (volontaire)
  - **Erreur fatale/Violation de protection** (involontaire)
  - **Tué** par un autre processus via

int kill(int pid, int status) (involontaire)

- **Note**: certains processus ne se terminent pas avant l'arrêt de la machine
  - nommés ''demons" (**deamon**) ou serveurs
  - réalisent des fonctions système (login user, impression, serveur web, …)

Bonne Terminaison d'un processus Unix Fork/exec/Wait

- *L'élimination d'un processus terminé de la table des processus ne peut se faire que par son père via la fonction* int wait(int *code_sortie)



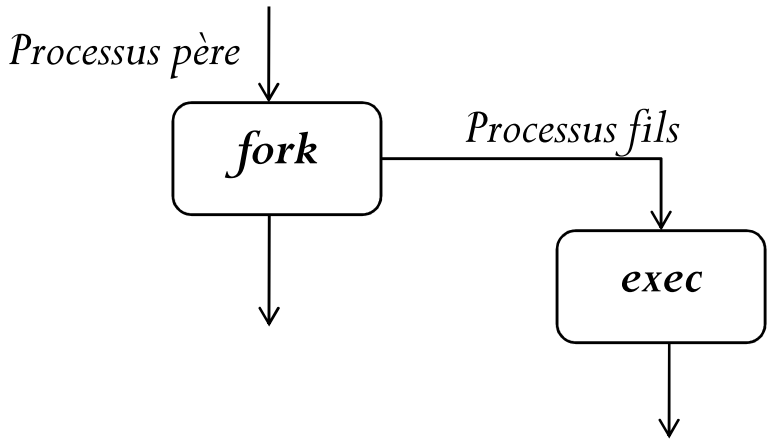
- ***Attendre la fin du fils (pid):*** **int waitpid(int pid, int *code_sortie)**

Action de Wait

- Un processus qui appelle **wait()**, va :
  - **Se bloquer**: si son (ses) processus fils sont en cours d'exécution, ou
  - **Retourner** immédiatement :
    - l'état **terminaison** de son fils correspondant, ou
    - **Erreur**, s'il n'y a pas de processus fils

Mauvaise Terminaison d'un processus Unix Fork/exec/???

- Si le processus père termine son exécution avant son fils, ce dernier devient un processus **orphelin**, qui sera attaché au ***processus initiateur (init)***.
- Si le processus fils meurt avant que son père ne se termine, celui-ci devient un processus **zombie**.



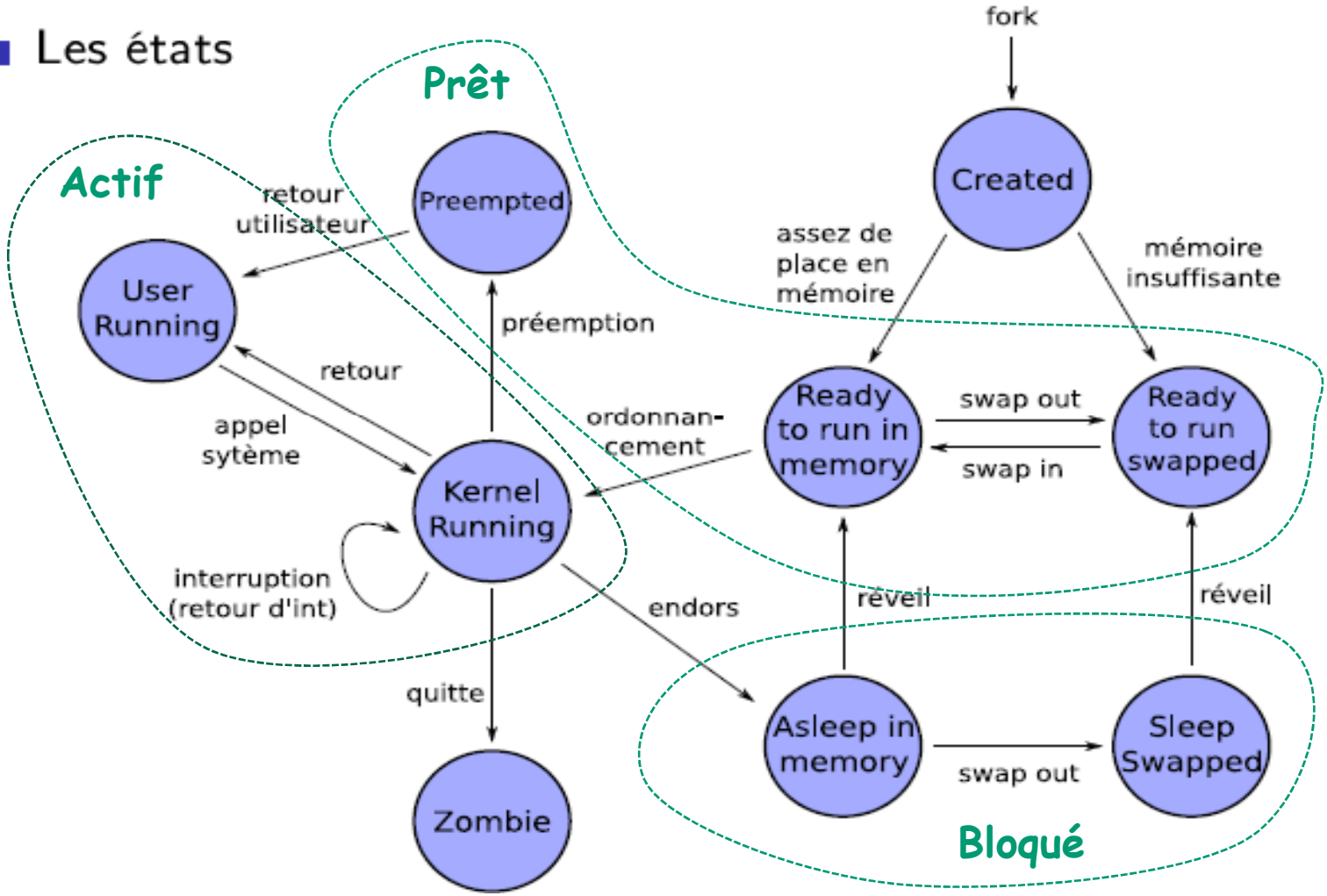
Vie et mort d'un processus Unix

- Grâce aux trois fonctions, *fork()*, *exec()*, et *wait()*, on peut écrire un interprète de commande simplifié:

```
While (1)  /* Boucle infini */
 {   /*** Attendre commande  *****/;
     Lire_cmde(cmde, param);
     if ((pid =fork())!= 0)    /* Processus parent */
           wait();
        else /* Processus fils */
          Exec(cmde, param);
 }
```

