M3101 : Programmation Système

Semestre 3, année 2015-2016

Département Informatique
IUT de Bordeaux

septembre 2015

version du 7 septembre 2015

Objectif du cours : MC Info3-Système

A l'issue des cours d'informatique jusqu'à présent vous savez :

- Programmer en langage C.
- Utiliser un système d'exploitation UNIX.

En **6** Semaines (Cours + TD de 2h) :

Comment faire communiquer 2 programmes d'une même machine ?

MC Info3-Système : Programmation Système

MC Info3-Système : Programmation Système

Remarque importante : la plupart des programmes exemples à exécuter (indispensable) et imprimer sont dans
`Bibliotheque/ASR3-Systemes`

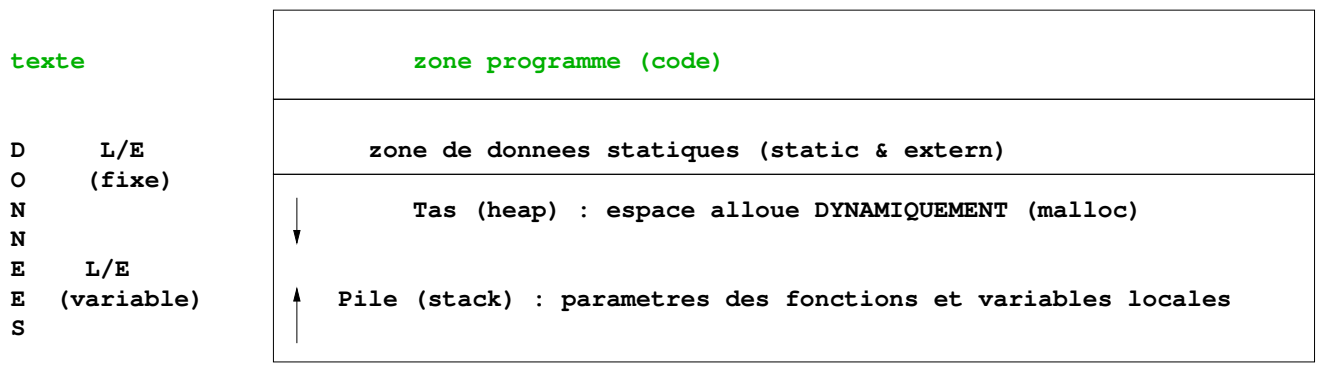
Par exemple :

`Bibliotheque/ASR3-Systemes/INTERRUPTION/int1.c`

Plan

- Un processus est un programme en cours d'exécution.
- Une zone mémoire est allouée à chaque processus.

Espace memoire d'un processus Unix



\+ le contexte :

. la valeur des registres
. la table des descripteurs de fichiers
. le repertoire courant
. le proprietaire,
. etc.

Notions sur les processus

- Lors de sa création, tout processus reçoit un numéro unique (entier positif de 0 à 32767) qui est son identificateur (**pid**).
- Tout processus est créé par un autre processus, excepté le **processus initial**, de nom **swapper** et de **pid 0**, créé artificiellement au chargement du système :

```
swapper (0)
    |
init (1)
```

Notions sur les processus (suite)

- Le swapper crée alors un processus appelé **init**, de **pid 1**, qui initialise le temps-partagé.
- Par convention, on considère que l'ensemble des processus existants à un instant donné forme un arbre dont la racine est le processus initial *init*.
- l'arbre des processus est obtenue par la commande : **pstree**.
- Tout processus a accès (par l'intermédiaire de fonctions système) à :
  - son pid (**getpid()**)
  - le pid de son père (**getppid()**)

Table des processus

- Elle est gérée par le noyau.
- Chaque entrée de la table contient des informations à propos d'un processus en cours d'exécution (structure sys/proc.h).
- Une entrée est allouée à la création du processus, désallouée à son achèvement.
- Listable par la commande ps : **ps -el**, **ps axu**.

La fonction system() : exemple en langage C

```
/* pid.c */
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main() {
printf("Je suis le processus N° : %d \n", getpid());

printf("Le pid de mon père est : %d \n", getppid());

system("ps -l");
exit(0);
}
```

La fonction system() : permet de lancer l'exécution d'un shell

```
#include <stdlib.h>

int system(const char* command);

Code de retour : 0 si OK, -1 sinon
```

La fonction system() : nouvel exemple en langage C

```
/* dater.c */
void main() {
    printf("La date est :"); fflush(stdout);
    system("/bin/date");
    /* suite du traitement */
}
```

La fonction system() : interprétation de l'exemple

- Lors de l'exécution de **dater**, un processus **shell** est créé à l'invocation de la fonction **system()** ;
- ce shell interprète la chaîne passée en argument :

```
dater
  |
sh (bash)
  |
/bin/date
```

Plan

Synchronisation des processus sous UNIX : les signaux

- Le traitement réalisé par un processus peut être interrompu par divers mécanismes d'interruptions.
- La réception d'un signal provoque une interruption logicielle : l'exécution d'un programme est interrompue pour traiter le signal reçu, puis reprend à l'endroit de son interruption.
- Les signaux sont en nombre fini (32 avec Linux).
- L'information véhiculée par un signal se borne à l'identité (le numéro) du signal.
- cf. `man -k signal` ou `man 7 signal`
- La liste des signaux disponibles sur le système peut être obtenue par la commande UNIX : **kill -l**

Synchronisation des processus sous UNIX : 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
```

Synchronisation des processus sous UNIX : le principe

Exemples : signaux visant à "terminer" un processus :

SIGHUP (1)	Lors de la déconnexion (fin du shell), ce signal est envoyé à tous les processus du même terminal.
SIGINT (2)	Interruption : généré au clavier par la touche Ctrl-C.
SIGQUIT (3)	Généré au clavier par Ctrl-\. Par rapport au précédent, son objectif est l'obtention d'un fichier **core**.
SIGTERM (15)	Terminaison : peut être généré par la commande **kill** (kill pid).
SIGKILL (9)	Peut également être généré par la commande **kill** (kill -9 pid). Par rapport au précédent, ce signal ne peut être intercepté par le processus.

Synchronisation des processus sous UNIX : le principe

Exemples : signaux visant à "stopper/reprendre" un processus :

SIGSTOP (19)	Stopper processus.
SIGTSTP (20)	Stopper le processus. Généré au clavier par Ctrl-Z. (reprise par les commandes **fg** ou **bg**)
SIGCONT (18)	Reprise du processus.

Synchronisation des processus sous UNIX : le principe

Pour émettre un signal :

- l'utilisateur peut "agir" sur le processus actif attaché au terminal : émission des signaux Ctrl-C (SIGINT), Ctrl-Z (SIGTSTP), Ctrl-\(SIGQUIT) au clavier.
- via la commande **kill**
- via des appels système dans les programmes (expliqués ci-après, par exemple **kill()**)

Synchronisation des processus sous UNIX : le principe

- Lorsqu'un processus est chargé en mémoire, le système initialise sa **table de traitement des signaux** : à chaque signal correspond un élément de la Table de Traitement des Signaux **TTS**.
- Par la suite, lorsque le processus recevra un signal, le traitement qu'il réalisait sera interrompu, et il exécutera la fonction associée au signal reçu.

La fonction kill() envoie un signal à un autre processus

La fonction système **kill()** permet à un processus d'envoyer un signal à un autre processus (voire plusieurs) :

```
#include<unistd.h>
int kill (pid_t pid, int signum);
```

La fonction signal() permet de changer la fonction de traitement d'un signal

Les processus ont tous un traitement prédéfini par rapport aux signaux. Néanmoins, celui-ci peut être redéfini par le programmeur pour la plupart des signaux.

- La fonction système **signal() (ou sigaction())** permet à un processus de changer la fonction de traitement d'un signal :

```
#include<stdio.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

La fonction système signal()

- Ainsi, dans la table de traitement des signaux **TTS**, la fonction associée au signal *signum* est remplacée par la fonction *handler()*.
- Deux fonctions ont un rôle particulier :
  - **SIG_IGN** : permet d'ignorer un signal,
  - **SIG_DFL** : permet de repositionner la fonction de traitement d'un signal à la fonction par défaut.

La fonction système signal()

Les appels les plus simples à la fonction **signal()** sont de la forme suivante :

- **signal(signum, fct_user) ;**
- **signal(signum, SIG_IGN) ;**
- **signal(signum, SIG_DFL) ;**

La fonction système signal()

- La fonction initiale de traitement de certains signaux (fonction par défaut) ne peut être modifiée ou ignorée : c'est notamment le cas des signaux **SIGSTOP et SIGKILL**
- Variantes suivant les signaux et les systèmes UNIX : lorsqu'un processus reçoit un signal, le système peut repositionner la fonction de traitement du signal à la fonction par défaut ...

