

SYSTÈME D'EXPLOITATION I

SIF-1015

Contenu du cours 2

- Processus
 - Concepts de base
 - Notions de processus et threads
 - État et attributs des processus
 - Appels système de base
 - Création des processus et des threads
 - Gestion des attributs d'un processus
 - Concepts avancés
 - ordonnancement des processus
 - modification de l'ordonnancement

Contenu du cours 2

- Lectures:
 - Chapitres 3, 4, 5 (OSC)
 - Chapitres 10, 11 (Matthew)
 - Chapitres 2, 9, 11, 12 (Blaess)
 - Chapitre 4 (Card)
 - Chapitres 3, 10, 11 (Bovet)
 - Chapitres 3, 4 (Mitchell)
 - Site: <http://csapp.cs.cmu.edu/>
 - Section étudiants
- Exemples:
 - site ftp UQTR, répertoire Exemples
 - DMILINUX: répertoire **Exemples_Meunier**

Contenu du cours 2

- Lectures (Processus):
 - Explication théorique des processus avec schémas et exemples commande (ANGLAIS)
<http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=83>
 - Explication FORK, EXEC, PROCESS avec des exemples de code (ANGLAIS)
<http://www.yolinux.com/TUTORIALS/ForkExecProcesses.html>
 - Tutorial sur les process avec des schémas (Code de programmation) (ANGLAIS)
http://www.ftt.co.uk/tutorials/Sysprog_tutorial1.html
 - Tutorial Programmation Multi-Processus (Code de programmation) (ANGLAIS)
<http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html>

Contenu du cours 2

- Lectures (Threads):
 - Tutorial Programmation Thread POSIX (Contient une liste des fonctions pour les THREADS)
<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>
 - Introduction à la programmation des THREADS (ANGLAIS)
http://www.mit.edu/people/proven/IAP_2000/index.html
 - Tutorial complet sur les PTHREAD (Avec question) (ANGLAIS)
<http://www.llnl.gov/computing/tutorials/pthreads/>
 - Tutorial THREAD POSIX (ANGLAIS)
<http://math.arizona.edu/~swig/documentation/pthreads/>
 - Norme des THREAD Portable par le GNU (Théorie des commandes de programmation) (ANGLAIS)
<http://www.gnu.org/software/pth/pth-manual.html>
 - Tutorial Introduction à la programmation MULTI-THREAD (Code de programmation) (ANGLAIS)
<http://vergil.chemistry.gatech.edu/resources/programming/threads.html>

Contenu du cours 2

- Lectures (Threads):

- Programmation multi-thread (Code de programmation) (ANGLAIS)

<http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html>

- Programmation multi-thread (ANGLAIS)

<http://www.awprofessional.com/articles/article.asp?p=679545&seqNum=1>

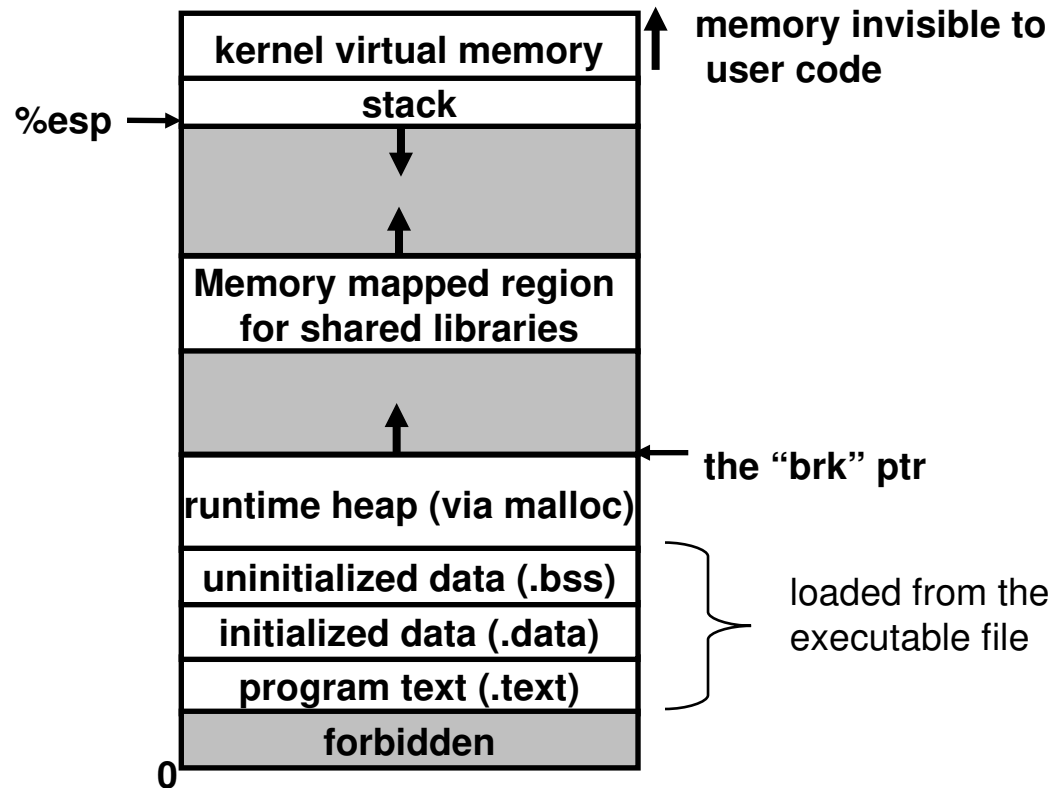
Processus (Concepts de base)

- **Un processus** est une instance d'un programme en cours d'exécution, est aussi appelé tâche (task).
- L'exécution progresse de façon séquentielle, une instruction au plus est exécutée à la fois par un processus
- Un processus ne se limite pas au programme qu'il exécute et ses données en mémoire. Il est aussi caractérisé par son activité courante (compteur ordinal, registres du processeur). Il comprend aussi une pile (données temporaires) et un segment de données (variables globales) et un ensemble de descripteurs de fichiers ouverts.

Processus (Concepts de base)

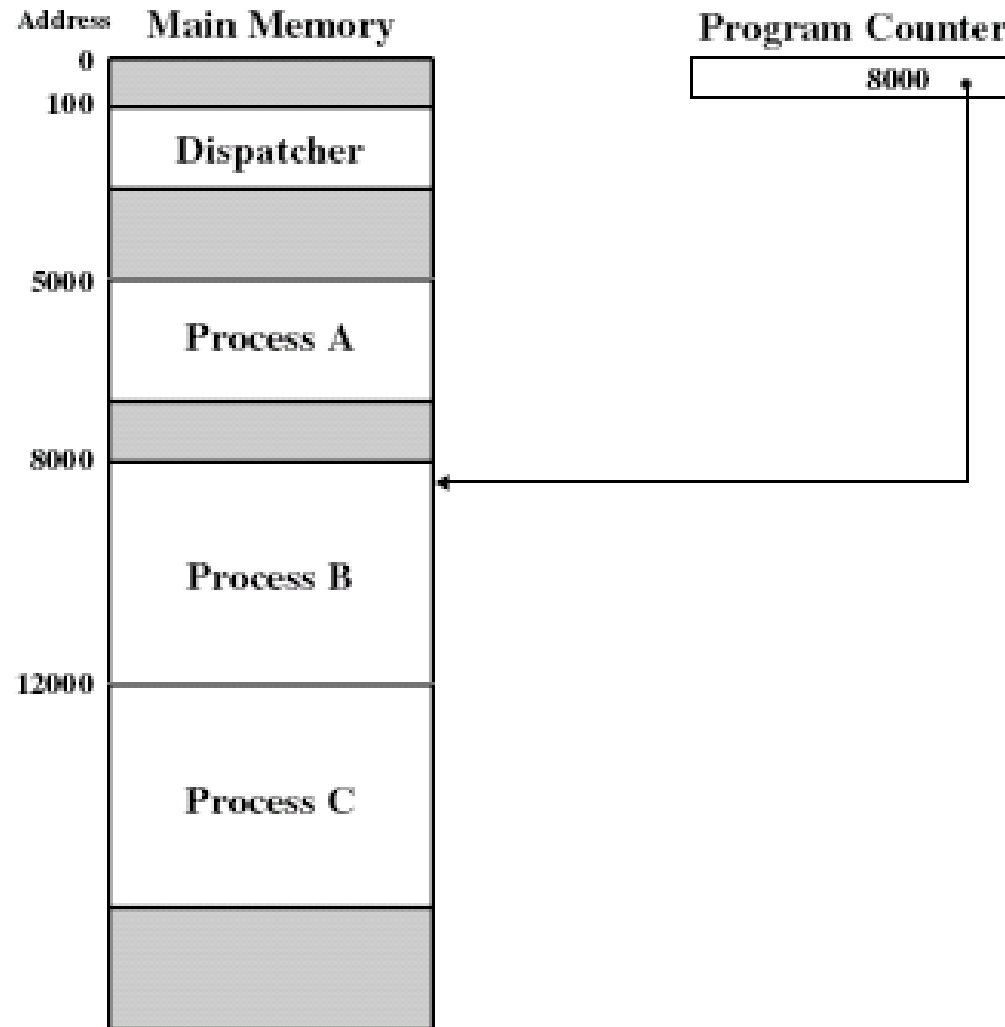
- Contexte d'exécution d'un processus

Linux/x86
Image d'un
processus
en mémoire



Processus (Concepts de base)

- **Contexte d'exécution d'un processus**



Processus (Concepts de base)

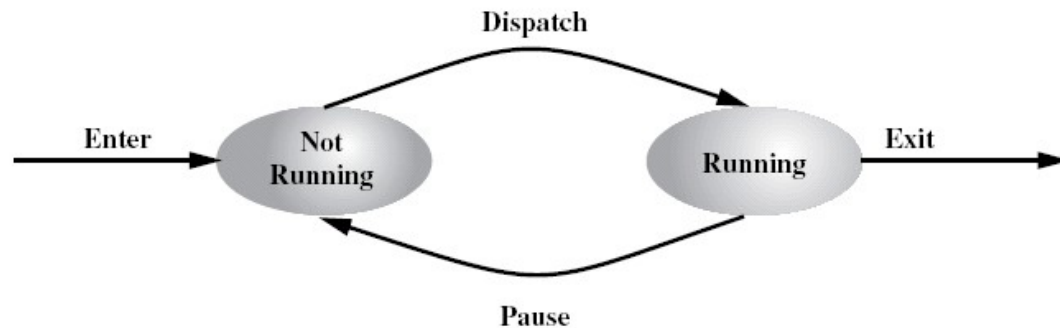
- Un programme est une entité passive (séquence d'instructions contenues dans un fichier exécutable sur disque). Un processus est une entité active avec un compteur ordinal et des ressources associées.
- Chaque fois qu'un usager lance l'exécution d'un programme en tapant le nom d'un fichier exécutable dans un SHELL (interpréteur de commandes). Le SHELL crée un nouveau processus et lance l'exécution de ce programme dans le contexte de ce nouveau processus

Processus (Concepts de base)

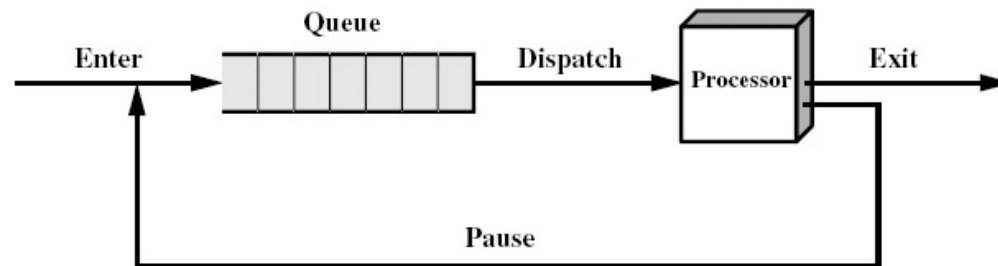
- Les processus sous LINUX
 - LINUX est multi-programmation : il peut gérer plusieurs processus à la fois. Si un des processus plante, ça n'aura pas d'impact sur les autres processus en cours d'exécution.
 - Les processus s'exécutent dans leur plage mémoire respective et ne peuvent interagir entre eux autrement que par les mécanismes sécuritaires du noyau (ex: IPC).
 - LINUX partage les ressources équitablement entre les processus.
 - LINUX peut supporter plusieurs processeurs. Son objectif est de faire exécuter un processus sur chaque processeur en tout temps. Sinon de partager le processeur équitablement entre les différents processus.

Processus (Concepts de base)

- États d'un processus
 - Modèle de processus à deux états (modèle simplifié)
 - En exécution: le processus est exécuté par le processeur
 - Pas en exécution: le processus est prêt pour être exécuté mais un autre processus est en cours d'exécution



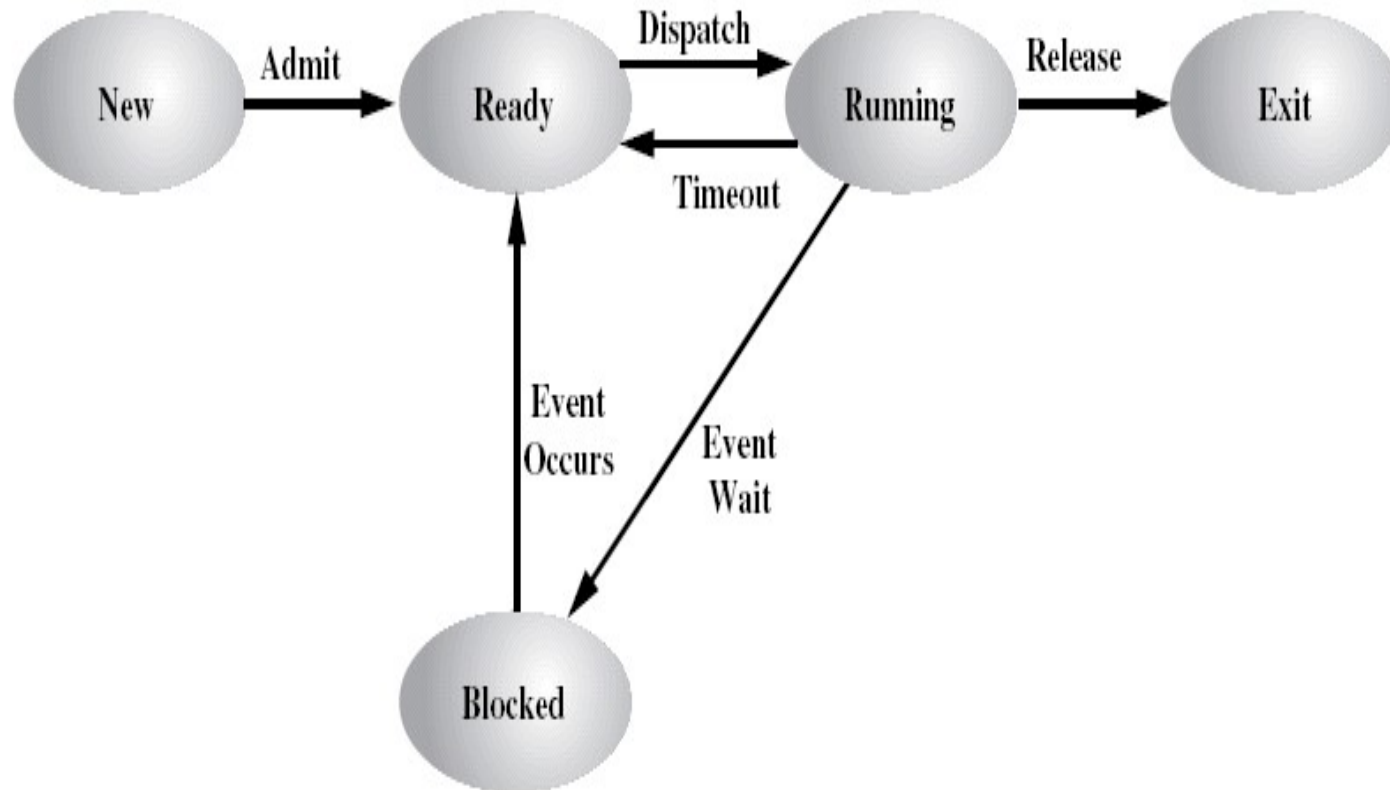
(a) State transition diagram



(b) Queuing diagram

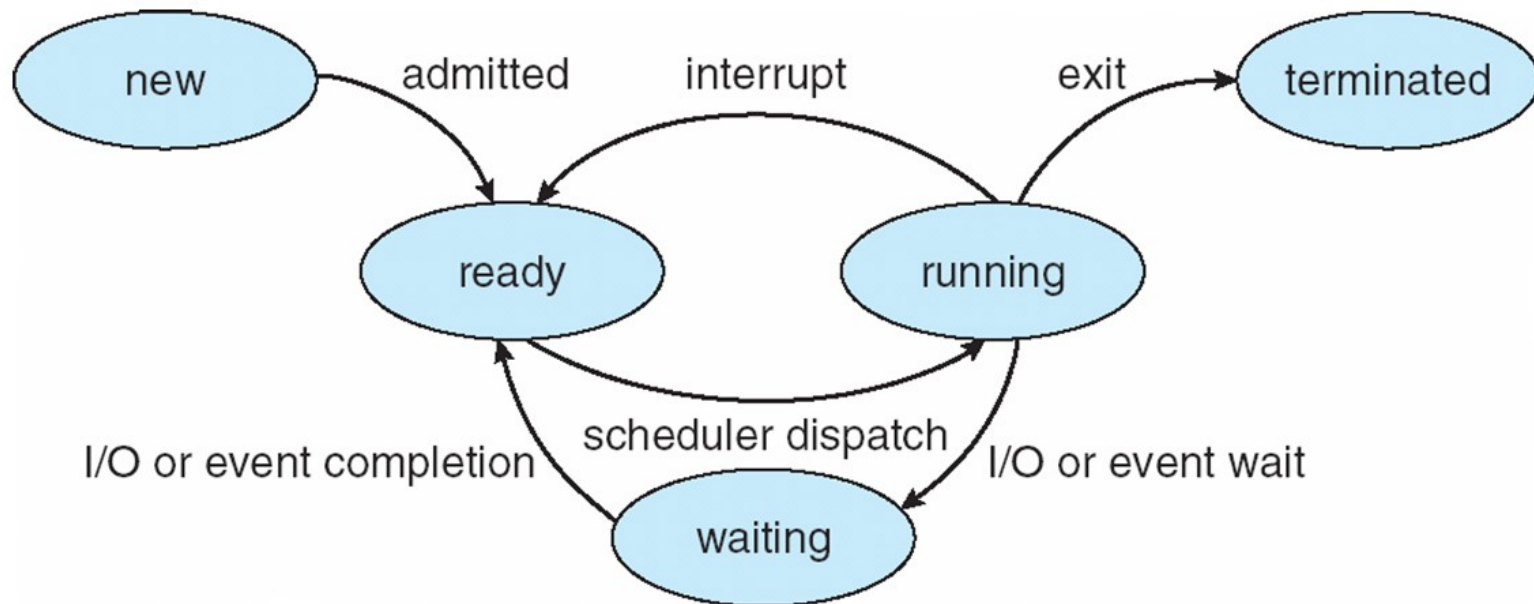
Processus (Concepts de base)

- États d'un processus
 - Modèle de processus à cinq états
 - Ajout d'un état bloqué pour distinguer les processus en attente de service de ceux prêt à être exécutés



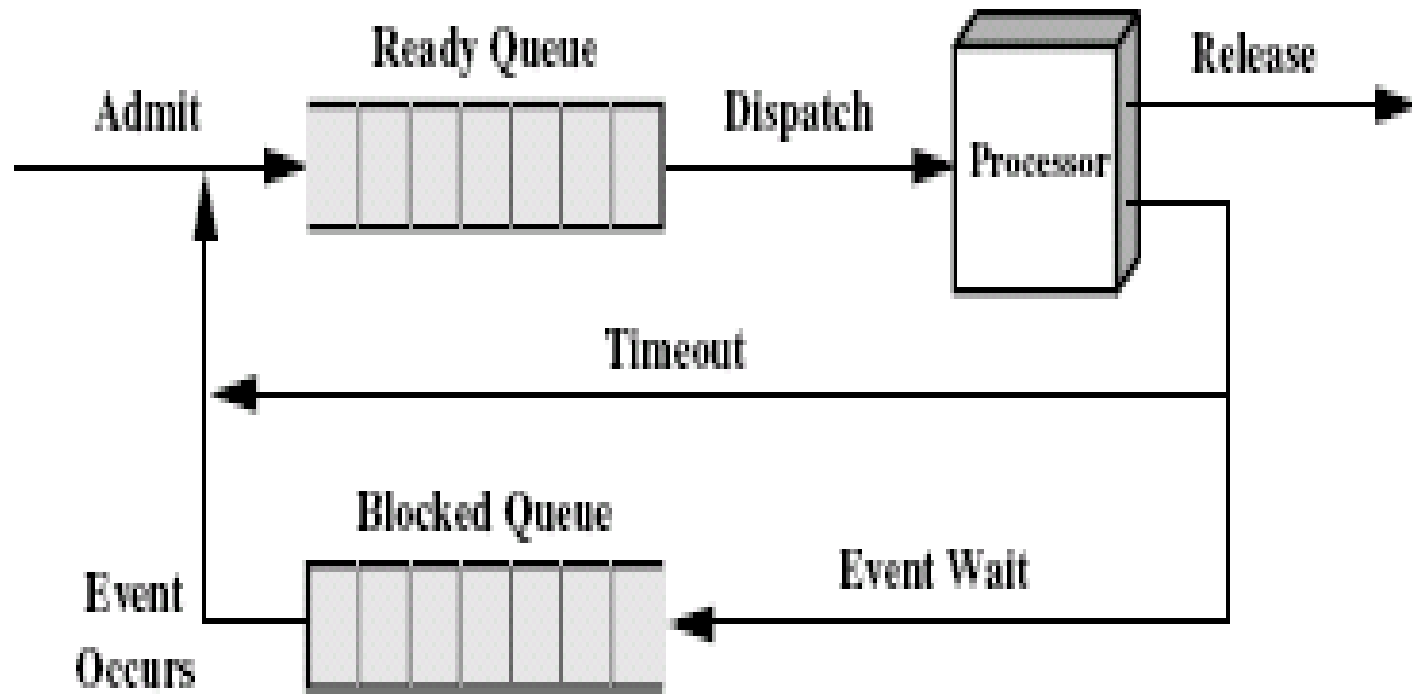
Processus (Concepts de base)

- États d'un processus
 - Modèle de processus à cinq états



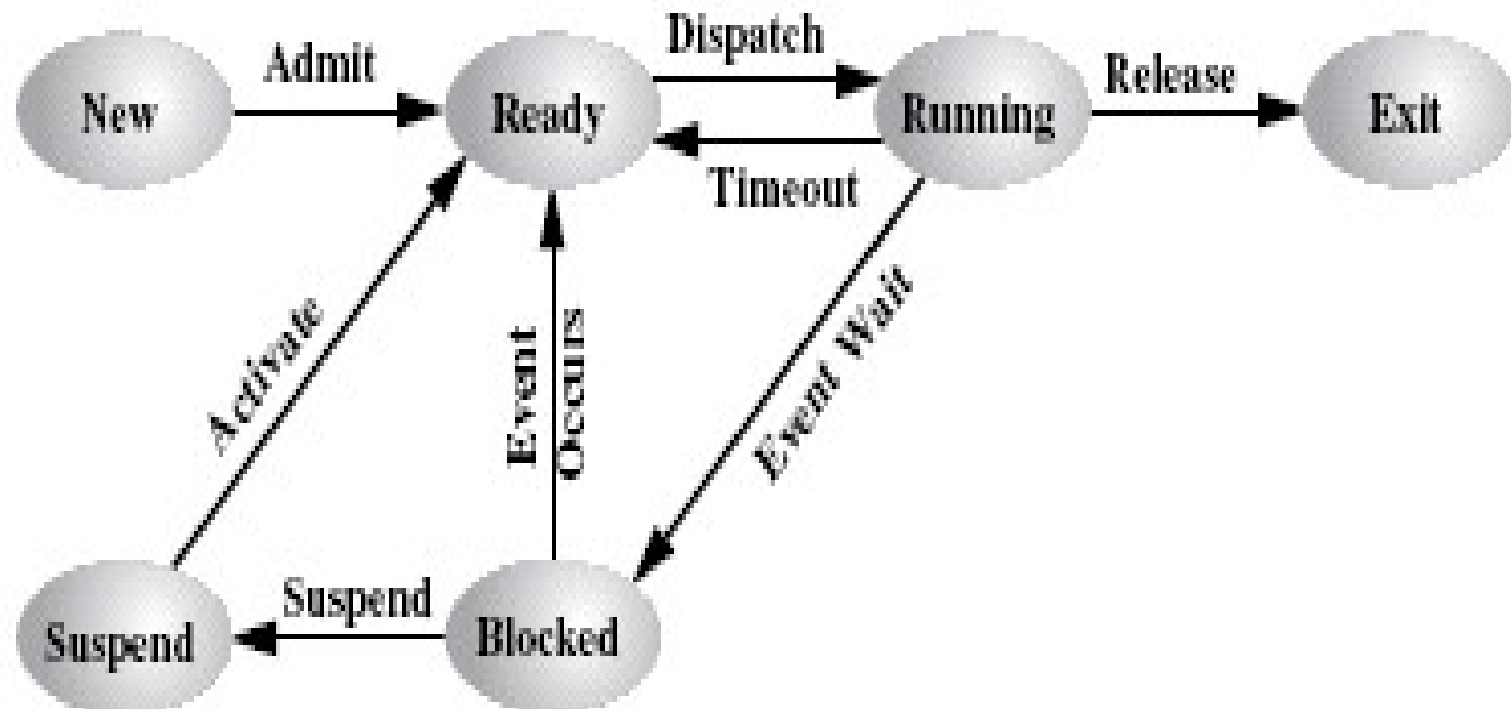
Processus (Concepts de base)

