TD 6 : Processus, fork, signaux

Objectifs pédagogiques :

- **création de processus, wait**
- **signaux**

Introduction

Dans ce TD, nous allons étudier l'API proposée par POSIX pour la création et la gestion de processus. Il faudra donc un système POSIX (en particulier pas windows) pour compiler et exécuter nos programmes.

Le standard POSIX définit une API en C à bas niveau pour l'interaction entre les processus utilisateurs et le système d'exploitation. Cette API reste accessible dans les programmes C++.

0.1 Fork en séquence et en parallèle

Question 1. Proposez un programme qui crée N processus fils en parallèle, puis attend leur terminaison. Chaque processus créé doit afficher son *pid*, le *pid* de son pere, et son numéro d'ordre de création.

Donc on boucle sur fork, puis on boucle sur wait.

fork engendre un nouveau processus, égal en presque tout points au processus initial, mais la valeur de retour de fork diffère.

Le père obtient le pid du nouveau fils. Le fils obtient 0, il pourra retrouver le pid de son père au besoin avec *getppid*.

Comme le fils a une copie des variables du père au moment du fork, il est aisé d'afficher le i du numero d'iteration. PAS de race condition ici, entre le pere qui refait un ++ sur i et le fils qui affiche i, ce sont des COPIES.

forkpar.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main () {
      int N = 4;
      std::cout << "main pid=" << getpid() << std::endl;
      for (int i =0 ; i < N ; i++ ) {
            if (fork() == 0) {
                  // code fils
                  std::cout << "fils " << i << " pid=" << getpid() << " ppid=" <<
                      getppid() << std::endl;
                  return i;
            }
      }
      for (int i =0 ; i < N ; i++ ) {
            wait(nullptr);
      }
      return 0;
}
```

Question 2. Chacun des fils doit rendre à la terminaison son numéro de création. Le processus

main doit afficher la valeur de sortie de chacun des fils et le pid du fils dont il vient de détecter la terminaison.

Donc on utilise un peu plus des options de wait :

- on lui passe un endroit ou ranger le status de sortie du processus mort (au lieu de nullptr qui signifie on ignore le résultat),
- on stocke la valeur de retour qui sera le pid du fils qui est mort (ou -1 + set errno en cas de souci, comme pas de fils à wait.)

Attention le status c'est plus que la valeur de retour du processus, on peut aussi savoir un peu comment il est mort. La valeur de retour d'un processus c'est sur 8 bit (0 à 255), on l'interroge avec la macro WEXITSTATUS.

forkpar.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main () {
      int N = 4;
      std::cout << "main pid=" << getpid() << std::endl;
      for (int i =0 ; i < N ; i++ ) {
            if (fork() == 0) {
                  // code fils
                  std::cout << "fils " << i << " pid=" << getpid() << " ppid=" <<
                      getppid() << std::endl;
                  return i;
            }
      }
      for (int i =0 ; i < N ; i++ ) {
            int status;
            int pid = wait(&status);
            std::cout << "Fin du fils de pid=" << pid << " return " << WEXITSTATUS(
                status) << std::endl;
      }
      return 0;
}
```

NB : il n'y a rien à inventer ici. "man 3 wait" et puis, ben on lit/on apprend l'API. http://man.cx/wait(3)

Question 3. En supposant que les fils ont la même durée d'exécution, vont-ils mourir dans l'ordre de leur création ? Modifiez le programme pour que le main attende la fin des fils dans l'ordre de leur création.

Extrait du man The pid argument specifies a set of child processes for which status is requested. The `waitpid(), kill()` and other API functions use this convention :

- If pid is equal to `(pid_t)-1`, status is requested for any child process. In this respect, waitpid() is then equivalent to wait().
- If pid is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If pid is 0, status is requested for any child process whose process group ID is equal to that of the calling process.
- If pid is less than `(pid_t)-1`, status is requested for any child process whose process group ID is equal to the absolute value of pid.

Donc non, ce sont des hypothèses sur l'ordonnanceur qu'il nous faudrait. On ne sait pas dans quel ordre ils mourront et encore moins dans quel ordre leur terminaison sera notifiée au père.

Il nous faut la version complète du wait, càd *waitpid* qui permet d'attendre un processus ou groupe de processus particulier.

$pid\_twaitpid(pid\_tpid, int * stat\_loc, intoptions)$;

Le pid c'est :

NB : Cette API sur les pid vaut la peine d'être commentée, elle est récurrente dans le standard (e.g. kill) :

- $pid = -1$ c'est n'importe quel processus
- $pid > 0$ c'est explicitement un pid de processus précis
- $pid = 0$ n'importe qui, mais de mon gid (group id)
- $pid < -1$ n'importe qui du groupe de $gid = -pid$, donc on vise un groupe.

Ici on veut attendre un fils particulier, dont il nous faut donc connaître le pid. Le père va stocker les pid de ses fils rendus par $fork()$ dans un tableau ou un vector.

forkpar.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main () {
      int N = 4;
      std::cout << "main pid=" << getpid() << std::endl;
      int pids [N];
      for (int i =0 ; i < N ; i++ ) {
            int pid = fork();
            if (pid == 0) {
                  // code fils
                  std::cout << "fils " << i << " pid=" << getpid() << " ppid=" <<
                      getppid() << std::endl;
                  return i;
            } else {
                  pids[i] = pid;
            }
      }
      for (int i =0 ; i < N ; i++ ) {
            int status;
            int pid = waitpid(pids[i],&status,0);
            std::cout << "Fin du fils de pid=" << pid << " return " << WEXITSTATUS(
                status) << std::endl;
      }
      return 0;
}
```

Question 4. On reprend la question 1, mais cette fois proposez un programme qui crée N processus à travers une chaîne de créations, c'est à dire qu'il crée un fils qui crée un fils... sur N générations. Le père ne doit se terminer que quand tous ses fils sont terminés.

On sort de la boucle après avoir créé un seul fils, ou quand on est le Nieme fils.

On wait sauf si on est le dernier de la chaine.

Si le dernier wait, il verra -1 comme valeur de retour mais il ne sera pas bloqué.

forkseq.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main () {
      int N = 4;
      std::cout << "main pid=" << getpid() << std::endl;
      int i = 0;
      for ( ; i < N ; i++ ) {
            int pid = fork();
            if (pid == 0) {
                  // code fils
                  std::cout << "fils " << i << " pid=" << getpid() << " ppid=" <<
                      getppid() << std::endl;
            } else {
                  break;
            }
      }
      if (i != N) {
            int status;
            int pid = wait(&status);
            std::cout << "Detection par " << getpid() << " Fin du fils de pid=" << pid
                << " return " << WEXITSTATUS(status) << std::endl;
      }
      return 0;
}
```

0.2 Fork arbitraires et Arbre des processus

On considère le programme suivant :

fork1.cpp

```cpp
#include <iostream>
#include <unistd.h>

int main () {
      int N = 4;
      int i = 0;
      std::cout << "pid=" << getpid() << std::endl;
      while (i < N) {
            i++;
            int j = i;
            if (fork() == 0) {
                  if (i%2==0) {
                        N = i - 1;
                        i = 0;
                  } else {
                        N = 0;
                  }
                  std::cout << "pid=" << getpid() << " ppid=" << getppid() << " j=" << j <<
                      " N=" << N << std::endl;
            }
      }
      return 0;
}
```

Question 5. En comptant le processus initial, combien de processus engendre l'execution du programme ? Donnez l'arbre des descendances de processus, et l'affichage de chacun d'entre eux.

10 processus total avec le main.
N donne le nombre de fils, j varie de 1 à N.
Si j est pair, on engendre j-1 fils.

```
[ythierry@localhost src]$ ./a
pid=26918
pid=26919 ppid=26918 j=1 N=0
pid=26920 ppid=26918 j=2 N=1
pid=26921 ppid=26918 j=3 N=0
pid=26922 ppid=26918 j=4 N=3

# pere = fils N=1
pid=26923 ppid=26920 j=1 N=0

# pere = fils N=3
pid=26924 ppid=26922 j=1 N=0
pid=26925 ppid=26922 j=2 N=1
pid=26926 ppid=26922 j=3 N=0

# pere = fils N=1 du fils N=3
pid=26927 ppid=26925 j=1 N=0
```

On peut faire un dessin arborescent au tableau.

Question 6. Modifiez le programme pour que le père ne se termine qu'après que tous les autres processus soient terminés. On n'attendra pas plus de fils qu'on en a crée et on ne limitera pas le parallélisme.

