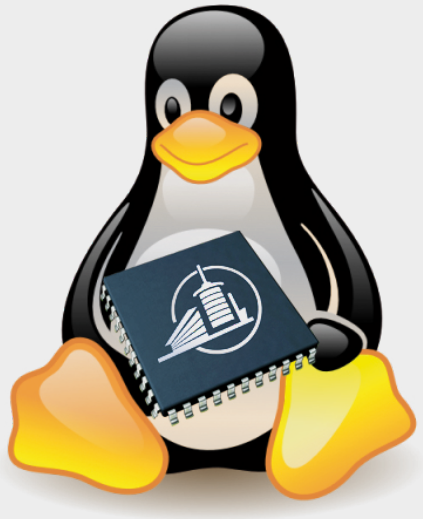


Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

CSEL1 – Construction Systèmes Embarqués sous Linux

Programmation système: Multiprocessing

HES-SO//Master TIC/TIN 2020

Daniel Gachet – HEIA-FR – Télécommunications

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Contenu

- Introduction
- Traitement des signaux
- Traitement multiprocessus
- Démons
- Accès concurrents
- Traitement multithreads
- Communication
- Ordonnanceur
- Control Groups

MASTER OF SCIENCE IN ENGINEERING

MSE

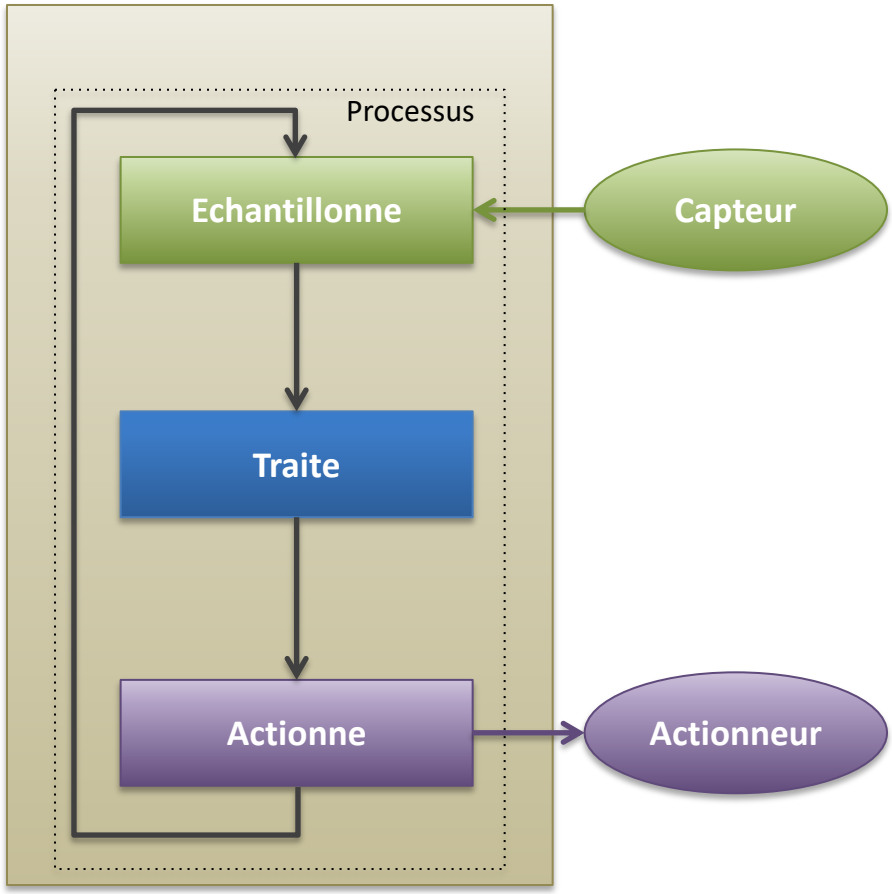
Introduction

MASTER OF SCIENCE IN ENGINEERING

MSE

Introduction – Problème

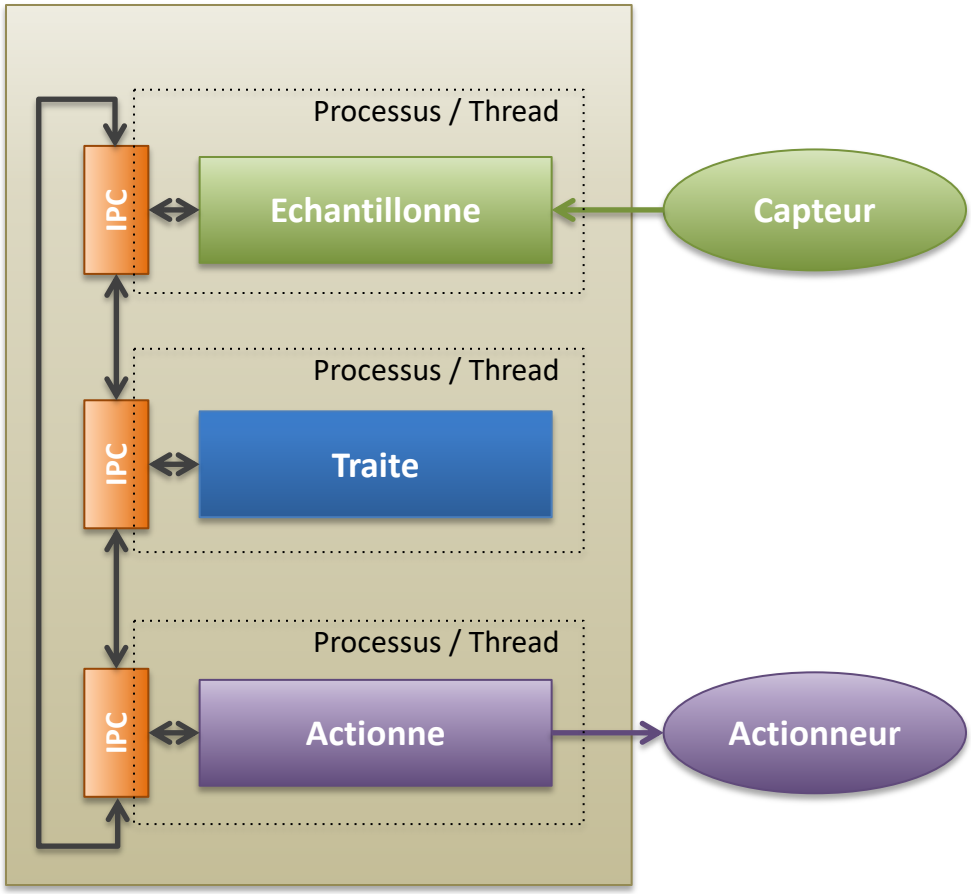
- Il est usuel pour un système embarqué de devoir traiter plusieurs tâches à la fois.
- Un schéma classique est d'implémenter un seul processus réalisant toutes les tâches de manière séquentielle, p.ex. échantillonnage d'un capteur, puis traitement des données reçues et finalement envoi des consignes sur un actionneur.
- Cette technique souffre d'inconvénients, p.ex. de petites modifications du logiciel peut impacter lourdement le comportement du système complet.



MASTER OF SCIENCE IN ENGINEERING
MSE

Introduction – Solution

- Un des chemins possibles pour pallier à ces inconvénients est l'utilisation de processus ou de threads propres à chaque tâche.
- L'échange de données entre les différentes tâches est alors assuré par une couche de communication entre processus / threads (IPC – Inter Process Communication).
- L'adaptation d'un des processus / threads n'aura ainsi plus ou peu d'impact sur les autres.



MASTER OF SCIENCE IN ENGINEERING
MSE

Traitement des signaux

MASTER OF SCIENCE IN ENGINEERING

MSE

Signaux – Introduction

- **Les signaux sont des interruptions logicielles permettant de traiter des événements asynchrones au bon déroulement du programme**
- **Les événements à l'origine de la levée des signaux sont souvent externes au système, p.ex. pression sur la touche « Ctrl-C » sur un clavier**
- **Un processus peut envoyer un signal à un autre processus ou groupe de processus, offrant ainsi une forme primitive de communication interprocessus**
- **Lorsqu'un signal est levé (généré ou envoyé), le noyau effectue une des actions suivantes:**
  - Ignorer le signal
    Le noyau ignore le signal et aucune action n'est entreprise.
  - Capturer et traiter le signal
    Le noyau arrête l'exécution du processus et traite le signal en appelant une fonction ayant été enregistrée préalablement par le processus en cours.
  - Effectuer une action par défaut
    Le noyau arrête l'exécution du processus et exécute une action par défaut implémentée par le noyau lui-même et dépendante du signal. Celle-ci consiste souvent à terminer le processus et générer un coredump.

MASTER OF SCIENCE IN ENGINEERING
MSE

Signaux – Liste

- Les signaux supportés par Linux sont disponibles dans le fichier « `signal.h` »
- Chaque signal est identifié par une valeur symbolique avec le préfix « `SIG` », laquelle représente une valeur entière positive non nulle
- Signaux disponibles sous Linux (`$ kill -l`)

Signal	No	Description
SIGHUP	1	Hangup
SIGINT	2	Terminal interrupt *(Ctrl-C)*
SIGQUIT	3	Terminal quit *(Ctrl-\)*
SIGILL	4	Illegal instruction
SIGTRAP	5	Trace trap
SIGABRT	6	Abort
SIGBUS	7	BUS error
SIGFPE	8	Floating point exception
SIGKILL	9	Kill *(can't be caught or ignored)*
SIGUSR1	10	User defined signal 1
SIGSEGV	11	Invalid memory segment access
SIGUSR2	12	User defined signal 2
SIGPIPE	13	Write on a pipe with no reader, Broken pipe
SIGALRM	14	Alarm clock
SIGTERM	15	Termination

Signal	No	Description
SIGSTKFLT	16	Stack fault
SIGCHLD	17	Child process has stopped or exited, changed
SIGCONT	18	Continue executing, if stopped
SIGSTOP	19	Stop executing *(can't be caught or ignored)*
SIGTSTP	20	Terminal stop signal *(Ctrl-Z)*
SIGTTIN	21	Background process trying to read, from TTY
SIGTTOU	22	Background process trying to write, to TTY
SIGURG	23	Urgent condition on socket
SIGXCPU	24	CPU limit exceeded
SIGXFSZ	25	File size limit exceeded
SIGVTALRM	26	Virtual alarm clock
SIGPROF	27	Profiling alarm clock
SIGWINCH	28	Window size change
SIGIO	29	I/O now possible
SIGPWR	30	Power failure restart

- Les signaux `SIGSTOP(19)` et `SIGKILL(9)` ne peuvent être ni capturés et ni ignorés

Signaux – Opérations

- **Pour le traitement des signaux, Linux propose des services permettant d'attacher une méthode spécifique à chaque signal respectivement de lever un signal pour un processus ou un groupe de processus.**
- **Opérations**
  - Capturer un signal (syscall: sigaction)
  - Lever un signal (syscall: kill)
  - Attendre sur un signal (syscall: pause)

Signaux – Capturer un signal

- **Pour capturer et traiter un signal, l'application doit préalablement attacher une méthode de traitement pour le signal souhaité. Pour ce faire, Linux propose l'appel système** `sigaction()`.

```
#include <signal.h>
int sigaction (int signo,
               const struct sigaction* act,
               struct sigaction *oldact);
```

- **Exemple**

```
struct sigaction act = {.sa_handler = catch_signal,};
int err = sigaction (SIGHUP, &act, NULL);
if (err == -1)
   /* error */
```

- **Comportement**
  - La fonction `sigaction()` associe une méthode de traitement au signal passé en 1er argument, ici `SIGHUP`. Il est possible de donner un pointeur sur une `struct sigaction` afin de récupérer l'action précédente.