- États d'un processus
 - Modèle de processus à cinq états (Tampon de processus bloqués)



Processus (Concepts de base)

- États d'un processus
 - Modèle de processus à six états
 - État suspendu ajouté pour permettre de libérer l'espace mémoire utilisé par un processus longuement bloqué

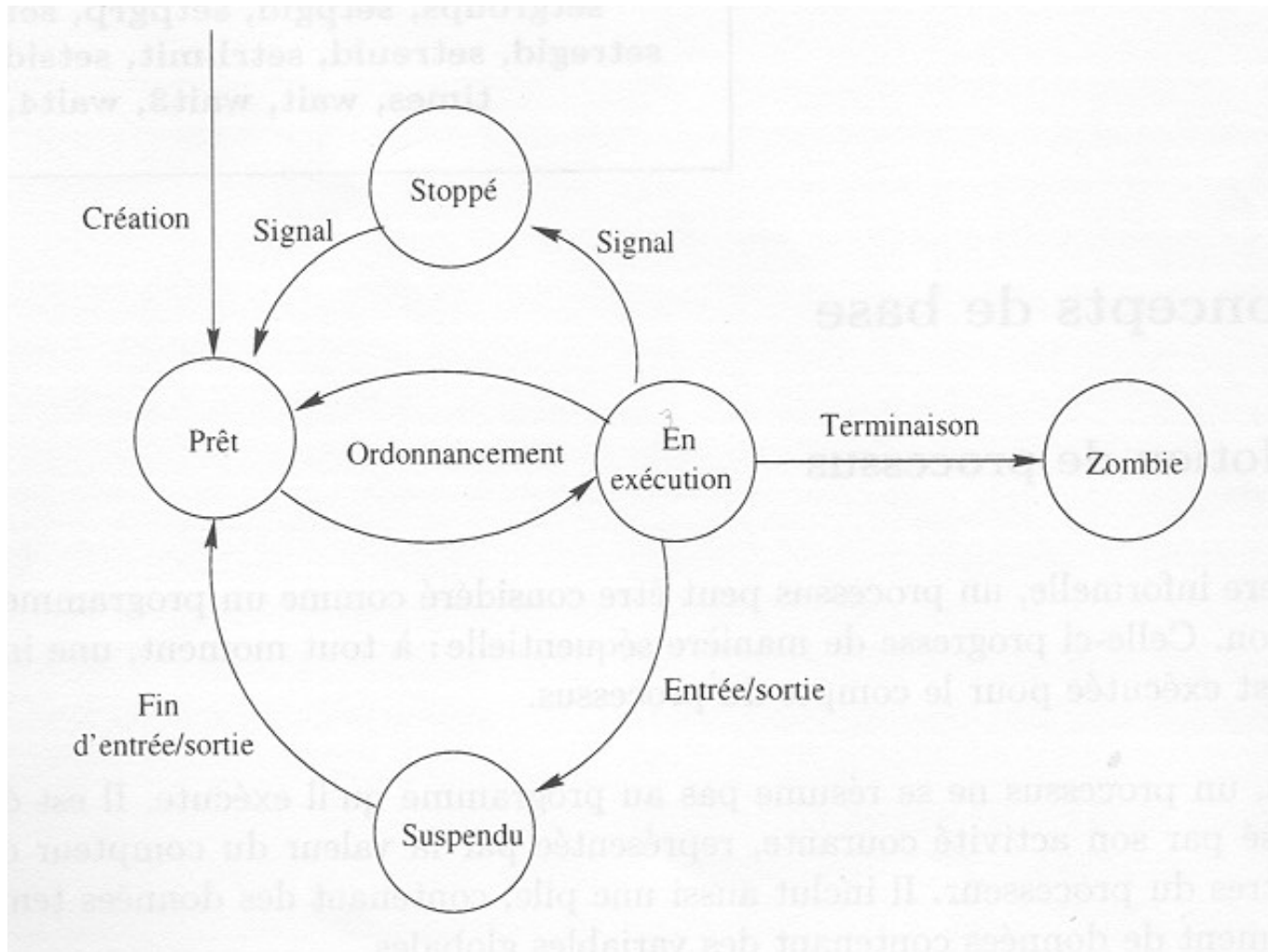


Processus (Concepts de base)

- États d'un processus (LINUX)
 - Durant son exécution, un processus change d'état. L'état du processus est fonction de son activité courante. Les états possibles sont:
 - en exécution: le processus est exécuté par le processeur
 - Prêt: le processus est prêt pour être exécuté mais un autre processus est en cours d'exécution
 - suspendu: le processus est en attente d'une ressource (ex: fin d'une I/O)
 - stoppé: le processus a été suspendu par une intervention extérieure
 - zombie: le processus a terminé son exécution mais il existe encore dans le système

Processus (Concepts de base)

- États d'un processus (Linux)



Processus (Concepts de base)

- Raisons qui occasionnent la création d'un processus

New batch job	The operating system is provided with a batch job control stream, usually on tape or disk. When the operating system is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The operating system can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Processus (Concepts de base)

- Raisons qui occasionnent la terminaison d'un processus

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource or a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.

Processus (Concepts de base)

- Raisons qui occasionnent la terminaison d'un processus

I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Processus (Concepts de base)

- Raisons qui occasionnent la suspension d'un processus

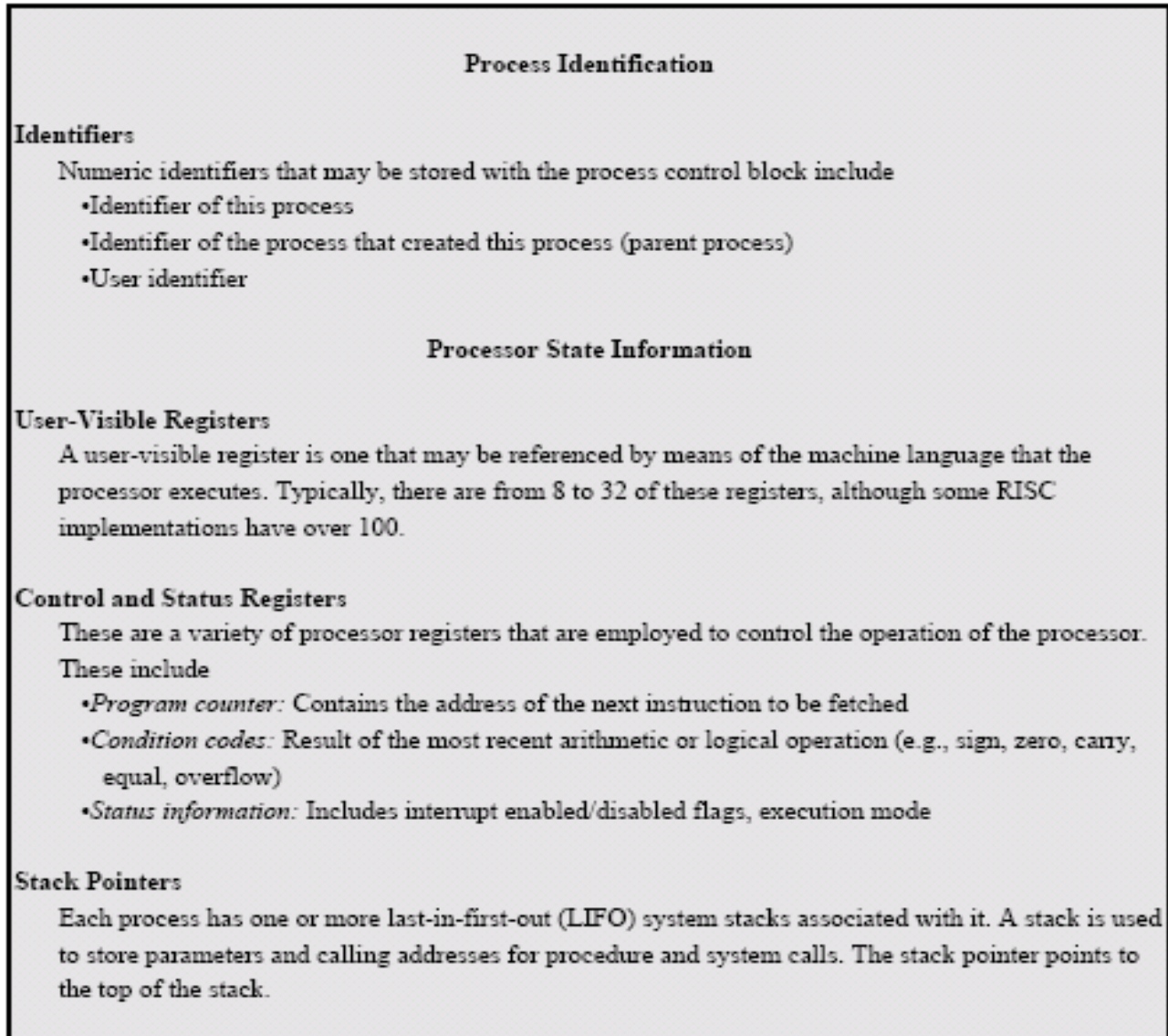
Swapping	The operating system needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The operating system may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendents.

Processus (Concepts de base)

- Attributs d'un processus
 - Durant son exécution, un processus est caractérisé par plusieurs attributs maintenus par le système:
 - son état
 - son identificateur (numéro unique)
 - les valeurs des registres et compteur ordinal
 - le nom de l'utilisateur du processus
 - la priorité du processus (utile pour l'ordonnancement du processus)
 - les informations sur l'espace d'adressage du processus (segments de code, de données, de pile)
 - les informations sur les I/O effectuées par le processus (descripteurs de fichiers ouverts, répertoire courant, etc.)
 - les informations de comptabilité sur les ressources utilisées par le processus (user time, real time, etc.)

Processus (Concepts de base)

- Attributs d'un processus (3 catégories)



Processus (Concepts de base)

Process Control Information

Scheduling and State Information

This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- *Process state*: defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- *Priority*: One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)
- *Scheduling-related information*: This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- *Event*: Identity of event the process is awaiting before it can be resumed.

Data Structuring

A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication

Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

Process Privileges

Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

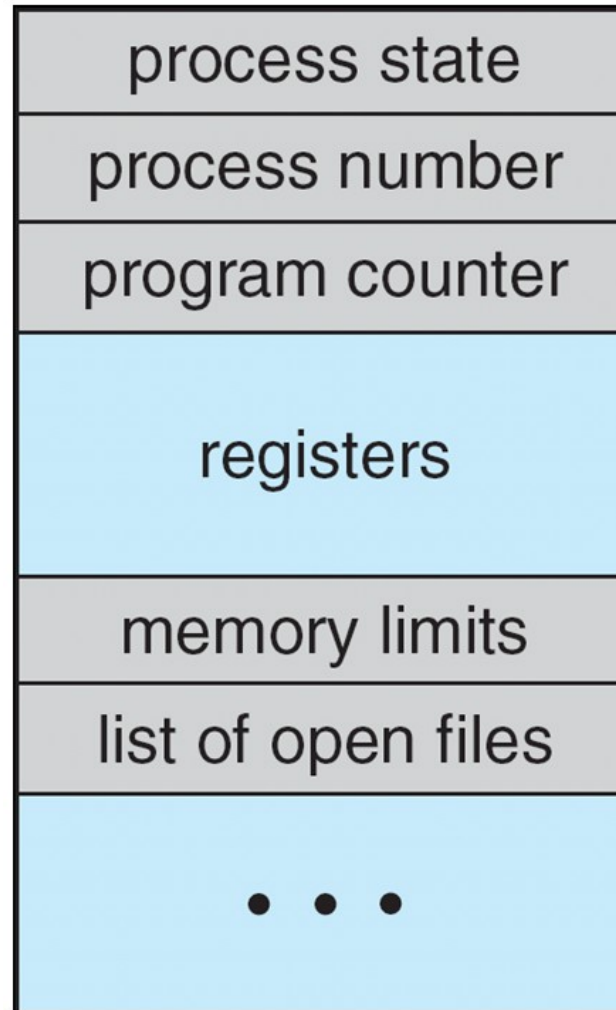
Memory Management

This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization

Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

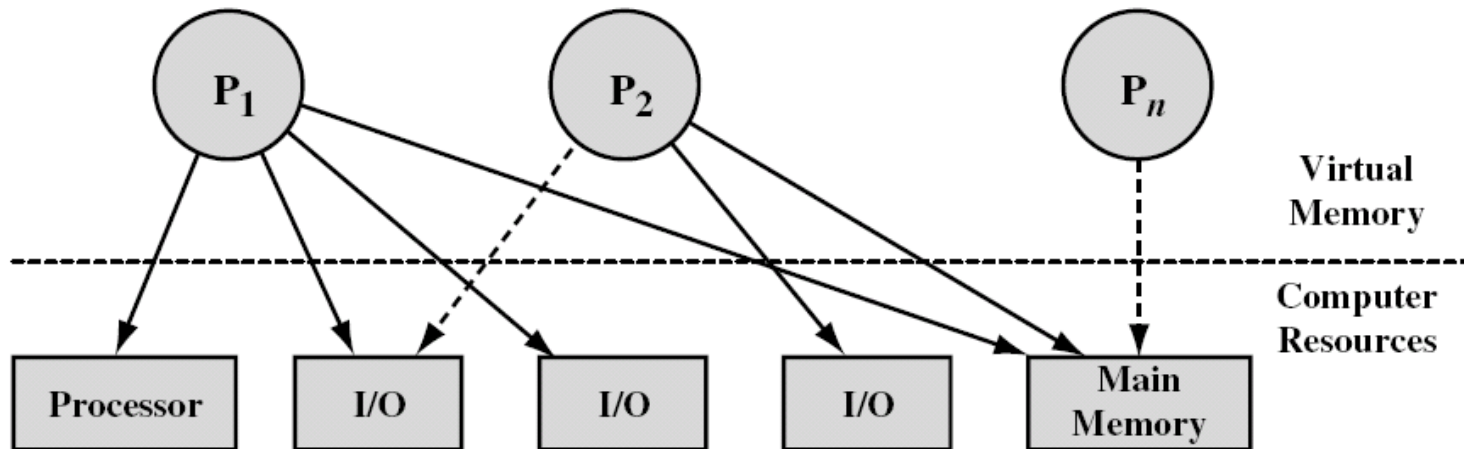
Processus (Concepts de base)



Process control block (PCB)

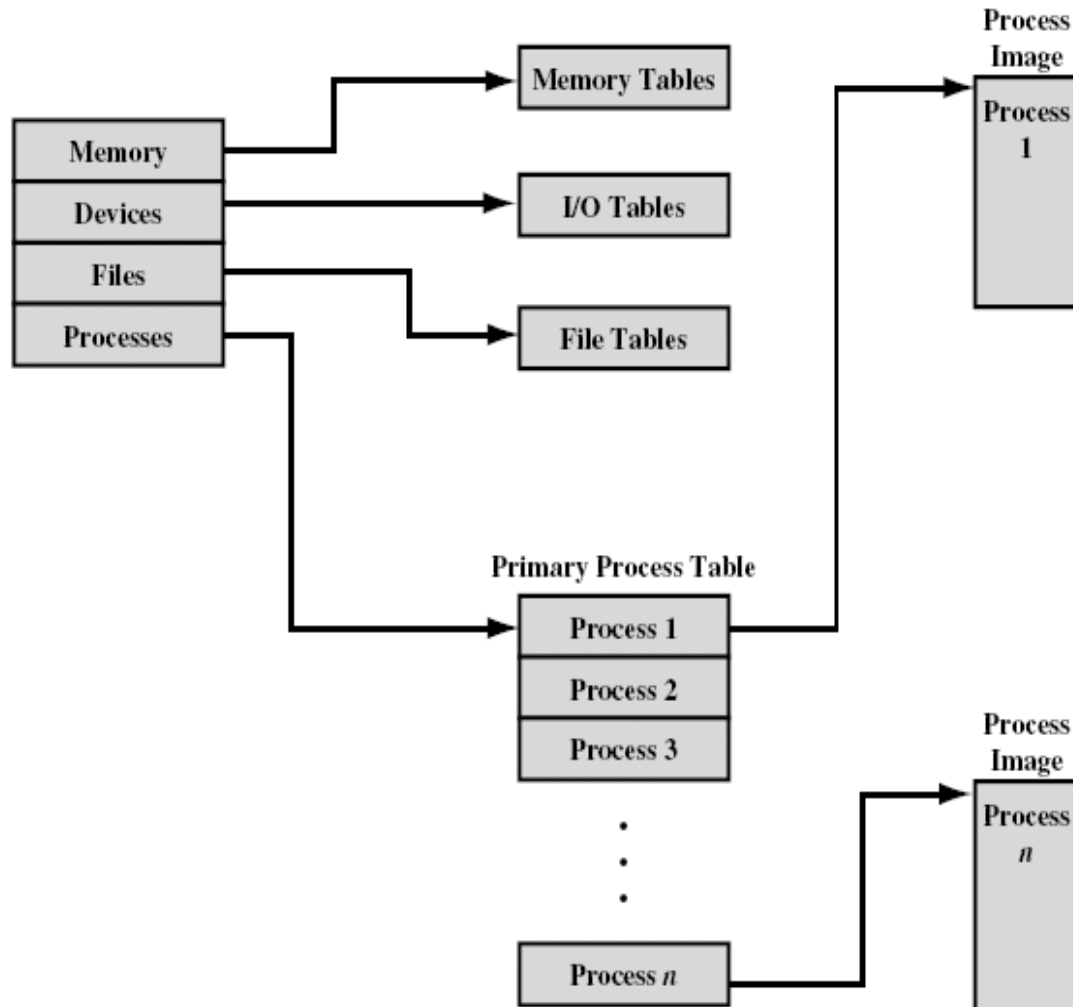
Processus (Concepts de base)

- Attributs d'un processus
 - Durant son exécution, un processus est aussi caractérisé par des attributs pointant sur des listes de ressources utilisées:



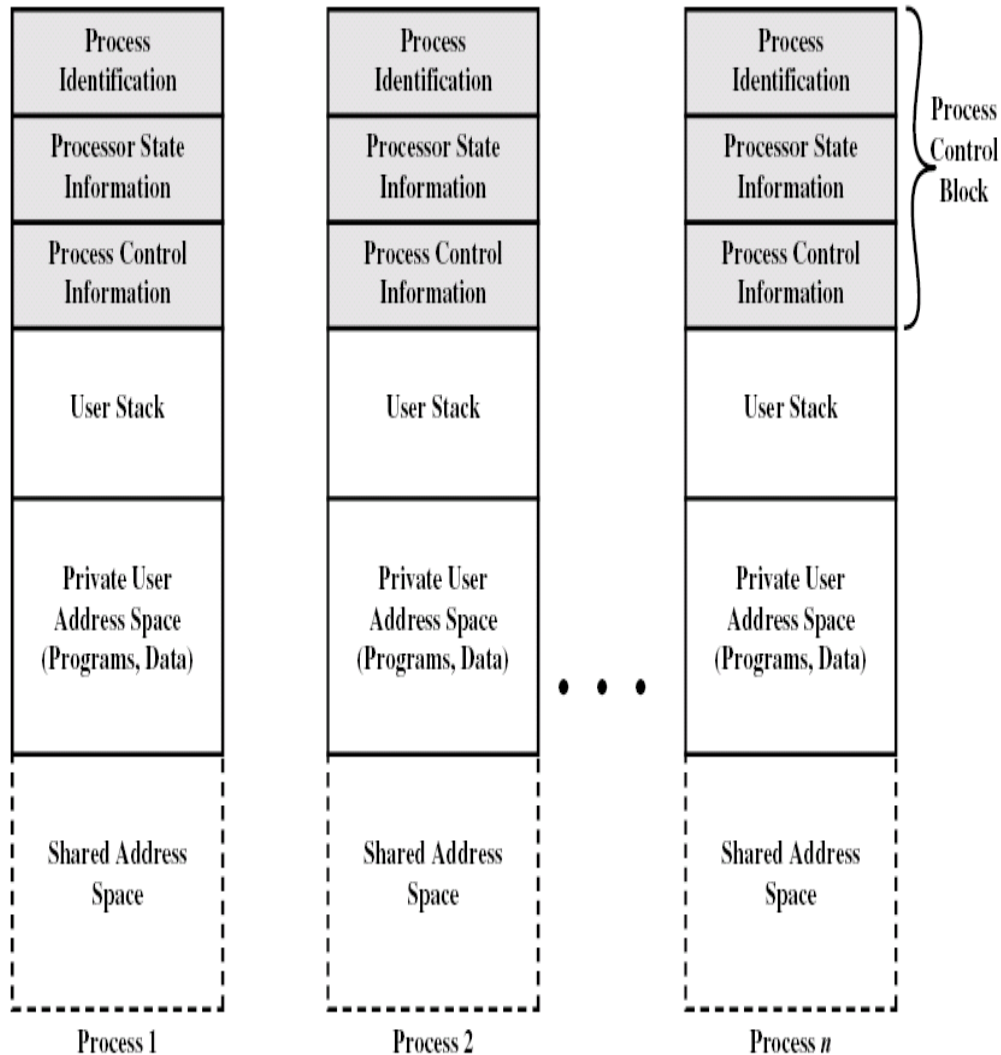
Processus (Concepts de base)

- Structure générale des listes maintenues par le SE



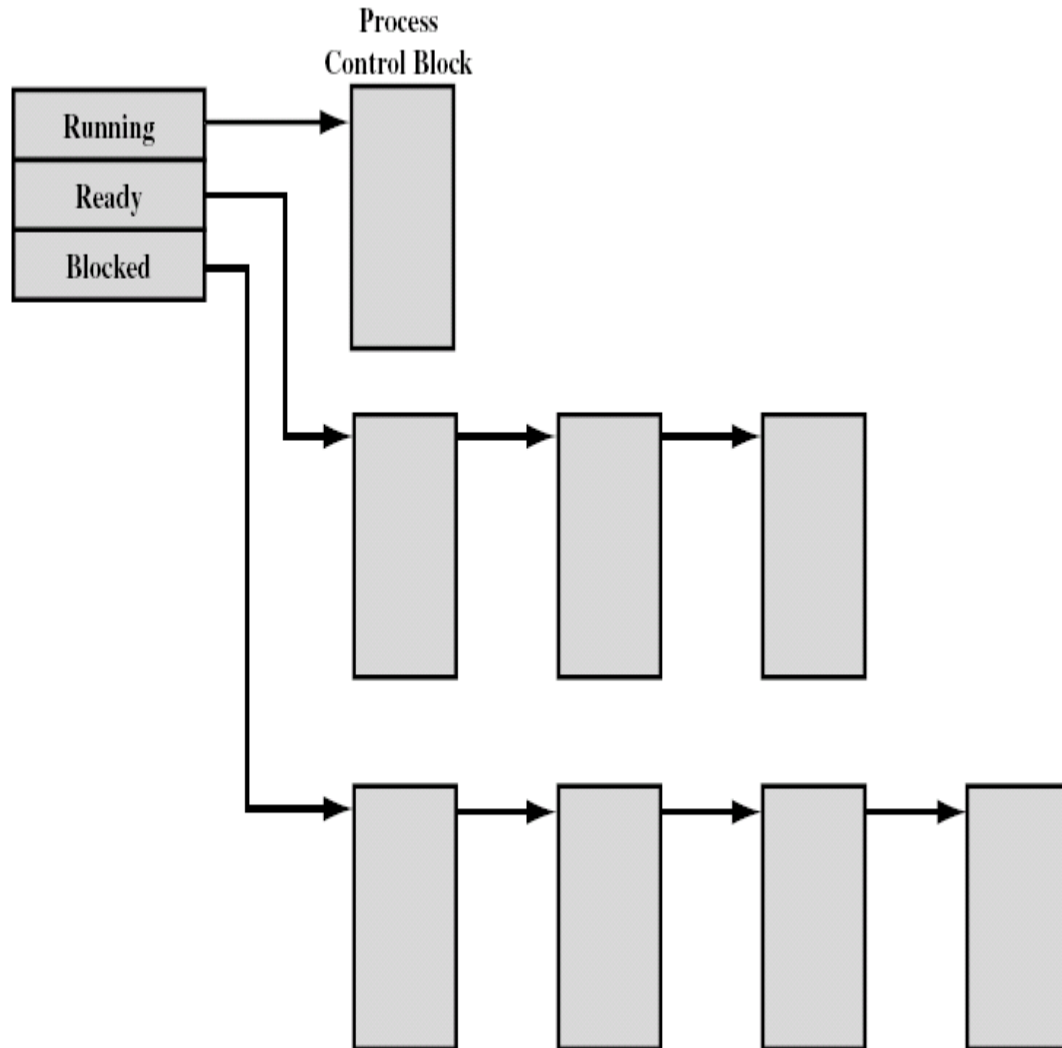
Processus (Concepts de base)