La fonction système signal() : exemple 1 interruption

```
/* Bibliotheque/ASR3-Systemes/INTERRUPTION/int1.c */
#include <unistd.h>
#include <signal.h>
void interruption (int), arret (int);
char cmpt = '1';
main ()
{
      signal(SIGINT, interruption); /* récuperation de Ctrl-C */
      signal(SIGQUIT, arret);       /* récuperation de Ctrl-\ */
      signal(SIGTSTP, SIG_IGN);     /* on ignore Ctrl-Z */
      for (;;) {
         write(1,&cmpt,1);
         sleep(1);
      }
}
void arret (int k) {
     write (1,"\n",1);
     write (1,"Au revoir\n",10);
     signal(SIGQUIT, SIG_DFL);
     exit(0);
}
void interruption (int k) {
     signal(SIGINT, interruption);
     cmpt++;
}
```

Le signal d'alarme SIGALRM / la primitive alarm()

```
/* Bibliotheque/ASR3-Systemes/INTERRUPTION/int2.c */
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#define DELAI 1
void initialise(), calcule(), sauve(), onalrm(int);
unsigned long i;
main () {
      initialise();
      signal(SIGALRM, onalrm);
      alarm(DELAI);
      calcule();
      fprintf(stdout,"calcul terminé\n");
      exit(0);
}
void onalrm (int k) {
      sauve();
      signal(SIGALRM,onalrm);
      alarm(DELAI);
}
void initialise () { i=0; }
void sauve() { fprintf(stderr,"sauvegarde de i : %lu\n",i); }
void calcule() { while (i += 2); }
```

Exercice sur les signaux

- On dispose du programme source écrit en langage C **Bibliotheque/ASR3-Systemes/INTERRUPTION/clock.c** dont le rôle est d'afficher la date et l'heure en gros caractères sur la console.
- Ce programme est normalement appelé sans arguments. On vous demande de le modifier de façon à étendre ses possibilités, et notamment qu'il puisse être lancé sous les formes suivantes :
- $ **clock** // provoque l'affichage habituel de l'heure
- $ **clock** <délai> // provoque l'affichage habituel de l'heure // avec réveil au bout de *délai* secondes

nécessite la librairie *curses* : utiliser l'option -lcurses

Plan

Création d'un processus sous UNIX

La **création d'un nouveau processus Unix passe par deux mécanismes** utilisés en général de façon complémentaire :

1. la **duplication** d'un processus existant provoqué par la fonction système **fork()** : mécanisme de *fourche*,
2. le **recouvrement** d'un processus par un nouveau code : fonction système **exec()**.

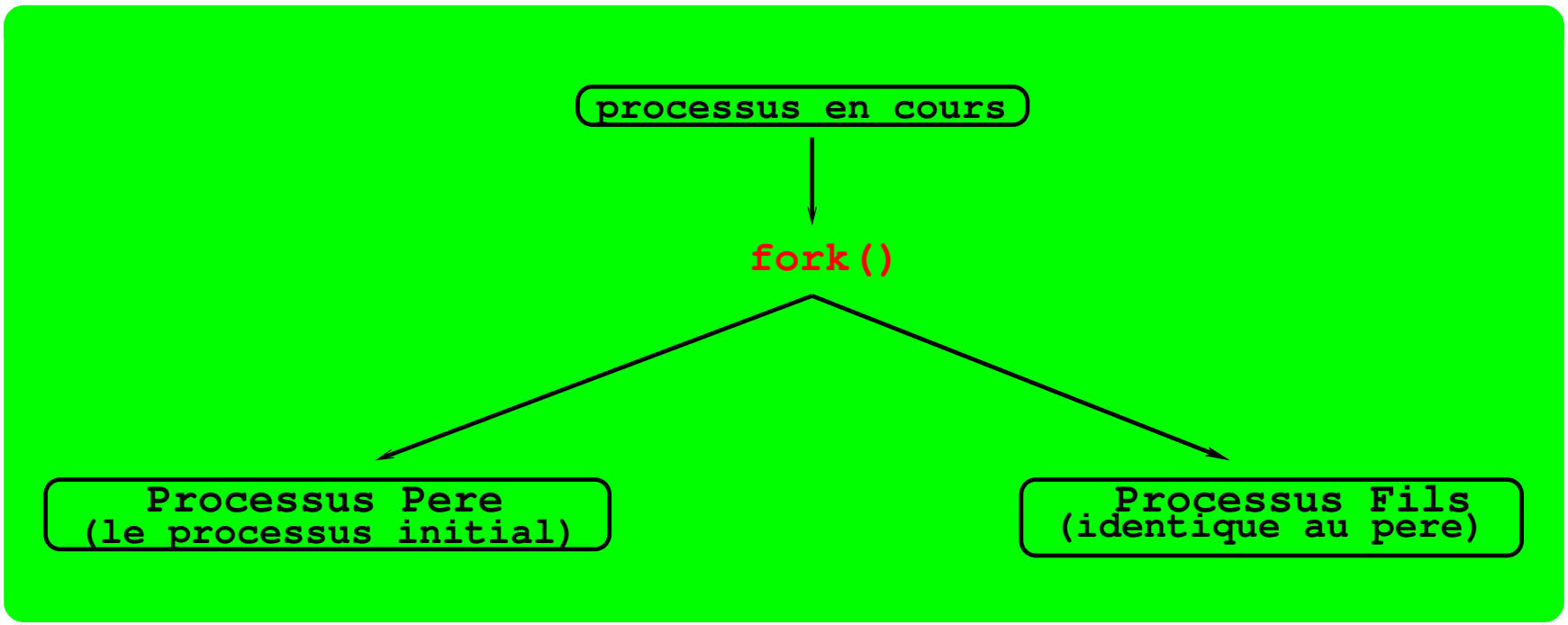
Création d'un processus sous UNIX

Ces mécanismes sont tels que **les processus** ainsi créés **pourront** :

- **se synchroniser** : envoi de signaux (appel système **kill()**), déroutement des fonctions de traitement des signaux (appel système **signal()**), mise en attente (appel système **wait()**), ...
- **communiquer** entre eux (appel système **pipe()**).

Duplication d'un processus

La fonction système **fork()** permet de dupliquer un processus en **créant dynamiquement un nouveau processus analogue au processus initial** :



Duplication d'un processus

Le processus créé (**processus fils**) hérite du **processus père** de certains de ses attributs :

- le même code,
- une copie de la zone de données,
- une copie de l'environnement,
- les différents propriétaires,
- une copie de la table des descripteurs de fichiers,
- une copie de la table de traitement des signaux,
- ...

Duplication d'un processus

Question : comment distinguer le processus père du fils ?
Réponse : leur **pid**
Plus précisément, le seul moyen (dans le code) de distinguer le *processus père* du *processus fils* est la **valeur de retour** de la fonction **fork()** qui est :

- **0 dans le processus fils**,
- **le pid du fils dans le processus père**.

Duplication d'un processus : exemple dupli.c

```
/* Bibliotheque/ASR3-Systemes/FORK/dupli.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
main(){
      int n;
      if((n=fork())==0) {/* processus fils ici */
        printf("pid processus fils %d \n", getpid());
        exit(EXIT_SUCCESS);
      }
      else {/* le processus père vient ici */
        sleep(5);
        printf("pid processus père %d \n", getpid());
        printf("mon fils porte le N° %d \n", n);
      }
exit(EXIT_SUCCESS);
}
```

```
% dupli # Exécution
pid processus fils 4647
pid processus père 4646
mon fils porte le N° 4647
%
```

Duplication d'un processus : remarques

- En cas de problème lors de la création du processus *fils* (impossibilité de création en général), la valeur retournée par *fork()* est -1. Cette éventualité n'est pas testée dans *dupli.c*
- Le **processus père** et le **processus fils** sont concurrents : ils **s'exécutent en parallèle**.
- Le **processus père** et le **processus fils** peuvent se **synchroniser** par l'envoi de signaux : en effet, le père connaît le *pid du fils* (valeur de retour de *fork()*) et le fils connaît le *pid du père* (fontion *getppid()*) : voir le source `pere_fils1.c`

Synchronisation de processus Unix : Père-Fils

```
/* Bibliotheque/ASR3-Systemes/FORK/pere_fils1.c */
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int nb_recu;

void hand (int sig){
  if(sig == SIGUSR1){ signal(SIGUSR1, hand); nb_recu++; printf("."); fflush(stdout); }
  else { printf("Nombre d'exemplaires reçus : %d\n", nb_recu); exit(EXIT_SUCCESS); }
}

void initialise () { nb_recu=0; }

main (){
  signal(SIGUSR1, hand);
  signal(SIGINT, hand);
  initialise();
  printf("Patientez 10 secondes svp ...\n");
  if(fork() == 0) {
    int i;
    for (i=0; i<10 ; i++){kill(getppid(), SIGUSR1); sleep(1); }
    printf("\nFin du fils \nVous pouvez taper Ctrl-C ...\n");
    exit(EXIT_SUCCESS);
  }
  while(1) pause();
}
```

La synchronisation de processus : l'appel système wait()

La primitive **wait()** provoque la **suspension** (mise en attente) du processus **jusqu'à** ce que l'un de ses **processus fils se termine**.

```
#include<unistd.h>
pid_t wait(int *status);
```

Cette primitive permet également à un processus d'**attendre un évènement** (voir **kill()**).