On peut utiliser "wait(nullptr)", N fois vu que c'est le nombre de fils.
On le place après la boucle.
Attention il faut <sys/wait.h> en plus de <unistd.h>.

fork1cor.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main () {
      int N = 4;
      int i = 0;
      std::cout << "pid=" << getpid() << std::endl;
      while (i < N) {
            i++;
            int j = i;
            if (fork() == 0) {
                  if (i%2==0) {
                        N = i - 1;
                        i = 0;
                  } else {
                        N = 0;
                  }
                  std::cout << "pid=" << getpid() << " ppid=" << getppid() << " j=" << j
                       << " N=" << N << std::endl;
```

```
            }
        }
        for (i=0 ; i < N ; i++) {
          wait(nullptr);
        }
        return 0;
}
```

0.3 Signaux

On utilisera les fonctions pour les signaux dans les exercices suivants :

- Mise en place d'un handler pour un signal : `signal-handler* signal(int sig, signal-handler *)`; où le `signal-handler` est une fonction prenant un *int* (le numéro du signal reçu) et rendant *void*, ou une des macros SIG_IGN (ignorer, le signal sera perdu), ou SIG_DFL (comportement par défaut face au signal, mourir ou ignorer le plus souvent)
- Envoyer un signal à un processus ou groupe de processus : `int kill(int pid, int signal)`
- Demander au système de nous envoyer un $SIGALRM$ au bout de n secondes : `int alarm(int seconds)`
- Manipulation d'un ensemble de signaux (type `sigset_t`): `sigfillset(sigset *); sigemptyset(sigset*); sigdelset(sigset *,signal); sigaddset(sigset*,signal);`
- Se bloquer en attente des signaux qui ne sont *PAS* dans le masque argument : `int sigsuspend(sigset *mask)`. NB: si un des signaux est "pending" il est traité immédiatement.
- Bloquer (mettre en attente dans "pending") les signaux qui sont dans le masque (ensemble de signaux) argument : `int sigprocmask(int how,sigset *mask,sigset * old)`. On passera SIG_BLOCK dans *how* pour ajouter un signal au masque actif, et *nullptr* dans old si on ne se soucie pas du masque précédent.

0.4 Tour par tour

On veut construire un programme ou deux processus s'exécutent tour à tour. Pour cela on demande de n'utiliser que le mécanismes des signaux.

Question 7. Ecrire un tel programme :

- Le père commence le premier tour, mais il attend une seconde avant de commencer. Le fils commence son tour par attendre un signal.
- Chaque processus quand il a la main affiche un message avec son pid, et le tour auquel il en est.
- A la fin de chaque tour on passe la main à l'autre processus, pour cela on lui envoie un signal, puis on se suspend en attente de sa réponse.
- Chacun des processus fait M tours d'alternance.

Conceptuellement : On place un handler, par exemple qui affiche simplement de pid du récepteur. Sinon on aura le comportement par défaut, avec SIGINT ce serait mourir; pour e.g. SIGUSR1 ce serait ignorer ce qui n'est pas bon.

signal(SIGUSR1,[](int))

On signale l'autre avec kill :

kill(other,SIGUSR1)

On attend la signalisation avec :

sigsuspend(SIGUSR1)
c'est tout.
On fork, les deux processus bouclent sur un affichage + passer la main.

alternate.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <csignal>


int main () {
      const int NBROUND= 5;
      std::cout << "main pid=" << getpid() << std::endl;
      // pid pere
      pid_t mainpid = getpid();

      // un masque pour passer a sisgsuspend dans les deux processus
      sigset_t setneg; // neg car c'est tout sauf SIGINT
      sigfillset(&setneg);
      sigdelset(&setneg,SIGINT);

      // un handler commun aux deux processus
      signal(SIGINT,[](int sig){ std::cout << "processus " << getpid() << "recu signal "
           << sig << std::endl; });

      ///// BLOC SUIVANT = REPONSE QUESTION 2 ***************
      // proteger le fils du premier signal avant sigsuspend
      sigset_t setpos; // pos car c'est seulement SIGINT
      sigemptyset(&setpos);
      sigaddset(&setpos,SIGINT);
      // SIG_BLOCK = ajouter au masque actif
      // l'ensemble de signaux a ajouter = setpos
      // l'ancien masque si ca nous interesse, ou nullptr
      sigprocmask(SIG_BLOCK,&setpos,nullptr);
      ////////////////////////////////////*********************

      // pid fils ou 0
      pid_t pid =fork();

      if (pid == 0) {
            // code fils
            std::cout << "Naissance fils "<< getpid() << std::endl;
            for (int i = 0 ; i < NBROUND ; i++) {
                  sigsuspend(& setneg);

                  std::cout << "fils " << i << " pid=" << getpid() << " ppid=" <<
                      getppid()
                              << " Tour : " << i << std::endl;


                  kill(mainpid,SIGINT);
            }
            std::cout << "Mort du fils" << std::endl;
            return 0;
      } else {
            // donner du temps au fils pour naitre et atteindre sigsuspend
            // En question 1 : on fait un sleep ici. Supprime en question 2
            //sleep(1);
```

```
            // code pere
            for (int i = 0 ; i < NBROUND ; i++) {
                  std::cout << "Pere " << i << " pid=" << getpid()
                                                                  << " Tour : " << i <<
                                                                      std::endl;

                  // si le fils n'existe pas encore ici, on perd le premier signal
                  kill(pid,SIGINT);
                  sigsuspend(& setneg);
            }
            wait(nullptr);
            std::cout << "Mort du pere" << std::endl;
      }
      return 0;
}
```

Question 8. Si le père n'attend pas avant de démarrer, on constate que parfois le fils reçoit le premier signal AVANT de faire le *sigsuspend*. Quel est l'effet sur le programme de ce comportement ? Proposez une correction utilisant les masques (sigprocmask) pour corriger ce problème.

Si le fils reçoit le signal AVANT le sigsuspend (et donc le traite), le sigsuspend deviendra bloquant, le programme entier se bloque quand le fils s'échoue sur sigsuspend.

Donc on invoque sigprocmask, AVANT de faire le fork.

Le masque n'empêche pas du tout de faire sigsuspend, l'appel à sigsuspend revient tout de suite si le signal était recu et pending.

0.5 Signaux, Alarmes

On désire réaliser un programme qui créé N processus fils en parallèle, puis attend T secondes avant d'envoyer un SIGINT à tous ses fils. A leur création, les processus fils se bloquent en attente du signal de leur père ; à la réception de ce signal, ils affichent un message indiquant qu'ils vont se terminer. Le processus père attend que tous ses fils soient finis avant de se terminer en affichant un message "Fin du programme".

Question 9. Ecrivez ce programme.

alarm.cpp

```
#include <csignal>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main () {
      int N = 4;
      std::cout << "main pid=" << getpid() << std::endl;
      int pids[N];
      for (int i =0 ; i < N ; i++ ) {
            int pid = fork();
            if (pid == 0) {
                  // code fils
                  std::cout << "fils " << i << " pid=" << getpid() << " ppid=" <<
                      getppid() << std::endl;
```

```
                  // handler sigint
                  signal(SIGINT,[](int sig) {
                        std::cout << "Received SIGINT, quitting pid=" << getpid() << std
                            ::endl;
                        exit(0);
                  });
                  // bloquer en attente de SIGINT
                  sigset_t set;
                  sigfillset(&set);
                  sigdelset(&set,SIGINT);
                  sigsuspend(&set);
            } else {
                  pids[i] = pid;
            }
      }
      alarm(3);
      // handler pour sigalrm
      // on pourrait aussi kill les fils dans le handler directement, mais autant eviter
      // la semantique des handlers est assez complique comme ca
      // ici un handler vide (!= SIG_IGN on prend quand même le signal !)
      signal(SIGALRM,[](int) {});

      // bloquer en attente de SIGALRM
      sigset_t set;
      sigfillset(&set);
      sigdelset(&set,SIGALRM);
      sigsuspend(&set);

      // on a été réveillé
      for (int i=0; i < N ; i++) {
            kill(pids[i],SIGINT);
      }

      for (int i =0 ; i < N ; i++ ) {
            wait(nullptr);
      }

      return 0;
}
```

Question 10. Le père peut-il notifier tous ses fils sans avoir stocké leur pid ? Ecrivez une version qui a ce comportement.

Retour sur l'API par groupes de pid déjà vue au début du TD, oui, il suffit de notifier 0 ça crée un broadcast du signal à tout le groupe.

Par contre on va s'auto kill si on ne fait pas attention ! Donc mettre un handler SIG_IGN pour SIGINT avant le broadcast.

alarm.cpp

```
      alarm(3);
      // handler pour sigalrm
      // on pourrait aussi kill les fils dans le handler directement, mais autant eviter
      // la semantique des handlers est assez complique comme ca
      signal(SIGALRM,[](int) {});

      // bloquer en attente de SIGALRM
```

```
8       sigset_t set;
9       sigfillset(&set);
10      sigdelset(&set,SIGALRM);
11      sigsuspend(&set);
12
13      // on a ete reveille
14      // se proteger du SIGINT pour ne pas s'auto kill
15      signal(SIGINT, SIG_IGN);
16      // ok on broadcast
17      kill(0,SIGINT);
18
19      for (int i =0 ; i < N ; i++ ) {
20            wait(nullptr);
21      }
```

Question 11. La terminaison (exit) d'un processus fils engendre un signal $SIGCHLD$ à destination du père, le comportement par défaut étant de l'ignorer. Le père peut-il s'appuyer sur ce mécanisme plutôt que sur *wait* pour attendre la terminaison de ses fils ?

Non, on ne peut pas compter les signaux recus, on a toutes les chances d'en perdre en fait, pas de façon de compter fiable.

Les signaux "pending" (i.e. délivrés par l'OS au processus, mais qu'il n'a pas encore traité, par exemple parce qu'il n'est pas élu) sont stockés dans un bitset, on ajoute les signaux recus avec un OR bit à bit.

