

Introduction aux systèmes d'exploitation

Abdelouahed Gherbi

Hiver 2024

Plan

- Qu'est ce qu'un système d'exploitation?
- Évolution des systèmes d'exploitation
- Modes d'opération des OS
- Mécanisme des interruptions
- Les appels systèmes
- Structures des systèmes d'exploitation

Qu'est ce qu'un système
d'exploitation?

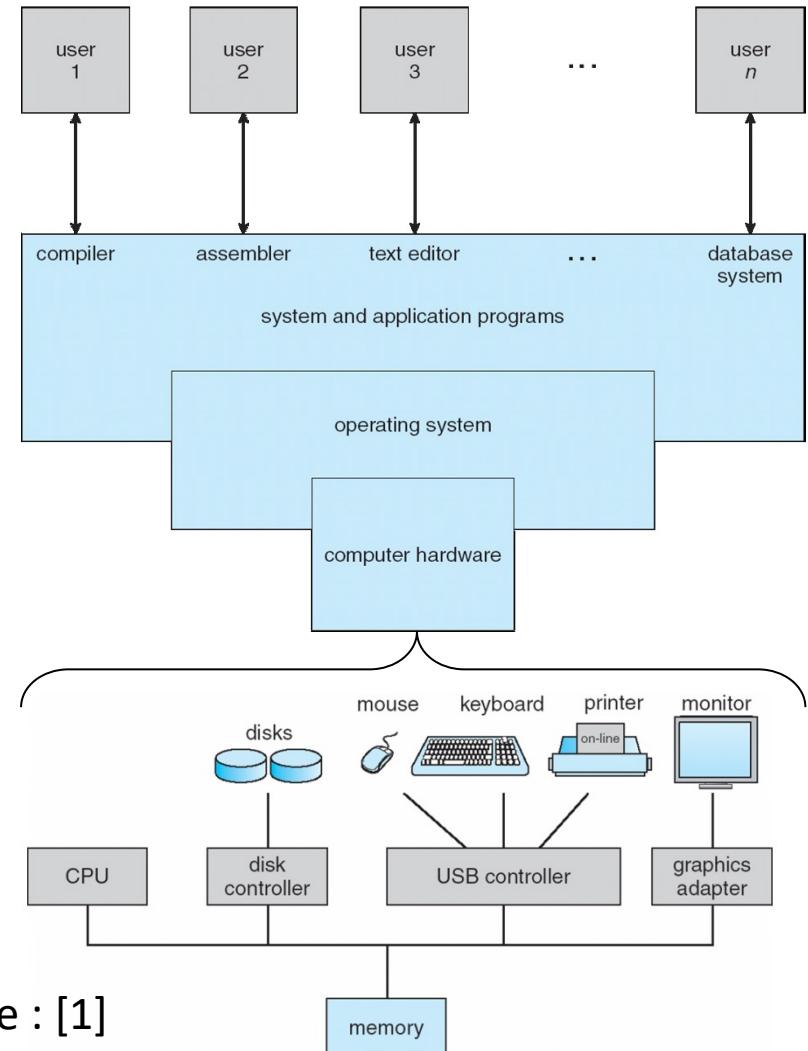
Quels sont les objectifs d'un OS?

Qu'est ce qu'un système d'exploitation?

- Un logiciel qui agit comme intermédiaire entre l'utilisateur d'un ordinateur et le matériel de l'ordi.
- Objectifs d'un OS:
 - Fournir un environnement d'exécution pour les programmes
 - Rendre l'ordinateur plus facile (commode) à utiliser
 - Utiliser le matériel de l'ordinateur plus efficacement

Structure générale d'un système informatique

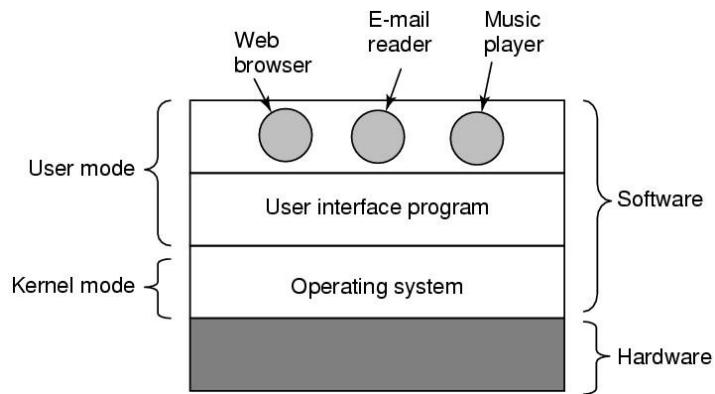
- Un système informatique est structuré comme suit :
 - Matériel (Hardware)
 - offre les ressources de traitement de base
 - CPU, mémoire, périphériques d'E/S
 - Système d'exploitation
 - Programme qui contrôle et coordonne l'utilisation du matériel entre les applications et utilisateurs
 - Programmes d'application
 - Définissent les façons d'utiliser les ressources du système pour résoudre les problème de calcul (traitement d'information) des utilisateurs
 - Traitement de texte, compilateurs, navigateur web, systèmes de bases de données, jeux vidéo
- Utilisateurs
 - Individus, machines, autres ordis.



Source : [1]

Définition d'un système d'exploitation

- Il n'y a pas de définition universellement acceptée de ce qui fait partie d'un système d'exploitation [1]
- Il est difficile cependant de décrire précisément en quoi un tel système consiste [2]
- Une définition plus commune : “Le programme d'un cours d'exécution en tout temps sur l'ordinateur” : le noyau (kernel).
- Tout autre chose est :
 - Un programme système
 - Un programme d'application
- En 1998, le Département of Justice des États Unis a poursuivit Microsoft prétendant que Microsoft incluait à tort trop de fonctionnalités dans son OS et ainsi empêchait la concurrence (compétition)
 - P. ex. : Un navigateur Web faisait partie de l'OS.

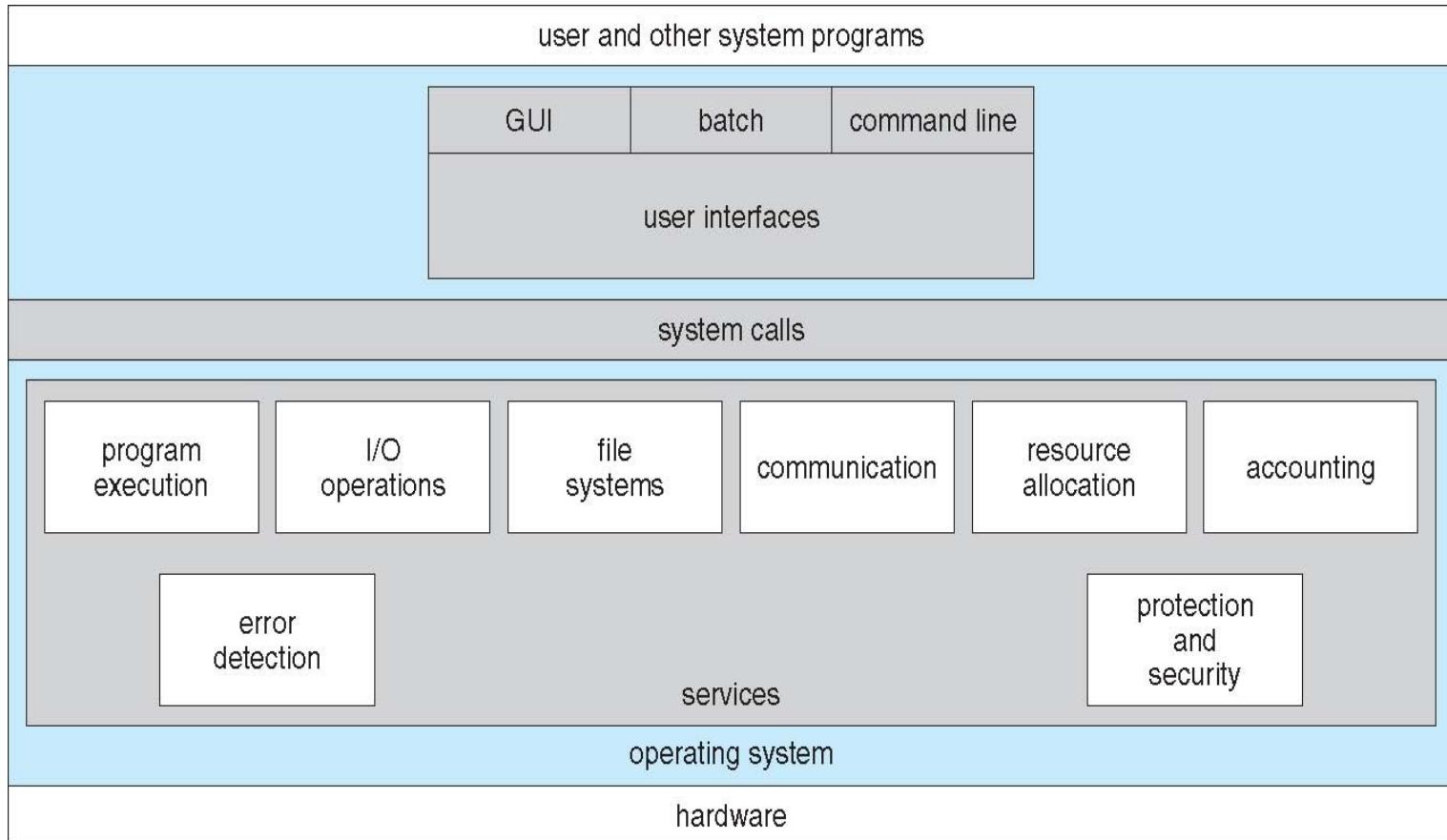


Source : [2]

Définition d'un système d'exploitation

- OS définit une abstraction par dessus la machine réelle
 - Cache les détails complexes à l'utilisateur
 - Présente à l'utilisateur une machine virtuelle facile à utiliser et à programmer
 - Principales abstractions définies par l'OS : Processus, Fichier, mémoire virtuelle, etc.
- Un OS est un allocateur de ressources
 - Gère les ressources matérielles
 - Prends des décisions pour assurer une utilisation équitable et efficace des ressources
- Un OS est un programme de contrôle
 - Il contrôle l'exécution des programmes afin de prévenir les erreurs et/ou mal utilisation des ressources de l'ordi

Services des systèmes d'exploitation



Une vue globale des services des OS [1]

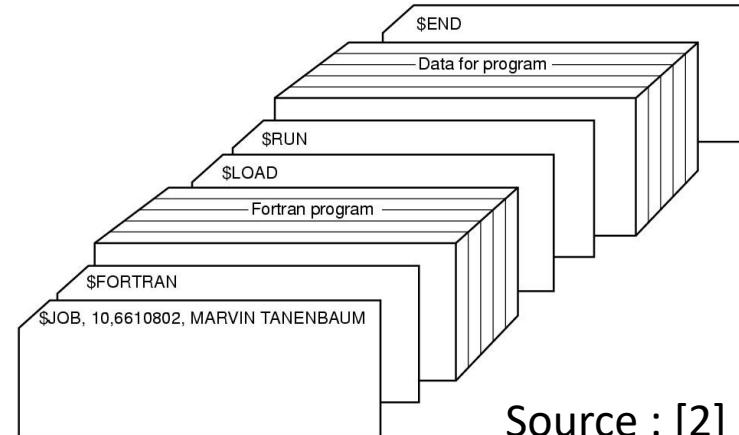
Évolution des systèmes d'exploitation

Évolution des systèmes d'exploitation

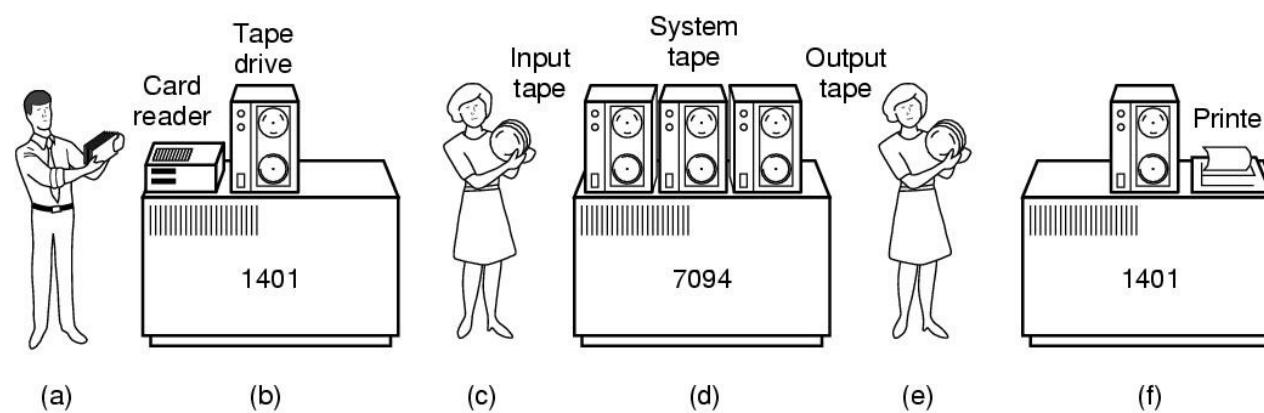
- Première génération (1945-55) : Exploitation porte ouverte [2]
- Tubes à vides
- Le même groupe de personnes font la conception, programmation, opération et maintenance des machines.
- Programmation en langage machine
- Pas de système d'exploitation
- Le programmeur a accès direct à la machine
- Problèmes traités : calcul numériques (tables de sinus, logarithmes etc.)

Évolution des systèmes d'exploitation

- Deuxième génération (1955-65)[2]
- Introduction des transistors
- Les machines (mainframes) plus fiables
- Les programmeurs n'ont plus l'accès direct à la machine
- Ils/Elles Soumettent des **Jobs** (program, données et directives sous forme de cartes perforées)
- Langage de programmation FORTRAN ou l'assembleur
- Treatment par lots (Batch systems)
- Exemple d'OS: Fortran Monitor System (FMS) et le système d'IBM pour le 7094 (IBSYS)



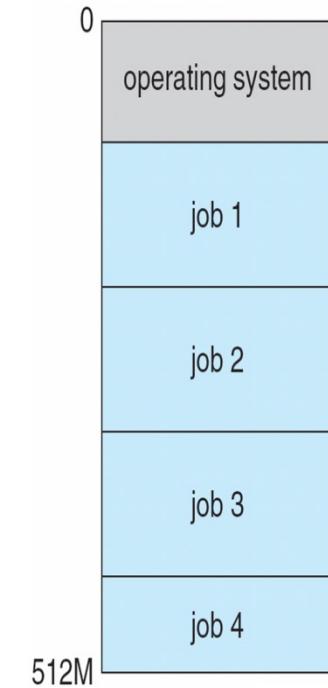
Source : [2]



Source : [2]

Évolution des systèmes d'exploitation

- Troisième génération (1965-80)[2]
- Circuits intégrés → avantage prix/performance
- Famille d'ordis compatibles : IBM 360 (OS/360)
- La technique la plus importante :
Multiprogrammation
 - Un seul programme (job) ne peut pas garder la CPU occupé tout le temps
 - Un sous ensemble des Jobs dans le système sont chargés en mémoire
 - La CPU a toujours un programme à exécuter
 - Un job est sélectionné pour exécution (ordonnancement – job scheduling)
Quand un job doit attendre une entrée/sortie, OS alloue la CPU à un autre Job immédiatement



Source : [1]

Évolution des systèmes d'exploitation

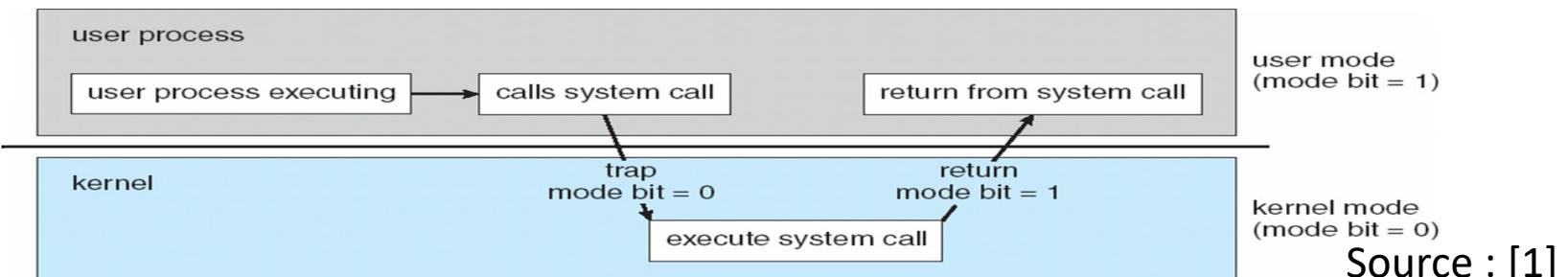
- La deuxième technique importante : temps partagé – (Timesharing)
 - Une extension logique de la multiprogrammation où la CPU est alloué successivement (et fréquemment) par quantum de temps aux programmes chargés en mémoire
 - Ceci arrive si fréquemment que les utilisateurs peuvent interagir avec le système et leurs programmes -> Système interactifs
 - Temps de réponses doit être < 1 seconde
 - Chaque utilisateur a au moins un programme en exécution en mémoire : processus
 - Si plusieurs jobs sont prêts pour exécution en même temps : ordonnancement de la CPU
 - Si un processus ne peut pas être chargé au complet en mémoire : technique du va et viens (Swapping) et mémoire virtuelle
- Exemple de OS:
 - Compatible Time Sharing System (CTSS) :MIT
 - MULTplexed Information and Computing Service (MULTICS)
 - Naissance et développement de UNIX (System V, BSD, IEEE POSIX)

Évolution des systèmes d'exploitation

- Quatrième génération (1980- à présent)[2]
- Circuits intégrés à haute densité LSI (Large Scale Integration)
- Control Program for MultiComputers (CP/M)
- Microsoft Disk Operating System (MS-DOS)
- Interface utilisateur graphique (GUI), Engelbart, Xerox PARC
- Apple Macintosh, Steve Jobs
- Microsoft Windows

Modes d'opération des OS

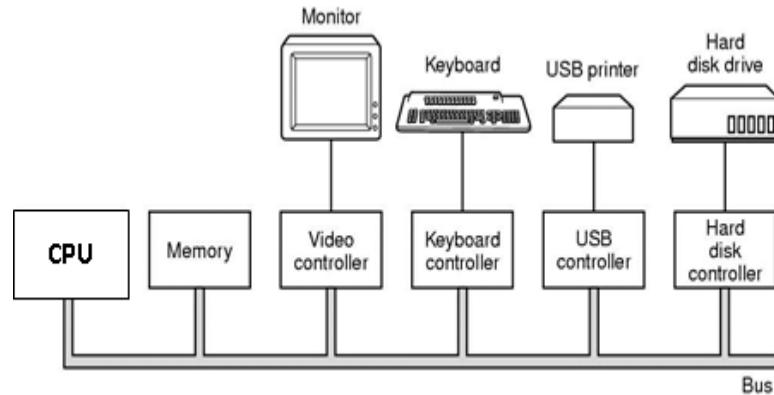
- L'OS et les programmes utilisateurs partagent les ressources (cohabitent)
- On a besoin de prévenir que les erreurs dans les programmes utilisateurs causent des problèmes pour les autres programmes en exécution
 - Inter-blocage, modification d'un processus d'autres processus, etc.
- On doit distinguer l'exécution du code de l'OS du code d'autres applications.
- Approche : Supporter deux modes de fonctionnement (dual mode) pour protéger l'OS et les autres composants
 - Mode utilisateur et le mode superviseur (kernel, privilégié)
 - Certaines instructions sont dites privilégiées : exécutables seulement en mode superviseur
 - Les appels systèmes provoquent le changement du modes



Mécanisme des interruptions

Mécanisme des interruptions

- L'OS (noyau) est responsable de la gestion du matériel composant le système informatique
- La CPU doit communiquer avec des périphériques externes
 - asynchrones (ayant des cycles d'horloge différents)
 - Beaucoup moins rapide
- Comment la CPU peut être capable de traiter les évènements générés par les périphs. externes et répondre rapidement?
- Comment la CPU peut faire du travail utile en attendant ces évènements ?