Synchronisation de processus Unix : attendre.c

```
/* Bibliotheque/ASR3-Systemes/FORK/attendre.c */

#include <stdlib.h>
#include <stdio.h>
main(){
   int m, p;
   if(fork()==0){/* processus fils */
     printf("pid processus fils : %d \n", getpid());
     sleep(3);
     exit(3);
   }
   else{/* processus père */
     m=wait(&p);
     printf("fin du processus fils %d avec valeur de retour %d \n", m, p);
   }
}
```

```
Exécution :
% attendre
pid processus fils : 153
fin du processus fils 153 avec valeur de retour 768
%
```

Synchronisation de processus Unix : explications sur attendre.c

- La fonction **wait() retourne le pid du fils** qui s'est terminé (et qui a donc provoqué le réveil du père) ; dans le cas où il n'y a pas de fils, **wait()** retourne **-1**.
- **Le paramètre passé par adresse (int *status)** permet d'obtenir des **informations** sur la façon dont **s'est terminé le processus fils**.
- Cette information de 16 bits doit être interprétée de la manière suivante :
  - si le processus se **termine normalement** par un **exit(k)**, alors la valeur est $k \times 256$ (d'où 768 dans l'exemple),
  - si le processus se **termine anormalement** (signal), les deux octets permettent d'obtenir le **numéro de ce signal** (cf. man wait)...

Synchronisation de processus Unix (synthèse) : course.c

```
        /* Bibliotheque/ASR3-Systemes/FORK/course.c */
#include <stdlib.h>
#include <stdio.h>
#define NB 5                    // nombre de concurrents
#define ARRIVEE 100000          // distance à parcourir
int main(){
   int i;             // compteur
   int r;             // retour fork
   pid_t pid;         // pid
   int status;        // valeur de retour
   for(i=1; i<=NB; i++) {
      if ((r = fork()) < 0) { // traitement erreur fork()
      fprintf(stderr,"Erreur fatale : fork()\n");
      exit(EXIT_FAILURE);
      }
      if (r == 0) { // un fils : il court ...
        int j;
        for (j=0; j<ARRIVEE; j++);
        exit(i);
      }
   } // dans le père : poursuite de la boucle for
   for (i=1; i<=NB; i++) {
    pid = wait(&status);
    fprintf (stdout, "le processus %d parti N° %d est arrivé N° %d\n", pid, status/256, i);
   }
   exit(EXIT_SUCCESS);
}
```

Synchronisation de processus Unix : résultats course.c

```
Exemples d'exécution ``course'' :
$ course
le processus 443 parti N° 4 est arrivé N° 1
le processus 442 parti N° 3 est arrivé N° 2
le processus 441 parti N° 2 est arrivé N° 3
le processus 440 parti N° 1 est arrivé N° 4
le processus 444 parti N° 5 est arrivé N° 5
$ course
le processus 447 parti N° 2 est arrivé N° 1
le processus 446 parti N° 1 est arrivé N° 2
le processus 450 parti N° 5 est arrivé N° 3
le processus 449 parti N° 4 est arrivé N° 4
le processus 448 parti N° 3 est arrivé N° 5
$
```

Synchronisation de processus Unix : Processus zombie

Lorsqu'un processus fils est terminé, il devient un **processus zombie** jusqu'à ce que :

- il soit rattrapé par un **wait** dans le processus père, ou
- le processus père meurt

Les processus zombie peuvent empêcher la création de nouveaux processus (32k max) : importance du **wait**.

Synchronisation de processus Unix : zombie.c

```
        /* Bibliotheque/ASR3-Systemes/FORK/zombie.c */
#include <stdlib.h>
#include <stdio.h>
int main ()
{
   if (fork()==0) {
      // le fils dort 10 secondes puis termine
      printf("Le fils (pid %d) dort... ", getpid()); fflush(stdout);
      sleep(10);
      printf("et termine...\n"); fflush(stdout);
      exit(0);
   } else {
      // le pere boucle...
      while (1) pause();
   }
}
```

Exercice sur la création de processus

Jeu du ShiFuMi

- **Objectif :** écrire un programme qui fait jouer un nombre quelconque de processus au ShiFuMi (≥ 2 joueurs).
- **Déroulement :** le processus père lance n processus fils qui tirent aléatoirement PIERRE, PAPIER et CISEAUX. Les fils se synchronisent avec le père qui s'occupe de récupérer les valeurs jouées, compte les points et les affiche.
- **Indications :** s'inspirer des exemples vus précédemment notamment sur la synchronisation via la primitive *wait()*.

Exercice sur la création de processus

Jeu du ShiFuMi : trace d'exécution

```
$ ./shifumi 3
Jeux des Processus

Processus 7235 joue PIERRE
Processus 7236 joue CISEAUX
Processus 7237 joue PAPIER

Points des Processus

Processus : 7235 Point : 1
Processus : 7236 Point : 1
Processus : 7237 Point : 1
```

Exercice sur la création de processus

Jeu du ShiFuMi : indications

1. Créer un programme **shifumi.c** qui prend en paramètre le nombre N de joueurs, et :
   - génère N fils
   - chaque fils affiche son PID
2. En plus de son PID, chaque fils
   génère un nombre entier aléatoire entre 0 et $max-1$, et l'affiche.

```
#include <time.h> // pour le random
srand(time(NULL)+getpid()); // initialiser le rand()
float max = 3;
int alea = (int)(max * rand() / RAND_MAX);
```

3. Faire en sorte que le père récupère les valeurs jouées par les fils (voir **course.c**).
4. Le père calcule les points et les affiche.

Recouvrement de processus : primitive **exec[l,v]()**

- La primitive système **exec[l,v]()** de recouvrement (ou **substitution**) permet de **lancer l'exécution d'un nouveau code**. Ce nouveau code recouvre l'ancien.
- Ainsi, il n'y a pas de création de nouveau processus. Il ne peut **pas** y avoir **de "retour d'exec réussi"**, et dans le cas où le recouvrement n'a pu se faire, la fonction *exec()* retourne -1.
- Le processus garde les mêmes caractéristiques (même contexte) :
  - même *pid*,
  - même *père*,
  - même priorité,
  - même propriétaire,
  - même répertoire de travail,
  - mêmes descripteurs de fichiers ouverts.

Le recouvrement par un nouveau code primitive **exec[l,v]()**

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

int execl(const char *filename, const char *arg0, ...);
```

Duplication et recouvrement (1/3) : exemple process.c

```
            /* Bibliotheque/ASR3-Systemes/EXEC/process.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
main(){
   int pid, a, i;
   fprintf(stdout,"début du processus de numéro %d \n",getpid());
   a = fork();  /* création d'un second processus */
   if (!a) {  /* cette partie de programme ne s'exécute que pour le processus fils créé */
      execl( "fils","fils",0);
      fprintf(stderr,"pb execl ");
      exit(3);
   }
   fprintf(stdout,"Je suis le père de %d\n", a);
   for (i=0;i < 10; i++){
      fprintf(stdout,"le père de numéro %d continue\n",getpid());
      sleep(1);
   }
   /* le reste */
   sleep(2);
   fprintf(stdout,"fin du père de numéro : %d \n",getpid());
   exit (0);
}
```

Duplication et recouvrement (2/3) : exemple fils.c et résultats

```
          /* Bibliotheque/ASR3-Systemes/EXEC/fils.c */

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

int main()
{
   int i;

   fprintf(stdout,"Début du fils de numéro %d \n",getpid());
   for (i=1;i<6;i++)
   {
      sleep(1);
      fprintf(stdout,"Le fils de numéro %d s'exécute\n",getpid());
   }
   fprintf(stdout,"Fin du processus de numéro : %d \n",getpid());
   exit(0);
}
```

Duplication et recouvrement (3/3) : résultats process/fils

```
Résultats ‘‘process/fils’’ :
% process
Début du processus de numéro 7326
Je suis père de 7327
Le père de numéro 7326 continue
Début du fils de numéro 7327
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Le père de numero 7326 continue
Le fils de numéro 7327 s'exécute
Fin du fils de numéro : 7327
Le père de numéro 7326 continue
Le père de numéro 7326 continue
Le père de numéro 7326 continue
Le père de numéro 7326 continue
Fin du père de numéro : 7326
%
```

Recouvrement et redirection : exemple substi.c et résultats

```
/* Bibliotheque/ASR3-Systemes/EXEC/substi.c */
main(){
   close(STDOUT_FILENO);
   open("toto", O_RDWR|O_CREAT|O_APPEND);
   execl("/bin/ls", "ls", "-l", 0);
   fprintf(stderr,"ERREUR EXEC\n");
}
```

Recouvrement et redirection : résultat

