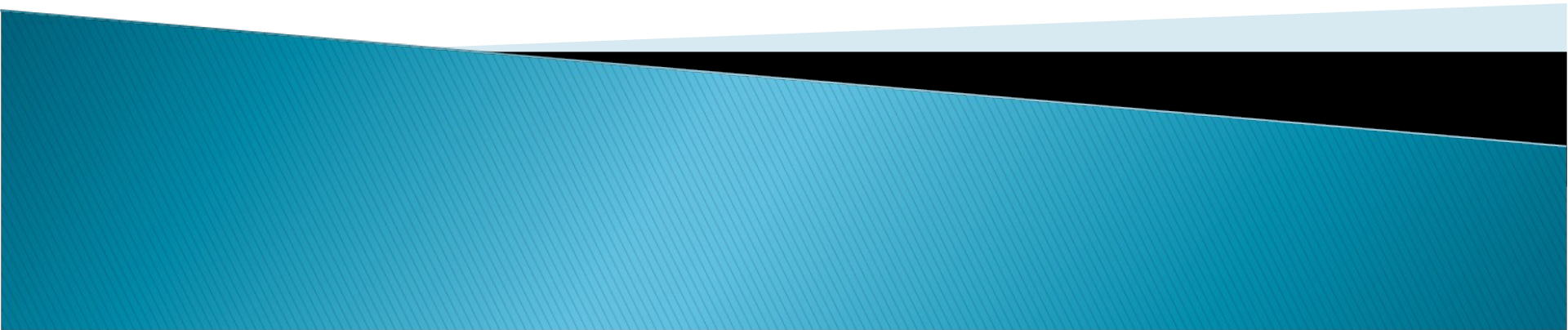
I-C1
Système d'exploitation et Programmation Système

Philippe.Arnould@univ-pau.fr



I-C1 : Plan

- Rappel de programmation
- Les processus
- Les threads
- La communication inter-processus
- Appels systèmes
- Sécurité
- Bibliographie

I-C1 : Rappel de programmation

- Programmation en C !
- Utilisation d'un éditeur avec coloration et « indentation »
  - Vim, gedit, emacs,...
- Relire le cours de S2!
- http://c.developpez.com

I-C1 : Rappel de programmation

- Voici ne qu'il ne faut pas faire :

```
int main ()
{printf ("Hello, world\n");
}
```

- Mais comme ceci (indentation)

```
int main ()
{
    printf ("Hello, world\n");
}
```

I-C1 : Rappel de programmation

- Comment compiler un fichier source ?
  - gcc –o mon_executable fichier_source.c
    –lbibliothèque_utilisées
- Comment compiler un projet avec plusieurs fichiers sources ?
  - Par exemple, pour écrire mon programme facto, j'ai utilisé deux fichiers :
    - main.c et calcul_factorielle.c

I-C1 : Rappel de programmation

Fichier main.c

```
#include<stdio.h>

long factorielle(int n) ;

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

	int n;

	printf("entrer un nombre entier:");
	scanf("%d",&n);
	fflush(stdin);

	printf("%d!=%ld\n",n,factorielle(n));
	return 0;

}
```

I-C1 : Rappel de programmation

- Fichier calcul_factorielle.c

```
/* Calcul de la factorielle par récurrence */


long factorielle(int n) {
    if (n==0)
       return 1;
    else
        return n*factorielle(n-1);
}
```

- Comment compiler ?
  - On va compiler séparément chaque source et quand tout sera ok on effectuera l'édition de lien.

I-C1 : Rappel de programmation

- Pour compiler sans effectuer l'édition de lien, i.e. produire seulement un fichier objet :
  - # gcc –c main.c
- gcc compile et génère le fichier objet facto.o
- Ensuite on réalise l'édition de lien avec l'option –o

```
# gcc –c calcul_factorielle.c
# gcc –o facto main.o calcul_factorielle.o
# ls
calcul_factorielle.c  calcul_factorielle.o  facto  main.c
  main.o
# ./facto
entrer un nombre entier:5
5!=120
```

I-C1 : Rappel de programmation

- Utilisation de l'outil make :
  - assure la compilation séparée grâce à gcc
  - utilise des macro-commandes et des variables
  - permet de ne recompiler que le code modifié
  - permet d'utiliser des commandes *shell*, et ainsi d'effectuer une installation
  - est essentiel lorsque l'on veut effectuer un portage, car la plupart des logiciels libres UNIX (c'est-à-dire des logiciels qui sont fournis avec le code source) l'utilisent pour leur installation
- Utililise un fichier Makefile :
  - Il indique à *make* comment exécuter les instructions nécessaires à l'installation d'un logiciel ou d'une librairie.
  - Il doit se trouver dans le répertoire courant lorsqu'on appelle *make* à l'invite du shell.
  - Les instructions contenues obéissent à une syntaxe spéciale.

