Université de Bretagne Occidentale

Systèmes d'Exploitation pour l'Embarqué

Cours 2 : Services des systèmes d'exploitation

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Hai Nam TRAN
UBO – M2 LSE

Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

Question du cours

Quels sont les services fournis par un OS ?

Comment les utiliser ?

Services des systèmes d’exploitation

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Gestion des fichiers

- **Système de gestion de fichiers (SGF)**
  - **La partie la plus visible d'un système d'exploitation**
  - **Gérer le stockage et la manipulation de fichiers**
    - Gérer les fichiers
    - Offrir les primitives pour manipuler ces fichiers
- **Cas d'étude : systèmes de fichiers Linux (TP1)**

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Gestion des tâches, processus, threads

- **Processus : l'activité qui exécute un programme incluant**
  - Le code source
  - Les données
  - L'état du processeur
  - Chaque processus a son propre espace d'adressage
- **Gestion de la création, suppression, changement de priorité, contraintes temporelles, besoins mémoire, etc.**
- **Les « petits » OS pour l'embarqué utilisent seulement les threads (exécutifs) alors que les « gros » OS peuvent utiliser plusieurs modèles processus/threads**
  - RTEMS : 1 processus - plusieurs threads
  - Linux embarqué : processus/threads

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Gestion des tâches, processus, threads

- **Création de processus et de threads**
  - **Statique : toutes les tâches sont connues à l'avance → il n'est pas possible d'en créer pendant que le système tourne.**
  - **Dynamique : appels système permettant de créer et détruire des tâches à la volée**
    - Système plus flexible
    - Plus de complexité (allocation dynamique, gestion d'erreurs)
    - *Faire attentions : le temps utilisé pour gérer/ordonnancer les tâches est le temps perdu*

Ordonnancement

- **Entité qui décide quelle tâche doit exécuter le processeur**
  - **Compromis entre la prédictibilité temps réel, complexité d'implémentation, et délai d'exécution**
- **RTOS (Real Time OS) supportent plusieurs politiques d'ordonnancement**
  - **FIFO avec priorité (statique)**
  - **EDF, LLF (priorité dynamique) → performant (uniprocesseur), couteux en temps de calcul**
  - **Sporadic server (perte de priorité en fonction du temps processeur consommé)**
  - **Plus d'information : M1-STR (Frank Singhoff)**
    - http://beru.univ-brest.fr/~singhoff/ENS/UE_temps_reel/

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Communication & Synchronisation Inter Processus

- **Sémaphores: (Dijkstra 1965)**
  - Synchro à travers 2 opérations atomiques P et V.
  - Bas niveau
  - Exclusion mutuelle assurée par le programmeur
- **Moniteurs (Hoare & Hansen 1974)**
  - Mécanisme de haut niveau
  - Exclusion mutuelle assurée par le compilateur
- **Passage de messages**
  - Transfert de données entre processus
  - Mise en tampon des messages
  - Rendez-vous

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Gestion d’interruption

Interrupt definition: a signal emitted by hardware or software when a process or an event needs immediate attention

- **Example**
  - **Hardware interrupt: mouse clicks, keyboard presses, print a document**
  - **Software interrupt: I/O requests, OS service requests (fork(), execl(),…), program errors (fault, abort)**
- **Why interrupts are needed**
  - **Multi-tasks**
  - **Communication purpose**

Gestion d'interruption

- Gestion de plusieurs périphériques: minuterie (timer), moteurs, capteurs, disques, etc
- Requêtes asynchrones signalées par des interruptions
- 2 types d'interruptions
  - Interruptions matérielles
  - Interruptions logicielles
- Le code exécuté lors d'une interruption est dicté par le CPU à l'aide du vecteur d'interruption. Mais l'OS intervient pour
  - Connecter une adresse mémoire à chaque ligne d'interruption
  - Que faut-il faire après avoir servi une interruption
- Gestion de l'aspect temps réel

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Gestion de la mémoire

- **Allocation**
  - Allouer à chaque tâche la mémoire dont elle a besoin
- **Mapping**
  - Faire la correspondance entre la mémoire physique et l'adressage utilisé par les tâches
- **Protection**
  - Etablir un ensemble de comportements à adopter lorsqu'une tâche utilise de la mémoire non allouée
- **Implémentation des mécanismes permettant cette gestion**

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Support réseau

- **Le standard POSIX → socket**
  - **Accès uniforme à n'importe quel mode/protocole de communication en réseau (domaine de communication + type de socket)**
- **Support réseau spécifique à un OS particulier**
  - **Plus de fonctionnalités (création de filtre de msg / spécification de l'ordre de lecture des msg)**
  - **Moins de réutilisabilité**

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Autres fonctions

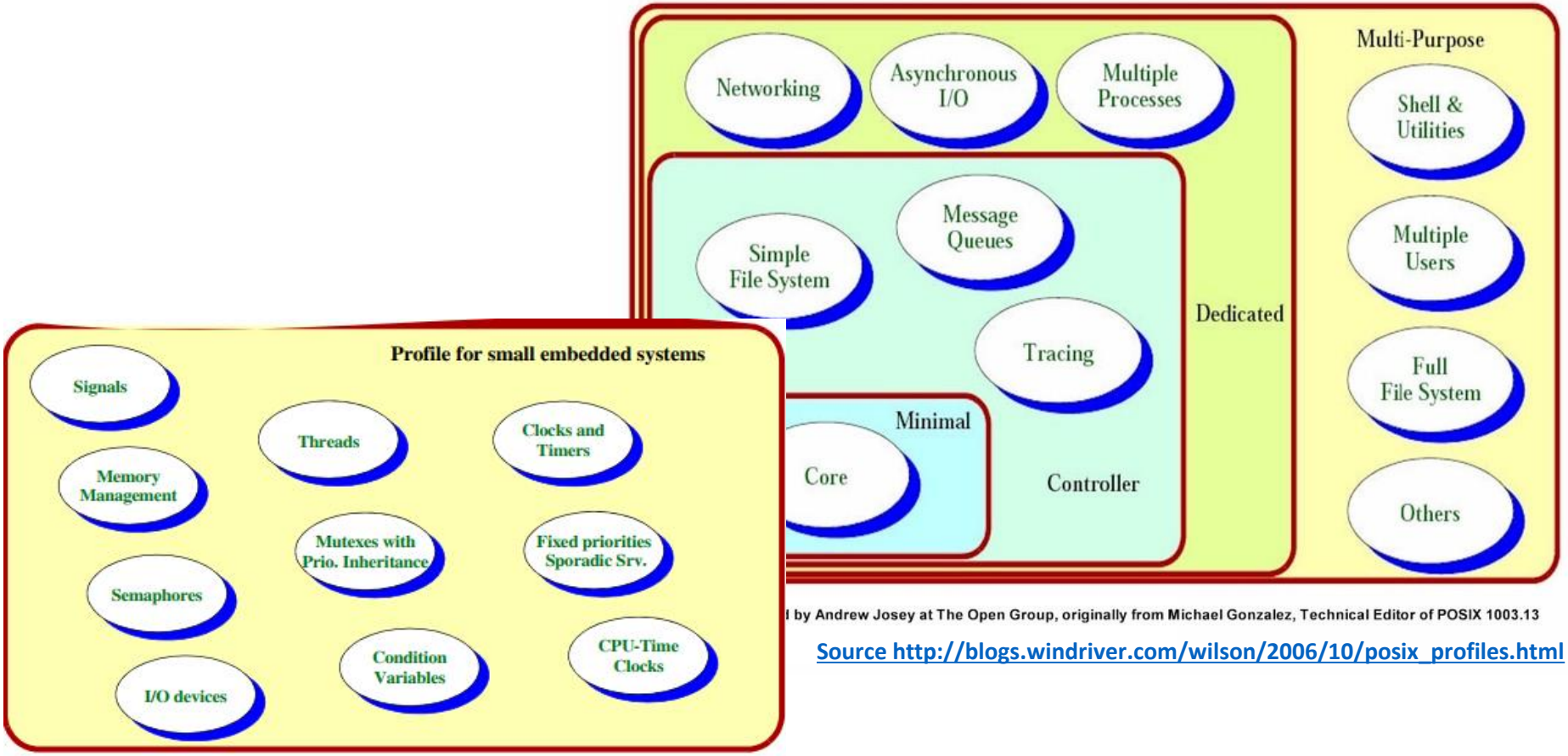
- **Signaux temps réel et asynchrones**
  - **Gestion des événements imprévus (pannes sw ou hw) et dégradation des performances en cas de surcharge du processeur**
- **Horloge et minuterie (timer) haute résolution**
  - **Donner au processus temps réel une mesure juste du temps écoulé**

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Les normes des systèmes d'exploitation

- **POSIX (Portable Operating System Interface)**
  - **Standard pour les appels de fonction (API) pour les OS UNIX-like. Il existe quelques spécifications pour des primitives temps réels**
  - **Plusieurs profils pour le temps réel**
    - PSE51: profile de système temps réel minimaliste : 1 seul processus POSIX pouvant exécuter plusieurs threads POSIX pouvant utiliser le passage de messages POSIX pour communiquer avec d'autres systèmes PS5x
    - PSE52: profile de système de contrôleur temps réel: PSE51+support pour un système de fichiers + E/S asynchrones
    - PSE53: profile de système temps réel dédié: PSE51+support multiprocessus(+MMU)
    - PSE54: profile de système temps réel polyvalent: englobe les autres profils. Il consiste de POSIX.1, POSIX.1b, POSIX.1c, et/ou POSIX.5b
  - **Exemple: RTLinux se réclame du profil PSE51**

Les normes des systèmes d'exploitation



l by Andrew Josey at The Open Group, originally from Michael Gonzalez, Technical Editor of POSIX 1003.13

Source http://blogs.windriver.com/wilson/2006/10/posix_profiles.html

Figure 3. Main services included in the minimal real-time system profile (PSE51)

Les normes des systèmes d'exploitation

- **Autres normes**
  - **UNIX98: normalisation de l'OS UNIX. Cette norme incorpore plusieurs des normes de POSIX**
  - **EL/IX: API pour les systèmes embarqués. Se veut un sous ensemble des normes POSIX et ANSI.**
  - **TRON: norme japonaise pour les systèmes embarqués**
  - **OSEK/VDX: norme allemande pour une architecture ouverte reliant les divers contrôleurs électroniques d'un véhicule.**
  - **RT Spec pour Java: spécification pour un runtime qui édicte des prescriptions (ramasse miettes, certaines politiques d'ordonnancement, etc.)**
  - **RT Corba: un ensemble de spécification temps réel**

Université de Bretagne Occidentale

Systèmes de gestion de fichiers (SGF)

Cours 2 : Services des systèmes d'exploitation

Hai Nam TRAN
UBO – M2 LSE

1. **Gestion des fichiers**
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

Introduction

- **Système de gestion de fichiers (SGF)**
  - **La partie la plus visible d'un système d'exploitation**
  - **Gérer le stockage et la manipulation de fichiers**
    - Gérer les fichiers
    - Offrir les primitives pour manipuler ces fichiers
- **Etude de cas : Systèmes de fichiers Linux**
  - **Catégories de fichiers**
    - Fichiers normaux (-) : texte, exécutable

```
-rwxrw-r-- 1 etudiant 2LR 34568 avril ...
```

    - Fichiers répertoires (d)

```
drwxr-x--- 1 etudiant 2LR 13242 avril ...
```

    - Fichiers spéciaux (/dev)
    - Fichiers liens symboliques (l)

```
lrwxrwxrwx 1 root root 14 Aug ...
```

Systèmes de fichiers Linux

- **Sous un système UNIX, un fichier quel que soit son type est identifié par un numéro appelé numéro d'i-node**
  - Derrière la façade du shell, un répertoire n'est qu'un fichier, identifié aussi par un i-node, contenant une liste d'i-node représentant chacun un fichier
- **Pour connaître le numéro d'inode d'un fichier, on utilise la commande**

```
$: ls -i mon_fichier
```

- **Parcourir et lister les répertoires**
  - ls, cd, pwd, mkdir, rmdir, mv
- **Commandes de gestion des fichiers**
  - touch, more, rm, mv, cp, file
- **Créer des liens**
  - liens **durs** : `$: ln`
  - liens **symboliques** : `$: ln -s`

Systèmes de fichiers Linux

- **Liens durs**
  - Indépendants au niveau gestion
  - Toute modification de l'un, modifie l'autre
  - La suppression de l'un, casse le lien, mais ne supprime pas physiquement l'autre
- **Liens symboliques**
  - La suppression du fichier source entraînera un changement de comportement du fichier lien qui sera considéré comme "cassé"

Systèmes de fichiers Linux

- **Monter un système de fichiers**
  - **Le système de fichiers Linux se concentre dans une seule arborescence de fichiers → l'accès et l'utilisation de systèmes extérieurs (disques, disquettes, CD..) doit s'effectuer par intégration de ces systèmes de fichiers dans le système fondamental "racine"**

```
$: mount
```

```
fichier /etc/fstab
```

  - Plus d'information :

```
$: man mount
```

```
$: man fstab
```

La hiérarchie des répertoires sous Linux

- **Dans un système Linux, les fichiers sont organisés selon une arborescence bien précise**
- **Filesystem Hierarchy Standard (FHS)**
  - **Première version : 14 février 1994**
  - **Définit l'arborescence et le contenu des principaux répertoires des systèmes de fichiers des systèmes d'exploitation GNU/Linux et de la plupart des systèmes Unix**
  - **La vaste majorité des distributions GNU/Linux ne respectent pas strictement le standard**
    - L'existence du répertoire /run dans Fedora, Debian, Ubuntu
- **Quelques exemples**
  - **/bin : contient les programmes (commandes) ou binaires qui peuvent être utilisés par l'administrateur ou les utilisateurs**
  - **/boot : contient tous les fichiers utiles pour l'exécution du chargeur (ou boot loader)**

La hiérarchie des répertoires sous Linux

- **/etc : contient les fichiers de configuration dont les divers programmes se servent pour avoir des informations sur la configuration du système**
- **/dev contient des fichiers spéciaux ou les fichiers périphériques qui sont des points d'entrée vers des périphériques physiques**
- **/home : contient les données propres à chaque utilisateur**
- **/opt : contient les logiciels optionnels**
- **/lib : contient les bibliothèques partagées nécessaires au démarrage et à l'exécution des commandes sur le système de fichiers racine**

La hiérarchie des répertoires sous Linux

- **/media : contient des répertoires utilisés comme points de montage pour les périphériques tels qu'un CDROM, clé USB ou autre**
- **/mnt : est le point de montage d'un système de fichiers temporaire**
- **/root : le répertoire « home » du superutilisateur**
- **/var : contient les fichiers de données variables. Ceci inclut les répertoires de spool, fichiers temporaires, données de connexion, etc.**
- **Plus d'information : FHS version 3.0 (50 pages)**
  - https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf

Le système de fichiers virtuel « /proc »

- **/proc**
  - **Un mécanisme qui donne accès via une arborescence de fichiers aux informations internes du noyau et en permet la modification pendant son exécution**
    - **/proc/cpuinfo** - informations sur le CPU (modèle, famille, taille du cache, etc.)
    - **/proc/meminfo** - informations concernant la RAM physique, l'espace réservé pour le « Swap » etc.
    - **/proc/mounts** - liste des systèmes de fichiers montés
    - **/proc/devices** - liste des périphériques disponibles
    - **/proc/filesystems** - les systèmes de fichiers supportés
    - **/proc/modules** - liste des modules activés
    - **/proc/version** - version du noyau
    - **/proc/cmdline** - les paramètres passés au noyau lors de la mise en route

Le système de fichiers virtuel « /proc »

```
int main(char *argc, char *argv[]) {

    const char fichier[14] = "/proc/cpuinfo";
    const char mode[11] = "model name";
    char line[256];
    char * ret;
    FILE * file;

    // Fichier CpuINfo
    file = fopen(fichier, "r");
    while (fgets(line, sizeof(line), file)) {
        char * ret = strstr(line, mode);
        if (ret != NULL)
            printf("%s", line);
        }
    fclose(file);

    return 0;
}
```

```
char *strstr(const char *haystack, const char *needle);
```

The **strstr**() function finds the first occurrence of the **substring *needle*** in the string ***haystack***. Return a pointer to the beginning of the **substring**

```
hntran@Darnassus:~/UBO/9_DILS9SEE/Tutorial$ ./getCPUInfo
model name      : Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
model name      : Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
model name      : Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
model name      : Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
model name      : Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
model name      : Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
model name      : Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
model name      : Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
```

Université de Bretagne Occidentale

Gestion des tâches, processus, threads

Cours 2 : Services des systèmes d'exploitation

Hai Nam TRAN
UBO – M2 LSE

Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

Question du cours

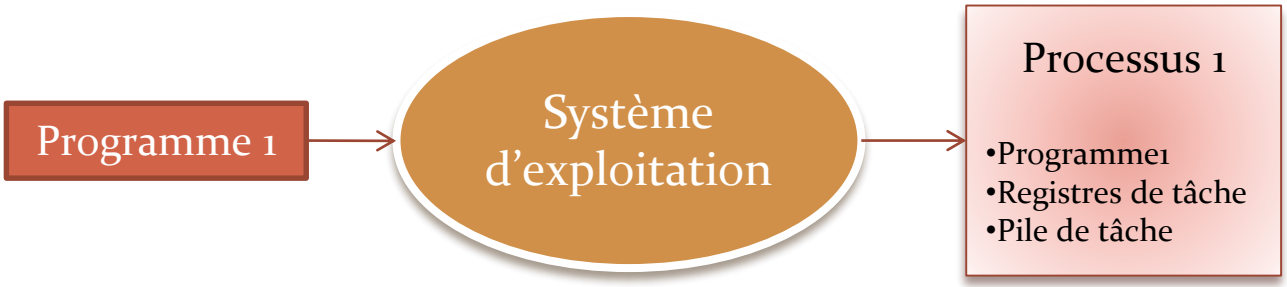
Comment les applications sont-elles gérées par un OS ?