```
% substi
% cat toto
-rwxrwx--x+ 1 tmorsell info_perso 9724 sep 22 02:12 substi
-rw-r--r--+ 1 tmorsell info_perso 0 sep 22 02:12 toto
```

Recouvrement et redirection : exercice

ShiFuMi avec exec[l,v]

Reprendre l'exercice du ShiFuMi :

- placer le code d'un joueur dans un fichier **joueur.c**
- utiliser **exec[l,v]** pour appeler ce code
- le résultat attendu est identique à l'exercice d'origine

Plan

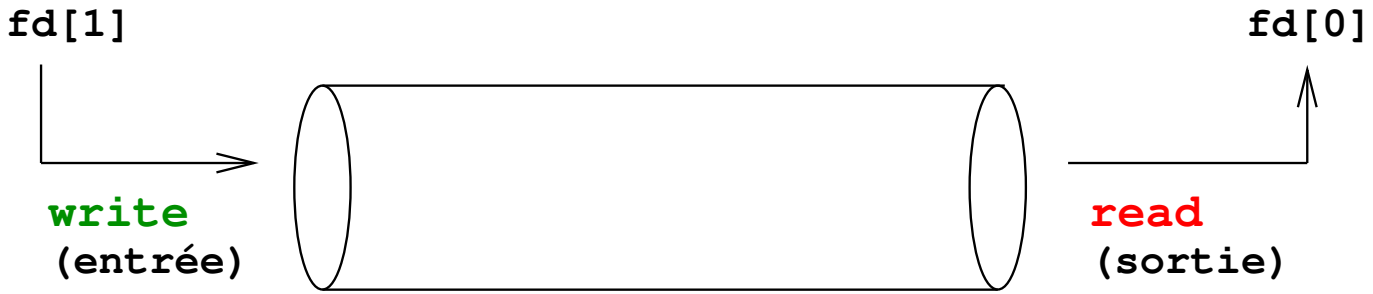
Les tubes (pipes)

La fonction système **pipe()** crée un **tube** de communication pour permettre à des **processus affiliés** de **communiquer** entre eux :

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

Les tubes (pipes)

- cet appel système crée un "tuyau" de communication et renvoie dans le tableau **fd** un **couple de descripteurs**.
- deux nouvelles entrées dans la **table de descripteurs** de fichiers ouverts sont initialisées (table des descripteurs du processus ayant réalisé cette ouverture de tube). ...
- la valeur de retour d'un appel réussi est 0 et -1 sinon.



- le descripteur **fd[1]** permettra à un premier (ou plusieurs) processus d'**écrire** à l'entrée du tube, et le descripteur **fd[0]** permettra à un autre processus (en général) de **lire** à la sortie du tube les informations écrites par le(s) premier(s).

Les tubes (pipes)

- Tous les processus "voulant" communiquer ainsi doivent "avoir accès" à ces descripteurs.
- Note : la table des descripteurs de fichiers ouverts d'un processus est **dupliquée** lors de la duplication d'un processus par *fork()* ou conservée lors de son recouvrement par *exec()*.
- Remarque : la valeur de retour de *pipe()* est 0 si le tube a été créé et -1 autrement.

Les tubes (pipes) : exemple simple

Exemple : un processus père écrivant à son fils dans un tube, et le fils lisant dans le tube ce que son père lui a écrit ...

```
main(){
   int fd[2];
   char p,c;

   pipe(fd);
   if(fork()) { /* processus père */
      ...
      write(fd[1], &c, 1);
      ...
      }
      else { /* processus fils */
      ...
      read(fd[0], &p, 1);
      ...
      }
}
```

Les tubes (pipes) : le protocole producteur/consommateur, un exemple type de communication par tube

- Dans cet exemple, un processus "père" ouvre un tube de communication pour permettre à ses deux fils, **fils1** et **fils2** créés par la suite, de communiquer.
- Le processus **fils1** lit au clavier des caractères et n'envoie au processus **fils2** que des caractères alphabétiques après les avoir capitalisés (filtre).
- Le processus **fils2** lit ces caractères dans le tube jusqu'à ce qu'il n'y en ait plus à lire : caractère *end of file/pipe*.
- Le processus père attend que ses fils aient terminé de communiquer.

Les tubes (pipes) : le protocole producteur/consommateur, un exemple type de communication par tube

Remarques :

- Une fin de fichier "caractère **end of file : EOF**" est envoyée dans un tube lorsque **tous** les processus ayant accès en écriture à ce tube (descripteur fd[1]) ont fermé ce descripteur (fichier).
- La fonction **read()** retourne la valeur 0 à la lecture du caractère "fin de fichier".
- La fonction **read()** est **bloquante** : si le tube est vide, le processus se bloque en attendant des données.
- La fonction **write()** est **bloquante** : si le tube est plein, le processus se bloque en attendant que le tube ait suffisamment d'espace disponible.

Les tubes (pipes) : exemple filtre.c (1/2)

```
      /* Bibliotheque/ASR3-Systemes/PIPE/filtre.c */

#include <stdio.h>
#include <unistd.h>
int p[2];

main(){
   int i,s;
   if (pipe(p) != 0) {
      fprintf(stderr,"pb ouverture pipe \n");
      exit(1);
   }
   if (fork()==0) {
      fils1();
   }
   if (fork()==0) {
      fils2();
   }
   close(p[0]);
   close(p[1]);
   fprintf(stderr,"debut attente \n");
   i=wait(&s);
   i=wait(&s);
   printf("fin du programme\n");
}
```

Les tubes (pipes) : exemple filtre.c (2/2)

```
        /* Bibliotheque/ASR3-Systemes/PIPE/filtre.c (suite) */
...
fils1(){
   char c;
   close(p[0]);
   fprintf(stderr,"debut fils1 ( taper 0 pour terminer ... )\n");
   while ((read(0,&c,1) > 0 ) && ( c != '0' ) )
      if ( (( c >= 'A') && (c <= 'Z')) || ((c >= 'a') && (c <= 'z'))) {
         if ((c >= 'a') && ( c <= 'z')) c-=32;
         write(p[1],&c,1);
         printf("fils1 %c\n",c);
      }
   close(p[1]);
   fprintf(stderr,"fin fils1 \n");
    exit(0);
}

fils2(){
   char c;
   close (p[1]);
   fprintf(stderr,"debut fils2 \n");
   while (read(p[0],&c,1) > 0)
      printf("fils2 :%c\n",c);
   close(p[0]);
   fprintf(stderr,"fin fils2\n");
    exit(0);
}
```

Les tubes (pipes) : exemple d'exécution de filtre

```
Exécution : ‘‘filtre’’
% filtre
début fils1 ( taper 0 pour terminer ... )
début fils2
début attente
é
9
#
iuT
fils1 I
fils1 U
fils1 T
fils2 :I
fils2 :U
fils2 :T
0
fin fils1
fin fils2
fin du programme
%
```

Exercice sur les tubes (pipes)

Application au jeu du ShiFuMi

- **Objectif :** écrire un programme qui fait jouer un nombre quelconque de processus au ShiFuMi (≥ 2 joueurs).
- **Déroulement :**
  - le processus père lance n processus fils qui tirent aléatoirement PIERRE, PAPIER et CISEAUX.
  - Les fils **communiquent** au père les valeurs jouées via des **pipes**.
  - Le père compte les points et les affiche.

Duplication de descripteurs : la fonction dup()

- La fonction système **dup()** permet de dupliquer un descripteur de fichier, **en utilisant le plus petit numéro de descripteur disponible** (première entrée libre dans la table de descripteurs de fichiers).
- L'exemple qui suit réalise l'analogue de la commande Unix **ls -l | wc -l** en utilisant les fonctions système *pipe()*, *fork()*, *dup()* et *execl()*.

Duplication de descripteurs : exemple tube.c

```
          /* Bibliotheque/ASR3-Systemes/PIPE/tube.c */
#include <stdio.h>
#include <unistd.h>
main(){
   int fd[2];
   if (pipe(fd) != 0) { fprintf(stderr, "Pb ouverture pipe\n"); _exit(1); }
   if (fork() == 0) { /* processus fils chargé d'exécuter "ls -l" */
      close(1);
      dup(fd[1]);
      close(fd[1]);
      close(fd[0]);
      execl("/bin/ls","ll","-l",NULL);
      /* sauf catastrophe, on ne passe pas ici... */
      fprintf(stderr, "Pb execl ll\n" );
       exit(2);
   }
   else{ /* processus père chargé d'exécuter "wc -l" */
      close(0);
      dup(fd[0]);
      close(fd[1]);
      close(fd[0]);
      execl("usr/bin/wc","wc","-l",NULL);
      /* sauf catastrophe, on ne passe pas ici... */
      fprintf(stderr, "Pb execl wc\n" );
       exit(3);
   }
}
```

Exercice sur la duplication de descripteurs

Pour les plus avancés

- **Objectif :** écrire un processus père qui lit une suite de caractères saisie par l'utilisateur. Une fois terminé, le père envoie cette suite à un fils qui s'occupe d'écrire le tout dans un fichier.
- **Déroulement :** le processus père lit l'entrée standard jusqu'au moment où l'utilisateur lui envoie un signal de fin de saisie. Une fois le signal lancé, le père envoie ces caractères au fils par l'intermédiaire d'un pipe qui s'occupe de copier le tout dans une fichier. Nous pouvons même imaginer que le fils précède chaque nouvelle entrée dans le fichier par la date du jour (à la manière d'un fichier de *log*).

Les mécanismes IPC

Il existe 3 ≪mécanismes IPC≫ de communication et de synchronisation :

- **mémoire partagée** : ce mécanisme permet à plusieurs processus tout à fait quelconques (pas nécessairement affiliés) de partager des segments en mémoire. Il s'agit d'un partage de mémoire qui n'induit aucune recopie de données ...
- **sémaphores** : ce mécanisme permet de résoudre le problème des accès concurrents à une même ressource telle que, par exemple, un segment de mémoire partagée entre plusieurs processus ...
- **files de messages** : implantation du concept de boîte à lettres qui permet l'échange de messages structurés entre processus ...

Ces trois types d'objets sont identifiés par des **clés**.

Les mécanismes IPC

Voici les points traités :

1. **constitution d'une clef**
2. **mémoire partagée**
3. **commandes shell de contrôle des mécanismes IPC**

Les **sémaphores IPC** et **files de message IPC** ne sont pas traités dans ce cours.

Les mécanismes IPC : (1) constitution d'une clef

- **utiliser les inclusions :**

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

- **créer une clé**

```
key=ftok(char *pathname, char project);
```

La fonction **ftok** (file to key) retourne (**key_t key**) l'identifiant d'une clé à partir d'un répertoire et d'un caractère ...

- **Exemples :**

