Systèmes d'exploitation pour l'embarqué

UV 5.2 - Exécution et Concurrence

Paul Blottière

ENSTA Bretagne

10 Novembre 2015

https://github.com/pblottiere

Amélioration continue

Contributions

- Dépôt du cours : `https://github.com/pblottiere/embsys`

Amélioration continue

Contributions

- Dépôt du cours : `https://github.com/pblottiere/embsys`
- Souhaits d'amélioration, erreurs, idées de TP, ... : ouverture d'Issues (avec le bon label!)
- Apports de corrections : Pull Request

Les processus et les threads

Plan

1. Définitions
2. Outils
3. Appels système
4. Identification (PID, ...)
5. Capacités d'un processus
6. Création de processus
7. Terminaison d'un processus
8. Les pthreads

Définitions (1)

Programme, Processus et Threads

Programme : fichier exécutable, enregistré sur le disque

Définitions (1)

Programme, Processus et Threads

Programme : fichier exécutable, enregistré sur le disque

Processus :

- programme en cours d'exécution
- disposent chacun d'un espace mémoire indépendant et protégé des autres processus (MMU)
- monothread ou multithread

Définitions (1)

Programme, Processus et Threads

Programme : fichier exécutable, enregistré sur le disque

Processus :

- programme en cours d'exécution
- disposent chacun d'un espace mémoire indépendant et protégé des autres processus (MMU)
- monothread ou multithread

Thread:

- les threads d'un même processus partagent le même espace mémoire
- ne partagent pas les informations d'ordonnancement

Définitions (2)

Ordonnanceur

Linux est un kernel préemptif (capacité à exécuter ou stopper une tâche en cours) : imite un comportement multitâche.

Définitions (2)

Ordonnanceur

Linux est un kernel préemptif (capacité à exécuter ou stopper une tâche en cours) : imite un comportement multitâche.

L'ordonnanceur distribue le temps CPU entre les différents processus selon des politiques d'ordonnancement.

[Exemple d'ordonnancement de tâches]

Définitions (3)

Ordonnanceur

Il existe plusieurs politiques d'ordonnancement :

- FIFO (First In First Out) : file d'attente avec niveaux de priorité
- RR (Round Robin) : idem que FIFO mais avec en plus un quantum de temps (un processus dépassant ce quantum est mis en veille si une tâche ayant un même niveau de priorité est prête)
- OTHER : priorité dynamique recalculée en fonction de la priorité par défaut et du travail réalisé pendant le quantum

Définitions (4)

Process Control Block

Un processus est représenté par un PCB (linux/sched.h).

Définitions (4)

Process Control Block

Un processus est représenté par un PCB (linux/sched.h).

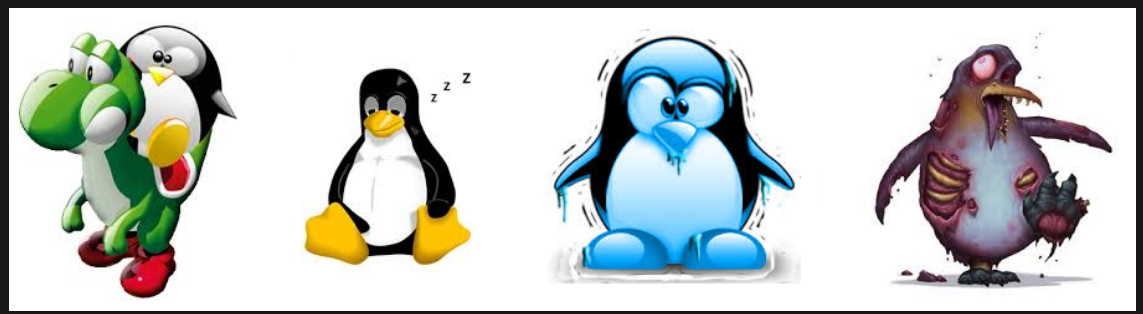
Les éléments principaux de cette structure :

- run_list : pointeur vers les processus suivants/précédents de la runqueue
- sibling : pointeurs vers les processus de même père
- sleep_avg : temps moyen dans l'état Sleeping
- policy : OTHER, RR, FIFO
- pid : process identifier
- parent : process père
- children : liste des processus fils

Définitions (5)