MASTER OF SCIENCE IN ENGINEERING

MSE

Signaux – Capturer un signal (II)

- **La `struct sigaction` permet d'attacher une méthode de traitement pour le signal que l'on souhaite capturer**

```
struct sigaction {
     void (*sa_handler)(int);
     void (*sa_sigaction)(int, siginfo_t*, void*);
     sigset_t sa_mask;
     int sa_flags;
     void (*sa_restorer)(void); /* obsolete: do not use */
};
```

  - Cette structure permet de choisir entre deux signatures pour la méthode de traitement, soit `sa_handler`, soit `sa_sigaction`. Si l'on souhaite traiter le signal avec la méthode `sa_sigaction`, il est nécessaire d'ajouter le fanion `SA_SIGINFO` dans `sa_flags`.
  - Si l'on souhaite ignorer le signal, on pourra associer `SIG_IGN` à `sa_handler`. Si l'on souhaite le comportement par défaut on assignera `SIG_DFL` à `sa_handler`.
  - `sa_mask` permet de bloquer certains signaux.
  - `sa_flags` permet de modifier le comportement du signal. Pour plus de détails voir la man page.

Signaux – Capturer un signal (III)

- **La méthode de traitement `catch_signal` prend la forme suivante**

```
void catch_signal (int signo) {
     /* do something... */
     /* to terminate process execution with
      *   1) success: exit(EXIT_SUCCESS);
      *   2) failure: exit(EXIT_FAILURE);
      */
}
```

- **Une fois le signal traité à l'aide de la méthode `catch_signal()`, le processus poursuivra son exécution. Par contre, si l'on souhaite terminer l'exécution du processus ayant capturé le signal, il suffit d'utiliser l'appel système `exit()`.**

Signaux – Lever un signal

- **Pour lever un signal, Linux propose l'appel système `kill()` pour envoyer un signal à un processus ou à un groupe de processus.**

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

- **Exemple**

```
int err = kill (0, SIGHUP);
if (err == -1)
   /* error */
```

- **Comportement**
  - Si `pid` est une valeur positive plus grande que `0` (zéro), la fonction `kill()` permet de lever le signal `sig` sur le processus `pid`, pour autant qu'il est les droits.
  - Si `pid` vaut `0` (zéro), alors `kill()` émet le signal `sig` à tous les processus du groupe du processus levant le signal.
  - Si `pid` est égal à `-1`, `kill()` lèvera le signal `sig` pour tous les processus dont il a la permission d'envoyer un signal.
  - Si `pid` est plus petit que `-1`, la fonction `kill()` envoie le signal `sig` à tous les processus du groupe identifié par `-pid`.

Signaux – Attendre sur un signal

- **Pour t'attendre qu'un signal soit levé ou que le processus se termine, Linux propose l'appel système `pause()`.**

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

- **Exemple**

```
while (1)
    pause();
```

- **Comportement**
  - La fonction `pause()` met en sommeil le processus jusqu'à qu'un signal soit levé et capturé ou que le processus se termine.

MASTER OF SCIENCE IN ENGINEERING

MSE

Signaux – Précaution et rappel

- Sous Linux, il est très courant que des méthodes faisant des appels système soient interrompues par la levée de signaux.
- Pour éviter un mauvais comportement de l'application, il est important de bien contrôler la valeur de la variable `errno` et de la tester contre l'erreur `EINTR`.
- Si l'erreur `EINTR` est signalée, il faut simplement répéter l'appel.
- Il est judicieux de toujours bien vérifier (p.ex. par la man-page) si un appel système dépend d'un signal.

- Exemple

```
/**
 * reliable implementation of a sleep against raising of the EINTR signal.
 * this method will sleep at least the specified value.
 * @param ns sleep time in nanoseconds
 */
void safe_sleep(int ns) {
    struct timespec sleep_time = {.tv_nsec = ns,};
    struct timespec remaining  = {.tv_nsec=0,};
    while(true) {
        int status = nanosleep (&sleep_time, &remaining);
        if(status == 0) break;
        if (errno == EINTR) {
            sleep_time = remaining;
        } else {
            perror ("nanosleep");
            exit(status);
        }
    }
}
```

MASTER OF SCIENCE IN ENGINEERING

MSE

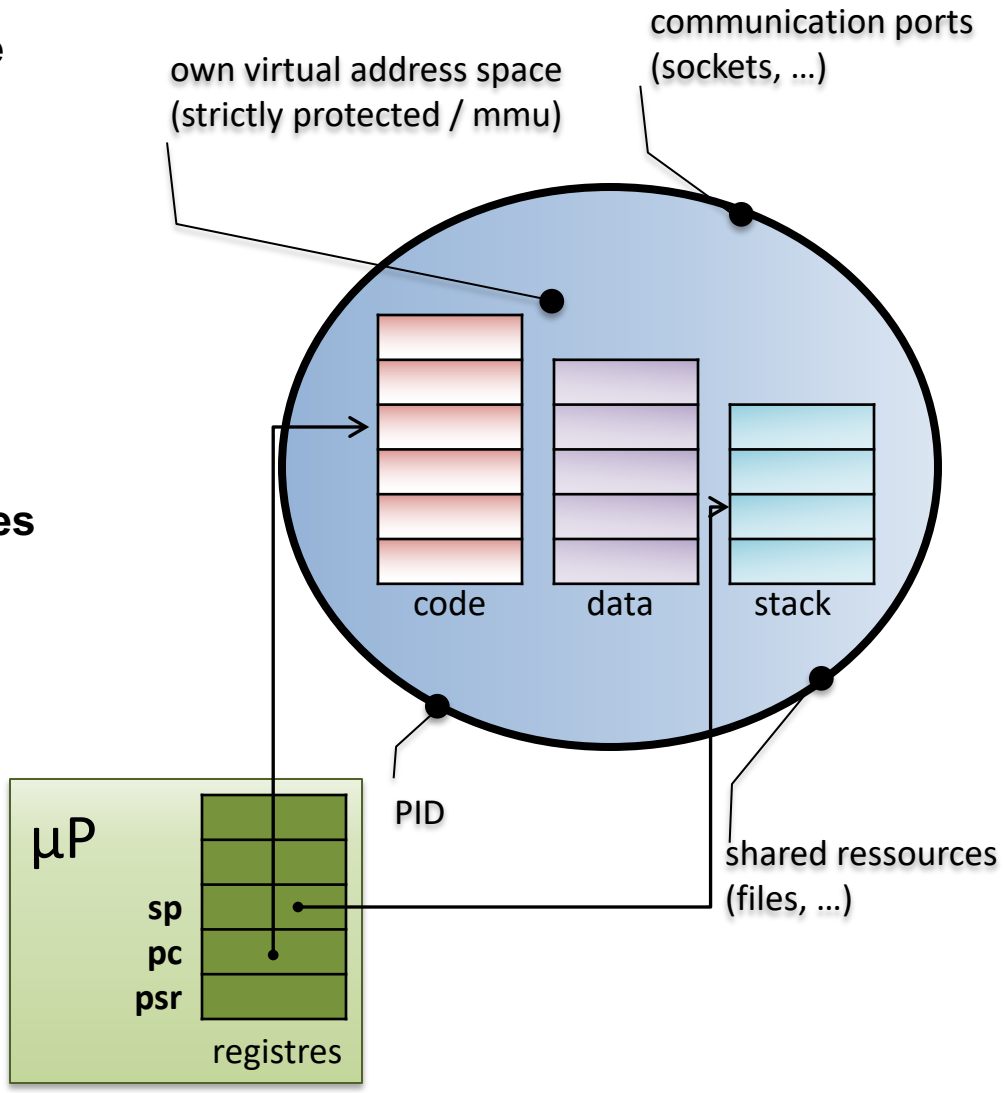
Traitement multiprocessus

MASTER OF SCIENCE IN ENGINEERING

MSE

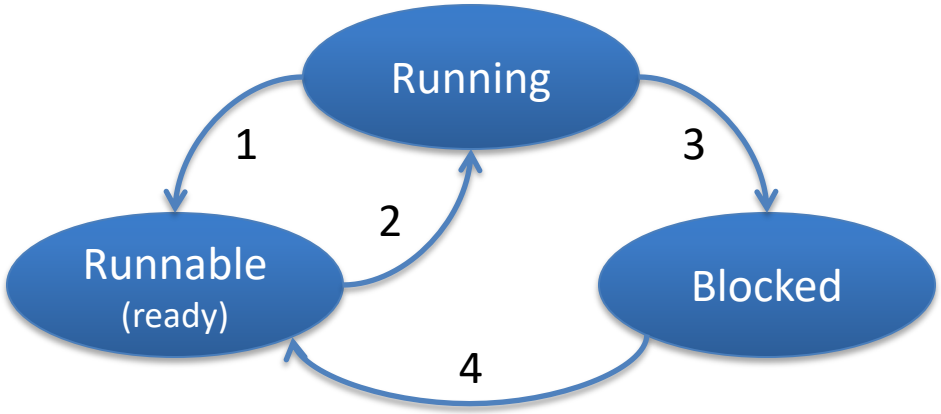
Processus – Modèle

- **Les processus sont des entités de calcul/traitement indépendantes disposant d'un processeur virtuel et d'un espace adressable propre**
- **Chaque processus est défini par**
  - PID – Process identifier
  - PGID – Process Group Identifier
  - PPID – Parent Process Identifier
- **Quelques attributs supplémentaires**
  - SID – Session Identifier
  - UID – User Identifier
  - GID – Group Identifier
  - Working directory
  - File descriptor table
  - …
- **Le processus « `init` » a le pid 1**
- **Le processus « `idle` » a le pid 0**