I-C1 : Rappel de programmation

- Exemple de fichier Makefile :

```
OBJECTS = main.o calcul_factorielle.o facto

all: $(OBJECTS)

facto: main.o calcul_factorielle.o
	gcc -o facto main.o calcul_factorielle.o
main.o: main.c
	gcc -c main.c
calcul_factorielle.o: calcul_factorielle.c
	gcc -c calcul_factorielle.c
clean:
	rm -f *.o facto
```

I-C1 : Rappel de programmation

```
 [arnould@ifrcir121 IC1]$ ls
calcul_factorielle.c  calcul_factorielle.o  facto  main.c  main.o
   Makefile
[arnould@ifrcir121 IC1]$ make clean
rm -f *.o facto
[arnould@ifrcir121 IC1]$ ls
calcul_factorielle.c  main.c  Makefile
[arnould@ifrcir121 IC1]$ make all
gcc -c main.c
gcc -c calcul_factorielle.c
gcc -o facto main.o calcul_factorielle.o
[arnould@ifrcir121 IC1]$ touch calcul_factorielle.c
[arnould@ifrcir121 IC1]$ make all
gcc -c calcul_factorielle.c
gcc -o facto main.o calcul_factorielle.o
[arnould@ifrcir121 IC1]$
```

I-C1 : Rappel de programmation

- Débugger les programmes :
  - Utilisation du printf
  - Utilisation d'un débuggeur : gdb
    - Compiler le programme avec –g
    - Visualise les variables
    - Insère des points d'arrêts
    - …

I-C1 : Rappel de programmation

- Makefile

```
CFLAGS= -g

OBJECTS = main.o calcul_factorielle.o facto

all: $(OBJECTS)

facto: main.o calcul_factorielle.o
	gcc $(CFLAGS) -o facto main.o calcul_factorielle.o
main.o: main.c
	gcc $(CFLAGS) -c main.c
calcul_factorielle.o: calcul_factorielle.c
	gcc $(CFLAGS) -c calcul_factorielle.c
clean:
	rm -f *.o facto
```

I-C1 : Rappel de programmation

- main.c

```
....
printf("%d!=%ld\n",n,factorielle(&n));
        return 0;
...
```

```
[arnould@ifrcir121 gdb]$ make clean
rm -f *.o facto
[arnould@ifrcir121 gdb]$ make all
gcc -g -c main.c
main.c: In function `main':
main.c:14: attention : passage de l'argument n°1 de « factorielle » transforme un
    pointeur en entier sans transtypage
gcc -g -c calcul_factorielle.c
gcc -g -o facto main.o calcul_factorielle.o
[arnould@ifrcir121 gdb]$ ./facto
entrer un nombre entier:5
Erreur de segmentation
```

I-C1 : Rappel de programmation

- [arnould@ifrcir121 gdb]$ gdb facto
- GNU gdb Red Hat Linux (6.3.0.0-1.132.EL4rh)
- Copyright 2004 Free Software Foundation, Inc.
- GDB is free software, covered by the GNU General Public License, and you are
- welcome to change it and/or distribute copies of it under certain conditions.
- Type "show copying" to see the conditions.
- There is absolutely no warranty for GDB. Type "show warranty" for details.
- This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library "/lib/tls/libthread_db.so.1".

- (gdb) run
- Starting program: /home/r/cri/arnould/IC1/gdb/facto
- entrer un nombre entier:5

- Program received signal SIGSEGV, Segmentation fault.
- 0x08048490 in factorielle (n=-1075695494) at calcul_factorielle.c:8
- 8 return n*factorielle(n-1);
- (gdb) print n
- $1 = -1075695494
- (gdb)

I-C1 : Rappel de programmation

- Trouver les informations
  - Pages du « man »
    - Divisé en sections, les principales sont :
      - (1) User commands
      - (2) System calls
      - (3) Standard library functions
      - (8) System/administrative commands
    - Exemples
      - Man sleep
      - Man 3 sleep
      - Man –k sleep

I-C1 : Rappel de programmation

- Trouver les informations
  - Page de « info »
    - Utilitaire utilisant le l'hypertexte comme Emacs
    - faire Ctrl+h pour avoir de l'aide
    - Documentation de :
      - gcc -> le compilateur C
      - libc -> la bibliothèque GNU C qui inclut les appels systèmes
      - gdb -> le debuggeur du GNU
  - Fichiers d'entêtes
    - /usr/include
    - /usr/include/sys
    - Pour chercher un petit grep…

I–C1 : Les processus

- Un processus = un programme en exécution
- Chaque processus est identifié par un identificateur pid :
  - numéro codé sur 16 bits incrémenté par le SE
  - ps
  - mon_pid=getpid(); /* #include<unistd.h> */
- Chaque processus à un processus père
  - ps
  - le_pid_de_mon_pere=getppid();

I–C1 : Les processus

- process1.c

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

int main ()
{
        printf ("Le numéro du processus est %d\n",(int) getpid ());
        printf ("Le numéro processus père est  %d\n", (int) getppid ());
        return 0;
}
```

- Exécution :