```
cle=ftok("/", 'A');
```

Tous les processus utilisant cette clé pour créer un objet partagé se réfèreront ainsi aux mêmes objets ... Mais il n'en sera pas de même pour une clé créée ainsi :

```
cle=ftok(".", 'A');
```

Les mécanismes IPC : (2) mémoires partagées

- **Utiliser l'inclusion :**

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

- **Création d'une mémoire partagée**

```
shmid = shmget(key_t key, int size, int shmflg);
```

shmget retourne l'identificateur (**int shmid**) du segment de mémoire partagée ayant la clé **key**. Un nouveau segment de taille **size** octets est créé si :

  - key=IPC_PRIVATE
  - ou si les indicateurs de **shmflg** contiennent **IPC_CREAT** ; **IPC_CREAT|IPC_EXCL** indiquent que le segment ne doit pas exister au préalable.

Les bits de poids faible de **shmflg** indiquent les droits d'accès (**rwxrwxrwx**).

- **Exemple :**

```
mem_number=shmget(cle, sizeof(int), IPC_CREAT|0666);
```

Les mécanismes IPC : (2) mémoires partagées (suite)

- **Attacher un segment de mémoire partagée à un processus**
  Attacher un segment à un processus lui permet l'accès aux données contenues dans ce segment à l'aide d'un pointeur :

```
mem_addr = shmat( int shmid,
                  char *shmaddr,
                  int shmflg);
```

  retourne l'adresse (**char *mem_addr**) où le segment identifié par (**shmid**) a été placé en mémoire : placement automatique (et conseillé) si **shmaddr = NULL**.

  shmflg spécifie quels sont les droits d'accès du processus au segment : **SHM_R, SHM_W, ...**

- **Exemple :**

```
var_partagee=shmat(mem_number,0,0);
```

Les mécanismes IPC : (2) mémoires partagées (suite)

- **Détacher un segment de mémoire partagée d'un processus**

```
ret = shmdt( char *mem_addr);
```

détache le segment du processus et retourne ((**int ret**)-**1**) en cas d'erreur **(0 sinon)**.

- **Exemple :**

```
if(shmdt(var_partagee) == -1) {
    fprintf(stderr,"segment indetachable\n");
    exit(-1);
}
```

Les mécanismes IPC : (2) mémoires partagées (suite)

- **Contrôler un segment de mémoire partagée**

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

permet diverses opérations, dont la destruction du segment une fois tous les processus détachés (**IPC_RMID**) et son verrouillage (**SHM_LOCK**).

- **Exemples :**

```
shmctl(shmid,SHM_LOCK,NULL); // verrouille mémoire parta
shmctl(shmid,IPC_RMID,NULL); // détruit une mémoire part

if(shmctl(mem_number,IPC_RMID,NULL)==-1){
  fprintf(stderr,"segment indestructible\n");
  exit(-1);
}
```

Les mécanismes IPC : (2) mémoires partagées (suite et fin)

exemple producteur / consommateur :

```
/* Bibliotheque/ASR3-Systemes/IPC/prod.c */
...
commun = (struct donnees *)shmat (id,NULL,SHM_R | SHM_W);
...
while(1)
{
  printf("+ ");
  if (scanf("%d",&reponse)!=1) break;
  commun->nb++;
  commun->total += reponse;
  printf("sous-total %d= %d\n",commun->nb,commun->total);
};
```

```
/* Bibliotheque/ASR3-Systemes/IPC/cons.c */
...
commun = (struct donnees *)shmat (id,NULL,SHM_R);
...
while(encore)
{
  sleep(2);
  printf("sous-total %d= %d\n",commun->nb,commun->total);
};
```

Les mécanismes IPC : (3) commandes shell associées

- Liste des mécanismes IPC utilisés :
  - **ipcs** => *propriétaire, nature, id, clé, mode ...*
  - **ipcs -m** liste des *mémoires partagées*
  - **ipcs -q** liste des *files de messages*
  - **ipcs -s** liste des *sémaphores*
- Destruction :
  - **% ipcrm -m shmid** : détruit la mémoire partagée identifiée par **shmid**

Exemple :

```
ipcs -m
------ Shared Memory Segments --------
key        shmid   owner   perms   bytes   nattch   status
0x00000000 32769   root    777     139264     1
0x00000000 491524  lepine  600     393216     2        dest
```

Exercice sur la mémoire partagée

Reprendre l'exercice du ShiFuMi :

- un processus joue le rôle d'arbitre, plus un processus par joueur
- pour chaque joueur, une zone de mémoire partagée est utilisée pour renvoyer son choix

Plan

Les processus légers (threads) : plan

1. introduction
2. threads Posix

Les processus légers (threads Posix) : 1-introduction

Les processus classiques (lourds) d'UNIX possèdent des ressources séparées (espace mémoire, table des fichiers ouverts, ...). Lorsqu'un nouveau processus est créé par **fork()**, il se voit attribuer une copie des ressources du processus père. Il s'ensuit deux problèmes :

- problème de performances, car la duplication est un mécanisme coûteux,
- problème de communication entre les processus, qui ont des variables séparées.

Plusieurs moyens permettent d'atténuer ces problèmes :

- technique du *copy-on-write* pour ne dupliquer les pages mémoire que lorsque nécessaire,
- utilisation de segments de mémoire partagée (IPC) pour mettre en commun des données.

Les processus légers (threads Posix) : 1-introduction suite

- définition d'un mécanisme permettant d'avoir plusieurs fils d'exécution (**threads**) dans un même espace de ressources *non dupliquées* : notion de **processus légers**.
- Remarque : la communication entre deux threads est une opération économique
- les processus légers ayant vocation à communiquer entre eux, il existe des mécanismes de synchronisation : exclusion mutuelle (**mutex**), sémaphores, et conditions ...

Les processus légers (threads Posix) : 1-introduction suite et fin

Pour résumer :

processus : indépendants, difficile de partager des infos (IPC...)

threads : infos partagées, mais accès concurrent à gérer

Les processus légers : 2-threads Posix

- Utiliser l'inclusion : `# include <pthread.h>`
- **pthread_create** lance un nouveau processus léger, avec les attributs pointés par **attr** (**NULL** = attributs par défaut).

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

- Ce processus léger exécutera la fonction **start_routine**, en lui donnant le pointeur **arg** en paramètre. L'identifiant du processus léger est rangé à l'endroit pointé par **thread**.
- Valeur de retour de `pthread_create` : 0 si ok

Les processus légers : 2-threads Posix

- Ce processus léger se termine (avec code de retour) lorsque
  - la fonction associée se termine par **return retcode**
  - ou lorsque le processus léger exécute un **pthread_exit**

