Système d'exploitation - Lab

Quentin Marques

28 mars 2013

Résumé

Ce document contient les réponses aux Travaux Pratiques/Dirigés du cours de système d'exploitation de M. Christian Khoury. Il contient en outre des notes de cours et des résumés de recherches internet sur les sujets incluant entre autres :

- Threads et processus (process)
- Verrous exclusifs (mutexes), conditions variables et sémaphores
- Accès concurrents (race conditions) et inter-blocage (deadlocks)

L'intégralité des sources de ce document, incluant les codes sources LaTeXet les exemples en C, ainsi que les schémas sont librement disponible et modifiable sur la page GitHub suivante : `https://github.com/Zaurak/Lab-OperatingSystem`

Table des matières

1 Threads

1.1 Processes vs Threads

1.1.1 Théorie

Process

Un *processus* est un *programme* (code exécutable) qui est *exécuté*. La technique par laquelle un même programme s'exécute lui-même plusieurs fois (en plusieurs processus) est un *fork()*. Les différents processus s'exécutent en parallèle, chacun avec son espace mémoire propre et ses ressources (même s'il est possible de définir des segments de mémoire partagés).

Thread

Un *thread* est un *processus léger*. Un thread fait partie des instructions exécutées par un processus en cours. Ainsi un processus peut contenir plusieurs threads et l'on parle alors d'application *multi-threadées*. Les différents threads s'exécutent quasi-simultanément sur une architecture à 1 cœur et simultanément s'il est possible de les répartir sur les différents cœurs de l'ordinateur. Les threads partagent le même espace mémoire et les même ressources, ce qui peut résulter en des problèmes divers de *synchronisations*, d'*accès concurrent* (race condition) ou encore d'*inter-blocage* (deadlock).

1.1.2 Exercice

Énoncé

Calculer l'expression suivante :

$$(a+b) - \left[\frac{(c*d)}{(e-f)}\right] + (g+h)$$

En utilisant :
- Des processes
- Des threads

Quelle est la différence principale entre ces deux implémentations ?

Solution

Nous découperons le calcul à raison d'un process/thread par groupement parenthésé. Dans les deux cas nous calculerons le temps d'exécution réel grâce au framework intégré de Code : :Blocks (la dernière ligne de la trace d'exécution est donc ajoutée).

Avec Process Codes/compute–with–processes.c

```
1 /** Lab Subject :
2   * Parallel computing using processes.
3   * 4 child processes (+1 parent process) :
4   *     ~ (a + b)
5   *     ~ (c * d)
```

```
 *      ~ (e - f)
 *      ~ (g + h)
 * => (a+b) - (c*d)/(e-f) + (g+h)
 *
 * (P)
 *  |__(1)
 *  |    |__(2)
 *  |    |    |__(3)
 *  |  a+b  |    |__(4)
 *  |    |  g+h  |    |
 *  |    |    |  c*d  e-f
 *  |    |    |    |___|
 *  |    |    |___|
 *  |    |___|
 *  |___|
 *  |
 */

#include <stdlib.h>     // Useful for exit() function
#include <stdio.h>      // Input/Output
#include <unistd.h>     // Useful for fork() function
#include <sys/types.h>  // System utilities...
#include <sys/shm.h>    //   ...
#include <sys/wait.h>   //   ...

#define KEY 4567
#define PERMS 0660      // Define an rw- permission for user and group.

int main(int argc, char* argv[]) {
    int id;             // Id of the shared memory segment
    int* shared_data;   // Pointer to the shared memory
    int result;         // "Local" result variable (Process dependant)

    // Parameters
    int a = 5, b = 4, c = 3, d = 2, e = 0, f = 6, g = 7, h = 8;

    // Allocate a new int sized shared memory segment and keep its id
    id = shmget(KEY, sizeof(int), IPC_CREAT | PERMS);

    // Get the access to the memory allocated previously
    shared_data = (int*) shmat(id, NULL, 0);

    // Start of a fork
    if (fork() == 0) {  // First Child process

        printf("Child process 1: "
            "Child1.res = (a+b)\n");
        result = a + b;

        if (fork() == 0) {  // Second Child process
            printf("Child process 2: "
                "Child2.res = (g+h)\n");
            result = g + h;
```

```
59
60              if (fork() == 0) {  // Third Child process
61                  printf("Child process 3: "
62                      "Child3.res = (c*d)\n");
63                  result = c * d;
64
65                  if (fork() == 0) {  // Fourth Child process
66                      printf("Child process 4: "
67                          "shared_data = (e-f)\n");
68                      *(shared_data) = e - f;
69                      exit(0);
70                  }   // End of Child 4
71
72                  wait(NULL);
73                  printf("Child process 3: "
74                      "shared_data = - Child3.res / shared_data\n");
75                  *(shared_data) = - *(shared_data) / result;
76                  exit(0);
77              } // End of Child 3
78              wait(NULL);
79              printf("Child process 2: "
80                  "shared_data = Child2.res + shared_data\n");
81              *(shared_data) = *(shared_data) + result;
82              exit(0);
83          } // End of Child 2
84          wait(NULL);
85          printf("Child process 1: "
86                  "shared_data = Child1.res + shared_data\n");
87          *(shared_data) = *(shared_data) + result;
88          exit(0);
89      } else {// End of Child 1
90          // Parent Process
91          printf("Parent process: Waiting end of child process 1...\n");
92
93          wait(NULL);      // Wait the termination of the first child process
94
95          printf("Result: %d\n", *(shared_data));
96
97          // Mark the segment to be destroyed : Free the shared memory
98          shmctl(id, IPC_RMID, NULL);
99      }
100     return 0;
101 }
```