```
[arnould@ifrcir121 processus]$ ps
  PID TTY          TIME CMD
27395 pts/135  00:00:00 bash
28692 pts/135  00:00:00 ps
[arnould@ifrcir121 processus]$ gcc -o process1 process1.c
[arnould@ifrcir121 processus]$ ./process1
Le numéro du processus est 28698
Le numéro processus père est  27395
[arnould@ifrcir121 processus]$
```

I-C1 : Les processus

- Utilisation de la commande ps
  - ps -e -o pid,ppid,command
  - -e affiche toutes les processus
  - -o pid,ppid,command spécifie l'affichage
- Supprimer un processus
  - Utilisation de la commande kill avec le signal SIGTERM (9)
  - kill -9 numero_pid_du_process_a_supprimer

I-C1 : Les processus : création

- la commande « system »
  - C'est une fonction (de la libc) qui permet d'exécuter une commande à l'intérieur d'un programme comme si elle était lancée dans un interpréteur de commandes.

```
#include <stdlib.h>
int main ()
{
        int variable_de_retour;
        variable_de_retour = system ("ls -l /");
        return variable_de_retour;
}
```

-> ne pas utiliser pour des raisons de sécurité liées à la création du shell lors de l'exécution de la commande.

I-C1 : Les processus : création

```
[arnould@ifrcir121 processus]$ gcc -o process2 process2.c
[arnould@ifrcir121 processus]$ ./process2
total 194
drwxr-xr-x     2 root root  4096 fév  6 04:18 bin
drwxr-xr-x     4 root root  1024 oct 26  2005 boot
drwxr-xr-x     9 root root  5420 fév  5 12:42 dev
drwxr-xr-x    81 root root  8192 avr 23 04:05 etc
drwxr-xr-x    15 root root  4096 jun 27  2006 home
...
dr-xr-xr-x  1419 root root     0 jan 10 17:09 proc
drwxr-x---    26 root root  4096 mar 26 15:03 root
drwxr-xr-x     2 root root  8192 fév  6 04:19 sbin
drwxr-xr-x     1 root root     0 jan 10 17:09 selinux
drwxr-xr-x     2 root root  4096 aoû 12  2004 srv
drwxr-xr-x     9 root root     0 jan 10 17:09 sys
drwxrwxrwt  1275 root root 49152 avr 27 16:16 tmp
drwxr-xr-x    15 root root  4096 déc 21  2005 usr
drwxr-xr-x    21 root root  4096 jun 13  2005 var
[arnould@ifrcir121 processus]$
```

I-C1 : Les processus : création « fork »

- La commande fork()
  - Appel système de la libc
  - La commande fork() duplique en mémoire le processus en cours d'exécution :
    - Le processus créé (i.e. dupliqué) est appelé processus fils.
    - Le processus « fils » exécute le même programme que le processus « père ».
    - Comment les distinguer ?
      - Par la valeur de retour de la commande fork():
        - 0 (zéro) dans le processus fils
        - Numéro du processus fils dans le processus père

I-C1 : Les processus : création « fork » et « exec »

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

int main ()
{
       pid_t pid_du_fils;

       printf ("Le pid du programme main est %d\n", (int)
     getpid ());

       pid_du_fils = fork ();
       if (pid_du_fils!= 0) {
              printf ("C\'est le processus père de pid %d\n",
     (int) getpid ());
              printf ("Mon fils a pour pid %d\n", (int)
     pid_du_fils);
       }
       else
              printf ("Je suis le fils,...mon pid est %d\n",(int)
     getpid ());
       return 0;
}
```

```
[arnould@ifrcir121 fork]$ ./fork
Le pid du programme main est 7324
Je suis le fils,...mon pid est 7325
C'est le processus père de pid 7324
Mon fils a pour pid 7325
[arnould@ifrcir121 fork]$
```

I-C1 : Les processus : création « exec »

- La commande « exec » remplace le programme en exécution par un nouveau programme.
- Trois catégories de commande exec
  - execp, exelp prend un nom de programme en paramètre et le cherche dans « path »
  - execv, execvp, execve acceptent des paramètres à passer au programme (execl paramètres à la sauce C argc)
  - Execve, execvl acceptent les paramètres d'environnement du type VARIABLE=valeur
- Comme exec remplace le programme, on n'a pas de variable de retour en cas d'erreur !