```
void pthread_exit(void *retval) ;
```

- La fonction **pthread_join** permet au processus père d'attendre la fin d'un processus léger, et de récupérer éventuellement son code de retour.

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

Les processus légers : 2-threads Posix

Exemple : hello.c

Remarque : le fonctionnement des processus légers peut être modifié (priorités, algorithme d'ordonnancement, etc.) en manipulant les attributs qui lui sont associés (cf. les fonctions **pthread_attr_xxx**).

Remarque : utiliser l'option `-pthread` de `gcc`.

Exercice : Shifumi threads

- reprendre l'exercice du Shifumi
- les joueurs sont maintenant implémentés par des processus légers (threads), et non plus des processus lourds
- à vous de voir comment l'arbitre récupère les valeurs jouées

Plan

Concurrence : plan

1. introduction
2. verrous d'exclusion mutuelle (mutex)
3. sémaphores
4. interblocage

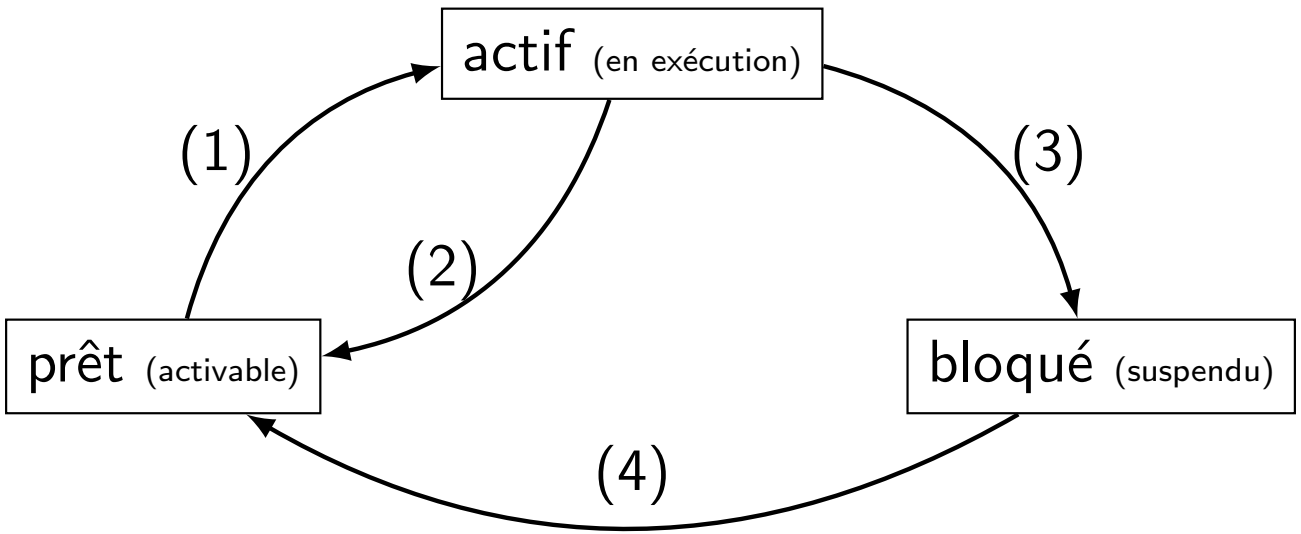
Ici ces concepts seront illustrés sur les processus légers (**threads**), mais ils peuvent également être utilisés entre processus lourds.

Non présentés dans ce cours :

- processus lourds : sémaphores via IPC
- processus lourds : files de messages via IPC
- processus légers : conditions

Pour les curieux, cf polycopié de M. Billaud.

Concurrence : graphe des états d'un processus



(1) l'exécution du processus est reprise sur un processeur

(2) processus interrompu (par exemple, l'ordonnanceur change de processus actif)

(3) le processus attend un évènement (fin d'E/S par exemple)

(4) l'évènement attendu est arrivé (fin d'E/S par exemple)

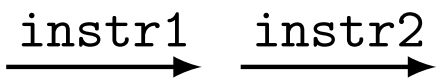
Concurrence : définition

Des processus sont **concurrents** s'ils existent en même temps.

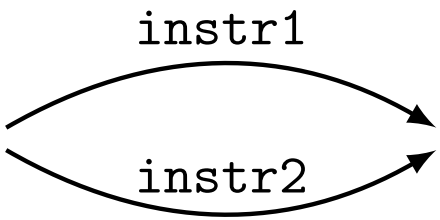
Modélisation de traitements parallèles :

1. graphes de processus
2. structure `parbegin` / `parend`

Concurrence : graphes de processus



`instr1` s'exécute avant `instr2`.



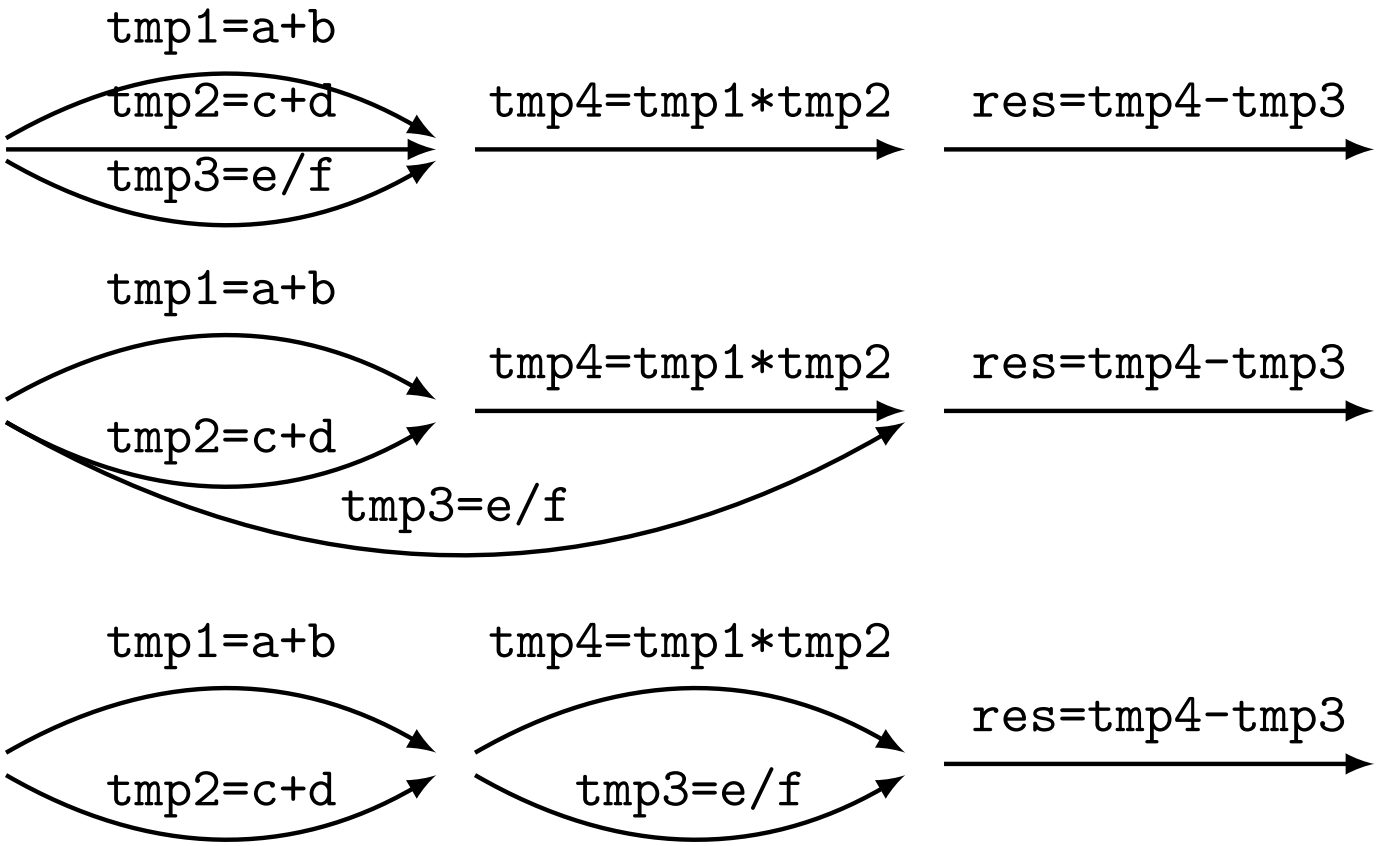
`instr1` et `instr2` s'exécutent en parallèle.

Chaque instruction doit être atomique.

Concurrence : graphes de processus

Exemple : `res = ((a+b) * (c+d)) - (e/f);`

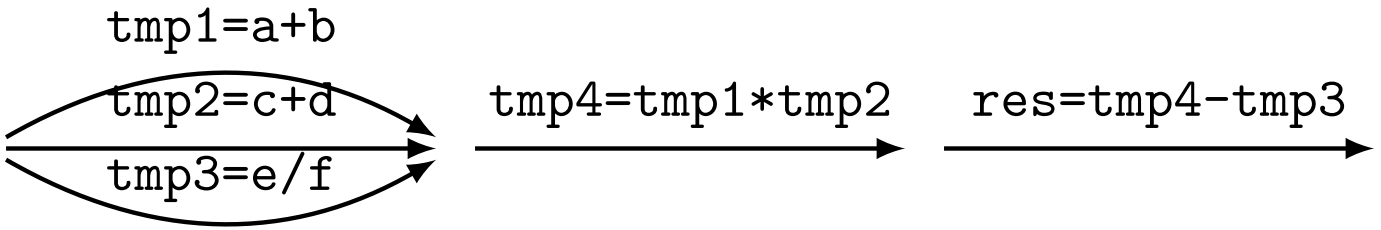
3 solutions correctes, laquelle est la meilleure ?

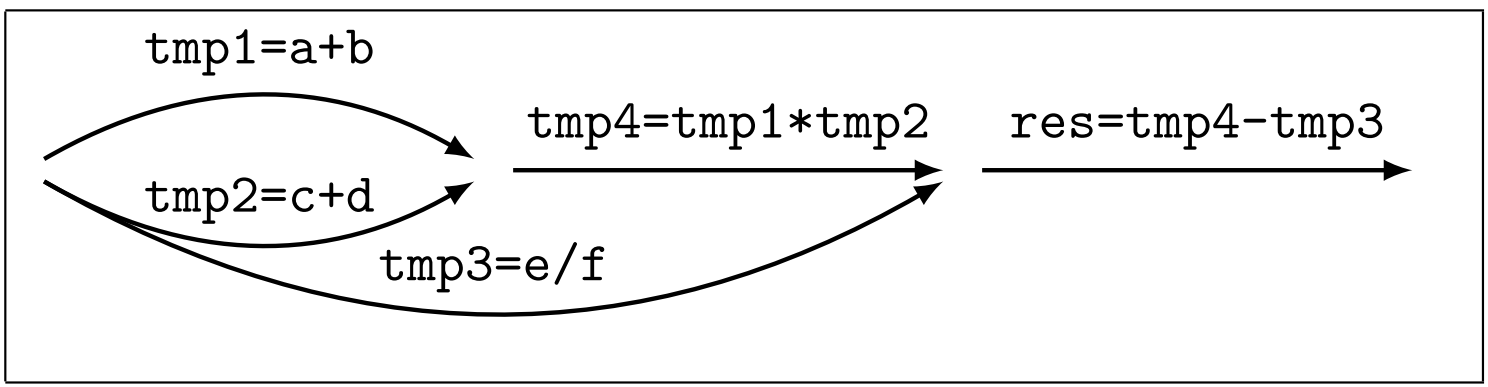


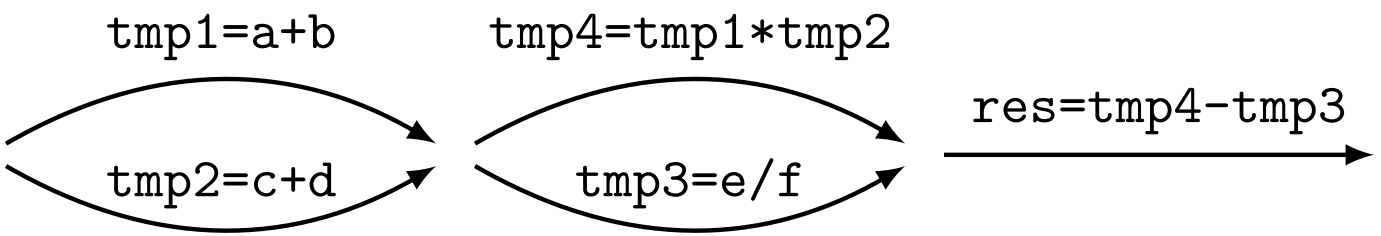
Concurrence : graphes de processus

Exemple : `res = ((a+b) * (c+d)) - (e/f);`

3 solutions correctes, laquelle est la meilleure ?







Concurrence : graphes de processus

Exercice : Proposer un graphe de processus pour le calcul de la racine d'une équation du second degré de manière parallèle :