Plan

1. **Processus et multitâches**
2. Création de tâche dans les OS embarqués
3. Suppression de tâche dans les OS embarqués
4. Gestion de processus dans Linux 2.6.x
   - Etat de processus
   - Identification
   - *wait queues*
   - Limite des ressources d'un processus
   - Changement de processus
   - Création de processus
   - Terminaison de processus

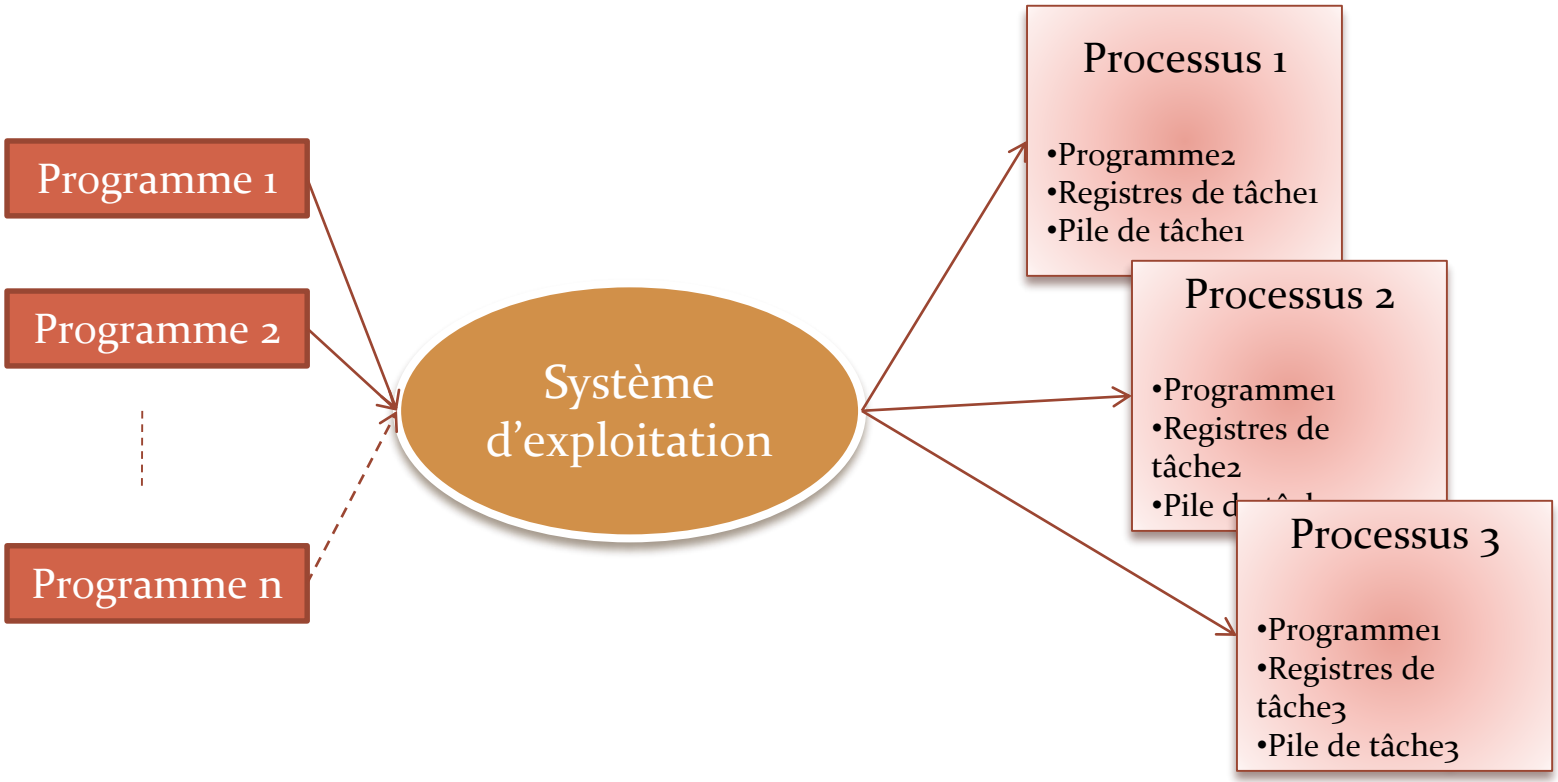
Processus & programme

- **Processus / Tâche : actif & dynamique**
- **Programme : passif & statique**



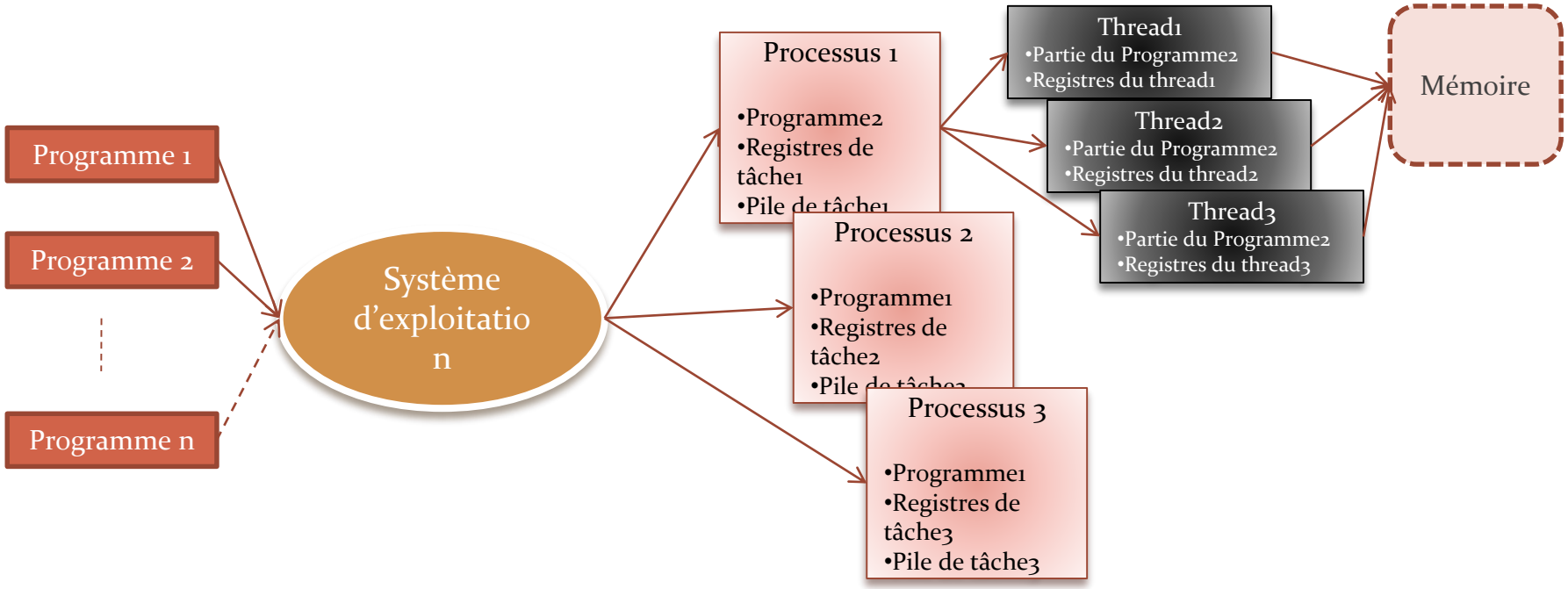
Mono tâche / Multitâche

- **Mono tâche**
- **Multitâches: mono thread / multithread**



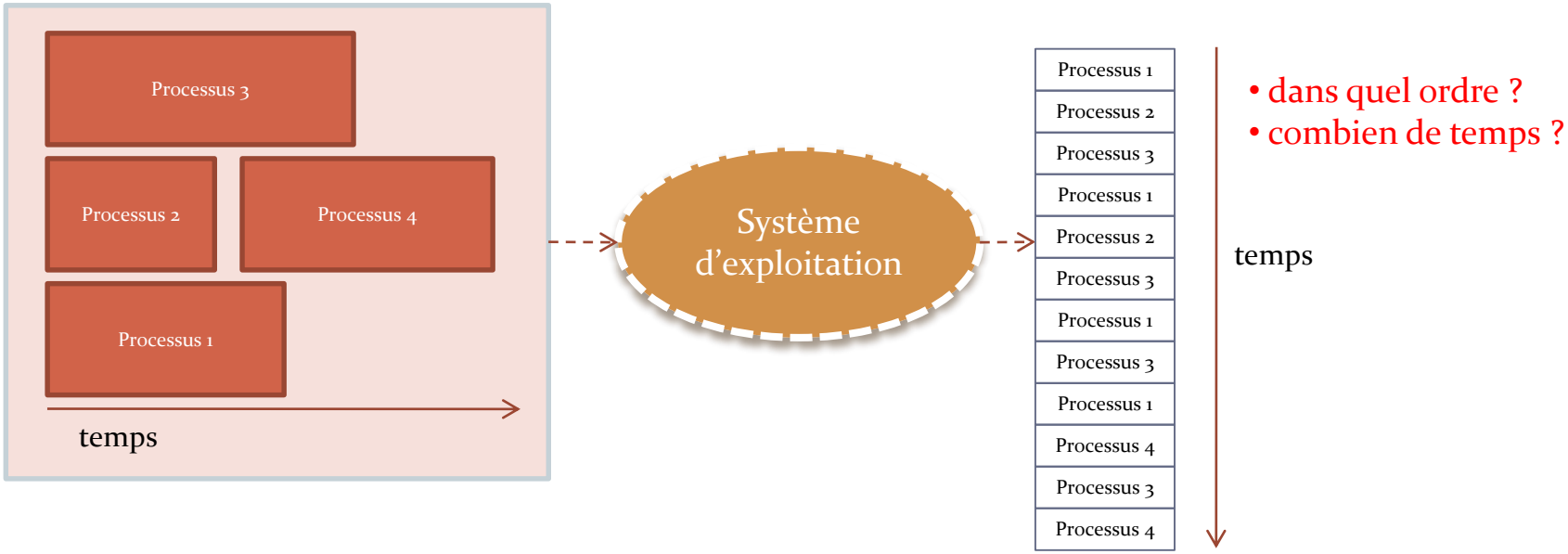
Threads (OU processus léger OU fil d'exécution)

- **Les threads d'un processus :**
  - **Partagent : le même répertoire de travail, fichiers, dispositifs d'E/S, données globales, espace d'adressage, code du programme, etc.**
  - **Ne partagent pas : le compteur de programme, la pile, les informations d'ordonnancement, etc.**



Multi-tâche et gestion des processus

- **Illusion de multi-tâche → implémentation, ordonnancement, synchronisation, communication entre tâches.**



Plan

1. **Processus et multitâches**
2. **Création de tâche dans les OS embarqués**
3. **Suppression de tâche dans les OS embarqués**
4. **Gestion de processus dans Linux 2.6.x**
   - **Etat de processus**
   - **Identification**
   - ***wait queues***
   - **Limite des ressources d'un processus**
   - **Changement de processus**
   - **Création de processus**
   - **Terminaison de processus**

Création de tâche dans les OS embarqués

- **2 modèles**
  - **fork/exec : dérivé du standard IEEE/ISO POSIX 1003.1**
    - Espace mémoire du père hérité/copié (ex: code du programme et variables
    - Héritage direct, flexibilité (changement des caractéristiques du processus)
  - **spawn : dérivé du fork/exec**
    - Création d'un nouvel espace d'adressage
    - Pas de duplication/destruction de l'espace d'adressage du nouveau processus
- **Création de processus**
  - **Fonction de création : OS**
    - Création du PCB (Process Control Block) ou TCB (Task CB) : ID, état, priorité, statut d'erreur, contexte du CPU
- **Différence entre fork/exec et spawn: allocation mémoire**

Exemple (Embedded) Linux

- int fork (void) .....voi

```
#include <sys/types.
#include <unistd.h>
int main(char *argc,
{
   processId child_processId;
   /* dupliquer le prcessus: processus fils*/
   child_processId = fork();
   if (child_processId == -1) {
     ERROR;
   }
   else if (child_processId == 0) {
     run_childProcessWork();
   }
   else {
     run_parentProcessWork();
}
```

man fork

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

Exemple (Embedded) Linux

- **Cas d'utilisation d'un exec**

```
int main(char* program, char** arg_list){
   processed child_processId;
   /* Duplication du processus*/
   child_processId = fork ();
   if (child_pId != 0)
     /* This is the parent process */
     return child_processId;
   else{
     /* Executer PROGRAM en le recherchant dans le PATH*/
     execvp (program, arg_list);
     /* execvp ne retourne qu'en cas d'erreur*/
     fprintf (stderr, « Error in execvp\n »);
     abort ();
   }
}
```

Plan

1. **Processus et multitâches**
2. **Création de tâche dans les OS embarqués**
3. **Suppression de tâche dans les OS embarqués**
4. **Gestion de processus dans Linux 2.6.x**
   - **Etat de processus**
   - **Identification**
   - ***wait queues***
   - **Limite des ressources d'un processus**
   - **Changement de processus**
   - **Création de processus**
   - **Terminaison de processus**

Suppression de tâche

- **Plusieurs raisons**
  - **Terminaison normale**
  - **Problème matériel : manque de mémoire, etc.**
  - **Problème logiciel : instruction invalide, etc.**
- **Suppression d'une tâche**
  - **L'OS libère toute la mémoire allouée: PCB, variables, code, etc.**
  - **Si parent supprimé**
    - Les fils sont **adoptés** par un autre processus
    - OU **supprimés** eux aussi avec libération des ressources partagées.

Plan

1. **Processus et multitâches**
2. **Création de tâche dans les OS embarqués**
3. **Suppression de tâche dans les OS embarqués**
4. **Gestion de processus dans Linux 2.6.x**
   - Création de processus
   - Etat de processus
   - Identification
   - *wait queues*
   - Limite des ressources d'un processus
   - Changement de processus
   - Terminaison de processus

Structure d'un Process Control Block (PCB)

```
struct task_struct //~350 lignes de code
{
....
// -1 unrunnable, 0 runnable, >0 stopped
volatile long state;
// number of clock ticks left to run in this scheduling slice,
        decremented by a timer.
long counter;
// the process' static priority, only changed through well-
        known system calls like nice, POSIX.1b
// sched_setparam, or 4.4BSD/SVR4 setpriority.
long priority;
unsigned long signal;
// bitmap of masked signals
unsigned long blocked;
// per process flags, defined below
unsigned long flags;
int errno;
// hardware debugging registers
long debugreg[8];
struct exec_domain *exec_domain;
struct linux_binfmt *binfmt;
struct task_struct *next_task, *prev_task;
struct task_struct *next_run, *prev_run;
unsigned long saved_kernel_stack;
unsigned long kernel_stack_page;
int exit_code, exit_signal;
unsigned long personality;
int dumpable:1;
int did_exec:1;
int pid;
int pgrp;
int tty_old_pgrp;
int session;
// boolean value for session group leader
int leader;
int groups[NGROUPS];
// pointers to (original) parent process, youngest child,
younger sibling, older sibling, respectively. (p->father
// can be replaced with p->p_pptr->pid)
struct task_struct *p_opptr, *p_pptr, *p_cptr,
*p_ysptr, *p_osptr;
struct wait_queue *wait_chldexit;
unsigned short uid,euid,suid,fsuid;
unsigned short gid,egid,sgid,fsgid;
unsigned long timeout;
// the scheduling policy, specifies which scheduling class
the task belongs to, such as : SCHED_OTHER
//(traditional UNIX process), SCHED_FIFO (POSIX.1b
FIFO realtime process - A FIFO realtime process will
//run until either a) it blocks on I/O, b) it explicitly
yields the CPU or c) it is preempted by another realtime
//process with a higher p->rt_priority value.) and
//SCHED_RR (POSIX round-robin realtime process –
//SCHED_RRis the same as SCHED_FIFO, except that
//when its timeslice expires it goes back to the end of
//the run queue).
//unsigned long policy;
//realtime priority
unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
long utime, stime, cutime, cstime, start_time;
// mm fault and swap info: this can arguably be seen as
either mm-specific or thread-specific */
unsigned long min_flt, maj_flt, nswap, cmin_flt,
cmaj_flt, cnswap;
int swappable:1;
unsigned long swap_address;
// old value of maj_flt
unsigned long old_maj_flt;
// page fault count of the last time
unsigned long dec_flt;
// number of pages to swap on next pass
unsigned long swap_cnt;
//limits
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
// file system info
int link_count;
// NULL if no tty
struct tty_struct *tty;
// ipc stuff
struct sem_undo *semundo;
struct sem_queue *semsleeping;
// ldt for this task - used by Wine. If NULL, default_ldt is
        used
struct desc_struct *ldt;
// tss for this task
struct thread_struct tss;
// filesystem information
struct fs_struct *fs;
// open file information
struct files_struct *files;
// memory management info
struct mm_struct *mm;
// signal handlers
struct signal_struct *sig;
#ifdef __SMP__
int processor;
int last_processor;
int lock_depth; /* Lock depth.
We can context switch in and out
of holding a syscall kernel lock... */
#endif}
```

Création de processus

- **4 possibilités**
  - **Utilisation du fork()**
  - **Utilisation du clone() → processus légers qui partagent avec le père**
    - Les tables de pages, donc l'espace mémoire
    - Les fichiers ouverts
    - etc.
  - **Utilisation de l'appel système vfork()**
    - Crée un processus qui partage l'espace mémoire de son père
    - Pour éviter des problèmes d'accès concurrent, le père est bloqué jusqu'à la terminaison du fils ou l'exécution d'un nouveau programme
  - **Utilisation du pthread_create()**

Exemple (Embedded) Linux

- int fork (void) .....voi

```
#include <sys/types.
#include <unistd.h>
void program(void)
{
   processId child_processId;
   /* dupliquer le prcessus: processus fils*/
   child_processId = fork();
   if (child_processId == -1) {
     ERROR;
   }
   else if (child_processId == 0) {
     run_childProcessWork();
   }
   else {
     run_parentProcessWork();
}
```

man fork

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

Threads noyau

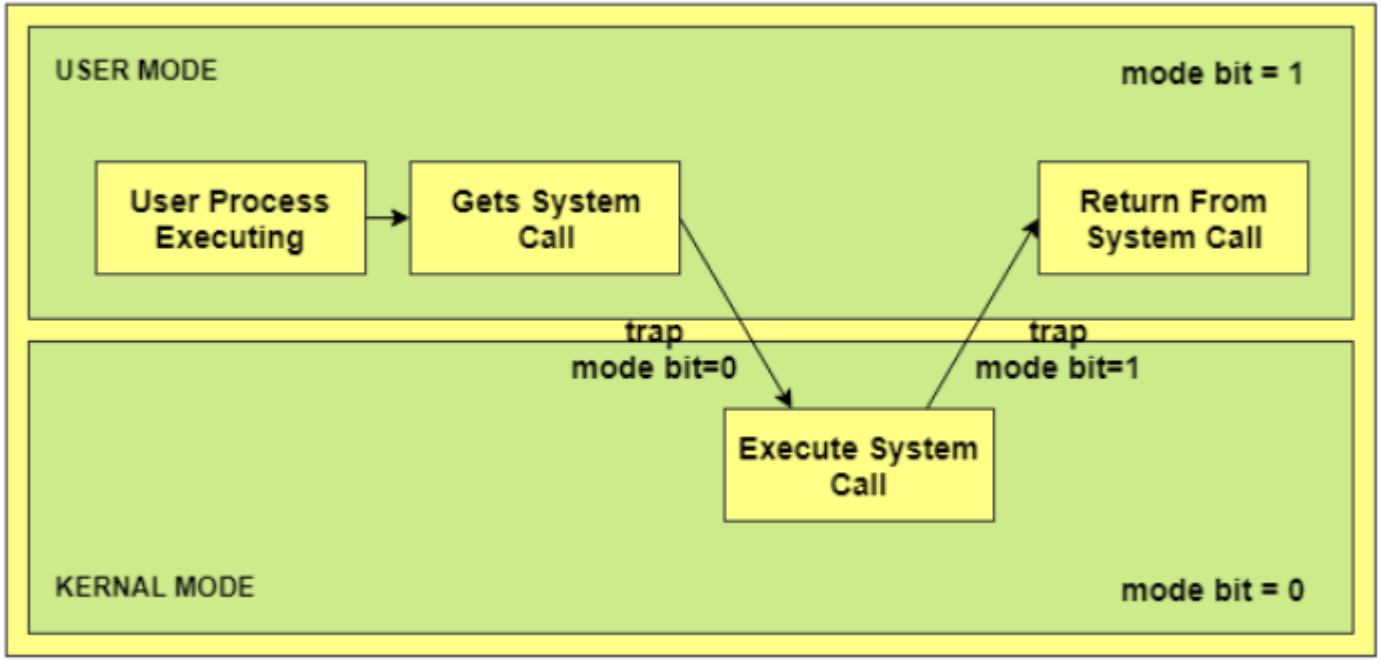
- **Des tâches vitales sont déléguées à ces « processus »**
  - **Vider les caches disque**
  - **Swapper certaines pages**
  - **...**
- **Pas forcément prioritaires, peuvent tourner en tâche de fond**
  - **Certaines de ces tâches ne peuvent tourner qu'en mode noyau**
  - **Aucun intérêt à avoir des structures pour gérer le mode utilisateur**
  - **On utilise donc des threads spéciaux → création par kernel_thread()**

Mode utilisateur et mode noyau

- **User mode**
  - **In User mode, the executing code has no ability to directly access hardware or reference memory.**
    - Code running in user mode must delegate to system APIs to access hardware or memory.
  - **Crashes in user mode are often recoverable.**
- **Kernel mode**
  - **In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware.**
    - Execute any CPU instruction and reference any memory address.
  - **Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system.**
  - **Crashes in kernel mode are catastrophic; they will halt the entire system.**
    - Reminder: advantages of micro kernel comparing to monolithic kernel (Lecture 01)

https://blog.codinghorror.com/understanding-user-and-kernel-mode/

Mode utilisateur et mode noyau



Processus 0

- Ancêtre de tous les processus
- Crée durant la phase d'initialisation du noyau
- Exécute `start_kernel()`
  - Initialise les données ainsi que les interruptions
  - Et démarre un nouveau processus
- Nouveau thread (noyau) de PID 1 (init
  - Partage toutes les données avec son père (0)
- Après avoir crée le processus num 1, le processus 0 appelle `cpu_idle()`
  - L'ordonnanceur ne choisit le processus 0 que si aucun processus n'est dans l'état TASK_RUNNING

Processus 1 et les autres

- Le processus 1 exécute la fonction `init()` qui finit l'initialisation du noyau
- Ensuite, utilise `execve()` pour exécuter le programme `init` et devient un processus
- Autres *kernel threads*
  - keventd : exécute la fonction `keventd_wq` (relative aux interruptions et exceptions)
  - kapmd : gère les événements relatifs à la gestion d'énergie
  - kswapd : récupération périodique de mémoire
  - pdflush : écrit les buffers sur disque pour récupérer de la mémoire
  - ...