I-C1 : Les processus : création « fork » et « exec »

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

/* création le processus fils va exécuter un nouveau programme
   - programm est le nom du programme à exécuter (recherché dans la PATH)
   - arg_list est une liste de chaine de caractères qui passe les arguments au programme à exécuter
   - retourne le numéro de pid du processus créé. */
int creation (char* program, char** arg_list)  {
        pid_t child_pid;

        pid_t  child_pid = fork (); /* Duplique le processus courant */
        if (child_pid != 0)
          return child_pid; /* C'est le processus père */
        else {
          execvp (program, arg_list); /* On exécute programm en le cherchant dans le path */
          fprintf (stderr, "Erreur dans execvp\n"); /* execvp retourne une valeur si une erreur survient */
          abort ();
        }
}
```

I-C1 : Les processus : création « fork » et « exec »

```
int main ()
{
    /* Les arguments passés au programme crée ls -l / */
    char* arg_list[] = {
        "ls", /* argv[0], le nom du programme ls */
        "-l",
        "/",
        NULL /* les arguments se terminent par NULL */
    };

    /* creation un processus fils va exécuter la commande ls
    On ignore la valeur de retour*/

    creation ("ls", arg_list);
    printf ("fin du programme principal\n");
    return 0;
}
```

```
[arnould@ifrcir121 fork]$ gcc -o fork_exec fork_exec.c
[arnould@ifrcir121 fork]$ ./fork_exec
fin du programme principal
[arnould@ifrcir121 fork]$ total 194
drwxr-xr-x     2 root root  4096 fév  6 04:18 bin
drwxr-xr-x     4 root root  1024 oct 26  2005 boot
....
drwxr-xr-x    15 root root  4096 déc 21  2005 usr
drwxr-xr-x    21 root root  4096 jun 13  2005 var

[arnould@ifrcir121 fork]$
```

I-C1 : Les processus : ordonnancement

- Pas de priorité dans l'ordre d'exécution d'un fils par rapport au à son père
- Pour modifier la priorité, le SE fournit la commande « nice » avec comme argument un nombre -20 (priorité haute, root<0) à +19 (priorité basse) :

  #nice -10 sort input.txt > output.txt

I-C1 : Les processus : les signaux

- Les signaux permettent de de communiquer et de manipuler les processus.
- Un signal est un message asynchrone envoyé à un processus :
  - Dès qu'un signal est reçu, le processus le traite immédiatement avant la fin de son code en exécution.
- Chaque signal est associé à un numéro
- Sous Linux voir /usr/include/bits/signum.h.

I-C1 : Les processus : les signaux

- Comment les programmer ?
  - Quand un programme reçoit un signal :
    - Si la fonction signal_handler existe :
      - le programme en exécution est suspendu et signal_handler est exécutée et à la fin, le programme initial est relancé.
    - Sinon le programme ne fait sauf si c'est un signal particuler tel que SIGBUS (bus error), SIGSEGV (segmentation violation), and SIGFPE (floating point exception)
  - On peut envoyer des signaux entre deux programmes avec les commandes SIGTERM(demande de terminaison) et SIGKILL(pas de demande)
  - On peut utiliser nos propres signaux SIGUSR1 et SIGUSR2.

I-C1 : Les processus : les signaux

Comment les programmer ?

- int sigaction (int sig, const struct sigaction * act, struct sigaction * oact);
  - sig : numéro du signal
  - Sigaction : structure avec trois éléments :
    - sighandler_t sa_handler : peut prendre 3 valeurs `SIG_DFL‘ (défaut) , `SIG_IGN‘ (ignore), ou un pointeur vers une fonction
    - sigset_t sa_mask : définit l ’ensemble des signaux à interdire pendant l’exécution de la fonction associée (signal_handler)
    - int sa_flags : permet des fonctions particulière(info sigaction)
  - act pour la description du signal
  - oact avant la réception du signal
- Eviter d’effectuer des E/S dans le signal_handler
- Eviter d’utiliser des variables globales car avec l’arrivée d’un signal peut modifier le contenu (accès concurrent)
  - Solution utiliser le type sig_atomic_t qui garantit que l’affectation de la variable ne sera pas interrompue.

I-C1 : Les processus : les signaux

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

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
      ++sigusr1_count;
}
```

```
int main ()
{
      int i=100000;

      struct sigaction sa;
      memset (&sa, 0, sizeof (sa));
      sa.sa_handler = &handler;

      sigaction (SIGUSR1, &sa, NULL);
      do {
            printf("Taper kill -10 %d, i:%d\n",
(int)getpid(),i--);
      } while (i>0);

      printf ("SIGUSR1 a é envoyé %d fois\n",
sigusr1_count);
      return 0;
}
```

I-C1 : Les processus : wait()

- Permet d'attendre la mort de son fils en retournant une variable de retour.
- Les macros WIFEXIT… permettent d'extraire le code de retour. (#include<sys/wait.h>)
- Que se passe –t-il quand un processus fils s'arrête sans que le père récupère les informations (pas de wait()) ?
  - Création d'un processus zombie
  - Sous linux, le programme init supprime les processus zombie (ps -> <defunct>)

I–C1 : Les processus : wait()