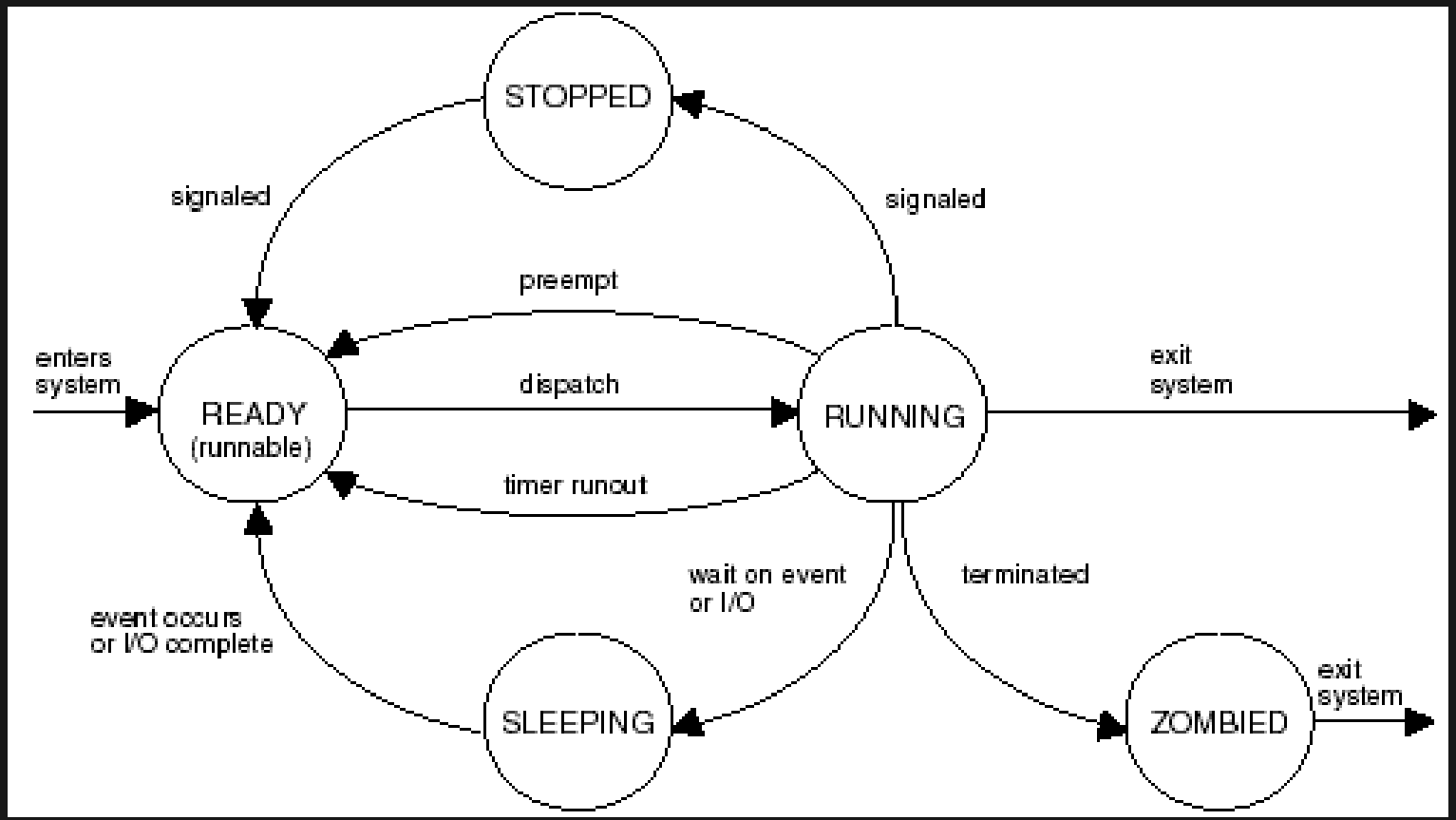
Les états

- Running (R) : en cours d'exécution
- Sleeping (S) : non actif mais susceptible d'être réveillé par un évènement
- Stopped (T) : tâche temporairement arrêté. Attend un signal de redémarrage
- Zombie (Z) : tâche terminée mais code de retour non lu



Définitions (6)

Les états



Définitions (7)

Limites

Un processus est évidemment limité en ressource!

Définitions (7)

Limites

Un processus est évidemment limité en ressource!

Champ *rlim* de la structure *signals* du PCB :

- RLIMIT_AS : taille max d'adressage (vérifiée lors d'un malloc)
- RLIMIT_CORE : taille max du core dump
- RLIMIT_CPU : temps CPU max en secondes
- ...