- Image des processus en mémoire virtuelle



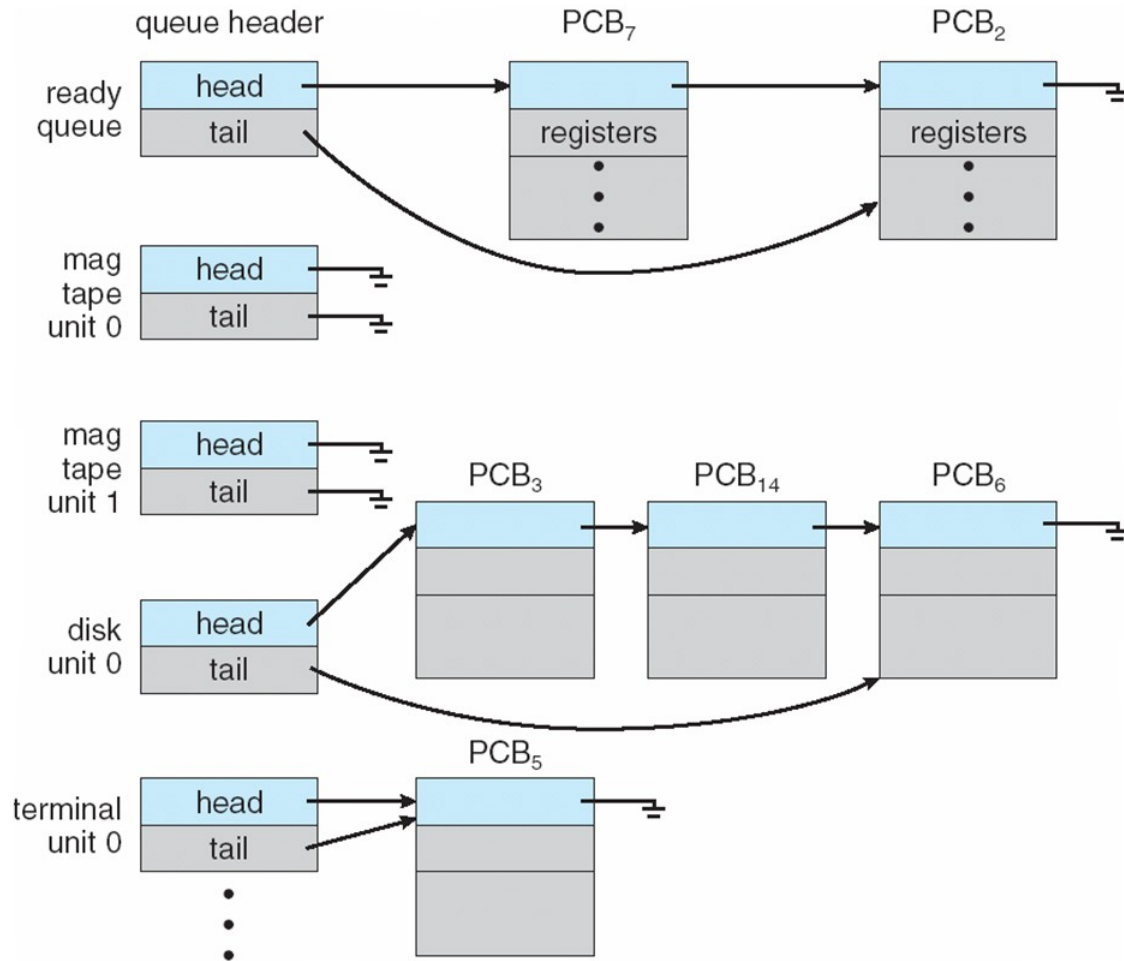
Processus (Concepts de base)

- Structure générale des listes de processus maintenues par le SE



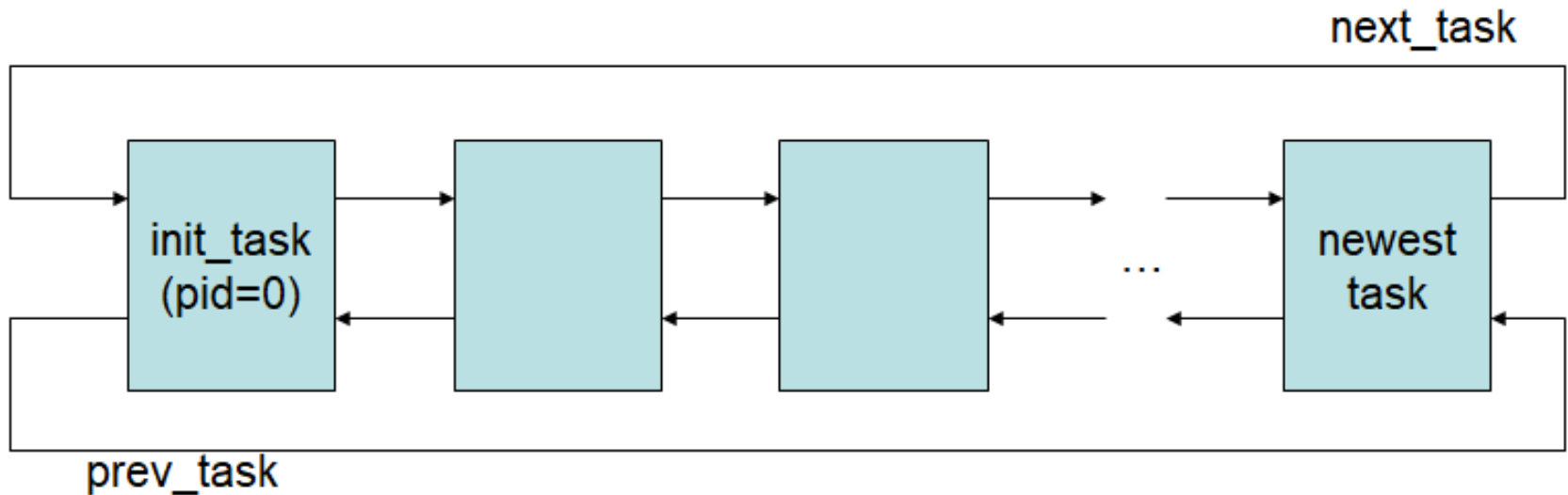
Processus (Concepts de base)

- Structure générale des listes de processus maintenues par le SE



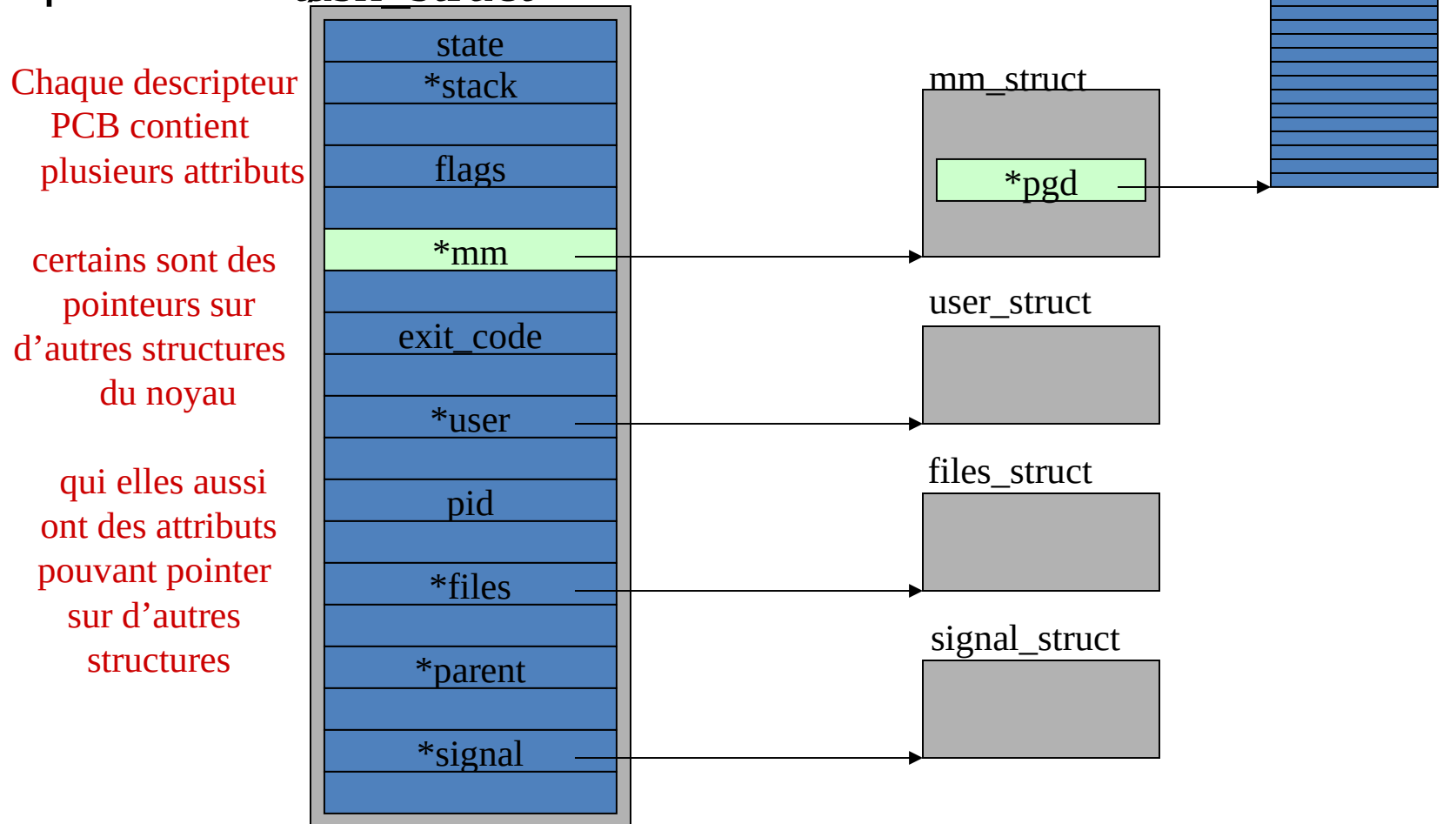
Processus (Concepts de base)

- Structure générale des listes de processus maintenues par le SE: liste doublement liée de task_struct (descripteur de processus)



Processus (Concepts de base)

- Structure générale des listes de processus maintenues par le SE: structure de données `task_struct` (descripteur de processus)

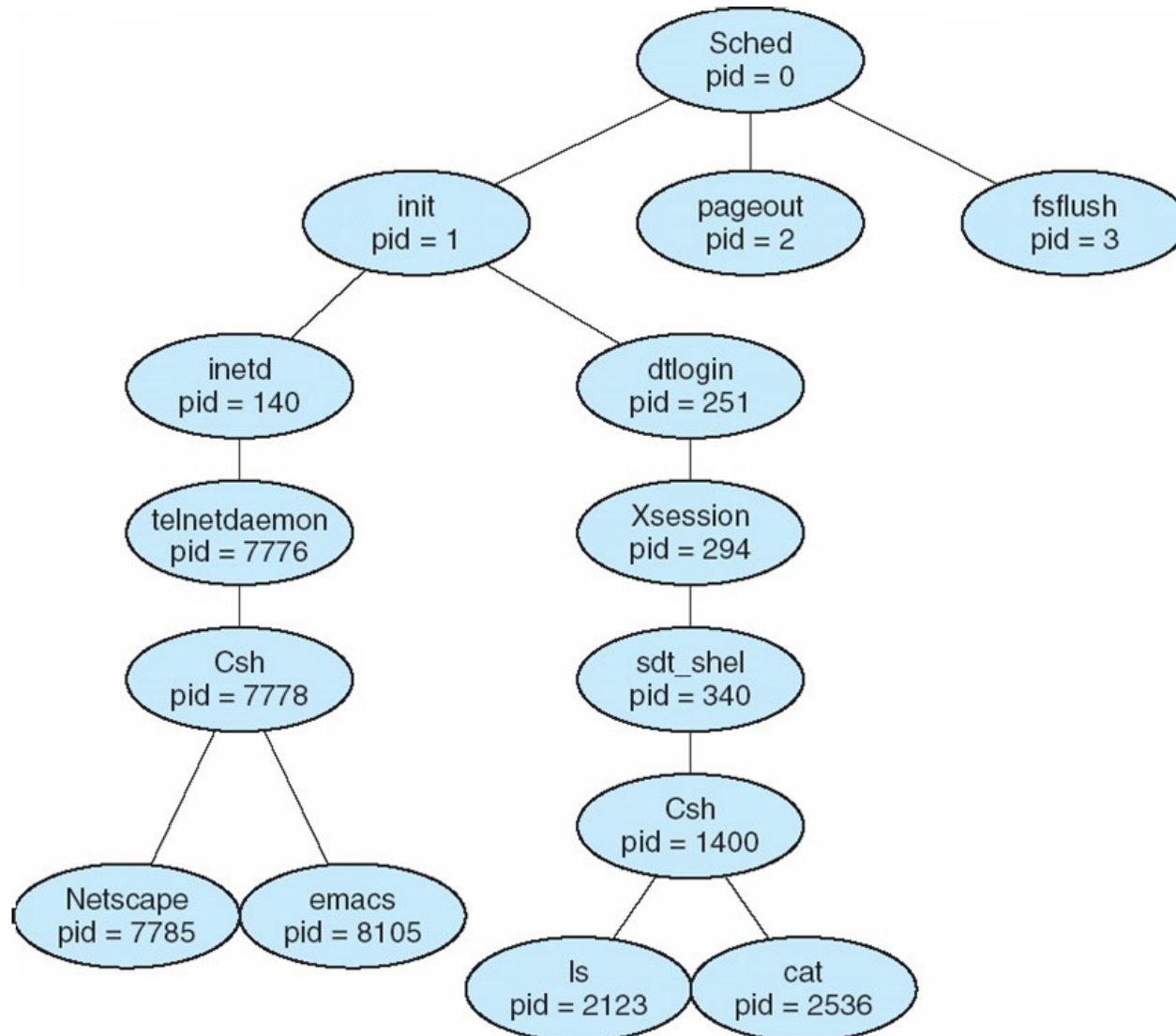


Processus (Concepts de base)

- Relation parentale entre processus
 - La création d'un processus est effectuée en dupliquant le processus courant. L'appel système ***fork()*** permet à un processus de créer une copie de lui-même (à l'exception de l'identificateur). Le processus qui exécute le ***fork()*** est le père et le nouveau processus le fils.
 - Au démarrage, LINUX crée le processus 1 qui exécute le programme ***init*** chargé d'initialisé le système. Ce processus crée lui-même d'autres processus pour accomplir diverses tâches.

Processus (Concepts de base)

- Relation parentale entre processus sur un système Solaris

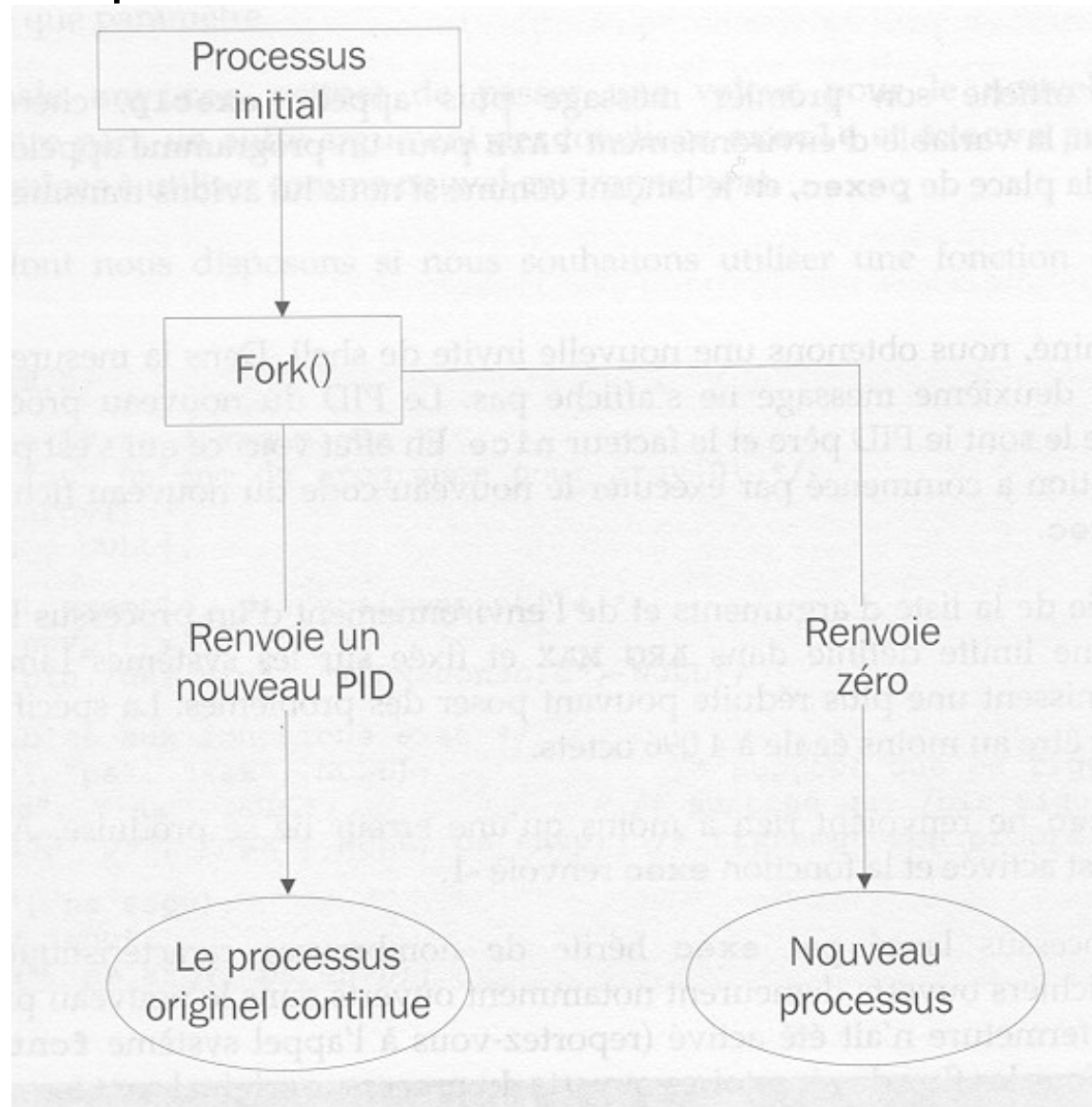


Processus (Appels système de base)

- Création des processus
 - Le processus courant peut créer un processus fils par un appel système ***fork()***. Cet appel provoque la duplication du processus courant.
 - Lorsque le ***fork()*** est exécuté par un processus, celui-ci est dupliqué en une copie qui lui est conforme sauf pour l'identificateur. Au retour de ***fork()***, deux processus, le père et le fils, exécutent le même code.
 - L'appel système ***fork()***, retourne la valeur 0 au processus fils et le ***pid*** du processus créé au processus père. La valeur -1 est retournée par le ***fork()*** si la duplication a rencontré un problème.
 - L'appel système ***exit()***, permet de terminer un processus. Cet appel ne retourne rien et requiert un entier comme argument.

Processus (Appels système de base)

- Création des processus



Processus (Appels système de base)

- Création des processus (voir ex1SIF1015.c)

```
/* -----  
Programme : ex1SIF1015.C  
Langage   : C sur UNIX  
Date      : Septembre 2001  
  
Auteur    : F. Meunier  
  
Ce programme permet d'afficher le pid du processus pere et du processus fils  
fork() retourne 0 au processus fils  
          retourne le pid du processus fils au processus pere  
----- */  
  
#include <stdio.h>  
#include <unistd.h>  
  
/* -----  
Programme principal.  
----- */  
  
void main (void)  
{  
    pid_t fork(void), val;  
  
    val = fork();  
  
    printf("Affichage du pid dans le main: %d\n",val);  
  
} /* main */
```

Utiliser *gcc*

```
helios> cc -o ex1SIF1015 ex1SIF1015.c  
helios> ex1SIF1015  
Affichage du pid dans le main: 21226  
Affichage du pid dans le main: 0  
helios> ▲
```

Processus (Appels système de base)

- Création des processus (voir ex2SIF1015.c)

```
/* -----  
Programme : ex2SIF1015.C  
Langage   : C sur UNIX  
Date      : Septembre 2001  
  
Auteur    : F. Meunier  
  
Ce programme permet d'afficher le pid du processus pere et du processus fils  
fork() retourne 0 au processus fils  
          retourne le pid du processus fils au processus pere  
----- */  
  
#include <stdio.h>  
#include <unistd.h>  
  
/* -----  
Programme principal.  
----- */  
  
void main (void)  
{  
    pid_t fork(void), val;  
  
    val = fork();  
    val = fork();  
    printf("Affichage du pid dans le main: %d\n",val);  
▲  
} /* main */
```

```
helios> cc -o ex2SIF1015 ex2SIF1015.c  
helios> ex2SIF1015  
Affichage du pid dans le main: 21258  
Affichage du pid dans le main: 0  
Affichage du pid dans le main: 21257  
Affichage du pid dans le main: 0  
helios> ▲
```

Processus (Appels système de base)

- Création des processus (voir ex3SIF1015.c)

```
/* -----  
Programme : ex3SIF1015.C  
Langage   : C sur UNIX  
Date      : Septembre 2001  
  
Auteur    : F. Meunier  
  
Ce programme permet d'afficher le pid du processus pere et du processus fils  
fork() retourne 0 au processus fils  
           retourne le pid du processus fils au processus pere  
----- */  
  
#include <stdio.h>  
#include <unistd.h>  
  
/* -----  
Programme principal.  
----- */  
  
void main (void)  
{  
    pid_t fork(void), val;  
  
    val = fork();  
  
    if(val == -1)  
        printf("Affichage du pid dans le main: %d\n",val);  
    else if(val == 0)  
        printf("Affichage du pid d'un fils: %d\n",val);  
    else  
        printf("Affichage du pid d'un pere: %d\n",val);  
  
} /* main */
```

```
helios> cc -o ex3SIF1015 ex3SIF1015.c  
helios> ex3SIF1015  
Affichage du pid d'un pere: 21281  
Affichage du pid d'un fils: 0  
helios> ▲
```


Processus (Appels système de base)

- Création des processus (voir ex4SIF1015.c)

```
/* -----  
Programme : ex4SIF1015.C  
Langage   : C sur UNIX  
Date      : Septembre 2001  
  
Auteur    : F. Meunier  
  
Ce programme permet d'afficher le pid du processus pere et du processus fils  
fork() retourne 0 au processus fils  
           retourne le pid du processus fils au processus pere  
----- */  
  
#include <stdio.h>  
#include <unistd.h>  
  
/* -----  
Programme principal.  
----- */  
  
void main (void)  
{  
    pid_t fork(void), val;  
  
    val = fork();  
    val = fork();  
  
    if(val == -1)  
        printf("Affichage du pid dans le main: %d\n",val);  
    else if(val == 0)  
        printf("Affichage du pid d'un fils: %d\n",val);  
    else  
        printf("Affichage du pid d'un pere: %d\n",val);  
  
} /* main */
```

```
helios> cc -o ex4SIF1015 ex4SIF1015.c  
helios> ex4SIF1015  
Affichage du pid d'un pere: 21301  
Affichage du pid d'un fils: 0  
Affichage du pid d'un fils: 0  
Affichage du pid d'un pere: 21300  
helios> ▲
```

Processus (Appels système de base)

- Création des processus (voir ex5SIF1015.c)

Programme : ex5SIF1015.C
Langage : C sur UNIX
Date : Septembre 2001

Auteur : F. Meunier

Ce programme permet de créer deux processus

processus père affiche un a
processus fils affiche un b

```
_____/

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

/* _____
Programme principal.
_____ */

void main (void)
{
    pid_t fork(void), val;
    void affiche_a(void), affiche_b(void);

    val = fork();

    if(val == -1)
        printf("Probleme de duplication");
    else if(val == 0)
        affiche_a();
    else
        affiche_b();
} /* main */

void affiche_a()
{
    while(1)
        printf("a\n");
}

void affiche_b()
{
    while(1)
        printf("b\n");
}
```

Processus (Appels système de base)

- Création des processus (voir ex5SIF1015-V2.c)

```
int cntLettre = 0;
void main (void)
{
    int NBProc = 2;
    int i = 0;
    pid_t fork(void), val=1;
    void affiche_char(char c);

    // Le pere démarre NBProc processus
    // Les fils affichent chacun leur lettre

    for(i=0; i != NBProc;i++)
    {
        val = fork();
        if(val == 0)
        {
            affiche_char('a'+ i);
        }
    }

} /* main */

void affiche_char(char c)
{
    nice(-20);
    while(1)
        printf("Le processus %d affiche le char %c pour la %d ieme fois\n", getpid(), c, cntLettre++);
}
```

Processus (Appels système de base)

- Création des processus (voir sur le site suivant: **fork.c**)
 - Voir le site: **<http://csapp.cs.cmu.edu/public/code.html>** pour des exemples de codes en langage C
 - Voir le fichier **csapp.h** contenant les **include** et les prototypes de plusieurs fonctions:
<http://csapp.cs.cmu.edu/public/ics/code/include/csapp.h>
 - Voir le fichier **csapp.c** contenant plusieurs fonctions de gestion des erreurs (ex: **Fork()**):

```
#include "csapp.h"
```

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

blic/ics/code/src/csapp.c

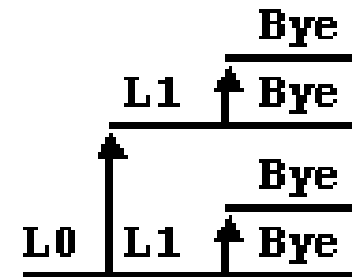
```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

Processus (Appels système de base)

- Création des processus

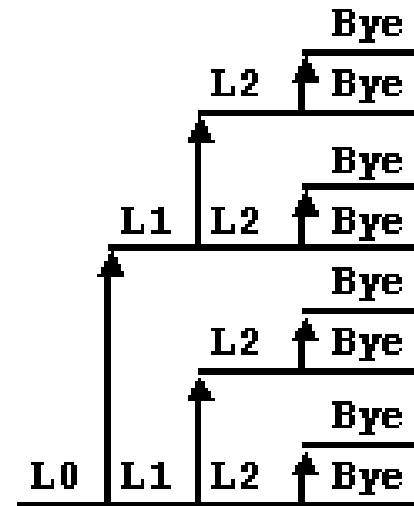
```
void fork2()  
{  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("Bye\n");  
}
```



Processus (Appels système de base)

- Création des processus

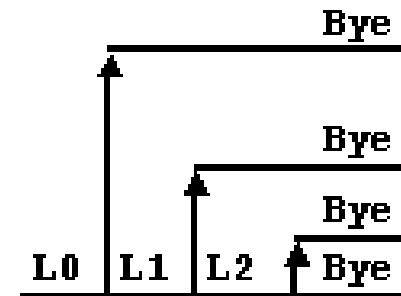
```
void fork3()  
{  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("L2\n");  
    fork();  
    printf("Bye\n");  
}
```



Processus (Appels système de base)

- Création des processus

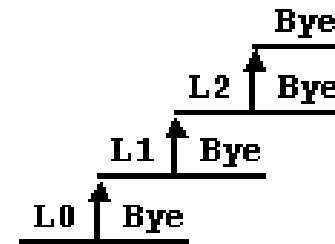
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Processus (Appels système de base)

- Création des processus

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Processus (Appels système de base)

- Création des processus (exemple d'appel système **exec()**)