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

main ()  {
  int pid;
  printf ("Bonjour, je me présente, je suis %d.\n",
              getpid ());
  printf ("Je sens quelque chose monter en moi...
     un       fils peut-être!\n");
  if (fork () == 0) {
      printf ("\tSalut, je suis %d, le gamin de %d.\n",
     getpid (),
             getppid ());
      sleep (3);
      printf ("\tJe suis si jeune, et déjà je sens que je
     m'affaiblis!\n");
      printf ("\tCa y est, je passe de vie a
     trépas!\n");
      exit (7);
} else {
      int ret1, status1;
      printf ("Attendons que ce morveux
disparaisse.\n");
      ret1 = wait (&status1);
      if ((status1 & 255) == 0) {
          printf ("Valeur de retour de wait(): %d\n",
ret1);
          printf ("Valeur de retour avec la macro
WEXITSTATUS =%d\n",
                  WEXITSTATUS (status1));
          printf ("Paramètre de exit(): %d\n",
                   (status1 >> 8));
          printf ("Un simple exit a eu raison du
petit.\n");
        }
    }
}
```

I-C1 : Les processus : wait()

```
[arnould@ifrcir121 fork]$ ./test_wait
Bonjour, je me présente, je suis 16676.
Je sens quelque chose monter en moi... un fils peut-etre!
        Salut, je suis 16677, le gamin de 16676.
Attendons que ce morveux disparaisse.
        Je suis si jeune, et déjà je sens que je m'affaiblis!
        Ca y est, je passe de vie a trépas!
Valeur de retour de wait(): 16677
Valeur de retour avec la macro WEXITSTATUS =7
Paramètre de exit(): 7
Un simple exit a eu raison du petit.
[arnould@ifrcir121 fork]$
```

IC1 : les threads vs fork()

- Multitâche sous Linux
- Utilisation du « fork »
- Avantages :
  - simplicité
- Inconvénients :
  - difficultés dans le partages des données car l'appel « fork » duplique le processus
  - la création de nouveau processus coûte en ressources processeur

IC1 : les threads vs fork()

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

int i;  /* variable globale */

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

  i = 1;

  if ((pid = fork()) == 0) {
    /* Dans le fils */
    printf ("Je suis le fils, pid = %d\n",
                getpid());
    sleep (2);
    printf ("Fin du fils, i = %d !\n", i);
    exit (0);
  }
else if (pid > 0) {
    /* Dans le pere */
    printf ("Je suis le pere, pid = %d\n", getpid());
    sleep (1);

    /* Modifie la variable */
    i = 2;
    printf ("le pere a modifie la variable a %d\n", i);

    sleep (3);
    printf ("Fin du pere, i = %d !\n", i);
    exit (0);
  }
  else {
    /* Erreur */
    perror ("fork");
    exit (1);
  }
}
```

IC1 : les threads vs fork()

```
[arnould@ifrcir121 fork]$ gcc -o fork1 fork1.c
[arnould@ifrcir121 fork]$ ./fork1
Je suis le fils, pid = 17790
Je suis le pere, pid = 17789
le pere a modifie la variable a 2
Fin du fils, i = 1 !
Fin du pere, i = 2 !
[arnould@ifrcir121 fork]$
```

IC1 : les threads

- Un thread <-> un processus fils classique à la différence qu'il partage beaucoup plus de données avec le processus qui l'a créé:
  - Les variables globales
  - Les variables statiques locales
  - Les descripteurs de fichiers (file descriptors)
- Tous les systèmes d’exploitations modernes les proposent
  - Solaris, Windows, Linux,…
- Sous Unix norme Posix 1003.1c : Pthread

IC1 : les threads

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

void *my_thread_process (void * arg)
{
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
}
```

IC1 : les threads

```
 main (int argc, char **argv)
{
  pthread_t th1, th2;
  void *ret;

  if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
    fprintf (stderr, "Erreur dans le pthread_create 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, my_thread_process, "2") < 0) {
    fprintf (stderr, " Erreur dans le pthread_create 2\n");
    exit (1);
  }

  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

IC1 : les threads

```
[arnould@ifrcir121 fork]$ gcc -o thread1 thread1.c -lpthread
[arnould@ifrcir121 fork]$ ./thread1
Thread 1: 0
Thread        2: 0
Thread 1: 1
Thread        2: 1
Thread 1: 2
Thread        2: 2
Thread 1: 3
Thread        2: 3
Thread 1: 4
Thread        2: 4
```