Codes/compute–with–processes.c

```
user@user-machine:~$ gcc compute-with-processes.c
user@user-machine:~$ ./a.out
Parent process: Waiting end of child process 1...
Child process 1: Child1.res = (a+b)
Child process 2: Child2.res = (g+h)
```

```
Child process 3: Child3.res = (c*d)
Child process 4: shared_data = (e-f)
Child process 3: shared_data = - Child3.res / shared_data
Child process 2: shared_data = Child2.res + shared_data
Child process 1: shared_data = Child1.res + shared_data
Result: 25

Process returned 0 (0x0)    execution time : 0.002 s
```

Avec Threads

Comme toutes les variables sont différentes d'un groupement parenthésé à l'autre, aucun problème de synchronisation n'est à prévoir et l'on peut utiliser simplement la valeur de retour des threads.

Codes/compute–with–threads.c

```c
/** Lab Subject :
  * Parallel computing using threads.
  * 2 functions :
  *     ~ add (Note that the substraction will be handle by add too)
  *     ~ mul (Multiplication. Division is not handled by a thread)
  * 4 threads :
  *     ~ (a + b)
  *     ~ (c * d)
  *     ~ (e - f)
  *     ~ (g + h)
  * => (a+b) - (c*d)/(e-f) + (g+h)
  */

#include <stdio.h>      // Input/Output
#include <pthread.h>    // Threads management
#include <sys/time.h>   // Time management

// Define a struct to contain 2 operands in one variable
// (Because we can only send one variable to a thread)
typedef struct operands {
    int a, b;       // Store the operands
    int ret;        // Store the return value
}t_operands;

// Handle addition and substraction
void* add(void* operands) {
    t_operands* ope = (t_operands*) operands;   // Parameter cast
    ope->ret = ope->a + ope->b;                 // Record the result
    return NULL;
}

// Handle addition and substraction
void* mul(void* operands) {
    t_operands* ope = (t_operands*) operands;   // Parameter cast
```

```
35      ope->ret = ope->a * ope->b;                     // Record the result
36      return NULL;
37  }
38
39  // Main function
40  int main(int argc, char* argv[]) {
41      int i;              // Iterator
42
43      int result;         // Store the result
44
45      pthread_t thread[4];    // Array of threads
46
47      int iret[4];            // Array of thread return values
48
49      // Parameters
50      int a = 5, b = 4, c = 3, d = 2, e = 0, f = 6, g = 7, h = 8;
51
52      t_operands params[4];
53
54      params[0].a = a;     params[0].b =   b;
55      params[1].a = c;     params[1].b =   d;
56      params[2].a = e;     params[2].b = - f;
57      params[3].a = g;     params[3].b =   h;
58
59      // Iteration loop (to have significant value for Execution time)
60      // Creating threads
61      iret[0] = pthread_create(&thread[0], NULL,
62          add, (void*) &params[0]);
63      iret[1] = pthread_create(&thread[1], NULL,
64          mul, (void*) &params[1]);
65      iret[2] = pthread_create(&thread[2], NULL,
66          add, (void*) &params[2]);
67      iret[3] = pthread_create(&thread[3], NULL,
68          add, (void*) &params[3]);
69
70      // Wait until threads are all complete
71      for (i = 0 ; i < 4 ; i++) {
72          pthread_join(thread[i], NULL);
73          printf("Thread %d returned %d\n", i, iret[i]);
74      }
75
76      if (params[2].ret != 0) {
77          result = params[0].ret
78              - params[1].ret/params[2].ret
79              + params[3].ret;
80          printf("Result: %d\n", result);
81          return 0;
82      } else {
83          printf("Error: Can't divide by 0 (e[%d] - f[%d])\n", e, f);
84          return -1;
85      }
86  }
```

Codes/compute–with–threads.c

```
user@user-machine:~$ gcc compute-with-threads.c -lpthread
user@user-machine:~$ ./a.out
Result: 25

Process returned 0 (0x0)    execution time : 0.001 s
```

Conclusion

On constate que les threads sont plus rapide que les processus car ces premiers n'ont pas à copier l'intégralité du contexte d'exécution les précédant. Cependant, ces tests sont peu représentatifs car sur de courtes durées de calculs.