Processus – Etats

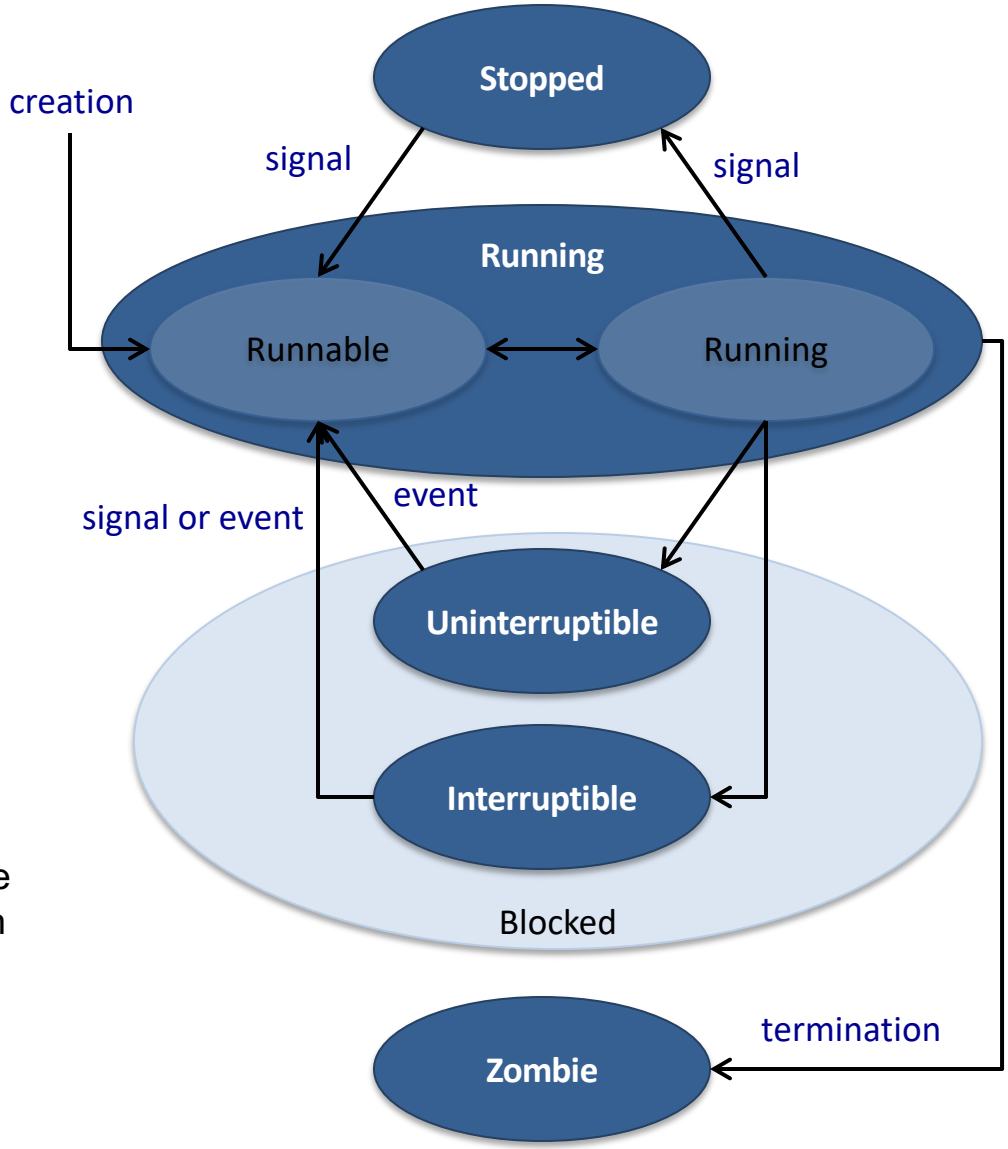
- **Les états principaux d'un processus**
  - **Running**: le processus utilise le processeur et ses ressources
  - **Blocked**: le processus est bloqué en attente sur un événement
  - **Runnable**: le processus est prêt, mais attend que le micro-processeur soit libéré



- **Les transitions possibles**
  1. L'ordonnanceur passe la main à un autre processus
  2. L'ordonnanceur donne la main au processus
  3. Le processus est bloqué en attende d'un événement, p.ex. des entrées/sorties
  4. L'événement sur lequel le processus était en attente est survenu

Processus – Etats sous Linux

- **L'état d'un processus sous Linux est défini légèrement différemment**
  - **Running**
    Linux ne fait pas de distinction entre l'état runnable (ready) et l'état running
  - **Interruptible**
    le processus est dans l'état bloqué, en attente d'un événement, p.ex. la levée d'une entrée/sortie, la disponibilité d'une ressource ou un signal d'un autre processus
  - **Unterruptible**
    le processus est également dans l'état bloqué, en attente d'un événement. Ce processus ne peut cependant pas être interrompu par un signal
  - **Stopped**
    L'exécution du processus a été stoppée et ne pourra être réactivée que par une action d'un autre processus
  - **Zombie**
    Le processus est terminé, mais en attente que le processus parent lise son statut/état



Processus – Etats sous Linux (II)

- **L'état d'un processus peut être obtenu en lisant les différents attributs disponibles sous le procfs, p.ex.**

```
$ cat /proc/205/status
Name:      scsi_eh_32
State:     S (sleeping)
Tgid:      205
Ngid:      0
Pid:       205
PPid:      2
TracerPid:          0
Uid:       0        0        0        0
Gid:       0        0        0        0
FDSize:    32
Groups:
Threads:   1
SigQ:      0/32117
SigPnd:    0000000000000000
ShdPnd:    0000000000000000
SigBlk:    0000000000000000
SigIgn:    ffffffffffffffff
SigCgt:    0000000000000000
CapInh:    0000000000000000
CapPrm:    0000001fffffffff
CapEff:    0000001fffffffff
CapBnd:    0000001fffffffff
Seccomp:   0
Cpus_allowed:       ff
Cpus_allowed_list:  0-7
Mems_allowed:       1
Mems_allowed_list:  0
voluntary_ctxt_switches:      2
nonvoluntary_ctxt_switches:   0
```

Processus – Etats sous Linux (III)

- **Avec la command « `ps` », Linux offre un service plus convivial pour obtenir la liste des processus et leur état, p.ex.**

```
$ ps aux | head
```

```
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0   4588  2584 ?        Ss   18:04   0:01 /sbin/init
root          2  0.0  0.0      0     0 ?        S    18:04   0:00 [kthreadd]
root          3  0.0  0.0      0     0 ?        S    18:04   0:00 [ksoftirqd/0]
root          4  0.0  0.0      0     0 ?        S    18:04   0:00 [kworker/0:0]
root          5  0.0  0.0      0     0 ?        S<   18:04   0:00 [kworker/0:0H]
root          7  0.0  0.0      0     0 ?        S    18:04   0:00 [rcu_sched]
root          8  0.0  0.0      0     0 ?        S    18:04   0:00 [rcu_bh]
root          9  0.0  0.0      0     0 ?        S    18:04   0:01 [migration/0]
root         10  0.0  0.0      0     0 ?        S    18:04   0:00 [watchdog/0]
```

ou

```
$ ps -p 205 o user,pid,%cpu,%mem,vsz,rss,tty,stat,start,cmd
```

```
USER        PID %CPU %MEM    VSZ   RSS TT       STAT  STARTED CMD
root        205  0.0  0.0      0     0 ?        S    18:05:03 [scsi_eh_32]
```

Processus – Etats sous Linux (IV)

PROCESS STATE CODES

Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process:

- D uninterruptible sleep (usually IO)
- R running or runnable (on run queue)
- S interruptible sleep (waiting for an event to complete)
- T stopped, either by a job control signal or because it is being traced
- W paging (not valid since the 2.6.xx kernel)
- X dead (should never be seen)
- Z defunct ("zombie") process, terminated but not reaped by its parent

For BSD formats and when the stat keyword is used, additional characters may be displayed:

- < high-priority (not nice to other users)
- N low-priority (nice to other users)
- L has pages locked into memory (for real-time and custom IO)
- s is a session leader
- l is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
- + is in the foreground process group.

MASTER OF SCIENCE IN ENGINEERING

MSE

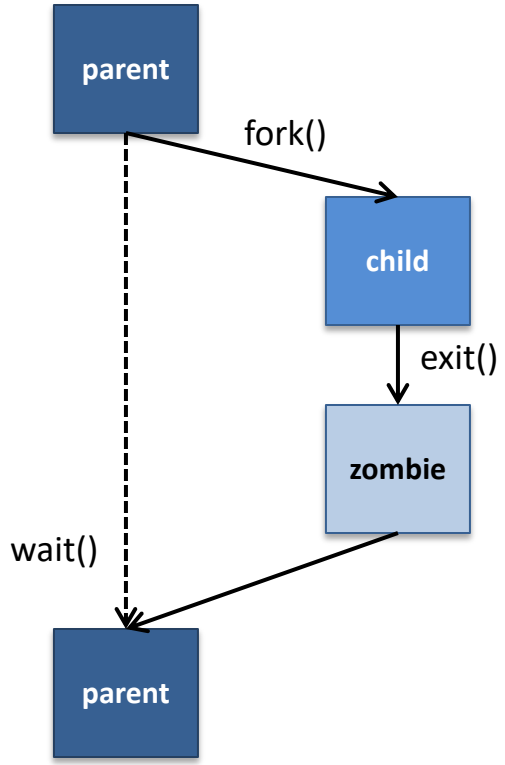
Processus – Opérations

- **Linux offre une longue liste de services pour la gestion des processus, ici quelques services intéressants :**
  - Clonage d'un processus (syscall: fork)
  - Terminaison d'un processus (syscall: exit)
  - Attente sur la terminaison d'un processus (syscall: waitpid, wait)
  - Exécution d'un nouveau programme (syscall: exec…)
  - Lecture de son propre identificateur (syscall: getpid)
  - Lecture de l'identificateur du processus parent (syscall: getppid)

Processus – Clonage

- **Le clonage d'un processus, c.à.d. la création d'une copie strictement idententique d'un processus est obtenue à l'aide de l'appel système `fork()`**

```
#include <sys/types.h>
#include <unistd.h>
int fork(void);
```



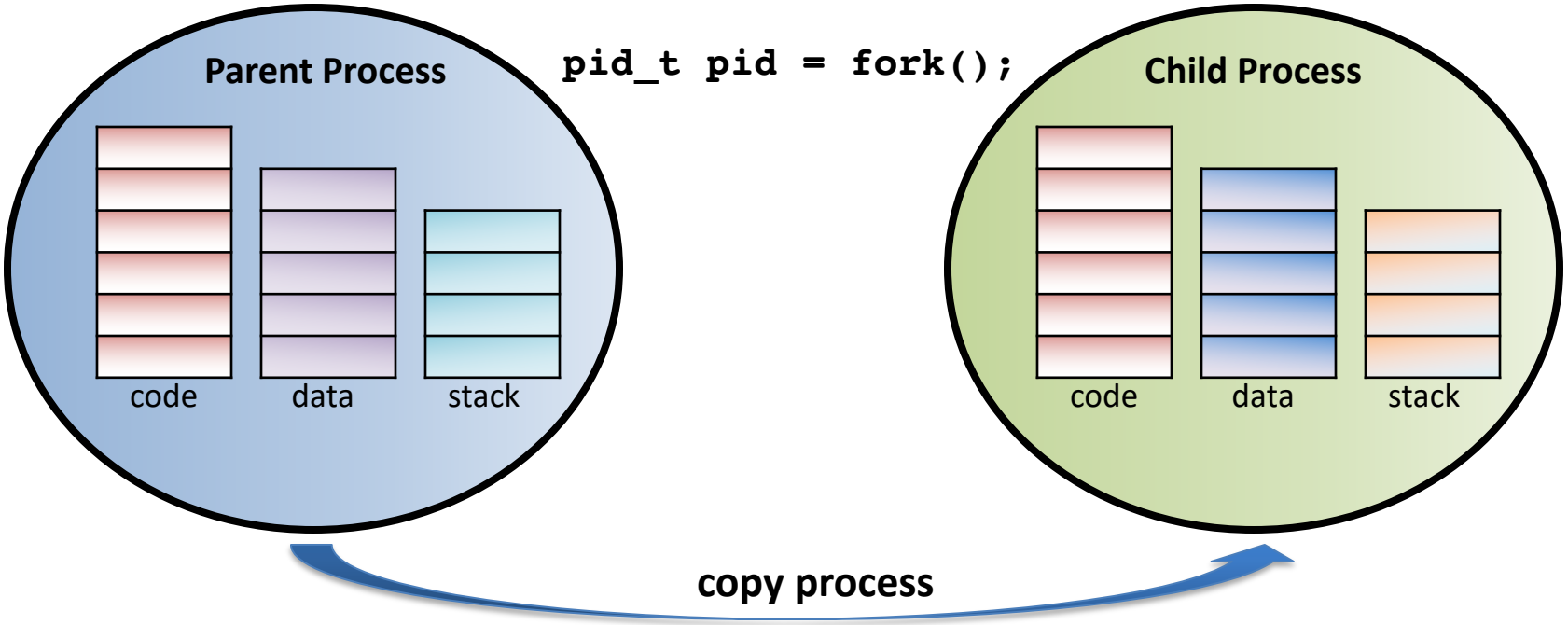
- **Exemple**

```
pid_t pid = fork();
if (pid == 0)
     /* code de l'enfant */
else if (pid > 0)
     /* code du parent */
else
     /* error */
```