Etats d'un processus Unix

- Les états

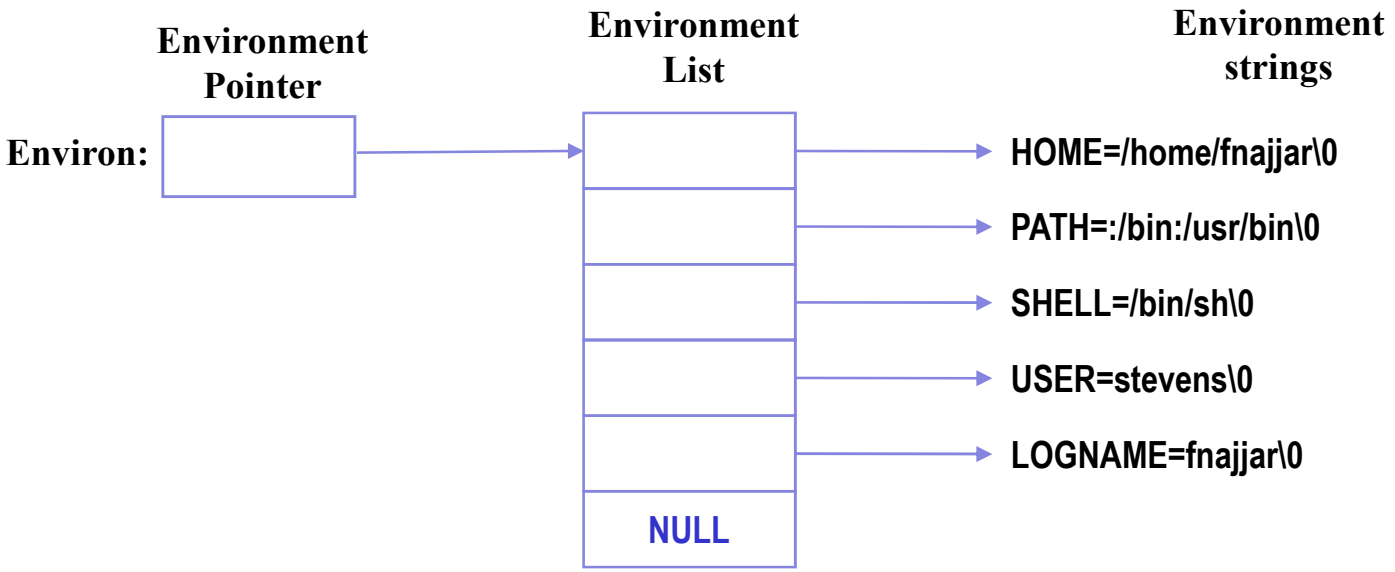


Etats d'un processus Unix (suite)

- ***A chaque état, (prêt / bloqué) il existe plus d'un processus***
  - ⇨ Nécessite d'associer des files d'attentes (FA)
  - ⇨ Unix est un système à mémoire virtuelle:
    - ↳ Une partie (toute) de la mémoire occupée par un processus peut être récupérée au profit d'un autre processus: recopie sur disque (***swap***)
- A sa création, le processus est chargé en mémoire (***FA des prêts***)
- Si le processeur disponible, alors *élire* un processus *éligible*:
  - ⇨ ***Exécution en mode système / user***
- ***Préemption***: interrompu par un processus + prioritaire ou temps épuisé
- Un processus est ***bloqué***: attente d'une ressource non disponible
- ***Terminaison***: un processus est effacé en passant par l'état ***zombie***

Environnement d'un processus Unix –API C

- *extern char **environ;*
  *int main(int argc, char *argv[ ], char *envp[ ])*



Environnement d'un processus Unix -Exemple

```
#include <stdio.h>
void main( int argc, char *argv[], char *envp[] )
  {
        int i;
        extern char **environ;
        printf("from argument envp\n");
        for( i = 0; envp[i]; i++ )
                puts( envp[i] );
        printf("\nFrom global variable environ\n");
        for( i = 0; environ[i]; i++ )
                puts(environ[i]);
  }
```

Environnement d'un processus Unix -Exemple

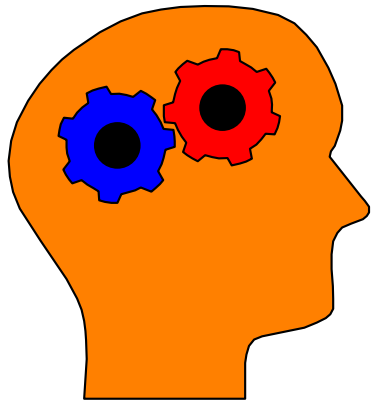
```
#include <stdio.h>
void main( int argc, char *argv[], char *envp[] )
        {
        int i;
        extern char **environ;
        printf("from argument envp\n");
        for( i = 0; envp[i]; i++ )
                puts( envp[i] );
        printf("\nFrom global variable environ\n");
        for( i = 0; environ[i]; i++ )
                puts(environ[i]);
}
```