1.2 Synchronizing Threads

1.2.1 Théorie

Les techniques de synchronisation

Il existe 3 moyens de synchroniser des threads entre eux :

- mutexes : *Vérrous* d'exclusion mutuelle.
- joins : Attendre qu'un thread spécifié *rejoigne* le flux du thread appelant.
- condition variables : Céder la main à d'autre threads tant qu'un signal n'a pas été reçu.

Mutex

Le *mutex* empêche, une fois vérrouiller, tout autre thread de modifier les variables utilisés entre le vérrouillage et le déverrouillage. Toute tentative de modification par un thread d'une variable verrouiller par un mutex entraînera un blocage de ce thread le temps que le mutex soit déverrouiller.

La section de code ainsi "protégé" par un mutex est appelé *section critique*.

Codes/snippet–mutex.c

```
1  /** Code snippet : Mutex
2    * This is a sample code explaining how mutex works.
3    * The goal is to print the 5 first multiple of a given value in one
4    * thread and to square this given value in another thread.
5    */
6
7  #include <stdio.h>       // Input/Output
8  #include <pthread.h>     // Threads management
9
10 // Creating mutex
11 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
12
13 // Global given value
```

```c
int value = 2;

void* print_first_multiples(void* nothing) {
    // Iterator
    int i;

    // MUTEX - LOCK
    pthread_mutex_lock( &mutex );

    printf("First multiples of %d are:\n", value);

    for(i = 1 ; i <= 5 ; i++) {
        printf("\t %d x %d = %d\n", value, i, value*i);
    }

    // MUTEX - UNLOCK
    pthread_mutex_unlock( &mutex );
    return NULL;
}

void* square_value(void* nothing) {
    // MUTEX - LOCK
    pthread_mutex_lock( &mutex );

    printf("Square of %d is ..", value);
    value = value*value;
    printf("... %d.\n", value);

    // MUTEX - UNLOCK
    pthread_mutex_unlock( &mutex );

    return NULL;
}

// Main function
int main(int argc, char* argv[]) {
    // Threads declaration
    pthread_t thread1;
    pthread_t thread2;

    // Creating threads
    pthread_create(&thread1, NULL, print_first_multiples, NULL);
    pthread_create(&thread2, NULL, square_value, NULL);

    // Wait the end of all threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

Codes/snippet–mutex.c

```
user@user-machine:~$ gcc snippet-mutex.c -lpthread
user@user-machine:~$ ./a.out
First multiples of 2 are:
    2 x 1 = 2
    2 x 2 = 4
    2 x 3 = 6
    2 x 4 = 8
    2 x 5 = 10
Square of 2 is ..... 4.
user@user-machine:~$ ./a.out
Square of 2 is ..... 4.
First multiples of 4 are:
    4 x 1 = 4
    4 x 2 = 8
    4 x 3 = 12
    4 x 4 = 16
    4 x 5 = 20
```

Dans le premier lancement, nous avons *thread1* qui est plus rapide et verrouille la ressource *value* pour son usage. Dans le second lancement, c'est l'inverse.

En aucun cas nous n'aurions pu avoir :

```
user@user-machine:~$ ./a.out
First multiples of 2 are:
    2 x 1 = 2
    2 x 2 = 4
    2 x 3 = 6
Square of 2 is ..... 4.
    4 x 4 = 16
    4 x 5 = 20
```

Join

Le *join* attend que le thread spécifié ait pu terminé avant de redonner la main à la fonction appelante. C'est notamment très utile pour s'assurer que le travail d'un thread a bien été effectué avant d'entamer une autre tâche dépendant du résultat de ce thread. Dans l'exemple précédent, nous l'avons utilisé pour attendre la fin de tout les threads mais on peux également l'utiliser pour d'autres tâches de synchronisation.

Dans le code suivant, on effectue une moyenne en divisant la somme dans deux threads :

$$Somme = \sum_{i=0}^{N} \text{array}[i] = \underbrace{\sum_{j=0}^{N/2} \text{array}[j]}_{\text{Thread 1}} + \underbrace{\sum_{k=N/2}^{N} \text{array}[k]}_{\text{Thread 2}}$$

Codes/snippet–join.c