Donc on perd des signaux $SIGCHLD$ potentiellement, pas de façon de contrôler ou remédier, à part utiliser *wait*.

TME 6 : Processus, Signaux

Objectifs pédagogiques :

- **fork, wait**
- **signaux**

1.1 Gitlab pour le TME

Les fichiers fournis sont a récupérer suivant la même procédure qu'au TME5 (faire un fork, puis cloner votre fork) sur le gitlab : https://pscr-gitlab.lip6.fr/ythierry/tme6/.
Vous ferez un push de vos résultats sur ce dépot pour soumettre votre travail.

1.2 Encore un Arbre des processus

forkexo2.cpp

```cpp
#include <iostream>
#include <unistd.h>

int main () {
      const int N = 3;
      std::cout << "main pid=" << getpid() << std::endl;

      for (int i=1, j=N; i<=N && j==N && fork()==0 ; i++ ) {
            std::cout << " i:j " << i << ":" << j << std::endl;
            for (int k=1; k<=i && j==N ; k++) {
                  if ( fork() == 0) {
                        j=0;
                        std::cout << " k:j " << k << ":" << j << std::endl;
                  }
            }
      }
      return 0;
}
```

Question 1. En comptant le processus initial, combien de processus engendre l'execution du programme ? Donnez l'arbre des descendances de processus, et l'affichage de chacun d'entre eux.

10 processus en comptant le main.

```
[ythierry@localhost src]$ ./a.out
main pid=32503

pid=32504 ppid=32503 i:j 1:3

pid=32505 ppid=32504 k:j 1:0
pid=32506 ppid=32504 i:j 2:3

pid=32507 ppid=32506 k:j 1:0
pid=32508 ppid=32506 k:j 2:0
pid=32509 ppid=32506 i:j 3:3

pid=32511 ppid=32509 k:j 1:0
pid=32512 ppid=32509 k:j 2:0
pid=32513 ppid=32509 k:j 3:0
```

Question 2. Modifiez le programme pour que le père ne se termine qu'après que tous les autres processus soient terminés. On n'attendra pas plus de fils qu'on en a crée et on ne limitera pas le parallélisme.

NB: On recommande d'ajouter et de maintenir à jour un compteur du nombre de fils engendrés par un processus, et de faire le `wait` dans une boucle séparée après le corps du programme fourni.

Ok c'est plus dur car il y a deux fork. Le premier est en plus dans une condition, il faut tester pour être sur de devoir faire un wait. Et il faut bien lire les conditions et comprendre le code pour ecrire les bons wait, bref galère.

Comme la question exige en plus de ne pas limiter le parallélisme, le plus simple reste de compter les fils crées et de wait à la fin. Attention on incrémente si on est le pere dans le fork, mais SINON on remet à zéro le compteur (sinon on comptera ses frères ainés comme des fils).

forkexo2.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main () {
      const int N = 3;
      std::cout << "main pid=" << getpid() << std::endl;

      int nbfils =0 ;
      for (int i=1, j=N ; i<=N && j==N ; i++ ) {
            if (fork() !=0) {
                  nbfils++;
                  break;
            } else {
                  nbfils = 0;
            }
            std::cout << "pid=" << getpid() << " ppid=" << getppid()
                                    << " i:j " << i << ":" << j << std::endl;
            for (int k=1; k<=i && j==N ; k++) {
                  if ( fork() == 0) {
                        nbfils = 0;
                        j=0;
                        std::cout << "pid=" << getpid() << " ppid=" << getppid()
                                                << " k:j " << k << ":" << j << std::
                                                    endl;
                  } else {
                        nbfils++;
                  }
            }
      }
      for (int i=0 ; i < nbfils ; i++) {
            int status;
            int pid = wait(&status);
            std::cout << "detection par " << getpid() << " de fin du fils de pid=" <<
                pid << std::endl;
      }
      std::cout << "Fin du processus " << getpid() << std::endl;
      return 0;
}
```

Avec des affichages supplémentaires une execution du corrigé.

```
main pid=2472
```

```
pid=2474 ppid=2472 i:j 1:3

pid=2476 ppid=2474 i:j 2:3

pid=2475 ppid=2474 k:j 1:0
Fin du processus 2475

pid=2477 ppid=2476 k:j 1:0
Fin du processus 2477

pid=2478 ppid=2476 k:j 2:0
Fin du processus 2478

detection par 2474 de fin du fils de pid=2475

pid=2479 ppid=2476 i:j 3:3

detection par 2476 de fin du fils de pid=2477
detection par 2476 de fin du fils de pid=2478

pid=2480 ppid=2479 k:j 1:0
Fin du processus 2480

pid=2484 ppid=2479 k:j 2:0
Fin du processus 2484

pid=2485 ppid=2479 k:j 3:0
Fin du processus 2485

detection par 2479 de fin du fils de pid=2480
detection par 2479 de fin du fils de pid=2484
detection par 2479 de fin du fils de pid=2485

Fin du processus 2479
detection par 2476 de fin du fils de pid=2479

Fin du processus 2476
detection par 2474 de fin du fils de pid=2476

Fin du processus 2474
detection par 2472 de fin du fils de pid=2474

Fin du processus 2472
```

1.3 Combat de Signaux

Nous allons construire une application où deux processus s'envoient respectivement des signaux avec kill pour simuler un combat (Vador et Luke). Le principe est que chaque combattant (processus) alterne entre une phase de défense où il se protège (en ignorant les signaux), et une phase d'attaque où il envoie un signal (coup de sabre laser) à l'adversaire (attaque) mais devient en contrepartie vulnérable aux signaux pendant un moment.

On va modéliser également des points de vie, c'est-à-dire que le fait de recevoir un signal quand on est vulnérable (dans sa phase d'attaque) va décrémenter un compteur (initialement trois vies). Quand le compteur atteint 0, le processus meurt et rend la valeur 1. Quand l'attaquant détecte la mort de son adversaire, c'est-à-dire que son pid n'existe plus (ce qui cause une exception sur kill), il meurt également et rend la valeur 0.

Le temps passé en phase de défense (signal masqué) ou d'attaque (signal reçu) est aléatoire et compris entre 0.3 et 1 seconde.

Définir les trois fonctions suivantes :

- `void attaque (pid_t adversaire)` :
  - La phase d'attaque commence par installer un gestionnaire pour le signal SIGINT, qui décrémente les « points de vie » du processus et affiche le nombre de points restants. Si 0 est atteint il affiche que le processus se termine, et celui-ci se termine en retournant 1.
  - Ensuite le processus envoie un signal SIGINT à l'adversaire ; si cette invocation échoue, on suppose que l'adversaire a déjà perdu, et le processus sort avec la valeur 0 ;
  - Ensuite le processus s'endort pour une durée aléatoire.
- `void defense()`
  - La phase de défense consiste à désarmer le signal SIGINT en positionnant son action à SIG_IGN ;
  - Ensuite le processus s'endort pour une durée aléatoire.
- `void combat(pid_t adversaire)` : boucle indéfiniment sur une défense suivie d'une attaque et invoquez-la dans le corps des deux fils.

Question 3. Assemblez ces éléments pour créer un combat entre le processus principal (vador) et son fils (luke).

combat.cpp

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <time.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include "rsleep.h"

// points de vie, variable globale, dupliquee au moment du fork
int PV = 8;

// quand on est en attaque : decremente les PV
void handler (int sig) {
      if (sig == SIGUSR1) {
            PV--;
            printf("Attaque reçue par %d; PV restants %d\n",getpid(),PV);
            if (PV == 0) {
                  printf("Plus de vie pour %d; mort du processus.\n",getpid());
                  exit(1);
            }
      }
}

void attaque (pid_t adversaire) {
 signal(SIGUSR1,handler);
 if (kill(adversaire,SIGUSR1) < 0) {
   printf("Detection de Mort de l'adversaire de pid=%d \n",adversaire);
   exit(0);
 }
 randsleep();
}

void defense () {
```

```
  signal(SIGUSR1,SIG_IGN);

  randsleep();
}


void combat(pid_t adversaire) {
  while (1) {
    defense();
    attaque(adversaire);
  }
}

int main() {
  pid_t pere = getpid();
  pid_t pid = fork();
  srand(pid);
  if (pid == 0) {
        combat(pere);
  } else {
        combat(pid);
  }
  return 0;
}
```

Question 4. Pour attendre une durée aléatoire dans ce scenario on propose la fonction suivante `randsleep` suivante. En lisant le manuel de nanosleep https://man.cx/nanosleep, en particulier les erreurs possibles, expliquez la boucle qui est proposée dans ce code.

```
#include <time.h>

void randsleep() {
 int r = rand();
 double ratio = (double)r / (double) RAND_MAX;
 struct timespec tosleep;
 tosleep.tv_sec =0;
 // 300 millions de ns = 0.3 secondes
 tosleep.tv_nsec = 300000000 + ratio*700000000;
 struct timespec remain;
 while ( nanosleep(&tosleep, &remain) != 0) {
   tosleep = remain;
 }
}
```

> La doc du man nous révèle que se faire signaler cause un EINTR, et écrit le temps restant dans remain. On a bien l'intention de se faire signaler pendant qu'on dort, du moins en attaque, donc il faut faire cette boucle.