Partie 2. Programmation Concurrente

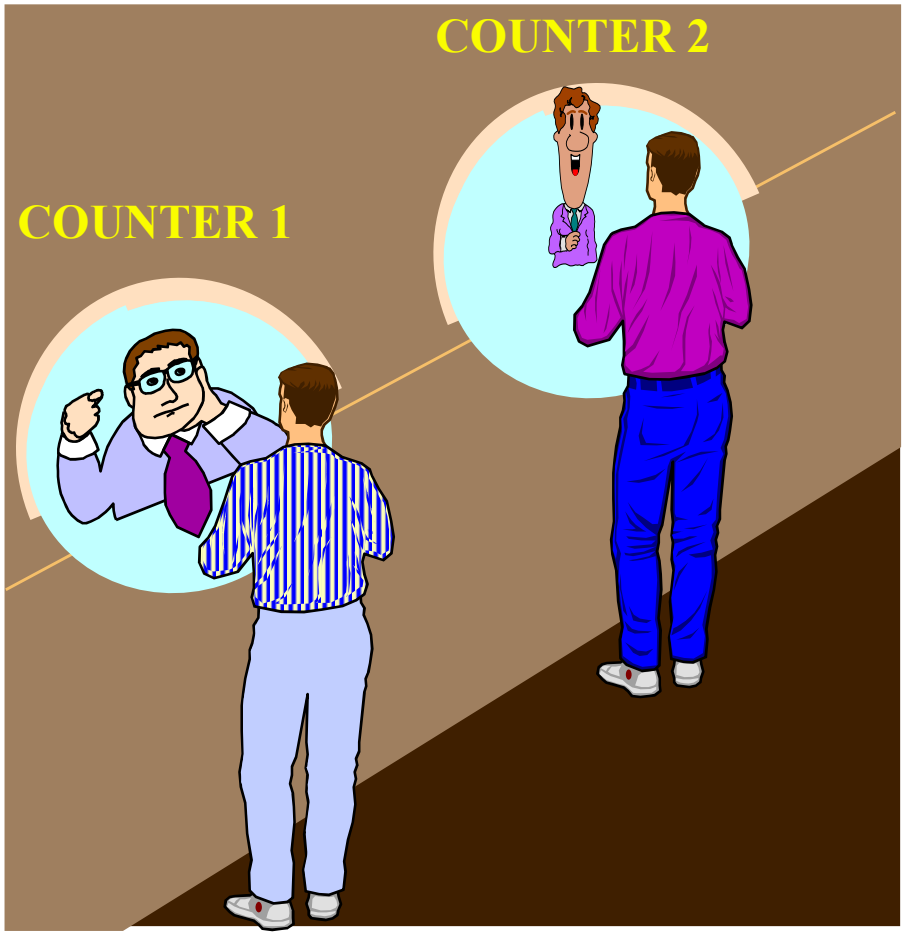
--Les POSIX THREADS

Motivations ?

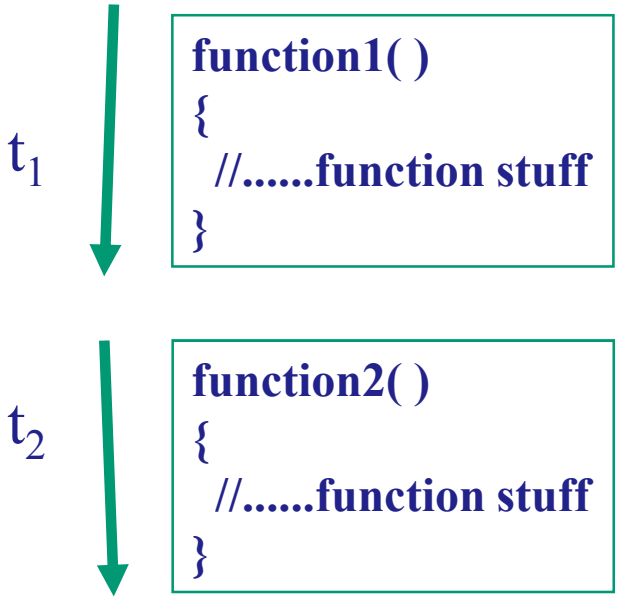
- ✓ Processus/tâche/acteur (selon les systèmes) --**unité de structuration:**
  - ⇨ Abstraction du SE pour l'allocation de la mémoire et du processeur
  - ⇨ Inconvénients des processus classiques: commutation des processus longue (applications multimédia, temps réel, …)
    - ⇩ 90% de ce temps est consacré à la gestion de la mémoire,
  - ⇨ Difficile à gérer dans des environnement distribués ou parallèles
  - ⇨ Pas de partage de mémoire (communication lentes)
  - ⇨ Manque d'outils de synchronisation
  - ⇨ Interface rudimentaire (fork, exec,exit,wait)



Motivations: Séquentiel vs. Parallèle ?



Traitement Séquentiel vs. Parallèle



Traitement séquentiel:

```
function1 ( );
function2 ( );
```

- 1 CPU

Temps: $t_1 + t_2$

Traitement parallèle:

```
function1( ); || function2 ( );
```

- 2 CPUs

Temps : max (t_1, t_2)