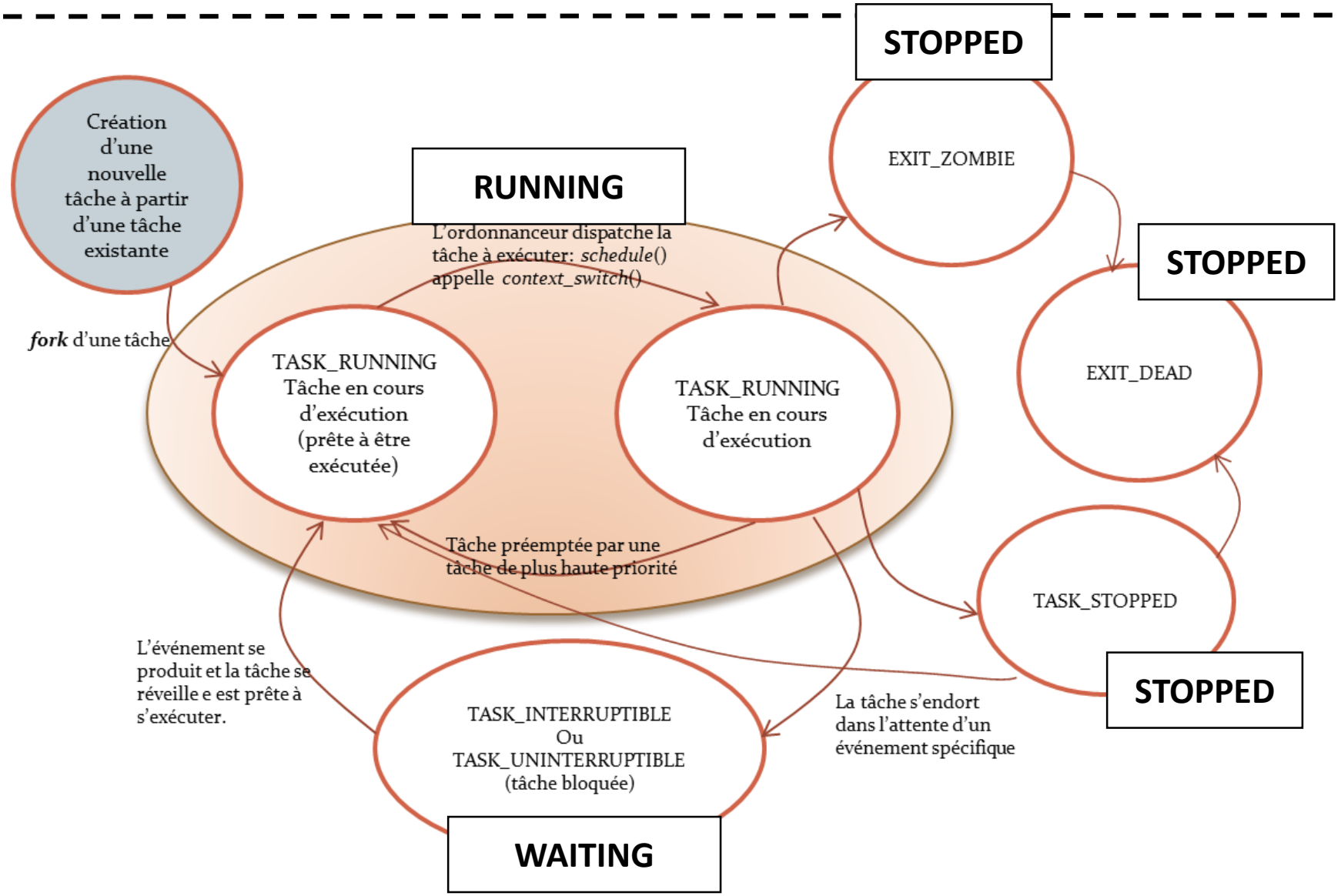
Les états d'un processus

- TASK_RUNNING
  - Processus en cours d'exécution ou en attente d'être exécuté.
- TASK_INTERRUPTIBLE
  - Le processus est suspendu/bloqué/endormi dans l'attente de la réception d'une interruption, signal ou autre
- TASK_UNINTERRUPTIBLE
  - Même que le précédent sauf que le processus n'est pas interruptible par des signaux
  - Rarement utilisé
- TASK_STOPPED
  - Processus stoppé par un signal
- TASK_TRACED
  - exécution du processus stoppée par un débogueur

Les états d'un processus

- **EXIT_ZOMBIE**
  - **Processus a terminé son exécution mais le père (en vie) n'a pas encore effectué de wait() sur ce dernier**
- **EXIT_DEAD**
  - **Après que le père aie reconnu la mort du fils, ce dernier se trouve dans cet état avant de disparaître. Cet état existe pour des raisons de synchronisation**

Les transition d'états

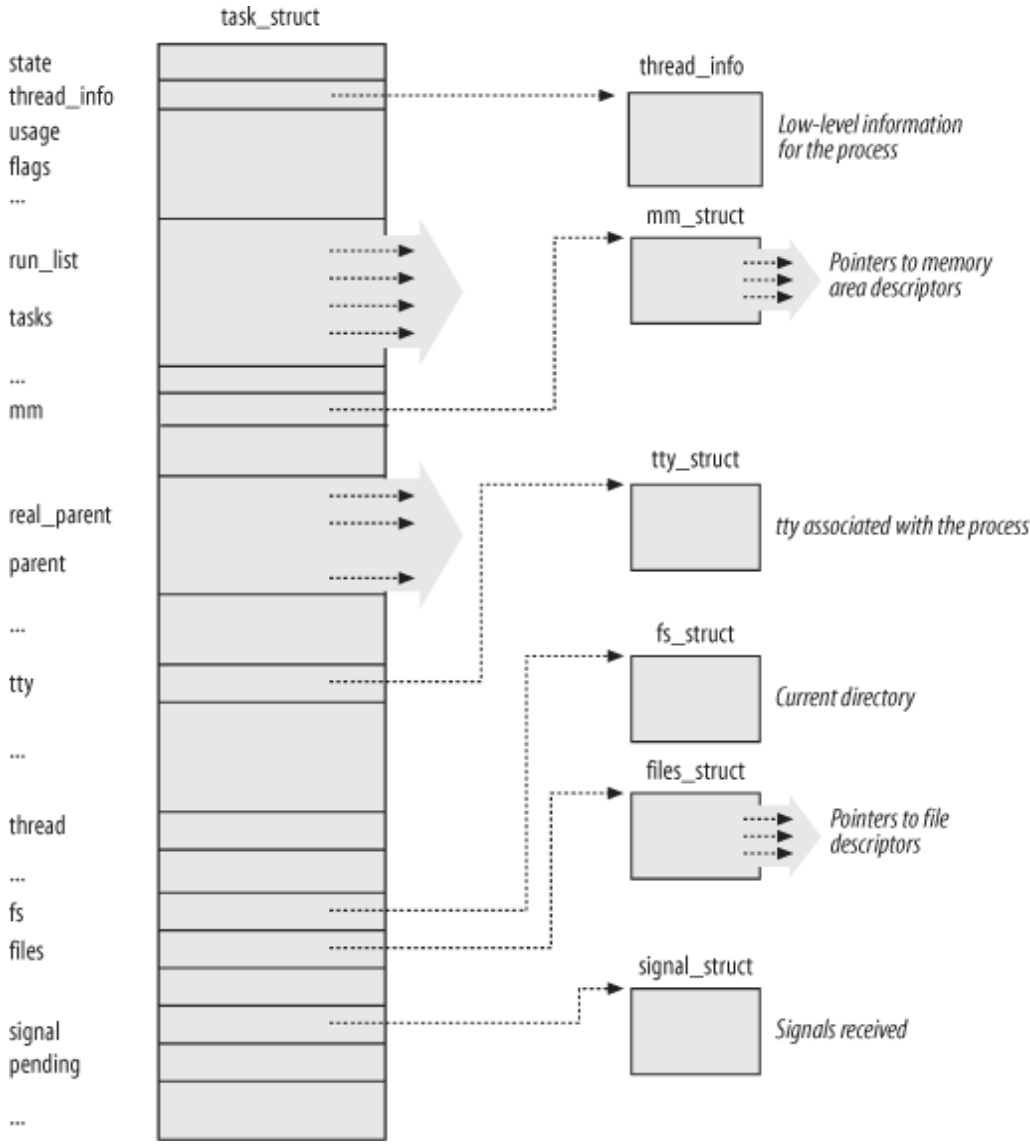


Identification d'un processus

- **Chaque processus (et processus léger / thread) est décrit par un descripteur de processus (`task_struct`)**
- **L'adresse de ce descripteur suffit pour différencier les processus**
  - Le noyau utilise souvent des pointeurs de descripteurs de processus
- **Mais on préfère manipuler des PID**
  - Stocké dans le champs pid du descripteur
- **Les PIDs sont générés séquentiellement**
  - Le PID d'un nouveau processus est celui du dernier crée +1
  - Valeur max : PID_MAX_DEFAULT -1 (32767)
    - Peut être changé dans /proc/sys/kernel/pid_max
    - 4 194 303 (max) sur processeur 64 bits
- **Peut être nécessaire de recycler les PIDs**
  - Utilisation d'une bitmap pidmap_array (1 page vérouillée)

Structure d'un PCB (Linux)

- **Descripteur de processus**
  - **Contient toutes les informations relatives à un processus à garder par le noyau (structure task_struct)**



Structure d'un PCB (Linux)

```
struct task_struct //~350 lignes de code
{
....
// -1 unrunnable, 0 runnable, >0 stopped
volatile long state;
// number of clock ticks left to run in this scheduling slice,
        decremented by a timer.
long counter;
// the process' static priority, only changed through well-
        known system calls like nice, POSIX.1b
// sched_setparam, or 4.4BSD/SVR4 setpriority.
long priority;
unsigned long signal;
// bitmap of masked signals
unsigned long blocked;
// per process flags, defined below
unsigned long flags;
int errno;
// hardware debugging registers
long debugreg[8];
struct exec_domain *exec_domain;
struct linux_binfmt *binfmt;
struct task_struct *next_task, *prev_task;
struct task_struct *next_run, *prev_run;
unsigned long saved_kernel_stack;
unsigned long kernel_stack_page;
int exit_code, exit_signal;
unsigned long personality;
int dumpable:1;
int did_exec:1;
int pid;
int pgrp;
int tty_old_pgrp;
int session;
// boolean value for session group leader
int leader;
int groups[NGROUPS];
// pointers to (original) parent process, youngest child,
younger sibling, older sibling, respectively. (p->father
// can be replaced with p->p_pptr->pid)
struct task_struct *p_opptr, *p_pptr, *p_cptr,
*p_ysptr, *p_osptr;
struct wait_queue *wait_chldexit;
unsigned short uid,euid,suid,fsuid;
unsigned short gid,egid,sgid,fsgid;
unsigned long timeout;
// the scheduling policy, specifies which scheduling class
the task belongs to, such as : SCHED_OTHER
//(traditional UNIX process), SCHED_FIFO (POSIX.1b
FIFO realtime process - A FIFO realtime process will
//run until either a) it blocks on I/O, b) it explicitly
yields the CPU or c) it is preempted by another realtime
//process with a higher p->rt_priority value.) and
//SCHED_RR (POSIX round-robin realtime process –
//SCHED_RRis the same as SCHED_FIFO, except that
//when its timeslice expires it goes back to the end of
//the run queue).
//unsigned long policy;
//realtime priority
unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
long utime, stime, cutime, cstime, start_time;
// mm fault and swap info: this can arguably be seen as
either mm-specific or thread-specific */
unsigned long min_flt, maj_flt, nswap, cmin_flt,
cmaj_flt, cnswap;
int swappable:1;
unsigned long swap_address;
// old value of maj_flt
unsigned long old_maj_flt;
// page fault count of the last time
unsigned long dec_flt;
// number of pages to swap on next pass
unsigned long swap_cnt;
//limits
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
// file system info
int link_count;
// NULL if no tty
struct tty_struct *tty;
// ipc stuff
struct sem_undo *semundo;
struct sem_queue *semsleeping;
// ldt for this task - used by Wine. If NULL, default_ldt is
        used
struct desc_struct *ldt;
// tss for this task
struct thread_struct tss;
// filesystem information
struct fs_struct *fs;
// open file information
struct files_struct *files;
// memory management info
struct mm_struct *mm;
// signal handlers
struct signal_struct *sig;
#ifdef __SMP__
int processor;
int last_processor;
int lock_depth; /* Lock depth.
We can context switch in and out
of holding a syscall kernel lock... */
#endif}
```

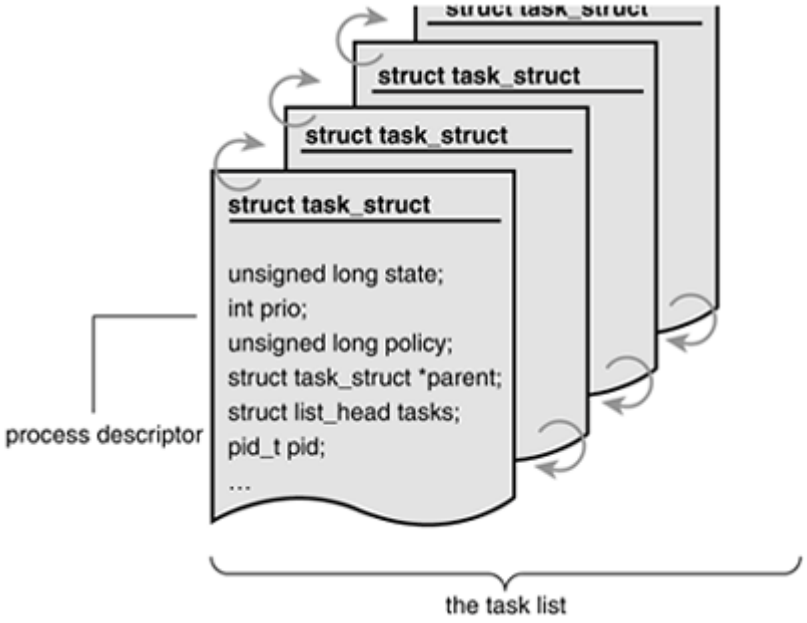
Identification d’un processus

- **Linux associe un PID à tous les processus (« lourds » et « légers » - threads)**
- **Mais le programmeur s’attend à ce que les processus légers de la même application aient le même PID (POSIX 1003.1c)**
- **Linux utilise des groupes de threads**
  - Ensemble des processus légers originaires d’une même tâche
  - Le premier est le thread group leader
  - Les threads partagent le PID du leader stocké dans le champs tgid
  - Un appel à getpid( )retourne le tgid et non le pid
- **Obtenir le descripteur du processus courant**
  - Macro current (« linux/thread_info.h »)

Listes des processus

- **Les processus sont gérés par le noyau grâce à des listes doublement chaînées**
  - **Liste de tous les processus**
    - Accessible depuis init_task (task_struct)
    - Process descriptor du processus 0 (swapper)
    - prev de l'init_task pointe vers le dernier processus inséré
  - **Liste des TASK_RUNNING (runqueue)**
    - Maintient d'une liste séparée pour les processus en attente du CPU
    - Évite de devoir scanner tous les processus pour trouver celui à exécuter
      - Mais il faut quand même scanner tous les candidats pour trouver le meilleur
      - Amélioration en 2.6 (plusieurs listes: une par priorité 0->139)
  - **Liste des TASK_(NON)INTERRUPTIBLE (waitqueue)**
    - Utilisation des **wait queues**

Listes des processus



Source*: Linux Kernel developement 2nd* edition, Prentice Hall ed, Venkateswaran

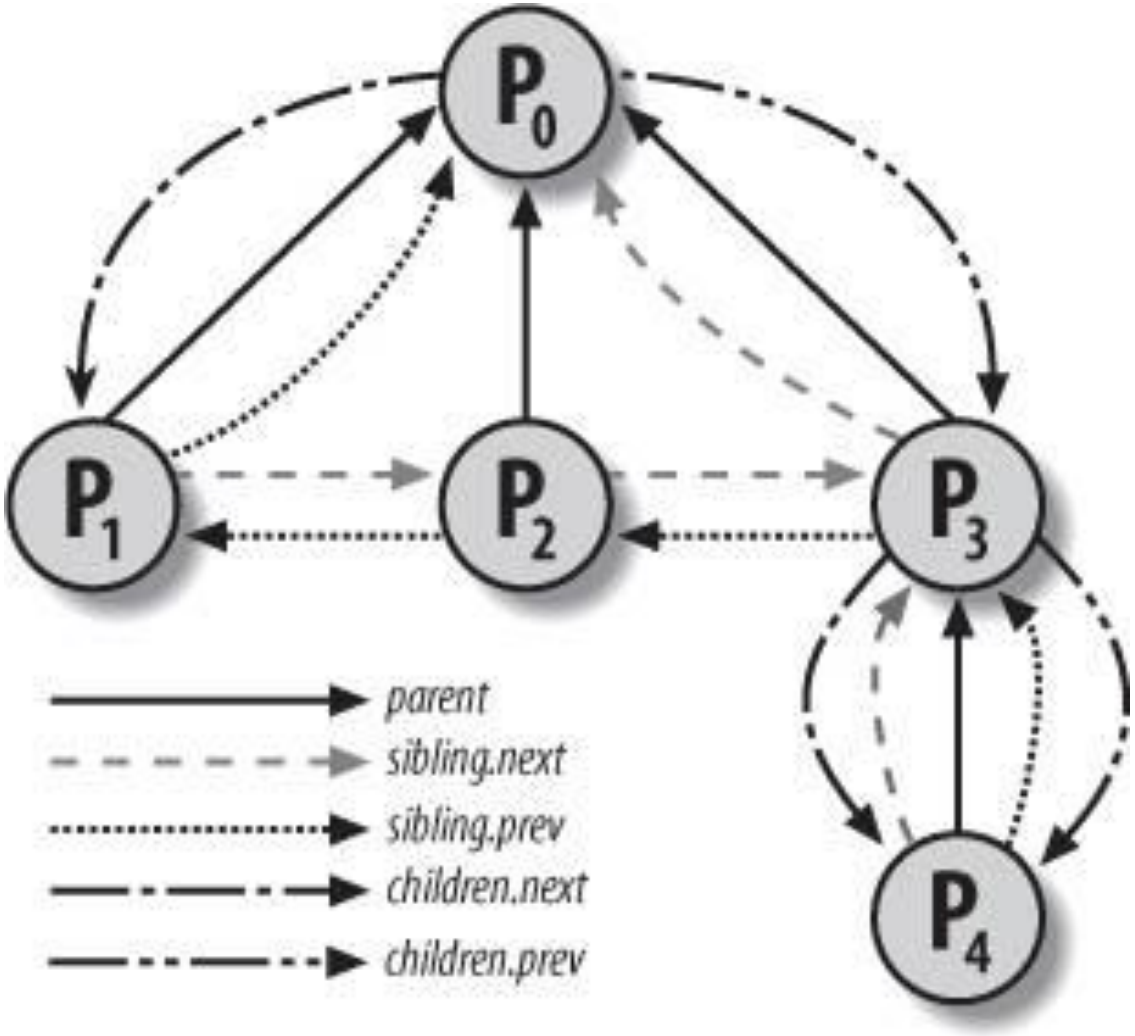
Structure d'un PCB (Linux)

```
struct task_struct //~350 lignes de code
{
....
// -1 unrunnable, 0 runnable, >0 stopped
volatile long state;
// number of clock ticks left to run in this scheduling slice,
        decremented by a timer.
long counter;
// the process' static priority, only changed through well-
        known system calls like nice, POSIX.1b
// sched_setparam, or 4.4BSD/SVR4 setpriority.
long priority;
unsigned long signal;
// bitmap of masked signals
unsigned long blocked;
// per process flags, defined below
unsigned long flags;
int errno;
// hardware debugging registers
long debugreg[8];
struct exec_domain *exec_domain;
struct linux_binfmt *binfmt;
struct task_struct *next_task, *prev_task;
struct task_struct *next_run, *prev_run;
unsigned long saved_kernel_stack;
unsigned long kernel_stack_page;
int exit_code, exit_signal;
unsigned long personality;
int dumpable:1;
int did_exec:1;
int pid;
int pgrp;
int tty_old_pgrp;
int session;
// boolean value for session group leader
int leader;
int groups[NGROUPS];
// pointers to (original) parent process, youngest child,
younger sibling, older sibling, respectively. (p->father
// can be replaced with p->p_pptr->pid)
struct task_struct *p_opptr, *p_pptr, *p_cptr,
*p_ysptr, *p_osptr;
struct wait_queue *wait_chldexit;
unsigned short uid,euid,suid,fsuid;
unsigned short gid,egid,sgid,fsgid;
unsigned long timeout;
// the scheduling policy, specifies which scheduling class
the task belongs to, such as : SCHED_OTHER
//(traditional UNIX process), SCHED_FIFO (POSIX.1b
FIFO realtime process - A FIFO realtime process will
//run until either a) it blocks on I/O, b) it explicitly
yields the CPU or c) it is preempted by another realtime
//process with a higher p->rt_priority value.) and
//SCHED_RR (POSIX round-robin realtime process –
//SCHED_RRis the same as SCHED_FIFO, except that
//when its timeslice expires it goes back to the end of
//the run queue).
//unsigned long policy;
//realtime priority
unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
long utime, stime, cutime, cstime, start_time;
// mm fault and swap info: this can arguably be seen as
either mm-specific or thread-specific */
unsigned long min_flt, maj_flt, nswap, cmin_flt,
cmaj_flt, cnswap;
int swappable:1;
unsigned long swap_address;
// old value of maj_flt
unsigned long old_maj_flt;
// page fault count of the last time
unsigned long dec_flt;
// number of pages to swap on next pass
unsigned long swap_cnt;
//limits
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
// file system info
int link_count;
// NULL if no tty
struct tty_struct *tty;
// ipc stuff
struct sem_undo *semundo;
struct sem_queue *semsleeping;
// ldt for this task - used by Wine. If NULL, default_ldt is
        used
struct desc_struct *ldt;
// tss for this task
struct thread_struct tss;
// filesystem information
struct fs_struct *fs;
// open file information
struct files_struct *files;
// memory management info
struct mm_struct *mm;
// signal handlers
struct signal_struct *sig;
#ifdef __SMP__
int processor;
int last_processor;
int lock_depth; /* Lock depth.
We can context switch in and out
of holding a syscall kernel lock... */
#endif}
```