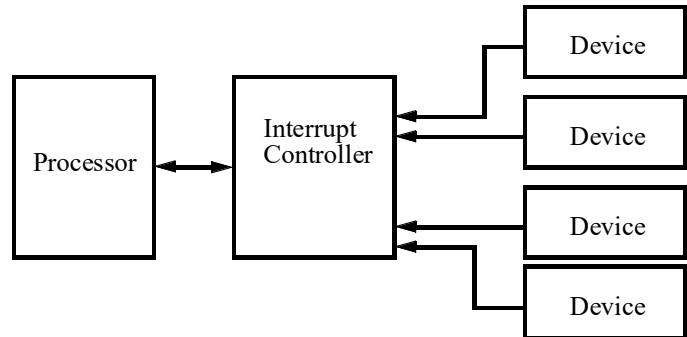
Source [2]

Mécanisme des interruptions

- Deux approches alternatives sont possibles
- Première solution : Scruter les périphériques (polling)
 - Aller vérifier l'état des périphériques (s'ils ont des requêtes?, terminé?) de temps en temps
- Avantage : Simple
- Inconvénients :
 - Prends du temps de la CPU même s'il n'y a pas de requêtes
 - Quand faire le polling ? Difficile à prédire.

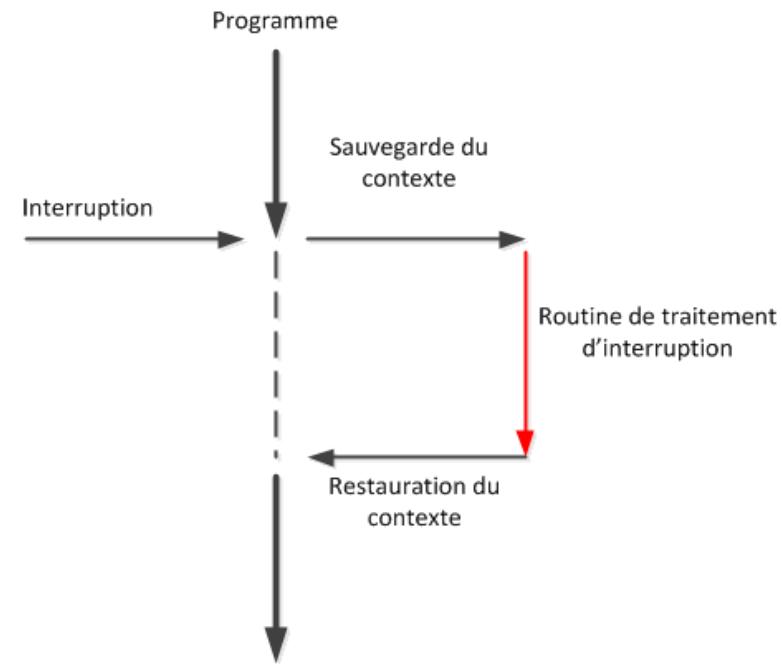
Mécanisme des interruptions

- Deuxième solution
- Interruption
- Chaque périph. peut envoyer un signal à la CPU pour indiquer un événement (requête)
- Pas de couts de scrutement des périphériques alors qu'il n'y a pas de requêtes
- Réponse plus rapide



Mécanisme des interruptions

- Une interruption est un événement qui interrompt le déroulement d'un programme en cours d'exécution par l'unité centrale
- Objectif :
 - Reprendre le contrôle de la CPU et exécuter une tache plus urgente
 - Réagir le plus rapidement possible aux événements externes
- L'interruption transfert le control de la CPU à une routine de traitement d'interruption
- Catégories des interruptions
 - Interruption externes (matériel, l'horloge)
 - Interruption interne :
 - Déroutement (Trap) : appel système
 - Exception utilisée pour traiter des erreurs comme la division par zéro, accès à une adresse protégée ou non existante, tentative d'exécution d'instruction privilégiée en mode utilisateur



Mécanisme des interruptions

- Le mécanisme des interruptions permet donc à la CPU de répondre aux évènements asynchrones
- Dans un OS moderne on a besoin de fonctionnalités de gestion d'interruption plus sophistiquées
 - On a besoin de pouvoir différer la gestion d'une interruption lors d'un traitement critique
 - On a besoin d'un moyen efficace de se brancher au gestionnaire d'interruption approprié à un périphérique sans avoir à scruter tous les périphériques pour déterminer celui qui a levé l'IT
 - On a besoin de plusieurs niveaux d'IT pour que l'OS soit capable de distinguer entre des interruptions de priorité élevée de celles moins prioritaires et répondre avec le degré d'urgence approprié

Mécanisme des interruptions

- Dans un système informatique moderne, ces trois caractéristiques sont supportées par la CPU et un **contrôleur d'interruptions**
- On distingue :
 - Interruptions **non masquables** :
 - réservées pour les évènements comme les erreurs de mémoire non récupérables
 - Interruptions **masquables** :
 - peut être **désactivée** par la CPU avant l'exécution d'une **séquence d'instructions critiques** à ne pas interrompre

Mécanisme des interruptions

- Le mécanisme des interruptions utilise le numéro de l'interruption
 - Ce numéro permet d'accéder (trouver l'adresse) d'une routine de traitement d'interruption adéquate
- Cette adresse est un déplacement (offset) dans une table appelée vecteur d'interruption
- Ce vecteur d'interruption contient les adresses en mémoire des routines de traitement d'interruption spécialisées
- Un mécanisme d'interruption vectorisé permet d'éviter un gestionnaire d'interruption unique qui cherche toutes les sources possibles d'interruption pour déterminer celle à traiter.

Mécanisme des interruptions

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

- Table des vecteurs d'évènement du processeur Intel Pentium [1]
- Les interruptions 0 à 31 sont non masquables et signalent diverses situations d'erreurs
- Les interruptions 32 à 255 sont masquables et sont utilisés pour traiter les interruptions des périphériques

Mécanisme des interruptions

- procfs est un système de fichiers virtuel qui existe seulement dans la mémoire du noyau Linux
- procfs monté sur /proc
- Permet de se renseigner sur diverses informations sur le noyau
- En particulier le fichier /proc/interrupts donne des statistiques sur les interruptions dans le système
- Exemple : cat /proc/interrupts

Mécanisme des interruptions

- Exemple

```
$ cat /proc/interrupts
      CPU0
 0: 865119901      IO-APIC-edge    timer
 1:          4      IO-APIC-edge    keyboard
 2:          0        XT-PIC         cascade
 8:          1      IO-APIC-edge    rtc
12:         20      IO-APIC-edge    PS/2 Mouse
14: 6532494      IO-APIC-edge    ide0
15:         34      IO-APIC-edge    ide1
16:          0      IO-APIC-level   usb-uhci
19:          0      IO-APIC-level   usb-uhci
23:          0      IO-APIC-level   ehci-hcd
32:         40      IO-APIC-level   ioc0
33:         40      IO-APIC-level   ioc1
48: 273306628      IO-APIC-level   eth0
NMI:          0
ERR:          0
```

- Colonnes: IRQ, nombre d'interrupt, Contrôleur d'interruption et périphérique

Mécanisme des interruptions

- Exemple

```
$ watch -d -n.5 "cat /proc/interrupts"
```

Cette commande affiche dynamiquement les informations par niveau d'interruptions (IRQ)

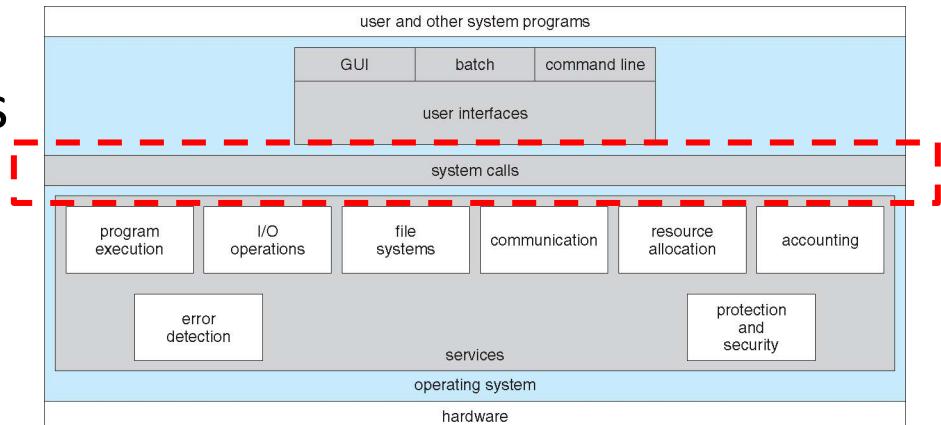
Mécanisme des interruptions

```
1  |
2  // LOG 710 - Semaine 01 - Exemple 01
3  #include <signal.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <sys/time.h>
7
8  void gestionnaire_timer (int signum)
9  {
10     static int count = 0;
11     printf ("timer a expire %d fois\n", ++count);
12 }
13
14 int main ()
15 {
16     struct sigaction sa;
17     struct itimerval timer;
18
19     /* Installe le gestionnaire du signal SIGVTALRM. */
20     memset (&sa, 0, sizeof (sa));
21     sa.sa_handler = &gestionnaire_timer;
22     sigaction (SIGVTALRM, &sa, NULL);
23
24     /* Configurer le timer pour expirer apres 250 msec */
25     timer.it_value.tv_sec = 0;
26     timer.it_value.tv_usec = 250000;
27     /* ... et a chaque 250 msec apres. */
28     timer.it_interval.tv_sec = 0;
29     timer.it_interval.tv_usec = 250000;
30     /* Demarrer le timer. . */
31     setitimer (ITIMER_VIRTUAL, &timer, NULL);
32
33     /* Simule un certain calcul ... */
34     while (1);
35 }
```

Les appels Système

Les appels système

- Une interface de programmation aux services fournis par l'OS
- Généralement disponibles comme fonctions dans un langage de haut niveau (C ou C++)
- Séquence d'appels systèmes impliqués dans une copie du contenu d'un fichier



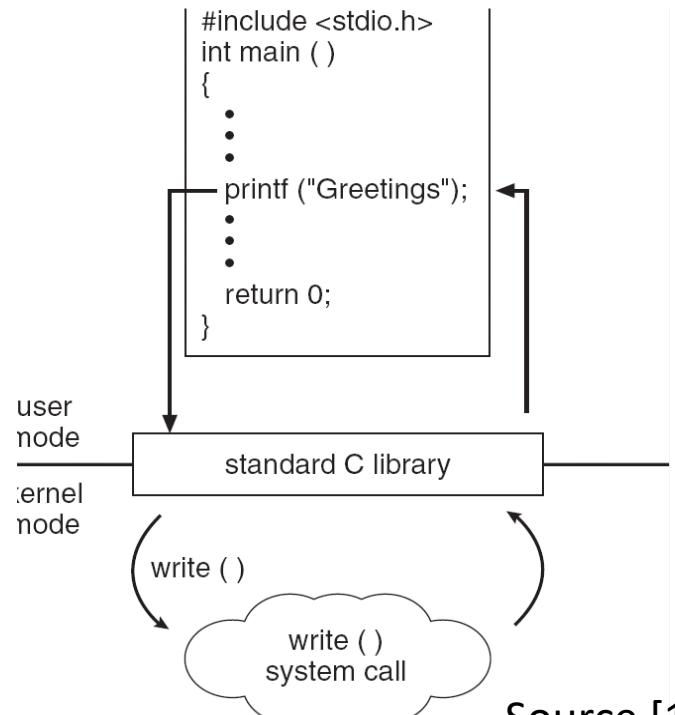
Example System Call Sequence

```
Acquire input file name
Write prompt to screen
Accept input
Acquire output file name
Write prompt to screen
Accept input
Open the input file
if file doesn't exist, abort
Create output file
if file exists, abort
Loop
Read from input file
Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Source [1]

Les appels système

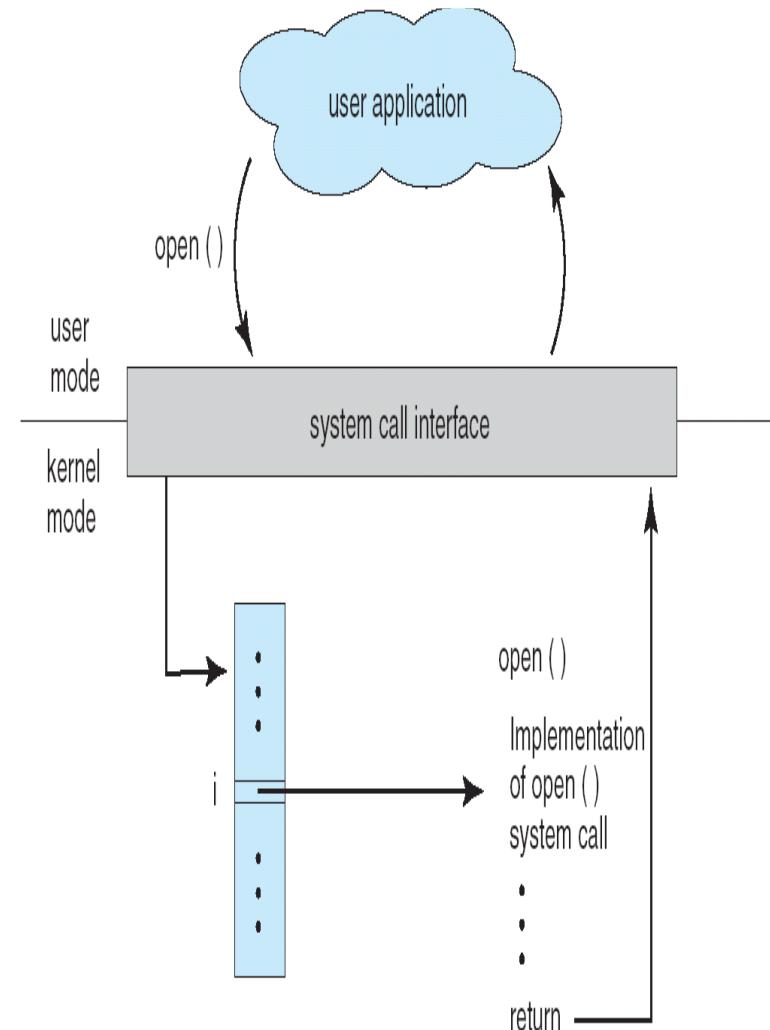
- Les appels systèmes ne sont pas accessibles directement mais via des interfaces de programmation d'application (API)
 - Une API spécifie un ensemble de fonctions disponibles pour le programmeur (noms, paramètres et valeurs de retour)
- Trois APIs communes :
 - API Win32 pour Windows
 - API POSIX (**Portable Operating System Interface for Unix**) pour les systèmes basés sur (UNIX, Linux, and Mac OS X)
 - API Java pour la machine virtuelle Java (JVM)
- Les bibliothèques d'exécution (runtime libraries) des langages de programmations (Java, C, C++, C#) implémentent l'interface d'appel système
- Cette interface d'appel système interceptent les appels de fonctions de l'API et invoquent l'appel système correspondant dans le noyau de l'OS et retournent le statut de l'appel système ainsi que la valeur de retour



Source [1]

Implémentation des appels système

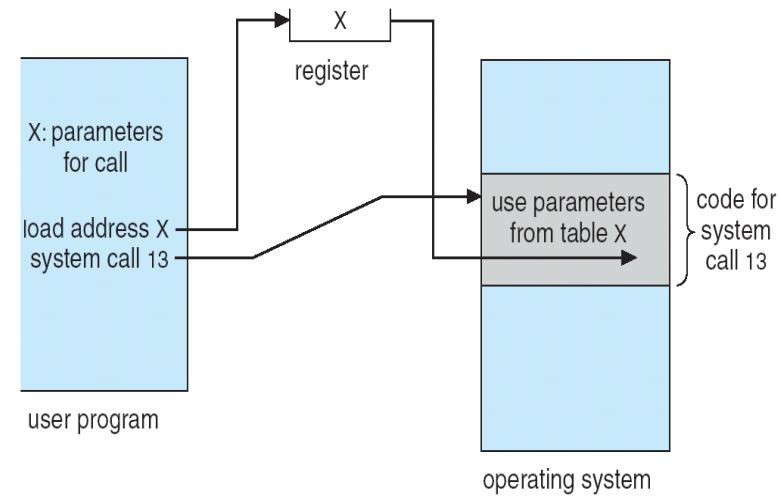
- Typiquement, un nombre est associé à chaque appel système (System Call ID)
 - Le noyau maintient une table indexée par ces nombres
- L'appelant n'a nullement besoin de connaître les détails d'implémentation et de fonctionnement des appels systèmes
 - Il a seulement besoin de respecter l'API et comprendre ce que l'OS fait pour chaque appel
 - L'implémentation de l'API (librairie C) cache les détails de l'OS au programmeur



Source [1]

Passage de paramètres l'OS

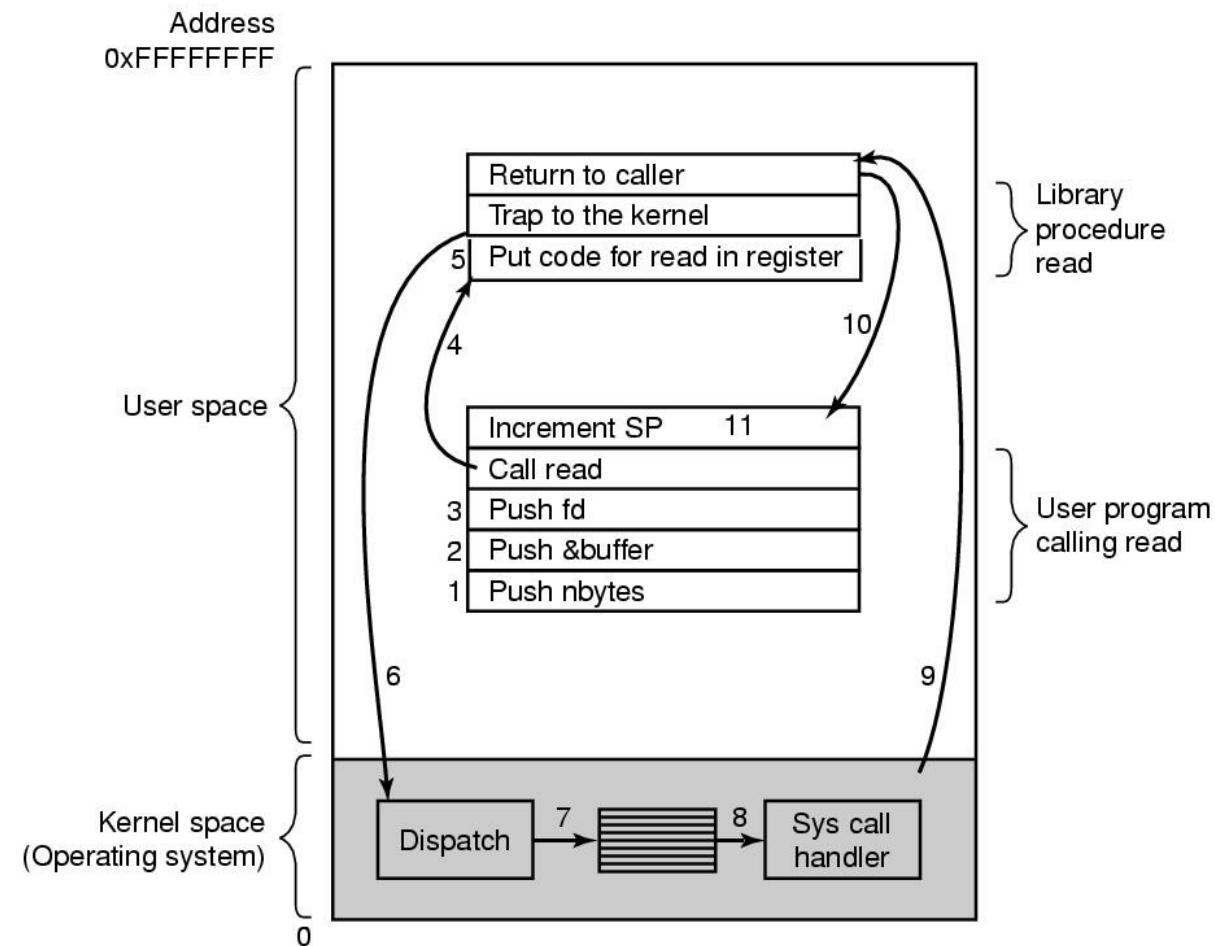
- Trois méthodes générales peuvent être utilisées pour passer des paramètres à l'OS lors d'un appel système
- Passage de paramètres via des registres
 - La plus simple mais parfois on a plus de paramètres que de registres disponibles
- Mettre les paramètres dans un bloc ou table en mémoire et passer l'adresse du bloc dans un registre
 - Cette approche adoptée par Linux et Solaris
- Le programme place les paramètres dans la pile et l'OS les dépilent
- Les méthodes du bloc et de pile ne limitent pas le nombre ni la longueur des paramètres transmis



Source [1]

Implémentation des appels système

- Les appels systèmes sont implémentés en utilisant le mécanisme des interruptions
- Les différentes étapes impliquées dans un appel système `read(fd, buffer, nbytes);`



Source [2]

Un exemple de programme C

- Les fonctions ‘write()’ et ‘exit()’ ne sont pas définies dans le code du programme
 - fonctions externes
- Les fichiers en-tête permettent au compilateur de générer le code machine (assembleur) qui appelle ces fonctions externes
- Ces fonctions sont définies dans la librairie standard GNU/C (‘glibc’)
- L’éditeur des liens (the linker) lie le code du programme à celui de la librairie ‘glibc’