- **Comportement**
  - La fonction `fork()` retourne deux fois, une première fois dans le code du processus parent et une deuxième dans celui du processus enfant.
  - Dans le code du parent, si `pid` est plus grand que zéro, il indique le `pid` de l'enfant. Si le `pid` est plus petit que 0, il indique une erreur.
  - Dans le code de l'enfant, la valeur du `pid` est toujours égale à zéro (0).

Processus – « Copy on Write »

- Lors du clonage d'un processus, le noyau Linux crèe un nouvel espace virtuel pour le nouveau processus (processus enfant).
- A quelques exceptions près (voir les « man pages »), toutes les données sont copiées dans le nouveau processus.
- Sous Linux, la copie des données se fait à la volée, c.à.d. les données du processus parent dans copiées dans l'espace du processus enfant que lorsque celles-ci sont modifiées par l'un ou l'autre des deux processus.



Processus – Terminaison

- **La destruction d'un processus arrive lorsque celui-ci exécute l'appel système `exit()`. Elle se produit également lorsque le processus sort de la routine `main()`.**

```
#include <stdlib.h>
void exit(int status);
```

- **Exemple**

```
if(error)
     exit (ERROR_CODE);
else
     exit (0);
```

- **Comportement**
  - La fonction `exit()` termine le processus courant et retourne au processus parent en lui passant la valeur de `status & 0377`.
  - La valeur 0 indique que le processus s'est terminé avec succès, tandis qu'une valeur différente de zéro indique une erreur.
  - Les fichiers sont flushées et fermés, puis toutes les ressources sont libérées.
  - Tant que le processus parent ne lit pas la raison pour laquelle le processus enfant a été détruit (à l'aide du service `wait()` ou `waitpid()`, ce dernier se trouve dans l'état « zombie ».

Processus – Attente d'un changement d'état

- **Après la création de processus enfant, le processus parent a la possibilité d'attendre sur un changement d'état de ses processus enfant à l'aide des appels systèmes `wait()` et `waitpid()`. Seuls les changements d'état « terminated », « stopped » et « resumed » sont considérés par ces services.**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- **Exemple**

```
while (1) {
     int status = 0;
     pid_t pid = waitpid (-1, &status, 0);
     if (pid < 0) break; // error or no more child
     // treat child process status...
}
```

- **Comportement**
  - La fonction `waitpid()` bloque le processus parent jusqu'à ce qu'un enfant change d'état.
  - L'appel `wait(&status)` est équivalent à `waitpid (-1, &status, 0);`

Processus – Exécution d'un nouveau programme

- **L'exécution d'un nouveau programme, est obtenue à l'aide des appels système `exec()`. Ceux-ci se déclinent en plusieurs variantes.**

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
```

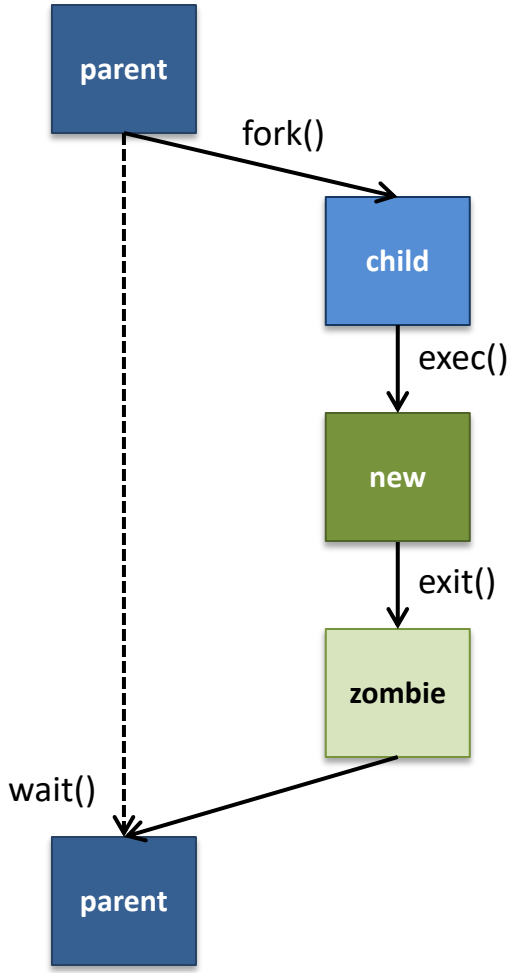
- **Exemple**