```
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```

Processus (Appels système de base)

- Création des processus (Appel système ***wait()***)

`wait, waitpid, waitid` - wait for process to change state

SYNOPSIS [top](#)

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

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

```
pid_t wait(int *wstatus);
```

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

```
/* This is the glibc and POSIX interface; see  
   NOTES for information on the raw system call. */
```

Processus (Appels système de base)

- Création des processus (Appel système ***wait()***)

DESCRIPTION

[top](#)

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of [sigaction\(2\)](#)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

wait() and **waitpid()**

The **wait()** system call suspends execution of the calling process until one of its children terminates. The call **wait(&wstatus)** is equivalent to:

```
waitpid(-1, &wstatus, 0);
```

Processus (Appels système de base)

- Création des processus (Appel système de la famille **exec()**)

execl, **execvp**, execl, execv, execvp, execvpe - execute a file

Synopsis

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

```
extern char **environ;
```

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

```
int execvp(const char *file, const char *arg, ...);
```

```
int execl(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 execvpe(const char *file, char *const argv[],  
char *const envp[]);
```

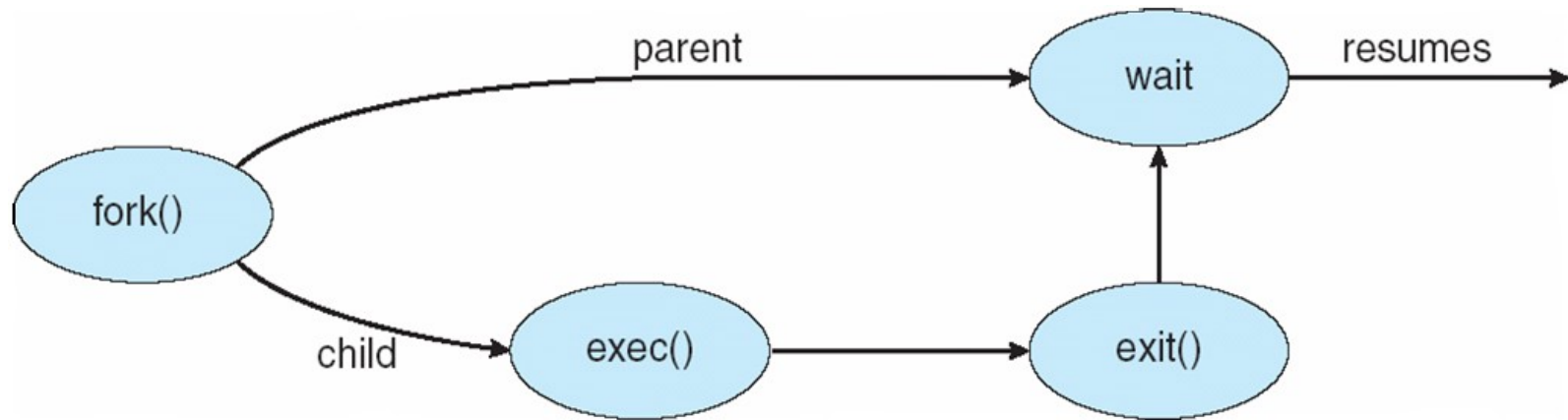
Processus (Appels système de base)

- Création des processus (exemple d'appel système **exec()**)
 - **int execl(char *path, char *arg0, char *arg1, ..., 0)**
 - Charge et exécute l'exécutable situé à path avec les arguments args arg0, arg1, ...
 - path est le chemin complet de l'exécutable
 - arg0 devient le nom du processus
 - arg0 est souvent comme le path, ou contient le nom du fichier exécutable
 - La liste d'arguments de l'exécutable commence à arg1, etc.
 - La liste de args est terminée par un (char *)0, (NULL)
 - returns -1 si erreur, sinon rien

```
main() {
    if (fork() == 0) {
        execl("/usr/bin/cp", "cp", "foo", "bar", 0);
    }
    wait(NULL);
    printf("copy completed\n");
    exit();
}
```

Processus (Appels système de base)

- Création des processus (exemple d'appel système **exec()**)



Processus Fils exécute une commande **ls**

Processus (Appels système de base)

- Création des processus (WIN32)

```
#include <windows.h>
#include <stdio.h>

int main( VOID )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( NULL,    // No module name (use command line).
        "C:\\WINDOWS\\system32\\mspaint.exe", // Command line.
        NULL,                    // Process handle not inheritable.
        NULL,                    // Thread handle not inheritable.
        FALSE,                   // Set handle inheritance to FALSE.
        0,                       // No creation flags.
        NULL,                    // Use parent's environment block.
        NULL,                    // Use parent's starting directory.
        &si,                     // Pointer to STARTUPINFO structure.
        &pi )                   // Pointer to PROCESS_INFORMATION structure.
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return -1;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Processus (Appels système de base)

- Élimination des processus ZOMBIE
 - Un processus enfant terminé (ZOMBIE) ne sera pas sorti du système avant que le processus père exécute un appel système ***waitpid()***. Cet appel retourne le ***pid*** du processus enfant terminer, ou **0** si aucun processus enfant est terminé et **-1** si une erreur est survenue.
 - Le premier paramètre ***pid*** est **-1** si on veut éliminer tous les processus enfant attachés à un processus père et **> 0** si on veut éliminer un processus enfant particulier.
 - Le paramètre ***status*** est un entier qui représente l'état de sortie d'un processus enfant (voir ***exit(status)***)
 - La fonction ***WIFEXITED(status)*** retourne vrai si le processus enfant s'est normalement terminé par un appel système ***exit(0)***; ou un ***return()***;
 - La ***WEXITSTATUS(status)*** retourne l'état de sortie d'un processus enfant

Processus (Appels système de base)

- Élimination des processus ZOMBIE

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9      00:00:00 tcsh
 6639 tttyp9      00:00:03 forks
 6640 tttyp9      00:00:00 forks <defunct>
 6641 tttyp9      00:00:00 ps
linux> kill 6639
[1] Terminated
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9      00:00:00 tcsh
 6642 tttyp9      00:00:00 ps
```

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
            getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
            getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

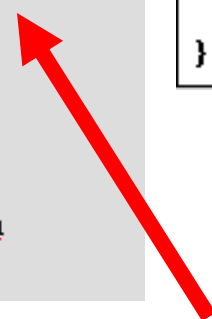
• ps processus enfants ZOMBIE sont "defunct"

• Un kill du processus père élimine le processus enfant du système

Processus (Appels système de base)

- Élimination des processus ZOMBIE

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttys00    00:00:00 tcsh
 6676 ttys00    00:00:06 forks
 6677 ttys00    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttys00    00:00:00 tcsh
 6678 ttys00    00:00:00 ps
```



```
void forks()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID =
               %d\n",
               getpid());
        exit(0);
    }
}
```

Processus enfant s'exécute seul

Processus (Appels système de base)

- Élimination des processus ZOMBIE (Voie exemple ***waitpid1.c*** sur le site de ***csapp***)

```
#include "csapp.h"
#define N 2

int main()
{
    int status, i;
    pid_t pid;

    for (i = 0; i < N; i++)
        if ((pid = Fork()) == 0) /* child */
            exit(100+i);

    /* parent waits for all of its children to terminate */
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n",
                pid, WEXITSTATUS(status));
        else
            printf("child %d terminated abnormally\n", pid);
    }
    if (errno != ECHILD)
        unix_error("waitpid error");

    exit(0);
}
```

Processus (Appels système de base)

- Élimination des processus ZOMBIE (Voie exemple ***waitpid1.c*** sur le site de ***csapp***)

Pour un nombre de processus fils $N = 5$

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

Processus (Appels système de base)

- Lecture des attributs du processus courant
 - Plusieurs appels système permettent à un processus d'obtenir ses attributs:
 - ***pid_t getpid(void);*** : retourne le ***pid*** du processus courant
 - ***pid_t getppid(void);*** : retourne le ***pid*** du processus père du processus courant
 - ***pid_t getuid(void);*** : retourne le ***uid*** du processus courant
- Informations de comptabilité
 - L'appel système ***getrusage()*** permet à un processus de connaître les ressources qu'il a consommées
 - L'appel système ***times()*** permet au processus de connaître le temps processeur qu'il a consommé
 - ***#include<sys/times.h>***
 - ***clock_t times(struct tms *buff);***
 - ***buff*** est un pointeur sur une structure ***tms***.
 - ***time_t tms_utime***: temps (seconde) processeur consommé par le processus en mode user
 - ***time_t tms_stime***: temps (seconde) processeur consommé par le processus en mode noyau
 - ***time_t tms_ctime***: temps (seconde) processeur consommé par les processus fils en mode user
 - ***time_t tms_cstime***: temps (seconde) processeur consommé par les processus fils en mode noyau

Processus (Appels système de base)

- Exécution de programme
 - Un nouveau processus, créé par un appel système ***fork()***, est un duplicata conforme de son processus père, exécutant donc le même programme.
 - Un appel système de la famille des fonctions ***exec()*** permet à un processus d'exécuter un nouveau programme.
 - L'appel à une de ces fonctions permet de remplacer l'espace mémoire du processus appelant par le code (programme) et les données de la nouvelle application.
 - Les variantes des fonctions ***exec()*** découlent de leur façon de transmettre les arguments

#include <unistd.h>

int execl(char *path, char *arg0,,0);

- ***path***: est le chemin où se trouve le fichier exécutable
- ***arg0 ...*** : arguments fournis au processus à exécuter

Processus (Appels système de base)

- Exécution de programme (*execl()*)

```
/* -----  
Programme : ex6PG1SIF1015.C  
Langage   : C sur UNIX  
Date      : Septembre 2001  
  
Auteur    : F. Meunier  
  
Ce programme permet d'afficher le nom du prog. principal et trois lettres  
----- */  
  
#include <stdio.h>  
  
/* -----  
Programme principal.  
----- */  
void main (int argc, char **argv)  
{  
    int i;  
  
    printf("%s:", argv[0]);  
    for(i=1; i < argc; i++)  
        printf("%s ", argv[i]);  
  
    printf("\n");  
  
} /* main */
```

Processus (Appels système de base)

- Exécution de programme

```
/* -----  
Programme : ex6MAINSIF1015.C  
Langage   : C sur UNIX  
Date      : Septembre 2001  
  
Auteur    : F. Meunier  
  
Ce programme permet de faire l'appel au programme à exécuter  
----- */  
  
#include <stdio.h>  
#include <unistd.h>  
  
/* -----  
Programme principal.  
----- */  
  
void main ()  
{  
    printf("Le processus parent démarre un autre processus\n");  
    execl("ex6PG1SIF1015", "pgm1", "a", "b", "c", 0);  
}  
/* main */
```

```
helios> cc -o ex6PG1SIF1015 ex6PG1SIF1015.c  
helios> cc -o ex6MAINSIF1015 ex6MAINSIF1015.c  
helios> ex6MAINSIF1015  
Le processus parent démarre un autre processus  
pgm1:a b c  
helios> ▲
```


Processus (Appels système de base)

- Exécution de programme
 - Il est possible d'exécuter un programme à partir d'un autre et de créer par le fait même un autre processus.
 - L'appel système ***system()*** exécute la commande passée en argument sous forme d'une chaîne de caractères.

#include <stdlib.h>

int system(char *chaine);

- ***chaine***: chaîne de caractères correspondant à la commande
- La fonction ***system()*** s'exécute comme si la commande avait été envoyée à un shell. ***system()*** retourne 127 si un shell ne peut être lancé pour exécuter la commande, -1 si une autre erreur est survenue. Sinon, ***system()*** retourne le code de retour de la commande.
- Cette approche de lancement de programme est moins efficace puisqu'elle implique le démarrage d'un autre processus (un shell) au lieu du simple remplacement du processus à partir duquel s'effectue l'appel à la fonction ***system()***.

Processus (Appels système de base)

- Exécution de programme (***system()***)

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("Exécution de ps avec system\n");
    system("ps -ax");
    printf("Terminé.\n");
    exit(0);
}
```

\$./système

Exécution de ps avec system

PID	TTY	STAT	TIME	COMMAND
-----	-----	------	------	---------

1	?	S	0:00	init
---	---	---	------	------

7	?	S	0:00	update (bdf flush)
---	---	---	------	--------------------

...

146	v01	S N	0:00	oclock
-----	-----	-----	------	--------

256	pp0	S	0:00	./system
-----	-----	---	------	----------

257	pp0	R	0:00	ps -ax
-----	-----	---	------	--------

Terminé.

Processus (Appels système de base)

- Exécution de programme (***system()***, fonction ***executeFile()***)

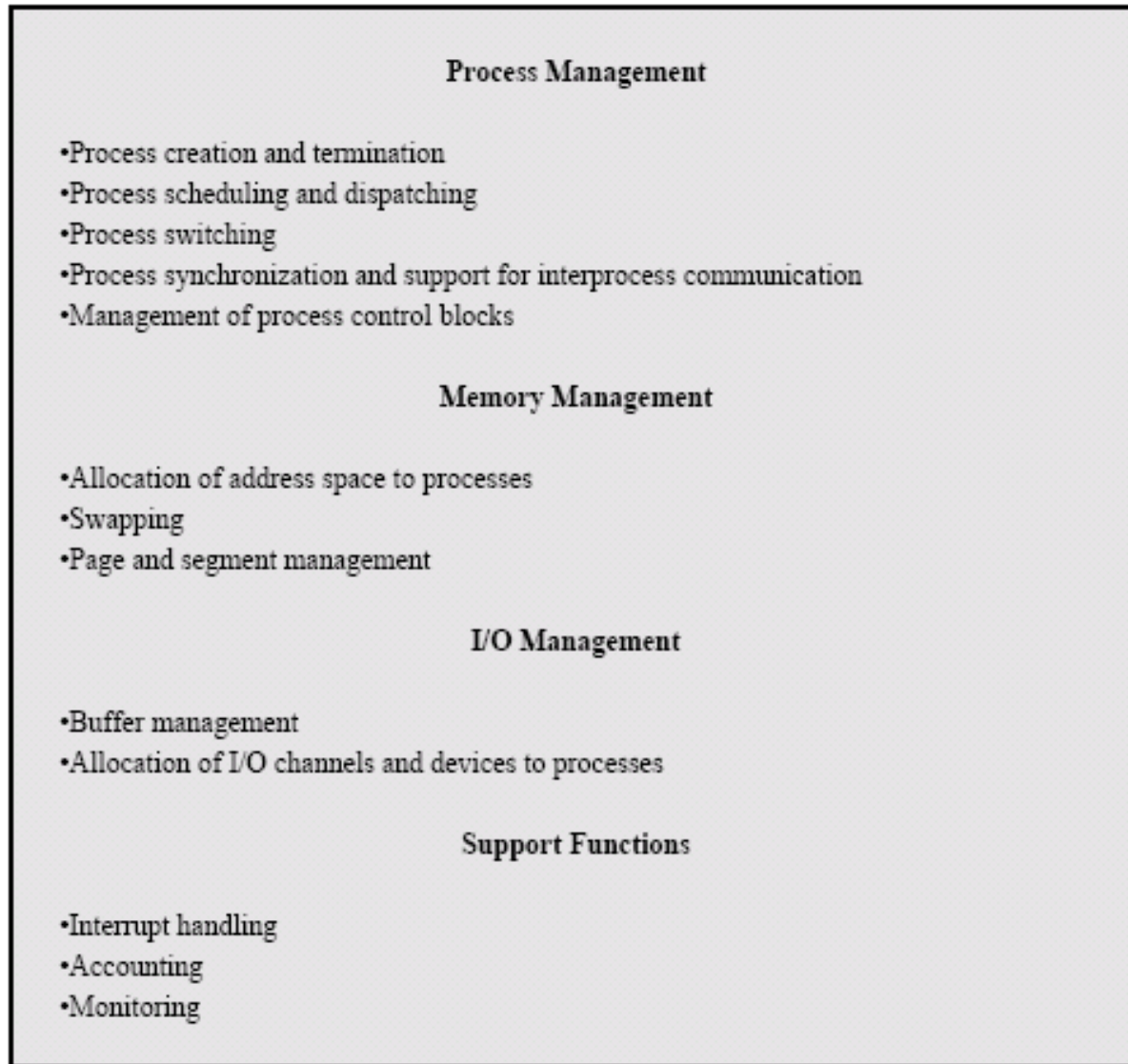
```
//#####  
//#  
//# Execute le fichier source .c  
//#  
void executeFile(const char* sourcefname){  
    char command[100];  
    char nameC[100];  
    char name[100];  
  
    FILE *f;  
    //Ouverture du fichier MakeCVS en mode "wt" : [w]rite [t]ext  
    f = fopen("MakeCVS", "wt");  
    if (f==NULL)  
        error(2, "ExecuteFile: Erreur lors de l'ouverture du fichier pour écriture en mode texte.");  
  
    strcpy(nameC,sourcefname);  
  
    fprintf(f,"fichCVSEXE: %s\n",sourcefname);  
    fprintf(f,"\tgcc -o fichCVSEXE %s\n",sourcefname);  
  
    //Fermeture du fichier  
    fclose(f);  
  
    // make du fichier MakeCVS  
    sprintf(command, "make -f MakeCVS");  
    system(command);  
  
    // execution du fichier fichCVSEXE  
    sprintf(command, "./fichCVSEXE");  
    system(command);  
}
```

Processus (Appels système de base)

- Suspension d'un processus
 - L'appel système ***sleep()*** retourne le temps restant de dormance d'un processus et requiert comme argument le temps de dormance en ***secs.***

Processus (Concepts avancés)

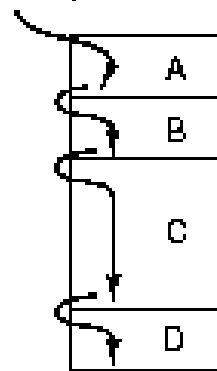
- Fonctions typiques d'un SE



Processus (Concepts avancés)

- Ordonnancement des processus

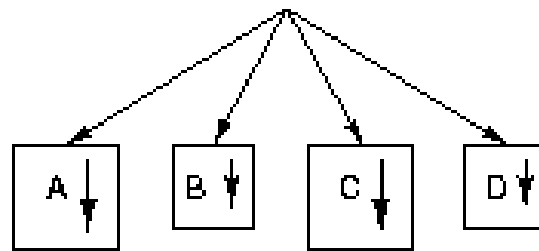
One program counter



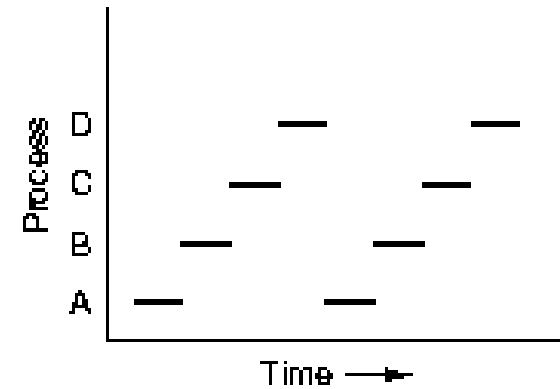
(a)

Process switch

Four program counters



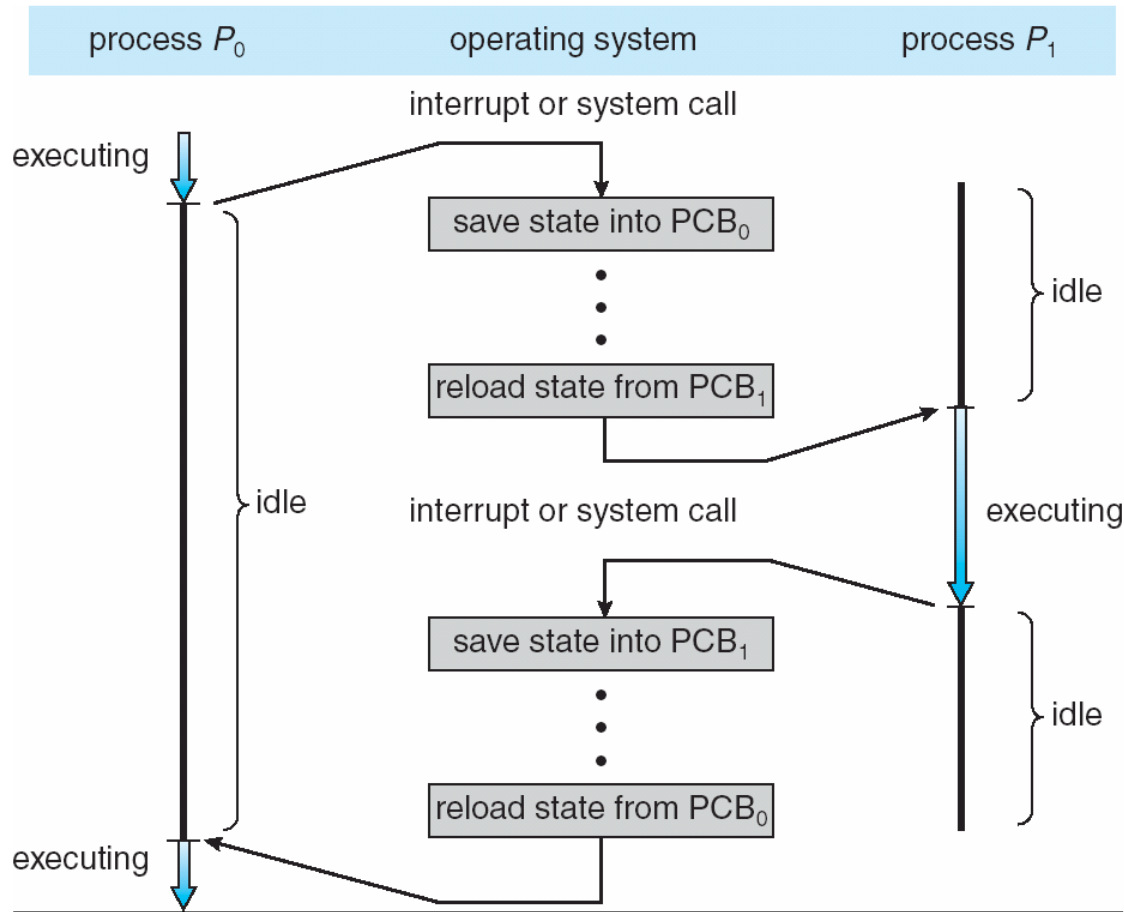
(b)



(c)

Processus (Concepts avancés)

- Ordonnancement des processus (changement de contexte)



Processus (Concepts avancés)

- Ordonnancement des processus (critères)

User Oriented, Performance Related	
Turnaround time	This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.
Response time	For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user's point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.
Deadlines	When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.
User Oriented, Other	
Predictability	A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.