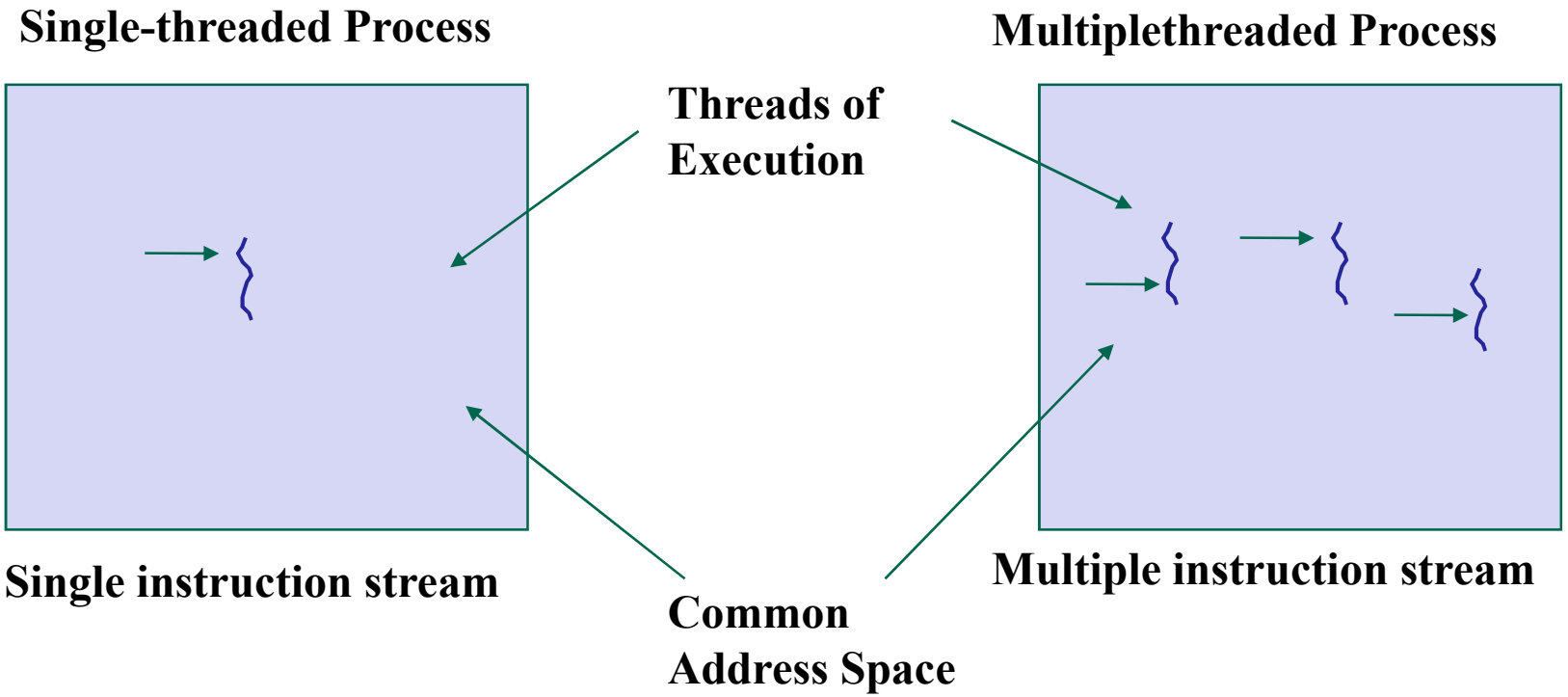
Parallélisme et Concurrence

Parallélisme:	Concurrence:
✓ More than one process is present and executing at a given time. ✓ Usually requires separate hardware, "cores" or CPU's. ✓ Used to scale programs, i.e. reduce execution time by a given factor.	✓ More than one thread is present and active, but not always executed at the same time. ✓ Can be achieved with single core and CPU that "switches". ✓ Increases flexibility and responsiveness.

Les THREADS --Définitions

- **Terminologie:** Thread/processus léger/activité
- **Définitions**
  - ⇨ Abstraction du SE pour l'allocation du processeur –unité d'exécution
  - ⇨ ***Sous-processus (procédure/fonction)*** lié/appartient à un processus (s'exécutant indépendamment du main).
- Processus classique (fork unix) ne comporte qu'un seul thread
  - ⇨ Processus monothreadé (main en C)
- Les threads permettent de dérouler plusieurs suites d'instructions, en parallèle (sur plusieurs CPUs ou cores), l'intérieur du même processus
  - ⇨ Processus multithreadé

Processus monothreadé & Processus multithreadé



Les THREADS --Généralités

- **Asynchronisme:**
  - ⇨ décrit le fait que les threads se produisent de manière indépendante
- **Synchronisation :**
  - ⇨ apparaît quand il existe des dépendances entre les threads.
- **Réentrance:** si plusieurs threads s'exécutent simultanément, une procédure appelée depuis un premier thread peut éventuellement être appelée par un deuxième thread en même temps :
  - ⇨ Une procédure est réentrante si elle admet de telles exécutions sans dommage.

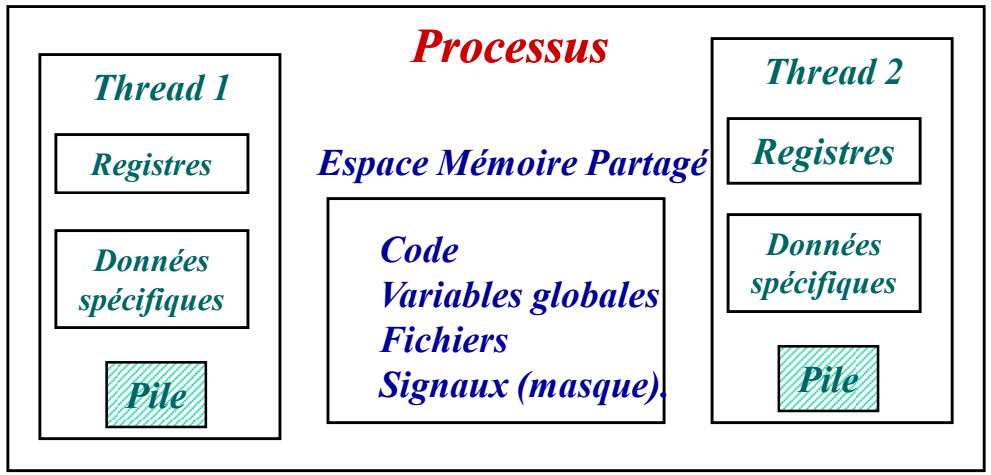
Caractéristiques des Threads

- Processus devient la structure d'ALLOCATION des ressources (fichiers, mémoire) pour les threads :
  - ***Partage de ressources*** (en particulier **l'espace d'adressage) :**
    - le code (chaque thread possède un CO et une pile d'exécution)
    - table des fichiers ouverts (*descripteurs*)
    - table des traitements des signaux
  - ***Taille(processus) >> Taille(ressource d'un thread)***

Thread	*Processus (lourd)*
CO (CP)	*Espace d'adressage*
Pile d'exécution	*Variables globales*
Registres	*Compteurs*
Threads fils	*Processus fils*
Etat	*Descripteurs de fichiers ouverts*

Threads vs. Processus

- Un **thread** exécute une **fonction**. Donc, un thread
  - ⇨ ne "voit" qu'une partie de la région de code du processus qui l'héberge;
  - ⇨ dispose de sa propre pile pour implanter les variables locales;
  - ⇨ partage les données globales avec les autres threads.