```
int err = execl("/bin/ls", "ls", "-r", "-l", NULL);
```

ou

```
char *args[] = {"ls", "-r", "-l", NULL };
int err = execv("/bin/ls", args);
```

- **Comportement**
  - La série de fonctions `exec()` lance l'application passée en premier paramètre et ne retourne jamais sauf en cas d'erreur.
  - La liste d'arguments doit impérativement être terminée par NULL

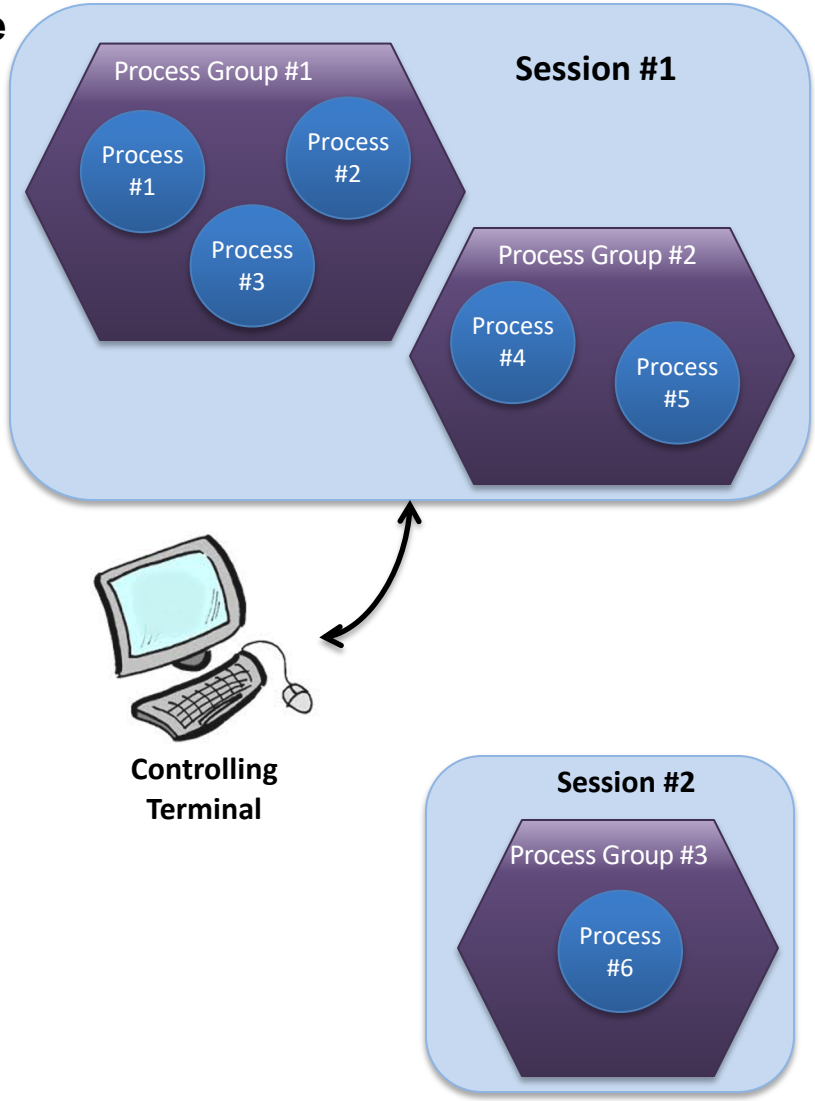


Processus – Utilisateur et Groupe

- **Chaque processus est associé a un utilisateur et à un groupe. Ces derniers sont identifiés par une valeur numérique. Il est important de noter que sous Linux, ces identifiants dictent les opérations que le processus sera autorisé à effectuer.**
- **Pour obtenir ces identifiants, Linux propose entre autres les services suivants:**
  - Lecture/écriture de l'identificateur de l'utilisateur (syscall: geteuid)
  - Lecture de l'identificateur du groupe (syscall: getegid)
- **Il est possible d'obtenir une représentation textuelle de ces valeurs. Pour cela il suffit de lire les fichiers `/etc/passwd` et respectivement `/etc/group`. Plus simplement, Linux propose les services suivants:**
  - Conversion du nom de groupe `struct group *getgrgid(gid_t gid);`
  - Conversion du nom d'utilisateur `struct passwd *getpwuid(uid_t uid);`

Processus – Session et groupe de processus

- **Chaque processus est membre d'un groupe de processus, identifié par un `pgid`.**
- **Le groupe de processus permet à des processus de contrôle d'envoyer des signaux à tous les processus du groupe.**
- **La session est une collection de groupes de processus. Elle est généralement le résultat du login d'un utilisateur. Un terminal de contrôle est généralement associé à une session.**
- **Quelques services:**
  - Lecture du pgid `pid_t getpgid(0);`
  - Lecture du sid `pid_t getsid (0);`
  - Création du nouveau groupe de processus
    ```
    int setpgid (0, 0);
    ```
  - Création d'une nouvelle session
    ```
    int setsid (0);
    ```



MASTER OF SCIENCE IN ENGINEERING
MSE

Démons
(Daemon)

MASTER OF SCIENCE IN ENGINEERING

MSE

Daemons – Introduction

- **Les démons (daemon) sont des processus qui fonctionnent en arrière-plan. Les démons ont comme processus parent le processus « init ». Ils sont généralement lancés au démarrage de la machine et s'exécutent avec les privilèges « root » ou tout autre utilisateur spécial.**

- **Marche à suivre pour qu'un processus se transforme en démon:**
  1. Créer un nouveau processus: `fork()` et terminer le processus parent: `exit()`
  2. Créer une nouvelle session pour le nouveau processus: `setsid()`
  3. Créer le processus démon: `fork()` et terminer le processus parent: `exit()`
  4. Capturer les signaux souhaités: `sigaction ()`
  5. Mettre à jour le masque pour la création de fichiers: `umask()`
  6. Mettre à jour le masque pour la création de fichiers: `chdir()`
  7. Fermer tous les descripteurs de fichiers: `close()`
  8. Rediriger `stdin`, `stdout` et `stderr` vers `/dev/null`: `open()` et `dup2()`
  9. Option: ouvrir un fichier de logging, p.ex. sous syslog: `openlog()`
  10. Option: chercher l'ID de l'utilisateur et du groupe avec moins de privilèges
  11. Option: changer le répertoire root vers un avec moins de visibilité: `chroot()`
  12. Option: changer l'ID de l'utilisateur et du groupe: `seteuid()` et `setegid()`
  13. Implémenter le corps du démon…

Daemons – Alternatives

- **Sous Unix System V, les systèmes s'initialisent grâce au daemon « `init` ». Ce programme est chargé de lire le fichier « `/etc/inittab` » et de lancer les différentes applications sous forme de daemon.**
- **Ce système a été tout naturellement repris sous Linux. Cependant il existe tout une série d'alternatives:**
  - busybox-init: version simplifiée d'init de Unix System V
  - systemd: semble devenir le remplaçant d'init de Unix System V
  - upstart: version Ubuntu
  - …

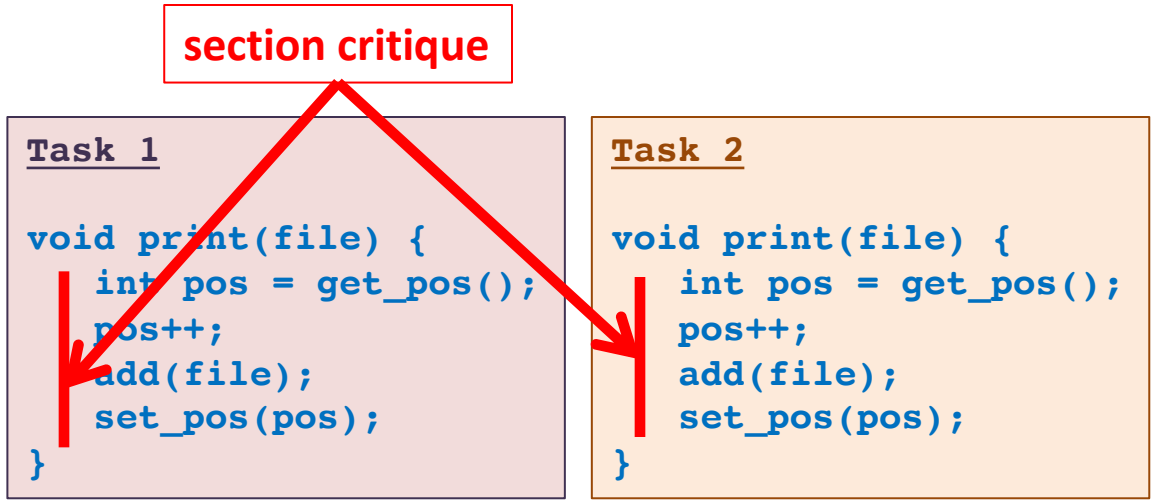
Accès concurrents (race conditions)

Race conditions – Introduction

- En programmation multitâches, il arrive fréquemment que plusieurs tâches (processus ou threads) cherchent à accéder au même instant à une ressource commune et que l'une d'entre elles tente de la modifier, p.ex. un spooler d'une imprimante. Dans de cas, on parle alors de situation de compétition (race condition).

pos	name
0	file0
1	file1
2	file2
3	file3
4	file4

→ next_pos = 5



- Si cette situation est mal gérée, elle peut déboucher à des résultats différents selon l'ordre dans lequel les différentes tâches vont agir sur le système. Dans des situations extrêmes cela peut même résulter à des résultats erronés.
- Par leur nature aléatoire, les situations de compétition sont des problèmes excessivement complexes à identifier et à corriger. Il est par conséquent essentiel d'avoir un excellent design du système avant de le réaliser.

Race conditions – Exclusion mutuelle

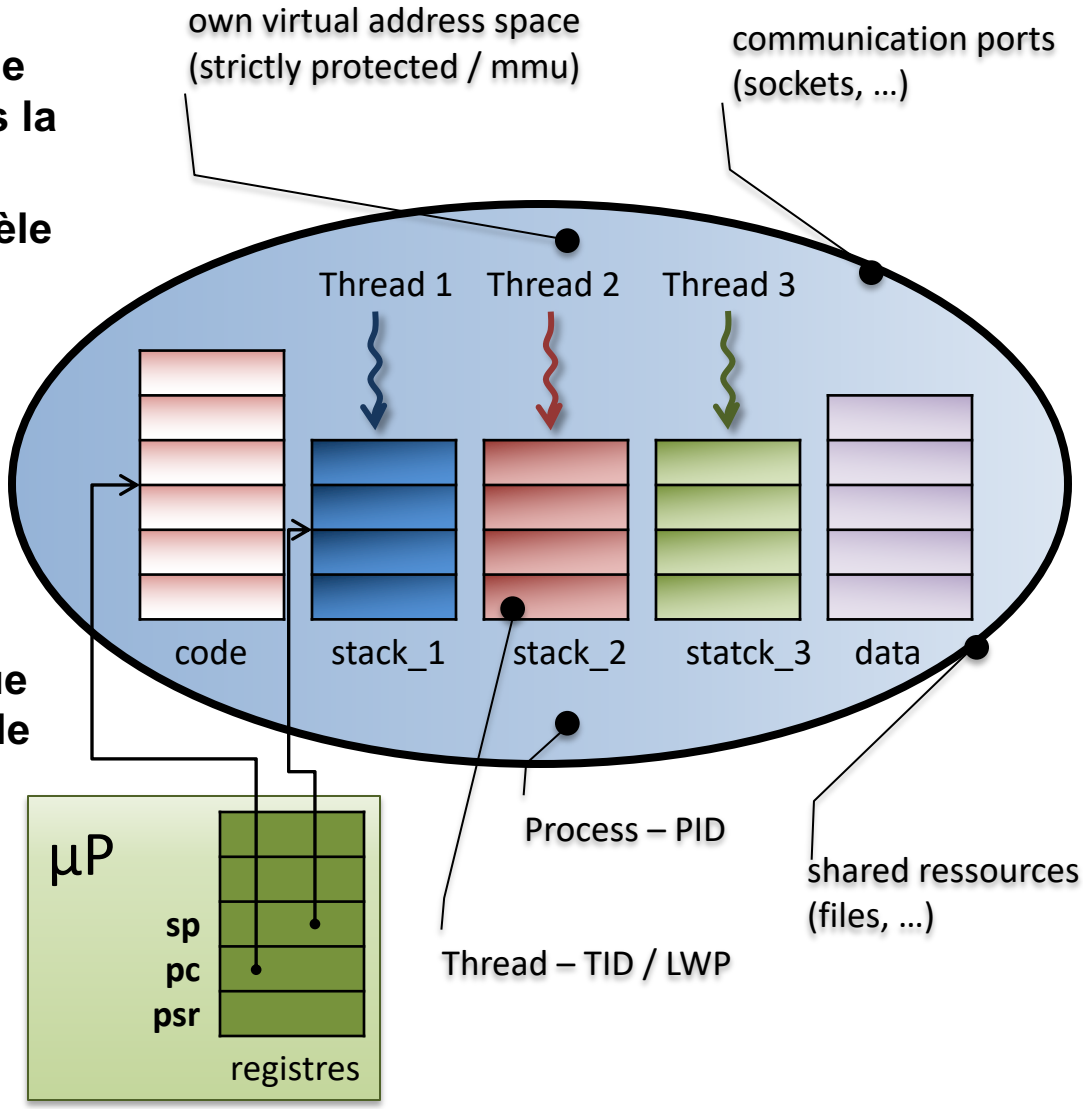
- **L'exclusion mutuelle est la méthode permettant de protéger les sections critiques et d'éviter des situations de compétitions.**
- **L'exclusion mutuelle consiste n'autoriser qu'une seule tâche à la fois à accéder à la ressource partagée et à bloquer toutes les autres.**
- **Sous Linux, il existe différents mécanismes pour réaliser l'exclusion mutuelle**
  - Les sémaphores
  - Les mutex (version simplifiée des sémaphores et spécialisée pour cette tâche)
  - La désactivation des interruptions (au niveau du noyau)
  - La désactivation de l'ordonnancement
  - Les moniteurs
  - Les verrous

- **Cependant la méthode la plus simple pour éviter des situations de compétition est de ne pas les créer. Pour cela il suffit de dédier un et un seul processus à une ressource. Si d'autres acteurs ont besoin d'y accéder, on peut simplement utiliser les services de communication offerts par Linux.**

Traitement multithreads

Threads – Modèle

- **Les threads sont également une abstraction très populaire dans la programmation moderne**
- **Ils offrent une exécution parallèle de plusieurs co-routines à l'intérieur d'un même espace adressable.**
- **Chaque thread est défini par**
  - PID – Process identifier
  - TID – Thread identifier
- **Dans cet espace virtuel fourni par le processus parent, chaque thread dispose de sa propre pile** *(stack)*
- **Au niveau du noyau Linux, chaque thread est vue comme un processus léger** *(light weight process – LWP)*



Threads – Avantages et inconvénients

- **Avantages**
  - Nécessitent moins de ressources
  - Changement de contexte plus simple et plus rapide
  - Communication entre thread moins compliquée
  - Variables globales accessibles par tous les threads

- **Inconvénients**
  - Protection des actions critiques plus complexe
  - Protection des sections critiques plus complexe
  - Variables globales accessibles par tous les threads

→ ***Race conditions***

MASTER OF SCIENCE IN ENGINEERING

MSE

Threads – Opérations

- **Grâce à la bibliothèque pthread de POSIX, Linux offre une longue liste de services pour la gestion des threads, ici quelques services intéressants :**
  - Création d'un thread (syscall: pthread_create)
  - Suppression d'un thread (syscall: pthread_cancel)
  - Terminaison d'un thread (syscall: pthread_exit)
  - Attente sur la terminaison d'un thread (syscall: pthread_join)
  - Envoi d'un signal à un thread (syscall: pthread_kill)

  - Gestion de l'état de suppression (syscall: pthread_setcancelstate)
  - Gestion du type de suppression (syscall: pthread_setcanceltype)
  - Ajout de méthodes de nettoyage (syscall: pthread_cleanup_push)
  - Suppression de méthodes de nettoyage (syscall: pthread_cleanup_pop)

  - Commandes pour afficher la liste des threads