Question 5. Comment assurer que les deux processus qui s'affrontent utilisent une graine aléatoire différente ?

> On pose une graine avec srand, basée sur time(0) ou sur getpid.

On propose de modifier la défense de Luke (le fils) pour qu'il affiche les coups parés. On propose procéder de la manière suivante :

- positionner un handler qui affiche "coup paré" avec sigaction
- masquer les signaux avec sigprocmask ;
- s'endormir pendant une durée aléatoire avec randsleep ;
- invoquer sigsuspend pour tester si une attaque a eu lieu, et donc afficher le message "coup paré" si le signal a été reçu.

Question 6. Le combat est-il encore équitable ? Expliquez pourquoi.

> Non, on attend le signal = on pare à tous les coups.

1.4 Pseudo wait avec des signaux

Nous voulons implémenter la fonction `int wait_till_pid(pid_t pid)`, qui suspend l'exécution du processus appelant jusqu'à ce qu'il prenne connaissance de la terminaison de son processus fils de pid *pid*. La fonction retourne *pid* lorsque le processus fils *pid* se termine, ou -1 en cas d'erreur (fils *pid* n'existe pas).

N.B: Si l'appelant a créé d'autres fils et que l'un de ceux-ci se termine durant l'appel, alors cette terminaison sera définitivement perdue.

Question 7. Sans s'appuyer sur *waitpid*, simplement avec *wait* et une boucle, implanter la fonction `wait_till_pid`.

> cf V1 dans le corrigé de la question suivante.

Nous voulons maintenant modifier la fonction précédente en une fonction `int wait_till_pid (pid_t pid, int sec)`.

Le paramètre *sec* indique le temps maximum en secondes durant lequel le processus appelant attendra la terminaison de son fils *pid*. La fonction retourne 0 si le délai expire avant la terminaison de ce fils, sinon elle retourne *pid* comme dans la question précédente.

Question 8. Toujours sans utiliser *waitpid*, mais à plutôt l'aide de signaux, implanter ce comportement. Attention à bien désarmer une alarme temporisée si le fils meurt avant le temps imparti. Indice : On pourra se placer en attente des deux signaux : SIGCHLD (un fils est mort) et SIGALRM (temporisateur).

pseudowait.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>


// V1
pid_t wait_till_pid(pid_t pid) {
      while (true) {
            pid_t p = wait(nullptr);
            if (p==-1 || p == pid) {
                  return p;
            }
      }
```

```cpp
}

// V2 : signal
pid_t wait_till_pid(pid_t pid, int sec) {
	static bool timeout ;
	timeout = false;
	signal(SIGALRM,[](int){ std::cout << "received SIGALRM" << std::endl; timeout =
	    true;});
	signal(SIGCHLD,[](int){ std::cout << "received SIGCHLD" << std::endl;});
	alarm(sec);

	sigset_t set;
	sigfillset(&set);
	sigdelset(&set,SIGALRM);
	sigdelset(&set,SIGCHLD);

	while (true) {
		std::cout << "waiting..." << std::endl;
		sigsuspend(&set);
		if (timeout) {
			std::cout << "Alarm wins" << std::endl;
			return -1;
		} else {
			pid_t p = wait(nullptr);
			std::cout << "wait answered " << p << std::endl;
			if (p==pid) {
				alarm(0);
			}
			if (p==-1 || p == pid) {
				return p;
			}
		}
	}
}


int main() {
	pid_t pid =fork();
	if (pid==0) {
		// si on met plus que le timeout il doit mourir
		sleep(5);
	} else {
		signal(SIGINT,[](int){});
		pid_t res = wait_till_pid(pid,3);
		std::cout << "wait gave :" << res << std::endl;
	}
}
```

TD 7 : Processus, fork, signaux

Objectifs pédagogiques :

- **Communications IPC : shm, sem**
- **Tube et Tube nommé**

Introduction

Dans ce TD, nous allons étudier l'API proposée par POSIX pour la communication entre processus. Il faudra donc un système POSIX (en particulier pas windows) pour compiler et exécuter nos programmes.

Le standard POSIX définit une API en C à bas niveau pour l'interaction entre les processus incluant des tubes, des segments de mémoire partagée, et des sémaphores.

1 Tube anonyme

Question 1. A l'aide de la primitive *pipe*, écrivez un programme P qui crée deux fils $F1$ et $F2$, tous les processus doivent afficher le pid des trois processus avant de se terminer proprement.

> `man 2 pipe, man 7 pipe` (extraits)
>
> int pipe(int pipefd[2]);
>
> *pipe*() creates a pipe, a unidirectional data channel that can be used for interprocess communication.
>
> The array pipefd is used to return two file descriptors referring to the ends of the pipe.
>
> pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.
>
> Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

> Donc pour P et $F2$, facile de connaitre les autres pid.
>
> Le père P n'a qu'à stocker les pid de ses fils (obtenus par fork).
>
> Le deuxième fils $F2$ peut hériter de son père le pid de $F1$ et utiliser e.g. *getppid*() pour trouver P.
>
> Le premier fils $F1$, on a un souci, comment lui communiquer le pid de son petit frère ? On ne dispose pas d'API pour le retrouver (pas de *getsiblingpid*...).
>
> Donc on utilise "pipe".
>
> Voir aussi "man 7 pipe".
>
> Une fois qu'on a fait "pipe", on obtient deux descripteurs de fichiers (des int), avec lesquels on va utiliser l'API standard sur fd : read(fd, addresse, taille), write(fd, addresse, taille), et close(fd).
>
> Pour être propre, on ferme toutes les extrémités que l'on n'utilise pas.

troisProc.cpp

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <sys/wait.h>
4  #include <iostream>
5
6  int main3 () {
7        pid_t P,F1,F2;
8
9        P = getpid();
10
```

```
      int pdesc[2];
      if (pipe(pdesc) != 0) {
            perror("pipe error");
            return 1;
      }

      F1 = fork();
      if (F1 < 0) {
            perror("fork error");
            return 1;
      } else if (F1 == 0){
            F1 = getpid();
            // F1
            close(pdesc[1]);
            if (read(pdesc[0],&F2,sizeof(pid_t)) == sizeof(pid_t)) {
                  std::cout << "Je suis :" << getpid() << " P=" << P << " F1=" << F1 <<
                      " F2=" << F2 << std::endl;
            } else {
                  perror("erreur read");
                  return 1;
            }
            close(pdesc[0]);
            return 0;
      }

      F2 = fork();
      if (F2 < 0) {
            perror("fork error");
            return 1;
      } else if (F2 == 0){
            // F2
            F2 = getpid();
            close(pdesc[0]);
            if (write(pdesc[1],&F2,sizeof(pid_t)) == sizeof(pid_t)) {
                  std::cout << "Je suis :" << getpid() << " P=" << P << " F1=" << F1 <<
                      " F2=" << F2 << std::endl;
            } else {
                  perror("erreur write");
                  return 1;
            }
            close(pdesc[1]);
            return 0;
      }

      close(pdesc[0]);
      close(pdesc[1]);

      std::cout << "Je suis :" << getpid() << " P=" << P << " F1=" << F1 << " F2=" << F2
           << std::endl;

      wait(0);
      wait(0);

      return 0;
}
```

2 Tube nommé

Pour cet exercice on a besoin des primitives suivantes (cf. man en section 2)

- mkfifo : crée une fifo (tube nommé)
- open/close : ouvrir/obtenir un filedescriptor
- read/write : avec un filedescriptor, un pointeur de char, un entier pour la taille.
- unlink : efface uen entrée du système de fichier

Question 1. Expliquez comment lire et écrire une string de longueur et contenu *arbitraire* dans un flux identifié par un filedescriptor (un entier fd), en utilisant les primitives POSIX *read* et *write*. On donnera le code de `int write(const std::string & s, int fd)` et `int read(std::string & s, int fd)` qui rendent la longueur de la chaîne écrite/lue, ou une valeur négative en cas d'erreur.

Attention si la chaine contient des '\0' les versions ici se moquent du contenu ce sont des data arbitraires, comme ce que stockent les std::string du c++.

On accède avec data() au contenu de la string (sans \0 terminal, mais possiblement avec des char 0 au milieu).

On construit la string en passant un pointeur et une taille, pas juste un pointeur de char, sinon il serait interprété comme une string du C et coupée sur un éventuel \0.

Donc vu qu'on est en flux, il est nécessaire de passer la taille puis le contenu en deux écritures et symétriquement en lecture.