- Les ressources propres à la gestion des threads sont dans l'espace utilisateur
- Le noyau garde un contrôle total sur les ressources de la machine
- Chaque thread peut utiliser directement les ressources du processus englobant
  - ⇨ Attention à l'intégrité des données partagées

Efficacité des Threads

- ***Exemple 1 : création de thread***
  - ⇨ allocation descripteur & pile;
  - ⇨ initialisation du contexte d'exécution (affectation de registres);
  - ⇨ insertion dans la file des threads prêts.
- ***Exemple 2: Changement de contexte de thread***
  - ⇨ Repositionnement de PSW
  - ⇨ Temps de commutation court seulement entre threads créés dans le même processus.

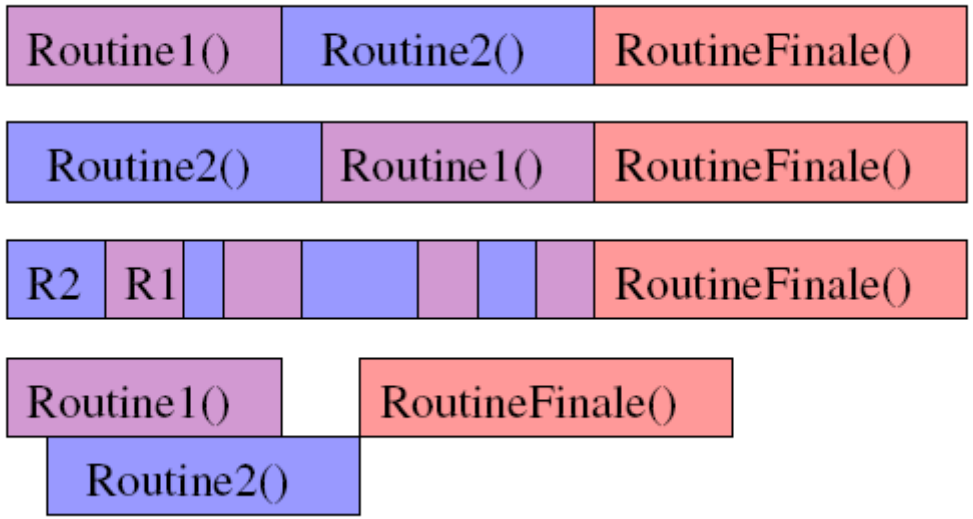
Les threads sont plus rapides à créer

Machine	fork()			threads		
	real	user	sys	real	user	sys
Celeron 2GHz	4.479s	0.364s	3.756s	1.606s	0.380s	0.388s
AMD64 2.5GHz (4CPU)	7.006s	0.936s	6.244s	0.903s	0.300s	0.640s

Source: M. Quinson, introduction aux systèmes, 2010

Conception d'une application multithread

- *Applications candidates?*
  - ⇨ Applications organisées en tâches indépendantes s'exécutant indépendamment
  - ⇨ Toutes routines pouvant s'exécuter en (pseudo-)parallèle sont candidates
- Dans l'exemple, routine1() et routine2() sont candidates



Source: M. Quinson, introduction aux systèmes, 2010

Avantages et Inconvénients des threads

Avantages	Inconvénients
❑ Exploitation of parallelism on multi-CPU hardware ❑ Exploitation of Concurrency on all systems ❑ Modularity and Flexibility	❑ Computing overhead, largely to synchronization ❑ Increased complexity and programming discipline ❑ Libraries may not be thread safe ❑ Harder to debug

Implémentation: les Threads Noyau vs. les Threads users

- ***Un thread peut être implanté au*** niveau du***:***
  - ***NOYAU***, il est alors ordonnancé indépend. du processus dans lequel il a été créé,
  - ***PROCESSUS*** qui l'accueille (espace user), il accède alors au processeur dans les quanta alloués à ce processus
- Dans le premier cas, le thread est l'unité d'ordonnancement.
- La seconde méthode sollicite moins le noyau (pas d'appel à celui-ci pour les changements de contexte), mais :
  - problème de gestion des E/S qui vont bloquer tout le processus (les appels systèmes bloquants doivent être redéfinis)
  - pas de réel contrôle sur l'ordonnancement, en particulier dans le cas des MP

Approches d'Implémentation des threads

3 approches possibles d'implémentation des threads :

1. ***Approche M-1 (Many-to-One) :*** *Systèmes sans multithreading au niveau noyau*

   ⇨ 1 thread noyau est associé à plusieurs threads utilisateur

   ☺ Gestion des threads est réalisée en mode utilisateur

   ☹ Le blocage d'un thread bloque tout le processus → parallélisme non possible entre les threads utilisateur correspondant au même thread noyau

2. ***Approche 1-1 (One-to-One) :***

   ⇨ A chaque thread utilisateur correspond un thread noyau

   ☺ Le blocage d'un thread n'entraîne pas le blocage du processus créateur

   ☹ La création d'un thread suppose la création d'un thread noyau correspondant (et ses ressources correspondantes)

   ⇨ Approche adoptée par *Windows NT, OS2 et Chorus*.