```
# include <unistd.h> // pour write()
# include <stdlib.h> // pour exit()

char      message[ ] = "Hello!\n";

int main( void )
{
    write( 1, message, 7 );
    exit( 0 );
}
```

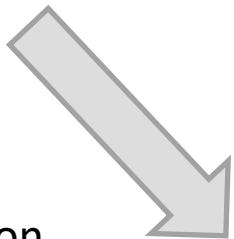
Un exemple de programme C

```
# include <unistd.h> // pour write()
# include <stdlib.h> // pour exit()

char message[ ] = "Hello!\n";

int main( void )
{
    write( 1, message, 7 );
    exit( 0 );
}
```

Après
compilation
et édition
des liens



```
pushl $7      # parameter no. 3
pushl message # parameter no. 2
pushl $1      # parameter no. 1
call  write   # call to
            # C library
addl $12, %esp # supprime
            # params

pushl $0      # parameter no. 1
call  exit   # call to C library
```

Détails d'implémentation de la fonction write

```
write: pushl  %ebp          # Sauvegarder EBP
       movl  %esp, %ebp        # EBP pointe vers le top de
                                # la pile

# Copier les parameters dans les registres
movl  $4, %eax           # Numéro de l'appel system
                           # sys_WRITE ID-number
movl  8(%ebp), %ebx      # parameter no. 1 to EBX
movl  12(%ebp), %ecx     # parameter no. 2 to ECX
movl  16(%ebp), %edx     # parameter no. 3 to EDX
int   $0x80                # Déclencher l'IT pour
                            # entrer the linux kernel

popl  %ebp          # restore previous EBP
ret                  # return to the caller
```

Détails d'implémentation de la fonction ‘exit’

```
exit: pushl    %ebp          # Sauvegarder EBP
      movl    %esp, %ebp       # EBP pointe vers le top de
                                # la pile

                                # copier les paramètres dans les registres
      movl    $1, %eax        # Numéro de l'appel
                                # système
      movl    8(%ebp), %ebx   # parameter no. 1 to EBX
      int     $0x80            # enter the linux kernel
```

Types d'appels systèmes

- Les appels systèmes peuvent être groupés en plusieurs catégories :
 - Control de processus
 - Communication
 - Gestion de mémoire
 - Gestion de fichiers
 - Etc.
- Dans ce cours on examinera les concepts, mécanismes et algorithmes utilisés dans les OS pour supporter ces différents types d'appels systèmes

Exemples d'appels systèmes dans Windows et Unix

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Source [2]

Appel systèmes : Exemple

```
1 // LOG 710 - Semaine 01 - Exemple 02
2 #include <linux/kernel.h>
3 #include <stdio.h>
4 #include <sys/sysinfo.h>
5
6
7 int main ()
8 {
9     /* Quelques constantes. */
10    const long minute = 60;
11    const long hour = minute * 60;
12    const long day = hour * 24;
13    const double megabyte = 1024 * 1024;
14    /* Obtenir les statistiques du système en utilisant l'appel système sysinfo. */
15    struct sysinfo si;
16    sysinfo (&si);
17    /* Sommaire de quelques valeurs intéressantes. */
18    printf ("Le système tourne pour (uptime) : %ld jours %ld:%02ld:%02ld\n",
19           si.uptime / day, (si.uptime % day) / hour,
20           (si.uptime % hour) / minute, si.uptime % minute);
21    printf ("Quantité totale de la RAM      : %5.1f MB\n", si.totalram / megabyte);
22    printf ("Quantité libre de la  RAM      : %5.1f MB\n", si.freeram / megabyte);
23    printf ("Nombre de processus   : %d\n", si.procs);
24
25    return 0;
26 }
```

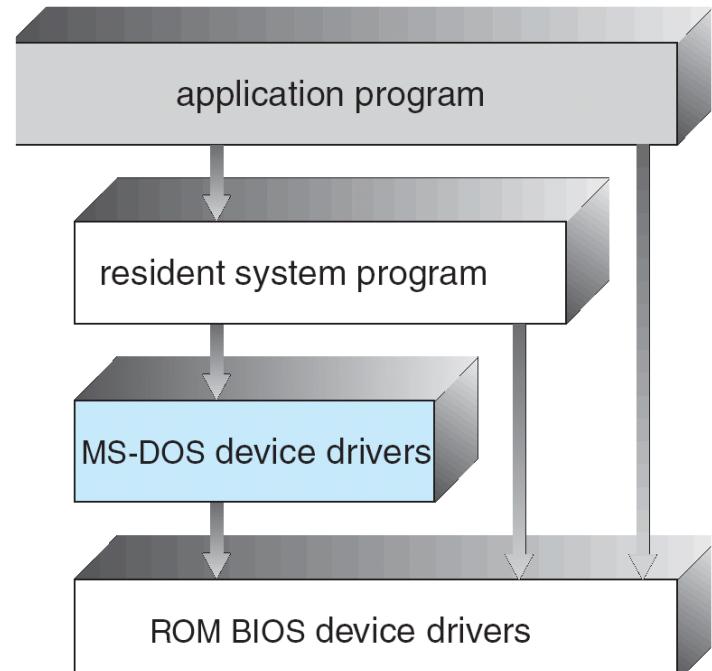
Structure des systèmes d'exploitation

Structure des systèmes d'exploitation

- Un OS moderne est un système large et complexe
- Il est important de le concevoir pour qu'il :
 - Fonctionne correctement
 - Rencontre certaines qualités de services (performance, robustesse, sécurité, équité)
 - Soit maintenu (modifié et mis à jour) facilement
- Approche commune :
 - Partitionner l'OS en un ensemble de composants au lieu d'un seul composant **monolithique**
- Chacun de ces composants doit être bien défini en termes d'entrées, sorties et fonction

Structure simple

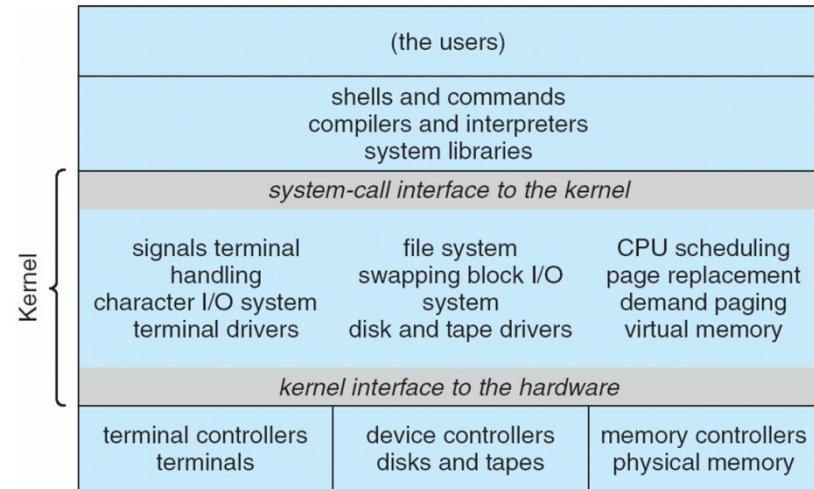
- Certains OS commerciaux n'ont pas une structure bien définie.
- Ils ont commencé comme des petits systèmes simples et limités
- Ils ont été étendus au fur et à mesure
- Exemple (MS-DOS)
 - Il a été implémenté pour fournir le plus de fonctionnalités avec le minimum d'espace
 - Il n'est pas divisé en modules
 - Ses interfaces et niveaux de fonctionnalités ne sont pas séparés



Structure générale de MSDOS [1]

Structure simple

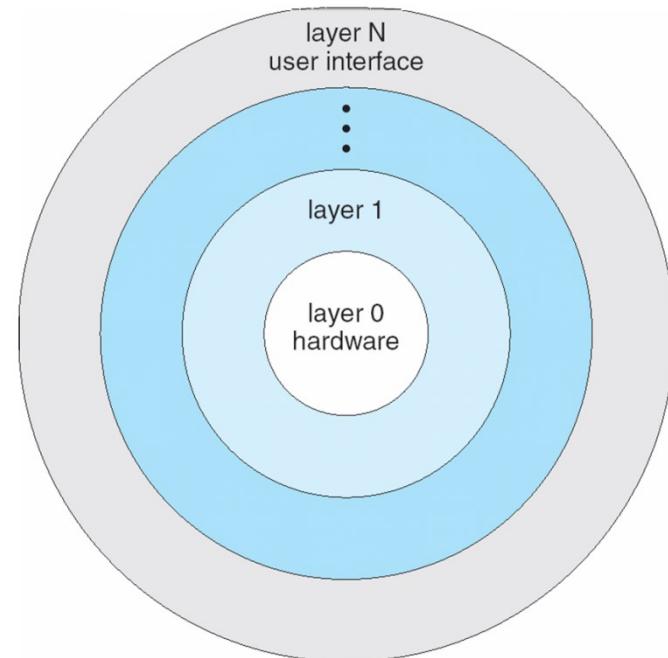
- Exemple 2 (UNIX)
- La version initiale d' UNIX avait une structure limitée
- UNIX était composé de deux parties
 - Les programmes systèmes
 - Le noyau (kernel)
 - Inclut tout ce qui est au dessous des appels systèmes et au dessus du matériel
 - Fournit dans un même niveau les fonctionnalités de gestion de fichiers, ordonnancement de la CPU, la gestion de la mémoire



Structure Traditionnelle de UNIX [1]

Approche en couches

- Selon cette approche, l'OS est divisé en un certain nombre de couches (layers) ou niveaux
- La couche 0 est le matériel
- La couche N est l'interface utilisateur
- Une couche M typique de l'OS consiste en structures de données et un ensemble de fonctions (routines) qui peuvent être appelées par les couches supérieures
- Une couche M à son tour peut invoquer les opérations des couches inférieures
- Une couche est implémentée en utilisant seulement les opérations offertes par les couches inférieure mais ne doit pas connaître comment ces opérations sont implémentées.
- Avantage
 - Modularité → Facilité de débogages/tests incrémentaux
- Inconvénients :
 - Rigidité
 - Difficulté d'une définition appropriée des couches : Les fonctionnalités de l'OS ne sont pas complètement indépendantes
 - Inefficacité



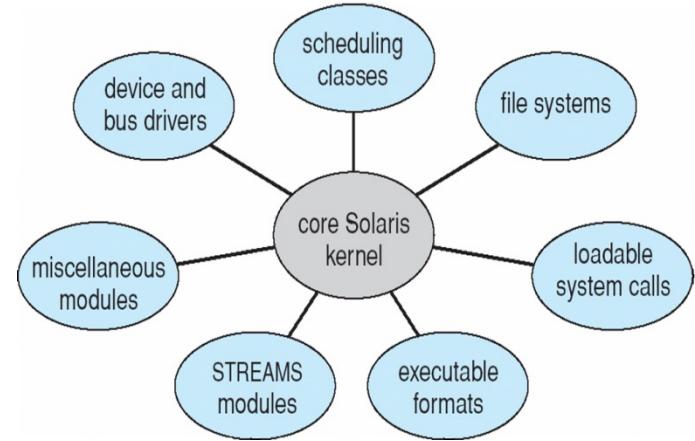
Un OS en couches [1]

Structure en micronoyau (microkernel)

- Déplacer les fonctionnalités non essentielles du noyau et les implémenter dans l'espace utilisateur
 - Résultat : un noyau très petit (un micronoyau)
- Le micronoyau fournit typiquement une gestion de processus et de mémoire minimale et une facilité de communication
- La communication entre les processus utilisateurs et les services qui tournent dans l'espace utilisateur
 - Cette communication se fait via passage de messages utilisant le micronoyau
- Exemples : Mach, OS/2, Minix 3
- Avantages :
 - Facilité d'étendre l'OS : tous le nouveau services ajouter dans l'espace utilisateur sans changer le micronoyau
 - Facilité d'extension (mise à jour du noyau) car plus petit.
 - Facilité de porter l'OS d'une architecture à une autre (noyau plus petit)
 - Plus de fiabilité et de sécurité (moins de code qui tourne en mode noyau)
- Inconvénients:
 - Moins de performance due au surcoût (overhead) de la communication entre l'espace utilisateur et l'espace noyau

Modules chargeables

- Les OS modernes implémentent un noyau modulaire
- Le noyau est composé d'un composant principal et peut lier dynamiquement (à l'initialisation ou en cours d'exécution) des services supplémentaires
 - Chaque composant est séparé
 - Chaque composant interagit avec les autres composants via des interfaces bien définies
 - Chaque composant est chargeable dynamiquement
- Cette approche est similaire à l'approche en couche mais plus flexible
- Cette approche est aussi similaire à l'approche du micronoyau mais elle est plus efficace
- Composant principal fait les fonctions essentielles et permet de charger et communiquer avec d'autres composants
- Plus efficace car les composants communiquent directement (dans le même espace noyau) et sans passage de message
- Exemples : Linux, Solaris



La structures en modules chargeables en Solaris [1]

Références

- [1] SILBERSCHATZ, A. et P.B. GALVIN, *Operating System Concepts*. 8th Edition, Addison Wesley.
- [2] Andrew Tanenbaum, Modern Operating Systems, 3rd edition, 2008

Processus et threads

Abdelouahed Gherbi

Log 710 Hiver 2024

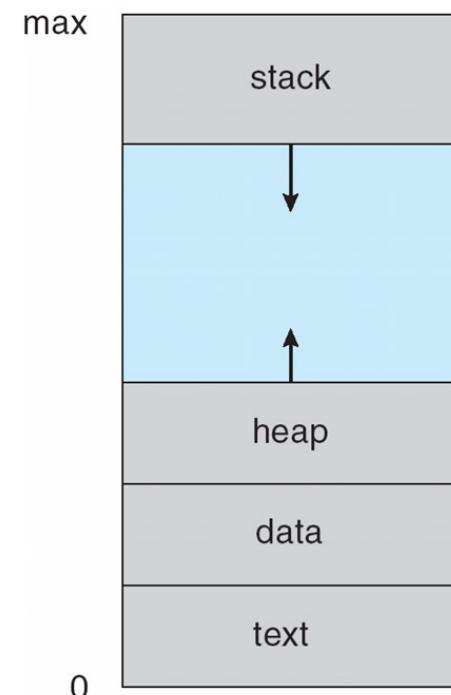
Plan

- Processus
 - Concept de processus
 - Ordonnancement de processus
 - Opérations sur les processus
- Threads
 - Qu'est ce qu'un thread ?
 - POXIS Threads

Processus

Concept de processus

- Un processus
 - une instance d'un programme en cours d'exécution
 - activité qui résulte, dans le système, de l'exécution d'un programme
- Un processus n'est pas seulement le code du programme
 - Section texte
 - Entité passive
- Un processus : **entité active** qui signifie l'activité courante représentée par
 - Le registre pointeur d'instruction (program counter)
 - Les autres registres du processeur
- Un processus inclut aussi
 - La pile d'exécution
 - La section de données
 - Le tas (heap)



Processus en mémoire [1]