IC1 : les threads

NOM

pthread_create - créé un nouveau thread

SYNOPSIS

```
#include <pthread.h>

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

DESCRIPTION

pthread_create créé un nouveau thread s'éxécutant concurremment avec le thread appelant. Le nouveau thread exécute la fonction start_routine en lui passant arg comme premier argument. Le nouveau thread s'achève soit explicitement en appelant pthread_exit(3), ou implicitement lorsque la fonction start_routine s'achève. Ce dernier cas est équivalent à appelér pthread_exit(3) avec la valeur renvoyée par start_routine comme code de sortie.

L'argument attr indique les attributs du nouveau thread. Voir pthread_attr_init(3) pour une liste complète des attributs. L'argument attr peut être NULL, auquel cas, les attributs par défaut sont utilisés: le thread créé est joignable (non détaché) et utilise la politique d'ordonnancement usuelle (pas temps-réél).

...

IC1 : les threads

NOM

pthread_exit - termine le thread appelant

SYNOPSIS

#include <pthread.h>

void pthread_exit(void *retval);

DESCRIPTION

pthread_exit termine l'exécution du thread appelant.

L'argument retval est la valeur de retour du thread. Il peut être consulté par un autre thread en utisant pthread_join(3).

IC1 : les threads

NOM

pthread_join - attend la mort d'un autre thread

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);

DESCRIPTION

pthread_join suspend l'exécution du thread appelant jusqu'à ce que le thread identifié par th achève son exécution, soit en appelant pthread_exit(3) soit après avoir été annulé.

IC1 : les threads

Comment compiler le programme :

```
gcc -D_REENTRANT -o thread1 thread1.c
-lpthread
```

Permet l'emploi de fonctions particulières compatible avec le multitâche

Bibliothèque pthread

IC1 : les threads

Partages des données et synchronisation

- Mutex
- Sémaphores Posix

- Rappel
  - un mutex est une primitive de synchronisation utilisée en programmation informatique pour éviter que des ressources non partagées d'un système ne soient utilisées en même temps. Son implémentation varie selon les systèmes (masquage des interruptions, lecture/écriture en un cycle, etc.)
  - Un sémaphore est une variable protégée (ou un type de donnée abstrait) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) dans un environnement de programmation concurrente. Le sémaphore a été inventé par Edsger Dijkstra

IC1 : les threads

PTHREAD_MUTEX_LOCK(P)

NAME

pthread_mutex_lock,
pthread_mutex_trylock,
pthread_mutex_unlock
- lock and unlock a mutex

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

IC1 : les threads

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

void *functionC ();
pthread_mutex_t mutex1 =
    PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main ()
{
  int rc1, rc2;
  pthread_t thread1, thread2;

  if ((rc1 = pthread_create (&thread1, NULL,
     &functionC, NULL))) {
      printf (“Erreur de création de thread: %d\n",
     rc1);
    }
 if ((rc2 = pthread_create (&thread2, NULL,
                &functionC, NULL))) {
    printf (" Erreur de création de thread :
                %d\n", rc2);
    }

  pthread_join (thread1, NULL);
  pthread_join (thread2, NULL);

  exit (0);
}
void *
functionC ()
{
  pthread_mutex_lock (&mutex1);
  counter++;
  printf (“Valeur du compteur: %d\n", counter);
  pthread_mutex_unlock (&mutex1);
}
```

IC1 : les threads

```
[arnould@ifrcir121 fork]$ ./mutex1
Valeur du compteur: 1
Valeur du compteur: 2
[arnould@ifrcir121 fork]$
```

IC1 : Communication interprocessus

- Voir cours de première année
- Problématique des accès concurrents
  - Partage de zone mémoire,
  - Utilisation des sémaphores (IPC V)
  - Partage de données entre processus (pipe)
  - Partage à travers le réseau (socket)

IC1 : Communication interprocessus

- La mémoire partagée permet aux processus de communiquer simplement en lisant ou écrivant dans un emplacement mémoire prédéfini.
- La mémoire mappée est similaire à la mémoire partagée, excepté qu'elle est associée à unfichier.
- Les tubes permettent une communication séquentielle d'un processus à l'autre.
- Les FIFO sont similaires aux tubes excepté que des processus sans lien peuvent communiquer car le tube reçoit un nom dans le système de fichiers.
- Les sockets permettent la communication entre des processus sans lien, pouvant se trouver sur des machines distinctes.

IC1 : Communication interprocessus : mémoire partagée

- La mémoire partagée permet à deux processus ou plus d'accéder à la même zone mémoire comme s'ils avaient appelé malloc et avaient obtenu des pointeurs vers le même espace mémoire.
- Lorsqu'un processus modifie la mémoire, tous les autres processus voient la modification.
- L'accès à cette mémoire partagée est aussi rapide que l’accès à la mémoire non partagée du processus et ne nécessite pas d'appel système ni d'entrée dans le noyau. Elle évite également les copies de données inutiles.
- Comme le noyau ne coordonne pas les accès à la mémoire partagée, vous devez mettre en place votre propre synchronisation : Utilisation avec les sémaphores

IC1 : Communication interprocessus : mémoire partagée

- Pour utiliser un segment de mémoire partagée, un processus doit allouer le segment.
- Puis, chaque processus désirant accéder au segment doit l'attacher. Après avoir fini d'utiliser le segment, chaque processus le détache. À un moment ou à un autre, un processus doit libérer le segment.
- Débogage avec la commande ipcs.