Relations entre les processus

- **Histoires de … « familles » (toujours compliquées!):**
  - `real_parent` : le processus père sinon le processus *init*
  - `parent` : généralement le *real_parent* sauf dans le cas d'un *ptrace*
  - `children` : tête de liste contenant les processus fils
  - `sibling` : pointeurs vers le *prev* et *next* dans la liste des processus ayant le même père
- **Autres relations**
  - `group_leader` : pointeur sur le descripteur de *processus* du group_leader
  - `ptrace_children` : tête de liste de tous les fils de P qui sont tracés par le dbg
  - `ptrace_list` : pointeur vers les éléments suivants et précédents dans la liste des processus tracés du parent réel

Relations entre les processus



Les files d'attente d'événement *wait queues*

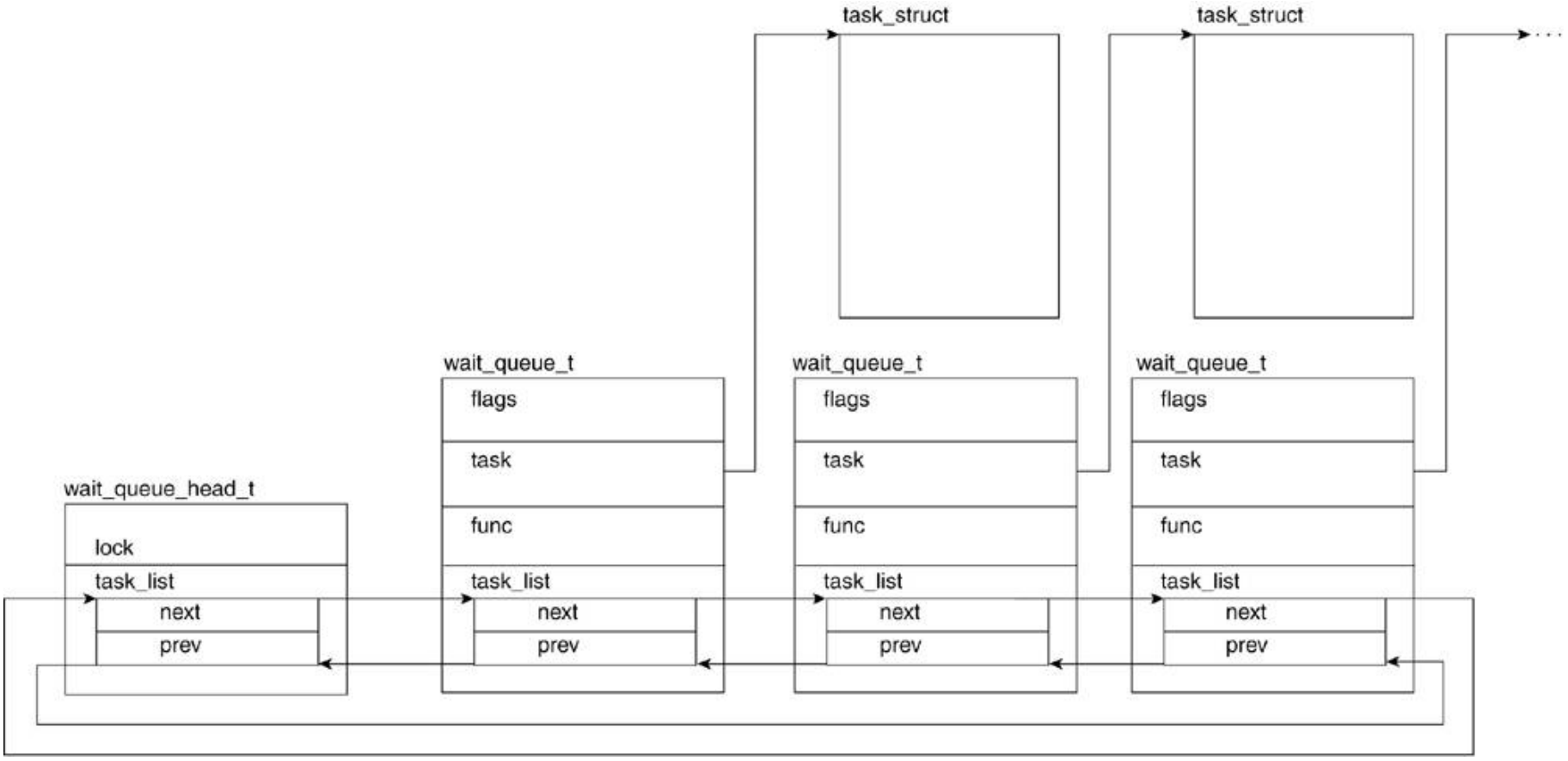
- **Les wait queues implémentent des attentes conditionnelles**
  - **Un processus voulant attendre un événement particulier se place dans la *wait queue* correspondante**
  - **C'est donc un ensemble de processus qui seront réveillés par le noyau lorsque l'événement en question se produit.**

```
struct __wait_queue_head {
   spinlock_t lock;
   struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
struct __wait_queue {
   unsigned int flags;
   struct task_struct * task;
   wait_queue_func_t fun;
   struct list_head task_list;
}
typedef struct __wait_queue wait_queue_t;
```

Tête de la *wait queue*

Éléments de la *wait queue*

Les files d'attente d'événement *wait queues*



Les files d'attente d'événement *wait queues*

- **Chaque élément de la wait queue est un processus en attente**
  - **L'adresse de son descripteur est dans le champs `task`**
  - **`task_list` relie les processus en attente du même évènement**
- **Réveiller tous les processus en attente n'est pas forcément une bonne idée**
  - **Plusieurs en attente d'une ressource exclusive**
- **Introduction de 2 types de processus endormis/bloqués**
  - **Processus exclusifs : réveillés chacun leur tour par le noyau**
  - **Processus non exclusifs : tous réveillés par le noyau**
  - **Champs flags (exclusif == 1)**
- **func : spécifie comment les processus doivent être réveillés**

Les files d'attente d'événement *wait queues*

- **Pour réveiller un processus (FUNC), le noyau utilise les macros suivantes**
  1. `wake_up`
  2. `wake_up_nr`
  3. `wake_up_all`
  4. `wake_up_interruptible`
  5. `wake_up_interruptible_nr`
  6. `wake_up_interruptible_all`
  7. `wake_up_interruptible_sync`
  8. `wake_up_locked`
  - **Ces macros prennent en compte les processus dans l'état TASK_INTERRUPTIBLE ou TASK_UNINTERRUPTIBLE (sauf si la macro inclut le mot `interruptible`)**

Les files d'attente d'événement *wait queues*

- **Elles réveillent tous les processus non exclusifs qui ont l'état requis**
- **Les macros avec *nr* réveillent un nombre donné de processus exclusifs**
- **Les macros avec *all* réveillent tous les processus exclusifs**
- **Les autres réveillent 1 processus exclusif**
- **Les macros qui ne contiennent pas « sync »**
  - Vérifient que la priorité des processus réveillés n'est pas supérieure à celle du processus courant
  - Appelle `schedule()` si nécessaire

Les files d'attente d'événement *wait queues*

```
#include <linux/wait.h>
// Data structure: wait_queue_head_t
wait_queue_head_t wq;
init_waitqueue_head(&wq);
```

```
Process A
...
//wait_event(queue, condition);
wait_event(wq, flag == 'y')
...
```

```
Process B
...
flag = 'y'
//wake_up(wake_queue_head_t *)
wake_up(wq);
...
```

Limites des ressources d'un processus

- **Chaque processus a un ensemble de ressources limitées**
  - **Stockés dans `current->signal->rlim` (champs relatif aux signaux), *current* étant le descripteur du processus courant**
  - **Tableau de `struct rlimit` (1 struct par ressource)**
    - `unsigned long rlim_cur;`
    - `unsigned long rlim_max`
  - **`RLIMIT_AS` : Taille maximale de l'espace d'adressage. Vérifiée lors d'un `malloc()`**
  - **`RLIMIT_CORE` : Taille maximale du *core dump***
  - **`RLIMIT_CPU` : Temps CPU max en secondes pour un processus. Il Reçoit un `SIGXCPU` si dépassement, puis `SIGKILL` si non terminaison.**
  - **`RLIMIT_DATA` : Taille maximale du tas (*heap)***
  - **`RLIMIT_FSIZE` : Taille maximale de fichier, `SIGXFSZ` si dépassement**

Limites des ressources d'un processus

- `RLIMIT_LOCKS` : nombre maximal de verrous (lock) sur fichiers (non utilisé actuellement)
- `RLIMIT_MEMLOCK` : Taille maximale de mémoire non swappable (verrouillée en mémoire physique)
- `RLIMIT_MSGQUEUE` : nombre max d'octets dans une file de msg POSIX
- `RLIMIT_NOFILE` : nombre max de descripteurs de fichiers ouverts
- `RLIMIT_NPROC` : nombre max de processus que l'utilisateur peut posséder
- `RLIMIT_RSS` : nombre maximum de cadres que le processus peut posséder (non utilisé actuellement)
- `RLIMIT_SIGPENDING` : Nombre max de signaux en attente
- `RLIMIT_STACK` : Taille maximale de la pile

Limites des ressources d'un processus

- **Obtenir la limite actuelle pour une ressource (ex. temps CPU)**

```
current->signal->rlim[RLIMIT_CPU].rlim_cur
```

- **On peut augmenter la limite sur une ressource jusqu'au maximum autorisé**
  - `getrlimit()` et `setrlimit()`
- **Seul le super utilisateur peut augmenter la limite max**
- **Possibilité de les mettre en « sans limite » (limités seulement par l'implémentation)**
  - `RLIM_INFINITY`
- **Quand un utilisateur se connecte sur la machine**
  - Le noyau démarre un processus super utilisateur
  - Ce processus fixe les limites
  - Et exécute ensuite un shell de login qui deviendra propriété de l'utilisateur
  - Chaque nouveau processus conserve les limites du parent

Université de Bretagne Occidentale

Ordonnancement

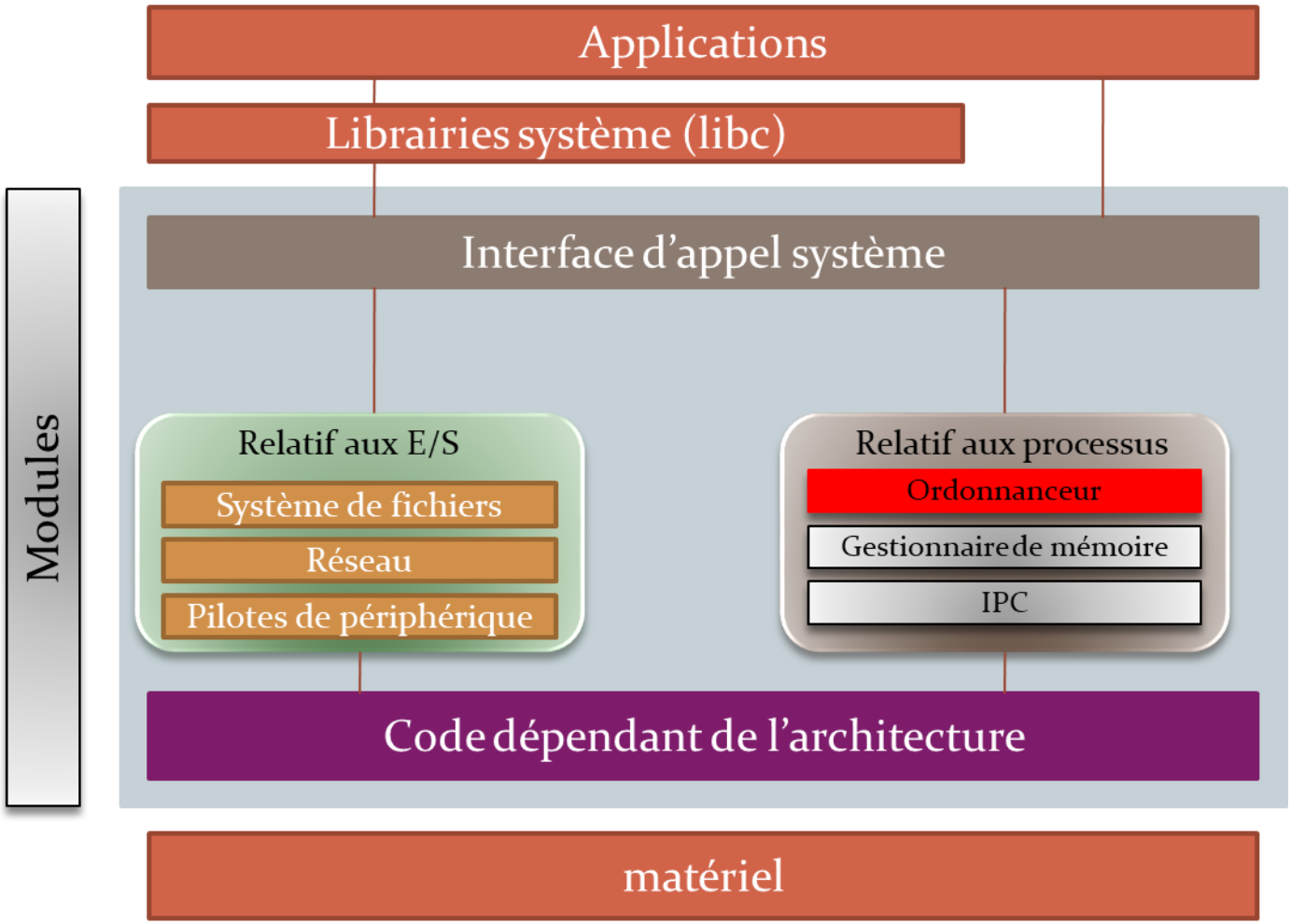
Cours 2 : Services des systèmes d'exploitation

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. **Ordonnancement**
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Hai Nam TRAN
UBO – M2 LSE

Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

Schéma d'un OS



Tâches de l'ordonnanceur

- **L'ordonnanceur est le composant de l'OS qui détermine l'ordre et la durée des tâches qui s'exécutent sur le CPU**
  - **L'ordonnanceur dicte l'état dans lequel doivent se trouver les tâches**
  - **Il charge et décharge le bloc de contrôle de la tâche (descripteur de processus ou *task_struct* dans le cas de Linux).**
  - **Certains OS utilisent un processus séparé pour allouer le CPU au processus nouvellement sélectionné, il s'appelle le dispatcheur.**
  - **Tous les processus ne sont pas égaux**
    - L'interactivité est importante
    - Les tâches de fonds… moins

Ordonnancement

- **Passer d'un processus à un autre est coûteux**
  1. **Sauvegarde de l'état du processus**
  2. **Chargement du nouvel état (registres, mémoire…)**
  3. **Démarrage du nouveau processus**
  4. **Invalidation probable du cache mémoire**
- **Les processus alternent souvent des phases de calcul avec des phases d'E/S**
  - **Si phases de calcul très importantes : *CPU-Bound***
  - **Si E/S : *I/O-Bound***

Ordonnancement

- **Métriques clés**
  - **Temps de réponse de l'ordo**
    - Temps pris par l'ordonnanceur pour effectuer le changement de contexte pour une tâche prête, ce temps inclut le temps d'attente de la tâche dans la file des tâches prêtes.
  - **Le temps pris par le processus pour compléter son exécution (*Turnaround time*).**
  - **Le temps et les données dont a besoin l'ordonnanceur pour choisir la tâche à exécuter (*Overhead*).**
  - **Débit: nombre de tâches traitées en un temps donné.**
  - **Équité: quels sont les facteurs déterminant quant au choix de la tâche à exécuter.**
  - **Famine: un ordonnanceur doit (dans certains cas) assurer que ce problème ne se produise pas…**
  - **Préemptivité et non préemptivité**
  - **Note: l'ordonnancement, c'est de la pure perte de temps: plus l'algorithme est complexe, moins il est bon de l'implémenter**

Quand ordonnancer

- **Les décisions d'ordonnancement sont prises dans plusieurs circonstances:**
  - **Fork : qui choisir entre le père et le fils**
  - **Fin d'un processus**
    - En trouver un autre
    - Si aucun, un processus spécial est exécuté (idle ou Processus inactif du système)
  - **Quand un processus devient bloqué**
    - Attente d'une E/S ou d'une condition
    - Peut influer sur le prochain à exécuter
  - **Interruption E/S**
    - Si indique fin de traitement, réordonnancer le processus qui attendait
- **Ordonnancement non préemptif**
  - **Un processus est choisi et détient le CPU jusqu'à ce qu'il bloque ou se termine**
- **Ordonnancement préemptif**
  - **Un processus n'a le CPU que pour une durée donnée (timeslice)**

Différents besoins/différents ordonnancements

- Des systèmes ont des propriétés et besoins différents
- Il faut des ordonnanceurs adaptés
- Batch
  - Pas d'utilisateurs devant des écrans
  - Ordonnanceurs non préemptifs ou presque…
  - Changements de processus réduits
  - Ex: compilateur de langage, recherche dans des BDs, calcul scientifique, etc.
- Interactif
  - Préemption pour maintenir la réactivité
  - Ex: éditeur de texte, shell, etc.
- Temps réel
  - Préemption
  - Ex: Application vidéo, audio, collecte de données à partir d'un capteur.