Approches d'Implémentation des threads (suite)

3. ***Approche M-N (Many-to-Many) :***

- ⇨ M ≥ N≥ 1
- ⇨ Différents threads user sont multiplexés sur un nombre (≤) de threads noyau
- ⇨ Approche adoptée par *Solaris*
  - ⇩ Un thread utilisateur peut être associé ou non à un thread noyau

Principaux Apports des threads

- ***Aspects système :***
  - la mémoire est partagée donc le changement de contexte est simple (pour les threads d'un même processus): il suffit de commuter quelques registres (d'où le synonyme processus léger -- *lightweight process*);
  - fonctionnement du noyau en parallèle sur plusieurs processeurs;
  - gain de performances et une meilleure utilisation des ressources sur des machines monoprocesseur.
- ***Aspects utilisateur :***
  - *Modularité :* découpage facile de l'application en activités parallèles, donc utilisation simple des différentes unités d'un multiprocesseur,
  - Les threads permettent de passer d'un modèle de programmation asynchrone à un modèle synchrone (les E/S bloquantes sont gérées par des threads).
  - *API puissante:* outils de synchronisation variés,…
- ✓ ***Aspects langage:***
  - Les threads permettent d'implanter le threading C/java, le tasking Ada

Support de programmation des threads

- Librairie **POSIX thread (pthread),** ***ISO/IEEE standard***
- Mach C threads, *CMU*
- Sun OS LWP threads, *Sun Microsystems*
- PARAS CORE threads, *C-DAC*
- **Java-Threads,** ***Sun Microsystems***
- Chorus threads, *Paris*
- OS/2 threads, *IBM*
- Windows NT/95 threads, *Microsoft*
- *Ada95, Modula-3*

Programmer avec les Threads --Notions de Base

- *Les ressources propres à un thread sont :*
  - ⇨ tid (*t*hread *i*dentifier équivalent du pid) : un identificateur de thread
  - ⇨ une priorité,
  - ⇨ une configuration de registres, une pile
  - ⇨ un masque de signaux,
  - ⇨ d'éventuelles données privées (état, …),
- Le nombre et l'identité des threads d'un processus sont *invisibles* depuis un autre processus,
  - ⇨ Spécifier le degré de concurrence souhaité dans le processus (thread principal du processus) au moyen de la fonction *pthread_setconcurrency*.
- *Attention* à l'utilisation par les threads des appels concernant tout le processus comme exit, …
- Une *bibliothèque* (ensemble de procédures et données) est *réentrante* si toutes ses procédures sont réentrantes les unes vis-à-des autres (et d'elles mêmes).

POSIX THREADS -- Bibliothèque des threads

- ✓ POSIX Threads: API standard
- ✓ POSIX : Portable Operating System Interface for UniX
- ✓ Approuvé par IEEE in 1995
- ✓ Nom officiel: POSIX 1003.1c-1995
- ✓ Nouvelle version ISO/IEC 9945-1:1996
- ✓ Pthreads:

POSIX THREADS -- Bibliothèque des threads (suite)

- ✓ ***Que contient la librairie ?***
  - ⇨ Manipulation des threads (création, terminaison, …);
  - ⇨ synchronisation : mutex, variables condition.
  - ⇨ Primitives annexes : données spécifiques à chaque thread, politique d'ordonnancement, …
  - ⇨ Ajustement des primitives standard : processus lourd, E/S, signaux, routines réentrantes.
- ✓ ***Vocabulaire Posix :***
  - ⇨ MT-safe : réentrance vis-à-vis du parallélisme
  - ⇨ asyn-safe : réentrance vis-à-vis des signaux
- ✓ ***Compilation :***
  - ⇨ *inclure <pthread.h>*
  - ⇨ *compiler avec -D_REENTRANT*
  - ⇨ *faire l'édition de liens avec -lpthread*

Are Libraries Safe?

✓ MT-Safe	This function is safe
✓ MT-Hot	This function is safe and fast
✓ MT-Unsafe	This function is not MT-safe, but was compiled with _REENTRANT
✓ Alternative Call.	This function is not safe, but there is a similar function (e.g. getctime_r())
✓ MT-Illegal	This function wasn't even compiled with _REENTRANT and therefore can only be called from the main thread.

Principales Fonctions de Manipulation

Nom de la fonction	Rôle de la fonction
Pthread_create(pthread_t tid, ..., void* ((*fonction), void* arg))	**Création** d'un thread Par défaut, tous les threads ont la même priorité. Son démarrage dépend de sa priorité. On peut changer sa priorité.
Pthread_exit(void* etat)	**Termine** le thread en fournissant un code de retour, à la différence de exit qui termine le processus et tous ses threads.
Pthread_self(void)	**Identification** du numéro (tid) du thread courant, équivalent de getpid().
pthread_join (pthread_t tid, void ** etat)	Pour **attendre** la **fin** d'un thread dont on donne le tid.

✓ *Une documentation plus complète de la librairie pthread est disponible dans le man :*

⇨ *man libpthread / man pthread_xxx*

PTHREADS -Identification des threads

- ***Chaque thread a un ID unique de type pthread_t*** (équivalent du ***pid_t*** , c'est une structure opaque)
  - ***Pthread_t pthread_self ()*** : retourne l'ID du thread qui appelle cette fonction identité du thread courant

PTHREADS -- Création de threads

pthread_create()

```
int
pthread_create(
   pthread_t *tid,                          // thread ID
   const pthread_attr_t *attr;              // thread attributes
   void *(*start_routine)(void *);          // pointer to function
   void *arg);                              // argument to function
```

Arguments

- The ID of the successfully create thread is returned in `*tid`
- `attr` specifies thread attributes (NULL for default values)
- The new thread begins by executing `start_routine`
- `arg` is passed to `start_routine()`

Return value

- 0 if successful. Error code otherwise (see `<errno.h>`)