Transmettre des chaînes de caractère = souci pour la taille de ce qui passe. On va écrire la taille de la chaîne (un size_t) puis son contenu au format brut tableau de char.

writer.cpp

```
1  int write (const std::string & s, int fd) {
2        size_t sz = s.length();
3        if (write (fd, &sz, sizeof(sz)) != sizeof(sz)) {
4              perror("write sz");
5              return -1;
6        }
7
8        if (write (fd, s.data(), sz) != sz) {
9              perror("write data");
10             return -1;
11       }
12       std::cout << "written string len=" << sz << " :" << s << std::endl;
13       return sz;
14 }
```

En lecture, obtenir 0 comme taille lue indique qu'on est au bout du flux/fichier. Sur un tube nommé ce n'est possible que si il n'y a aucun écrivain (aucun processus n'a de fd ouvert en écriture sur le tube). S'il reste des écrivains la lecture est bloquante.

reader.cpp

```
1  int read (std::string & s, int fd) {
2        size_t sz = 0;
3        int rd = read (fd, &sz, sizeof(sz));
4        if (rd ==0) {
5              // Ceci arrivera ssi il n'y a plus d'ecrivain => introduit la notion de EOF
6              return 0;
7        } else if ( rd != sizeof(sz)) {
8              perror("read error");
9              return -1;
```

```
10        }
11        char buff [sz];
12        if (read (fd, buff, sz) != sz) {
13              perror("read error");
14              return -1;
15        }
16        /* ctor a deux arguments, pour éviter de couper la chaine s'il y a des \0 dedans
              */
17        s = std::string(buff, sz);
18        return sz;
19  }
```

NB : Il n'y a pas de boucles sur read ni write dans le corrigé, donc les grosses écritures vont mal se passer. Voir dans le cours la boucle proposée pour "fullread" et quelque chose de similaire pour write.

Question 2. Ecrivez un programme *writer* qui prend un chemin en argument, y crée un tube nommé, puis attend que l'utilisateur saisisse du texte et le recopie dans le tube. Sur controle-C (SIGINT) le programme se ferme proprement sans laisser de traces.

Lecture ligne à ligne de l'entrée en C++.
On pourra utiliser la forme suivante, qui capture l'entrée ligne à ligne dans une `std::string`

```
while (true) { std::string line; std::cin >> line; /* agir sur line */ }
```

Les soucis à traiter :

- mkfifo + tester que ça marche + droits pour l'usager $S\_IRUSR \| S\_IWUSR$.
- open => O_RDONLY ou O_WRONLY selon l'extrémité
- On accroche un "unlink"+exit au signal Ctrl-C

writer.cpp

```
1
2   void handleCtrlC (int sig) {
3         unlink(path);
4         exit(0);
5   }
6
7   int main (int argc, char ** argv) {
8         if (argc < 2) {
9               std::cerr << "Provide a name for the pipe." << std::endl;
10        }
11
12        if ( mkfifo(argv[1],S_IRUSR|S_IWUSR) != 0) {
13              perror("mkfifo problem");
14              return 1;
15        }
16        path = argv[1];
17
18        int fd = open(argv[1],O_WRONLY);
19        if (fd < 0) {
20              perror("open problem");
21              return 1;
22        }
23
24        signal(SIGINT, handleCtrlC);
```

```
25
26
27       while (true) {
28             std::string s;
29             std::cin >> s;
30             if (write(s,fd) <0) {
31                   break;
32             }
33       }
34
35       return 0;
36 }
```

Question 3. Ecrivez un programme *reader* qui prend un chemin en argument correspondant à un tube, puis lit le texte envoyé par writer dans le tube. Le programme se termine quand il n'y a plus d'écrivain sur le tube.

Le lecteur est plus simple que writer.

reader.cpp

```
1
2
3  int main (int argc, char ** argv) {
4        if (argc < 2) {
5              std::cerr << "Provide a name for the pipe." << std::endl;
6        }
7
8        int fd = open(argv[1],O_RDONLY);
9        if (fd < 0) {
10             perror("open problem");
11             return 1;
12       }
13
14       while (true) {
15             std::string s;
16             int rd = read(s,fd);
17             if (rd == 0) {
18                   break;
19             }
20             std::cout << s << std::endl;
21       }
22
23       return 0;
24 }
```

Question 4. Que se passe-t-il si on a plusieurs lecteurs ou plusieurs écrivains ?

Plusieurs écrivains = aucun problème, les écritures de petite taille (BUF_SIZE vaut 4096 sur ma machine) sont atomiques. MAIS avec le protocole tel qu'il est fait, on n'écrit pas atomiquement (deux appels à write). Pour que ça se passe bien, une seule écriture est nécessaire. On pourrait assez facilement le faire en préparant un buffer qui contienne la taille ET la string.

Plusieurs lecteurs, si on a forcé une seule écriture par message, ça devrait bien se passer, mais attention, un seul lecteur peut voir les messages à la fois la lecture est destructrice.

De plus on ne peut pas faire un seul read, car il faut lire la taille avant de lire le contenu. Donc on devrait passer vers un modèle où les messages sont de taille fixe pour permettre le mode multi-lecteur, tel quel on est obligé de faire deux lecture.

Morale : ça marchotte, mais ce n'est pas terrible, si on a des paquets de taille fixe, c'est plus envisageable d'avoir plusieurs lecteurs.

3 Sémaphore

Question 1. On considère une application constituée de deux processus P1 et P2. A l'aide de l'API sémaphore, réaliser une alternance, où le processus P1 affiche "Ping", et le processus P2 "Pong".

Donc il faut deux semaphores, avec un seul, on va s'auto-libérer en boucle.

P1 libère P2, qui libère P1 ...

On les laisse réfléchir, et on invite la discussion sur "un seul sémaphore, A fait P puis V, B fait V puis P". Non, ça ne marche pas.

Donc bien 2 sémaphores.

pingpong.cpp

```cpp
#include <fcntl.h> /* For O_* constants */
#include <sys/stat.h> /* For mode constants */
#include <semaphore.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>


int main() {

	// create mutex initialisé 0
	sem_t * smutex1 = sem_open("/monsem1", O_CREAT |O_EXCL | O_RDWR , 0666, 0);
	if (smutex1==SEM_FAILED) {
		perror("sem create");
		exit(1);
	}
	// create mutex initialisé 0
	sem_t * smutex2 = sem_open("/monsem2", O_CREAT |O_EXCL | O_RDWR , 0666, 0);
	if (smutex2==SEM_FAILED) {
		perror("sem create");
		exit(1);
	}

	pid_t f = fork();
	if (f==0) {
		// fils
		for (int i=0; i < 10; i++) {
			sem_wait(smutex1);
			std::cout << "Ping" << std::endl;
			sem_post(smutex2);
		}
		sem_close(smutex1);
		sem_close(smutex2);
	} else {
		// pere
```

```
38              for (int i=0; i < 10; i++) {
39                    sem_post(smutex1);
40                    sem_wait(smutex2);
41                    std::cout << "Pong" << std::endl;
42              }
43              sem_close(smutex1);
44              sem_close(smutex2);
45
46              // on nettoie
47              sem_unlink("/monsem1");
48              sem_unlink("/monsem2");
49              wait(nullptr);
50        }
51        return 0;
52  }
```

On va traiter ici directement avec l'API pour créer des sémaphores (nommés), sans déployer un segment partagé. Cela signifie que l'on crée avec *sem_open* et un nom de fichier (qui DOIT commencer par / puis ne pas contenir de slash, et n'être pas trop long), comme on ouvrirait une inode dans le filesystem. On crée un objet dans le système, qu'il faut nettoyer à l'aide de unlink.

Question 2. On considère à présent N processus qui doivent s'alterner (circulairement). Combien de sémaphores faut-il introduire ? Proposez une réalisation de ce comportement.

Donc il faut N semaphores, et cadrer tout ça avec des boucles. Pour plus d'homogénéité, je laisse les N fils jouer entre eux, la version a deux processus c'était père vs fils.

On pourrait ajouter un numéro Ping1, Ping2... pour mieux constater l'alternance.

On crée un nouveau nom unique par semaphore.

pingNpong.cpp

```
1   #include <fcntl.h> /* For O_* constants */
2   #include <sys/stat.h> /* For mode constants */
3   #include <semaphore.h>
4   #include <sys/wait.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdlib.h>
8   #include <iostream>
9
10
11  #define N 4
12
13  int main() {
14
15        sem_t * mutex [N];
16
17        for (int i=0; i < N ; i++) {
18              // create mutex initialisé 0 sauf le premier
19              int semval = 0;
20              if (i==0) {
21                    semval = 1;
22              }
23              std::string semname = "/monsem"+std::to_string(i);
24              mutex[i] = sem_open(semname.c_str(), O_CREAT |O_EXCL | O_RDWR , 0666, semval
                    );
25              if (mutex[i]==SEM_FAILED) {
```

```
                perror("sem create");
                exit(1);
            }
        }

        for (int i=0; i < N ; i++) {
            pid_t f = fork();
            if (f==0) {
                // fils
                for (int round=0; round < 10; round++) {
                    sem_wait(mutex[i]);
                    std::cout << "Ping" << i << std::endl;
                    sem_post(mutex[(i+1)%N]);
                }
                return 0;
            }
        }

        for (int i=0; i < N ; i++) {
            wait(nullptr);
        }

        // on nettoie
        for (int i=0; i < N ; i++) {
            sem_unlink( ("/monsem"+std::to_string(i)).c_str() );
        }
        return 0;
}
```

Remarque importante : Si on réfléchit en termes de l'API thread, il nous faut N condition, donc aussi N mutex pour obtenir ce comportement. Le point important est que A fait P et UN AUTRE processus B fait le V, ce qui n'est pas possible avec un mutex. Donc on a des sémaphores dont le compteur oscille entre 0 et 1, mais ce ne sont pas des équivalents à des mutex ici.