Outils (1)

Lister les processus en cours : commande ps

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus$ ps u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
tergeist  2944  0.0  0.1   7104  5328 pts/0    Ss   13:35   0:00 bash
tergeist  3317  0.0  0.1   7056  5152 pts/2    Ss+  13:36   0:00 bash
tergeist  3954  0.0  0.1   7088  5240 pts/1    Ss   13:38   0:00 bash
tergeist  3974  2.9  1.4 336804 43256 pts/1    Sl+  13:38   5:21 vim README.md
tergeist  4266  0.3  2.5 233184 78876 pts/0    Sl   13:39   0:33 evince processus.pdf
tergeist 27854  0.0  0.1   7620  3480 pts/0    R+   16:41   0:00 ps u
```

Outils (1)

Lister les processus en cours : commande ps

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus$ ps u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
tergeist  2944  0.0  0.1   7104  5328 pts/0    Ss   13:35   0:00 bash
tergeist  3317  0.0  0.1   7056  5152 pts/2    Ss+  13:36   0:00 bash
tergeist  3954  0.0  0.1   7088  5240 pts/1    Ss   13:38   0:00 bash
tergeist  3974  2.9  1.4 336804 43256 pts/1    Sl+  13:38   5:21 vim README.md
tergeist  4266  0.3  2.5 233184 78876 pts/0    Sl   13:39   0:33 evince processus.pdf
tergeist 27854  0.0  0.1   7620  3480 pts/0    R+   16:41   0:00 ps u
```

Les champs principaux :

- USER : propriétaire du processus
- PID : numéro d'identification du processus
- CPU : pourcentage du temps CPU consacré
- MEM : mémoire totale utilisée par le processus
- TTY : terminal associé. Un *?* si il n'y en a pas
- STAT : code d'état du process
- TIME : temps CPU utilisé
- COMMAND : nom de la commande

Outils (2)

Diverses commandes

- nohup : ignore les déconnexion SIGHUP
- nice / renice : affectation de priorité (changement dans l'ordonnancement)
- kill / killall / pkill : terminaison de processus
- crontab : lancement programmé de processus
- cpulimit : limite le temps CPU d'un processus

Outils (3)

/proc

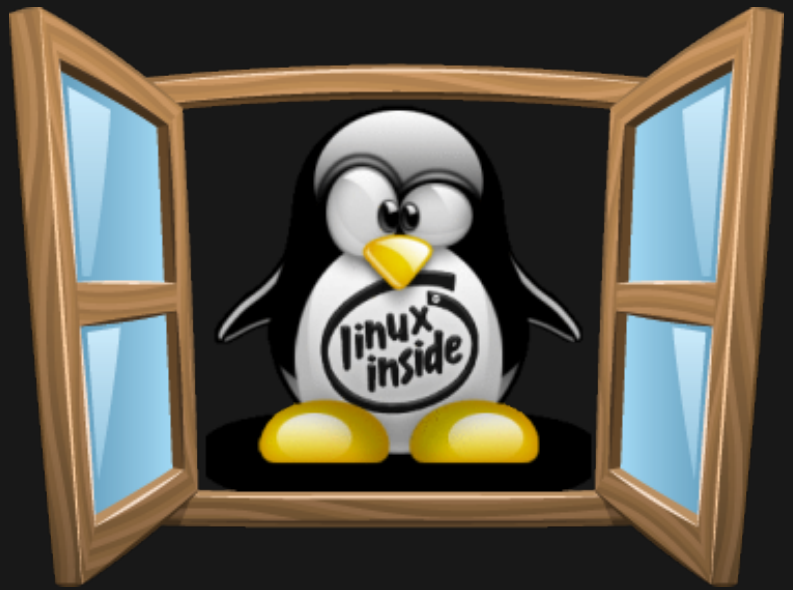
/proc est un système de fichier virtuel : les fichiers sont générés à la volée par le kernel!

Outils (3)

/proc

/proc est un système de fichier virtuel : les fichiers sont générés à la volée par le kernel!

/proc est une fenêtre sur le Kernel à travers laquelle on peut récupérer des informations sur le système et configurer certains comportements.