IC1 : mémoire partagée Allocation

#include <sys/ipc.h>

#include <sys/shm.h>

int **shmget**(key_t **clé**, int **size**, int **shmflg**);

DESCRIPTION

shmget() renvoie l'identificateur du segment de mémoire partagée associé à la valeur de l'argument **clé**. Un nouveau segment mémoire, de taille **size** arrondie au multiple supérieur de PAGE_SIZE, est créé si **clé** a la valeur IPC_PRIVATE ou si aucun segment de mémoire partagée n'est associé à **clé**, et IPC_CREAT est présent dans **shmflg**.

shmflg est composé de :

- IPC_CREAT pour créer un nouveau segment. Sinon shmget() recherchera lesegment associé à clé, vérifiera que l'appelant a la permission de recevoir l'identifiant shmid associé au segment, et contrôlera que le segment n'est pas détruit.
- IPC_EXCL est utilisé avec IPC_CREAT pour garantir l'échec si le segment existe déjà.
- mode d'accès (les 9 bits de poids faibles) indiquant les permissions pour le propriétaire, le groupe et les autres. Actuellement la permission d'exécution n'est pas utilisée par le système.

IC1 : mémoire partagée Allocation

Exemple :

- L'appel suivant à shmget crée un nouveau segment de mémoire partagée (ou accède à un segment existant, si shm_key est déjà utilisé) qui peut être lu et écrit par son propriétaire mais pas par les autres utilisateurs.
  - int segment_id = shmget (shm_key , getpagesize () , IPC_CREAT | S_IRUSR | S_IWUSR);
- Si l'appel se passe bien, shmget renvoie un identifiant de segment. Si le segment de mémoire partagée existe déjà, les permissions d'accès sont vérifiées et le système s'assure que le segment n'est pas destiné à être détruit.

IC1 : mémoire partagée
Attachement et détachement

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

void *shmat ( int shmid, const void * shmaddr, int
   shmflg);
int shmdt (const void * shmaddr);
```

- La fonction **shmat** attache le segment de mémoire partagée identifié par **shmid** au segment de données du processus appelant. L'adresse d'attachement est indiquée par *shmaddr.*
- La fonction **shmdt** détache le segment de mémoire partagée situe à l'adresse indiquée par *shmaddr*. Le segment doit être effectivement attaché, et l'adresse *shmaddr* doit être celle renvoyée précédement par **shmat.**

IC1 : Contrôler et libérer la mémoire partagée

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

shmctl() permet à l'utilisateur d'obtenir des informations concernant un segment de mémoire partagée, ainsi que de fixer le propriétaire le groupe et les permissions d'accès à ce segment. Cette fonction permet également de détruire un segment. Les informations concernant le segment identifié par *shmid* sont renvoyées dans une structure *shmid_ds* :

```
struct shmid_ds {
  struct          ipc_perm shm_perm; /* Permissions d'accès       */
  int             shm_segsz;         /* Taille segment en octets  */
  time_t          shm_atime;         /* Heure dernier attachement */
  time_t          shm_dtime;         /* Heure dernier détachement */
  time_t          shm_ctime;         /* Heure dernier changement  */
  unsigned short  shm_cpid;          /* PID du créateur           */
  unsigned short  shm_lpid;          /* PID du dernier opérateur  */
  short           shm_nattch;        /* Nombre d'attachements     */
  ...
};
```

IC1 : mémoire partagée

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int
main ()
{
  int segment_id;
  char *shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

     /* Alloue le segment de mémoire partagée . */
  segment_id = shmget (IPC_PRIVATE,
     shared_segment_size, IPC_CREAT | IPC_EXCL    |
     S_IRUSR | S_IWUSR);

     /* Attache le segment de mémoire partagée . */
  shared_memory = (char *) shmat (segment_id, 0, 0);
  printf (" mémoire partagée attachée à l'adresse
     %p\n", shared_memory);

     /* Détermine la taille du segment . */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
printf (" taille du segment : %d\n", segment_size);

     /* Écrit une chaîne dans le segment de mémoire
     partagée . */
 sprintf (shared_memory, "Hello , world .");

     /* Détache le segment de mémoire partagée . */
 shmdt (shared_memory);

     /* Réattache le segment de mémoire partagée à
     une adresse différente . */
 shared_memory = (char *) shmat (segment_id, (void
     *) 0x5000000, 0);
 printf (" mémoire partagée réattachée à l'adresse
     %p\n",
         shared_memory);

     /* Affiche la chaîne de la mémoire partagée . */
 printf ("%s\n", shared_memory);
     /* Détache le segment de mémoire partagée . */
 shmdt (shared_memory);

     /* Libère le segment de mémoire partagée . */
 shmctl (segment_id, IPC_RMID, 0);

 return 0;
```

IC 1 : Sémaphores
Création

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

int semget (key_t key, int nsems, int semflg);
```

- Cette fonction retourne l'identificateur de l'ensemble de sémaphores associé à la valeur de clé *key*. Un nouvel ensemble contenant *nsems* sémaphores est créé si *key* a la valeur **IPC_PRIVATE** ou si aucun ensemble n'est associé à *key*, et si l'option **IPC_CREAT** est présente dans *semflg.* (i.e. *semflg* & **IPC_CREAT** différent de zéro).

IC 1 : Sémaphores Libération

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

int semctl (int semid, int semno, int cmd, ...);
```