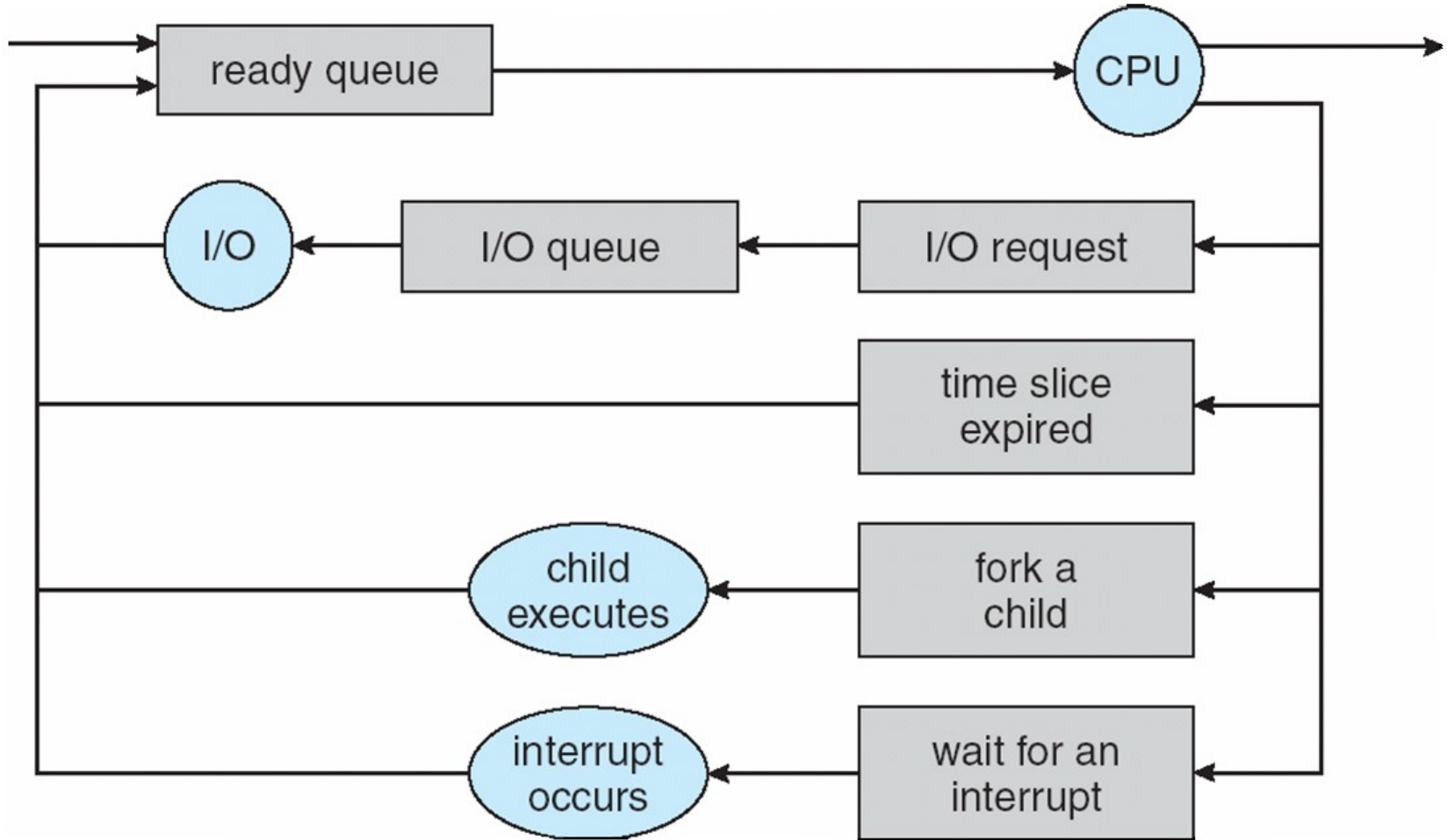
Processus (Concepts avancés)

- Ordonnancement des processus (critères)

System Oriented, Performance Related	
Throughput	The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.
Processor utilization	This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.
System Oriented, Other	
Fairness	In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.
Enforcing priorities	When processes are assigned priorities, the scheduling policy should favor higher-priority processes.
Balancing resources	The scheduling policy should keep the resources of the system busy. Processes that will underutilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling.

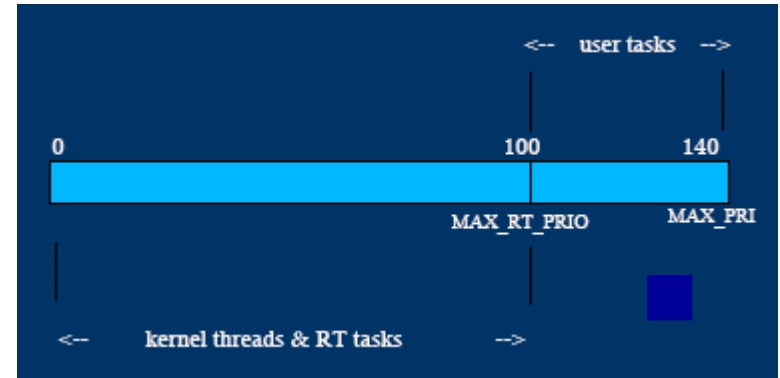
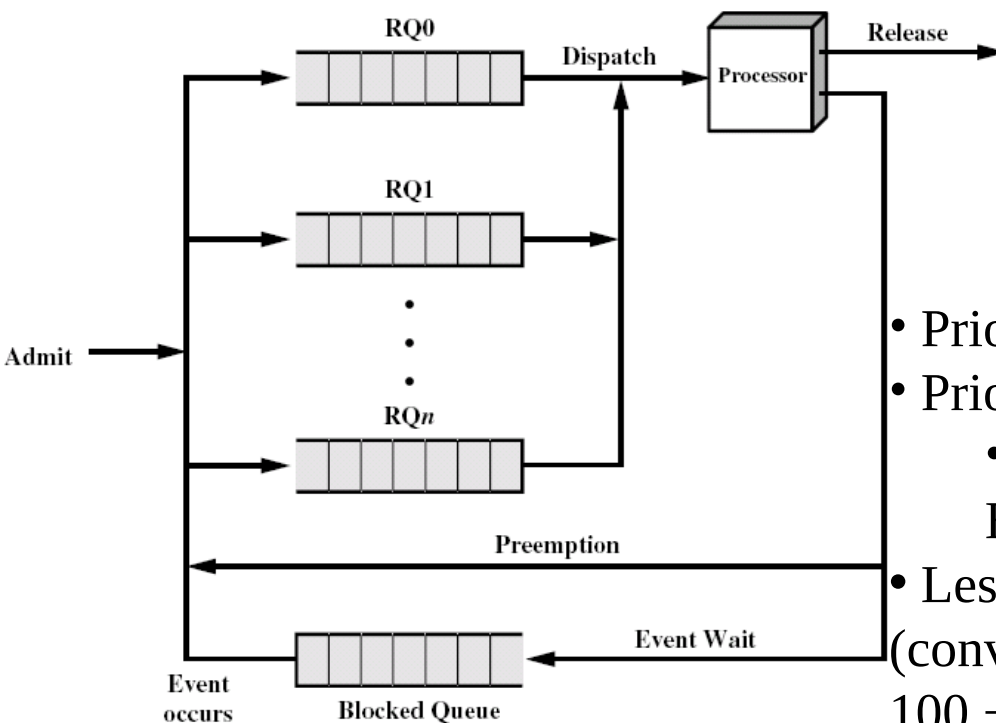
Processus (Concepts avancés)

- Ordonnancement des processus (Basée sur la notion de file d'attente)



Processus (Concepts avancés)

- Ordonnancement des processus basé sur la priorité



- Priorité statique [0, 139]
- Priorité dynamique [0, 139]
 - Fonction du type de tâche (CPU Bound ou I/O bound)
- Les processus/threads en temps partagé (user) (conventionnels) ont une priorité entre [0, 139] 100 + nice value [-20, 19] (0 par défaut). Seul l'admin. peut utiliser une nice value < 0


Processus (Concepts avancés)

- Ordonnancement des processus basé sur la priorité (P)
 - ▶ Chaque processus conventionnel a une priorité statique (PS)
 - ▶ Utilisée par le scheduler pour ordonner les processus conventionnels entre eux
 - ▶ Valeur de 100 (haute priorité) à 139 (basse priorité)
 - ▶ Un nouveau processus hérite de la priorité de son parent
 - ▶ Un processus de basse priorité aura un quantum faible

If (SP < 120): qt = (140 - SP) × 20
if (SP ≥ 120): qt = (140 - SP) × 5

Processus (Concepts avancés)

- Ordonnancement des processus basé sur la priorité (priorité dynamique: Noyau 2.6)

- ▶ Chaque processus a en plus une priorité dynamique (PD) de 100 (plus haute) à 139 (plus basse)
 - ▶ La priorité dynamique sert au scheduler à choisir le processus
 - ▶ $PD = \max(100, \min(SP - \text{bonus} + 5, 139))$
 - ▶ Bonus est une valeur entre 0 et 10
 - ▶ Une valeur < 5 est une pénalité qui baissera la PD
 - ▶ Valeur ≥ 5 augmente la priorité dynamique
 - ▶ La valeur du bonus dépend de l'histoire passée du processus
 - ▶ Son temps moyen de sommeil
 - ▶ Temps moyen de sommeil
 - ▶ Mesuré en nanosecondes, jamais plus grand que 1 seconde
 - ▶ Dépend de l'état (TASK_INTERRUPTIBLE vs TASK_UNINTERRUPTIBLE)
 - ▶ Diminue quand le processus s'exécute
- sleep_avg  [0, 10] / [5, -5]
< 100 ms, ≈ 1 sec

Processus (Concepts avancés)

- Ordonnancement des processus (politiques possibles)

	Selection Function	Decision Mode	Throughput	Response Time	Overhead	Effect on Processes	Starvation
FCFS	$\max[w]$	Nonpreemptive	Not emphasized	May be high, especially if there is a large variance in process execution times	Minimum	Penalizes short processes; penalizes I/O bound processes	No
Round Robin	constant	Preemptive (at time quantum)	May be low if quantum is too small	Provides good response time for short processes	Minimum	Fair treatment	No
SPN	$\min[s]$	Nonpreemptive	High	Provides good response time for short processes	Can be high	Penalizes long processes	Possible
SRT	$\min[s - e]$	Preemptive (at arrival)	High	Provides good response time	Can be high	Penalizes long processes	Possible
HRRN	$\max\left(\frac{w + s}{s}\right)$	Nonpreemptive	High	Provides good response time	Can be high	Good balance	No
Feedback	(see text)	Preemptive (at time quantum)	Not emphasized	Not emphasized	Can be high	May favor I/O bound processes	Possible

w = time spent in system so far, waiting and executing
 e = time spent in execution so far
 s = total service time required by the process, including e

T_s = Service time

T_r = Residence time (wait+service)

SPN: Shortest Process Next SRT: Shortest Remaining Time

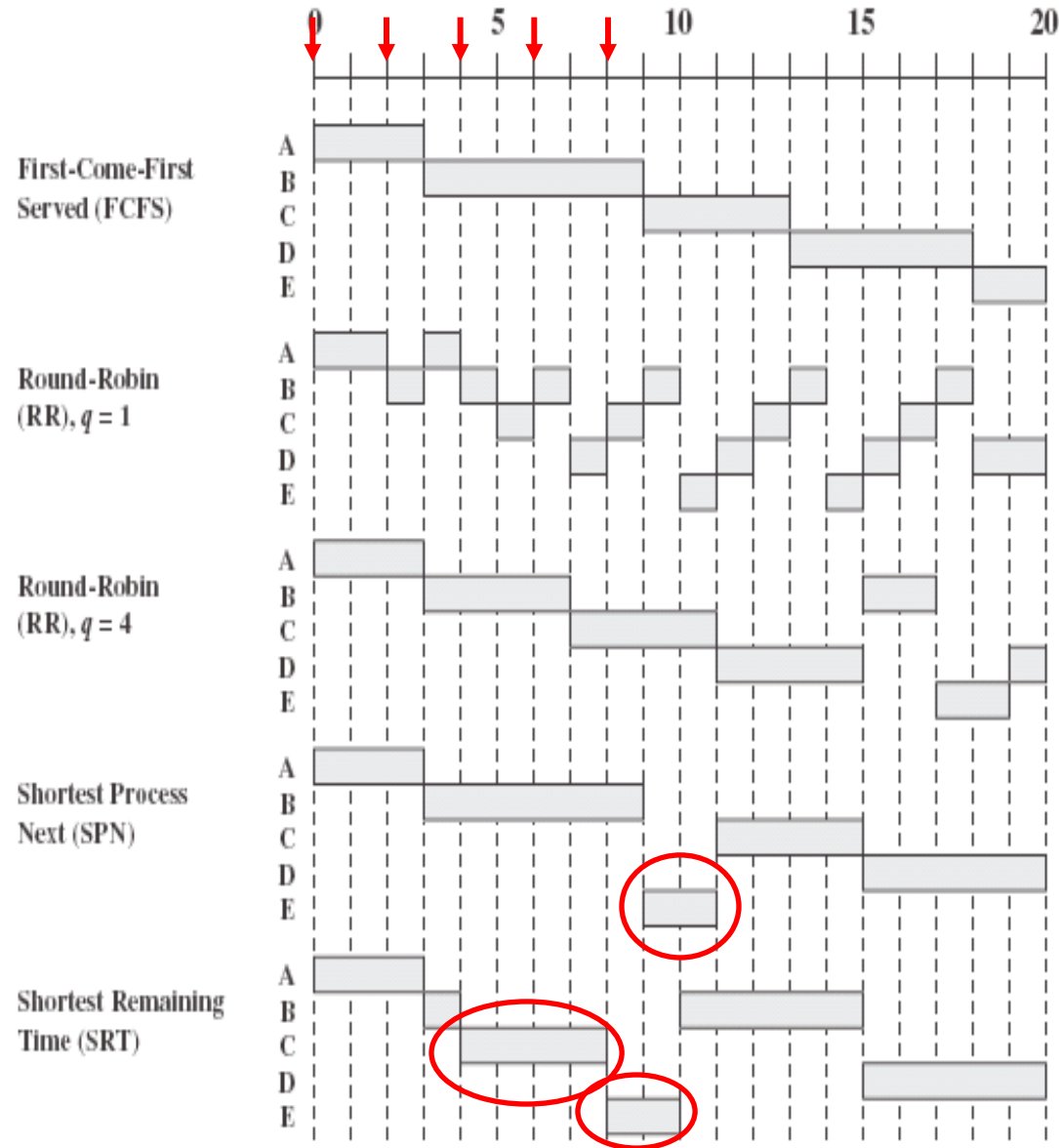
Processus (Concepts avancés)

- Ordonnancement des processus (comparaison entre politiques)

	Process	A	B	C	D	E	Mean
	Arrival Time	0	2	4	6	8	
	Service Time (T_s)	3	6	4	5	2	
FCFS	Finish Time	3	9	13	18	20	
	Turnaround Time (T_r)	3	7	9	12	12	8.60
	T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$	Finish Time	4	18	17	20	15	
	Turnaround Time (T_r)	4	16	13	14	7	10.80
	T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$	Finish Time	3	17	11	20	19	
	Turnaround Time (T_r)	3	15	7	14	11	10.00
	T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN	Finish Time	3	9	15	20	11	
	Turnaround Time (T_r)	3	7	11	14	3	7.60
	T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT	Finish Time	3	15	8	20	10	
	Turnaround Time (T_r)	3	13	4	14	2	7.20
	T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN	Finish Time	3	9	13	20	15	
	Turnaround Time (T_r)	3	7	9	14	7	8.00
	T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$	Finish Time	4	20	16	19	11	
	Turnaround Time (T_r)	4	18	12	13	3	10.00
	T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$	Finish Time	4	17	18	20	14	
	Turnaround Time (T_r)	4	15	14	14	6	10.60
	T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

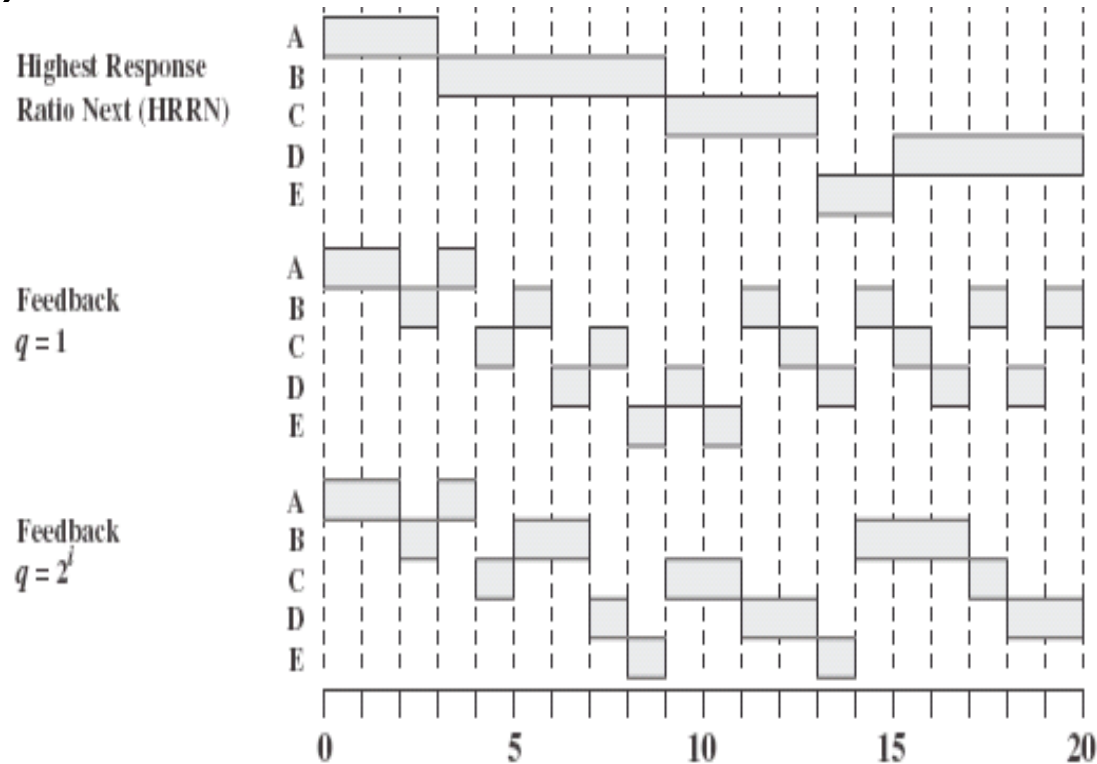
Processus (Concepts avancés)

- Ordonnancement des processus (comparaison entre politiques)



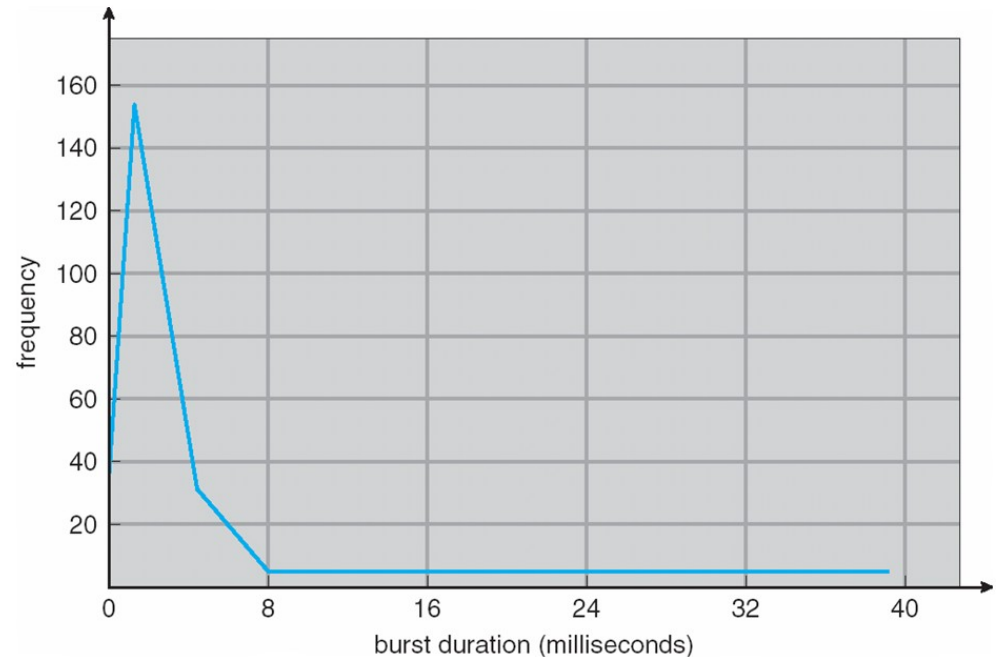
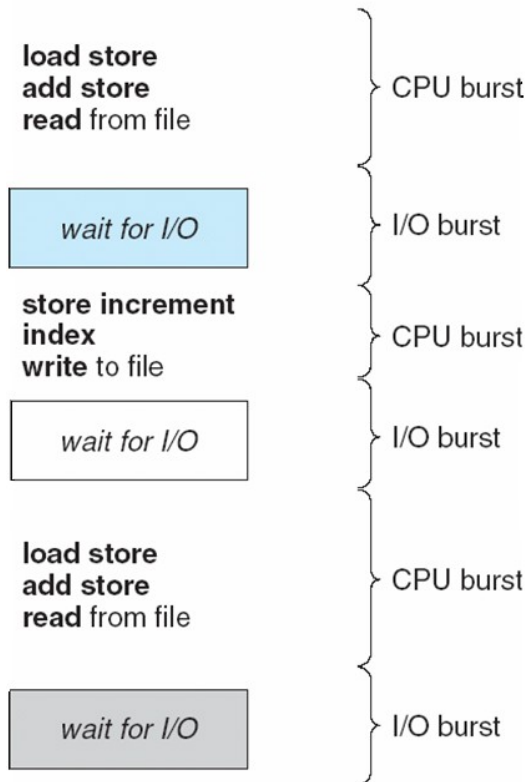
Processus (Concepts avancés)

- Ordonnancement des processus (comparaison entre politiques)



Processus (Concepts avancés)

- Ordonnancement des processus (Estimation du temps d'exécution)
 - Estimation du temps d'exécution basée sur la connaissance du fonctionnement de l'exécution des programmes (Exécution CPU et I/O)



Processus (Concepts avancés)

- Ordonnancement des processus (Estimation du temps d'exécution)

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

t_n : temps d'exécution réel d'un processus au temps n

τ_n : temps d'exécution estimé (temps n)

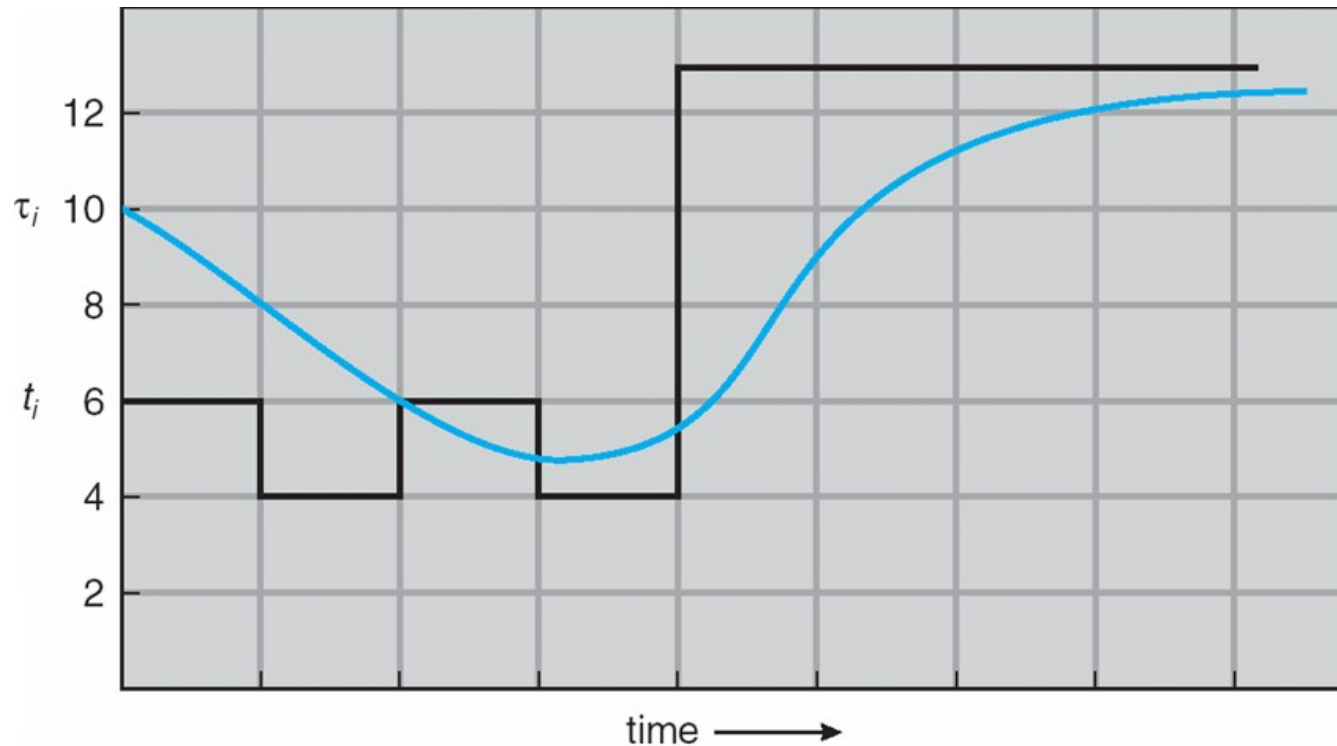
$\alpha : 0 \leq \alpha \leq 1$

En développant cette formule:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

Processus (Concepts avancés)

- Ordonnancement des processus (Estimation du temps d'exécution)

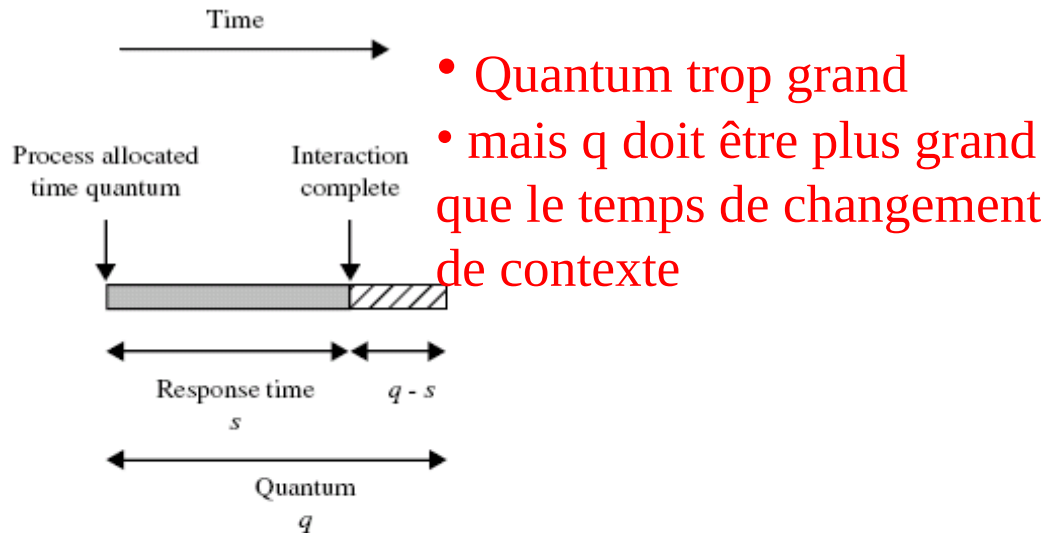


$$\alpha = 0.5$$

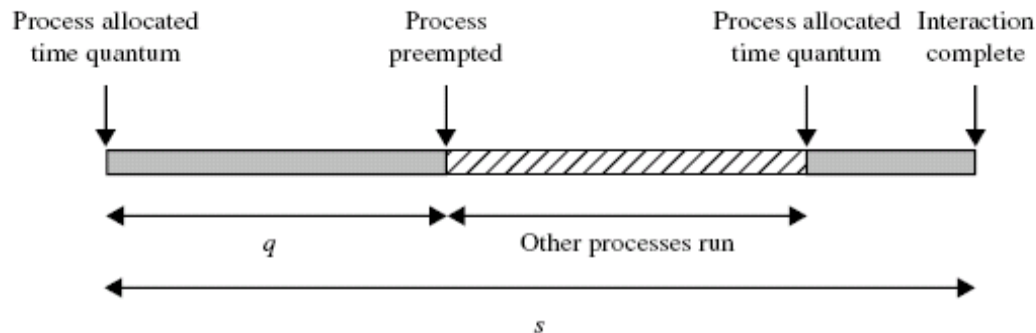
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Processus (Concepts avancés)