```
res = (-b + (b**2 - 4*a*c)**.5) / (2*a);
```

Concurrence : structure parbegin / parend

`parbegin / parend` : structure de contrôle pour le parallélisme (Dijkstra, 1965)

```
...
parbegin;
        instruction 1;
        ...                 } exécutées en parallèle
        instruction k;
parend;
...
```

Concurrence : structure parbegin / parend

Exemple : `res = ((a+b) * (c+d)) - (e/f);`

```
parbegin;
    tmp1 = a+b;
    tmp2 = c+d;
    tmp3 = e/f;
parend;
tmp4 = tmp1*tmp2;
res = tmp4-tmp3;
```

Concurrence : graphes de processus

Exercice : Proposer de paralléliser le calcul de la racine d'une équation du second degré à l'aide de la structure `parbegin/parend` :

```
res = (-b + (b**2 - 4*a*c)**.5) / (2*a);
```

Concurrence : processus indépendants ou coopérants

Des processus concurrents peuvent être **indépendants** ou **coopérants** (exécution d'une tâche commune).

- *Intérêts* d'être **coopérant** :
  - optimise l'utilisation de ressources partagées
  - exemple : CPU, fichier, base de données, file, imprimante...
  - permet également l'échange d'informations entre processus
- ... mais *inconvénients* :
  - gérer les accès concurrents à ces ressources

Concurrence : accès concurrents

Exemple d'**accès concurrent** à une **variable partagée** "a".

thread 1 : `a++;` thread 2 : `a++;`

en langage de plus bas niveau (i.e. une fois compilé) :

thread 1 :
```
tmp = a
a = tmp + 1;
```
thread 2 :
```
tmp = a
a = tmp + 1;
```

une exécution possible au niveau du CPU :

thread actif	instruction	contexte
		a $\rightarrow$ 0
thread 1	`tmp = a;`	a $\rightarrow$ 0, `tmp` $\rightarrow$ 0
thread 2	`tmp = a;`	a $\rightarrow$ 0, `tmp` $\rightarrow$ 0
thread 2	`a = tmp + 1;`	a $\rightarrow$ 1, `tmp` $\rightarrow$ 0
thread 1	`a = tmp + 1;`	a $\rightarrow$ 1, `tmp` $\rightarrow$ 0

Ainsi "a" passe de 0 à 1 après deux incrémentations...

Concurrence : accès concurrents

Il faut donc identifier des **sections critiques** durant lesquelles le processeur a un accès exclusif aux variables :

thread 1 :
```
// debut section critique
a++;
// fin section critique
```

thread 2 :
```
// debut section critique
a++;
// fin section critique
```

On parle d'**exclusion mutuelle**.

Concurrence : exclusion mutuelle par algorithmes

Une façon de résoudre le problème de l'exclusion mutuelle est d'utiliser un algorithme, par exemple l'**algorithme de Dekker** (ici entre 2 threads)

```
// initialisation variables partagées : veux[] et peux
veux[0] = false; veux[1] = false; peux = 0;
```

```
// thread 0
veux[0] = true;
while (veux[1] == true) {
  if (peux != 0) {
    veux[0] = false;
    while (peux != 0) {
      // ne rien faire
    }
    veux[0] = true;
  }
}
// section critique ici, puis :
peux = 1
veux[0] = false
```

```
// thread 1
veux[1] = true;
while (veux[0] == true) {
  if (peux != 1) {
    veux[1] = false;
    while (peux != 1) {
      // ne rien faire
    }
    veux[1] = true;
  }
}
// section critique ici, puis :
peux = 0
veux[1] = false
```

Concurrence : exclusion mutuelle par algorithmes

Inconvénient de l'algorithme de Dekker :

```
// thread 0                    | // thread 1
...                            | ...
    while (peux != 0) {        |     while (peux != 1) {
      // ne rien faire         |       // ne rien faire
    }                          |     }
...                            | ...
```

→ attente active, utilise des cycles CPU inutilement...

solutions plus efficaces :

- verrous d'exclusion mutuelle (mutex)
- sémaphores

Concurrence : verrous d'exclusion mutuelle (mutex)

Un **verrou d'exclusion mutuelle** (ou **mutex**) permet de gérer l'accès concurrent à **une** ressource.

Un processus souhaitant utiliser cette ressource :

- peut demander à utiliser le verrou
- si le verrou est déjà utilisé, le processus reste en attente (passive)
- dès que le verrou devient inutilisé, un processus en attente obtient le verrou et redevient actif

Concurrence : verrous d'exclusion mutuelle (mutex)

Illustration sur les threads.

- La fonction **pthread_mutex_init** crée un verrou d'exclusion mutuelle (**mutex**). Il en existe de différents types (attributs pointés par le paramètre **mutexattr**, par défaut **mutexattr = NULL**). L'identificateur du verrou est placé dans la variable pointée par **mutex**.

```
int pthread_mutex_init(
        pthread_mutex_t *mutex,
        const pthread_mutexattr_t *mutexattr);
```

- La fonction **pthread_mutex_destroy** détruit le verrou.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Concurrence : verrous d'exclusion mutuelle (mutex)

- La fonction **pthread_mutex_lock** tente de bloquer le verrou (met le thread en attente s'il est déjà bloqué),
- La fonction **pthread_mutex_unlock** le débloque,
- La fonction **pthread_mutex_trylock** tente de bloquer le verrou, et échoue s'il est déjà bloqué.

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

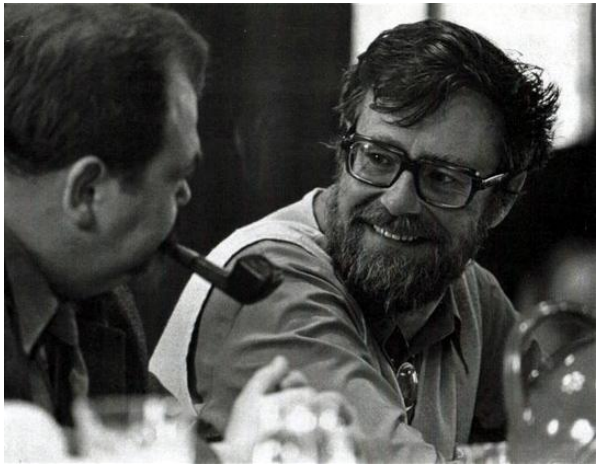
Exercices

Les programmes ci-dessous sont dans *Bibliotheque/ASR3-Systemes/CONCURRENCE*.

- Corrigez le programme `course.c` afin que le compteur affiche bien la valeur correcte.
- Le programme `subtab.c` calcule la somme des éléments d'un tableau.
  - Corrigez ce programme afin que la somme soit correcte lorsque plusieurs threads se partagent le travail.
  - En passant d'un thread à deux threads sur vos machines dual-core, la durée du calcul est bien divisée par deux, non ?
- Le programme `prod_cons.c` simule un tube reliant deux threads. Corrigez-le pour qu'il fonctionne avec plusieurs producteurs et plusieurs consommateurs.

Concurrence : sémaphores

Une généralisation des mutex : les **sémaphores**.



Dijkstra (1930-2002)

The origin of the complications [...] is the fact that [...] when a process wants to react to the current value of a common variable, its value may be changed by the other processes between the moment of its inspection and the following effectuation of the reaction to it.

(Dijkstra, EWD123, 1965)

Concurrence : sémaphores

Un **sémaphore** est comme un entier, hormis 3 différences :

1. il peut être **initialisé** à n'importe quelle valeur, mais ensuite il ne peut être qu'incrémenté ($+1$) ou décrémenté (-1).

```
initialiser(sem, k);
```

2. si un thread **décrémente** le sémaphore, et que sa valeur devient négative, ce thread se bloque.

```
decrementer(sem);
```

3. si un thread **incrémente** le sémaphore, et que certains threads sont bloqués, alors l'un des threads bloqués est débloqué.