```

#define _BSD_SOURCE
#include <stdio.h>
#include <stdlib.h>

char globBuf[65536]; /* Uninitialized data segment */
int primes[] = { 2, 3, 5, 7 }; /* Initialized data segment */

static int
square(int x) /* Allocated in frame for square() */
{
    int result; /* Allocated in frame for square() */

    result = x * x;
    return result; /* Return value passed via register */
}

static void
doCalc(int val) /* Allocated in frame for doCalc() */
{
    printf("The square of %d is %d\n", val, square(val));

    if (val < 1000) {
        int t; /* Allocated in frame for doCalc() */

        t = val * val * val;
        printf("The cube of %d is %d\n", val, t);
    }
}

int
main(int argc, char *argv[]) /* Allocated in frame for main() */
{
    static int key = 9973; /* Initialized data segment */
    static char mbuf[1024000]; /* Uninitialized data segment */
    char *p; /* Allocated in frame for main() */

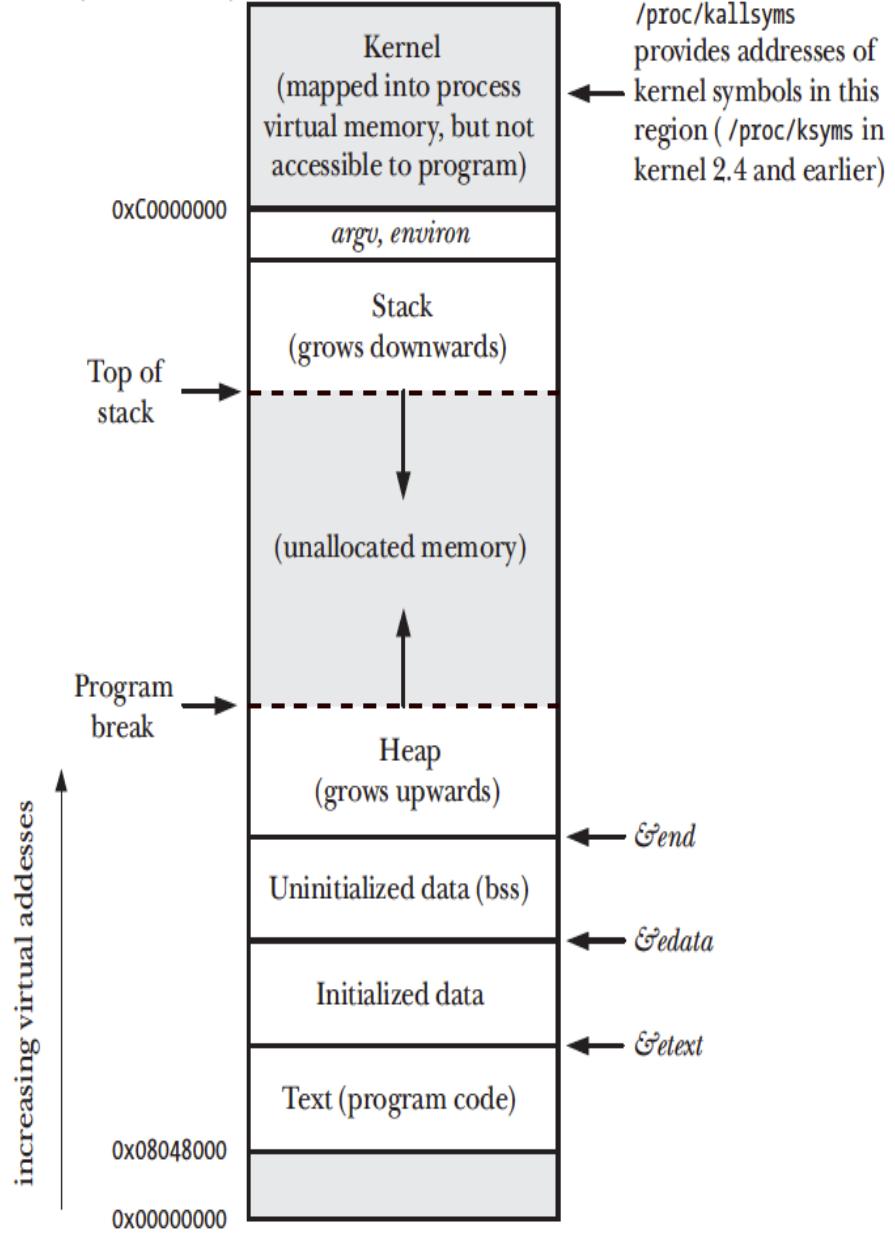
    p = malloc(1024); /* Points to memory in heap segment */

    doCalc(key);

    exit(EXIT_SUCCESS);
}

```

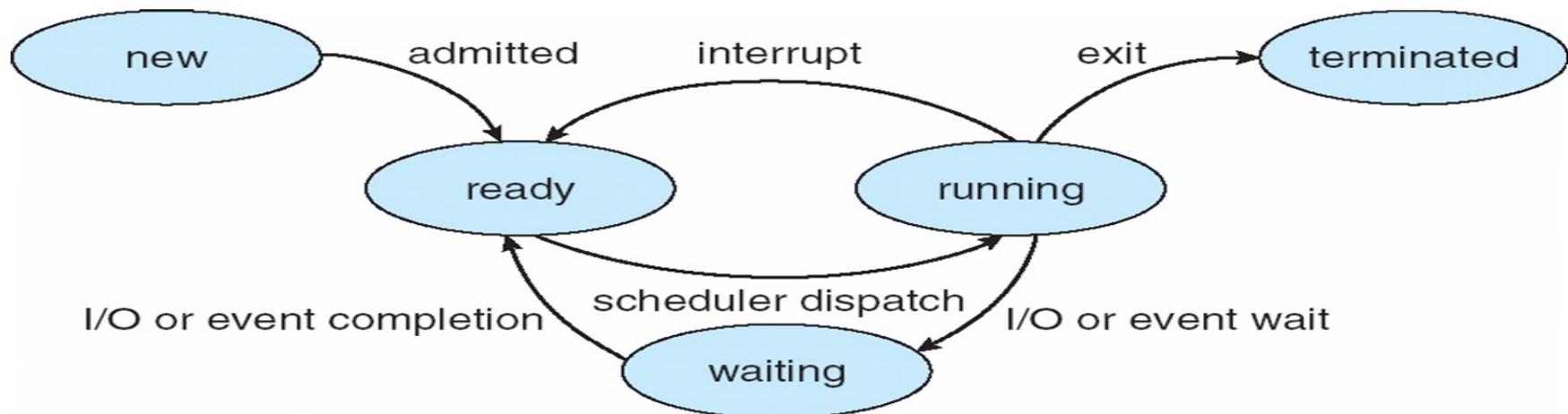
Virtual memory address
(hexadecimal)



État d'un processus

- Un processus change d'état au cours de son déroulement

- **Nouveau** : Le processus est entrain d'être créé
- **En exécution** : Les instructions du programme associé sont en cours d'exécution
- **En attente** : Le processus est en attente d'un évènement (La fin d'une opération d'E/S, Réception d'un signal)
- **Prêt** : Le processus est en attente d'être assigné au processeur
- **Terminé** : Le processus a terminé son exécution

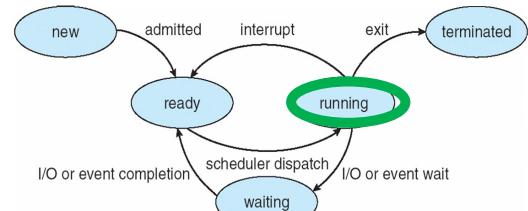


Combien de processus dans l'état
running ?
Ready ? Waiting ?

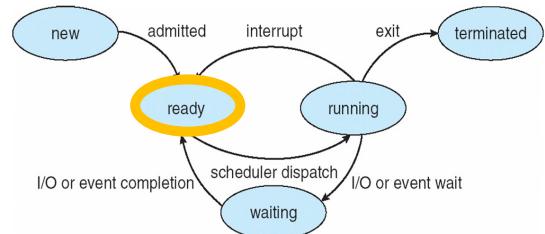
Diagramme d'état d'un processus [1]

État d'un processus

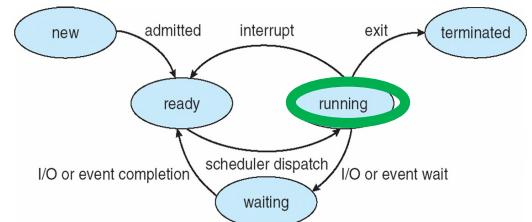
Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process ₀ now done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	Process ₁ now done



Process₀ aux instants 1, 2, 3 et 4



Process₁ aux instants 1, 2, 3 et 4

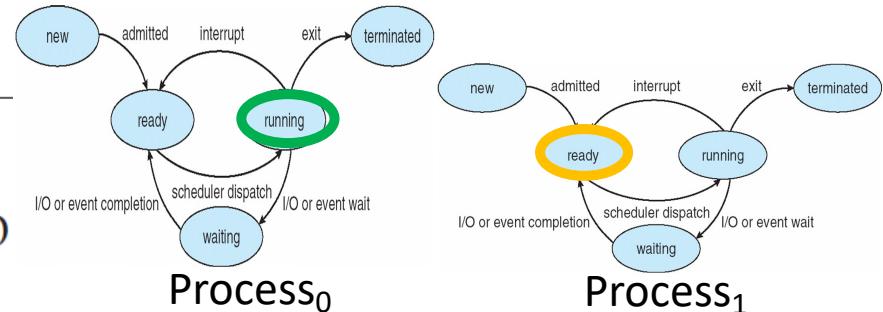


Process₁ aux instants 5, 6, 7 et 8

État d'un processus

Aux instants 1, 2, et 3

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process ₀ initiates I/O
4	Blocked	Running	Process ₀ is blocked, so Process ₁ runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process ₁ now done
9	Running	—	
10	Running	—	Process ₀ now done



Process₀ aux instants 4,
5 et 6

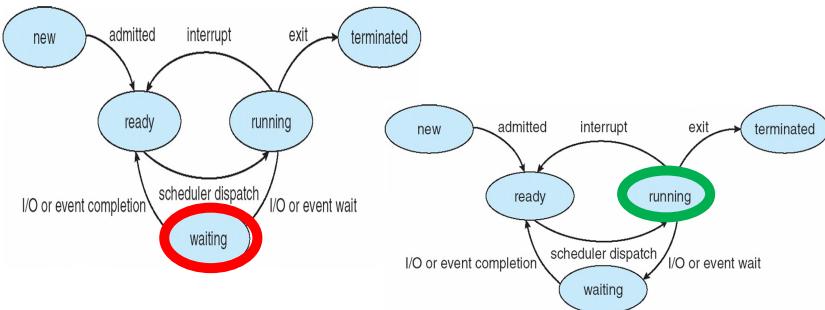
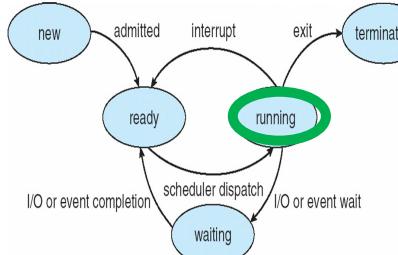
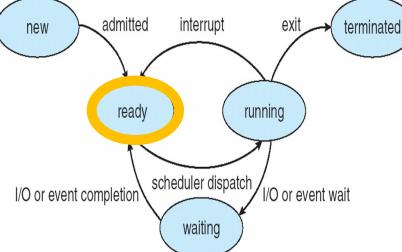


Figure 4.4: Tracing Process State: CPU and I/O

Process₀ aux instants 9
et 10



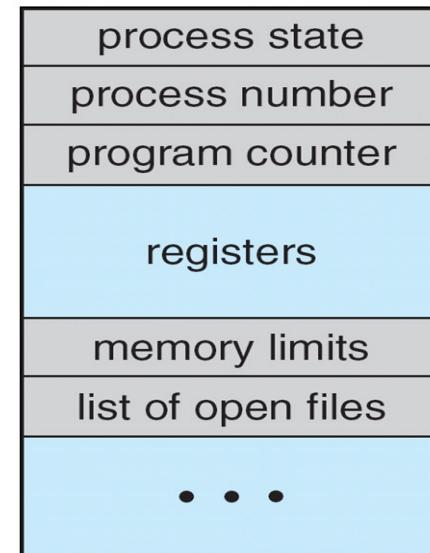
Process₀ aux instants 7
et 8



Process₁ aux
de l'instant 4
a l'instant 8

Bloc de control de processus

- Chaque processus est représenté dans l'OS par une structure de données
 - Bloc de control de processus (Process Control Block – PCB)
- Le PCB contient des informations sur le processus incluant
 - État du processus
 - Compteur d'instructions
 - Registres de la CPU
 - Information sur l'ordonnancement de la CPU (priorité, pointeurs sur les files d'attentes, etc.)
 - Information de gestion de la mémoire (registres de base, limite, table des pages,)
 - Information de comptabilisation
 - Information de statut d'E/S (liste des périphériques assignées, liste des fichiers ouverts)



Bloc de control de processus [1]

Bloc de control de processus