- Ordonnancement des processus (politique RR et quantum de temps)

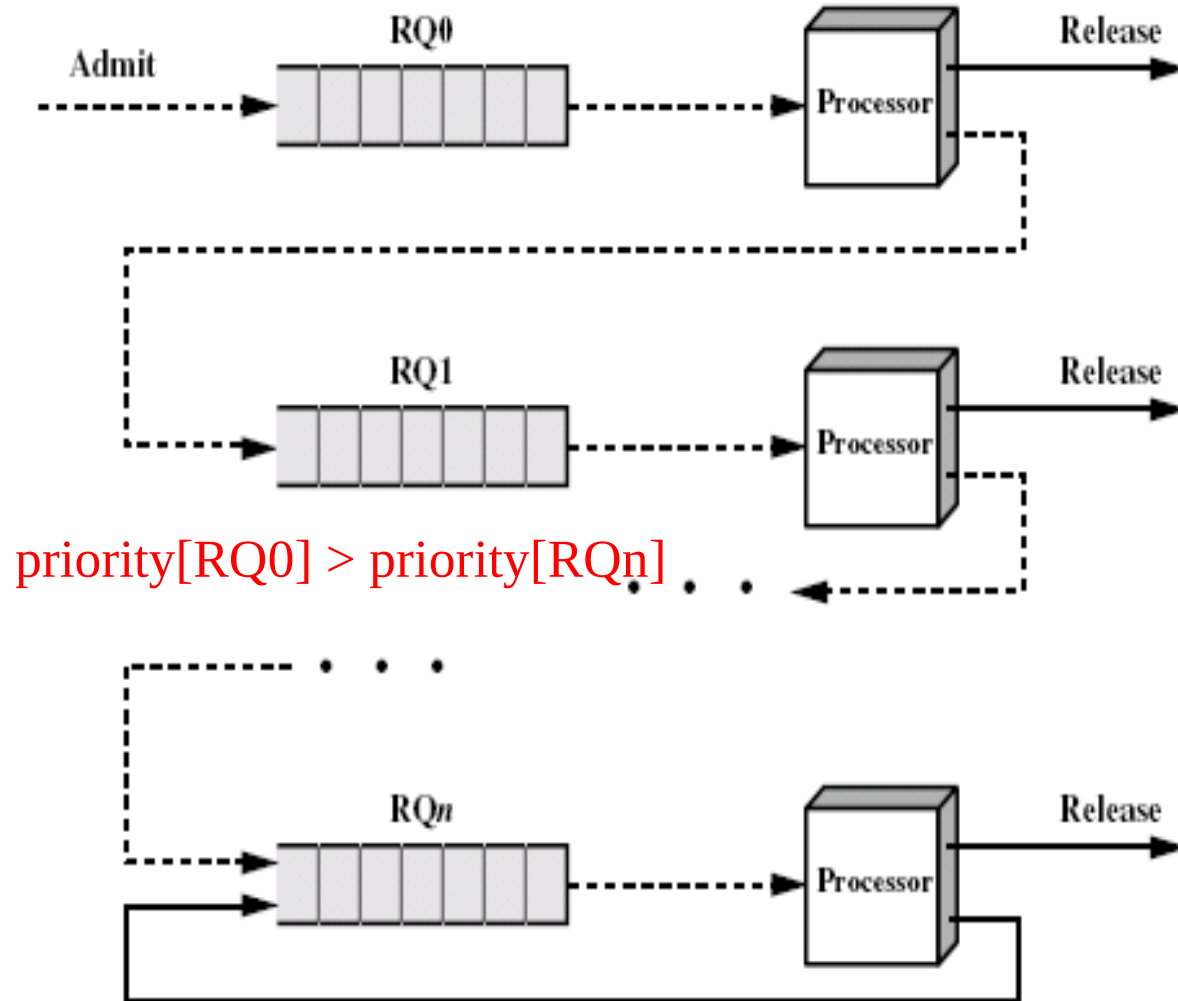


(a) Time quantum greater than typical interaction



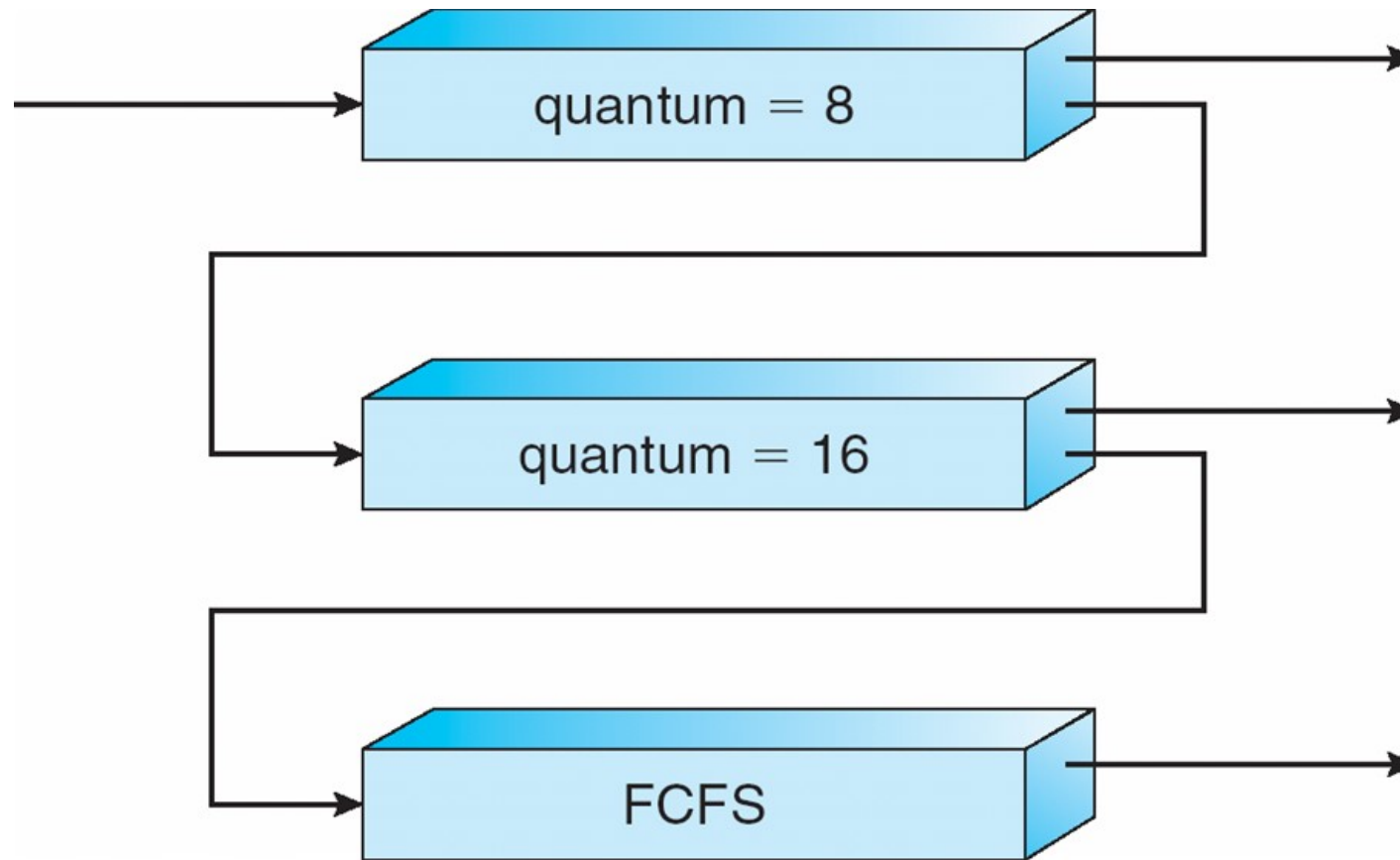
Processus (Concepts avancés)

- Ordonnancement des processus (priorités et feedback)



Processus (Concepts avancés)

- Ordonnancement des processus (priorités et feedback)



Processus (Concepts avancés)

- Ordonnancement des processus (Modèle UNIX)

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

$CPU_j(i-1)$ = Measure of processor utilization by process j through interval i

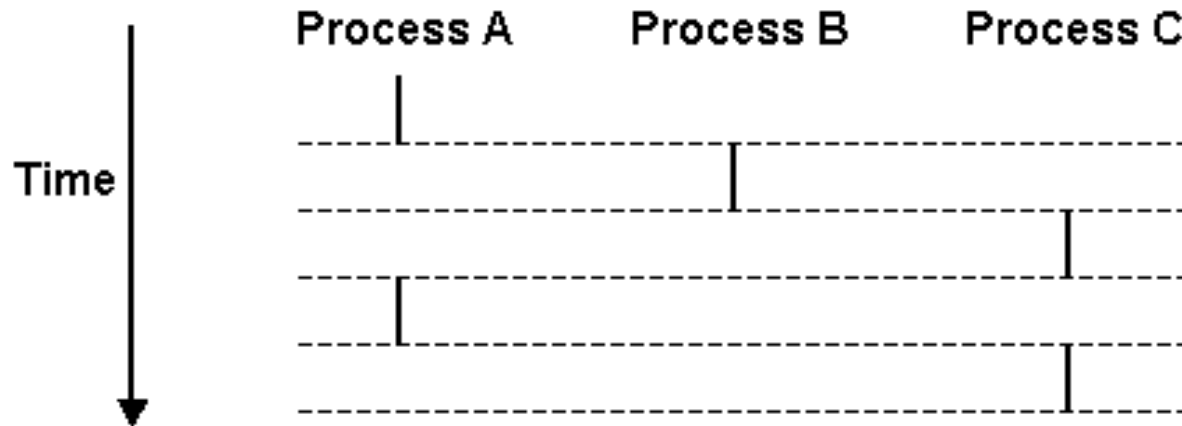
$P_j(i)$ = Priority of process j at beginning of interval i : lower values equal higher priorities

$Base_j$ = Base priority of process j

$nice_j$ = user-controllable adjustment factor

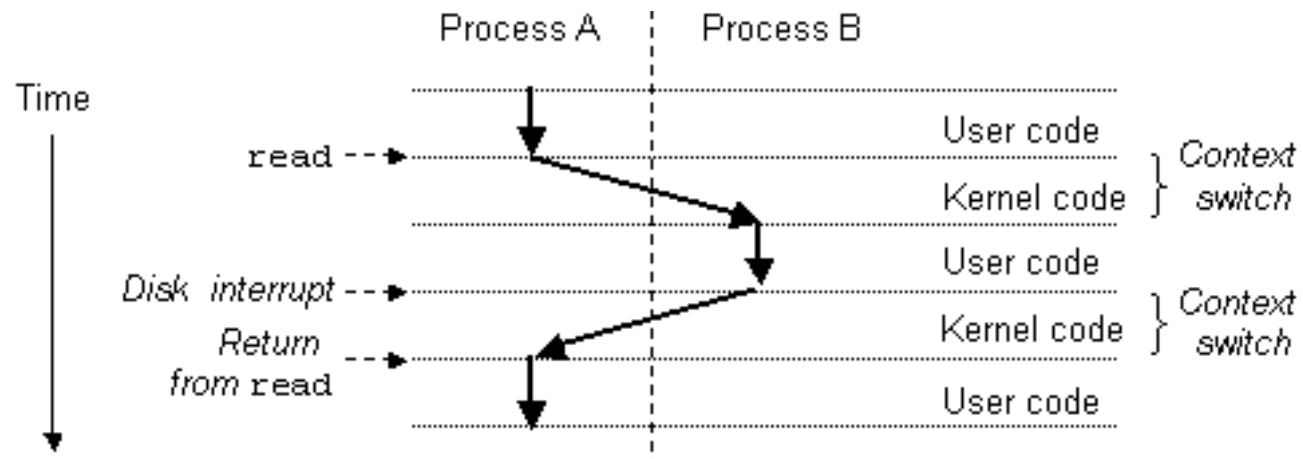
Processus (Concepts avancés)

- Ordonnancement des processus
 - Le OS implémente le multi-tâche par le changement de contexte (context switch)
 - Le noyau maintient le contexte (registres internes, compteur de programme, pile de l'utilisateur, registres d'état, pile du noyau, structures de données du noyau) de chaque processus ce qui permet le redémarrage des processus en attente du CPU



Processus (Concepts avancés)

- Ordonnancement des processus
 - L'ordonnanceur (séquenceur, scheduler) est la fonction du noyau qui décide quel processus doit être exécuté par le processeur
 - Le scheduler effectue les changements de contexte
 - Sauve le contexte du processus courant
 - Restaure le contexte d'un processus en attente du CPU
 - Passe le contrôle au processus sélectionné
 - Par exemple:
 - Si une fonction **read()** requiert l'accès à un disque
 - Un changement de contexte surviendra pour permettre à un autre processus de s'exécuter jusqu'au retour des données du disque



Processus (Concepts avancés)

- Ordonnancement des processus
 - L'ordonnanceur parcourt la liste des processus prêts et utilise plusieurs critères pour choisir le processus à exécuter
 - L'ordonnanceur de LINUX utilise trois politiques d'ordonnancement différentes: une pour les **processus normaux**, et deux pour les **processus temps-réel**
 - Chaque processus possède un type, une priorité fixe et une priorité variable
 - Les types de processus possibles:
 - **SCHED_FIFO**: processus temps-réel non préemptible
 - **SCHED_RR**: processus temps-réel préemptible
 - **SCHED_OTHER**: processus classique

Processus (Concepts avancés)

- Ordonnancement des processus
 - La politique d'ordonnancement est fonction du type de chaque processus contenus dans la liste des processus prêts à s'exécuter
 - SI un processus de type **SCHED_FIFO** devient prêt (TASK_READY), SI ce processus possède la plus haute priorité il sera exécuté immédiatement
 - Ce processus n'est pas normalement préemptible, il possède alors le processeur, il ne peut être interrompu que dans trois cas:
 - Un autre processus de type SCHED_FIFO avec une priorité supérieure passe à l'état prêt, celui-ci est alors exécuté
 - Le processus se suspend dans l'attente d'un événement (ex: fin d'une I/O)
 - Le processus abandonne volontairement le processeur par un appel à la primitive ***sched_yield***, le processus passe à l'état prêt et d'autres processus peuvent alors être exécutés
 - SI un processus de type **SCHED_RR** devient prêt, il sera traité de façon similaire à un processus de type **SCHED_FIFO**, sauf qu'un quantum de temps lui est attribué, lorsque ce quantum est expiré, un processus de type **SCHED_FIFO** ou **SCHED_RR** de priorité supérieure ou égale à celle du processus courant peut être sélectionné et exécuté

Processus (Concepts avancés)

- Ordonnancement des processus
 - La politique d'ordonnancement est fonction du type de chaque processus contenus dans la liste des processus prêts à s'exécuter
 - Les processus de type **SCHED_OTHER** sont exécutés lorsqu'aucun processus temps-réel à l'état prêt n'existe. Le processus à exécuter est choisi en fonction de sa priorité dynamique. La priorité dynamique est basée sur le niveau spécifié par l'utilisateur (Priorité statique: appel système **nice()** et **setpriority()**) ainsi que sur une variation calculée par le système. Un processus qui s'exécute pendant plusieurs cycles machine voit sa priorité diminuer et peut ainsi devenir moins prioritaire que les processus qui ne s'exécutent pas

Processus (Concepts avancés)

- Ordonnancement des processus (Priorité VS Time Quantum:

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100		other tasks	
•			
•			
•			
140	lowest		10 ms

Processus (Concepts avancés)

- Ordonnement des processus(Priorité: XP)

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1



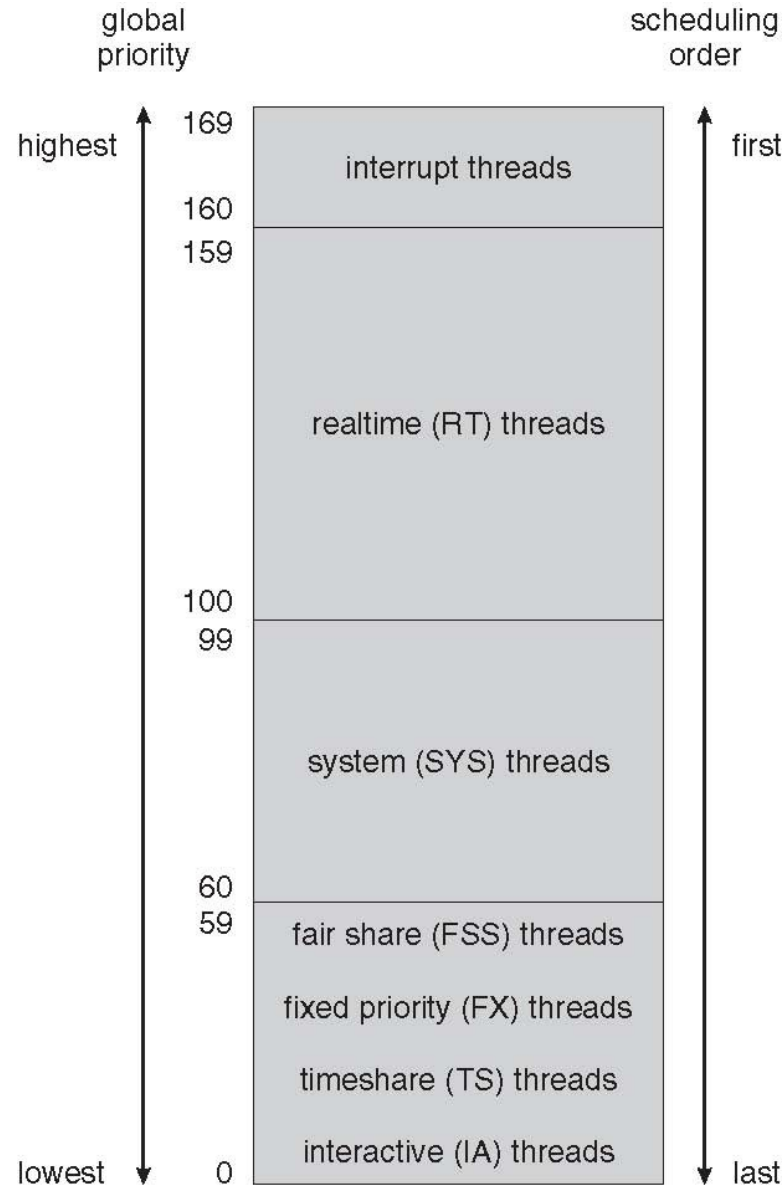
Classes de
priorité



Intervalle des valeurs de priorité

Processus (Concepts avancés)

- Ordonnancement des processus (Priorité: Solaris)



Processus (Concepts avancés)

- Modification de l'ordonnancement
 - Appel système permettant de manipuler les priorités des processus
 - **#include <unistd.h>**
 - **int nice(int inc);**
 - **#include <sys/resource.h>**
 - **int setpriority(int which, int who, int prio);**
 - **int getpriority(int which, int who);**
 - L'appel système **nice()** permet de modifier la priorité statique du processus courant. Le paramètre **inc** est ajouté à la priorité courante. Seul un processus privilégié peut spécifier un **inc** négatif afin d'augmenter sa priorité. Si **nice()** retourne -1 et que la variable système **errno == EPERM** cela indique que le processus appelant ne possède pas les privilèges suffisant pour augmenter sa priorité
 - L'appel système **setpriority()** permet de modifier la priorité d'un processus. Le paramètre **which** dans le cas de processus individuel prend la valeur **PRIO_PROCESS**. Le paramètre **who** indique un numéro de processus. Le paramètre **prio** indique la valeur de la priorité

Processus (Concepts avancés)

- Modification de l'ordonnancement
 - Appel système permettant de manipuler les priorités des processus
 - L'appel système ***getpriority()*** permet d'obtenir la priorité d'un processus

exemple_nice.c (modification de la priorité statique)

```
#define _GNU_SOURCE
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

volatile long  compteur = 0;
static int     gentillesse;

void
gestionnaire (int numero)
{
    if (compteur != 0) {
        fprintf (stdout, "Fils %u (nice %2d) Compteur = %9ld\n",
                 getpid(), gentillesse, compteur);
        exit (0);
    }
}

#define NB_FILS 5
```

\$./exemple_nice

Fils 1849 (nice 10) Compteur = 91829986

Fils 1850 (nice 15) Compteur = 42221693

Fils 1851 (nice 20) Compteur = 30313573

Fils 1847 (nice 0) Compteur = 183198223

Fils 1848 (nice 5) Compteur = 133284576

\$

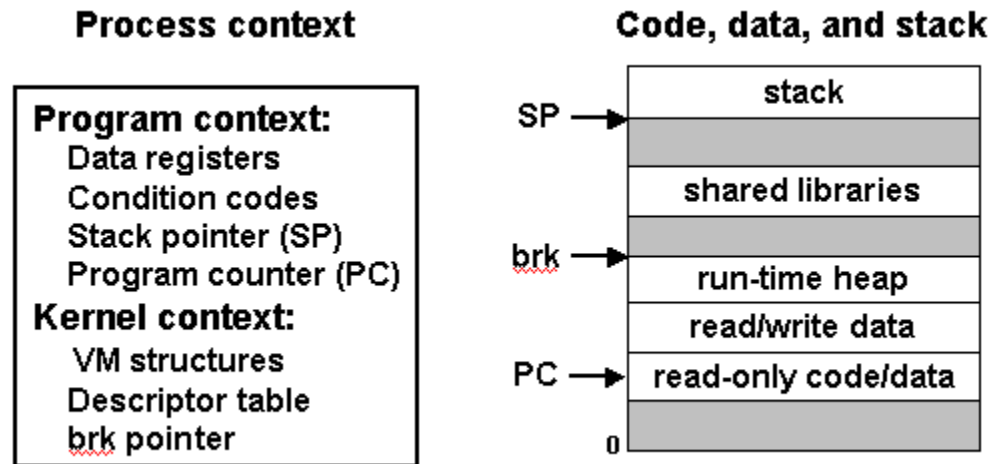
```
int
main (void)
{
    pid_t  pid;
    int     fils;

    /* Création d'un nouveau groupe de processus */
    setpgid (0, 0);

    signal (SIGUSR1, gestionnaire);
    for (fils = 0; fils < NB_FILS; fils++) {
        if ((pid = fork()) < 0) {
            perror ("fork");
            exit (1);
        }
        if (pid != 0)
            continue;
        gentillesse = fils * (20 / (NB_FILS - 1));
        if (nice (gentillesse) < 0) {
            perror ("nice");
            exit (1);
        }
        /* attente signal de démarrage */
        pause ();
        /* comptage */
        while (1)
            compteur++;
    }
    /* processus père */
    signal (SIGUSR1, SIG_IGN);
    sleep (1);
    kill (- getpgid (0), SIGUSR1);
    sleep (5);
    kill (- getpgid (0), SIGUSR1);
    while (wait (NULL) > 0)
        ;
    exit (0);
}
```

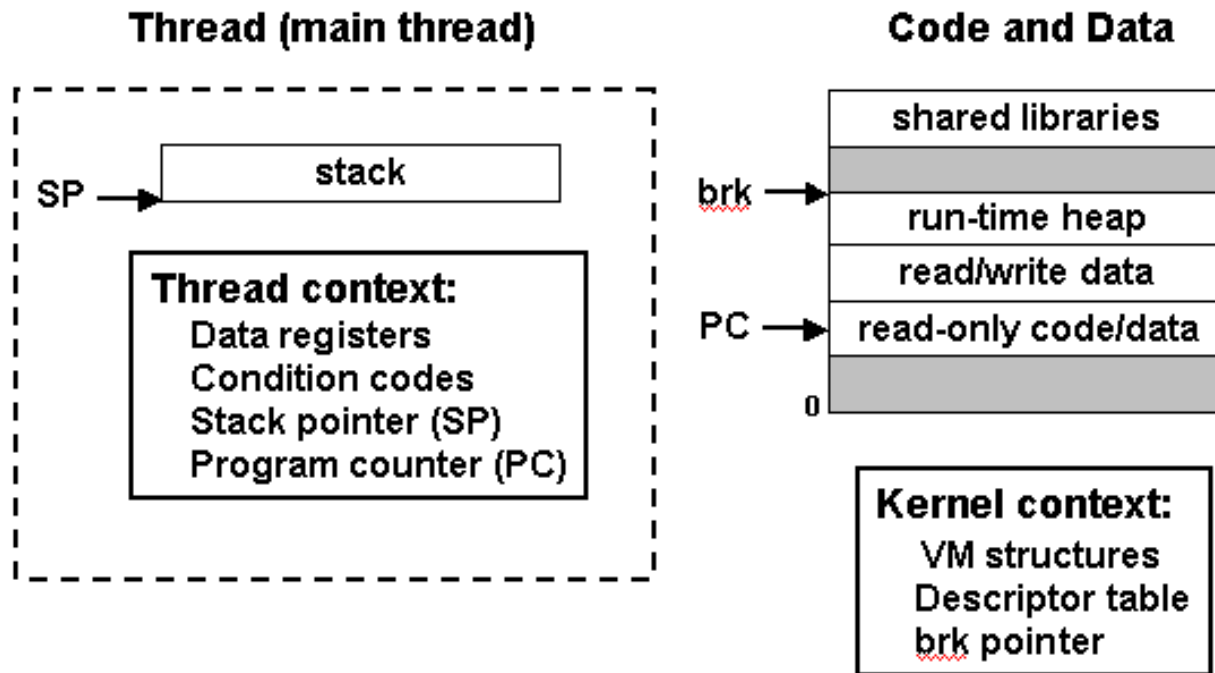
Threads (Concepts de base)