Outils (4)

/proc

Il existe un répertoire /proc/<PID> par processus qui contient toutes les informations le concernant :

- cmdline : ligne de commande
- limits : limites des ressources
- mem : mémoire tenue par le processus
- d'autres : `http://man7.org/linux/man-pages/man5/proc.5.html`

Outils (4)

/proc

Il existe un répertoire /proc/<PID> par processus qui contient toutes les informations le concernant :

- cmdline : ligne de commande
- limits : limites des ressources
- mem : mémoire tenue par le processus
- d'autres : `http://man7.org/linux/man-pages/man5/proc.5.html`

```
tergeist@multi:~$ ps -a
  PID TTY          TIME CMD
 3974 pts/1    00:07:25 vim
 4266 pts/0    00:00:45 evince
 4412 pts/2    00:00:00 ps
tergeist@multi:~$ cat /proc/4266/stat
4266 (evince) S 2944 4266 2944 34816 2944 4202496 897866 0 0 0 4054 46
4 0 0 20 0 6 0 48238 240545792 20164 4294967295 2147889152 2148336264
3214446608 3214446084 3077569504 0 0 4224 0 4294967295 0 0 17 0 0 0 0
0 0 2148342828 2148352540 2164310016 3214448382 3214448403 3214448403
3214450668 0
tergeist@multi:~$
```

Appels Système

Qu’est-ce?

Syscalls : Interface fondamentale entre le Kernel et les programmes de l’espace utilisateur et fournit via des wrappers de la libc.

Appels Système

Qu'est-ce?

Syscalls : Interface fondamentale entre le Kernel et les programmes de l'espace utilisateur et fournit via des wrappers de la libc.

Liste des appels système SUSv4 :

`http://pubs.opengroup.org/onlinepubs/9699919799/`

Appels Système

Qu'est-ce?

Syscalls : Interface fondamentale entre le Kernel et les programmes de l'espace utilisateur et fournit via des wrappers de la libc.

Liste des appels système SUSv4 :

`http://pubs.opengroup.org/onlinepubs/9699919799/`

Par exemple, *unistd.h* est défini pour tous les UNIX et donne accès à des fonctions de l'API POSIX (read, write, fork, ...).

Identification (1)

PID et PPID

Systèmes UNIX : règles précises concernant l'identification des utilisateurs et des processus!

Identification (1)

PID et PPID

Systèmes UNIX : règles précises concernant l'identification des utilisateurs et des processus!

Les appels système POSIX associés :

- pid_t getpid (void) : entier 32 bits sous Linux. PID du processus courant.
- pid_t getppid (void) : PID du père.

Identification (2)

PID et PPID (pid.c)

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

int main()
{
    pid_t pid_child = getpid();
    pid_t pid_father = getppid();

    printf("PID: %ld\n", (long) pid_child);
    printf("PPID: %ld\n", (long) pid_father);

    return EXIT_SUCCESS;
}
```

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ ./pid
PID: 13175
PPID: 9174
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ echo $$
9174
```

Identification (3)

UID

Chaque processus s'exécute sous une identité propre. Il existe trois identifiants d'utilisateurs par processus :

- UID réel : UID de l'utilisateur ayant lancé le programme
- UID effectif : indique les privilèges accordés au processus (flag setuid)
- UID sauvé : copie de l'ancien UID effectif lorsque celui-ci est modifié par le processus (automatique par le kernel)

Identification (3)

UID

Chaque processus s'exécute sous une identité propre. Il existe trois identifiants d'utilisateurs par processus :

- UID réel : UID de l'utilisateur ayant lancé le programme
- UID effectif : indique les privilèges accordés au processus (flag setuid)
- UID sauvé : copie de l'ancien UID effectif lorsque celui-ci est modifié par le processus (automatique par le kernel)

Les appels système associés :

- POSIX : getuid / geteuid / setuid / setreuid
- Linux : setresuid

Identification (4)

Groupe d'utilisateur, groupe de processus et groupe de groupe

Il existe encore beaucoup de méthodes d'identification :

- par groupe d'utilisateurs du processus : GID
- par groupe de processus : PGID
- par session : SID

Capacités d'un processus

Privilèges

Depuis la version 2.2 du Kernel Linux, chaque processus possède un jeu de capacités définissant précisément ses privilèges :

- CAP_SYS_BOOT : shutdown autorisé
- CAP_SYS_RAWIO : accès aux ports d'entrées / sorties
- d'autres : `http://manpagesfr.free.fr/man/man7/capabilities.7.html`

Capacités d'un processus

Privilèges

Depuis la version 2.2 du Kernel Linux, chaque processus possède un jeu de capacités définissant précisément ses privilèges :

- CAP_SYS_BOOT : shutdown autorisé
- CAP_SYS_RAWIO : accès aux ports d'entrées / sorties
- d'autres : `http://manpagesfr.free.fr/man/man7/capabilities.7.html`

Pour cela, utiliser la librairie libcap!