```
$ ps -eLf
$ ps H -o user,ppid,pid,tid,stat,cmd,comm,wchan
```

MASTER OF SCIENCE IN ENGINEERING
MSE

Threads – Synchronization

- **La bibliothèque pthread propose également des services de synchronisation et de gestion de sections critiques entre threads d'un même processus, ici quelques services intéressants :**
  - Mutex
    - Création d'un mutex (syscall: pthread_mutex_init)
    - Destruction d'une mutex (syscall: pthread_mutex_destroy)
    - Prise d'une mutex (syscall: pthread_mutex_lock)
    - Libération d'une mutex (syscall: pthread_mutex_unlock)
  - Variables conditionnelles
    - Création d'une variable (syscall: pthread_cond_init)
    - Destruction d'une variable (syscall: pthread_cond_destroy)
    - Attente d'une condition (syscall: pthread_cond_wait)
    - Signalisation d'une condition (syscall: pthread_cond_signal)
- **Il est bien évident que les autres mécanismes de communications proposés aux processus sont également disponibles au niveau des threads, p. ex. les sémaphores, les sockets, etc.**

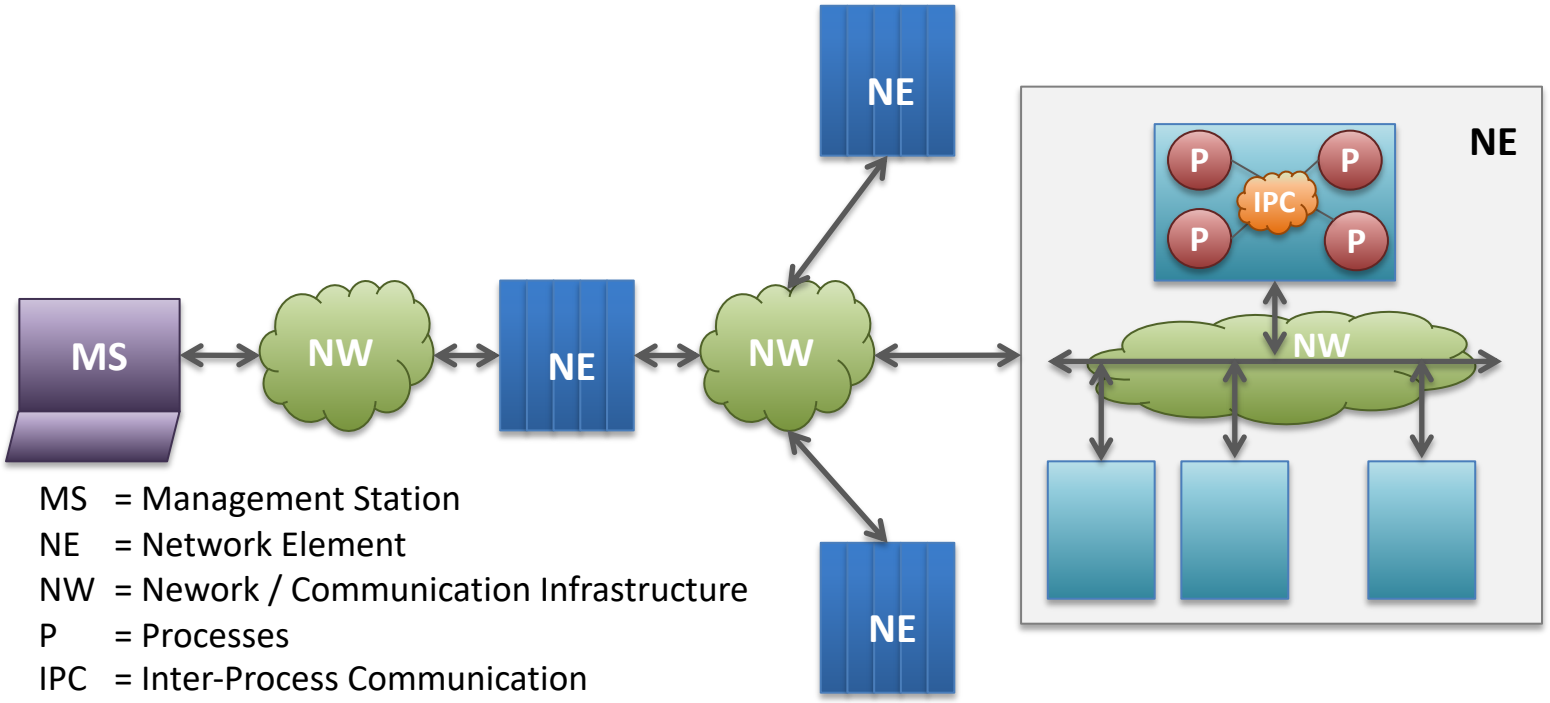
Communication

MASTER OF SCIENCE IN ENGINEERING

MSE

Communication – Introduction

- **Dans tous systèmes embarqués, l'échange d'information est essentiel entre différents systèmes et processus formant le logiciel de ces systèmes**
- **On peut schématiquement distinguer 4 niveaux de communication**
  - Communication entre stations de gestion et équipements
  - Communication entre équipements mis en réseau
  - Communication entre éléments d'un même équipement
  - Communication entre processus d'un même élément



MS = Management Station
NE = Network Element
NW = Nework / Communication Infrastructure
P = Processes
IPC = Inter-Process Communication

Communication – IPC – Inter-Process Communication

- **Dans ce dernier cas, on parle de communication interprocessus (Inter-Process Communication – IPC), laquelle comprend 2 catégories principales de mécanismes**
  - Mécanismes d'échange d'information ou de données
    - Fichiers (Files)
    - Mémoires partagées (Shared Memories)
    - Tubes nommés ou non (FIFO, Pipes)
    - Files d'attente de messages (Messages Queues)
    - Sockets (Unix ou Internet)
    - Protocoles de communication/gestion (D-Bus, SOAP, SNMP, REST, RPC, XML-RPC, etc.)
  - Mécanismes de synchronisation
    - Signaux
    - Sémaphores
    - Verrous (locks)
- **Il est intéressant de noter que les mécanismes d'échange d'information servent souvent également de mécanisme de synchronisation.**

Communication – Pipes

- **Les pipes implémentent un canal de communication unidirectionnel entre 2 processus.**
- **Ce canal est accessible par l'intermédiaire de 2 descripteurs de fichiers virtuels. Le premier descripteur (`fd[0]`) sert à la lecture, tandis que le deuxième (`fd[1]`) sert à l'écriture.**

- **Le canal est de type « byte stream », c.à.d. sans concept de trames délimitant le début et la fin d'un message.**
- **Pour la mise en œuvre des « pipes », Linux propose les services suivants**
  - Création d'un pipe (syscall: pipe)
  - Lecture de données (syscall: read)
  - Ecriture de données (syscall: write)
  - Fermeture de pipe (syscall: close)

Communication – Pipes – Création d'un pipe

- **Pour créer un pipe, Linux propose l'appel système `pipe()`.**

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

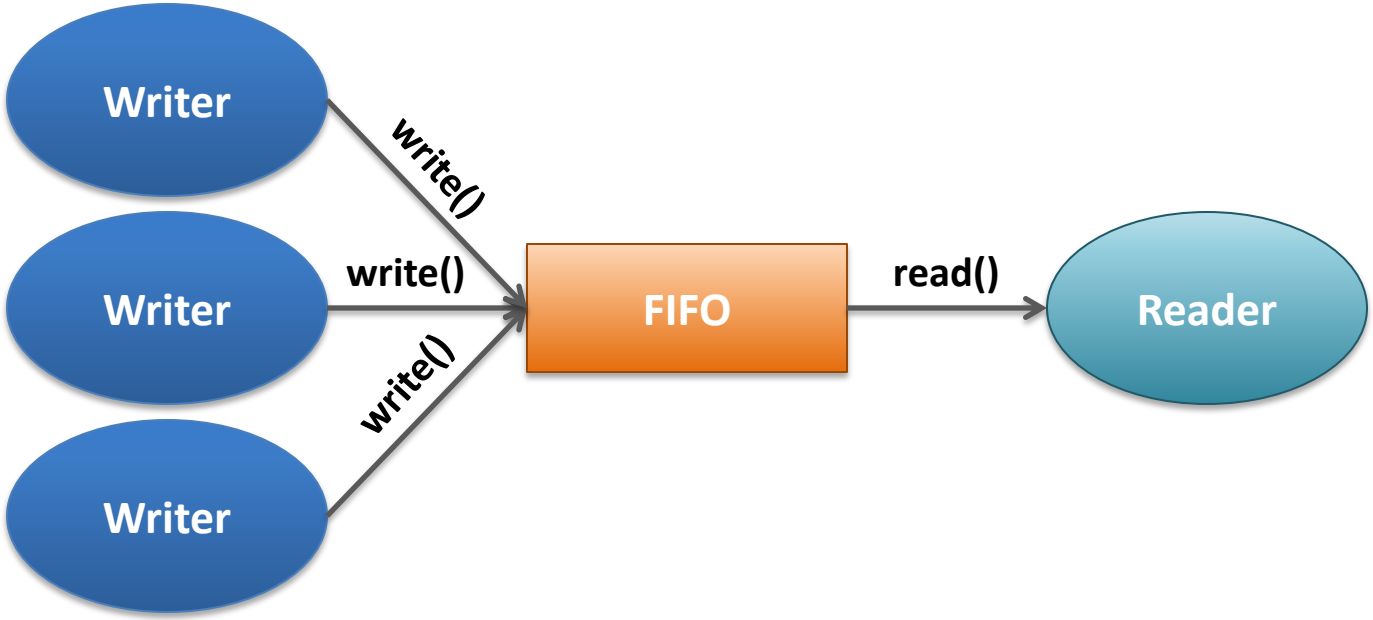
- **Exemple**

```
int fd[2];
int err = pipe(fd);   // create unidirectional pipe
if (err == -1)
     /* error*/
pid_t pid = fork();
if (pid == 0) {       // child
     close(fd[0]);    // close unused read descriptor
     write (fd[1], msg, sizeof(msg));
} else {              // parent
     close(fd[1]);    // close unused write descriptor
     int len = read (fd[0], msg, sizeof(msg));
}
```

- **Comportement**
  - La fonction `pipe()` crée un pipe et retourne deux descripteurs, le 1er `fd[0]` pour la réception de données et le 2e `fd[1]` pour l'envoi de données.

Communication – FIFO – Named Pipes

- Les FIFO (named pipes) implémentent un canal de communication unidirectionnel généralement entre un processus récepteur et un ou plusieurs processus émetteur.
- Le FIFO est un fichier spécial situé dans le système de fichiers virtuels. Il peut être accédé par les opérations utilisées sur les fichiers ordinaires, pour autant de disposer des droits d'accès nécessaires.



- Le canal est de type « byte stream », c.à.d. sans concept de trames délimitant le début et la fin d'un message

Communication – FIFO – Opérations

- **Avant de pouvoir utiliser un FIFO, celui-ci doit être créé soit avant le lancement des applications le mettant en œuvre soit par les applications elles-mêmes.**
  - Création d'un FIFO à l'aide de commandes utilisateur