- Un **thread** est une unité d'exécution associée à un processus, avec son propre identificateur, sa pile, son compteur de programme et registres
- Comparons une vue traditionnelle d'un processus avec une vue alternative



Vue traditionnelle d'un processus

Threads (Concepts de base)



Vue alternative d'un processus (1 thread)

Threads (Concepts de base)

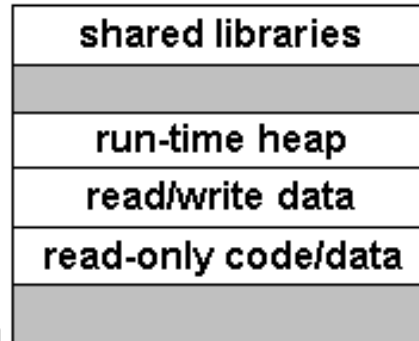
Thread 1 (main thread)

stack 1

Thread 1 context:

Data registers
Condition codes
SP1
PC1

Shared code and data



Kernel context:

VM structures
Descriptor table
brk pointer

Thread 2 (peer thread)

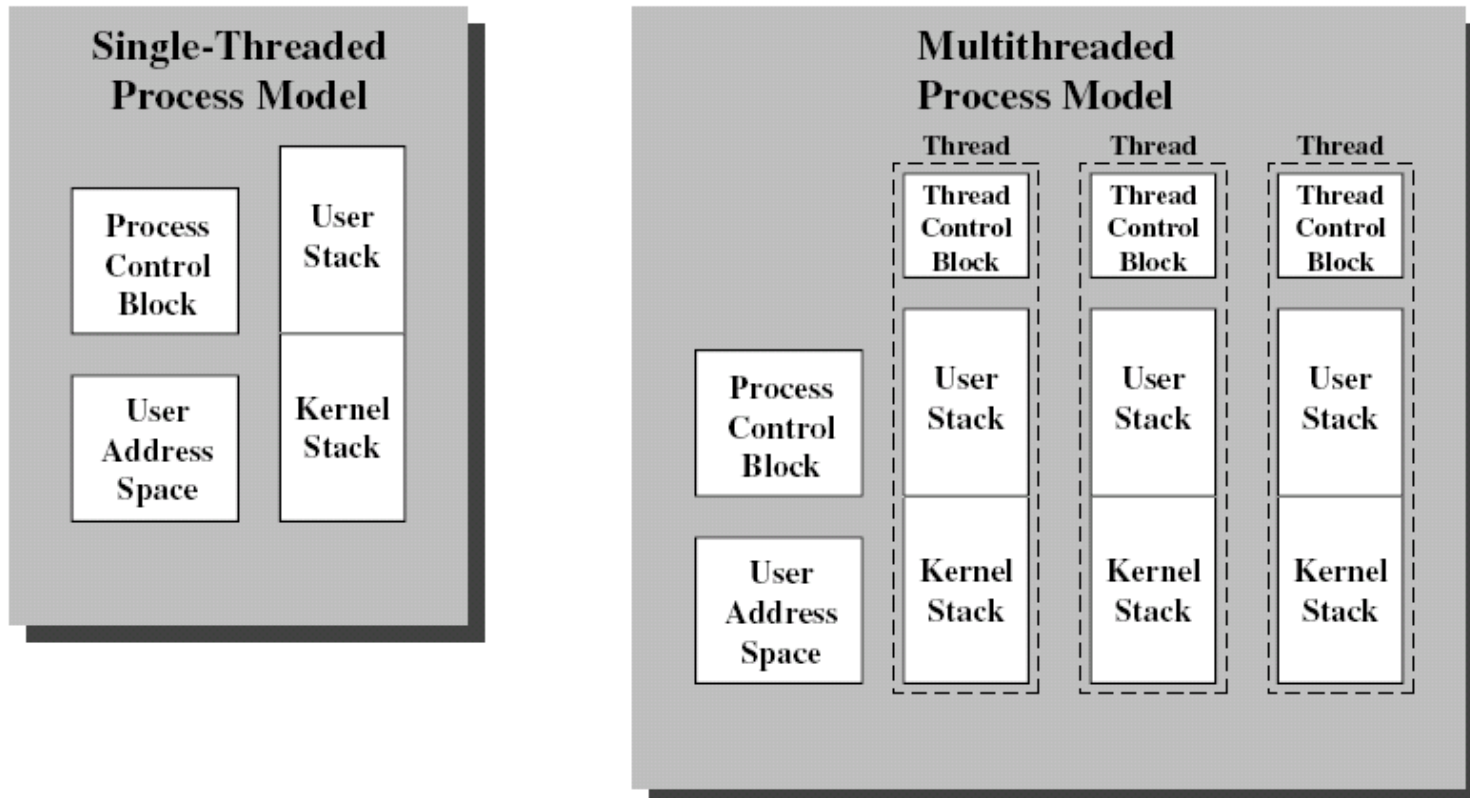
stack 2

Thread 2 context:

Data registers
Condition codes
SP2
PC2

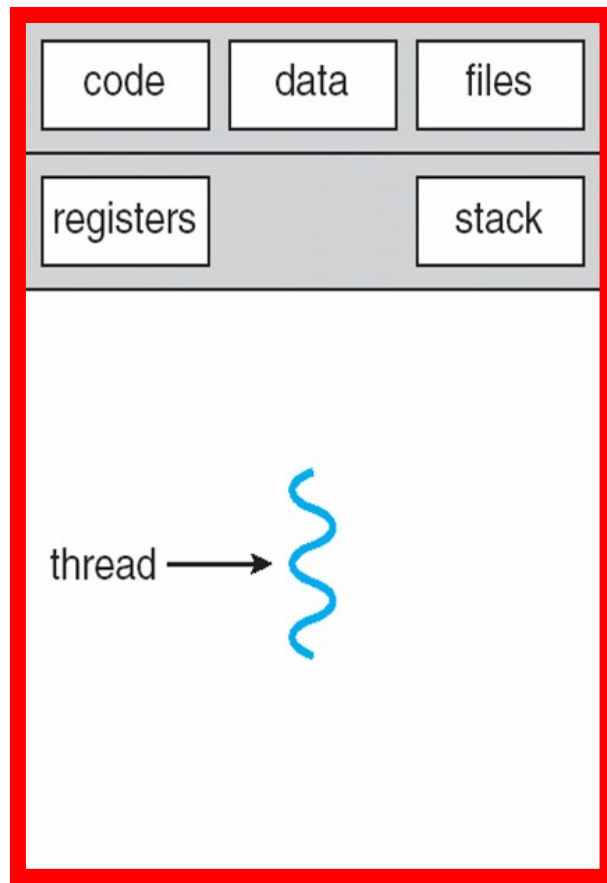
Vue alternative d'un processus (2 threads)

Threads (Concepts de base)

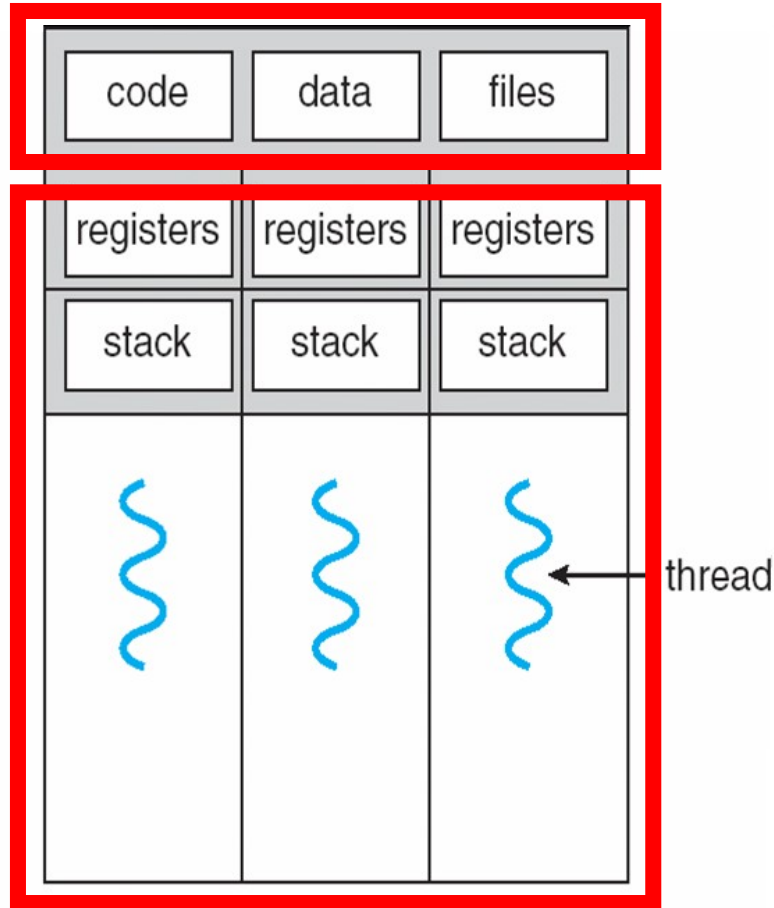


Processus à thread unique et multiples

Threads (Concepts de base)



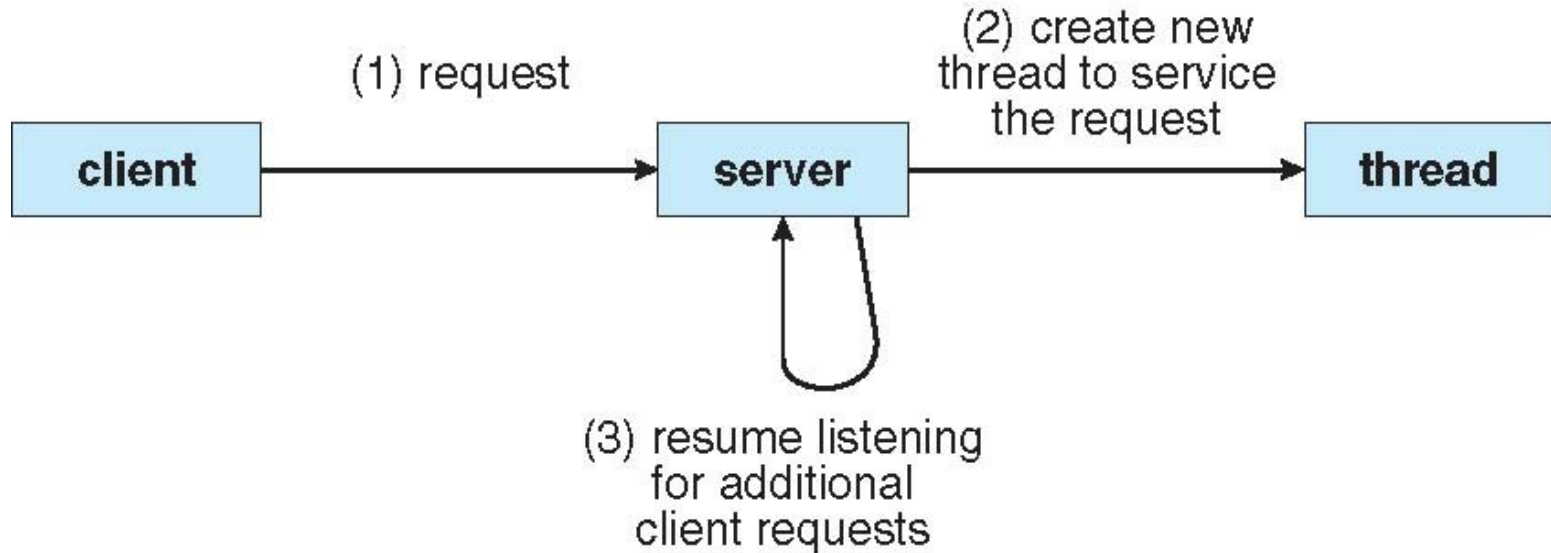
single-threaded process



multithreaded process

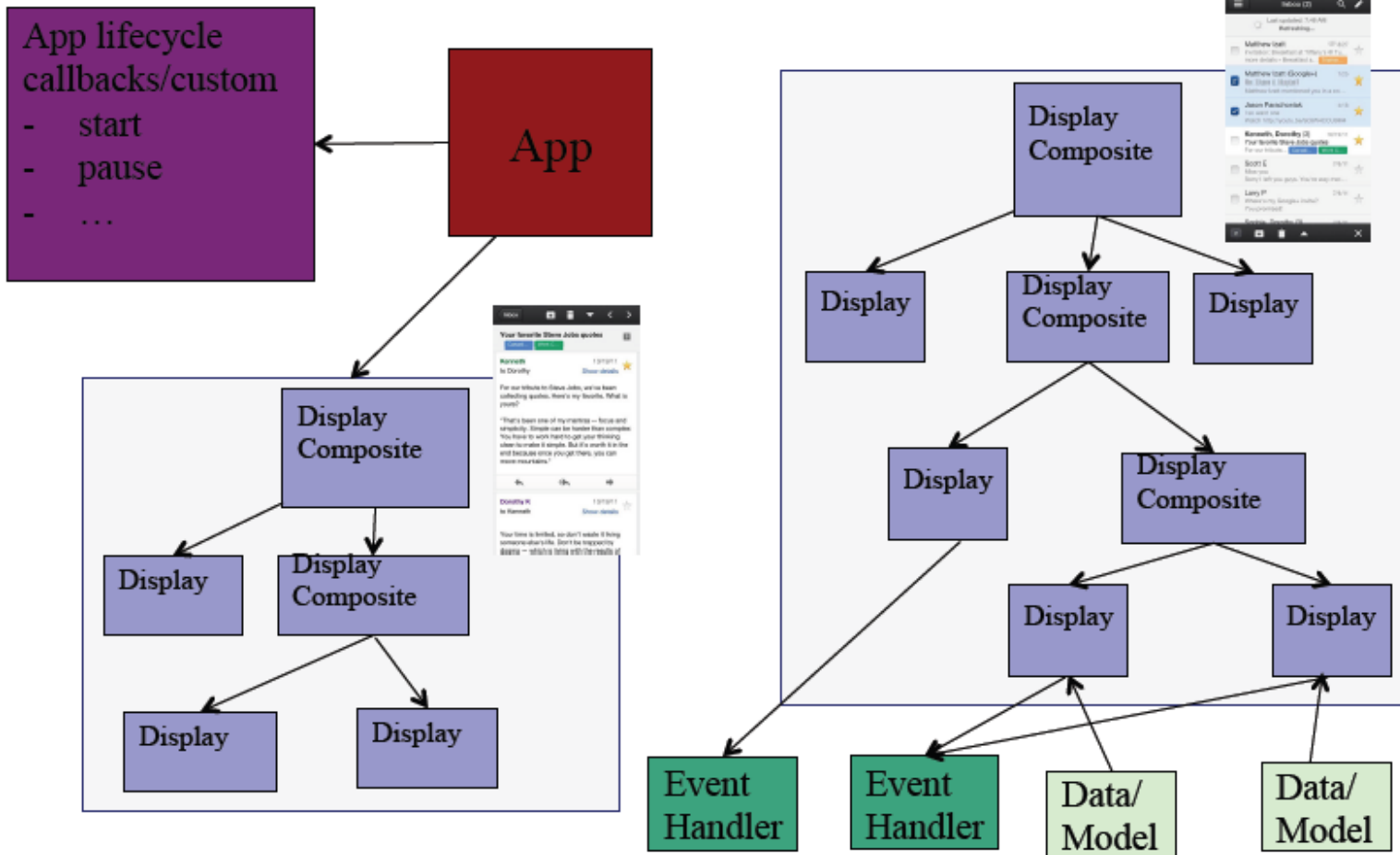
Processus à thread unique et multiples

Threads (Concepts de base)

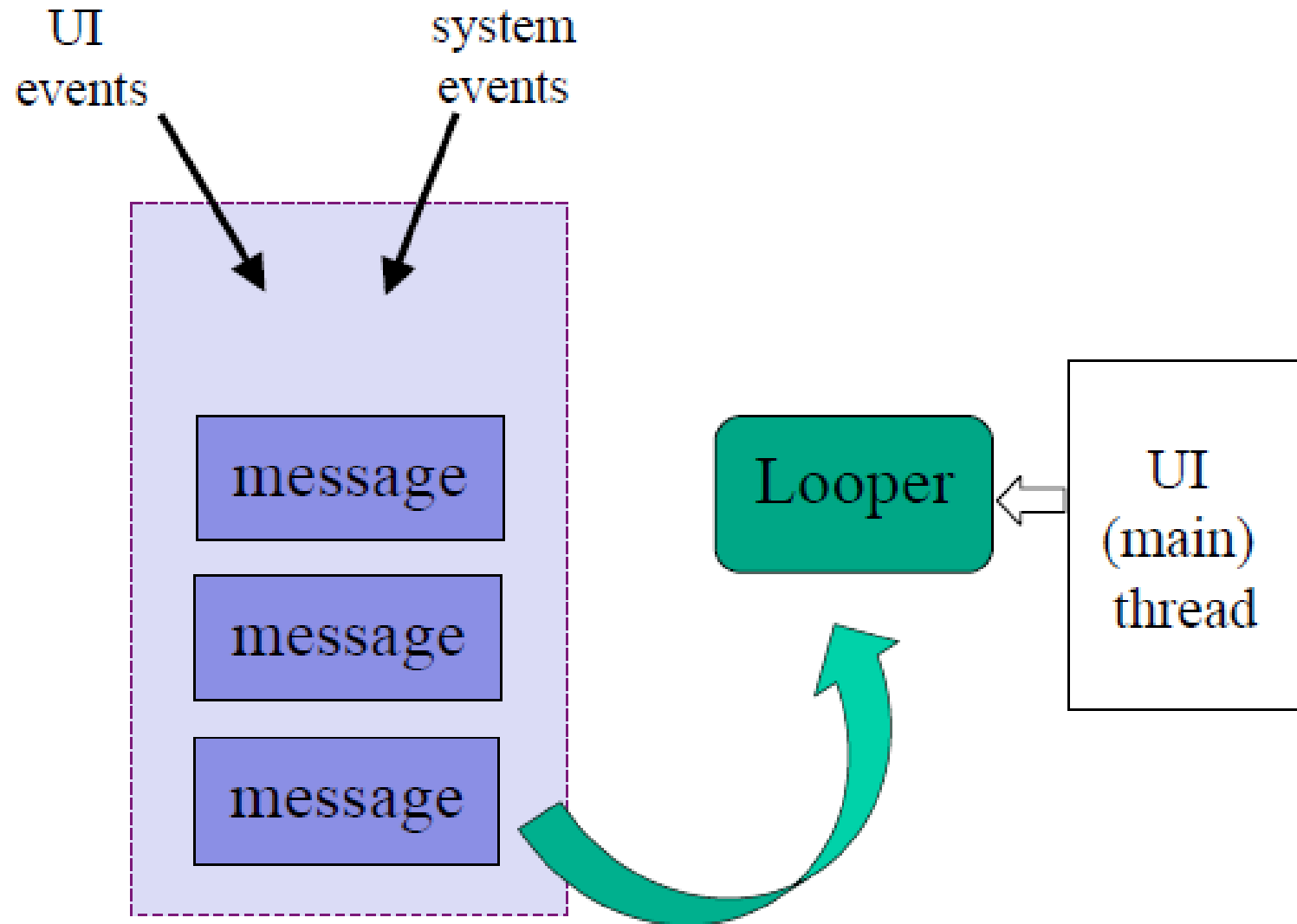


Processus à thread multiples comme base des architectures Client/Serveur

Threads (Concepts de base: Android)

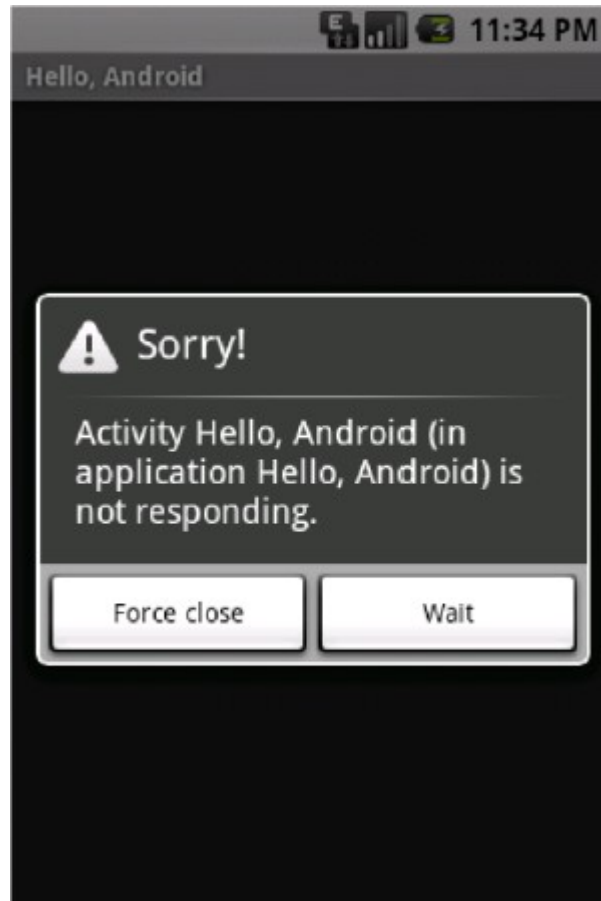


Threads (Concepts de base: Android)



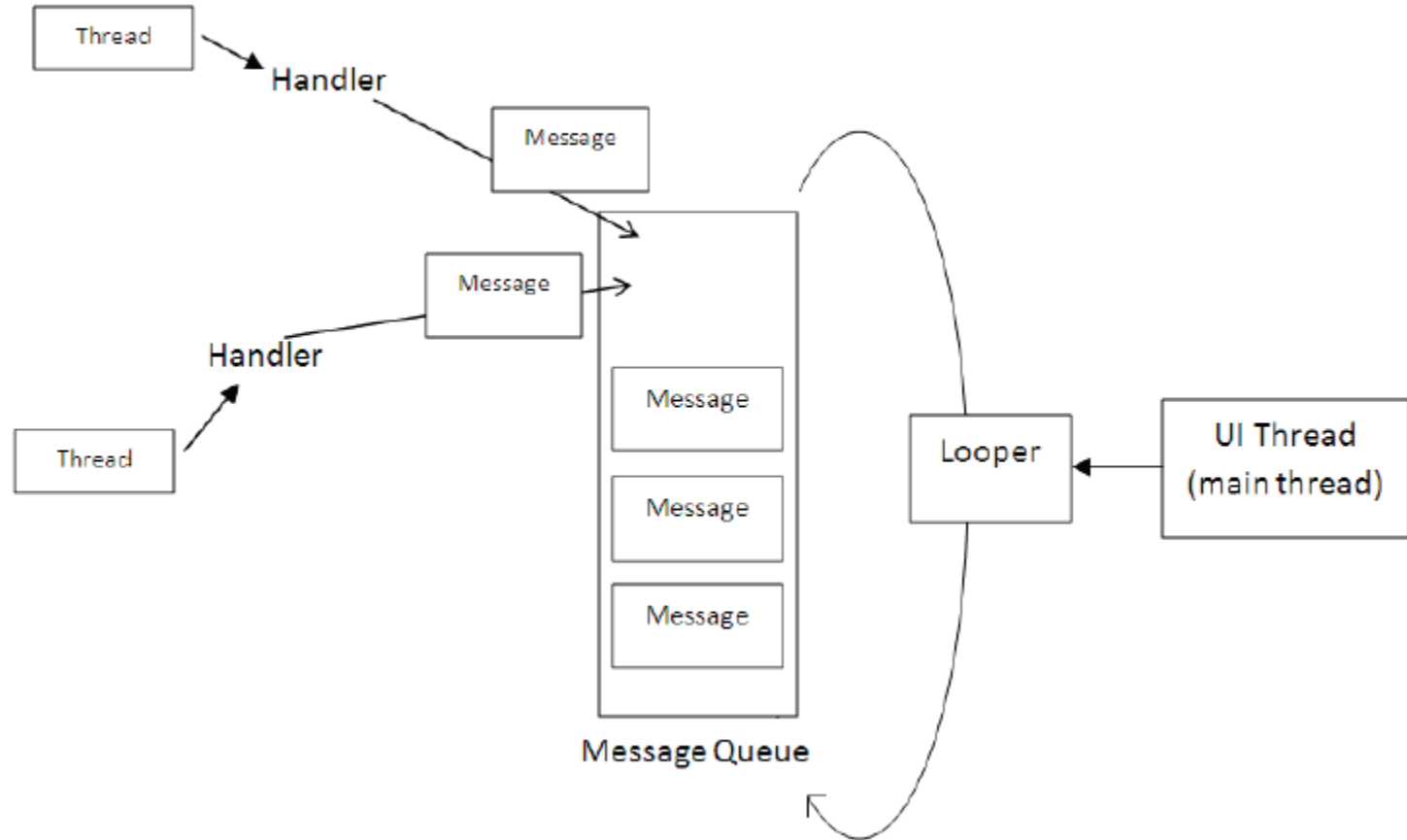
Threads (Concepts de base: Android)