Création de processus (1)

fork

fork est un appel système qui :

- duplique le processus appelant (PID différent)
- les processus fils ne partagent pas le même espace mémoire que le père
- a un cout très faible en ressources (mécanisme COW)

Création de processus (1)

fork

fork est un appel système qui :

- duplique le processus appelant (PID différent)
- les processus fils ne partagent pas le même espace mémoire que le père
- a un cout très faible en ressources (mécanisme COW)

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ ./fork
Father: PID=16401 / PPID=9174
Child: PID=16402 / PPID=16401
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ echo $$
9174
```

Création de processus (2)

fork (fork.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>

int main()
{
    pid_t pid_fork = fork();
    if (pid_fork == -1)
    {
        printf("fork error (%s)", strerror(errno));
        exit(EXIT_FAILURE);
    }

    if (pid_fork == 0)
    {
        printf("Child: PID=%ld / PPID=%ld\n", (long) getpid(), (long) getppid());
        return EXIT_SUCCESS;
    }
    else
    {
        printf("Father: PID=%ld / PPID=%ld\n", (long) getpid(), (long) getppid());
        wait(NULL);
        return EXIT_SUCCESS;
    }
}
```

Création de processus (3)

vfork

vfork est un appel système qui :

- créé un processus qui partage le même espace mémoire que le père
- bloque le père jusqu'à la terminaison du fils

Création de processus (3)

vfork

vfork est un appel système qui :

- créé un processus qui partage le même espace mémoire que le père
- bloque le père jusqu'à la terminaison du fils

Peut créer des problèmes d'inversion de privilège car le fils n'est pas privilégié même si le père l'est : ne pas utiliser vfork!

=> relevé dans un audit de sécurité de Linux en 2000!

Création de processus (4)

clone, execve, system, popen, ...

Il existe de nombreux autres moyens d'initier un processus :

- **clone** : paramétrage précis du comportement du fils
- **execve** : espace mémoire totalement remplacé
- **system** : exécute une commande shell (à bannir bien sûr...)
- **popen / pclose** : pipe + fork + invocation dans le shell (idem)

Terminaison d'un processus (1)

Généralité

Un processus peut se terminer :

- volontairement : tâche terminée, erreur gérée, ...
- involontairement : erreur non gérée, Ctrl-C, ...

Terminaison d'un processus (1)

Généralité

Un processus peut se terminer :

- volontairement : tâche terminée, erreur gérée, ...
- involontairement : erreur non gérée, Ctrl-C, ...

Un processus retourne un code d'erreur. Pour être portable :

- un main doit toujours retourner un **int**!
- utiliser EXIT_SUCCESS et EXIT_FAILURE

Terminaison d'un processus (1)

Généralité

Un processus peut se terminer :

- volontairement : tâche terminée, erreur gérée, ...
- involontairement : erreur non gérée, Ctrl-C, ...

Un processus retourne un code d'erreur. Pour être portable :

- un main doit toujours retourner un **int**!
- utiliser EXIT_SUCCESS et EXIT_FAILURE

Ce code d'erreur est ensuite lu par le père :

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ ls
fork  fork.c  Makefile  pid  pid.c
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ echo $?
0
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ ls -w
ls : l'option requiert un argument -- w
Saisissez « ls --help » pour plus d'informations.
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ echo $?
2
```

Terminaison d'un processus (2)

Je vais bien, tout va bien...

Pour une terminaison normale, plusieurs appels système :

- **exit** : termine le programme avec un code de retour.
- **atexit** : permet de spécifier des fonctions à appeler en fin d'exécution
- **on_exit** : comme atexit mais beaucoup moins standard (pas sur OSX) donc préférer atexit

Terminaison d'un processus (2)

Je vais bien, tout va bien...

Pour une terminaison normale, plusieurs appels système :

- **exit** : termine le programme avec un code de retour.
- **atexit** : permet de spécifier des fonctions à appeler en fin d'exécution
- **on_exit** : comme atexit mais beaucoup moins standard (pas sur OSX) donc préférer atexit

File d'exécution de **exit** :

1. exécute les fonctions enregistrées par atexit / on_exit
2. ferme les flux d'entrées / sorties
3. appel _exit qui termine le processus et retourne le code d'erreur

Terminaison d’un processus (3)

Terminaison anormale

Lors d’une terminaison anormale, un fichier core est généré.

Ce fichier est une image mémoire de l’instant de l’anomalie permettant de rejouer le scénario (avec gdb)!

Terminaison d'un processus (3)

Terminaison anormale

Lors d'une terminaison anormale, un fichier core est généré.

Ce fichier est une image mémoire de l'instant de l'anomalie permettant de rejouer le scénario (avec gdb)!

On peut modifier le pattern de création du fichier core!