- Cette fonction effectue l'opération de contrôle indiquée par *cmd* sur l'ensemble de sémaphores (ou sur le *semno-ième* sémaphore de l'ensemble) identifié par *semid*, (les sémaphores sont numérotés à partir de zéro).
  Cette fonction prend trois ou quatre arguments. Lorsqu'il y en a quatre, l'appel est **semctl***(***semid**,**semno**,**cmd**,**arg***);* où l'argument *arg* est de type **union semun.**
- Les valeurs autorisées pour l'opération *cmd* sont :
  - **IPC_STAT** copier dans la structure pointée par *arg***.buf**
  - **IPC_SET** Fixer la valeur de certains champs de la structure **semid_ds** pointée par *arg***.buf** dans la structure de contrôle de l'ensemble de sémaphores
  - **IPC_RMID** Supprimer immédiatement l'ensemble de sémaphores en réveillant tous les processus en attente.
  - **GETALL** Renvoyer la valeur **semval** de chaque sémaphore de l'ensemble dans le tableau *arg***.array**.
  - ...

IC 1 : Sémaphores exemple 1

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

 /* Nous devons définir l'union
semun nous - mêmes . */

union semun
{
       int val;
       struct semid_ds *buf;
       unsigned short int *array;
       struct seminfo *__buf;
     };

 /* Obtient l'identifiant d'un sémaphore
binaire , l'alloue si nécessaire . */

Int binary_semaphore_allocation (key_t key, int sem_flags) {
       return semget (key, 1, sem_flags);
}
```

```
 /* Libère un sémaphore binaire . Tous les utilisateurs
doivent avoir fini de s'en servir . Renvoie -1 en cas
d'échec . */

int
binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1, IPC_RMID,
ignored_argument);
}
```

IC 1 : Initialisation des sémaphores

- L'instanciation et l'initialisation des sémaphores sont deux opérations distinctes. Pour initialiser un sémaphore,
  - semctl (semid , 0, SETALL , argument);
- Avec argument de type union semun et faire pointer son champ array vers un tableau de valeurs unsigned short.

```
int binary_semaphore_initialize (int semid )
 {
   union semun argument ;
   unsigned short values [1];
   values [0] = 1;
   argument . array = values ;
   return semctl (semid , 0, SETALL , argument );
}
```

IC 1 : Sémaphores down et up

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

int semop (int semid, struct sembuf *sops, unsigned nsops);

Les champs de la struct sembuf sont les suivants :

- sem_num est le numéro du sémaphore dans l'ensemble sur lequel est eectuée l'opération.
- sem_op est un entier spécifant l'opération à accomplir.
  - Si sem_op est un entier positif, ce chire est ajouté à la valeur du sémaphore immédiatement.
  - Si sem_op est un nombre négatif, la valeur absolue de ce chire est soustraite de la valeur du sémaphore. Si cela devait rendre la valeur du sémaphore négative, l'appel est bloquant jusqu'à ce que la valeur du sémaphore atteigne la valeur absolue de sem_op (par le biais d'incrémentations eectuées par d'autres processus).
  - Si sem_op est à zéro, l'opération est bloquante jusqu'à ce que la valeur atteigne zéro.
- sem_flg est un indicateur. Positionnez-le à IPC_NOWAIT pour éviter que l'opération ne soit bloquante ; au lieu de cela, l'appel à semop échoue si elle devait l'être. Si vous le positionnez à SEM_UNDO, Linux annule automatiquement l'opération sur le sémaphore lorsque le processus se termine.

IC 1 : Sémaphores down (wait)

- /* Se met en attente sur un sémaphore binaire . Bloque jusqu'à ce que la valeur du sémaphore soit positive , puis le décrémente d'une unité . */
- int binary_semaphore_wait (int semid)
- {
  - struct sembuf operations [1];
  - /* Utilise le premier (et unique) sémaphore . */
  - operations [0]. sem_num = 0;
  - /* Décrémente d'une unité . */
  - operations [0]. sem_op = -1;
  - /* Autorise l'annulation . */
  - operations [0]. sem_flg = SEM_UNDO ;

  - return semop (semid , operations , 1);
- }

IC 1 : Sémaphores
up (post)

```
/* Envoie un signal de réveil à un sémaphore
  binaire : incrémente sa valeur d'une unité . Sort de
  la fonction immédiatement . */
int binary_semaphore_post (int semid )
{
   struct sembuf operations [1];
   /* Utilise le premier (et unique ) sémaphore . */
   operations [0]. sem_num = 0;
   /* Incrémente d'une unité . */
   operations [0]. sem_op = 1;
   /* Autorise l'annulation . */
   operations [0]. sem_flg = SEM_UNDO ;

   return semop (semid , operations , 1);
}
```