- Lorsque le thread principal ne répond pas rapidement (< 5 sec) à un message
 - Traitements lourds: reconnaissance vocale, m-a-j de cartes, accès réseaux



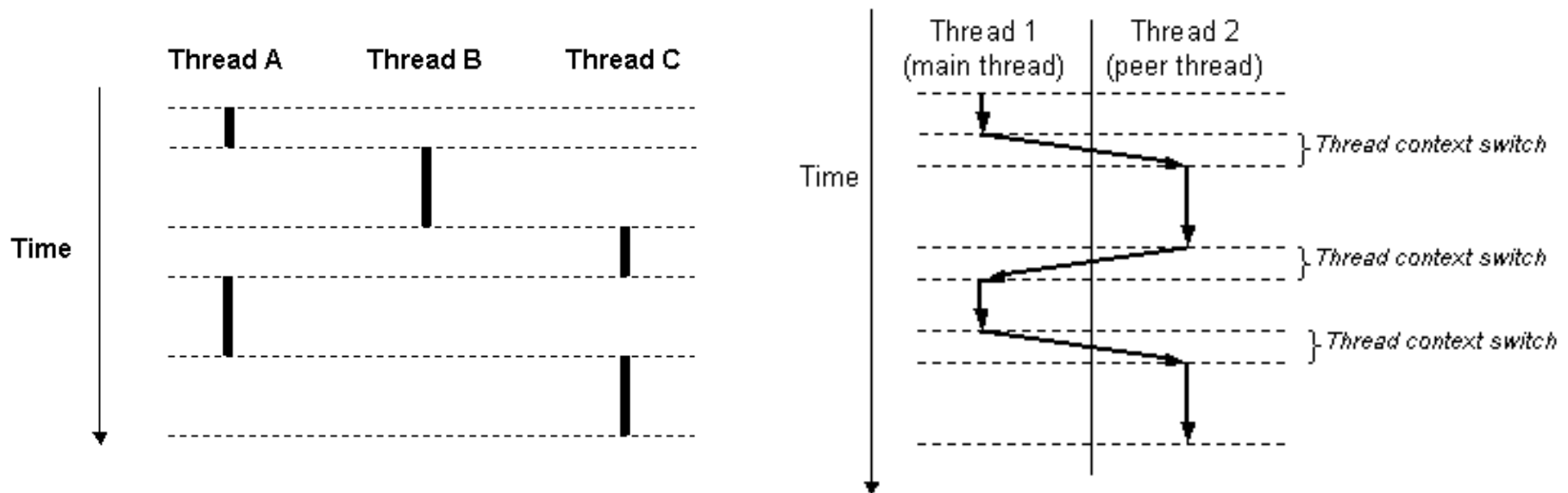
Threads (Concepts de base: Android)

- Les threads peuvent diminuer la charge de traitement du thread principal



Threads (Concepts de base)

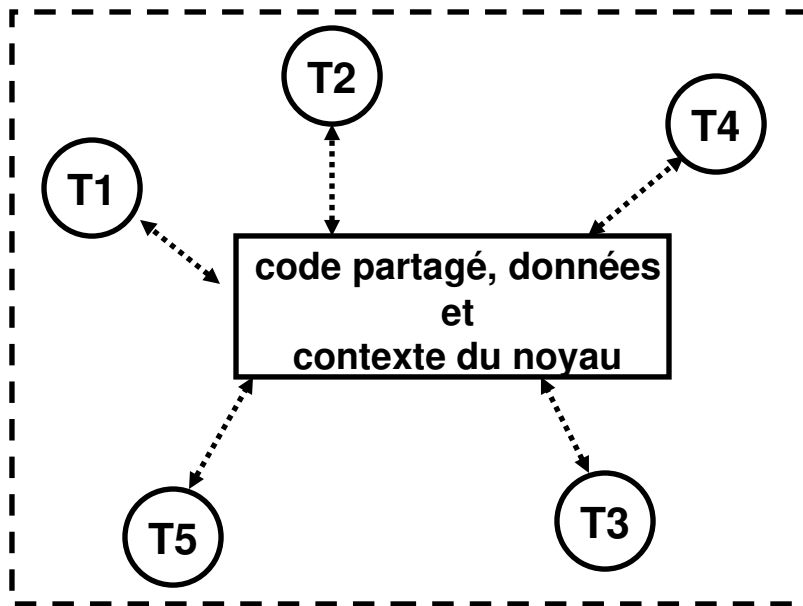
- L'exécution concurrente des threads est similaire à celle des processus
- Par contre, le changement de contexte d'un thread est plus rapide puisque le contexte est beaucoup plus petit
- Chaque thread, associé à un processus est indépendant, donc un thread peut en tuer un autre, peut attendre qu'un autre thread se termine, peut aussi faire la LECTURE/ÉCRITURE des mêmes données partagées



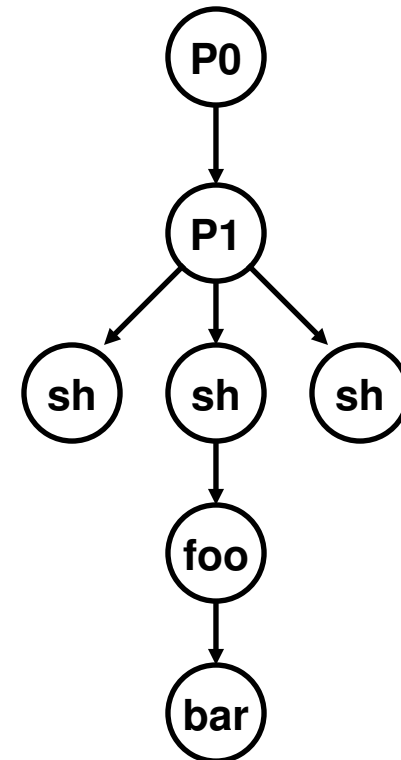
Threads (Concepts de base)

- Les threads associés à un processus forment un ensemble indépendant de pairs.
 - Les processus eux forment une hiérarchie sous forme d'arborescence

Threads associés avec le processus foo



Hiérarchie de processus



Threads (Appel système de base)

- Création d'un thread
 - L'appel système ***pthread_create()*** permet de créer un nouveau thread et exécuter une fonction.
 - ***pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);***
L'argument ***tid*** représente l'identificateur du thread créé, ***arg*** est un pointeur sur les arguments en entrée fourni à la fonction ***f***, ***f*** étant l'unité d'exécution associée au thread, ***attr*** est généralement NULL. Cette appel système retourne 0 si aucune erreur n'est survenue et une valeur différente de 0 en cas contraire
- Attente de la terminaison d'un thread
 - L'appel système ***pthread_join()*** permet à un thread (souvent main) d'attendre la terminaison d'un autre thread.
 - ***pthread_join(pthread_t *tid, void **thread_return);*** l'argument ***tid*** est l'identificateur du thread à attendre, ***thread_return*** est un pointeur auquel est assigné le pointeur retourné par le thread, est souvent NULL

Threads (Appel système de base)

- Création d'un thread (Site csapp, hello.c)

```
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

Threads (Appel système de base)

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
    pthread_t tid;

    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

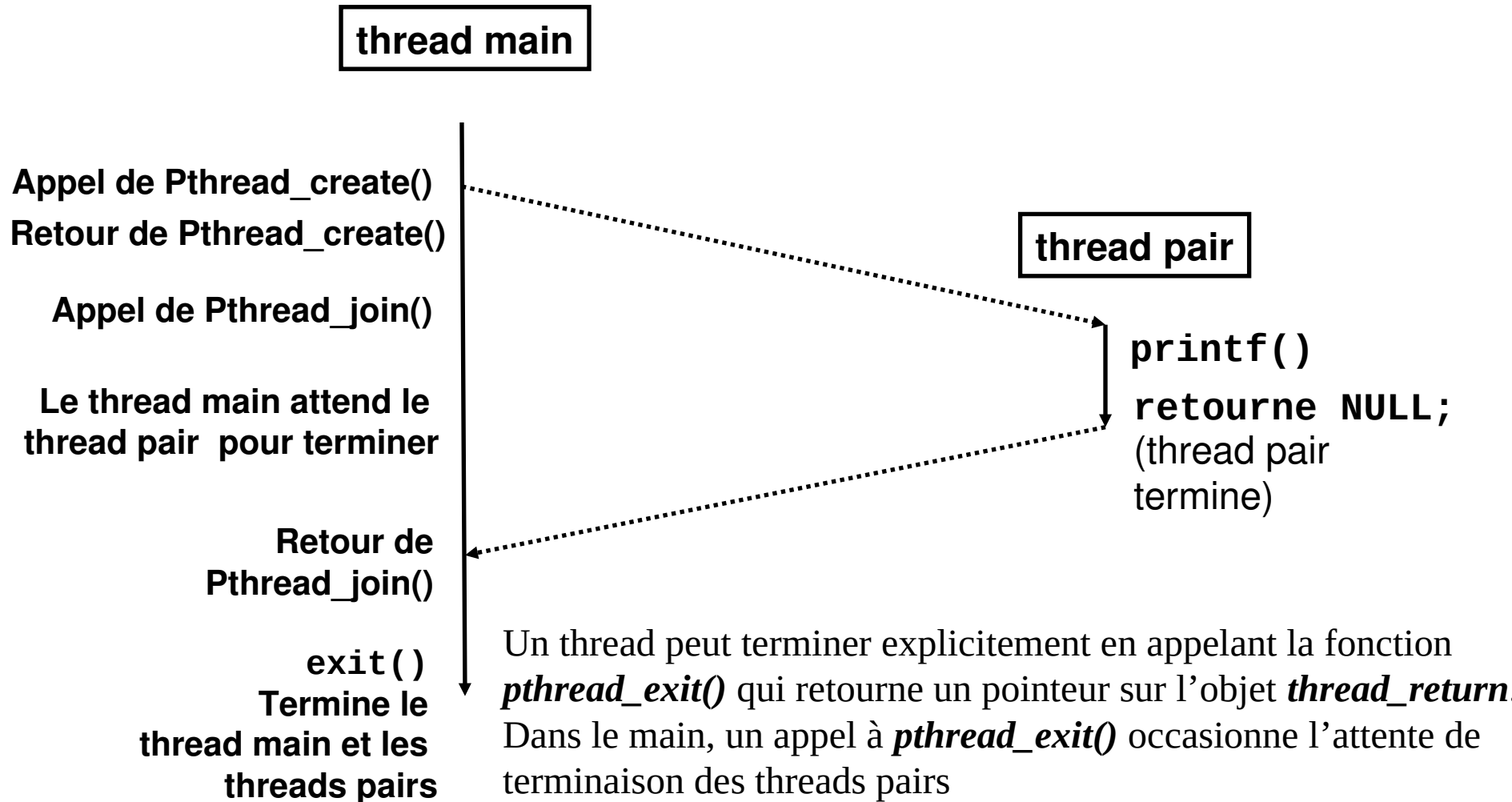
/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
```

*Attributs du thread
(Généralement NULL)*

*Arguments du thread
(void *p)*

*Valeur retournée
(void **p)*

Exécution du “hello, world”



Threads (Appels système de base)

- Création d'un thread (Site csapp, hellobug.c, version bugger)
 - Pourquoi lors de l'exécution nous ne voyons pas Hello, world! ??

```
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;

    Pthread_create(&tid, NULL, thread, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp)
{
    Sleep(1);
    printf("Hello, world!\n");
    return NULL;
}
```

Threads (Appels système de base)

- Création d'un thread
 - Le thread principal affiche des 'o' et le thread créé affiche des 'x'

Créer un thread thread-create.c

```
#include <pthread.h>
#include <stdio.h>
/* Affiche des x sur stderr. Paramètre inutilisé. Ne finit jamais. */
void* print_xs (void* unused)
{
    while (1)
        fputc ('x', stderr);
    return NULL;
}
/* Le programme principal. */
int main ()
{
    pthread_t thread_id;
    /* Crée un nouveau thread. Le nouveau thread exécutera la fonction
       print_xs. */
    pthread_create (&thread_id, NULL, &print_xs, NULL);
    /* Affiche des o en continue sur stderr. */
    while (1)
        fputc ('o', stderr);
    return 0;
}
```

```
% cc -o thread-create thread-create.c -lpthread
```

Ne pas oublier **-lpthread**

Threads (Appels système de base)

- Création d'un thread
 - Le thread 2 affiche un certain nombre de 'o' et le thread 1 affiche un certain nombre de 'x'

Créer Deux Threads thread-create2.c

```
#include <pthread.h>
#include <stdio.h>
/* Paramètres de la fonction print. */
struct char_print_params
{
    /* Caractère à afficher. */
    char character;
    /* Nombre de fois où il doit être affiché. */
    int count;
};
/* Affiche un certain nombre de caractères sur stderr, selon le contenu de
PARAMETERS, qui est un pointeur vers une struct char_print_params. */
void* char_print (void* parameters)
{
    /* Effectue un transtypage du pointeur void vers le bon type. */
    struct char_print_params* p = (struct char_print_params*) parameters;
    int i;
    for (i = 0; i < p->count; ++i)
        fputc (p->character, stderr);
    return NULL;
}
/* Programme principal. */
int main ()
{
    pthread_t thread1_id;
    pthread_t thread2_id;
    struct char_print_params thread1_args;
    struct char_print_params thread2_args;
    /* Crée un nouveau thread affichant 30 000 x. */
    thread1_args.character = 'x';
    thread1_args.count = 30000;
    pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
    /* Crée un nouveau thread affichant 20 000 'o'. */
    thread2_args.character = 'o';
    thread2_args.count = 20000;
    pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
    return 0;
}
```

DANGER:


- structures libérées quand le thread main se termine
- si possible passer des structures dynamiques plutôt que statiques
- sinon, le passage de structures statiques d'adresses différentes

Threads (Appel système de base)

- Création d'un thread
 - Le thread 2 affiche un certain nombre de 'o' et le thread 1 affiche un certain nombre de 'x'
 - Le thread main attend que les deux threads créés se terminent

Fonction main de thread-create2.c corrigée thread-create2a.c

```
int main ()
{
    pthread_t thread1_id;
    pthread_t thread2_id;
    struct char_print_parms thread1_args;
    struct char_print_parms thread2_args;
    /* Crée un nouveau thread affichant 30 000 x. */
    thread1_args.character = 'x';
    thread1_args.count = 30000;
    pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
    /* Crée un nouveau thread affichant 20 000 o. */
    thread2_args.character = 'o';
    thread2_args.count = 20000;
    pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
    /* S'assure que le premier thread est terminé. */
    pthread_join (thread1_id, NULL);
    /* S'assure que le second thread est terminé. */
    pthread_join (thread2_id, NULL);
    /* Nous pouvons maintenant quitter en toute sécurité. */
    return 0;
}
```



Threads (Appels système de base)

- Création d'un thread et politique de séquençement

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *runner(void *param)
int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread doing just nothing\n");
    pthread_exit(0);
}
```


Threads (Appels système de base)

- Appels système de gestion des threads (*pthread_create()*)

`pthread_create` - create a new thread

SYNOPSIS [top](#)

```
#include <pthread.h>
```

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

Compile and link with `-pthread`.

DESCRIPTION [top](#)

The `pthread_create()` function starts a new thread in the calling process. The new thread starts execution by invoking `start_routine()`; `arg` is passed as the sole argument of `start_routine()`.

The new thread terminates in one of the following ways:

- * It calls `pthread_exit(3)`, specifying an exit status value that is available to another thread in the same process that calls `pthread_join(3)`.
- * It returns from `start_routine()`. This is equivalent to calling `pthread_exit(3)` with the value supplied in the `return` statement.
- * It is canceled (see `pthread_cancel(3)`).
- * Any of the threads in the process calls `exit(3)`, or the main thread performs a return from `main()`. This causes the termination of all threads in the process.

Threads (Appels système de base)

- Appels système de gestion des threads (*pthread_create()*)

The `attr` argument points to a `pthread_attr_t` structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using `pthread_attr_init(3)` and related functions. If `attr` is NULL, then the thread is created with default attributes.

Before returning, a successful call to `pthread_create()` stores the ID of the new thread in the buffer pointed to by `thread`; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

The new thread inherits a copy of the creating thread's signal mask (`pthread_sigmask(3)`). The set of pending signals for the new thread is empty (`sigpending(2)`). The new thread does not inherit the creating thread's alternate signal stack (`sigaltstack(2)`).

The new thread inherits the calling thread's floating-point environment (`fenv(3)`).

The initial value of the new thread's CPU-time clock is 0 (see `pthread_getcpuclockid(3)`).

Threads (Appels système de base)

- Appels système de gestion des threads (*pthread_join()*)

`pthread_join` - join with a terminated thread

Synopsis

```
#include <pthread.h> int pthread_join(pthread_t thread, void **retval);
```

Compile and link with `-pthread`.

Description

The `pthread_join()` function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then `pthread_join()` returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then `pthread_join()` copies the exit status of the target thread (i.e., the value that the target thread supplied to `pthread_exit(3)`) into the location pointed to by **retval*. If the target thread was canceled, then `PTHREAD_CANCELED` is placed in **retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling `pthread_join()` is canceled, then the target thread will remain joinable (i.e., it will not be detached).

Return Value

On success, `pthread_join()` returns 0; on error, it returns an error number.

Threads (Appels système de base)

- Appels système de gestion des threads (*pthread_attr_init()*)

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

Compile and link with -pthread.
```

DESCRIPTION

[top](#)

The `pthread_attr_init()` function initializes the thread attributes object pointed to by `attr` with default attribute values. After this call, individual attributes of the object can be set using various related functions (listed under SEE ALSO), and then the object can be used in one or more `pthread_create(3)` calls that create threads.

Calling `pthread_attr_init()` on a thread attributes object that has already been initialized results in undefined behavior.

When a thread attributes object is no longer required, it should be destroyed using the `pthread_attr_destroy()` function. Destroying a thread attributes object has no effect on threads that were created using that object.

Once a thread attributes object has been destroyed, it can be reinitialized using `pthread_attr_init()`. Any other use of a destroyed thread attributes object has undefined results.

RETURN VALUE

[top](#)

On success, these functions return 0; on error, they return a nonzero error number.

Threads (Appels système de base)

- Appels système de gestion des threads (*pthread_attr_setscope()*)

```
#include <pthread.h>
```

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope);  
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);
```

Compile and link with `-pthread`.

DESCRIPTION

[top](#)

The `pthread_attr_setscope()` function sets the contention scope attribute of the thread attributes object referred to by `attr` to the value specified in `scope`. The contention scope attribute defines the set of threads against which a thread competes for resources such as the CPU. POSIX.1-2001 specifies two possible values for `scope`:

PTHREAD_SCOPE_SYSTEM

The thread competes for resources with all other threads in all processes on the system that are in the same scheduling allocation domain (a group of one or more processors).

`PTHREAD_SCOPE_SYSTEM` threads are scheduled relative to one another according to their scheduling policy and priority.

PTHREAD_SCOPE_PROCESS

The thread competes for resources with all other threads in the same process that were also created with the `PTHREAD_SCOPE_PROCESS` contention scope. `PTHREAD_SCOPE_PROCESS` threads are scheduled relative to other threads in the process according to their scheduling policy and priority.

POSIX.1-2001 leaves it unspecified how these threads contend with other threads in other process on the system or with other threads in the same process that were created with the `PTHREAD_SCOPE_SYSTEM` contention scope.

POSIX.1-2001 requires that an implementation support at least one of these contention scopes. Linux supports `PTHREAD_SCOPE_SYSTEM`, but not `PTHREAD_SCOPE_PROCESS`.

Threads (Appels système de base)

- Appels système de gestion des threads (*pthread_attr_setscope()*)

On systems that support multiple contention scopes, then, in order for the parameter setting made by `pthread_attr_setscope()` to have effect when calling `pthread_create(3)`, the caller must use `pthread_attr_setinheritsched(3)` to set the inherit-scheduler attribute of the attributes object `attr` to `PTHREAD_EXPLICIT_SCHED`.

The `pthread_attr_getscope()` function returns the contention scope attribute of the thread attributes object referred to by `attr` in the buffer pointed to by `scope`.

RETURN VALUE [top](#)

On success, these functions return 0; on error, they return a nonzero error number.

Threads (Appels système de base)

- Appels système de gestion des threads
(*pthread_attr_setschedpolicy()*)

```
#include <pthread.h>
```

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);  
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
```

Compile and link with `-pthread`.

DESCRIPTION

[top](#)

The `pthread_attr_setschedpolicy()` function sets the scheduling policy attribute of the thread attributes object referred to by `attr` to the value specified in `policy`. This attribute determines the scheduling policy of a thread created using the thread attributes object `attr`.

The supported values for `policy` are `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`, with the semantics described in `sched_setscheduler(2)`.

The `pthread_attr_getschedpolicy()` returns the scheduling policy attribute of the thread attributes object `attr` in the buffer pointed to by `policy`.

In order for the policy setting made by `pthread_attr_setschedpolicy()` to have effect when calling `pthread_create(3)`, the caller must use `pthread_attr_setinheritsched(3)` to set the inherit-scheduler attribute of the attributes object `attr` to `PTHREAD_EXPLICIT_SCHED`.

RETURN VALUE

[top](#)

On success, these functions return 0; on error, they return a nonzero error number.

Threads (Appels système de base)

- Appels système de gestion des threads
(*pthread_attr_setdetachstate()*)

```
#include <pthread.h>
```

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);  
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
```

Compile and link with `-pthread`.

DESCRIPTION

[top](#)

The `pthread_attr_setdetachstate()` function sets the detach state attribute of the thread attributes object referred to by `attr` to the value specified in `detachstate`. The detach state attribute determines whether a thread created using the thread attributes object `attr` will be created in a joinable or a detached state.

The following values may be specified in `detachstate`:

PTHREAD_CREATE_DETACHED

Threads that are created using `attr` will be created in a detached state.

PTHREAD_CREATE_JOINABLE

Threads that are created using `attr` will be created in a joinable state.

The default setting of the detach state attribute in a newly initialized thread attributes object is **PTHREAD_CREATE_JOINABLE**.

The `pthread_attr_getdetachstate()` returns the detach state attribute of the thread attributes object `attr` in the buffer pointed to by `detachstate`.

RETURN VALUE

[top](#)

On success, these functions return 0; on error, they return a nonzero error number.

Threads (Appels système de base)

- Appels système de gestion des threads (***pthread_detach()***)

```
#include <pthread.h>

int pthread_detach(pthread_t thread);

Compile and link with -pthread.
```

DESCRIPTION [top](#)

The `pthread_detach()` function marks the thread identified by `thread` as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Attempting to detach an already detached thread results in unspecified behavior.

RETURN VALUE [top](#)

On success, `pthread_detach()` returns 0; on error, it returns an error number.

Threads (Appels système de base)

- Appels système de gestion des threads (*pthread_exit()*)

```
#include <pthread.h>

void pthread_exit(void *retval);

Compile and link with -pthread.
```

DESCRIPTION

[top](#)

The `pthread_exit()` function terminates the calling thread and returns a value via `retval` that (if the thread is joinable) is available to another thread in the same process that calls `pthread_join(3)`.

Any clean-up handlers established by `pthread_cleanup_push(3)` that have not yet been popped, are popped (in the reverse of the order in which they were pushed) and executed. If the thread has any thread-specific data, then, after the clean-up handlers have been executed, the corresponding destructor functions are called, in an unspecified order.

When a thread terminates, process-shared resources (e.g., mutexes, condition variables, semaphores, and file descriptors) are not released, and functions registered using `atexit(3)` are not called.

After the last thread in a process terminates, the process terminates as by calling `exit(3)` with an exit status of zero; thus, process-shared resources are released and functions registered using `atexit(3)` are called.

RETURN VALUE

[top](#)

This function does not return to the caller.

Threads (Appels système de base)

- Création d'un thread (Posix)

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */

void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of attributes for the thread */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        /*exit(1);*/
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "Argument %d must be non-negative\n", atoi(argv[1]));
        /*exit(1);*/
        return -1;
    }

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);

    /* now wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}

/**
 * The thread will begin control in this function
 */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    pthread_exit(0);
}
```

// argv[1] pointe sur la chaîne de caractères du nombre d'itérations

Threads (Appels système de base)

- Création d'un thread (Win32)

```
#include <stdio.h>
#include <windows.h>

DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(PVOID Param)
{
    DWORD Upper = *(DWORD *)Param;

    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;

    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    // do some basic error checking
    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }

    Param = atoi(argv[1]);

    if (Param < 0) {
        fprintf(stderr, "an integer >= 0 is required\n");
        return -1;
    }

    // create the thread
    ThreadHandle = CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId);

    if (ThreadHandle != NULL) {
        WaitForSingleObject(ThreadHandle, INFINITE);
        CloseHandle(ThreadHandle);
        printf("sum = %d\n", Sum);
    }
}
```

Threads (Appels système de base)

- Création d'un thread (JAVA)

```
class Sum
{
    private int sum;

    public int get() {
        return sum;
    }

    public void set(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        if (upper < 0)
            throw new IllegalArgumentException();

        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;

        for (int i = 0; i <= upper; i++)
            sum += i;

        sumValue.set(sum);
    }
}

public class Driver
{
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage Driver <integer>");
            System.exit(0);
        }

        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);

        Thread worker = new Thread(new Summation(upper, sumObject));
        worker.start();
        try {
            worker.join();
        } catch (InterruptedException ie) { }
        System.out.println("The sum of " + upper + " is " + sumObject.get());
    }
}
```