```
> cat /proc/sys/kernel/core_pattern
core
```

Terminaison d'un processus (3)

Terminaison anormale

Lors d'une terminaison anormale, un fichier core est généré.

Ce fichier est une image mémoire de l'instant de l'anomalie permettant de rejouer le scénario (avec gdb)!

On peut modifier le pattern de création du fichier core!

```
> cat /proc/sys/kernel/core_pattern
core
```

Pour créer des fichiers core sans limite de taille :

```
> ulimit -c unlimited
```

Terminaison d'un processus (4)

Signaux

Un processus peut aussi se terminer lors de la réception d'un signal :

- Ctrl-Alt gr-\: SIGQUIT
- Ctrl-C : SIGINT
- kill / pkill : SIGKILL
- beaucoup d'autres, dont certains définis par les normes POSIX temps réel

Terminaison d’un processus (4)

Signaux

Un processus peut aussi se terminer lors de la réception d’un signal :

- Ctrl-Alt gr-\: SIGQUIT
- Ctrl-C : SIGINT
- kill / pkill : SIGKILL
- beaucoup d’autres, dont certains définis par les normes POSIX temps réel

Pour fermer proprement une application, il est indispensable de gérer l’interception des signaux!

Terminaison d'un processus (5)

Signaux (signals.c)

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ ./signals
1 : Hangup
2 : Interrupt
3 : Quit
4 : Illegal instruction
5 : Trace/breakpoint trap
6 : Aborted
7 : Bus error
8 : Floating point exception
9 : Killed
10 : User defined signal 1
11 : Segmentation fault
12 : User defined signal 2
13 : Broken pipe
14 : Alarm clock
15 : Terminated
16 : Stack fault
17 : Child exited
18 : Continued
19 : Stopped (signal)
```

Terminaison d'un processus (6)

Signaux

Pour gérer les signaux, deux méthodes :

- **signal** : très simple, défini par Ansi C et SUSv4 mais peut avoir des problèmes de portabilité entre systèmes UNIX
- **sigaction** : plus complexe mais complètement portable!

Terminaison d'un processus (6)

Signaux

Pour gérer les signaux, deux méthodes :

- **signal** : très simple, défini par Ansi C et SUSv4 mais peut avoir des problèmes de portabilité entre systèmes UNIX
- **sigaction** : plus complexe mais complètement portable!

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ ./sigaction
sleep...
sleep...
sleep...
^CSignal 2 received
sleep...
sleep...
sleep...
^\Signal 3 received
sleep...
^\Signal 3 received
sleep...
^\Signal 3 received
sleep...
^C
```

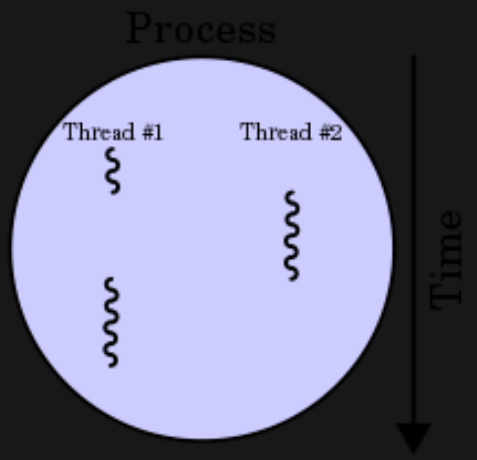
Terminaison d'un processus (7)

Signaux (sigaction.c)

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signals_handler(int signal_number)
{
    printf("Signal %d received\n", signal_number);
}

int main()
{
    struct sigaction action;

    action.sa_handler = signals_handler;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = 0;
    sigaction(SIGQUIT, & action, NULL);

    action.sa_flags = SA_RESETHAND;
    sigaction(SIGINT, & action, NULL);

    while(1)
    {
        printf("sleep...\n");
        sleep(2);
    }

    return EXIT_SUCCESS;
}
```

Les pthreads (1)

Présentation



Threads avec portabilité SUSv4 sous Linux : les pthreads!