Objectifs de l'ordonnancement

- **Les objectifs des algorithmes d'ordonnancement dépendent du système considéré**
  - **Mais certaines propriétés sont communes**
- **Ce qui est commun**
  - **Respect des règles : permettre à certains processus d'avoir un ordonnancement particulier**
  - **Équilibre : Maximiser l'utilisation du système (alterner des E/S et du CPU)**
  - **Équité : chaque processus doit avoir accès au CPU de manière équitable**
- **Batch**
  - **Débit : maximiser le nombre de jobs par heure**
  - **Temps d'exécution de l'appli: minimiser le temps entre la soumission et la complétion**
  - **CPU : maximiser l'utilisation du CPU**

Objectifs de l'ordonnancement

- **Interactif**
  - **Réactivité: répondre rapidement aux demandes (temps de réponse)**
  - **Proportionnalité**
    - Les utilisateurs ont souvent une idée du temps que va prendre une opération (click == rapide…)
    - L'ordonnanceur doit choisir les processus pour être conforme à leurs attentes
- **Temps réel**
  - **Respecter les contraintes temporelles**
  - **Prédictibilité : l'ordonnanceur doit être prévisible**
- **Certains de ces objectifs peuvent ne pas être maximisés en même temps (voire contradictoires)**
  - **Débit et temps de réponse par exemple…**

Politique FCFS / exécution jusqu'à terminaison

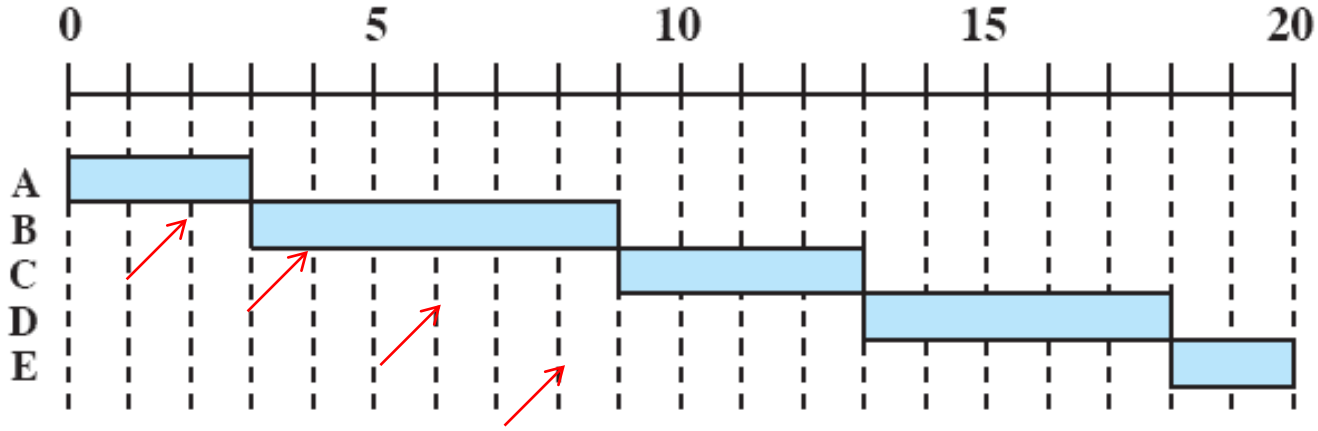
- First Come First Served (premier arrivé, premier servi).
- Ordonnancement utilisé pour les batchs
- Approche non préemptive
  - Pas de file d'attente de tâches bloquées
- Temps de réponse lent
  - Surtout s'il existe des tâches longues
  - Dans ce cas: pb d'équité (les tâches courtes attendent plus que les tâches longues)
- Famine: impossible.

FCFS

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

First-Come-First Served (FCFS)



W. Stallings, Operating Systems Internal Design and Principles, 6th ed. Pearson Int. Edition

Le plus court d'abord (SPN/Shortest Process Next)

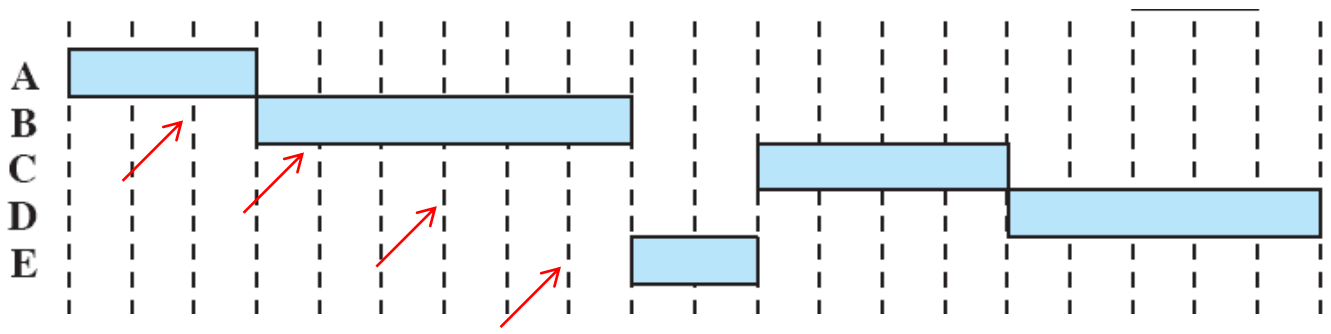
- **On suppose que le temps d'exécution est connu à l'avance (dur dur !).**
  - **La tâche la plus courte de l'ensemble des processus s'exécute en premier.**
  - **Temps de réponse plus court que le précédent**
  - **Famine possible**
  - **Temps de calcul (de l'ordonnanceur) important : pour savoir quel processus doit s'exécuter**
  - **Version préemptive : SRT (Shortest Remaining Time)**

SPN

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Shortest Process Next (SPN)



Round Robin

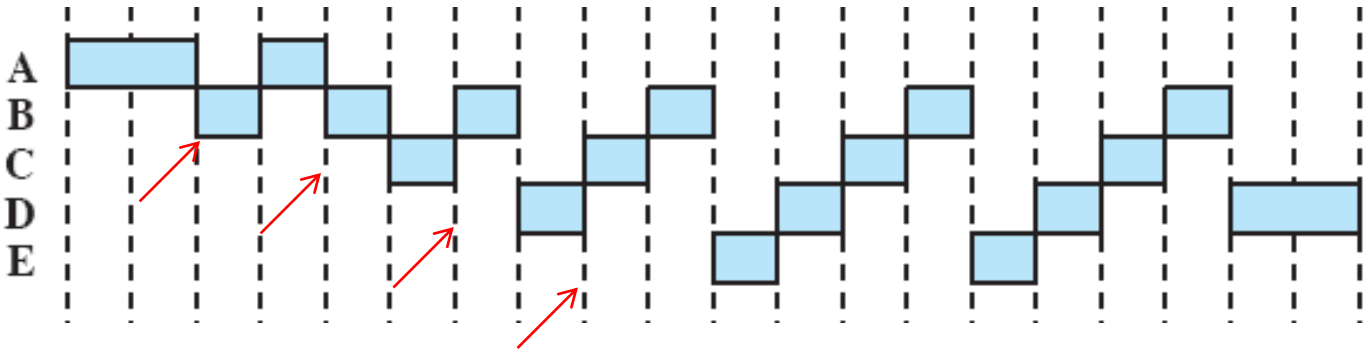
- File d'attente **FIFO** stocke les processus prêts.
- Exécution indépendante de la charge de travail et de l'interactivité.
- **Quantum de temps** égal pour tous
  - Durée maximale durant laquelle il peut s'exécuter
  - S'il atteint son quantum, il est préempté
  - S'il est bloqué avant, un autre processus est lancé
- Famine : impossible
- Quantum de temps:
  - **Trop grand** : trop d'attente par processus
  - **Trop petit** : temps de changement de contexte relativement important (en pourcentage).
  - En général, entre 10 et 50ms

Round Robin

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

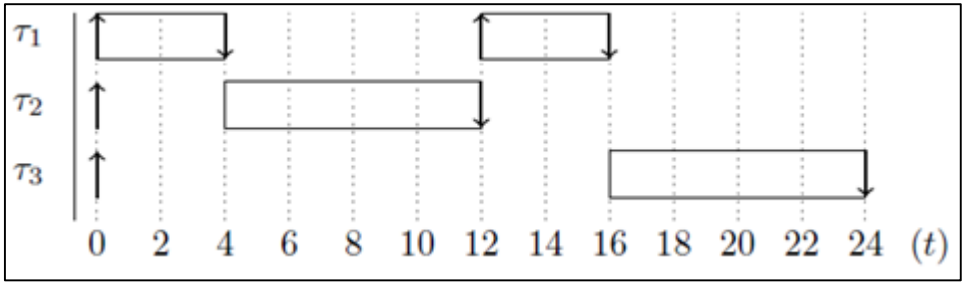
Round-Robin (RR), $q = 1$



Ordonnancement prioritaire

- **Principe**
  - **Tous les processus ne sont pas aussi importants les uns que les autres**
  - **Chaque processus se voit attribuer une priorité**
  - **Le processus de priorité supérieure préempte ceux de priorité inférieure.**

Task	C_i	T_i	D_i	O_i	Π_i
τ_1	4	12	12	0	3
τ_2	8	24	24	0	2
τ_3	8	24	24	0	1



Ordonnancement prioritaire

- **Pb de famine → solution:**
  - **Priorité dynamique: à chaque tic d'horloge la priorité du processus en exécution est réduite**
  - **Si inférieure à un autre processus, changement de contexte**

- **Optimiser les performances**
  - **Un processus qui fait beaucoup d'E/S devrait avoir une grande priorité car il consomme peu de CPU (linux 2.6 …)**
  - **Simple : la priorité est une fraction du quantum effectivement utilisé**
    - Un processus qui tourne 1ms sur 50 a une priorité de 50
    - Un processus qui tourne 50ms a une priorité de 1
- **Pb *d'inversion de priorité:***
  - **Processus de priorité importante est en attente d'exécution d'un processus de priorité inférieure.**
  - **Et les processus avec une priorité entre les 2 derniers sont en cours d'exécution.**

La notion d'espace de priorités

- **2 possibilités :**
  - **Certains OS laissent toutes les tâches (système et utilisateur) « vivre » dans un même espace de priorités.**
  - **D'autres isolent l'espace des priorités des processus système et utilisateurs (WindowsNT, Linux, UNIX)**
    - Sous Unix par ex, les tâches utilisateurs voulant augmenter leur priorités peuvent utiliser l'appel à `nice()`
    - … mais sont tous préemptées par une tâche système.

La notion d'espace de priorités

- **L'espace noyau a 3 types de *« tâches » (en termes de priorités):***
  1. **Interruptions: c'est une interruption hw (timer, clavier, réseau, etc). Ce type de tâche est appelé ISR (Interrupt Service Routine). Ce n'est pas vraiment une tâche mais une fonction exécutée *indépendamment de l'ordonnanceur* à chaque interruption. L'OS n'est pas impliqué.**
  2. **Fonctions *tasklets (minitâches)* et routines de service différées: fonctions qui peuvent être activées par n'importe quelle tâche du noyau (pour la mini tâche) ou par une fonction d'interruption (pour les DSR). Les interruptions sont actives lors de l'exécution de ces fonctions *et l'OS est impliqué* pour déterminer l'ordre d'exécution.**
  3. **Toutes les autres tâches du noyau: le niveau le moins prioritaire du noyau, préempte toutes les tâches utilisateurs.**

Université de Bretagne Occidentale

Ordonnancement sous Linux

Cours 2 : Services des systèmes d'exploitation

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. **Ordonnancement**
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Hai Nam TRAN
UBO – M2 LSE

Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

A little bit of history

- **Linux 0.01, 1991**
  - **Only one process queue, check all processes when need to choose a new one to run**
    - At that time, the maximum number of tasks was greatly restricted – 32 by default
- **Linux < 2.6, 1991 - 2003**
  - **The O(n) scheduler – not much different from the one in 0.01…**
  - **At each process change the kernel goes through the list of ready processes**
    - Compute their priorities
    - And choose the winner
  - **Still very expensive algorithm if many processes**
  - **Not well-adapted to multi-core architectures**

A little bit of history

- **Linux 2.6.x (x ≤ 23), 2003**
  - **The O(1) scheduler**
  - **Select a process to run in a constant amount of time**
    - *“the trick of speeding up the scheduler was in splitting the runqueue into many lists of runnable processes, one list per priority → 140 lists”*
  - **One list of process / CPU**
  - **Better distinction between interactive and batch process**

- **Staircase schedulers, 2004**
  - **Con Kolivas – an Australian anaesthetist**
  - **Not really welcomed by the kernel community because it was too desktop oriented ???**
    - At that time, Linux was mostly used for server and database segment
    - … on 31 August 2009, Kolivas posted a new scheduler called BFS (Brain Fuck Scheduler)

A little bit of history

- **Linux 2.6.23, 2007 - Now**
  - **Completely Fair Scheduler**
  - **“*Basically models an “ideal, precise multitasking CPU” on real hardware*”**
  - **Homework**
    - Research and find information about CFS
    - Write > 2-page document to explain your understanding about this scheduler
    - To be included in your final report

Université de Bretagne Occidentale

Ordonnancement sous Linux 2.6.x (avec x≤23)

Cours 2 : Services des systèmes d'exploitation

Hai Nam TRAN
UBO – M2 LSE

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. **Ordonnancement**
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

Politique d'ordonnancement

- L'ordonnancement de Linux fonctionne avec des quanta dynamiques
- Les processus sont ordonnés par priorité
  - La valeur de cette dernière indique à l'ordonnanceur quel processus exécuter en premier
- Les priorités sont dynamiques
  - Les processus qui n'ont pas eu le CPU voient leur priorité augmentée (moins de pb de famine).
- Linux reconnaît les processus temps réel (mou)
- Mais aucune notion « explicite » de batch vs interactif, … mais implicite:
  - Décision prise avec une heuristique basée sur le comportement antérieur des processus

kernel/sched.c (Linux 2.6.18)

```
/*
 * task_timeslice() scales user-nice values [ -20 ... 0 ... 19 ]
 * to time slice values: [800ms ... 100ms ... 5ms]
 *
 * The higher a thread's priority, the bigger timeslices
 * it gets during one round of execution. But even the lowest
 * priority thread gets MIN_TIMESLICE worth of execution time.
 */

#define SCALE_PRIO(x, prio) \
        max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO / 2),
MIN_TIMESLICE)

static unsigned int static_prio_timeslice(int static_prio)
{
        if (static_prio < NICE_TO_PRIO(0))
                return SCALE_PRIO(DEF_TIMESLICE * 4, static_prio);
        else
                return SCALE_PRIO(DEF_TIMESLICE, static_prio);
}
```

Préemption

- **Les processus peuvent être préemptés**
  - **Quand un processus devient TASK_RUNNING**
    - Le noyau vérifie si sa priorité est plus grande que celle du processus en cours d'exécution
    - Si oui, l'exécution du processus courant est interrompue et l'ordonnanceur est invoqué
  - **Quand un processus fini tout son quantum de temps**
    - Drapeau (champs flag) `TIF_NEED_RESCHED` mis à 1 dans la structure `thread_info`
    - Quand interruption horloge → le noyau invoque l'ordonnanceur si le drapeau est à 1
  - **Avant 2.6**
    - Un processus en mode noyau ne pouvait pas être préempté

Exemple

- **2 processus**
  - **Un éditeur de texte → interactif → plus haute priorité dynamique**
    - L'utilisateur alterne temps de réflexion et entrée des données
    - Temps mis entre pressions sur le clavier est assez important
  - **Un compilateur → batch → plus basse priorité**
- **Dès qu'il y a une pression sur un bouton**
  - **Une interruption (relative à la pression sur le bouton) est générée**
  - **Le noyau réveille le processus de l'éditeur de texte (sort de la waitqueue et devient prêt → TASK_RUNING)**
  - **Le noyau détermine/voit aussi la priorité de l'éditeur qui est supérieure à celle du compilateur**
  - **Il met le drapeau TIF_NEED_RESCHED à 1 (du compilateur) pour forcer l'ordonnanceur à s'activer lorsque le noyau aura fini de gérer l'interruption**
  - **L'ordonnanceur prend alors le relais et effectue le changement de processus**
  - **Le caractère est alors affiché sur l'écran**

Classes d'ordonnancement

- **Les processus sont ordonnancés selon 3 classes**
  - **SCHED_FIFO**
    - Processus temps réel en fifo
    - Ordonnancement à priorité
    - Quand l'ordonnanceur assigne le CPU à un processus, il ne change pas sa position dans la *runqueue* des processus prêts (jusqu'à terminaison de l'exécution, sauf si processus de priorité plus importante)
    - Si un autre processus de même classe et même priorité est prêt …tant pis
  - **SCHED_RR**
    - Processus temps réel en Round Robin
    - Le descripteur de processus est mis en fin de liste une fois le CPU assigné
    - Permet à tous les processus temps réel de même priorité de partager le CPU
  - **SCHED_NORM (SCHED_OTHER)**
    - Processus **conventionnel** (temps partagé)

Ordonnancement de processus conventionnels

- **Priorité statique**
  - **Chaque processus conventionnel a une priorité statique (PS)**
    - Utilisée par l'ordonnanceur pour ordonner les **processus conventionnels** entre eux
    - **Valeur de 100** (haute priorité → -20 (nice)) **à 139** (basse priorité → +19)
  - **Un nouveau processus hérite de la priorité de son père**
  - **Quantum de base (QB en ms)**
    - Déterminé à partir de la priorité statique
    - Assigné à un processus qui a épuisé son quantum précédent
      - Si PS < 120, QB = (140-PS)*20
      - Si PS >=120, QB = (140-PS)*5
    - ➔ Un processus de basse priorité aura un quantum faible

Ordonnancement de processus conventionnels

- **Priorité dynamique**
  - **Chaque processus a, en plus une priorité dynamique (PD), de 100 (plus haute) à 139 (plus basse)**
  - **La priorité dynamique sert à l'ordonnanceur pour choisir le processus à exécuter**
    - PD=max(100,min(PS-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'historique du processus**
    - Son **temps moyen de sommeil**
  - **Temps moyen de sommeil:**
    - Mesuré en nanosecondes, jamais plus grand que 1 seconde
    - Mesure différente selon l'état (TASK_INTERRUPTIBLE vs TASK_UNINTERRUPTIBLE)
    - Diminue quand le processus s'exécute

Ordonnancement de processus conventionnels

Temps de sommeil moyen	Bonus
Entre 0 et 100ms	0
Entre 100 et 200ms	1
Entre 200 et 300ms	2
Entre 300 et 400ms	3
Entre 400 et 500ms	4
Entre 500 et 600ms	5
Entre 600 et 700ms	6
Entre 700 et 800ms	7
Entre 800 et 900ms	8
Entre 900 et 1 s	9
1 seconde	10

- **Équivalence sommeil - bonus**
  - **Le temps moyen de sommeil sert à déterminer si le processus est batch ou interactif**
    - Si ***PD ≤ 3 * PS/4 + 28*** alors **interactif**
    - Ce qui est équivalent à

    bonus – 5 ≥ PS/4 – 28
    - ***PS/4-28*** est le **delta interactif**

Ordonnancement de processus conventionnels

Exemple de valeurs de priorités pour des processus conventionnels

Description	PS	Valeur de *nice*	Quantum de base	Delta interactif
Priorité statique la plus haute	100	-20	800ms	-3
Haute priorité statique	110	-10	600ms	-1
Priorité normale	120	0	100ms	+2
Basse priorité	130	+10	50ms	+4
Priorité la plus basse	139	+19	5ms	+6

bonus – 5 ≥ PS/4 – 28

Ordonnancement de processus conventionnels

- **Exemple**
  - **Il est plus facile pour un processus de haute priorité statique d'être interactif.**
  - **Un processus ayant une PS de 100**
    - Est considéré comme interactif si la valeur du bonus >2
      - C'est-à-dire lorsque son temps moyen de sommeil est de 200ms.
  - **Un processus avec une PS de 139 ne peut être considéré comme interactif**
    - ***bonus – 5 ≥ PS/4 – 28*** ⇨ **bonus ≥ *139/4-28+5***

    bonus ≥ 11 ce qui est impossible

Ordonnancement de processus conventionnels

- **Processus actifs et expirés**
  - **Un processus conventionnel de haute priorité ne devrait pas empêcher ceux de basse priorité de tourner**
  - **Quand un processus a fini son quantum, il peut être remplacé par un processus de plus basse priorité qui n'a pas fini le sien**
  - **L'ordonnanceur maintient 2 listes de processus**
    - Processus **actifs** : ils n'ont pas encore fini leur quantum
    - Processus **expirés** : ont épuisé leur quantum et ne peuvent pas s'exécuter tant qu'ils restent des processus actifs
  - **Plus compliqué que cela en pratique:**
    - Un processus batch qui finit son quantum devient **toujours** *expiré*
    - Un processus interactif qui finit son quantum **reste souvent *actif***:
      - Sauf si le plus vieux processus expiré attend depuis « très longtemps »
      - Ou un processus expiré a une priorité statique plus élevée