Notes

- Use a structure to pass multiple arguments to the start routine

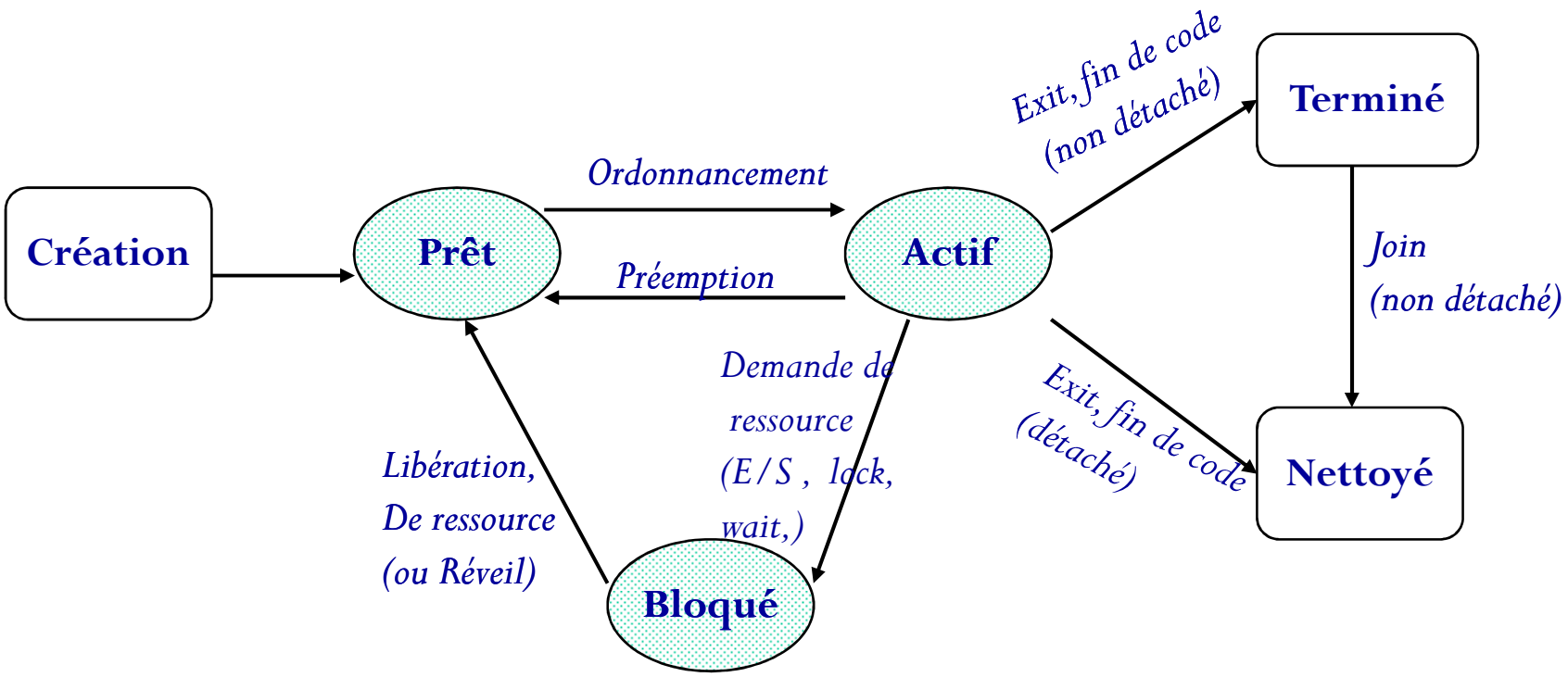
Programmer avec les Threads --Exemple

thread-ex1.c

```
#include <pthread.h>
void *hello( void *arg ) {
  int *id = (int*)arg;
  printf("%d : hello world \n", *id);
  pthread_exit(NULL);
}
int main (int argc, char *argv[ ]) {
  pthread_t thread[3];
  int id[3]={1,2,3};
  int i;

  for (i=0;i<3;i++) {
    printf("Crée thread %d\n",i);
    pthread_create(&thread[i], NULL,
                   hello, (void *)&id[i]);
  }
  pthread_exit(NULL);
}
```

```
$ gcc -pthread -o thread-ex1 thread-ex1.c
$ ./thread-exemple1
Crée thread 1
Crée thread 2
Crée thread 3
1 : hello world
2 : hello world
3 : hello world
$
```