Les pthreads (2)

NTPL

Les pthreads sont intégrés dans le Kernel Linux depuis la version 2.6 via la bibliothèque Native Posix Thread Library (NTPL).

Les pthreads (2)

NTPL

Les pthreads sont intégrés dans le Kernel Linux depuis la version 2.6 via la bibliothèque Native Posix Thread Library (NTPL).

Pour utiliser les threads de la NTPL :

- #include <pthread.h>
- gcc -pthread

Les pthreads (3)

Création et identification

Identifiant :

- d'un processus : pid_t
- d'un thread : pthread_t

Pour la création d'un pthread :

```
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attributes,
                   void * (* funct) (void * arg),
                   void * argument)
```

Les pthreads (3)

Création et identification

Identifiant :

- d'un processus : pid_t
- d'un thread : pthread_t

Pour la création d'un pthread :

```
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attributes,
                   void * (* funct) (void * arg),
                   void * argument)
```

```
tergeist 25832  0.0  0.0  10452   592 pts/0    -    00:31   0:00 ./thread
tergeist     -  0.0    -      -     - -        Sl+  00:31   0:00 -
tergeist     -  0.0    -      -     - -        Sl+  00:31   0:00 -
```

Les pthreads (4)

Application (thread.c)

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

void * function(void *arg)
{
    while(1)
    {
        printf("Secondary thread\n");
        sleep(2);
    }
}

int main()
{
    pthread_t thread;
    if (pthread_create(& thread, NULL, function, NULL) != 0)
    {
        exit(EXIT_FAILURE);
    }

    while(1)
    {
        printf("Main thread\n");
        sleep(2);
    }

    return EXIT_SUCCESS;
}
```

Les pthreads (5)

Terminaison

Cas où tout le processus (et donc tous les threads) est tué :

- le thread principal fait appel à **return** dans le main
- le thread principal fait appel à **exit**
- un des threads fait appel à **exit**
- un des threads se termine involontairement

Les pthreads (5)

Terminaison

Cas où tout le processus (et donc tous les threads) est tué :

- le thread principal fait appel à **return** dans le main
- le thread principal fait appel à **exit**
- un des threads fait appel à **exit**
- un des threads se termine involontairement

Cas où juste un threads est tué :

- le thread fait appel à la fonction **pthread_exit**

Les pthreads (6)

join

Un pthread stocke sa valeur de retour dans la pile.

Pour récupérer la valeur de retour, il faut utiliser **pthread_join**. Cependant, le thread appelant bloque jusqu'à la fin du thread!

Les pthreads (6)

join

Un pthread stocke sa valeur de retour dans la pile.

Pour récupérer la valeur de retour, il faut utiliser **pthread_join**. Cependant, le thread appelant bloque jusqu'à la fin du thread!

Un processus a un espace mémoire limité donc ???

Les pthreads (6)

join

Un pthread stocke sa valeur de retour dans la pile.

Pour récupérer la valeur de retour, il faut utiliser **pthread_join**. Cependant, le thread appelant bloque jusqu'à la fin du thread!

Un processus a un espace mémoire limité donc ???

=> comme un thread stocke sa valeur de retour dans la pile, un processus peut instancier seulement un nombre limité de thread!

Les pthreads (7)

Nombre maximum de threads (maxthread.c)

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

void * function(void * arg) { return NULL;  }

int main()
{
    pthread_t thread;
    int number_of_thread = 0;

    while(pthread_create(&thread, NULL, function, NULL) == 0)
    {
        number_of_thread ++;
        usleep(100000);
    }

    printf("Number of threads: %d\n", number_of_thread);

    return EXIT_SUCCESS;
}
```

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ ./maxthread
Number of threads: 382
```

Les pthreads (8)

detach (unlthread.c)

Pour indiquer au kernel de ne pas stocker les valeurs de retours dans la pile, on peut utiliser **pthread_detach**.

Les pthreads (8)

detach (unlthread.c)

Pour indiquer au kernel de ne pas stocker les valeurs de retours dans la pile, on peut utiliser **pthread_detach**.

=> un processus peut alors créer un nombre illimité de thread!