```
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
};
```

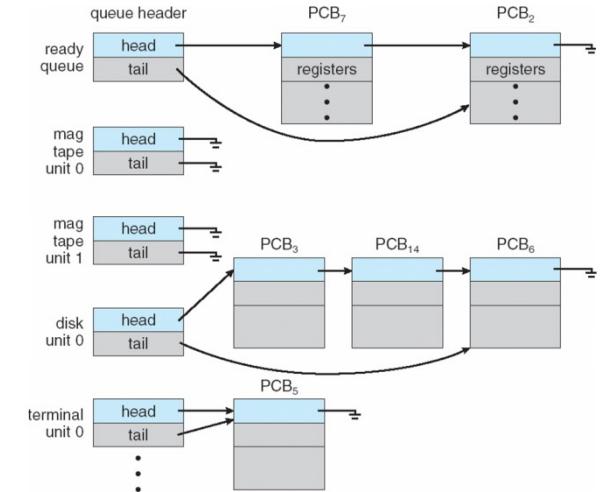
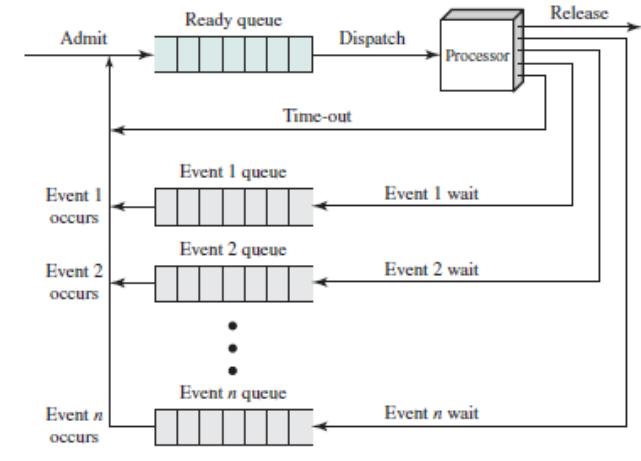
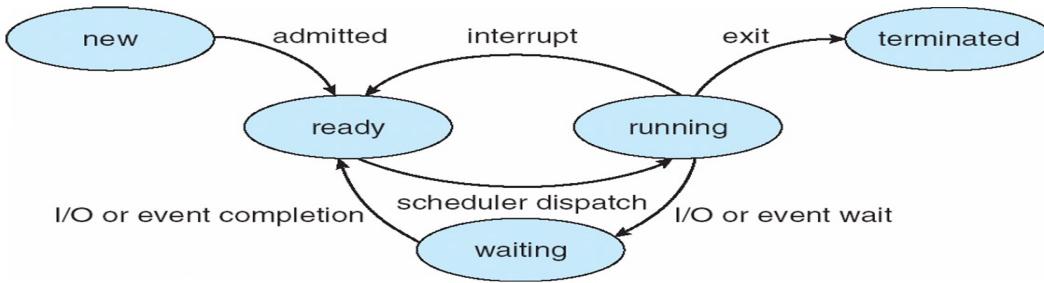
Bloc de control de processus et les états d'un processus dans XV6

Ordonnancement des processus

- Multiprogrammation
 - Avoir plusieurs programmes chargés en mémoire en même temps
 - Donc : Plusieurs processus en cours d'exécution
 - Objectif : maximiser **l'utilisation** de la CPU
- Temps partagé
 - Commuter (switcher) la CPU entre les processus très fréquemment
 - Objectif : Les utilisateurs peuvent interagir avec les programmes en exécution
- Afin d'atteindre ces objectifs, l'ordonnanceur des processus sélectionne un processus, parmi un ensemble de processus prêts, pour l'exécuter par la CPU.

Files d'attentes d'ordonnancement

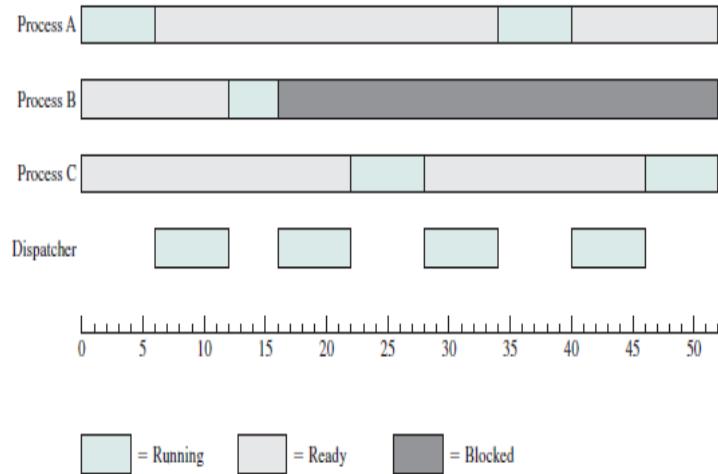
- Un OS utilise un ensemble de files d'attentes afin d'accomplir sa tâche d'ordonnancement des processus
- File d'attente des processus prêts (Ready queue)
 - Ensemble de tous les processus résidants en mémoire, prêt et en attente d'exécution
 - Liste chaînée des PCBs avec une tête pointant vers le premier PCB et le dernier PCB
 - Chaque PCB a un pointeur vers le PCB suivant.
- Liste d'attente des périphériques
 - Ensemble des processus en attente pour un périphérique d'E/S
 - Chaque périphérique a sa propre liste



Files d'attente des processus [1]

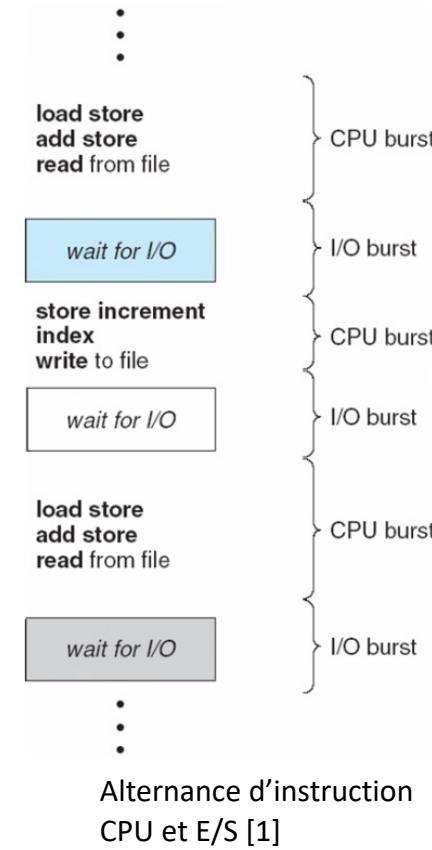
Ordonnanceurs

- Les processus passent d'une file à une autre au cours de leur exécution
- L'OS doit sélectionner des processus dans ces files d'attentes pour des fins d'ordonnancement (scheduling)
- Ordonnanceur
 - Le processus système qui réalise cette sélection
- **Ordonnanceur à long terme** (ordonnanceur de travaux)
 - Sélectionne des processus parmi un ensemble et les charge en mémoire pour les exécuter
 - S'exécute beaucoup moins fréquemment : peut être long
 - Contrôle le degré de multiprogrammation (nombre de processus en mémoire)
 - Peut prendre du temps pour décider la sélection
- **Ordonnanceur à court terme** (ordonnanceur d'exécution)
 - Sélectionne un processus parmi les processus prêts et lui alloue la CPU
 - S'exécute très fréquemment
 - un processus s'exécute pour qlqs millisecondes et ensuite attend d'E/S
 - Doit être très rapide

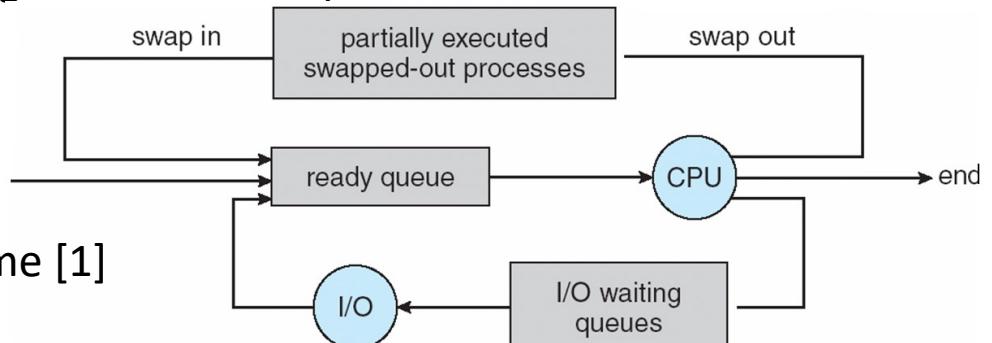


Ordonnanceurs

- Les processus peuvent être décrits comme :
- **Processus tributaire des E/S (I/O-bound process)**
 - processus qui passent la majeure partie de son temps à faire des E/S plutôt que des calculs
- **Processus tributaire de la CPU (CPU-bound process)**
 - Génère peu d'E/S et passe beaucoup de temps à faire des calculs
- **Ordonnanceur à moyen terme**
 - Dans certains systèmes (temps partagé)
 - Retire des processus de la mémoire (donc de la compétition à la CPU) et réduire ainsi le degré de la multiprogrammation
 - Les processus peuvent être rechargés en mémoire plus tard (permutation – Swapping)



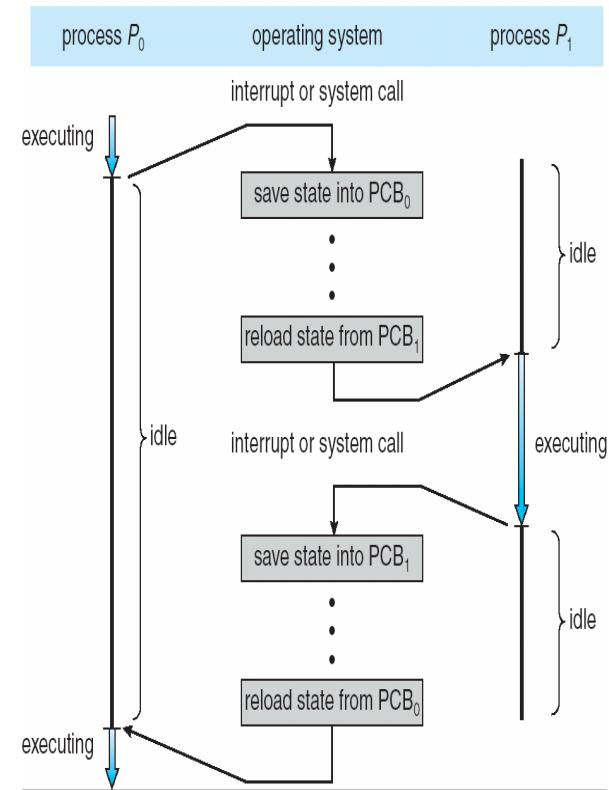
Alternance d'instruction
CPU et E/S [1]



Ordonnancement à moyen terme [1]

Commutation de Context

- Suite à une interruption, l'OS doit permuter la CPU de la tache courante à une autre routine
- Le contexte d'un processus est représenté par le PCB
- Commutation de contexte
 - Le système doit sauvegarder le contexte courant du processus en cours d'exécution par la CPU pour être restauré plus tard et charger le contexte du nouveau processus
- Le temps utilisé pour la commutation de contexte est un surcout (overhead)
 - Le système ne fait pas du travail utile durant la commutation
 - Dépend du support matériel



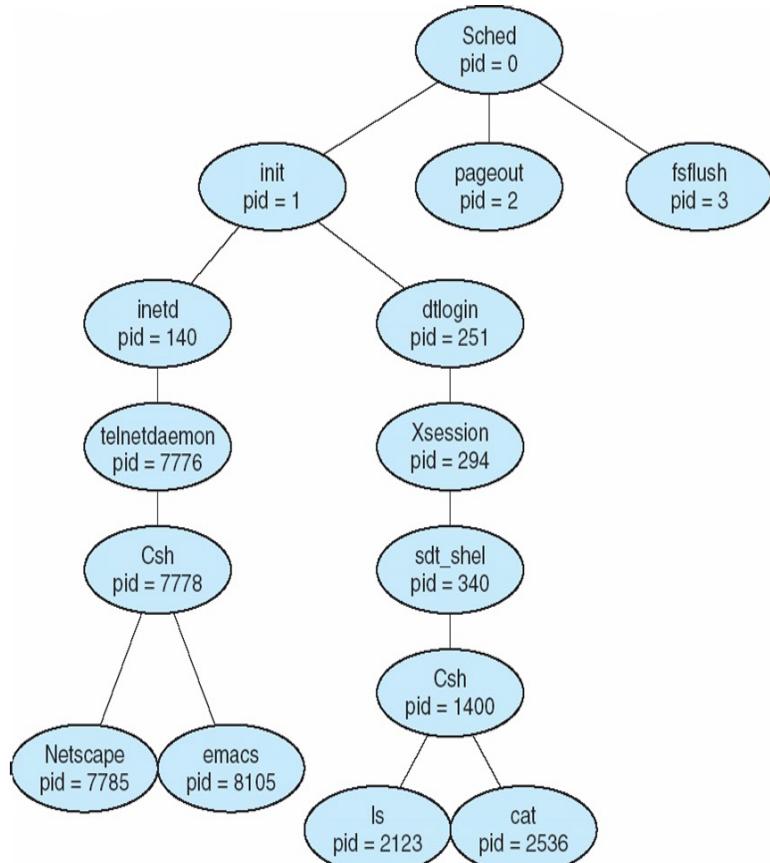
Commutation de contexte de la CPU [1]

Opérations sur les processus

- Les processus dans un système s'exécutent en concurrence (parallèle)
- Ils peuvent être créés et supprimés dynamiquement
- L'OS doit fournir un mécanisme pour la création et la terminaison des processus

Création de processus

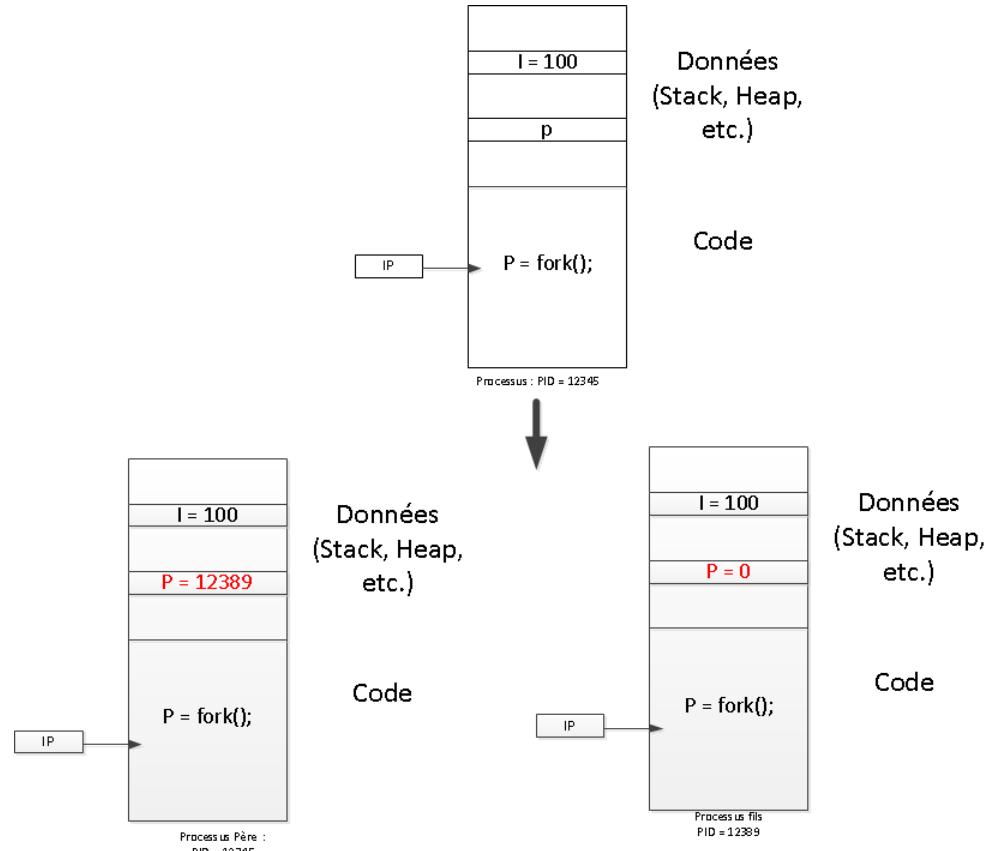
- Un processus peut créer un ou plusieurs autres processus
 - Utilise un appel système de création de processus
- Le processus créateur s'appelle le processus **père**
- Les nouveaux processus s'appellent les processus **fils**
- Chacun de ces processus peuvent créer de nouveaux processus formant un arbre de processus
- En général un processus est identifié par identificateur de processus unique (PID) : nombre entier
- Exemple :
 - Arbre de processus typique dans Solaris [2]
- Sous Unix (Linux) :
 - La commande `ps -el` donne une information complète sur tous les processus dans le système



Arbre de processus dans Solaris [1]

Création de processus

- Un nouveau processus est créé par l'appel système fork()
- Le nouveau processus consiste en **une copie de l'espace d'adressage** du processus original
 - Ceci permet au processus père de communiquer facilement avec le processus fils
- Les deux processus continuent l'exécution après fork()
- Différence : Le code de retour de fork()
 - pour le processus fils est zéro
 - Pour le processus père est l'identificateur du processus créé (PID)
- Appel system : getpid(), getppid()

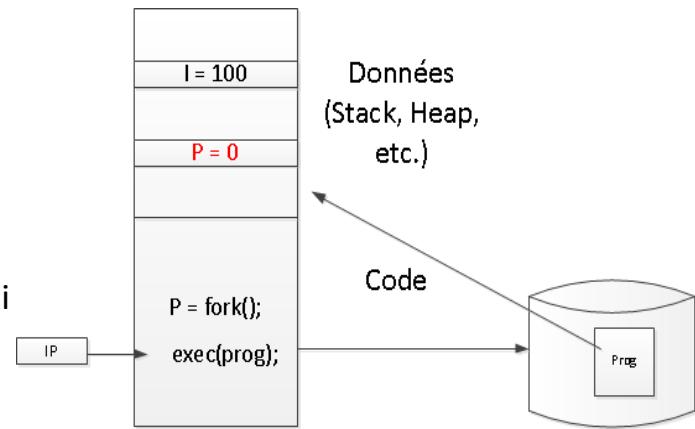


Création de processus – Exemple 01

```
1 // LOG710
2 // compiler : gcc exe02.1.c -o exe02.1
3 // executer : ./exe02.1
4
5
6 #include <sys/types.h> // pour le type pid_t
7 #include <unistd.h> // pour fork
8 #include <stdio.h> // pour perror, printf
9
10 int a=20;
11
12
13 int main ()
14 {
15     pid_t pid;
16
17     switch (pid = fork())
18     {
19         case -1:
20             /* ici pid est -1, the fork echouee */
21             /* raisons possibles: manque d'espace memoire ou le */
22             /* ou nombre maximal de creations autorisees est atteint */
23             perror("L'operation fork() a echouee!");
24             break;
25
26         case 0:// seul le processus fils execute ce "case"
27             /* pid zero est le fils */
28             printf("ici le processus fils %d, le PID de mon pere est %d.\n",getpid(),getppid());
29             a+=10;
30             break;
31
32         default:// seul le processus pere execute cette instruction
33             /* pid plus grand que zero est le parent prenant le pid du fils */
34             printf("ici processus pere %d, le PID de mon fils est %d.\n",getpid(), pid);
35             a+=100;
36     }
37
38     // les deux processus executent ce qui suit
39     printf("Fin du Process %d. avec a = %d\n", getpid(),a);
40     return 0;
41 }
```

Création de processus

- Typiquement, l'appel système `exec()` est utilisé après le `fork()` par un des processus
 - Remplacer l'espace mémoire du processus par un nouveau programme
 - `exec()` charge un nouveau fichier binaire dans la mémoire détruisant l'ancien contenu et démarre l'exécution
 - Donc, les deux processus peuvent communiquer et continuer sur des chemins séparés (exécution asynchrone)
 - Les processus père et fils continuent de partager certaines ressources (les fichiers ouverts)
 - Le processus père peut utiliser l'appel système `wait()` pour attendre la terminaison de son fils
- Le systèmes du type UNIX offrent une famille d'appels système `exec` qui permettent à un processus de remplacer son code exécutable par un autre
 - `int execl(const char *path, const char *argv,);`
 - `int execlp(const char *file, const char *argv,);`
 - `int execv(const char *path, char *const argv[]);`
 - `int execvp(const char *file, char *const argv[]);`
- Le processus conserve, notamment, son PID, et son lien de parenté.
- En cas d'échec, le processus poursuit l'exécution de son code à partir de l'instruction qui suit l'appel (il n'y a pas eu de remplacement de code).



Création de processus – Exemple 02

```
1 // LOG710
2 // compiler : gcc exe02.2.c -o exe02.2
3 // executer : ./exe02.2
4
5 #include <unistd.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <errno.h>
9 int main ()
10 {   char* arg[] = {"/bin/ps", "f", NULL};
11     printf("Bonjour..\n");
12     execvp("/bin/ps", arg);
13     printf("Echec de execvp\n");
14     printf("Erreur %s\n", strerror(errno));
15     return 0;
16 }
```

Attente de la fin d'un processus fils appel système wait

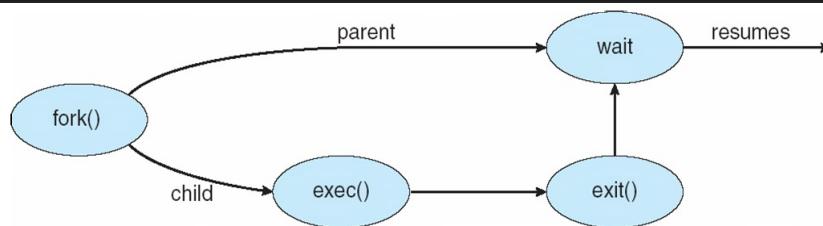
- Les appels système `wait()` et `waitpid()` permettent à un processus père d'attendre ou de vérifier la terminaison d'un de ses fils :
 - `int wait (int * status);`
 - `int waitpid(int pid, int * status, int options);`
- `wait` et `waitpid` retournent :
 - le PID du fils qui vient de se terminer en cas normal (succès)
 - -1 en cas d'erreur (exemple : le processus n'a pas de fils).
 - des informations concernant la terminaison du fils dans le paramètre **status**.
- La valeur (int) de `status` peut être récupérée et investiguée au moyen de macros telles que :
 - `WIFEXITED(status)` : retourne VRAI si le processus s'est terminé normalement avec `exit`
 - `WIFSIGNALED(status)` : tué par un signal
 - `WEXITSTATUS(status)` : retourne la valeur de retour du processus fils (`exit(valeur)`)

Attente de la fin d'un processus fils appel système wait

- `wait(status)` et `waitpid(-1,status,0)` :
 - bloquent le processus appelant jusqu'à ce que l'un (quelconque) des fils du processus se termine.
- `waitpid(pid, status,0)` avec `pid>0`:
 - bloque le processus appelant jusqu'à ce qu'un fils spécifique identifié par `pid` se termine.
- `waitpid(pid, status, WNOHANG)` :
 - vérifie seulement la terminaison, retourne 0 en cas de non terminaison (pas d'attente).

Création de processus – Exemple 03