Ordonnancement de processus conventionnels

- **Résumé**
  - **Priorité statique ↗ : le quantum de temps ↗**
  - **Priorité dynamique ↗ : ordre d'exécution**
    - PS
    - Bonus → temps moyen de sommeil
  - **Interactivité (delta interactif): reste dans la liste des processus actifs**
    - Priorité statique ↗
    - Temps de sommeil ↗

Ordonnancement temps réel

- Chaque processus temps réel a une **priorité temps réel** de 1 (plus basse) à 99, (0 si non temps réel)
- L'ordonnanceur favorise toujours le processus temps réel de plus haute priorité
  - Aucun autre ne peut s'exécuter
- Les processus temps réel **sont toujours considérés comme « actifs »**
- Un processus temps réel est remplacé par un autre processus si:
  - Il est préempté par un autre de plus haute priorité temps réel
  - Il effectue une opération bloquante et est mis en sommeil (`TASK_INTERRUPTIBLE` ou `TASK_UNINTERRUPTIBLE`)
  - Il est stoppé (`TASK_STOPPED` ou `TASK_TRACED`) ou tué (`EXIT_ZOMBIE` ou `EXIT_DEAD`).
  - Il rend volontairement le CPU avec un appel à `sched_yield( )`
  - Il est SCHED_RR et a fini son quantum

Champs relatifs à l'ordonnancement dans le descripteur de processus (*task_struct*)

Type	Nom	Description
unsigned long	thread_info->flags	Enregistre le drapeau TIF_NEED_RESCHED qui est activé si l'ordonnanceur doit être appelé au retour d'une interruption
unsigned int	thread_info->cpu	Numéro logique du CPU auquel appartient la runqueue dans laquelle se trouve le processus
unsigned long	state	L'état courant du processus
int	prio	**Priorité dynamique du processus**
int	static_prio	**Priorité statique du processus**
struct list_head	run_list	Pointeur vers l'élément suivant et précédent dans la runqueue du CPU en question
prio_array_t *	array	Pointeur vers l'ensemble prio_array_t qui inclus le processus
unsigned long	sleep_avg	**Temps de sommeil moyen du processus**
unsigned long long	timestamp	Temps de la dernière insertion du processus dans la runqueue ou temps du dernier changement de processus incluant le processus actuel.
unsigned long long	last_ran	Temps du dernier changement de processus ayant remplacé le processus
int	activated	Code de condition utilisé lorsque le processus est réveillé
unsigned long	policy	Politique d'ordonnancement du processus (SCHED_NORMAL, SCHED_FIFO, ou SCHED_RR)
cpumask_t	cpus_allowed	Masque de bit du CPU qui exécute le processus
unsigned int	time_slice	**Nombre de tics restant dans le quantum du processus**
unsigned int	first_time_slice	drapeau mis à 1 si le processus n'a jamais fini son quantum de temps
unsigned long	rt_priority	**Priorité temps réel du processus**

Université de Bretagne Occidentale

La norme POSIX

Cours 2 : Services des systèmes d'exploitation

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. **Ordonnancement**
4. Communication interprocessus et synchronisation
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Hai Nam TRAN
UBO – M2 LSE

Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

La norme POSIX

- 1ère version publiée entre 1988 et 1990
- **But:** Définition d'une méthode standard pour interfacer les applications avec les systèmes d'exploitation.
- L'ensemble inclut actuellement plus de 30 standards couvrant plusieurs sujets (gestion des processus, …, test de conformité de l'OS avec la norme).
- Standard concernant les interfaces du système: *(volume XSH) de l'IEEE Std 1003.1-2001* définissant un standard pour l'interface du système d'exploitation et l'environnement. Ce standard inclut une extension temps réel:
  - Fonctions et routines des services système
  - Services système spécifiques au langage C.
  - Notes sur la portabilité, gestion d'erreurs et récupération.

La norme POSIX

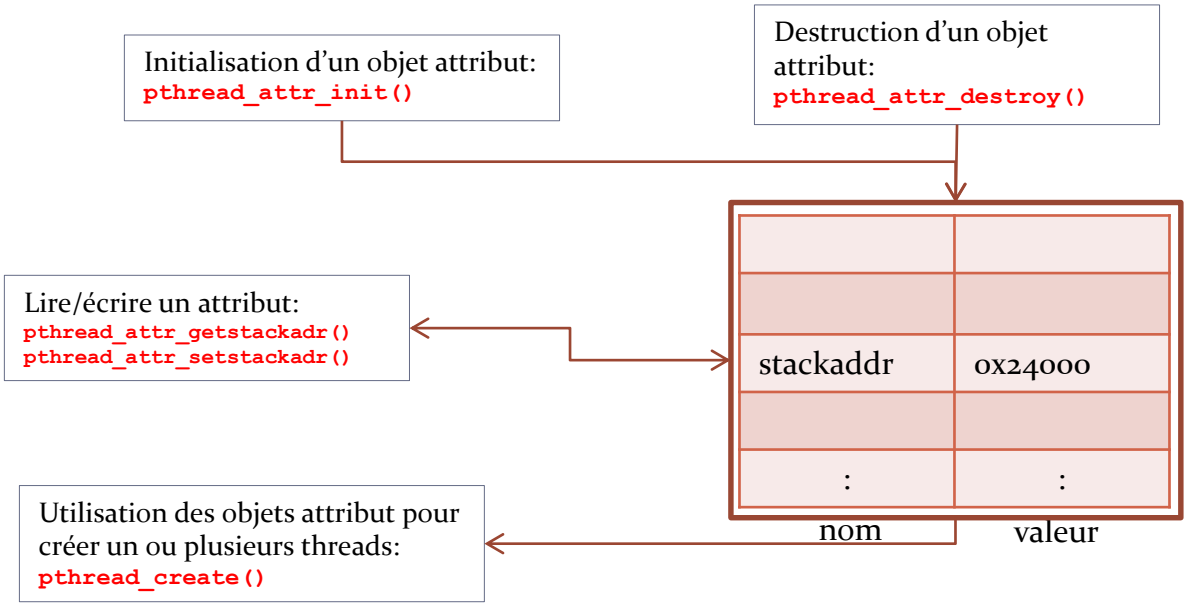
Norme	Description
1003.1b	Extensions temps réel basiques (à partir de 1993)
1003.1c	Extensions pour les threads (1995)
1003.1d	Extensions temps réel additionnelles (1999)
1003.1j	Extensions temps réel avancées (2000)

La norme POSIX

Groupe fonctionnel	Quelques fonctions principales
Multi-Threads	pthread_create, pthread_exit, pthread_join, pthread_detach, pthread_equal, pthread_self
Ordonnancement de processus et de Threads	sched_setscheduler, sched_getscheduler, sched_setparam, sched_getparam, pthread_getschedparam, pthread_setschedprio, …
Signaux temps réel	sigqueue, pthread_kill, sigaction, sigaltstack, sigemptyset, etc
Synchro. et comm. inter processus	mq_open, mq_close, mq_receive, sem_init, sem_open, sem_wait, pthread_mutex_init, pthread_mutex_destroy, pthread_cond_init, shm_open, shm_unlink, mmap, etc.
Données spécifiques aux threads	pthread_key_create, pthread_get_specific, etc.
Gestion Mémoire	mlock, mlockall, munlock, munlockall, mprotect
E/S Async	aio_read, aio_write, aio_error, aio_return, aio_fsync, etc.
Horloges et minuteries (*timers*)	clock_gettime, clock_settime, clock_getres, times_create, timer_gettime, etc.
Annulation	pthread_cancel, pthread_setcancelstate, pthread_cleanup_push, pthread_cleanup_pop

Les objets attributs

- **Mécanismes conçus pour supporter les normalisations futures ainsi que des extensions portables de quelques entités spécifiées par le standard POSIX sans que les fonctions qui opèrent sur ces dernières ne changent; threads, dispositifs d'exclusion mutuelle, variables de conditions.**
- **Centralisation de certains attributs par type d'objets et non pas par instance.**



Multithreading

- **Création de thread pour l'exécution d'une fonction `pthread_create()`**

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

  - **Possibilité de passer un argument à la fonction exécutée par le thread (*`arg`*).**
  - **Possibilité (optionnel) de spécifier un objet attribut (taille et adresse du *stack*, politique d'ordonnancement) → *`*attr`***
  - **Terminaison du thread:**
    - Retour à/de la fonction principale
    - Appel explicite à `void pthread_exit(void *retval);`
    - Acceptation d'une requête d'annulation

Multithreading

- `int pthread_join(pthread_t thread, void **value_ptr)`: attente de la terminaison d'un autre thread
- Détachement d'un thread: `int pthread_detach(pthread_t th);` récupération des ressources à la terminaison
- `pthread_t pthread_self(void):` retourne l'identifiant d'un thread

Exemple

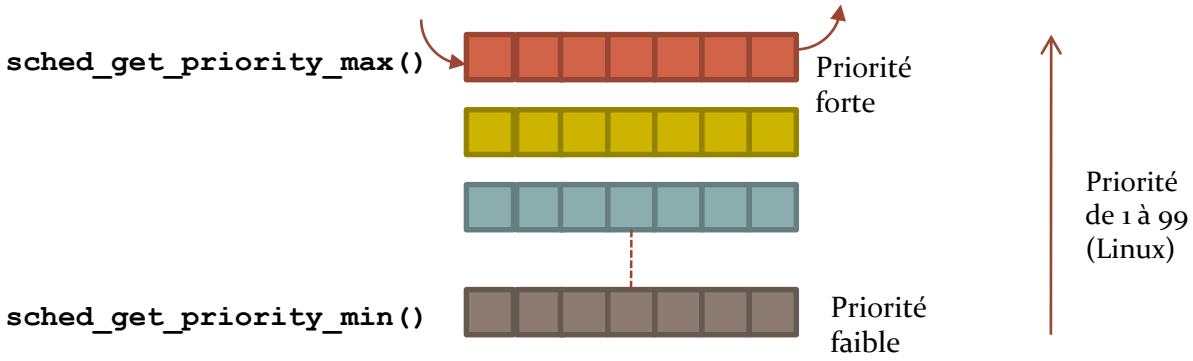
```
int sum;//donnée partagée par les threads

void *runner(void *param); //prototype du thread
main(int argc, char *argv[])
{
   char *val_ret;
   int arg_int=8;
   pthread_t tid; // l'identifiant du thread
   pthread_attr_t attr; //ensemble d'attributs d
   thread
   // Avoir les attributs par défaut (optionnel)
   pthread_attr_init(&attr);
   // création du thread
   pthread_create(&tid,&attr,runner,(void *)
   &arg_int);
   // attente que le thread se termine
   pthread_join(tid, (void *)&val_ret);
   printf("%s →sum = %d\n",(char *) &val_ret,
   sum);
}
```

```
void *runner(void *param) {
   int upper = *(int *) param;
   int i;
   sum = 0;
  char *c;
   if (upper > 0) {
      for (i = 1; i <= upper; i++)
        sum += i;
   }
   c= (char *) malloc(3*sizeof(char));
   c=strdup("ok");
  // terminaison du thread
  pthread_exit(c);
}
```

Ordonnancement de threads et de processus

- **L'interface POSIX propose trois politiques différentes,**
  - **Une pour les processus classiques (conventionnels) temps partagé,**
  - **Deux pour les applications à vocation temps-réel.**
  - **Une valeur de priorité statique *sched_priority* est assignée à chaque processus (modif: appels sys.)**
  - **Conceptuellement, l'ordonnanceur dispose d'une liste de tous les processus prêts pour chaque valeur possible de *sched_priority* (intervalle 0 à 99).**



Ordonnancement de threads et de processus (2)

- **Politiques d'ordonnancement:**
  - **SCHED_FIFO**
    - Priorités statiques > 0 (privilèges super utilisateur)
    - Arrêt: bloqué par une opération d'entrée/sortie OU préempté par un processus de priorité supérieure OU appel ***`sched_yield`***.
  - **SCHED_RR (Round Robin)**
    - Priorités statiques > 0
    - Quantum de temps (lecture avec **`sched_rr_get_interval`**)
  - **SCHED_OTHER**
    - Priorités statiques =0
    - Utilisation d'une priorité dynamique basée sur la valeur de « *gentillesse* » du processus (**nice**, **setpriority**) → incrémentée à chaque *time quantum* où le processus est prêt mais non sélectionné par l'ordonnanceur. Garantit d'une progression équitable de tous les processus *SCHED_OTHER*.

Ordonnancement de threads et de processus (3)

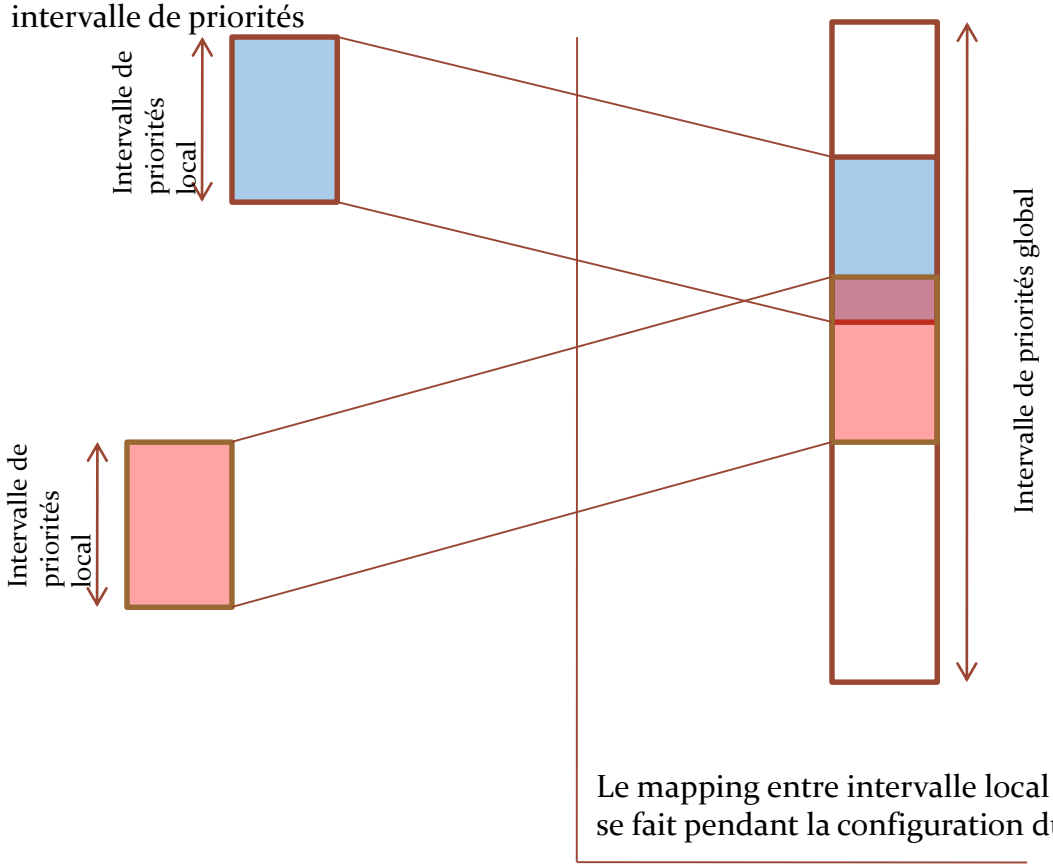
- **Fonctions permettant de sélectionner la politique d'ordonnancement, configurer et lire les paramètres de cette dernière.**
- **Lire / fixer la *politique* d'ordonnancement des processus et ses paramètres**
  - `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p),`
  - `int sched_getscheduler(pid_t pid)`
  - `struct sched_param { ... int sched_priority; ... };`
- **Lire / écrire les *paramètres* d'ordonnancement (la priorité)**
  - `int sched_setparam(pid_t pid, const struct sched_param *p)`
  - `int sched_getparam(pid_t pid, struct sched_param *p).`

Ordonnancement de threads et de processus (4)

- `int pthread_setschedparam(pthread_t thread_cible, int politique, const struct sched_param *param),`
- `int pthread_getschedparam(pthread_t thread_cible, int *politique, struct sched_param *param);`
- `int pthread_setschedprio(pthread_t thread_cible, int prio)` **(accès dynamique; utilisé avec un thread existant)**
- **Modification dans l'objet attribut (et non pas pour un thread particulier) → utilisable pour la création d'autres threads:**
  - `int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);`
  - `int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);`
  - `int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);`
  - `int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);`
- **Politiques d'ordonnancement: FIFO, RR, ou autre.**

Ordonnancement sous POSIX

Chaque politique d'ordonnancement contrôle le placement des threads dans son intervalle de priorités



Sélection du thread avec la priorité la plus importante par l'ordonnanceur.
Les threads sont remis dans leur liste lors de la préemption

Le mapping entre intervalle local et global se fait pendant la configuration du système

Appels système relatifs à l'ordonnancement

- **Conventionnels**
  - **`nice()`: changement de la valeur de « gentillesse ». Maintenu pour des raisons de compatibilité et remplacé par `setpriority`()**
    - Valeurs négatives: augmentation de priorité statique (superutilisateur)
  - **`getpriority()` et `setpriority()`: concerne un processus ou tout le groupe de processus (priorité statique max).**
  - **`sched_getaffinity()` et `sched_setaffinity()`: quels sont les CPUs qui peuvent exécuter ce processus.**
- **Temps réel**
  - **`sched_getscheduler()` et `sched_setscheduler()`: politique d'ordonnancement**
  - **`sched_ getparam()` et `sched_ setparam()`: retourne les paramètres de la politique d'ordonnancement (priorité temps réel).**
  - **`sched_yield()`: relâche le CPU.**
  - **`sched_get_priority_min()` et `sched_get_priority_max()`: retourne la priorité min et max utilisable par le processus.**
  - **`sched_rr_get_interval()`: retourne la taille du quantum de temps de la politique du tourniquet.**

Université de Bretagne Occidentale

Communication et synchronisation interprocessus Introduction

Cours 2 : Services des systèmes d'exploitation

1. Gestion des fichiers
2. Gestion des tâches, processus, threads
3. Ordonnancement
4. **Communication et synchronisation interprocessus**
5. Gestion de la mémoire
6. Gestion des interruptions
7. Entrées/Sorties et pilotes de périphériques
8. Protocoles de communication

Hai Nam TRAN
UBO - M2 LSE

Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

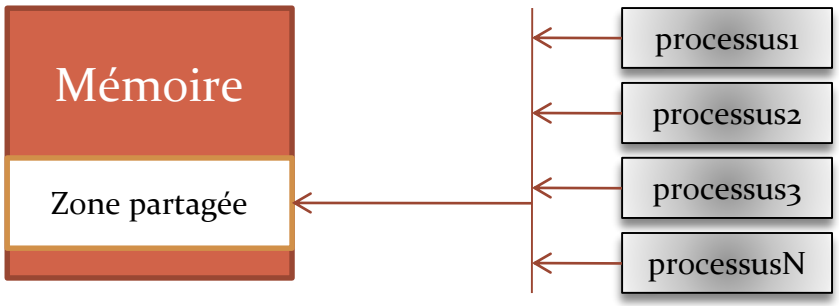
Communication et synchronisation interprocessus