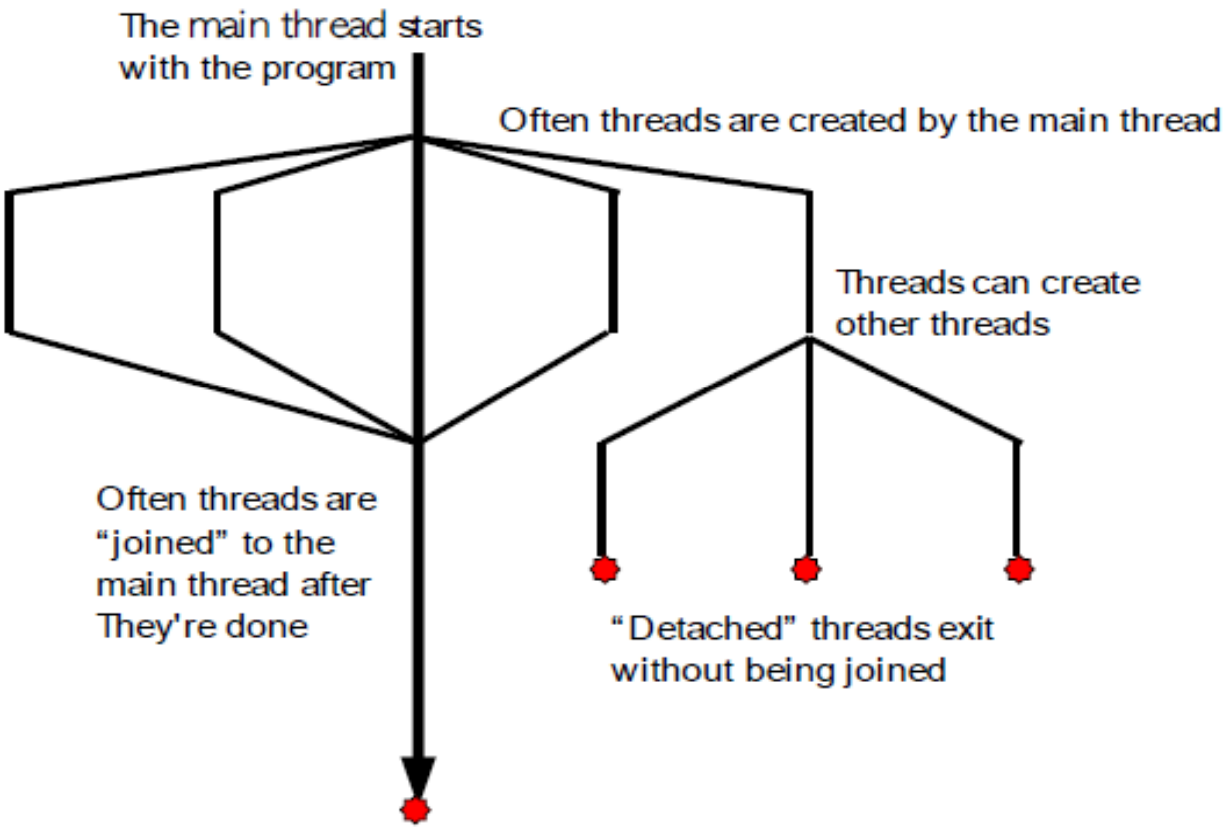
Le Cycle de Vie d'un thread



Thread states

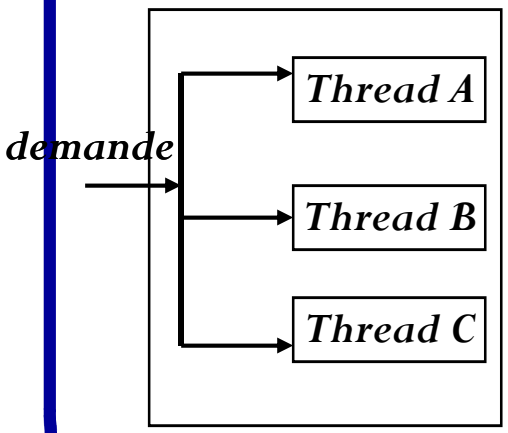
- State shared by all threads in process/addr space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)
- State "private" to each thread
  - Kept in **TCB --Thread Control Block**
  - CPU registers (including, program counter)
  - Execution stack
    - Parameters, Temporary variables
    - return PCs are kept while called procedures are executing

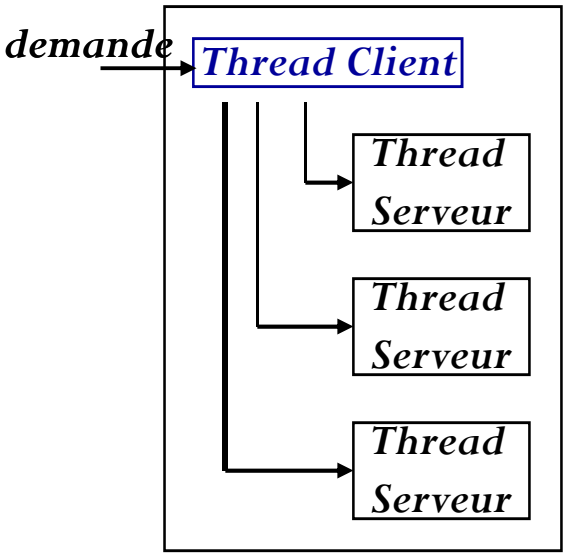
Les Threads sont des Processus Dynamiques

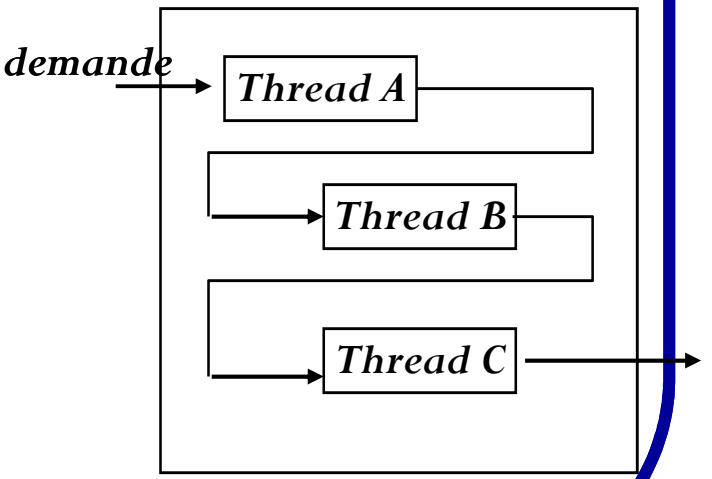


Ordonnancement - Paradigmes de Structuration

- ✓ 3 paradigmes de structuration de threads issus d'un même processus
  - ⇨ ***Paradigme ouvriers :*** ensemble (généralement fixe) de threads réalisant le même travail (le même code), en extrayant des travaux parmi un ensemble de soumissions
  - ⇨ ***Paradigme Client/Serveur :*** décomposition du travail
  - ⇨ ***Paradigme Producteurs/Consommateurs*** --tubes et filtres: chaîne de threads







Récapitulation

- *Threads provide a more natural programming paradigm*
- *Improve efficiency on uniprocessor systems*
- *Allows to take full advantage of multiprocessor Hardware*
- *Improve Throughput: simple to implement asynchronous I/O*
- *Leverage special features of the OS*
- *Many applications are already multithreaded*
- *MT is not a silver bullet for all programming problems.*
- *There is already standard for multithreading--POSIX*
- *Multithreading support already available in the form of language syntax--Java*
- *Threads allows to model the real world object (ex: in Java)*

Lectures Complémentaires

1. *Claude Evéquoz, Introduction à la programmation concurrente. Polycopié de cours, HES, Suisse Occidentale 2009.*

Questions ...???