```
1 // LOG710
2 // compiler : gcc exe02.3.c -o exe02.3
3 // executer : ./exe02.3
4
5
6 #include <stdio.h>
7 #include <unistd.h>
8 #include <sys/types.h>
9
10 int main()
11 {
12
13     pid_t pid;
14
15     /* Creer un processus fils avec fork */
16     pid = fork();
17
18     if (pid < 0) { /* En cas d'erreur */
19         fprintf(stderr, "Creation de processus avec fork echouee");
20         exit(-1);
21     }
22     else if (pid == 0) { /* Processus fils */
23         execlp("/bin/ls", "ls", NULL); // Charger l'image binaire du programme
24                                         // de la commande Unix/Linux ls
25     }
26     else { /* Processus pere */
27         /* le processus pere va attendre que le processus fils complete */
28         wait (NULL);
29
30         printf ("Processus fils a complete");
31
32         exit(0);
33     }
34 }
```



Création de processus
avec l'appel système
fork() [1]

Terminaison de processus

- Après qu'un processus exécute sa dernière instruction, il demande d'être terminé (supprimé) en utilisant l'appel système (**exit**)
 - Le processus peut retourner une valeur de statut (un entier) à son processus père (via l'appel système **wait**)
 - Les ressources du processus sont libérés par l'OS
- Un processus peut également être terminé d'une façon anormale (suite à un problème) en envoyant un signal adéquat (appel système **kill**)
- Raisons :
 - Le processus fils a dépasser son quota d'utilisation de ressources allouées
 - La tache assignée au processus n'est plus nécessaire
 - Si la parent termine et l'OS n'autorise pas un processus sans père
 - Tous les processus fils sont terminé **terminaison en cascade**
 - Sous Linux, si le processus père est terminé, les processus fils seront assigné le processus init comme père

Terminaison de processus – Exemple 04

```
1 // LOG710
2 // compiler : gcc exe02.4.c -o exe02.4
3 // executer : ./exe02.4
4 //| Avec cette version : les processus crees affichent comme pid de leur pere 1. Pourquoi ?
5
6 #include <sys/types.h>      // pour le type pid_t
7 #include <unistd.h> // pour fork
8 #include <stdio.h> // pour printf
9 int main ( )
10 {   pid_t p;
11     int i, n=5;
12     printf("Processus pere : %d\n", getpid());
13     for (i=1; i<n; i++)
14     {   p = fork();
15         if (p > 0 )
16             break;
17         sleep(1);
18         printf(" Processus %d de pere %d. \n", getpid(), getppid());
19     }
20
21
22     return 0;
23 }
```

Exercice

Exercice 01

On considère le code Listing 1. Y compris le processus parent initial, déterminer le nombre de processus qui sont créés par le programme.

```
#include <stdio.h>
#include <unistd.h>
int main() {

    /* fork a child process */
    fork();

    /* fork a child process */
    fork();

    /* fork a child process */
    fork();

    return 0;
} /* Listing 1 */
```

Exercice

Exercice 02

On considère l'instruction suivante :

```
for(int i=0; i<3; i++)
    fork();
```

On suppose que l'appel système `fork()` ne retourne pas d'erreurs. a) Quel est le nombre de processus créés par l'instruction suivante. b) Donnez l'arborescence des processus.

Exercice

Exercice 03

On considère le programme Listing 2. Quelle est la sortie du programme au niveau des lignes A, B, C et D. On suppose que le pid du processus parent est 2600 et celui du fils est 2603.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {

pid_t pid, pid1;

/* fork a child process */
pid = fork();

if (pid < 0{
    fprintf(stderr, "Fork failed");
    return 1;
}
else if(pid == 0){
    pid1 = getpid();
    printf("pid = %d", pid); /* A */
    printf("pid1 = %d", pid1); /* B */
}
else{
    pid1 = getpid();
    printf("pid = %d", pid); /* C */
    printf("pid1 = %d", pid1); /* D */
    wait(NULL);
}
return 0;
} /* Listing 2 */
```

Exercice

Exercice 04

On considère le programme Listing 3. Expliquer la sortie au niveau de la ligne A

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

int value = 5;

int main() {

pid_t pid, pid1;

/* fork a child process */
pid = fork();

if (pid == 0) {
    value += 15;
    return 0;
} else if(pid > 0) {
    wait(NULL);
    printf("value = %d", value); /* A */
    return 0;
}
} /* Listing 3 */
```

Exercice

Exercice 05

Que fait le programme suivant :

```
int main( )
{
    int p=1;
    while(p>0) p=fork();
        execvp("prog", "prog", NULL);
    return 0;
}
```

Structure général d'un shell

```
#define TRUE 1

while (TRUE) {                                /* repeat forever */
    type_prompt( );                          /* display prompt on the screen */
    read_command(command, parameters);      /* read input from terminal */

    if (fork( ) != 0) {                      /* fork off child process */
        /* Parent code. */
        waitpid(-1, &status, 0);            /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0);    /* execute command */
    }
}
```

Threads

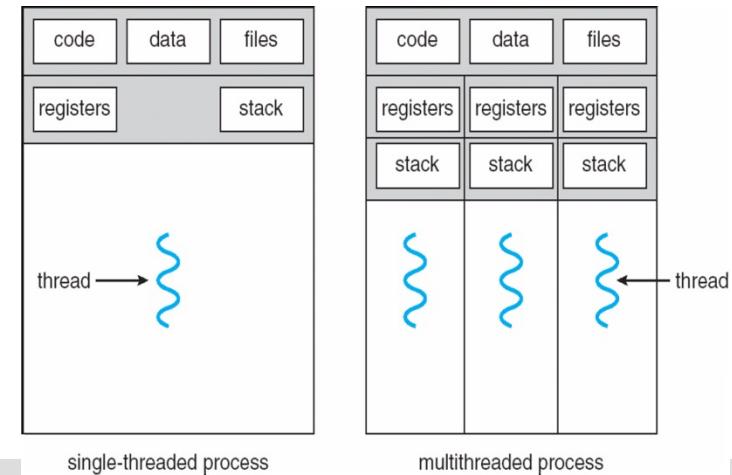
Threads

- Modèle de processus :
 - Un processus est un programme qui s'exécute selon un chemin d'exécution unique.
 - Il a un seul flot de contrôle (un seul thread).
- Les systèmes d'exploitation modernes supportent l'association à un même processus plusieurs chemins d'exécution (multithreading)

Qu'est ce qu'un thread?

- Un thread est
 - une unité de base d'utilisation de la CPU (unité d'exécution)
 - rattaché à un processus et chargé d'exécuter une partie du processus.
- Un processus peut être considéré comme un ensemble de ressources (espace d'adressage, fichiers, périphériques...) partagés par ses threads.
- Traditionnellement, un processus a un seul thread (heavyweight)
- Un processus peut avoir plusieurs threads qui font plusieurs tâches à la fois
- Lorsqu'un processus est créé, un seul thread est associé au processus.
 - Ce thread peut en créer d'autres.
- Chaque thread a :
 - un identificateur unique
 - Un compteur ordinal
 - Des registres
 - une pile d'exécution

Processus monothread vs. multithread[1]

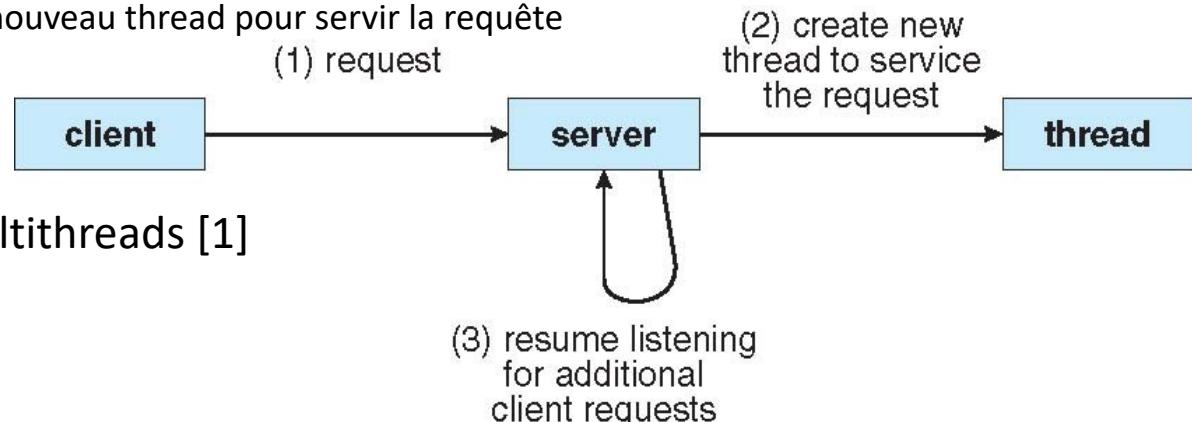


Pourquoi les threads?

- Avantages
- Réactivité (Responsiveness)
 - Une application « multithreaded » interactive peut continuer à s'exécuter même si certaines de ses parties sont bloquées
- Partage de ressources
 - Les threads partagent les ressources de leur processus par défaut
 - facilite la coopération et améliore la performance
- Économie d'espace mémoire et de temps.
 - Plus rapide de créer et terminer un thread
 - Plus rapide de commuter (switch) entre deux threads d'un même processus.

Exemple d'utilisation des threads

- Certaines applications doivent faire plusieurs tâches similaires
- Un serveur web
 - doit accepter les requêtes du client pour des ressources comme des pages web, images, son, etc.
 - Peut servir plusieurs clients en parallèle
- Si le serveur web est conçu comme processus avec un seul thread
 - Il serait capable de servir un seul client à la fois seulement (temps d'attente)
- Le serveur web peut avoir un thread qui écoute les requêtes
- Pour chaque requête, il crée un nouveau thread pour servir la requête



Architecture d'un serveur multithreads [1]

POSIX Threads

- Implémentations propriétaires de threads
 - Différences substantielles
 - Portabilité difficile.
- Posix Thread (Pthreads)
 - Interface de programmation **standard**
 - IEEE POSIX 1003.1c standard (1995)
- Dernière version IEEE Std 1003.1, 2004
- Implémentation disponible pour Unix, Linux et Windows
- Interface de programmation C :
 - un ensemble de types et appels de fonctions
 - fichier header **pthread.h**
 - une librairie indépendante ou faisant partie de **librairie C**

Pthreads : API

- The fonctions composant l'API Pthreads peuvent être groupées en:
- **Gestion de threads**
 - Exprimer la concurrence
 - fonctions qui opèrent directement sur les threads – création, détachement, jointure et opérations sur les attributs

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

- **Mécanismes de synchronisation**

Fichier à inclure et Compilation

- Afin d'utiliser la librairie Pthread, il faut inclure le fichier :

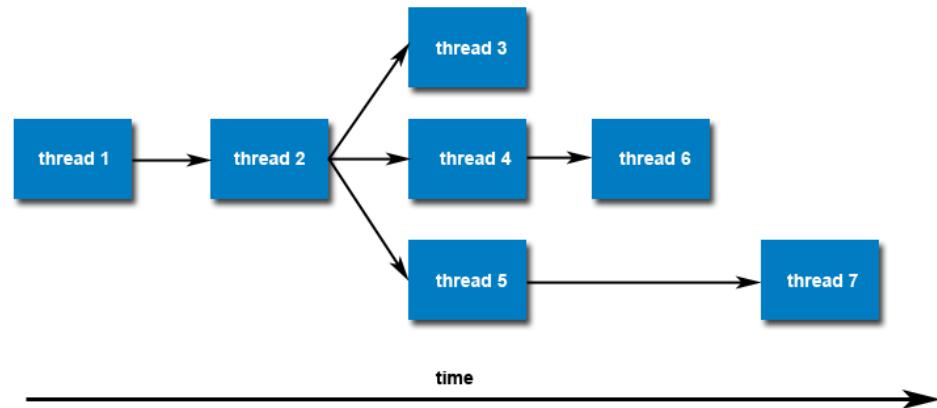
```
#include <pthread.h>
```

- Compilation et édition de liens d'un programme utilisant Pthreads :

```
gcc fichier.c -o fichier -lpthread
```

Création de threads

- Un thread est créé par défaut (automatiquement) pour exécuter la fonction main()
- Tout autre thread doit être créé explicitement
- La fonction `pthread_create()` est utilisée pour créer un nouveau thread.
- Une fois créés, les threads sont des pairs :
 - peuvent créer d'autres threads
 - Il n'y a pas d'hiérarchie ou dépendance entre eux



Création d'un Thread

```
int pthread_create(pthread_t *thandle,
                  const pthread_attr_t *tattr,
                  void *(*threadfunction) (void*),
                  void *arg)
```

- **Pthread_t** : chaque thread est du type pthread_t
- **thandle**: Identificateur (handle) du thread nouvellement créé
- **tattr**:
 - Un objet utilisé pour fixer les attributs du thread (ex. taille de la pile, priorité, etc.)
 - NULL pour les valeurs par défaut
- **threadfunction**: La fonction exécutée par le thread créé (tâche)
- **arg**:
 - argument de la fonction `threadfunction`
 - NULL si pas d'arguments
- Cette fonction retourne 0 en cas de succès, une autre valeur en cas d'échec

Terminaison d'un Thread

- Un Thread se termine de plusieurs façons:
 - Le thread retourne de la fonction appelée lors de sa création.
 - Le thread fait appel à la fonction **pthread_exit()**.
 - Le processus incluant le thread est terminé par l'appel à **exit()**.

Terminaison d'un Thread

```
void pthread_exit (void *status)
```

- Cette fonction termine le thread appelant
- Lorsque le thread principal (main) appelle cette fonction, il ne termine que lui-même, les autres Threads dans le processus continuent d'exister
 - le processus ne se termine que quand tous ses threads sont terminés
- Si le thread principal retourne du main (appelle implicitement exit()) ou appelle la fonction exit() explicitement, le processus se termine ainsi que tous les threads
- Si un thread quelconque appelle la fonction exit(), le processus se termine aussi.

Autres fonctions

- Obtenir le ID du “Thread” courant:

```
pthread_t pthread_self(void)
```

- Comparaison des Threads:

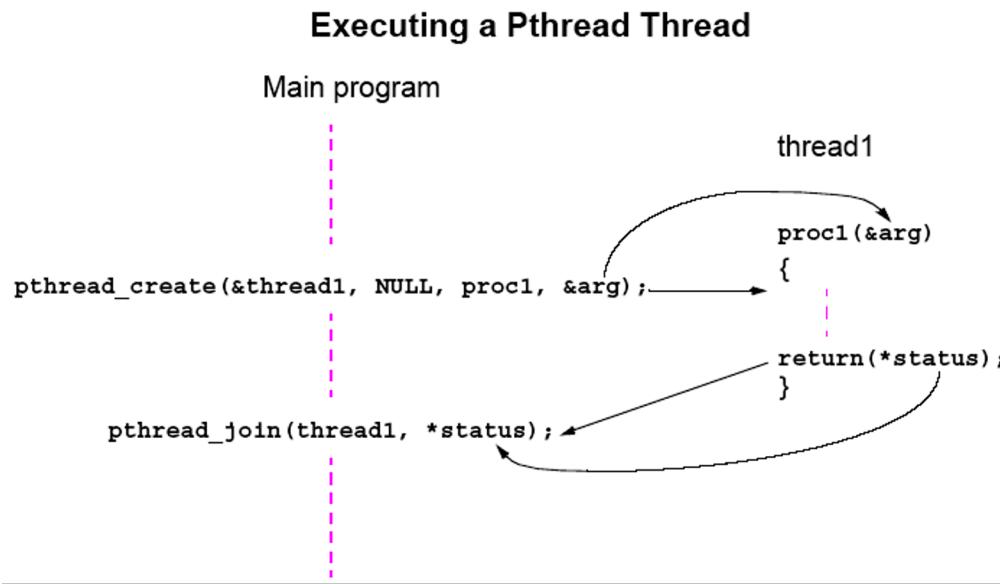
```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Cette fonction retourne une valeur non nulle si les deux threads t1 et t2 sont **identiques** (il sont le même “Thread”), si non elle retourne 0

Opération Joining des threads

- L'opération "Joining" permet à un thread d'attendre un autre (une façon de synchroniser les threads):

```
int pthread_join(pthread_t thread, void **ptr)
```



Threads POSIX – Exemple 5

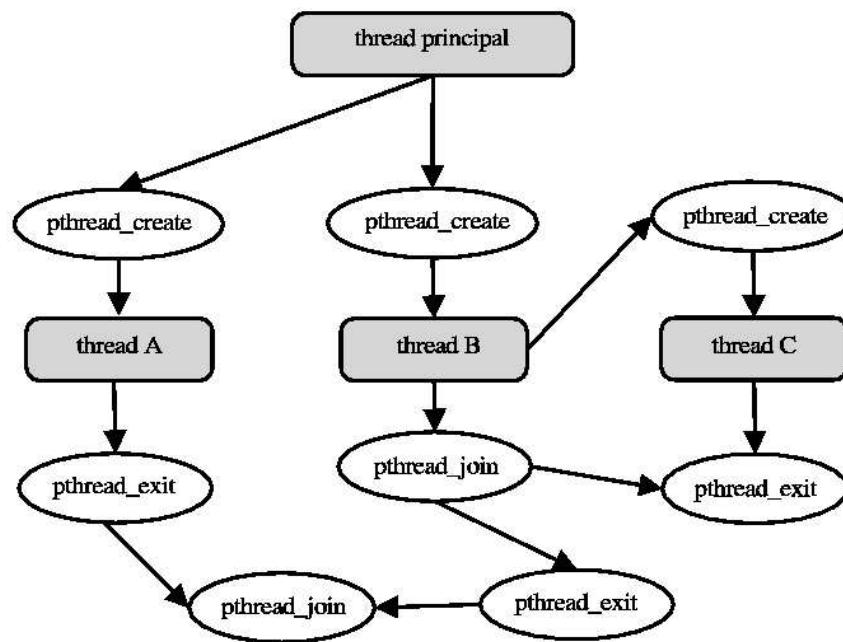
```
2 // LOG 710
3 // Exemple avec POSIX Threads (PThreads)
4 // compiler : gcc exe02.7.c -o exe02.7 -lpthread
5 // executer : ./exe02.7
6 #include <pthread.h>
7 #include <unistd.h>
8 #include <stdio.h>
9 #include <stdlib.h>
10 void afficher(int n, char lettre)
11 {
12     int i,j;
13     for (j=1; j<n; j++)
14     {
15         for (i=1; i < 10000000; i++);
16         printf("%c",lettre);
17         fflush(stdout);
18     }
19     void *threadA(void *inutilise)
20     {
21         afficher(100,'A');
22         printf("\n Fin du thread A\n");
23         fflush(stdout);
24         pthread_exit(NULL);
25     }
26     void *threadC(void *inutilise)
27     {
28         afficher(150,'C');
29         printf("\n Fin du thread C\n");
30         fflush(stdout);
31         pthread_exit(NULL);
32     }
```