```
tergeist@multi:~/devel/packages/embsys/lectures/2_processus/code$ ./unlthread
Number of threads: 100
Number of threads: 200
Number of threads: 300
Number of threads: 400
Number of threads: 500
Number of threads: 600
Number of threads: 700
Number of threads: 800
Number of threads: 900
Number of threads: 1000
^C
```

Les pthreads (9)

Thread Safety et _REENTRANT

Un code Thread Safe peut travailler **correctement** en mode multitâches, c'est à dire être utilisé simultanément par plusieurs threads au sein d'un même espace mémoire.

Les pthreads (9)

Thread Safety et _REENTRANT

Un code Thread Safe peut travailler **correctement** en mode multitâches, c'est à dire être utilisé simultanément par plusieurs threads au sein d'un même espace mémoire.

L'option -D_REENTRANT indique au compilateur que les fonctions peuvent être utilisées simultanément (évite la duplication en RAM).

```
COLLECT_GCC_OPTIONS='-v' '-o' 'thread' '-pthread' '-mtune=generic' '-march=i586'
 /usr/lib/gcc/i586-linux-gnu/5/cc1 -quiet -v -imultiarch i386-linux-gnu -D_REENT
RANT thread.c -quiet -dumpbase thread.c -mtune=generic -march=i586 -auxbase thre
ad -version -o /tmp/ccstLfvc.s
```

Les pthreads (9)

Thread Safety et _REENTRANT

Un code Thread Safe peut travailler **correctement** en mode multitâches, c'est à dire être utilisé simultanément par plusieurs threads au sein d'un même espace mémoire.

L'option -D_REENTRANT indique au compilateur que les fonctions peuvent être utilisées simultanément (évite la duplication en RAM).

```
COLLECT_GCC_OPTIONS='-v' '-o' 'thread' '-pthread' '-mtune=generic' '-march=i586'
 /usr/lib/gcc/i586-linux-gnu/5/cc1 -quiet -v -imultiarch i386-linux-gnu -D_REENT
RANT thread.c -quiet -dumpbase thread.c -mtune=generic -march=i586 -auxbase thre
ad -version -o /tmp/ccstLfvc.s
```

Une fonction réentrante n'est pas forcément Thread Safe.
=> des mécanismes d'exclusion mutuelle sont nécessaires dû au caractère préemptif du Kernel!

Les pthreads (10)

Synchronisation

Utilisation de **mutex** (mutual exclusion) pour la synchronisation d'accès à des données partagées.

Les pthreads (10)

Synchronisation

Utilisation de **mutex** (mutual exclusion) pour la synchronisation d'accès à des données partagées.

Les mutex sont des verrous à deux états (libre ou verrouillé) et représentés par des variables de type **pthread_mutex_t**.

Les pthreads (11)

Mutex et RW Lock

Mutex avec type **pthread_mutex_t** :

- création statique : PTHREAD_MUTEX_INITIALIZER
- création dynamique : **pthread_mutex_init**
- libération de la variable : **pthread_mutex_destroy**
- verrouillage : **pthread_mutex_lock**
- déverrouillage : **pthread_mutex_unlock**

Les pthreads (11)

Mutex et RW Lock

Mutex avec type **pthread_mutex_t** :

- création statique : PTHREAD_MUTEX_INITIALIZER
- création dynamique : **pthread_mutex_init**
- libération de la variable : **pthread_mutex_destroy**
- verrouillage : **pthread_mutex_lock**
- déverrouillage : **pthread_mutex_unlock**

RW Lock avec type **pthread_rwlock_t** :

- création statique : PTHREAD_RWLOCK_INITIALIZER
- création dynamique : **pthread_rwlock_init**
- libération de la variable : **pthread_rwlock_destroy**
- demande Read Only : **pthread_rwlock_rdlock**
- demande RW : **pthread_rwlock_wrlock**
- déverrouillage : **pthread_rwlock_unlock**

Les pthreads (12)

Programmation avancée

Les applications concurrentes sont complexes à mettre en place et de très nombreuses fonctions existent pour gérer de manière précise le comportement d'un thread :

Les pthreads (12)

Programmation avancée

Les applications concurrentes sont complexes à mettre en place et de très nombreuses fonctions existent pour gérer de manière précise le comportement d'un thread :

- nettoyage : pthread_cleanup_push, pthread_cleanup_pop
- taille de la pile : pthread_attr_getstackaddr, pthread_attr_setstackaddr
- variable globale à portée d'un seul thread : pthread_key_create
- et encore beaucoup d'autres...

Conclusion

Il faut être un vieux sage pour maîtriser tous les aspects de la programmation multithread!

Références

- Linux Embarqué - Pierre Ficheux
- Développement système sous Linux - Christophe Blaess
- Modern Operating Systems - Andrew Tanenbaum