4 Mémoire partagée

On considère un programme qui manipule une pile partagée entre plusieurs processus.
On fournit la base de code suivante :

Stack.h

```
#pragma once

#include <cstring> // size_t,memset

namespace pr {

#define STACKSIZE 100

template<typename T>
class Stack {
      T tab [STACKSIZE];
      size_t sz;
public :
      Stack () : sz(0) { memset(tab,0,sizeof tab) ;}

      T pop () {
```

```
17            // bloquer si vide
18            T toret = tab[--sz];
19            return toret;
20      }
21
22      void push(T elt) {
23            //bloquer si plein
24            tab[sz++] = elt;
25      }
26 };
27
28 }
```

prodcons.cpp

```
1  #include "Stack.h"
2  #include <iostream>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  #include <vector>
6
7
8  using namespace std;
9  using namespace pr;
10
11 void producteur (Stack<char> * stack) {
12       char c ;
13       while (cin.get(c)) {
14             stack->push(c);
15       }
16 }
17
18 void consomateur (Stack<char> * stack) {
19       while (true) {
20             char c = stack->pop();
21             cout << c << flush ;
22       }
23 }
24
25 int main () {
26       Stack<char> * s = new Stack<char>();
27
28       pid_t pp = fork();
29       if (pp==0) {
30             producteur(s);
31             return 0;
32       }
33
34       pid_t pc = fork();
35       if (pc==0) {
36             consomateur(s);
37             return 0;
38       }
39
40       wait(0);
41       wait(0);
42
43       delete s;
44       return 0;
45 }
```

Question 1. En se limitant aux seuls semaphores, modifier le Stack pour que : pop bloque si vide, push bloque si plein, et que les données partagées *sz* et *tab* soit protégées des accès concurrents. On suppose que le Stack est alloué dans un segment de mémoire partagée, et qu'il sera partagé entre processus.

On les laisse réfléchir, on a ici une vraie question du type que l'on poserait volontiers en examen.

Les précisions sur le contexte "mémoire partagée" et "entre processus" affecte la façon de créer et initialiser les semaphores. Dans ce contexte, on sait qu'on peut jouer la carte OO, et loger des semaphores dans la classe elle même. On a ici un modèle de classe C++ dont les instances sont multi-process safe.

On se contente ici de *sem_init* auquel on passe 1 en deuxième arg pour préciser que c'est des processus. Réciproquement, le destructeur va utiliser *sem_destroy* pour nettoyer ces objets.

On note au passage les différences avec l'exo précédent: on y a utilisé sem_open, sem_unlink. Nos sémaphores sont ici "anonymes" au sens où ils n'ont pas d'entrée spécifique (inode) dans le système de fichier.

Donc on a besoin de trois semaphores :

- un sem qui joue le role de mutex, il protège sz et tab. Initialisé à 1, les accès aux attibuts sont des sections critiques, chacun fait un P avant et un V après avoir lu/écrit *tab* ou *sz*.
- deux sem qui jouent le rôle de mécanisme de notification et de contrôle anti débordement.
  - sempop : initialisé à 0, on P dessus avant de pop, on V dessus après un push. Donc au départ 0, c'est vide, la valeur du compteur reflète size.
  - sempush : initialisé à MAX, on P dessus avant de push, on V dessus après un pop. Donc au départ MAX, c'est vide on peut fair MAX push. La valeur du compteur reflète $MAX - size$.

Stack.h

```
#ifndef __STACK_HH__
#define __STACK_HH__

#include <cstring> // size_t
#include <semaphore.h>
#include <iostream>

namespace pr {

#define STACKSIZE 100

// T = un type numerique (contrat de valarray)
template<typename T>
class Stack {
      T tab [STACKSIZE];
      size_t sz;
      sem_t mutex;
      sem_t sempop;
      sem_t sempush;
public :
      Stack () : sz(0) {
            sem_init(&mutex,1,1);
            sem_init(&sempop,1,0);
            sem_init(&sempush,1,STACKSIZE);
            memset(tab,0,sizeof tab) ;
      }

      ~Stack() {
            sem_destroy(&mutex);
            sem_destroy(&sempop);
            sem_destroy(&sempush);
      }

```

```
34        T pop () {
35              sem_wait(&sempop);
36              sem_wait(&mutex);
37              T toret = tab[--sz];
38              sem_post(&mutex);
39              sem_post(&sempush);
40              return toret;
41        }
42
43        void push(T elt) {
44              sem_wait(&sempush);
45              sem_wait(&mutex);
46              tab[sz++] = elt;
47              sem_post(&mutex);
48              sem_post(&sempop);
49        }
50  };
51
52  }
53
54
55  #endif
```

Attention à bien initialiser dans le constructeur, et aussi à les libérer à la destruction. On veut s'en servir entre *processus* (et non entre thread) du coup on passe 1 en deuxième argument au *sem_init*.

Question 2. Les processus consommateur et producteur ne peuvent pas communiquer à travers le stack actuellement, car il n'est pas dans une zone partagée. Proposez une correction du code pour que le Stack soit dans un segment de mémoire partagée anonyme. Comparez le code à une version dans un segment nommé.

NB : In place new. Le C++ permet de construire un objet avec new, mais dans un endroit arbitraire, supposé déjà alloué à une taille suffisante. Pour une classe *MyClass*, on peut écrire : `void * zone = /*une adresse vers une zone pré-allouée */ ;`
`MyClass * mc = new (zone) MyClass(arg1,arg2);`

Avec cette syntaxe, le constructeur de la classe est invoqué, mais il n'y pas d'allocation. On peut s'en servir pour initialiser des objets C++ dans un segment de mémoire partagée. Attention dans ce cas à la destruction : il faut invoquer le destructeur explicitement `mc->~MyClass();`, mais sans faire un `delete` qui ferait aussi une désallocation.

Donc deux versions, la version anonyme est plus facile, en gros on dirait un malloc, mais on ne peut s'en servir qu'au sein d'une arborescence de processus. L'astuce donc on ajoute "MAP_ANONYMOUS" aux flags de *mmap*, et on passe -1 pour le filedescriptor.

La version nommée, on commence par "shm_open" (qui crée une inode pour ce shm), puis "ftruncate" pour lui allouer une taille (l'inode par défaut, c'est vide, comme si on "touch" un fichier). Ensuite on passe le filedescriptor obtenu à l'invocation à mmap. Cette invocation monte les pages en mémoire de l'inode, de façon à garantir que les écritures sont vues immédiatement par les lecteurs, l'ensemble est *memory mapped*.

Et on ajoute un "shm_unlink" à la fin du code pour nettoyer l'inode créée par le *shm_open*.

prodcons.cpp

```
1  #include <iostream>
```

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include "Stack_cor.h"
#include <vector>


using namespace std;
using namespace pr;

void producteur (Stack<char> * stack) {
      char c ;
      while (cin >> c) {
            stack->push(c);
      }
}

void consomateur (Stack<char> * stack) {
      while (true) {
            char c = stack->pop();
            cout << c << std::flush ;
      }
}

std::vector<pid_t> tokill;

void killem (int) {
      for (auto p : tokill) {
            kill(p,SIGINT);
      }
}

int main () {
      size_t shmsize = sizeof(Stack<char>);
      std::cout << "Allocating segment size "<<shmsize << std::endl;

      int fd;
      void * addr;

      bool useAnonymous = false;
      if (useAnonymous) {
            addr = mmap(nullptr, shmsize, PROT_READ | PROT_WRITE, MAP_SHARED |
                MAP_ANONYMOUS, -1, 0);
            if (addr == MAP_FAILED) {
                  perror("mmap anonymous");
                  exit(1);
            }
      } else {
            fd = shm_open("/myshm",O_CREAT|O_EXCL|O_RDWR,0666);
            if (fd < 0) {
                  perror("shm_open");
                  return 1;
            }
            if (ftruncate(fd,shmsize) != 0) {
                  perror("ftruncate");
                  return 1;
            }
```