```
incrementer(sem);
```

Concurrence : sémaphores

Ce que représente la **valeur** n d'un sémaphore :

$n > 0$ n ressources disponibles, donc les threads peuvent décrémenter n fois sans bloquer

$n == 0$ pas de thread bloqué, mais si un thread décrémente, il sera bloqué

$n < 0$ n représente le nombre de threads bloqués.

Concurrence : sémaphores

Les sémaphores sont plus généraux que les signaux et les mutex :

- un sémaphore **initialisé à** 0 revient à un échange de **signaux**

```
initialiser(sem, 0); // code commun
```

// thread 0	// thread 1
...	...
a = init();	decremente(sem);
incremente(sem);	lire(a); // maintenant a est initialisée
...	...

- un sémaphore **initialisé à** 1 est un **mutex**

```
initialiser(sem, 1); // code commun
```

// thread 0	// thread 1
...	...
decremente(sem);	decremente(sem);
// section critique	// section critique
incremente(sem);	incremente(sem);
...	...

Concurrence : sémaphores

Un exemple plus complet : producteur/consommateur avec buffeur borné

```
int bufferSz;
msg buffer[bufferSz];
initialiser(semMutexIn, 1);          // mutex pour l'écriture
initialiser(semMutexOut, 1);         // mutex pour la lecture
initialiser(semNonPlein, bufferSz);// sémaphore sur la taille du buffer
initialiser(semNonVide, 0);          // signal buffer non vide
```

```
// thread 0
void produce(msg m) {
  decremente(semNonPlein);
  decremente(semMutexIn);
  buffer[in] = msg;
  in = (in + 1) % bufferSz;
  incremente(semMutexIn);
  incremente(semNonVide);
}
```

```
// thread 1
msg consume() {
  decremente(semNonVide);
  decremente(semMutexOut);
  msg = buffer[out];
  out = (out + 1) % bufferSz;
  incremente(semMutexOut);
  incremente(semNonPlein);
  return(msg);
}
```

Concurrence : sémaphores

Illustration sur les threads, en C.

Remarque : Les sémaphores, qui font partie de la norme POSIX, ne sont pas implémentés dans toutes les bibliothèques de threads.

Concurrence : sémaphores

- La primitive **sem_init** crée un sémaphore, place l'identificateur du sémaphore à l'endroit pointé par **sem** et l'initialise à **value**. Si **pshared** est nul, le sémaphore est local au processus lourd.

```
#include <semaphore.h>

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

int sem_destroy(sem_t *sem);
```

→ équivalent de `initialiser(sem, value);`

Concurrence : sémaphores

- Les fonctions **sem_wait** et **sem_post** sont les primitives :

```
int sem_wait(sem_t *sem); // décrémente
int sem_post(sem_t *sem); // incrémente
```

- La fonction **sem_trywait** échoue (au lieu de bloquer) si la valeur du sémaphore est nulle.

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

- La fonction **sem_getvalue** consulte la valeur courante du sémaphore.

```
int sem_getvalue(sem_t *sem, int *sval);
```

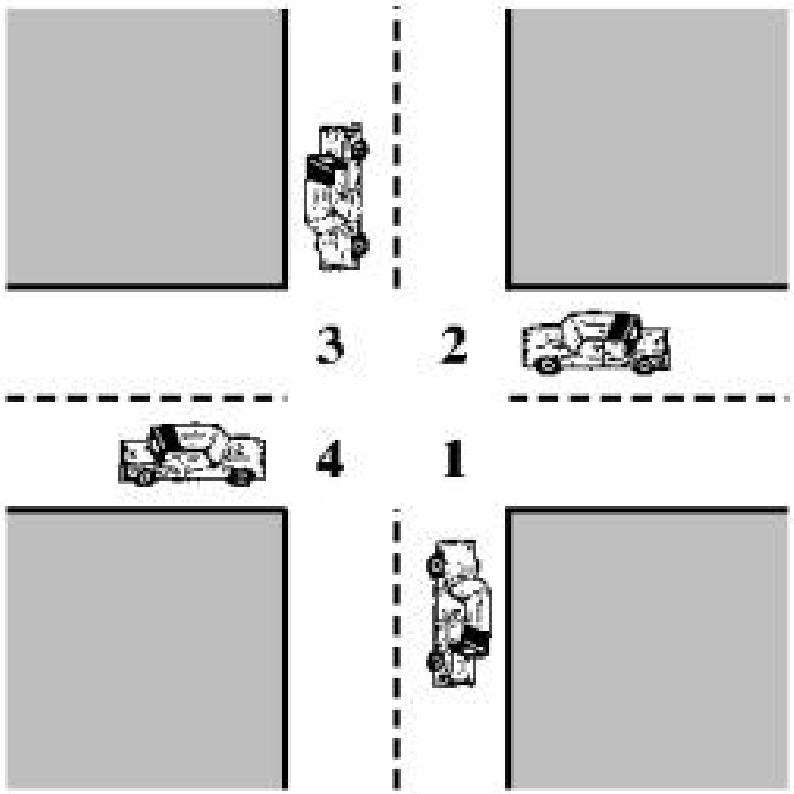
Exercice

Shifumi sémaphores :

- reprendre la version du Shifumi avec les threads.
- désormais les threads écrivent leur résultat dans un buffer, qui ne peut contenir que 2 de ces résultats à la fois.
- le père, par contre, peut utiliser des tableaux plus grands.

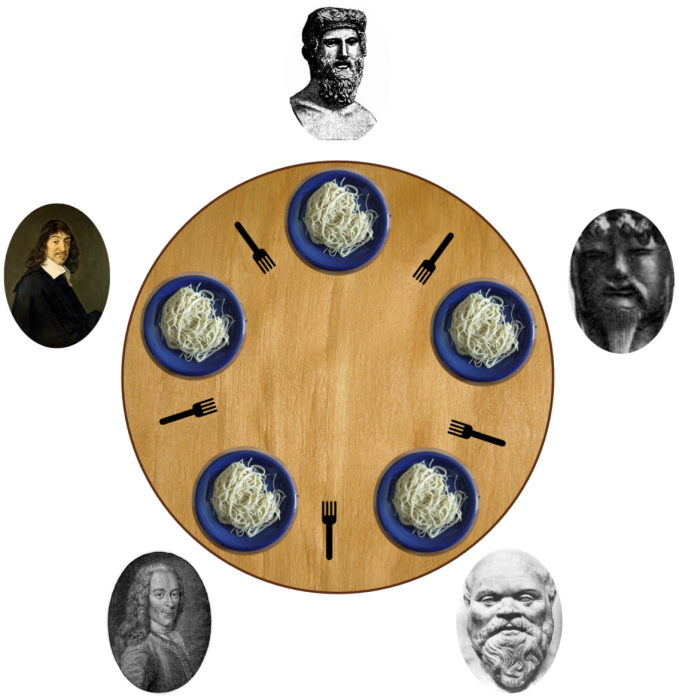
Interblocage (deadlock)

Quatre priorités à droite...



source : Jonathan Nieto

Interblocage : le dîner des philosophes



source : Benjamin D. Esham

Un philosophe :

1. pense,
2. est affamé, ou
3. mange

Un philosophe a besoin de ses deux fourchettes pour manger...

Que se passe-t-il si chaque philosophe est affamé, prend sa fourchette gauche, et attend que la droite se libère ?

Interblocage : définition

Un ensemble de processus (thread) est en **interblocage** si chaque processus de l'ensemble attend qu'un événement advienne, événement que seul un autre processus de l'ensemble peut engendrer.

Interblocage : condition

Pour qu'il y ait interblocage, il faut les 4 conditions suivantes :

1. **exclusion mutuelle** : chaque ressource est soit attribuée à un seul, soit disponible
2. **détention et attente** : les entités qui détiennent des ressources peuvent en demander de nouvelles (sans relâcher celle qu'elles détiennent)
3. **pas de réquisition** : les ressources obtenues par un processus ne peuvent pas lui être retirées de force
4. **attente circulaire** : il y a un cycle orienté d'au moins deux entités, chacune en attente d'une ressource détenue par une autre entité du cycle.

Interblocage : condition

Pour résoudre un interblocage, il faut casser au moins une de ces conditions :

1. **exclusion mutuelle** : difficile (par exemple en utilisant une file)
2. **détention et attente** : ne pas autoriser une entrée en section critique si le processus bloque déjà une ressource : difficile aussi (regrouper les sections critiques...)
3. **pas de réquisition** : une réquisition implique d'interrompre le processus, de lui retirer la ressource en cours d'utilisation...
4. **attente circulaire** : analyse du graphe à chaque entrée en section critique : lourd...

La meilleure solution est d'y penser à la **conception** du programme.

Autre outil (non abordé ici) : les conditions (pthread_cond_init...).

Exercice de synthèse : processus lourds vs légers

- le main() affiche "3... 2... 1... Go !"
- lorsque Go ! s'affiche, un signal `SIGUSR1` est émis vers ce même processus, cela déclenche la fonction `compet`
- la fonction `compet` fait :
  - un `fork+exec(lourds)`
  - un `fork+exec(legers)`
- dans `lourds` : on lance 100 forks, chacun tire une valeur aléatoire entre 0 et 9, et la renvoie au père via un pipe (le même pour tous). Puis on affiche la somme de ces valeurs.
- dans `legers` : on lance 100 threads, chacun tire une valeur aléatoire entre 0 et 9, et la renvoie dans une variable du processus principal, protégée par mutex. Puis le processus principal affiche la somme.

Sources

- Cours d'ASR3 Système de Serge Dulucq
- "Programmation concurrente multi-thread", Supports de cours de Françoise Baude, Université de Nice Sophia-Antipolis
- EWD123, Edsger W. Dijkstra
- "The Little Book of Semaphores" (version 2.1.5), Allen B. Downey