Threads POSIX – Exemple 5 (Suite)

```
34 void *threadB(void *inutilise)
35 {
36     pthread_t thC;
37     pthread_create(&thC, NULL, threadC, NULL);
38     afficher(100,'B');
39     printf("\n Le thread B attend la fin du thread C\n");
40     pthread_join(thC,NULL);
41     printf("\n Fin du thread B\n");
42     fflush(stdout);
43     pthread_exit(NULL);
44 }
45
46 int main()
47 {
48     int i;
49
50     pthread_t thA, thB;
51
52     printf("Creation du thread A");
53
54     pthread_create(&thA, NULL, threadA, NULL);
55     pthread_create(&thB, NULL, threadB, NULL);
56
57     sleep(1);
58     //attendre que les threads aient termine
59     printf("Le thread principal attend que les autres se terminent\n");
60
61     pthread_join(thA,NULL);
62     pthread_join(thB,NULL);
63     printf("Fin du thread principal\n");
64
65     exit(0);
66 }
```

Poxis Threads – Exemple 1

- Graphe d'exécution



Threads POSIX – Exemple 6

```
1 // LOG 710
2 // Exemple avec POSIX Threads : variable globale
3 // compiler : gcc exe02.8.c -o exe02.8 -lpthread
4 // executer : ./exe02.8
5 #include <unistd.h> //pour sleep
6 #include <pthread.h>
7 #include <stdio.h>
8 int glob=0;
9 void* decrement(void * x)
10 {
11     int dec=1;
12     sleep(1);
13     glob = glob - dec ;
14     printf("ici decrement[%d], glob = %d\n",pthread_self(),glob);
15     pthread_exit(NULL);
16 }
17
18 void* increment (void * x)
19 {
20     int inc=2;
21     sleep(10);
22     glob = glob + inc;
23     printf("ici increment[%d], glob = %d\n",pthread_self(), glob);
24     pthread_exit(NULL);
25 }
```

Threads POSIX – Exemple 6 (Suite)

```
26 int main( )
27 {
28     pthread_t tid1, tid2;
29     printf("ici main[%d], glob = %d\n", getpid(),glob);
30     //creation d'un thread pour increment
31     if ( pthread_create(&tid1, NULL, increment, NULL) != 0)
32         return -1;
33     printf("ici main: creation du thread[%d] avec succes\n",tid1);
34     // creation d'un thread pour decrement
35     if ( pthread_create(&tid2, NULL, decrement, NULL) != 0)
36         return -1;
37     printf("ici main: creation du thread [%d] avec succes\n",tid2);
38     // attendre la fin des threads
39     pthread_join(tid1,NULL);
40     pthread_join(tid2,NULL);
41     printf("ici main : fin des threads, glob = %d \n",glob);
42     return 0;
43 }
```

Références

[1] SILBERSCHATZ, A. et P.B. GALVIN, *Operating System Concepts*. 8th Edition, Addison Wesley.

Communications Interprocessus

Abdelouahed Gherbi

LOG710 Hiver 2024

Plan

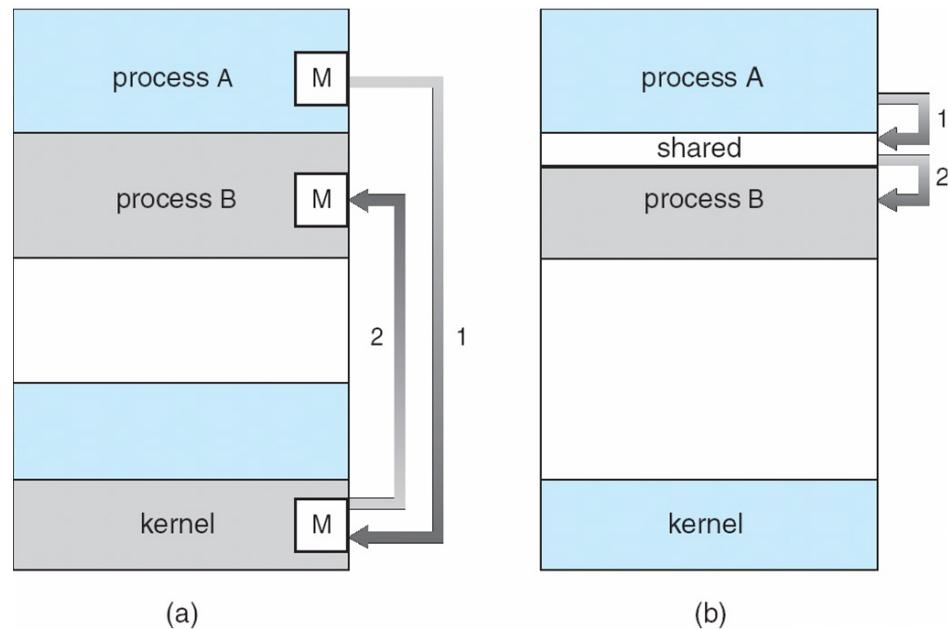
- Communication Interprocessus
- Les signaux
- Les tubes anonymes
- Les tubes nommés (FIFOs)

Communication interprocessus

- Les processus d'un système ont besoin de
 - coopérer et communiquer
 - peuvent s'influencer
 - partagent des données communes
 - collaborent pour la réalisation de tâches complexes
- Raisons pour la coopération entre processus
 - Partage d'information
 - Accélération du calcul
 - Modularité
- Les processus coopérants ont besoin de mécanismes pour permettre la communication entre eux et la synchronisation de leurs actions
- Mécanismes de communication inter processus **interprocess communication (IPC)**

Communication interprocessus

- Deux modèles de communication interprocessus
- Mémoire partagée
 - Les processus partagent une région en mémoire
 - Les processus peuvent échanger de l'information en lisant/écrivant dans la région partagée
 - Communication plus rapide que le passage de messages
 - Appels système seulement pour établir la région de mémoire commune
- Passage de messages
 - La communication a lieu via l'échange de messages entre les processus
 - Utile pour échanger de petites quantités de données
 - Plus facile à implémenter dans le cas de communication entre ordinateurs
 - Moins rapide que la mémoire partagée car typiquement basé sur les appels système



Modèles de communications [1]

Les signaux

- Un signal est une interruption logicielle asynchrone
 - Notifier le processus d'un événement
 - `kill()` : permet d'envoyer un signal à un processus
- Un processus est informé de plusieurs erreurs détectées par le matériel sous la forme signaux
- Un signal est caractérisé par un nom, un numéro, un gestionnaire (handler).
- Dans un système du type UNIX (e.g. Linux), utilisez la commande **man 7 signal** pour plus de détails
- A l'arrivée d'un signal : le noyau interrompt le processus et déclenche le gestionnaire du signal pour réagir.
- Un processus peut se mettre en attente de signaux : **pause()** et **sleep()**.

Les signaux

- Gestionnaire de signal
 - Une **fonction** enregistrée pour chaque signal et qui fait une action correspondante au signal dès qu'il arrive.
- L'OS prévoit pour chaque signal **un traitement par défaut** :
 - abort : génération d'un fichier **core** et arrêt le processus
 - exit : terminaison du processus
 - ignore : ignorer le signal
 - stop : suspension du processus
 - continue : reprendre l'exécution du processus
- Exemple
 - SIGUSR1 et SIGUSR2 : action par défaut terminer le processus.

Les signaux

(Quelques exemples de signaux – output de man 7 signal)

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

Les signaux

- Il est possible de **redéfinir** le gestionnaire de certains signaux
- Dans ce cas, un processus indique à l'OS ce qui doit se faire à la réception du signal
 - **ignorer** le signal (pas possible pour certains signaux)
 - **capturer** le signal : Exécuter un traitement spécifié par le processus dans une fonction. Le traitement des signaux se fait dans le contexte d'exécution du processus
 - **exécuter le traitement par défaut**
- Exemple :
 - La combinaison de touches **Ctrl+C** génère le signal **SIGINT**
 - Par défaut, ce signal arrête le processus
 - Un processus peut capturer ce signal en associant à ce signal un gestionnaire de signal.
- Les signaux **SIGKILL** et **SIGSTOP** ne peuvent pas être capturés, ni ignorés, ni bloqués

Les signaux

- Un processus peut **envoyer un signal** à un autre processus en utilisant l'appel système `kill`

- Détails :

```
man 2 kill
```

```
#include <sys/types.h>  
  
#include <signal.h>  
  
int kill(pid_t pid, int sig);
```

- Le signal `sig` est envoyé au processus `pid`
- Cet appel système retour : 0 en cas de succès et -1 en cas d'erreur

Les signaux

- Capture d'un signal
- Utiliser la fonction signal() :

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

- Paramètre **signum** : numéro ou le nom du signal à capturer
- Paramètre **handler** : fonction gestionnaire du signal
- **SIG_DFL** : action par défaut
- **SIG_IGN** : le processus appelant veut ignorer le signal
- Retour : adresse du gestionnaire précédent ou **SIG_ERR** en cas d'erreur

Les signaux

- Capture d'un signal
- L'appel système `sigaction()` permet de redéfinir le gestionnaire d'un signal

```
#include <signal.h>

int sigaction(int signum, const struct sigaction * act,
              struct sigaction * oldact);
```

- Cet appel système utilise une structure `sigaction` ayant pour membres des attributs qui incluent :
- On peut associer un même gestionnaire à des signaux différents

```
void(*sa_handler)(int);
```

Les signaux

- Attendre un signal
- L'appel système `pause()` suspend le processus appelant jusqu'à la réception d'un signal

```
#include <unistd.h>
int pause(void);
```

- L'appel système `sleep(v)` suspend le processus appelant jusqu'à la réception d'un signal ou l'expiration du délai (v en secondes)

```
#include <unistd.h>
void sleep(int);
```

Les signaux – Exemple #1

```
1 // LOG710 Hiver 2020 - Semaine 03 - Exemple 1
2 // compiler : gcc exe03.1.c -o exe03.1
3 // executer : ./exe03.1
4
5 #include <signal.h>
6 #include <stdio.h>
7 #include <unistd.h>
8 static void action(int sig)
9 {
10    printf("On peut maintenant m'eliminer\n");
11    signal(SIGTERM, SIG_DFL);
12 }
13
14 int main()
15 {
16    if( signal(SIGTERM, SIG_IGN) == SIG_ERR)
17        printf("Erreur de traitement du code de l'action\n");
18    if( signal(SIGUSR2, action) == SIG_ERR)
19        printf("Erreur de traitement du code de l'action\n");
20    while (1)
21        pause();
22 }
```

Les signaux - Exemple #2

```
1 // LOG710 - Semaine 03 - Exemple 2
2 // compiler : gcc exe03.2.c -o exe03.2
3 // executer : ./exe03.2
4 #include <signal.h>
5 #include <stdio.h>
6 #include <unistd.h>
7 #include <wait.h>
8 static void action(int sig)
9 {
10     switch (sig)
11     {
12         case SIGUSR1: printf("Signal SIGUSR1 recu\n");
13             break;
14         case SIGUSR2: printf("Signal SIGUSR2 recu\n");
15             break;
16         default:      break;
17     }
18 }
```

Les signaux - Exemple #2

```
19 int main()
20 {
21     struct sigaction new_action, old_action;
22     new_action.sa_handler = action;
23     sigemptyset (&new_action.sa_mask);
24     new_action.sa_flags = 0;
25
26     int i, pid, etat;
27     // Specification de l'action du signal
28
29     if( sigaction(SIGUSR1, &new_action,NULL) <0)
30         printf("Erreur de traitement du code de l'action\n");
31
32     if( sigaction(SIGUSR2, &new_action,NULL) <0)
33         printf("Erreur de traitement du code de l'action\n");
34
35     if((pid = fork()) == 0){
36         sleep(1); // Mise en attente d'un signal
37         kill(getppid(), SIGUSR1); // Envoyer signal au parent.
38         while(1);
39     }
40     else {
41         kill(pid, SIGUSR2); // Envoyer un signal à l'enfant
42         pause();
43         printf("Parent : terminaison du fils\n");
44         kill(pid, SIGTERM); // Signal de terminaison à l'enfant
45         wait(&etat); // attendre la fin de l'enfant
46         printf("Parent : fils termine\n");
47     }
48 }
```

Exercices: #1 a #3

Descripteurs de fichiers

- Pour le noyau, les fichiers ouverts sont identifiés et référencés par des **descripteurs de fichiers**.
- Un descripteur de fichiers est un nombre non négatif.

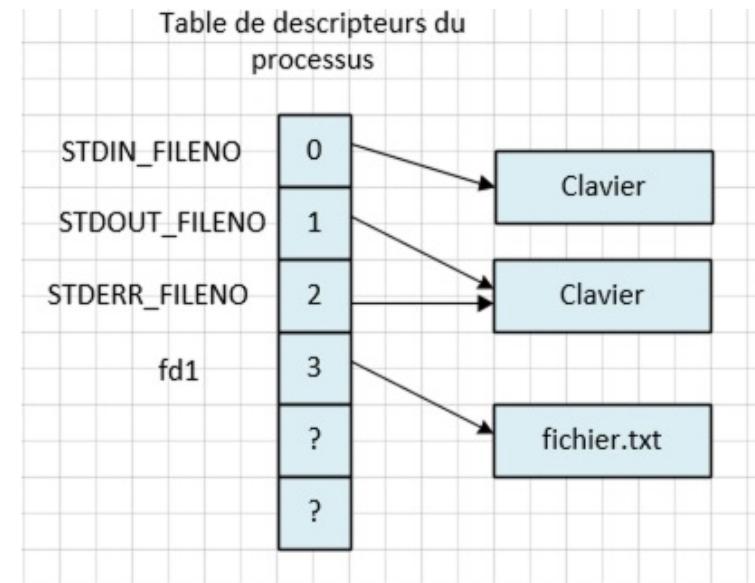
Descripteurs de fichiers

- Les descripteurs de fichiers sont utilisés pour faire référence à divers objets : les fichiers réguliers ouverts mais aussi les tubes, terminaux, et périphériques.
- Quand on crée un nouveau fichier ou on ouvre un fichier existant, le noyau retourne un descripteur de fichier au processus

Descripteurs de fichiers

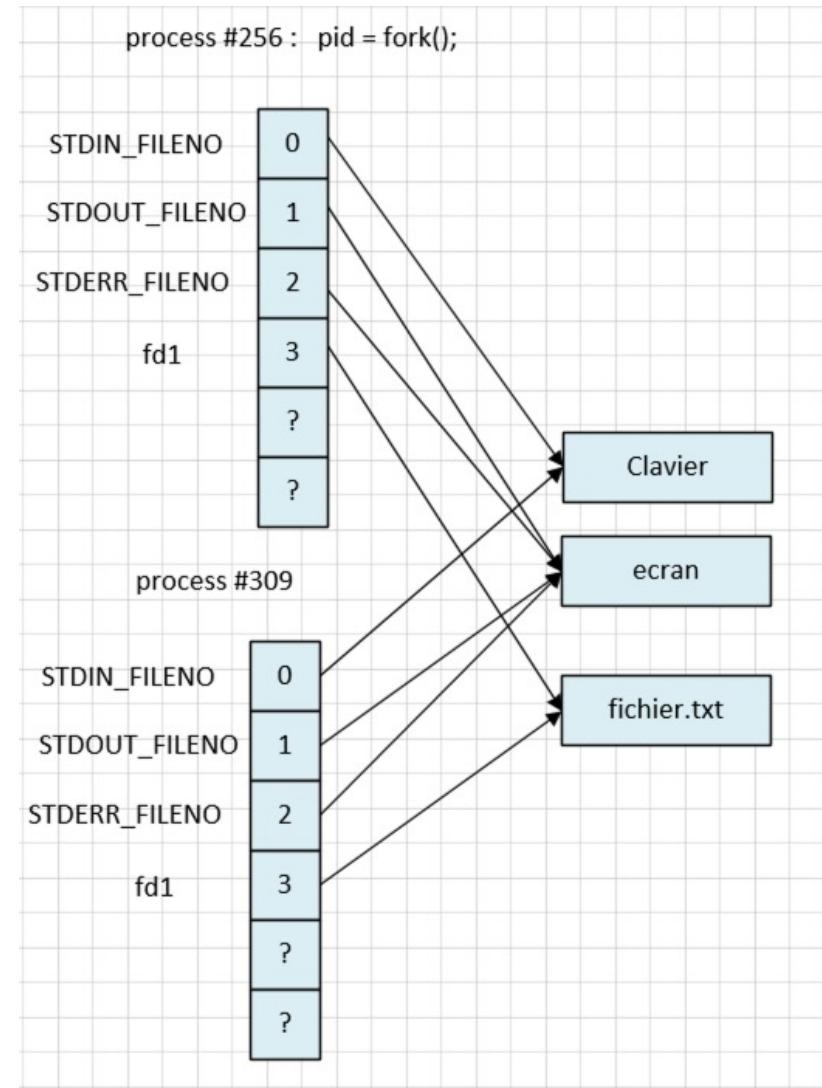
- Convention :
 - Les descripteurs 0, 1 et 2 sont réservés pour l'entrée standard (clavier), la sortie standard (écran) et la sortie d'erreur.
- Chaque processus a un ensemble de descripteurs de fichiers dans une table typiquement maintenue par le noyau.

File descriptor	Purpose	POSIX name	stdio stream
0	standard input	STDIN_FILENO	<i>stdin</i>
1	standard output	STDOUT_FILENO	<i>stdout</i>
2	standard error	STDERR_FILENO	<i>stderr</i>



Descripteurs de fichiers

- Retour sur fork() :
 - Le processus enfant créé par fork() hérite dans son espace d'adresses tous les descripteurs de fichier du processus parent.