```
$ mkfifo [--mode=MODE] FIFO_NAME
```

  - Création d'un FIFO par l'application

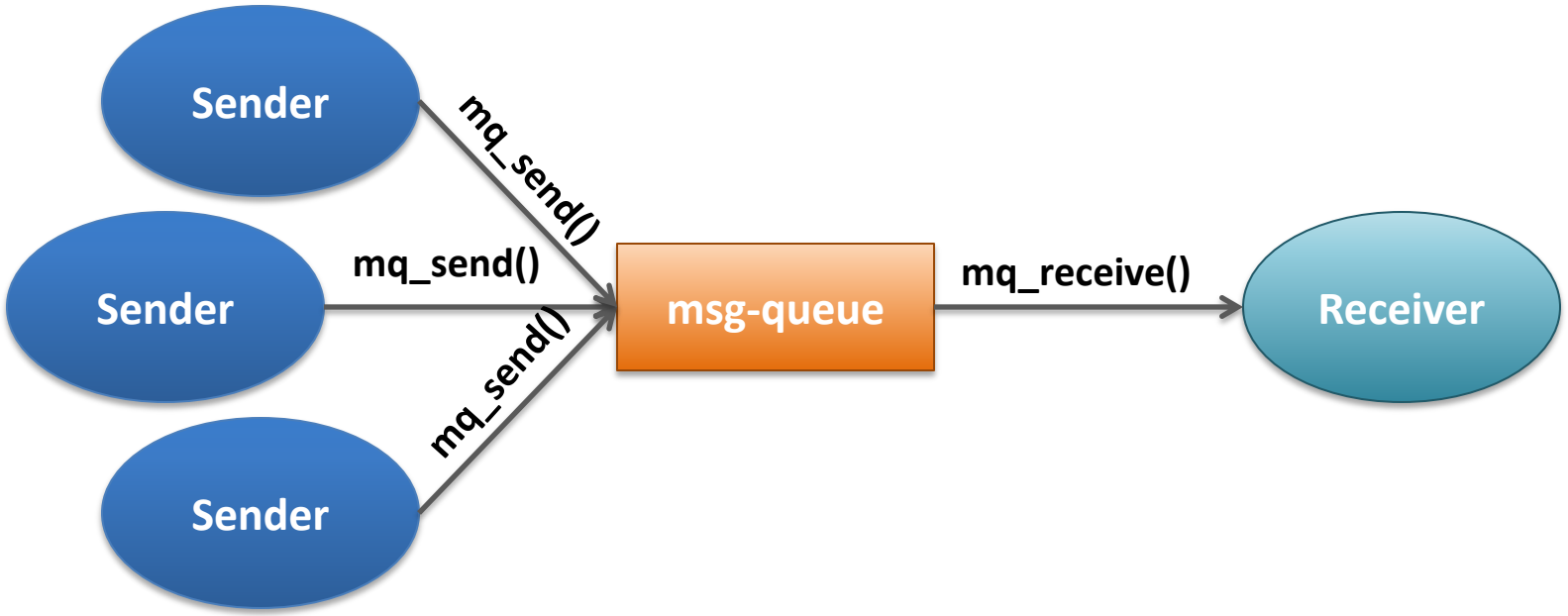
```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char* pathname, mode_t mode);
```

- **Pour accéder au FIFO, Linux propose les services utilisés pour les fichiers ordinaux**
  - Ouverture du FIFO (syscall: open)
  - Lecture du FIFO (syscall: read)
  - Ecriture dans le FIFO (syscall: write)
  - Fermeture du FIFO (syscall: close)
  - Destruction du FIFO (syscall: unlink)

MASTER OF SCIENCE IN ENGINEERING
MSE

Communication – Message Queues

- Les files d'attente de messages (message queues) offrent un canal de communication unidirectionnel généralement entre un processus récepteur et un ou plusieurs processus émetteurs.
- Contrairement aux pipes et aux FIFO, les files d'attende de messages sont de type datagramme, c.à.d. chaque message ou paquet est stocké de façon indépendante dans la file d'attente.
- La taille maximale d'un message ainsi que le nombre de messages pouvant être stockés dans la file d'attende sont déterminés lors de sa création.



Communication – Message Queues – Opérations

- **Pour accéder aux services des files d'attende de messages, Linux propose une série de méthodes spécifiques avec la bibliothèque `<mqueue.h>`.**
- **Opérations**
  - Ouverture et création d'une file d'attende (syscall: mq_open)
  - Lecture d'un message d'une file (syscall: mq_receive)
  - Ecriture d'un message dans une file (syscall: mq_send)
  - Fermeture d'une file d'attente (syscall: mq_close)
  - Destruction d'une file d'attente (syscall: mq_unlink)
  - Lecture des attributs d'une file d'attente (syscall: mq_getattr)
- **La `struc mq_attr` permet de décrire les caractérisques d'une file d'attente**

```
struct mq_attr {
   long mq_flags;      // flags: 0 or O_NONBLOCK
   long mq_maxmsg;     // maximum number of messages on queue
   long mq_msgsize;    // maximum message size in bytes
   long mq_curmsgs;    // number of messages currently in queue
};
```

Communication – Sockets

- **Les sockets ont été conçus par l'université de Berkeley au début des années 1980. Ils proposent un ensemble de services normalisés pour l'échange d'information entre processus locaux ou distants.**
- **Le socket fournit une prise permettant aux différents processus d'une application ou d'un système d'envoyer et de recevoir des données. Cette prise est très polyvalente et permet d'interfacer une large palette de protocole, dont le plus fréquemment utilisé TCP/IP.**
- **Les sockets proposent deux grands modes pour le transfert de données**
  - SOCK_STREAM : bidirectionnel, sûr/fiable, flux de données
  - SOCK_DGRAM : bidirectionnel, pas sûr/fiable, paquet de données
- **Opérations**
  - Création d'un socket (syscall: socket)
  - Liaison du socket à une adresse donnée (syscall: bind)
  - Connexion du socket à un socket distant (syscall: connect)
  - Ecoute sur de nouvelles connexions (syscall: listen)
  - Acception d'une nouvelle connexion (syscall: accept)
  - Envoi de données (syscall: write, send, sendto)
  - Réception de données (syscall: read, recv, recvfrom)
  - Fermeture du socket (syscall: close)

Communication – Socketpair

- `socketpair` **offre un service bidirectionnel très intéressant pour l'échange de données entre deux processus.**

```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair (int domain, int type, int protocol, int fd[2]);
```

- **Exemple**

```
int fd[2];
int err = socketpair(AF_UNIX, SOCK_STREAM, 0, fd);
if (err == -1)
    /* error*/
```

- **Comportement**
  - La fonction `socketpair()` crée un canal de communication bidirectionnelle sous la forme d'un socket Unix (le seul supporté par Linux) et retourne deux descripteurs, le 1er `fd[0]` pour un processus et le 2e `fd[1]` pour le deuxième processus. Ces descripteurs permettent d'émettre des données avec la méthode `write()` et d'en recevoir avec la méthode `read()`.

Ordonnanceur (scheduler)

MASTER OF SCIENCE IN ENGINEERING

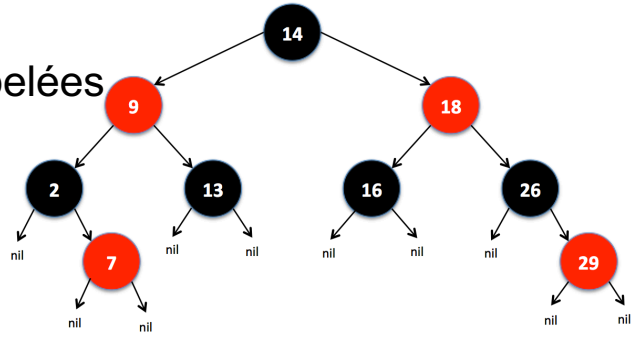
MSE

Ordonnanceur – Introduction

- Linux implémente un double ordonnanceur, soit un ordonnanceur pour les processus normaux et un ordonnanceur pour les processus temps réel.
- Les processus normaux sont des processus/applications exigeant un partage équitable du processeur et de ses ressources, soit la majorité des processus d'une machine Linux standard (desktop ou server).
- Les processus temps réel sont des processus/daemons avec de grandes contraintes temporelles et/ou d'accès aux périphériques. De tels processus sont courants dans les systèmes embarqués où les exigences sont fortes.
- L'ordonnanceur de Linux est préemptif. Il implémente 40 niveaux de priorités pour les processus normaux et 100 priorités pour les processus temps réel.

Ordonnanceur – CFS – Completely Fair Scheduler

- **L'ordonnanceur pour les processus normaux a énormément évolué. Depuis la version 2.6.23, Linux implémente un nouvel ordonnanceur appelé « CFS – Completely Fair Scheduler ».**
- **Les caractéristiques de cet ordonnanceurs sont**
  - Répartition fair des ressources du processeur en garantissant une proportion identique du temps processeur à chaque processus, p.ex. pour un système à N processus, chaque processus recevra 1/N du temps processeur
  - Implémente un arbre bicolore (red-black tree) pour la recherche du prochain processus à faire tourner.
  - Les 40 priorités (-20 à 19) de cet ordonnanceur, appelées « nice level » permettent d'attribuer une plus ou moins grande proportion du temps CPU aux différents processus. La priorité -20 représente la priorité la plus haute et 19 la plus basse. Par défaut un processus obtient la priorité 0.



  - 2 « policies »peuvent être choisies lors de la création d'un processus
    - SCHED_NORMAL : processus pour une exécution standard en round-robin
    - SCHED_BATCH : processus pour une exécution de type « batch »

MASTER OF SCIENCE IN ENGINEERING

Ordonnanceur – Realtime Scheduler

- **L'ordonnanceur temps réels est prioritaire sur l'ordonnanceur CFS.**
- **Les caractéristiques de cet ordonnanceurs sont**
  - Répartition des ressources du processeur selon un schéma basé strictement sur le niveau de priorité des processus, c.à.d. un processus d'une certaine priorité ne peut être interrompu que par un processus de plus haute priorité.
  - Cet ordonnanceur dispose de 100 priorités (0 à 99). La priorité 0 est la plus basse et 99 la plus haute. La priorité 0 est réservée aux processus normaux et gérée par l'ordonnanceur CFS.
  - 2 stratégies peuvent être choisies lors de la création d'un processus
    - SCHED_FIFO (first in first out)
      Un processus de cette catégorie ne peut être interrompu que par un processus de plus haute priorité. Si un tel processus appelle un appel système bloquant, l'ordonnanceur le retire de la liste des processus « runnable ». Lorsqu'il devient à nouveau « runnable », il est placé en fin de queue.
    - SCHED_RR (round-robin)
      Les processus de cette catégorie sont similaires aux processus FIFO, à l'exception qu'une fenêtre de temps leur est assignée. Lorsque celle-ci est écoulée, le processus est replacé en fin de queue.

MASTER OF SCIENCE IN ENGINEERING
MSE

Ordonnanceur – Opérations

- **Pour gérer les processus temps réel et leur niveau de priorité, Linux propose divers services.**
- **Opérations**
  - Obtention de la « policy » d'un processus (syscall: sched_getscheduler)
  - Affection d'une « policy » à un processus (syscall: sched_setscheduler)
  - Lecture de l'intervalle « round-robin » (syscall: sched_rr_get_interval)
  - Passation du processeur à un autre processus (syscall: sched_yield)

MASTER OF SCIENCE IN ENGINEERING

MSE

Ordonnanceur – Gestion de la « policy » d'un processus

- **Linux propose deux services pour la gestion de la « policy » d'un processus, soit l'appel système `sched_getscheduler()` et `sched_setscheduler()`.**