```
60              addr = mmap(nullptr, shmsize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
61              if (addr == MAP_FAILED) {
62                    perror("mmap anonymous");
63                    exit(1);
64              }
65        }
66
67        // In place new : faire le new à l'addresse fournie, supposée assez grande.
68        Stack<char> * s = new (addr) Stack<char>();
69
70        pid_t pp = fork();
71        if (pp==0) {
72              producteur(s);
73              return 0;
74        }
75
76        pid_t pc = fork();
77        if (pc==0) {
78              consomateur(s);
79              return 0;
80        } else {
81              tokill.push_back(pc);
82        }
83
84        signal(SIGINT, killem);
85
86        wait(0);
87        wait(0);
88
89        // invoque le destructeur, mais pas delete
90        s->~Stack();
91        if (munmap(addr,shmsize) != 0) {
92              perror("munmap");
93              exit(1);
94        }
95        if (! useAnonymous) {
96              if (shm_unlink("/myshm") != 0) {
97                    perror("sem unlink");
98              }
99        }
100 }
```

On fait effectivement un "in place new" pour le Stack. Attention le delete à la fin saute par contre, en faveur d'une invocation explicite au dtor.

Question 3. Le programme ne se termine pas actuellement, ajoutez une gestion de signal pour que le main interrompe le(s) consommateur(s) si on lui envoie un Ctrl-C (SIGINT).

Ok, donc on revient un peu sur les signaux. Je stocke les pid des consommateurs (qui vont finir bloqués), le père quand il se prend Ctrl-C transmet le signal à ses fils, qui en mourront (comportement par défaut).

J'ai fait une var globale (vecteur des pid des fils) pour éviter les soucis de conversion d'une lambda avec capture vers un pointeur de fonction "void () (int sig)".

Seuls les consommateurs sont bloqués; le code est prévu pour facilement généraliser à plusieurs consommateurs, i.e. on utilise un vector pour stocker un seul pid.

Question 4. Généraliser le programme à plusieurs producteurs et/ou consommateurs.

> Rien de bien méchant, mais pas de corrigé ici désolé.
>
> On remarque simplement que les synchros sur le stack et le comportement global de notre programme permet effectivement de supporter un nombre arbitraire de consommateurs ou producteurs.

TME 7 : Mémoire Partagée, Sémaphores

Objectifs pédagogiques :

- **shared memory**
- **semaphores**

0.1 Git pour le TME

Les fichiers fournis sont a récupérer suivant la même procédure qu'au TME précédent sur https://pscr-gitlab.lip6.fr/ythierry/TME7.

Vous ferez un fork de ce dépot, travaillerez sur votre copie, et effectuerez un push de vos résultats sur ce dépôt pour soumettre votre travail.

Attention à activer les flags $-pthread$ mais aussi $-lrt$ au link pour avoir les sémaphores.

1 Fork, exec, pipe

Question 1. On souhaite simuler le comportement d'un shell pour chaîner deux commandes : la sortie de la première commande doit alimenter l'entrée de la deuxième commande.

Ecrire un programme "pipe" qui a ce comportement. On utilisera :

- fork et exec, on suggère d'utiliser la version execv,
- pipe pour créer un tube,
- dup2(fd1,fd2) pour remplacer fd2 par fd1, de façon à substituer aux entrées sorties "normales" les extrémités du pipe

On pourra tester avec par exemple (le backslash protège l'interprétation du pipe par le shell) :

```
pipe /bin/cat pipe.cpp \| /bin/wc -l
```

NB 1 : Il faut itérer sur les arguments $argc, argv$ passés au main pour chercher le symbole pipe '|', et construire deux tableaux d'arguments (des $const\ char*$) terminés par $nullptr$ pour invoquer $execv$.

NB 2 : le cours comporte un exemple très proche, en page 57 du cours 7.

Donc une boucle un peu moche/bancale, on suppose que les deux sous commandes ne peuvent pas avoir plus de $argc$ éléments pour dimensionner, du coup on fait un memset à 0 pour avoir un nullptr à la fin de la boucle dans chacun des tableaux.

Bref, à l'issue de ça on a collecté les deux sous paquets d'arguments dans deux tableaux du C, qu'on peut passer correctement à l'API de $execv$. Le execv rappelons le ne va jamais revenir, soit on se prend un erreur sur execv, soit le contexte entier du processus est absorbé, pas de retour comme si c'était un call de fonction, on écrase le segment de code la pile tout ça, on place le curseur de programme en 0 et on tourne sur cette nouvelle image.

La suite c'est dans le cours, en gros entre le fork et le exec on substitue les flux sous les pieds du processus. On note que le père et le fils après un fork partagent les descripteurs de fichiers ouverts, c'est à dire que un "int fd" que le père a "open" est utilisable par le fils pour faire read/write/close. En particulier stdin/out/err sont partagés, on le constate bien dans nos petits programmes avec des fork.

Donc en remplaçant le sens de 0 (=STDIN) ou le sens de 1 (=STDOUT) entre le fork et l'exec, on contrôle où les flux d'entrées / sorties du fils vont aller.

pipe.cpp

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <cstring>
#include <iostream>


void myexec (int argc, char ** args) {
      std::cerr << "args :" ;
      for (int i =0; i < argc ; i++) {
            if (args[i] != nullptr) {
                  std::cerr << args[i] << " ";
            } else {
                  break;
            }
      }
      std::cerr << std::endl;

      if (execv (args[0], args) == -1) {
            perror ("exec");
            exit (3);
      }
}

int main (int argc, char ** argv) {

      // On partitionne les args en deux jeux d'arguments
      char * args1[argc];
      char * args2[argc];

      memset(args1,0,argc*sizeof(char*));
      memset(args2,0,argc*sizeof(char*));

      int arg = 1;
      for ( ; arg < argc ; arg++) {
            if (! strcmp(argv[arg],"|")) {
                  arg++;
                  break;
            } else {
                  args1[arg-1] = argv[arg];
            }
      }
      for (int i=0; arg < argc ; i++,arg++) {
            args2[i] = argv[arg];
      }

      // OK, args1 = paramètre commande 1, args2 = deuxième commande.

      int tubeDesc[2];
      pid_t pid_fils;
      if (pipe (tubeDesc) == -1) {
            perror ("pipe");
            exit (1);
      }
      if ( (pid_fils = fork ( )) == -1 ){
            perror ("fork");
            exit (2);
      }
      if (pid_fils == 0) { /* fils 1 */

```

```
63              dup2(tubeDesc[1],STDOUT_FILENO);
64              close (tubeDesc[1]);
65              close (tubeDesc[0]);
66
67              myexec(argc,args1);
68              // on ne revient pas du exec
69      }
70
71      // pere donc ici
72      if ( (pid_fils = fork ( )) == -1 ){
73              perror ("fork");
74              exit (2);
75      }
76      if (pid_fils == 0) { /* fils 2 */
77              dup2(tubeDesc[0],STDIN_FILENO);
78              close (tubeDesc[0]);
79              close (tubeDesc[1]);
80
81              myexec(argc,args2);
82      }
83
84      // important
85      close (tubeDesc[0]);
86      close (tubeDesc[1]);
87
88      wait(0);
89      wait(0);
90      return 0;
91  }
```

2 Sémaphore, Mémoire partagée

Reprenez l'exercice du TD 7 sur le stack partagé entre producteurs et consommateurs.

Question 1. Ecrivez progressivement une version avec N producteurs et M consommateurs, qui utilise un segment de mémoire partagée nommé, et se termine proprement sur Ctrl-C.

Question 2. Assurez vous de bien avoir libéré les ressources, on utilisera $O\_CREAT|O\_EXCL$ pour faire les *open* et on vérifiera que l'on peut exécuter deux fois le programme d'affilée.

> Voir corrigé du TD.

3 Messagerie par mémoire partagée

Résoudre l'exercice "Une messagerie instantanée en mémoire partagée" https://www-master.ufr-info-p6.jussieu.fr/2017/Une-messagerie-instantanee-en.

Le fichier "chat_common.h" est dans le dépot git.

> Solution en C brute de l'an dernier.

common.h

```h
#ifndef H_CHAT_COMMON
#define H_CHAT_COMMON

#define _XOPEN_SOURCE 700
#define _REENTRANT
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <ctype.h>

#include <sys/ipc.h>
#include <sys/mman.h>
#include <sys/stat.h> /* Pour les constantes des modes */
#include <fcntl.h> /* Pour les constantes O_* */
#include <semaphore.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <errno.h>

#include <string.h>

#define MAX_MESS 50
#define MAX_USERS 10
#define TAILLE_MESS 10

struct message {
  long type;
  char content[TAILLE_MESS];
};

struct myshm {
  int read; /* nombre de messages retransmis par le serveur */
  int write; /* nombre de messages non encore retransmis par le serveur */
  int nb; /* nombre total de messages emis */
  sem_t sem;
  struct message messages[MAX_MESS];
};

char *getName(char *name);

#endif
```