Descripteurs de fichiers

- Les 4 appels système suivant servent à faire des entrées/sorties (IO):

```
fd = open(pathname, flags, mode);
```

- `open` ouvre un fichier et retourne un FD
- Si le fichier n'existe pas, `open` peut le créer dépendamment du setting du masque de bits `flags`
- `flags` spécifie si le fichier doit s'ouvrir en lecture ou écriture

Mode d'accès	Description
O_RDONLY	Ouvrir le fichier en lecture seulement
O_WRONLY	Ouvrir le fichier en écriture seulement
O_RDWR	Ouvrir le fichier en mode lecture/écriture
O_CREAT	Créer le fichier s'il n'existe pas

- `mode` spécifie les permissions sur le fichier si c'est une création, sinon il est ignoré

Descripteurs de fichiers

```
nbread = read(fd, buffer, count);
```

- `read` lit dans au maximum `count` bytes du fichier `fd`
- `read` retourne le nombre de bytes réellement lus.
- Si aucun bytes n'a pu être lu (fin de fichier), `read` retourne 0

Descripteurs de fichiers

```
nbwrite = write(fd, buffer, count);
```

- `write` écrit jusqu'à `count` bytes de `buffer` dans fichier `fd`.
- `write` retourne le nombre de bytes réellement écrits (peut être inférieure à `count`).

Descripteurs de fichiers

```
status = close(fd);
```

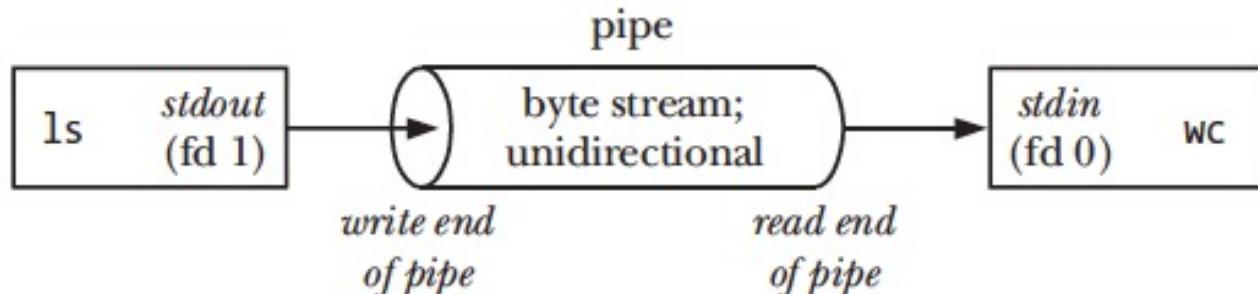
- `close` est appelé quand l'opération IO est complétée afin de libérer le fd et les ressources du noyau.

Tubes de communication (pipes)

- Parmi, les mécanismes de communication inter-processus :
 - Les tubes de communication (pipes)
- On peut distinguer deux types de tubes :
 - Les tubes anonymes ou ordinaires (named pipes),
 - Les tubes nommés (named pipes ou FIFOs)

Tubes de communication (pipes)

- On peut utiliser le shell pour executer des commandes comme :
\$ ls | wc -l
- Le shell crée deux processus qui exécutent respectivement ls et wc
- Ces deux processus utilise un tube anonyme



Tubes anonymes

- Les tubes anonymes sont des fichiers spéciaux (temporaires).
 - Utilisés avec des opérations (appels systèmes) **read** et **write**
- Ils permettent d'établir une liaison **unidirectionnelle** entre **un processus père et un processus fils**.
- Un tube est un flux d'octets (byte stream) : **pas de notion ou structure de messages**. Le processus lecteur lit des blocs de tailles quelconques.
- Les données traversent le tube **séquentiellement** : pas d'accès aléatoire.
- La tentative de lecture d'un tube vide est **bloquante** jusqu'à quand au moins un octet est écrit.

Tubes anonymes

- Quand l'extrémité d'écriture (write end) d'un tube est fermée, un processus qui lit a partir de ce tube reçoit une 'fin de fichier' (End-Of-File, EOF) - read() retourne 0.
- Plusieurs processus peuvent écrire sur le même tube. Leurs données ne seront pas mélangées si la nombre d'octets écrits par chaque process ne dépasse pas PIPE_BUF (défini dans <limits.h>)
- Un tube est un buffer maintenu par le noyau et il a une taille maximale limitée. Quand le tube est plein, tout écriture est bloquante.
- Lorsque tous les descripteurs du tube seront fermés, le tube est détruit.

Tubes Anonymes

- Un tube de communication anonyme est créé par l'appel système:

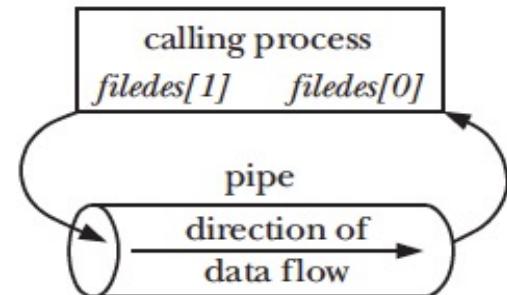
```
#include <unistd.h>  
  
int pipe(int filedes[2]);
```

- Cet appel système crée dans `p` les deux descripteurs de fichiers :

- `filedes[0]` descripteur pour la lecture à partir du tube
- `filedes[1]` descripteur pour l'écriture dans le tube.

- `pipe()` tourne
 - -1 : erreur de création du tube (manque d'espace)
 - 0 : Succès

- L'accès au tube se fait via les descripteurs de fichiers.



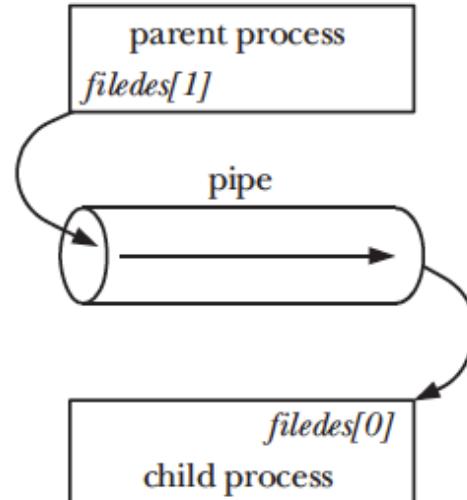
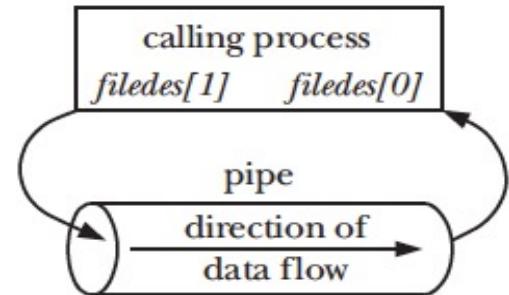
Tubes Anonymes

- En général les tubes sont utilisés dans le scénario suivant :
 - Le processus père crée un tube de communication anonyme en utilisant l'appel système

```
pipe();
```
 - Le processus père crée les fils en utilisant l'appel système

```
fork();
```
 - Le processus qui écrit (père ou fils) ferme le descripteur de fichier de la de lecture du tube
 - Le processus qui lit ferme le descripteur de fichier de l'écriture du tube
 - Les processus communiquent en utilisant les appels système:

```
read(fd[0], buffer, n);  
write(fd[1], buffer,n);
```
 - Chaque processus ferme son descripteur fichier lorsqu'il veut mettre fin à la communication via le tube.



Tubes Anonymes – Exemple #3

```
1 // LOG710 - Semaine 03 - Exemple 3
2 // compiler : gcc exe03.3.c -o exe03.3
3 // executer : ./exe03.3
4 |
5 #include <stdio.h>
6 #include <unistd.h>
7 #include <sys/types.h>
8 #include <string.h>
9
10#define BUFFER_SIZE 25
11#define READ_END    0
12#define WRITE_END   1
13
14int main(void)
15{
16    char write_msg[BUFFER_SIZE] = "Bonjour!";
17    char read_msg[BUFFER_SIZE];
18    pid_t pid;
19    int fd[2];
20
21    /** Creation d'un tube (pipe) temporaire */
22    if (pipe(fd) == -1) {
23        fprintf(stderr,"Creation du tube echouee");
24        return 1;
25    }
26
27    /** Creation d'un nouveau processus */
28    pid = fork();
29
30    if (pid < 0) {
31        fprintf(stderr, "Creation du processus fils echouee!");
32        return 1;
33    }
34
35    if (pid > 0) { /* Processus pere */
36        /* Fermer le descripteur de lecture du
37         | tube non utilisee */
38        close(fd[READ_END]);
39
40        /* Ecrire sur le tube */
41        write(fd[WRITE_END], write_msg,
42              strlen(write_msg)+1);
43
44        /* Fermeture le descripteur d'ecriture du tube
45         | (donc le tube sera supprime) */
46        close(fd[WRITE_END]);
47    }
48    else { /* Processus fils */
49        /* Fermeture du descripteur d'ecriture non
50         | utilise du tube */
51        close(fd[WRITE_END]);
52
53        /* Lire dans le tube */
54        read(fd[READ_END], read_msg, BUFFER_SIZE);
55        printf("Processus fils a lu : %s\n",read_msg);
56
57        /* Fermeture du descripteur de lecture du tube */
58        close(fd[READ_END]);
59    }
60
61    return 0;
62 }
```

Tubes Anonymes

- Duplication de descripteurs
 - Permet à un processus de créer un nouveau descripteur synonyme d'un descripteur déjà existant.

```
#include <unistd.h>

int dup(int desc);

int dup2(int desc1, int desc2);
```

- **dup** : crée et retourne un descripteur synonyme à **desc** .
- **dup2** : transforme **desc2** en un descripteur synonyme de **desc1** .
- Ces fonctions peuvent être utilisées pour réaliser **des redirections** des **fichiers d'entrées et sorties standards** vers les tubes de communication.

Tubes Anonymes – Exemple #4

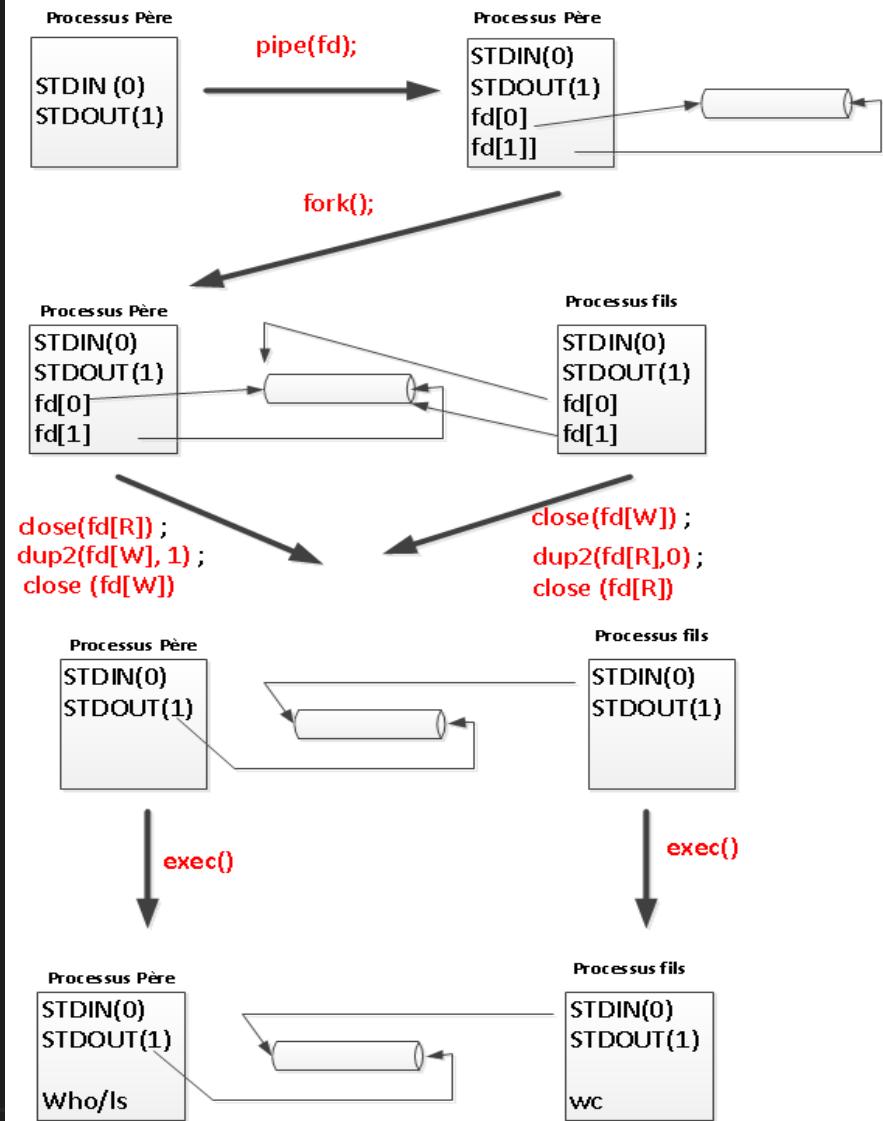
```
1 // LOG710 - Semaine 03 - Exemple 4
2 // Exemple manipulation d'un tube (pipe) temporaire (non nommés)
3 // compiler : gcc exe03.4.c -o exe03.4
4 // executer : ./exe03.4 ls  wc
5
6
7 #include <unistd.h> //pour fork, close
8 #include <stdio.h>
9
10 #define R 0
11 #define W 1
12
13 int main (int argc, char * argv [ ] )
14 {
15     int fd[2] ;
16
17     pipe(fd) ; // creation d'un tube sans nom
18
19     char message[100] ;
20     int nboctets ;
21     char * phrase = " message envoyé au pere par le fils";
22
23     if (fork() !=0) // Processus pere
24     {
25         close(fd[R]) ;
26         dup2(fd[W], 1) ;
27         close (fd[W]) ;
28     }
29     if(execlp(argv[1], argv[1], NULL) ==-1);
30         perror("error dans execlp") ;
31 }
```

```
31
32     else // processus fils (lecteur)
33     {
34         close(fd[W]) ;
35
36         dup2(fd[R],0) ;
37         close (fd[R]) ;
38
39         execlp(argv[2], argv[2], NULL) ;
40         perror("Erreur dans execlp") ;
41     }
42     return 0
43 ;
44 }
```

Tubes Anonymes – Exemple #4

```

13 int main (int argc, char * argv [ ] )
14 {
15     int fd[2] ;
16
17     pipe(fd) ; // creation d'un tube sans nom
18
19     char message[100] ;
20     int nboctets ;
21     char * phrase = " message envoyé au pere par le fils";
22
23     if (fork() !=0) // Processus pere
24     {
25         close(fd[R]) ;
26         dup2(fd[W], 1) ;
27         close (fd[W]) ;
28         if(execlp(argv[1], argv[1], NULL) ==-1);
29             perror("error dans execlp") ;
30     }
31
32     else // processus fils (lecteur)
33     {
34         close(fd[W]) ;
35
36         dup2(fd[R],0) ;
37         close (fd[R]) ;
38
39         execlp(argv[2], argv[2], NULL) ;
40         perror("Erreur dans excelp") ;
41     }
42     return 0;
43 }
```



Exercice #4

Tubes nommés

- Les tubes nommés fonctionnent aussi comme des files FIFO (first in first out).
- Ils ont un nom qui existe dans le système de fichiers comme fichiers spéciaux.
- Ils peuvent être utilisés par des processus indépendants mais sur une même machine.
- Ils sont persistants : restent dans le système de fichiers jusqu'à ce qu'ils soient supprimés explicitement
- Une capacité plus grande (max. 40K).

Tubes nommés

- Ils sont créés par:
 - la commande
 - l'appel système `mkfifo`

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

int mkfifo(char* nomfichier, mode_t mode)
```

- Exemple

```
$ mkfifo montube
$ ls -l montube
prw-rw-r-- 1 agherbi agherbi 0 Jan 22 12:04 montube
```

- Après création du tube, les processus peuvent l'utiliser pour communiquer
- Chacun des deux processus communiquant doit ouvrir le tube comme un fichier ordinaire.
- l'un en mode écriture et l'autre en mode lecture.
 - La communication via le tube est une simple opération d'écriture/lecture sur le fichier

Tubes nommés – Exemple 5

Code d'un processus Lecteur :

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{
    int fd, n;
    char message[100];

    fd = open("montube", O_RDONLY);
    printf("Processus Lecteur : [%d] \n",getpid());

    if (fd!=-1)
    {
        while ((n = read(fd,message,100))>0)
            printf("%s\n", message);
    } else
        printf( " Erreur, le tube non disponible\n");

    close(fd);
    return 0;
}
```

Code d'un processus Ecrivain

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{
    int fd;
    char message[100];
    sprintf(message, "Message du processus ecrivain : [%d]",
            getpid());

    fd = open("montube", O_WRONLY);

    printf("ici writer[%d] \n", getpid());
    if (fd!=-1)
    {
        write(fd, message, strlen(message)+1);
    } else
        printf( " Erreur : Tube non disponible \n");

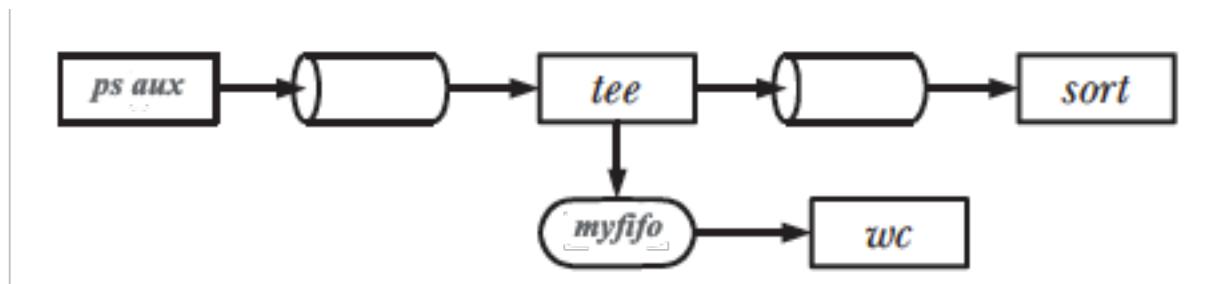
    close(fd);
    return 0;
}
```

Tubes nommés

- Exemple avec la commande `tee` :
- `tee` lit à partir l'entrée standard et copie cet input dans la sortie standard et un autre fichier désigné.

```
$ mkfifo myfifo  
$ wc -l < myfifo &  
$ ps aux | tee myfifo | sort
```

- Exemple



Références

[1] SILBERSCHATZ, A. et P.B. GALVIN, *Operating System Concepts*. 8th Edition, Addison Wesley.