```
#include <sched.h>
int sched_getscheduler(pid_t pid);
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *sp);
```

- **Exemple**

```
struct sched_param sp = {.sched_priority = 50, };
int ret = sched_setscheduler (0, SCHED_RR, &sp);
if (ret == -1)
   /* error */
```

- **Comportement**
  - La fonction `sched_setscheduler()` permet de contrôler la « policy » de l'ordonnanceur attaché un processus, soit le type d'ordonnancement et le niveau de priorité. L'argument `pid` indique l'identifiant du processus (la valeur 0 indique le processus courant). Le 2e argument policy permet de spécifier le type d'ordonnanceur (`SCHED_FIFO` ou `SCHED_RR`). L'attribut `sched_priority` de la `struct sched_param` permet de fixer le niveau de priorité du processus.
  - Pour utiliser ce service, le processus doit disposer des privilèges suffisants.

Ordonnanceur – CFS – Operations

- **Pour gérer les priorités des processus normaux de l'ordonnanceur CFS, Linux propose divers services.**
- **Opérations**
  - Gestion des priorités « nice » (syscall: nice)
  - Lecture du niveau de priorité (syscall: getpriority)
  - Affectation du niveau de priorité (syscall: setpriority)

MASTER OF SCIENCE IN ENGINEERING

MSE

Ordonnanceur – CFS – Gestion des priorités « nice »

- **Un processus placé dans la catégorie des processus normaux peut gérer son taux d'affectation du µP à l'aide de l'appel système `nice()`.**

```
#include <unistd.h>
int nice(int inc);
```

- **Exemple**

```
errno = 0;
int prio = nice(1); // increase by 1 nice level
if (prio == -1 && errno != 0)
   /* error */
```

- **Comportement**
  - La fonction `nice()` permet de diminuer (`inc` positif) ou d'augmenter (`inc` négatif) le niveau de priorité du processus courant et retourne le nouveau niveau de priorité du processus. Si `inc` est 0, la fonction retourne le niveau de priorité actuel du processus.
  - Pour augmenter (`inc` négatif) son niveau de priorité, le processus doit impérativement disposer des privilèges `root`.
  - Depuis la ligne de commande, il est possible de lancer une application avec un niveau de priorité défini, p. ex.
    `$ nice –n -20 ./my_appl`

MASTER OF SCIENCE IN ENGINEERING
MSE

Ordonnanceur – CFS – Gestion du niveau de priorité

- **Linux propose deux services pour la gestion du niveau de priorité d'un processus, soit l'appel système `getpriority()` et `setpriority()`.**

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio)
```

- **Exemple**

```
errno = 0;
int ret = setpriority(PRIO_PROCESS, getpid(), 2); // nice level = 2
if (ret == -1)
   /* error */
```

- **Comportement**
  - La fonction `setpriority()` permet de contrôler le niveau de priorité d'un processus (`which = PRIO_PROCESS`), d'un groupe de processus (`which = PRIO_PGRP`) ou d'un utilisateur (`which = PRIO_USER`). L'identifiant du processus, groupe de processus ou utilisateur est spécifié avec l'argument `who`. Le paramètre `prio` indique le niveau de priorité nice (-20 à 19).
  - Pour augmenter son niveau de priorité, le processus doit disposer des privilèges suffisants.

Ordonnanceur – Processor Affinity

- **Par défaut, Linux implémente un mécanisme de « load balancing » pour la gestion des CPU de processeurs multicoeurs. Cependant, il arrive que pour des applications il soit utile et nécessaire d'attribuer un ou plusieurs CPU à un processus ou groupe de processus. A cet effet, Linux propose une série de macros et services.**
- **Opérations**
  - Lecture de l'affectation d'un CPU (syscall: sched_getaffinity)
  - Affectation d'un CPU à un processus (syscall: sched_setaffinity)
- **Les methodes ci-dessous permettent de définir un ensemble de CPU à affecter à un processus**

```
#define _GNU_SOURCE
#include <sched.h>
void CPU_SET(unsinged long cpu, cpu_set_t *set);
void CPU_CLR(unsinged long cpu, cpu_set_t *set);
int CPU_ISSET(unsinged long cpu, cpu_set_t *set);
void CPU_ZERO(cpu_set_t *set);
```

MASTER OF SCIENCE IN ENGINEERING
MSE

Ordonnanceur – Processor Affinity – Affection d'un CPU ou groupe de CPU

- **Un CPU ou groupe de CPU peut être assigné à processus par l'intermédiaire de l'appel système `sched_setaffinity`.**

```
#define _GNU_SOURCE
#include <sched.h>
int sched_setaffinity(pid_t pid, size_t setsize, const cpu_set_t *set);
```

- **Exemple**

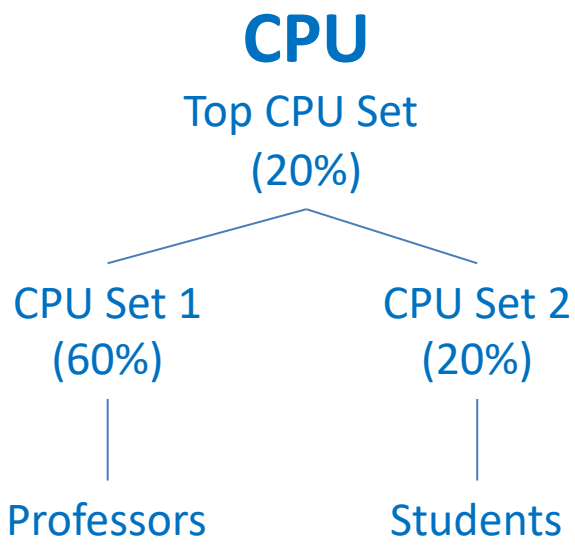
```
cpu_set_t set;
CPU_ZERO(set);
CPU_SET(1, &set);
int ret = sched_setaffinity(0, sizeof(set), &set);
if (ret == -1)
   /* error */
```

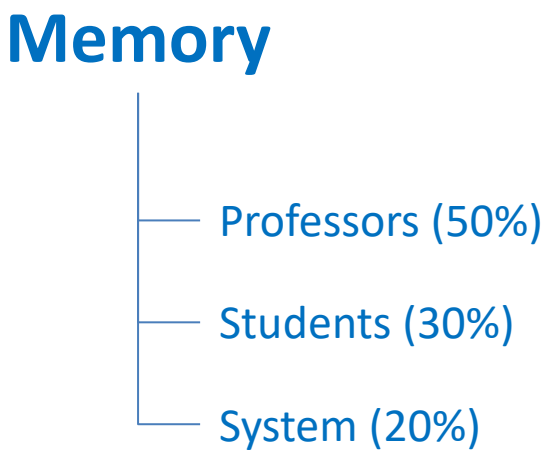
- **Comportement**
  - La fonction `sched_setaffinity()` permet d'assigner un ou plusieurs CPU (dans notre cas le CPU 1) à un processus. L'argument `pid` indique l'identifiant du processus (la valeur 0 indique le processus courant).

Control Groups (CGroups)

CGroups – Introduction

- La gestion des ressources des µP pour les différentes tâches que le système doit assurer est un problème récurrent sur les systèmes embarqués.
- Les groupes de contrôle (CGroups) offrent un mécanisme puissant pour limiter, compter et isoler les ressources du µP, p.ex. les cœurs, la mémoire, utilisation des I/O.
- Les CGroups permettent de partitionner le système en différentes activités. Ces dernières seront ensuite agrégées dans des groupes hiérarchiques avec des comportements spécialisés.





MASTER OF SCIENCE IN ENGINEERING
MSE

CGroups – Subsystems

- **Un sous-système représente une ressource du µP, p. ex. le temps CPU ou la capacité d'une mémoire.**
- **Les CGroups proposent**
  - **cpuset** – pour assigner des CPU individuels (multi-cœurs) et des nœuds de mémoire à des tâches dans un groupe de contrôle
  - **cpu** – pour fournir aux tâches des groupes de contrôle accès au CPU
  - **memory** – pour établir les limites d'utilisation de la mémoire par les tâches d'un groupe de contrôle et pour générer des rapports automatiques sur les ressources mémoire utilisées
  - **blkio** – pour établir des limites sur l'accès des entrées/sorties à partir et depuis des périphériques blocs tels que des lecteurs physiques (disques, lecteurs USB, etc.)
  - **cpuacct** – pour générer des rapports automatiques sur les ressources CPU utilisées
  - **devices** – pour autoriser ou refuser l'accès aux périphériques
  - **freezer** – pour suspendre ou réactiver les tâches dans un groupe de contrôle
  - **net_cls** – pour repérer les paquets réseau avec un identifiant de classe (classid) qui permet au contrôleur de trafic Linux (tc) d'identifier les paquets provenant d'une tâche particulière d'un groupe de contrôle
  - **ns** – le sous-système namespace
- **Documentation: <Linux sources>/Documentation/cgroups**

CGroups – Configuration du noyau Linux

- **Pour utiliser les CGroups, les flags suivants doivent être mis en place dans la configuration du noyau Linux**
- **Commande**

```
$ make linux-menuconfig

General setup --->
[*] Control Group support --->
      [*]  Cpuset support
      [*]  Simple CPU accounting cgroup subsystem
      [*]  Device controller for cgroups
      [*]  Resource counters
      [*]     Memory Resource Controller for Control Groups
      [*]  Group CPU scheduler --->
            [*] Group scheduling for SCHED_OTHER
            [*] Group scheduling for SCHED_RR/FIFO
[ ] Automatic process group scheduling
[*] Enable the block layer --->
[*] Block layer bio throttling support
    IO Schedulers --->
            [*] CFQ Group Scheduling support
```

CGroups – Utilisation des CGroups

- **Les CGroups sont disponible en espace utilisateur. Pour cela il suffit de les monter dans l'arborescence du rootfs.**

```
$ mount -t tmpfs none /sys/fs/cgroup
$ mkdir /sys/fs/cgroup/set
$ mount -t cgroup -o memory,cpu,cpuset cgroups /sys/fs/cgroup/set
```

- **L'option –o permet de choisir les sous-systèmes que l'on souhaite exporter. Si aucune option n'est spécifiée, l'ensemble des sous-systèmes seront visibles.**
- **Ensuite il suffit de créer la hiérarchie souhaitée, de configurer les ressources allouées à ces différents groupes de contrôle et de placer les différents processus dans cette hiérarchie.**
- **Exemple pour un programme**

```
$ mkdir /sys/fs/cgroup/set/program

$ echo 2-3 > /sys/fs/cgroup/set/program/cpuset.cpus
$ echo 0   > /sys/fs/cgroup/set/program/cpuset.mems

$ echo 20M > /sys/fs/cgroup/set/program/memory.limit_in_bytes

$ echo PID(high-program) > /sys/fs/cgroup/set/program/tasks
```

MASTER OF SCIENCE IN ENGINEERING

MSE