- **Les tâches (dans un système multi tâches) dans un OS embarqués ont AUSSI besoin de communiquer et se synchroniser.**
  - **Communication: implique au moins 2 tâches (receveur et émetteur, lecteur et rédacteur ou producteur et consommateur)**
    - Communiquer efficacement sans avoir à connaître grand-chose de l'autre
  - **Synchronisation: pas d'ordre ou de direction**
    - Être sûr que les tâches se trouvent (ou pas) à des endroits spécifiques dans leur code respectif en même temps.
  - **Une communication utile n'a lieu que si la synchronisation est efficace**
  - **Communication/synchronisation:**
    - Bloquante → communication synchrone
    - Non bloquante → communication asynchrone
    - Variantes : Blocage pendant une durée max (timeout), Blocage conditionnel

Inter Process Communication (IPC); propriétés

- **Couplage**
  - **Connexions nommées : Les tâches émettrices et celles réceptrices doivent se connaître et s'adresser par leurs identifiants respectifs.**
    - Un à un, un à plusieurs , plusieurs à un ou plusieurs à plusieurs.
  - **Diffusion (*broadcast*) : l'émetteur envoie son message à tous ceux qui « écoutent » et il ne sait pas quelle tâche l'a écouté. Ecoute qui le veut. Les messages sont perdus s'ils ne sont pas écoutés.**
  - **Tableau noir (*blackboard*) : similaire à la diffusion sauf que les messages sont sauvegardés quelque part.**
  - **ORB (Object Request Broker) : l'émetteur enregistre son interface avec l'ORB et les receveurs intéressés peuvent demander au « broker » de transférer leur requêtes au serveur appropriés sans avoir à connaître son identité.**
- **Mise en tampon**
  - **Avec : mise en tampon des messages**
  - **Sans : si message non reçu à l'envoi, il est perdu**

Accès concurrent à une zone partagée




```
data number_1;
data number_2;
task A
{           data A_number;
            A_number = read(number_1);
            A_number = A_number + 1;
            write(number_1,A_number);
}
task B
{ if ( read(number_1) == read(number_2) )
            do_something();
  else
            do_something_else();

}
```


- **Problème d'accès concurrent (*race condition*)**
- **Processus préempté avant d'avoir fini un accès en écriture**
  - **Intégrité du système n'est plus garantie**
- **L'accès est alors une section critique**
- **On doit accéder à la zone en exclusion mutuelle**
  - **L'exclusion mutuelle permet à un processus de verrouiller l'accès à une ressource pendant que lui y accède.**
  - **Technique de synchronisation.**

Sections critiques

- **3 types de sections critiques :**
  - **Accès aux mêmes *données* par des tâches différentes**
  - **Accès à un *service* : allocation d'une ressource**
  - **Accès à un code de *procédure* : quelques portions de code ne doivent pas être réentrantes.**
- **Les problèmes d'accès concurrent:**
  - **Interblocage (*deadlock*) toutes les ressources sont prises et toutes les tâches ont besoin de plus de ressources pour poursuivre leur exécution.**
  - ***Livelock :* les tâches ne sont pas bloquées mais demandent en continu des ressources qu'elles n'obtiennent pas.**
  - **Famine (*starvation*) : une tâche ne réussi jamais à obtenir la ressource qu'elle demande.**

Conditions pour un inter blocage

1. **L’accès aux ressources se fait en exclusion mutuelle**
2. **Une tâche qui a une ressource peut en demander une nouvelle**
3. **Une tâche qui a une ressource est la seule à pouvoir la libérer**
4. **Réalisation d’une boucle avec les demandes et les obtention de ressource.**

- **Verrouillage assisté par le processeur**
  - Les tâches qui accèdent à des données partagées sont ordonnancées de telle sorte à ne pas être interrompues
  - … sauf par des interruptions
  - On peut aussi désactiver les interruptions pour éviter les problèmes d'accès concurrent entre la fonction/*handler* d'interruption et la tâche

```
FuncA ()
{
int lock = intLock ();
.
.section critique ne pouvant être interrompue
.
intUnlock (lock);
}
```

- **Test and set lock: appelé aussi variable de condition (matérielle)**
  - **Le drapeau d'un registre est testé et modifié en une seule opération atomique → événement non interruptible.**
  - **Ce drapeau est testé par n'importe quel tâche voulant accéder à une section critique.**

```
int test_and_set(int *lock){
    int temp = *lock;
    *lock = 1; // je modifie la valeur du verrou (ou non)
    return temp; // je retourne l'ancienne valeur
}
```

- ***Test and set lock* et variable de verrouillage:**
  - **Le système ne peut répondre à aucun événement pendant la durée de l'accès à la section critique (monopole du système)**

Les types de verrou

- **Les sémaphores**
  - **Introduit par Dijkstra 1965**
  - **Variables entières**
    - Sémaphores binaires: 0 ou 1
    - Sémaphores généraux: 0..nbRessources
    - Deux opération wait (P) and signal (V)
  - **Verrouillage de l'accès à une (ou plusieur) ressource partagée**
  - **Coordination de processus avec des événements extérieurs ou entre eux.**
  - **Operations sur les sémaphores: atomiques**
    - Invoqués via des appels systè

```
function V(semaphore S, integer I):
    [S ← S + I]

function P(semaphore S, integer I):
    repeat:
        [if S ≥ I:
        S ← S - I
        break]
```

Les types de verrou

- **Les spinlocks**
  - Verrou approprié pour les systèmes multiprocesseurs.
  - Peuvent être utilisés dans tous les contextes (appel système, routine d'interruption, etc.)
  - Attente active (!!!)
- **Verrou en lecture/écriture**
  - Ce qui est critique, c'est surtout l'écriture concurrente.
  - Plusieurs verrous peuvent être posés en lecture pour une ressource donnée
  - Mais un seul en écriture.
- **Barrière**
  - Synchro d'un ensemble de tâches/processus /threads
  - RDV entre plusieurs threads
  - Barrière initialisée avec le nombre de threads à attendre, à chaque arrivée, le compteur est décrémenté.

Definition : polling - attente active

- **Vérification périodique d'une condition ou d'une valeur**
  - **Utiliser de temps processeurs en attendant de pouvoir entrer dans la section critique**

Les types de verrou : Variable de condition

- **Les variables de conditions permettent:**
  - **À une tâche de se mettre en sommeil jusqu'à ce qu'un certain critère logique définit par une application devienne vrai.**
  - **Permet à une tâche de se mettre en sommeil dans une section critique (différent des sémaphores)**
- **Solution consiste à combiner:**
  1. **Un sémaphore binaire mutex**
  2. **Une expression booléenne qui représente la condition logique.**
  3. **Un signal que les autres tâches utilisent pour réveiller la tâche bloquée pour qu'elle puisse revérifier la variable de condition.**
- **Détaillé plus tard …**

Les IPCs… quelques politiques

- Avec perte de données ou pas
  - Est ce que tout ce qu'envoie une tâche va être reçu (par la tâche ou le médiateur)?
- Bloquant ou pas.
- Couplage
  - Un à un, un à plusieurs, plusieurs à un, plusieurs à plusieurs.
- Nommés/anonymes
  - L'émetteur doit-il donner l'identifiant des receveurs?
- *Bufferisé* ou pas:
  - Envoi direct ou indirect ?
- Avec ou sans priorité

Les différents IPCs

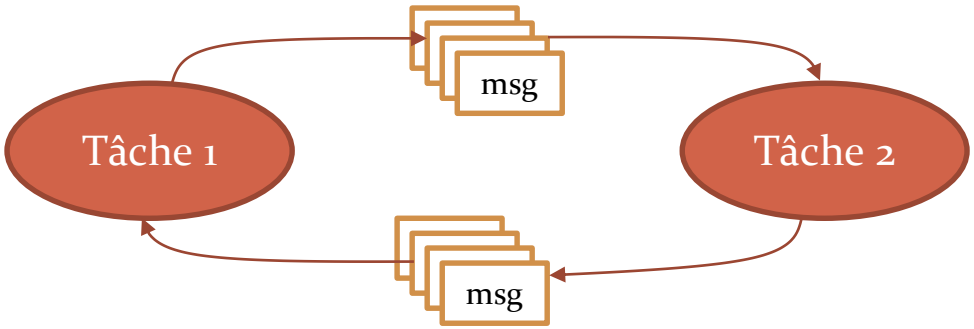
- **Mémoires partagées**
  - **pas de copie « en trop » (pas de médiateur, non bufferisé)**
  - **Synchronisation doit être explicite**
  - **Limité par la taille de la RAM**
- **FIFO**
  - **Bufferisé/un à un (généralement)**
  - **Pas de synchronisation à faire.**

Les différents IPCs

- **Messages et boîtes aux lettres**
  - **Communication de messages de tailles arbitraires**
  - **En plus des données effectives, un message contient des « méta données » concernant la structure du message.**
  - **Messages: création du message dans l'espace de la tâche, envoi de l'adresse et de l'identifiant du receveur à l'OS. Le receveur demande le message à l'OS.**
  - **Boîte aux lettres: l'émetteur notifie l'OS qu'il a un message, ce dernier copie le message dans la boîte aux lettres de la tâche réceptrice.**

Les différents IPCs

- **Files de messages**
  - **Messages envoyés via des files de messages.**
  - **L'OS définit:**
    - Le protocole de communication
    - La méthode d'authentification
    - Tailles (de la file et du message)
    - *Les micronoyaux font un usage intensif de ce mécanisme pour la synchronisation.*



Les différents IPCs

- **Tampon/Buffer circulaire: même chose que la mémoire partagée mais avec un tableau d'emplacements circulaire:**
  - **Lecture plus rapide que l'écriture → lecture de la même donnée**
  - **Écriture plus rapide → écrasement de données non lues**
- **Swinging buffer**
  - **Un tampon circulaire avancé: utilisation de plusieurs tableaux circulaires**
  - **Verrou sans inter blocage possible: lecture et écriture sur des tableaux différents.**
- ***RPC (Remote Procedure calls):***
  - **Invoquer des exécutions de tâches distantes: basé sur l'envoi/réception de messages.**
  - **plus haut niveau que les autres IPC.**

Les différents IPCs

- **Signal**
  - **Asynchrone**
    - La tâche qui envoie et celle qui reçoit le signal sont dans des états indépendants et ne partagent pas de mémoire → mécanisme de synchronisation
    - La tâche envoie un signal et poursuit son exécution, c'est l'OS qui délivre le signal et si personne n'en veut … tant pis !
  - **Généralement, les signaux:**
    - ne sont pas mis en file d'attente
    - ne transportent pas de données
    - temps de livraison est non déterministe (tâche réceptrice non ordonnancée de suite)
    - ordre de livraison non déterministe (plusieurs signaux du même type reçus)
    - sauf les signaux temps réel …. on verra cela plus tard

Les différents IPCs

- **Signal**
  - **A la réception d'un signal:**
    - Suspension de l'exécution actuelle et exécution d'une autre fonction (*handler*)
    - Dans le contexte de la tâche actuelle
  - **Signaux généralement utilisés pour la gestion des interruptions dans un OS (nature asynchrone) … implémentés de telle sorte qu'ils ne peuvent causer un inter blocage ou un blocage du *handler*.**
  - **Exceptions:**
    - Signal synchrone, levé par une tâche pour son propre compte (exécution anormale, ex: division par zéro) → exécution d'un *handler* (ex: message d'erreur + terminaison).

Université de Bretagne Occidentale

Communication et synchronisation interprocessus Implémentation

Cours 2 : Services des systèmes d'exploitation

Hai Nam TRAN
UBO - M2 LSE

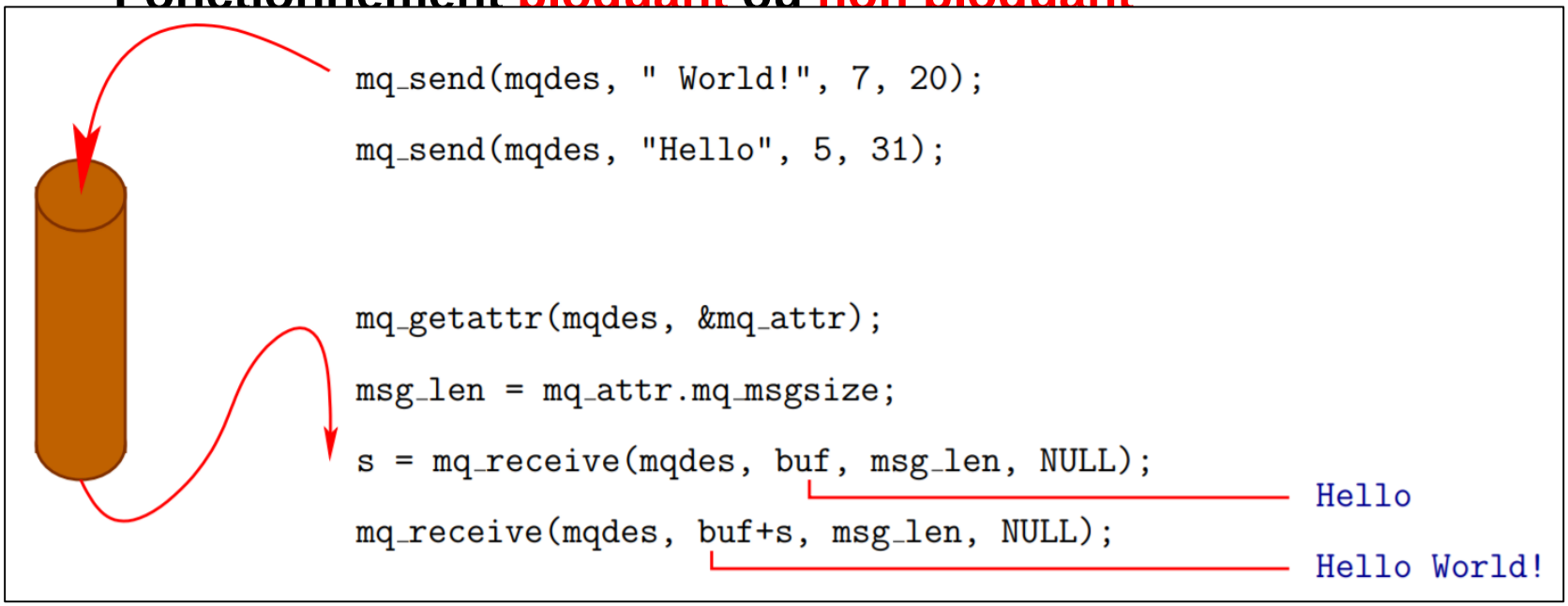
Le CM est inspiré du cours précédemment dispensé par Jalil Boukhobza

Synchronisation et communication inter processus

- **Files de message**
- Mémoire partagé
- E/S asynchrones
- Sémaphores
- Mutex
- Variables de condition

Files de messages

- **Quelques caractéristiques:**
  - **Priorité à l'émission / réception**
  - **Fonctionnement bloquant ou non bloquant**



`#include <mqueue.h>`	`/dev/mqueue`	`Compile and link : -lrt`

Files de messages : mq_open()

```
mqd_t mq_open(const char *name, int oflag, ...)
```

- **Création ou ouverture d'une file de message retournant un descripteur utilisé par toutes les autres fonctions.**
- **1 : const char *name**
  - **Nom de la MQ. Le nom DOIT commencer par un slash « / », et contenir une chaîne alphanumérique**
  - **Si d'autres slashs « / » sont présents dans le nom (ou aucun), POSIX ne donne aucune consigne, c'est à l'OS de décider comment réagir**
- **2 : `int oflag`**
  - **O_RDONLY : lecture** seulement - mq_receive
  - **O_WRONLY : écriture** seulement - mq_send
  - **O_RDWR : lecture** et en **écriture** - mqsend, mqreceive
  - **O_CREAT :** créer une MQ.
  - **O_EXCL :** avec O_CREAT, il permet d'empêcher l'ouverture de la file si celle-ci existe
  - **O_NONBLOCK :** permet de dire si mq_send() et mq_receive() doivent attendre que les ressources nécessaires soient disponibles ou non

Files de messages : mq_open()

- **Avec le flag O_CREAT, la fonction prend 2 autres paramètres (soit 4 au total)**
  - mode_t mode : droit d'accès (en nombre chmod)
  - mq_attr *attr : structure spécifique à la file

```
struct mq_attr
{
  long mq_flags;   /* Flags de la file */
  long mq_maxmsg;  /* Nombre maximum de messages dans la file */
  long mq_msgsize; /* Taille maximale de chaque message */
  long mq_curmsgs; /* Nombre de messages actuellement dans la file */
};
```

Files de messages : mq_send()

```
int mq_send(mqd_t, const char *, size_t, unsigned);
```

- L'écriture dans la file se fait en envoyant un message
- La fonction renvoie 0 en cas de succès, sinon -1

- 1 : `mqd_t`
  - Le descripteur de la MQ (on l'a reçu lors du mq_open())
- 2 : `const char *`
  - Le message à envoyer
- 3 : `size_t`
  - La taille du message envoyé.
- 4 : `unsigned`
  - La priorité avec laquelle on souhaite envoyer le message.
    - Par défaut, il est conseillé de mettre une priorité de 0 pour les messages classiques, et plus si l'information à transmettre n'est pas prévue dans le cadre du fonctionnement normal

Files de messages : mq_receive()

```
ssize_t mq_receive(mqd_t, char *, size_t, unsigned *);
```

- **La lecture (destructrice) d'un message (**celui de plus haute priorité présent dans la file, et le plus ancien)
- **La valeur de retour est le nombre de caractères lus, ou -1 en cas d'échec.**

- **1 : `mqd_t`**
  - **Le descripteur de MQ (récupéré avec mq_open())**
- **2 : `char *`**
  - **Un buffer de taille suffisante pour réceptionner le message**
  - **Si le buffer est trop petit, le message ne sera pas lu ni détruit**
- **3 : `size_t`**
  - **La taille du buffer.**
  - **Si cette taille est inférieure à la taille donnée à la file à sa création (via mq_attr.mq_msgsize), alors l'appel échoue.**
- **4 : `unsigned *`**
  - **Si l'argument n'est pas NULL, mq_receive() va écrire dedans la priorité du message réceptionné → Créer une variable locale et la passer par référence à la fonction**

Files de messages : mq_notify()

```
int mq_notify(mqd_t, const struct sigevent *);
```

- **notifie lors de la transition file de msg vide → file de msg non vide (**quand une file vide commence à être alimentée)
- **La fonction renvoie l'entier 0 si elle a réussi, sinon -1**

- **1 : `mqd_t`**
  - **Le descripteur de la file que l'on souhaite scruter**
- **2 : `const struct sigevent *`**
  - **Un pointeur sur structure sigevent (à explorer)**
  - **Le mécanisme de notification est lié aux signaux**

Files de messages : mq_close()

```
int mq_close(mqd_t);
```

- La fermeture d'une file ne sert qu'à retirer le descripteur de file associé au processus
- La file n'est PAS détruite suite à mq_close()
- Le retour indique seulement si la fermeture a réussi (0) ou non (-1)

- 1 : `mqd_t`
  - **Le descripteur de MQ (récupéré avec mq_open())**

Files de messages : mq_unlink()

```
int mq_unlink(const char *);
```

- Supprimer une file de message

- 1 : `const char *`
  - Nom de la file

Files de messages : mq_getattr()

```
int mq_getattr(mqd_t, struct mq_attr *);
```

- Récupérer les propriétés de la file

- 1 : `mqd_t`
  - le descripteur de la file
- 1 : `struct mq_attr *`
  - un pointeur sur une structure mq_attr

```
struct mq_attr
{
  long mq_flags;   /* Flags de la file */
  long mq_maxmsg;  /* Nombre maximum de messages dans la file */
  long mq_msgsize; /* Taille maximale de chaque message */
  long mq_curmsgs; /* Nombre de messages actuellement dans la file */
};
```

Files de messages : mq_setattr()

```
int mq_setattr(mqd_t, const struct mq_attr *, struct
mq_attr *);
```

- modifier **certains** paramètres de la file lors du fonctionnement du programme

- 1 : `mqd_t`
  - **Le descripteur de la file**
- 2 : `const struct mq_attr *`
  - **Un pointeur sur structure mq_attr, contenant les nouvelles valeurs**
- 3 : `const struct mq_attr *`
  - **Un pointeur sur structure mq_attr sert à sauvegarder l'état précédent de la file (peut être mis à NULL)**

Files de messages : Exemple (1)