serveur.c

```c
#include <chat_common.h>

int shouldexit = 0;

void exithandler(int sig){
  shouldexit = sig; /* i.e !=0 */
}

struct user {
  char *name;
  struct myshm *shm;
```

```
12 } *users[MAX_USERS];
13
14 int main(int argc, char *argv[]){
15   char *shmname;
16   int shm_id, i;
17   struct myshm *shm_pere;
18   struct sigaction action;
19
20   if (argc <= 1) {
21     fprintf(stderr, "Usage: %s id_server\n", argv[0]);
22     exit(EXIT_FAILURE);
23   }
24
25   /* Met un handler pour arrêter le programme de façon clean avec Ctrl-C */
26   action.sa_handler = exithandler;
27   action.sa_flags = 0;
28   sigemptyset(&action.sa_mask);
29   sigaction(SIGINT, &action, 0);
30
31   /* On crée l'identifiant "/<server_id>_shm:0" */
32   shmname = (argv[1]);
33
34   /* Crée le segment en lecture écriture */
35   if ((shm_id = shm_open(shmname, O_RDWR | O_CREAT, 0666)) == -1) {
36     perror("shm_open shm_pere");
37     exit(EXIT_FAILURE);
38   }
39
40   /* Allouer au segment une taille de NB_MESS messages */
41   if(ftruncate(shm_id, sizeof(struct myshm)) == -1) {
42     perror("ftruncate shm_pere");
43     exit(EXIT_FAILURE);
44   }
45
46   /* Mappe le segment en read-write partagé */
47   if((shm_pere = mmap(NULL, sizeof(struct myshm), PROT_READ | PROT_WRITE, MAP_SHARED,
       shm_id, 0)) == MAP_FAILED){
48     perror("mmap shm_pere");
49     exit(EXIT_FAILURE);
50   }
51   shm_pere->read = 0;
52   shm_pere->write = 0;
53   shm_pere->nb = 0;
54
55   /* Initialisation du sémaphore (mutex) */
56   if(sem_init(&(shm_pere->sem), 1, 1) == -1){
57     perror("sem_init shm_pere");
58     exit(EXIT_FAILURE);
59   }
60
61   /* Initialisation du tableau de structure users à NULL */
62   for(i=0; i<MAX_USERS; i++){
63     users[i] = NULL;
64   }
65
66   while(!shouldexit){
67     sem_wait(&(shm_pere->sem));
68     /* Si il y a eu au moins une écriture */
69     if(shm_pere->read != shm_pere->write || shm_pere->nb == MAX_MESS){
```

```
70      /* On récupère le premier message et on incrémente le nombre lu */
71      struct message message = shm_pere->messages[shm_pere->read];
72
73      switch(message.type){
74      case 0:{
75        /* Connexion */
76        char *username;
77        int shm_id_user;
78        i=0;
79
80        /* Récupère la première case user vide */
81        while(i<MAX_USERS && users[i] != NULL) i++;
82        if(i == MAX_USERS){perror("impossible d'ajouter l'user"); exit(EXIT_FAILURE);}
83
84        users[i] = malloc(sizeof(struct user));
85        if(users[i] == NULL){perror("impossible d'ajouter l'user"); exit(EXIT_FAILURE);}
86
87        /* Copie le nom du fils */
88        users[i]->name = malloc((strlen(message.content) + 1) * sizeof(char));
89        strcpy(users[i]->name, message.content);
90        users[i]->name[strlen(message.content)] = '\0';
91
92        printf("Connexion de %s\n", users[i]->name);
93
94        /* Récupère le segment partagé de l'user et le mappe */
95        username = message.content;
96        if((shm_id_user = shm_open(username, O_RDWR | O_CREAT, 0666)) == -1) {
97          perror("shm_open shm_fils");
98          exit(EXIT_FAILURE);
99        }
100       if((users[i]->shm = mmap(NULL, sizeof(struct myshm), PROT_READ | PROT_WRITE,
              MAP_SHARED, shm_id_user, 0)) == MAP_FAILED){
101         perror("mmap shm_fils");
102         exit(EXIT_FAILURE);
103       }
104
105       break;
106     }
107     case 1:{
108       /* Envoi de message */
109       int temp = 0;
110       printf("Réception message serveur : %s\n", message.content);
111       for(i=0; i<MAX_USERS; i++){
112         if(users[i] != NULL){
113           struct message msg;
114           struct myshm *shm = users[i]->shm;
115           temp++;
116
117           /* On copie le message dans la file du fils */
118           msg.type = 1;
119           strcpy(msg.content, message.content);
120
121           sem_wait(&(shm->sem));
122
123           while(shm->nb == MAX_MESS){
124             sem_post(&(shm->sem));
125             sleep(1);
126             sem_wait(&(shm->sem));
127           }
```

```c

        /* TODO : vérifier que c'est pas plein */

        printf("\t\tEnvoi à %s\n", users[i]->name);


        shm->messages[shm->write] = msg;
        shm->write = (shm->write + 1) % MAX_MESS;
        shm->nb++;

        sem_post(&(shm->sem));
       }
      }

      if(temp == 0){
       printf("Nobody found!\n");
      }
      break;
     }
     case 2:{
      printf("Déconnexion de %s\n", message.content);
      /* Déconnexion */
      i=0;

      /* On récupère le user correspondant à l'user qui se déconnecte */
      while(i<MAX_USERS && (users[i] == NULL || strcmp(users[i]->name, message.content)
          != 0)) i++;
      if(i == MAX_USERS){
       perror("Something went wrong! L'user n'existe pas!");
       exit(EXIT_FAILURE);
      }

      /* On libère l'espace mémoire */
      free(users[i]->name);
      munmap(users[i]->shm, sizeof(struct myshm));
      free(users[i]);
      users[i] = NULL;

      break;
     }
     }
     shm_pere->read = (shm_pere->read + 1) % MAX_MESS;
     shm_pere->nb--;
    }
    sem_post(&(shm_pere->sem));
  }

  sem_close(&(shm_pere->sem));

  munmap(shm_pere, sizeof(struct myshm));

  shm_unlink(shmname);

  return EXIT_SUCCESS;
}
```

client.c

```c
#include <chat_common.h>

int shouldexit = 0;

struct myshm *shm_client, *shm_pere;

void *reader(void* arg){
  while(!shouldexit){
    sem_wait(&(shm_client->sem));

    /* Si il y a eu au moins une écriture */
    if(shm_client->read != shm_client->write || shm_client->nb == MAX_MESS){
      /* On récupère le premier message et on incrémente le nombre lu */
      struct message message = shm_client->messages[shm_client->read];
      shm_client->read = (shm_client->read + 1) % MAX_MESS;
      printf("%s", message.content);
      shm_client->nb--;
    }
    sem_post(&(shm_client->sem));
  }
  pthread_exit(NULL);
  return NULL;
}

void *writer(void* arg){
  while(!shouldexit){
    struct message msg;

    msg.type = 1;
    fgets(msg.content, TAILLE_MESS, stdin);

    /* Evite d'envoyer un message avec Ctrl-C dedans */
    if(shouldexit) break;

    sem_wait(&(shm_pere->sem));

    while(shm_pere->nb == MAX_MESS){
      sem_post(&(shm_pere->sem));
      sleep(1);
      sem_wait(&(shm_pere->sem));
    }

    shm_pere->messages[shm_pere->write] = msg;
    shm_pere->write = (shm_pere->write + 1) % MAX_MESS;
    shm_pere->nb++;

    sem_post(&(shm_pere->sem));
  }
  pthread_exit(NULL);
  return NULL;
}

void exithandler(int sig){
  shouldexit = sig; /* i.e. !=0 */
}

int main(int argc, char *argv[]){
  char *shm_client_name, *shm_pere_name;
  int shm_client_id, shm_pere_id;
```

```c
  pthread_t tids[2];
  struct message msg;
  struct sigaction action;

  if (argc <= 2) {
    fprintf(stderr, "Usage: %s id_client id_server\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  printf("Connexion à %s sous l'identité %s\n", argv[2], argv[1]);

  /* Met un handler pour arrêter le programme de façon clean avec Ctrl-C */
  action.sa_handler = exithandler;
  action.sa_flags = 0;
  sigemptyset(&action.sa_mask);
  sigaction(SIGINT, &action, 0);

  shm_client_name = (argv[1]);
  shm_pere_name = (argv[2]);

  /* Crée le segment en lecture écriture */
  if ((shm_client_id = shm_open(shm_client_name, O_RDWR | O_CREAT, 0666)) == -1) {
    perror("shm_open shm_client");
    exit(errno);
  }
  if ((shm_pere_id = shm_open(shm_pere_name, O_RDWR | O_CREAT, 0666)) == -1) {
    perror("shm_open shm_pere");
    exit(errno);
  }

  /* Allouer au segment une taille de NB_MESS messages */
  if(ftruncate(shm_client_id, sizeof(struct myshm)) == -1) {
    perror("ftruncate shm_client");
    exit(errno);
  }

  /* Mappe le segment en read-write partagé */
  if((shm_pere = mmap(NULL, sizeof(struct myshm), PROT_READ | PROT_WRITE, MAP_SHARED,
      shm_pere_id, 0)) == MAP_FAILED){
    perror("mmap shm_pere");
    exit(errno);
  }
  if((shm_client = mmap(NULL, sizeof(struct myshm), PROT_READ | PROT_WRITE, MAP_SHARED,
      shm_client_id, 0)) == MAP_FAILED){
    perror("mmap shm_pere");
    exit(errno);
  }
  shm_client->read = 0;
  shm_client->write = 0;
  shm_client->nb = 0;

  /* Initialisation du sémaphore (mutex) */
  if(sem_init(&(shm_client->sem), 1, 1) == -1){
    perror("sem_init shm_client");
    exit(errno);
  }

  msg.type = 0;
  strcpy(msg.content, argv[1]);
```

```
117   sem_wait(&(shm_pere->sem));
118
119   shm_pere->messages[shm_pere->write] = msg;
120   shm_pere->write = (shm_pere->write + 1) % MAX_MESS;
121   shm_pere->nb++;
122
123   sem_post(&(shm_pere->sem));
124
125   /* On crée les threads et on attend qu'ils se terminent */
126   pthread_create(&(tids[0]), NULL, reader, NULL);
127   pthread_create(&(tids[1]), NULL, writer, NULL);
128   pthread_join(tids[0], NULL);
129   pthread_join(tids[1], NULL);
130
131   printf("Déconnexion...\n");
132
133   msg.type = 2;
134   sem_wait(&(shm_pere->sem));
135
136   shm_pere->messages[shm_pere->write] = msg;
137   shm_pere->write = (shm_pere->write + 1) % MAX_MESS;
138   shm_pere->nb++;
139
140   sem_post(&(shm_pere->sem));
141
142   sem_close(&(shm_client->sem));
143
144   munmap(shm_client, sizeof(struct myshm));
145
146   shm_unlink(shm_client_name);
147
148   return EXIT_SUCCESS;
149 }
```