Emetteur

```
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

int main(int argc,char * argv[])
{
    mqd_t mq;
    struct timeval heure;

    mq = mq_open("/mymq", O_WRONLY | O_CREAT, 0600, NULL);
    if (mq == (mqd_t) -1) {
        perror("/mymq"); //prints an error message to stderr
        exit(EXIT_FAILURE);
    }
    while (1) {
        gettimeofday(& heure, NULL);
        mq_send(mq, (char *) & heure, sizeof(heure), 1);
        sleep(1);
    }
    return EXIT_SUCCESS;
}
```

https://www.blaess.fr/christophe/2011/09/17/efficacite-des-ipc-les-files-de-messages-posix/

Files de messages : Exemple (1)

```
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
```

```
int main(int argc,char * argv[])
{
    mqd_t mq;
    int taille;
    char * buffer;
    long int duree;
    struct mq_attr attr;
    struct timeval heure;
    struct timeval * recue;
```

```
    mq = mq_open(("/mymq", O_RDONLY | O_CREAT, 0600, NULL);
    if (mq == (mqd_t) -1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    if (mq_getattr(mq, & attr) != 0) {
        perror("mq_getattr");
        exit(EXIT_FAILURE);
    }
    taille = attr.mq_msgsize;
    buffer = malloc(taille);

    if (buffer == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    recue = (struct timeval *) buffer;
    while (1) {
        mq_receive(mq, buffer, taille, NULL);
        gettimeofday(& heure, NULL);
        duree  = heure.tv_sec - recue->tv_sec;
        duree *= 1000000;
        duree += heure.tv_usec - recue->tv_usec;
        fprintf(stdout, "%ld usec\n", duree);
    }
    return EXIT_SUCCESS;
}
```

Récepteur

Files de messages : Exemple (2)

```
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#define MSG_SIZE      4096

// Cette fonction est appelée lorsque la file devient non vide
void handler (int sig_num) {
   printf ("sig %d recu.\n", sig_num);
}

void main () {
 struct mq_attr attr, old_attr; // sauvegarde des attr de la file
 struct sigevent sigevent;      // pour notification
 mqd_t mqdes, mqdes2;           // descripteur de file de msg
 char buf[MSG_SIZE];            // taille du buffer
 unsigned int prio;             // Priority

 // First we need to set up the attribute structure
 attr.mq_maxmsg = 300;
 attr.mq_msgsize = MSG_SIZE;
 attr.mq_flags = 0;

 // Open a queue with the attribute structure, les 2 derniers
      paramètres ne sont pas obligatoires
 mqdes = mq_open ("/lafile", O_RDWR |O_CREAT,
      0664, &attr);

 // ouverture d'une file avec les attr par défaut
 mqdes2 = mq_open (" /autrefile", O_RDWR | O_CREAT,
            0664, 0);

 // une file temporaire, dès qu'elle sera fermée, elle sera
      supprimée
 mq_unlink (" autrefile ");

 // demader les attr de la file de msg lafile
 mq_getattr (mqdes, &attr);
 printf ("%d messages dont actuellement dans la file.\n",
      attr.mq_curmsgs);
if (attr.mq_curmsgs != 0) {

  // Il existe des msg dans la file....à consommer
    // spécifier que la file ne se bloque à aucun appel
  attr.mq_flags = O_NONBLOCK;
  mq_setattr (mqdes, &attr, &old_attr);

  // consommer tous les messages
  while (mq_receive (mqdes, &buf[0], MSG_SIZE, &prio)
      != -1)
   printf ("Received a message with priority %d.\n", prio);

  // l'appel a échoué, s'assurer que errno est EAGAIN
  if (errno != EAGAIN) {
   perror ("mq_receive()");
   _exit (EXIT_FAILURE);
  }

 // restaurer les attributs
   mq_setattr (mqdes, &old_attr, 0);
 }

 // notifier lorsque quelque chose est dans la file
 signal (SIGUSR1, handler);
 sigevent.sigev_signo = SIGUSR1;

 if (mq_notify (mqdes, &sigevent) == -1) {
  if (errno == EBUSY)
   printf (
    « un autre processus veut recevoir cette
notif.\n");
  _exit (EXIT_FAILURE);
 }

 for (prio = 0; prio <= MQ_PRIO_MAX; prio +=
8) {
  printf (« ecrire un message avec une priorité
%d.\n", prio);
  if (mq_send (mqdes, "I8-)", 4, prio) == -1)
   perror ("mq_send()");
 }

 // Fermeture de tous les descripteurs
 mq_close (mqdes);
 mq_close (mqdes2);

}
```

Synchronisation et communication inter processus

- Files de messages
- **Mémoire partagé**
- E/S asynchrones
- Sémaphores
- Mutex
- Variables de condition

Mémoire partagée

- **Usage**
  1. **Créer ou ouvrir un objet de mémoire partagée avec shm_open()**
     - Un descripteur de fichier (**fd**) sera retourné si shm_open() crée avec succès un objet de mémoire partagée.
  2. **Configurer la taille de l'objet de mémoire partagée avec ftruncate()**
  3. **Crée une nouvelle projection dans l'espace d'adressage virtuel du processus avec mmap() et MAP_SHARED**
  4. **Lire / écrire la mémoire partagée**
  5. **Unmap la mémoire partagée munmap()**
  6. **Supprimer l'objet de mémoire partagée avec close()**
  7. **Détruit l'objet de mémoire partagée avec shm_unlink()**

```
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
```

```
/dev/shm
```

```
Compile and link : -lrt
```

Mémoire partagée : `shm_open()`

```
int shm_open(const char *nom, int oflag, mode_t mode)
```

- Crée et ouvre un nouvel objet de mémoire partagé POSIX, ou ouvre un objet existant
- Si elle réussit, la fonction **shm_open**() renvoie un nouveau descripteur décrivant l'objet de mémoire partagée

- **1 : `const char *nom`**
  - Un objet mémoire partagé doit être identifié par un nom au format */un_nom*
- **2 : `int oflag`**
  - O_RDONLY
  - O_RDWR
  - O_CREAT
  - O_EXCL
  - O_TRUNC
- **3 : `mode_t mode`**
  - **Spécifie les permissions**
  - **`https://man.developpez.com/man2/open/`**

Mémoire partagée : `ftruncate()`

```
int ftruncate(int fd, off_t length);
```

- Tronquer un fichier à une longueur donnée

- **1 : `int fd`**
  - **Le descripteur de fichier**
- **2 : `off_t length`**
  - **Une longueur d'exactement length octets**

Mémoire partagée : `mmap()`

```
void *mmap(void *addr, size_t length, int prot, int flags,
int fd, off_t offset)
```

- Établi une projection en mémoire (map/unmap) des fichiers ou des périphériques
- Crée une nouvelle projection dans l'espace d'adressage virtuel du processus appelant

- **1 : `void *addr`**
  - L'adresse de démarrage de la nouvelle projection
  - `NULL` : le noyau choisit l'adresse à laquelle démarrer la projection
- **2 : `size_t length`**
  - La longueur de la projection
- **3 : `int prot`**
  - La protection que l'on désire pour cette zone de mémoire
- **4 : `int flags`**
  - Détermine si les modifications de la projection sont visibles depuis les autres processus projetant la même région
- **5 : `int fd`**
  - descripteur de fichier
- **6 : `off_t offset`**
  - *offset* dans le fichier (doit être un multiple de la taille de page)

Mémoire partagée : `munmap()`

```
int munmap(void *addr, size_t length);
```

- Supprime une projection en mémoire des fichiers ou des périphériques

- **`1 : void *addr`**
  - L'adresse de démarrage de la projection
- **`2 : size_t length`**
  - La longueur de la projection

Mémoire partagée : `shm_unlink()`,`shm_close()`

```
int shm_unlink(const char *nom);
```

- Détruit l'objet de mémoire partagée à condition qu'il n'y ai aucun processus utilisant cette mémoire

```
int close(int fd)
```

- Supprime l'association entre le descripteur et la mémoire partagée associée

Mémoire partagée : Exemple

```
#i
#i
#i
#i
#i
#i
int main() {
    const int SIZE = 4096; /* the size (in bytes) of shared memory object */
    const char* name = "OS"; /* name of the shared memory object */
    const char* message_0 = "Hello";
    const char* message_1 = "World!";

    int shm_fd; /* shared memory file descriptor */
    void* ptr; /* pointer to shared memory object */

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
    sprintf(ptr, "%s", message_0); /* write to the shared memory object */

    ptr += strlen(message_0);
    sprintf(ptr, "%s", message1);
    ptr += strlen(message_1);
    return 0;
}
```

/dev/shm **Producer**

Mémoire partagée : Exemple

```
int main() {
    const int SIZE = 4096;
    const char* name = "OS";
    int shm_fd;
    void* ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char*)ptr);

    munmap(ptr, SIZE);
    close(shm_fd)
    shm_unlink(name);
    return 0;
}
```

Consommateur

Mémoire partagée

- **Gestion de la mémoire**
  - **Possibilité de verrouiller des parties de l'espace d'adressage → `int mlock(const void *addr, size_t len)` : verrouillage à partir d'une adresse et d'un offset et `int mlockall(int flags)` : tout l'espace.**
  - **Verrouillage: force la présence des données dans la mémoire centrale et leur évite d'être « flushées » sur le stockage secondaire:**
    - Essentiel pour le temps réel → prédictibilité + performance
  - **Déverrouillage de la mémoire avec `int munlock(const void *addr, size_t len)` et `int munlockall(void)`.**
  - **Protéger des emplacements mémoire → `int mprotect(const void *addr, size_t len, int prot)`**

Synchronisation et communication inter processus

- Files de messages
- Mémoire partagé
- **E/S asynchrones**
- Sémaphores
- Mutex
- Variables de condition

E/S Asynchrones

- **E/S synchrones: lecture d'un fichier → envoi de la requête et attente du résultat (une seule requête pour un processus à un moment donné):**
  - **Temps d'achèvement d'une E/S: imprévisible**
  - **Parallélisme au niveau des E/S non exploité.**

  →mauvais pour le temps réel

E/S Asynchrones

- **E/S asynchrones → lancement de plusieurs E/S et réception asynchrone des acquittements.**
- **Bloc de contrôle des E/S asynchrones (`struct aiocb`) contenant toutes les informations permettant de décrire une E/S, on peut:**
  - **Spécifier le type d'opération(Lec/Ecr)**
  - **Identifier le fichier sur lequel l'opération doit se faire.**
  - **Déterminer la portion de fichier considérée**
  - **Localiser un tampon de données à utiliser**
  - **Donner des priorités aux opérations**
  - **Demander la notification de l'achèvement de la requête par un signal ou par l'exécution d'une fonction.**

E/S Asynchrones

- `int aio_read` et `int aio_write`
  - Requête de lec/ecr en prenant en paramètre un bloc de contrôle d'E/S.
- `lio_listio`
  - Prépare une liste d'E/S chacune définie par un bloc de contrôle d'E/S.
- `int aio_error` et `aio_return`
  - Permettent de récupérer les erreurs ou les informations relatives à un bloc de contrôle d'E/S après achèvement de ces dernières.
- `int aio_fsync`
  - Provoque une synchronisation de toutes les opérations d'E/S asynchrones en cours associées au bloc de contrôle.
- `int aio_suspend`
  - Bloque un thread jusqu'à ce qu'une des E/S spécifiées en argument s'achève (ou sinon jusqu'à un temps maximum).
- `int aio_cancel`
  - Annule une opération d'E/S qui n'a pas été achevée.

E/S Asynchrones : struct aiocb

```
int aio_fildes;                 /* file descriptor */
volatile void *aio_buf;         /* buffer location */
size_t aio_nbytes;              /* length of transfer */
off_t aio_offset;               /* file offset */
struct sigevent aio_sigevent;   /* signal number and offset */
```

Synchronisation et communication inter processus

- Files de messages
- Mémoire partagé
- E/S asynchrones
- **Sémaphores**
- Mutex
- Variables de condition

Les sémaphores

- `sem_init` : création
- `sem_destroy` : suppression de l'association entre descripteur et sémaphore + suppression de sémaphore.
- `sem_open` : création
- `sem_close` : suppression de l'association entre descripteur et sémaphore
- `sem_unlink` : suppression de sémaphore
- `sem_wait` → P();
  - sem_trywait
  - sem_timedwait
- `sem_post(sem_t * sem)` → V();
- `sem_getvalue` : retourne la valeur du sémaphore.

Les sémaphores : sem_init()

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- **Initialise le sémaphore non nommé situé à l'adresse pointée par sem**

- **1 : sem_t *sem**
  - **Un pointeur vers sem_t**
- **2 : int pshared**
  - **Indique si ce sémaphore sera partagé entre les threads d'un processus ou entre processus**
    - 0 : le sémaphore est partagé entre les threads d'un processus
    - != 0 : Le sémaphore est partagé entre processus et devrait être situé dans une région de mémoire partagée
- **3 : unsigned int value**
  - **La valeur initiale du sémaphore**

Les sémaphores : sem_destroy()

```
int sem_destroy(sem_t *sem);
```

- détruit le sémaphore non nommé situé à l'adresse pointée par sem
- Seul un sémaphore initialisé avec sem_init() peut être détruit avec sem_destroy()

- **1 : sem_t *sem**
  - **Un pointeur vers sem_t**

Les sémaphores : sem_open()

```
sem_t *sem_open(const char *name, int oflag, mode_t mode,
unsigned int value)
```

- Initialiser et ouvrir un sémaphore nommé

- 1 : `const char *name`
  - **Nom du sémaphore**
- 2 : `int oflag`
  - **Spécifie les attributs qui contrôlent la manière d'opérer de l'appel**
    - **O_CREAT** : le sémaphore est créé s'il n'existe pas déjà
    - **O_EXCL** : une erreur sera renvoyée si le sémaphore du nom de *name* existe déjà
- 3 : `mode_t mode`
  - **Spécifie les permissions à placer sur le nouveau sémaphore**
  - **`https://man.developpez.com/man2/open/`**
- 4 : `unsigned int value`
  - **La valeur initiale du nouveau sémaphore**

Les sémaphores : sem_unlink(), sem_close()

```
int sem_unlink(const char *name)
```

- Supprime un sémaphore nommé référencé par name
- Le sémaphore est détruit une fois que tous les autres processus qui l'avaient ouvert l'ont fermé

```
int sem_close(sem_t *sem)
```

- Ferme le sémaphore nommé référencé par *sem*, permettant à toutes les ressources que le système a alloué au processus appelant pour ce sémaphore d'être libérées.

Les sémaphores : Exemple

```
#include <semaphore.h>
#define S_MODE S_IRUSR | S_IWUSR
//permissions de lecture et d'écriture pour le propriétaire
void main();
{
    if ((my_lock = sem_open ("/my.dat", O_CREAT|O_EXCL,S_MODE, 1) == -1) &&
      errno== ENOENT)
      perror("semaphore open failed"); exit(1);
    …
    for (i = 1; i < n; ++i)
      {if (childpid= fork()) break;}
    …
    if (sem_wait (&my_lock) == -1)
      {perror("semaphore invalid); exit (1); }
      //Critical Section
    if (sem_post (&my_lock) == -1)
      {perror("semaphore done"); exit(1); }
    …
    if (sem_close (&my_lock) == -1)
    { perror("semaphore close failed"); exit(1); }
}
```

Synchronisation et communication inter processus

- Files de messages
- Mémoire partagé
- E/S asynchrones
- Sémaphores
- **Mutex**
- Variables de condition

Mutex

- **Sémaphore binaire (?) → assure une exclusion mutuelle entre plusieurs threads.**
- `pthread_mutex_init` **: initialisation d'une mutex.**
- `pthread_mutex_destroy` **: suppression d'une mutex**
- `pthread_mutex_lock` **: équivalent d'un P() sur un sémaphore binaire**
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex)`
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex)` **: équivalent d'un V() sur un sémaphore binaire.**

Mutex

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++);

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

```c
int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
     pthread_create(&(tid[i]), NULL, &trythis, NULL);
     i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Output sans mutex

```
Job 1 has started
Job 2 has started
Job 2 has finished
Job 2 has finished
```

Output avec mutex

```
Job 1 started
Job 1 finished
Job 2 started
Job 2 finished
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

Synchronisation et communication inter processus

- Files de messages
- Mémoire partagé
- E/S asynchrones
- Sémaphores
- Mutex
- Variables de condition

Variables de condition

- **Les variables de condition peuvent être utilisées pour bloquer atomiquement les threads jusqu'à ce qu'une condition particulière se réalise.**
  - **Les variables de condition sont toujours utilisées avec des mutex (variables d'exclusion mutuelle).**
- **La condition est testée sous la protection d'une mutex:**
  - **Lorsque la condition est fausse, le thread bloque sur la variable de condition et relâche atomiquement la mutex (implicite) en attendant le changement de condition.**
  - **Lorsqu'un autre thread change de condition, il peut spécifier à la variable de condition s'il veut qu'un seul ou plusieurs threads en attente se réveille(nt), prenne(nt) la mutex et réévalue la condition.**

Variables de condition

```
Lock (mutex)
   wait (event)
      work
   signal (event)
Unlock (mutex)
```

Variables de condition

- `int pthread_cond_init`
  - Initialisation des variables conditionnelles.
- Utilisation de *mutex* et de variable de condition
  - Pour ce bloquer sur une variable de condition, un appel à la fonction `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)` doit être effectué en prenant en paramètre la variable et la *mutex* utilisée. Cette fonction déverrouille automatiquement la *mutex* et se bloque sur la variable de condition; la *mutex* est reprise lorsque le thread est débloqué
  - La fonction `int pthread_cond_signal(pthread_cond_t *cond)` peut être appelée pour débloquer au moins l'un des threads bloqués sur la variable de condition.
  - `int pthread_cond_broadcast(pthread_cond_t *cond)` peut être utilisée pour débloquer tous les threads.
- Destruction en utilisant `pthread_cond_destroy`

```
#include <pthread.h>
/* définition du tampon */
#define N 10 /* Nb de cases du tampon */
#define NbMess 20 /* Nb de messages échangés */
int NbPleins=0;
int tete=0, queue=0;
int tampon[N];
/* définition des conditions et du mutex */
pthread_cond_t vide;
pthread_cond_t plein;
pthread_mutex_t mutex;
pthread_t tid[2];

void Deposer(int m){
        pthread_mutex_lock(&mutex);
        if(NbPleins == N) pthread_cond_wait(&plein, &mutex);
        tampon[queue]=m;
        queue=(queue+1)%N;
        NbPleins++;
        pthread_cond_signal(&vide);
        pthread_mutex_unlock(&mutex);
}
int Prelever(void){
        int m;
        pthread_mutex_lock(&mutex);
        if(NbPleins ==0) pthread_cond_wait(&vide, &mutex);
        m=tampon[tete];
        tete=(tete+1)%N;
        NbPleins--;
        pthread_cond_signal(&plein);
        pthread_mutex_unlock(&mutex);
        return m;}
```

Déclaration des variables de condition

Section critique

Section critique

```
void * Prod(void * k) /********** PRODUCTEUR */
{
        int i; int mess;
        srand(pthread_self());
        for(i=0;i<=NbMess; i++){
                usleep(rand()%10000); /* fabrication du message */
                mess=rand()%1000;
                Deposer(mess);
                printf("Mess depose: %d\n",mess);
        }
}
void * Cons(void * k) /********** CONSOMMATEUR */
{
        int i; int mess;
        srand(pthread_self());
        for(i=0;i<=NbMess; i++){
                mess=Prelever();
                printf("\tMess preleve: %d\n",mess);
                usleep(rand()%1000000); /* traitement du message */}
}

void main() /* M A I N */
{
        int i, num;
        pthread_mutex_init(&mutex,0);
        pthread_cond_init(&vide,0);
        pthread_cond_init(&plein,0);
        /* creation des threads */
        pthread_create(tid, 0, Prod, NULL);
        pthread_create(tid+1, 0, Cons, NULL);
        // attente de la fin des threads
        pthread_join(tid[0],NULL);
        pthread_join(tid[1],NULL);
        // libération des ressources
        pthread_mutex_destroy(&mutex);
        pthread_cond_destroy(&vide);
        pthread_cond_destroy(&plein);
        exit(0);
}
```

Initialisation