```c
/** Code snippet : Join
  * This is a sample code explaining how join works.
  * The goal is to compute/print the average value of a really long
  * list of numbers.
  * We will use 2 threads to compute the sum and 1 (the main) to divide
  * it by the number of records.
  *                    Inspired by the Monte-Carlo Algorithm.
  */

#include <stdio.h>      // Input/Output
#include <stdlib.h>     // Useful for rand() function
#include <pthread.h>    // Threads management

#define NB_REC 1000                                     // Records number

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  // Creating mutex

int recordList[NB_REC];                                 // Records array

int result = 0;                                         // Result variable

// Define the range of the array to be computed by the thread
typedef struct range {
    int start;
    int end;
}t_range;

void* compute_sum(void* range) {

    // Initialisation
    t_range r = *((t_range*) range);    // Cast conversion
    int i;                              // Iterator
    int partialSum = 0;                 // Partial sum

    // Computation
    for (i = r.start ; i < r.end ; i++) {
        partialSum += recordList[i];
    }

    // Synchronisation of the partial sums
    pthread_mutex_lock( &mutex );       // MUTEX - LOCK
    result += partialSum;
    pthread_mutex_unlock( &mutex );     // MUTEX - UNLOCK
```

```
44
45      return NULL;
46  }
47
48  // Main function
49  int main(int argc, char* argv[]) {
50      // Iterator
51      int i;
52
53      // Threads declaration
54      pthread_t thread1;
55      pthread_t thread2;
56
57      // struct range declaration/initialisation
58      t_range r1 = {0, NB_REC/2};
59      t_range r2 = {NB_REC/2, NB_REC};
60
61      // Initialising fake values for the array
62      for (i = 0 ; i < NB_REC ; i++) {
63          recordList[i] = rand() % 101 + 1;    // value range: [1, 100]
64      }
65
66      // Creating threads
67      pthread_create(&thread1, NULL, compute_sum, (void*) &r1);
68      pthread_create(&thread2, NULL, compute_sum, (void*) &r2);
69
70      // Wait the end of thread 1 & 2
71      pthread_join(thread1, NULL);
72      pthread_join(thread2, NULL);
73
74      // We are now sure that the sum is complete
75      result = result / NB_REC;
76
77      // Print the result
78      printf("Result = %d\n", result);
79
80      return 0;
81  }
```

Codes/snippet–join.c

Condition variables

Les *condition variables* sont des verrous conditionnels. Un thread qui attend qu'une condition soit réalisé cède alors la main (et déverrouille son mutex) au profit des autres threads. Un thread qui remplit la condition enverra alors un signal à ce thread pour l'avertir qu'il peux arrêter d'attendre et récupérer son verrou.

Codes/snippet–condvar.c

```
1  /** Code snippet : Condition Variables
2    * This is a sample code explaining how condition variables works.
```

```
3   * The goal is to show the Collatz Conjecture : take a random number,
4   * if it's even, divide by 2, if it's odd, multiply by 3 and add 1.
5   * The conjecture say that it should reach 1 at some point.
6   * 1 thread will handle even numbers
7   * 1 thread will handle odd numbers
8   */
9
10 #include <stdio.h>       // Input/Output
11 #include <stdlib.h>      // Useful for atoi() function
12 #include <pthread.h>     // Threads management
13 #include <unistd.h>      // Useful for usleep() function
14
15 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  // Creating mutex
16 pthread_cond_t cond_var = PTHREAD_COND_INITIALIZER; // Creating cond_var
17
18 // Random number
19 int value;
20
21 void* collatz_even(void* nothing) {
22
23     for(;;) {                // Infinite loop
24         // MUTEX - LOCK
25         pthread_mutex_lock( &mutex );
26         printf("[EVEN THREAD] LOCK\n");
27
28         if (value % 2 == 0) {    // If it's even
29             printf("[EVEN THREAD] value: %2d => %2d\n", value, value/2);
30             value = value / 2;
31             printf("%d\n", value);
32         } else {                 // If it's odd
33             // Signal sent to stop waiting of (one) other thread
34             pthread_cond_signal(&cond_var);
35             printf("[EVEN THREAD] value: %2d => SIGNAL SENT\n", value);
36         }
37
38         // MUTEX - UNLOCK
39         printf("[EVEN THREAD] UNLOCK\n");
40         pthread_mutex_unlock( &mutex );
41
42         // Make a short pause to let time to the other thread
43         // to receive the signal. This avoid useless loops in this
44         // even thread
45         usleep(1000);     // Wait 1 ms
46
47         // End condition
48         if (value == 1) {
49             // Send signal so that the other thread don't wait forever
50             pthread_cond_signal(&cond_var);
51             return NULL;
52         }
53     }
54 }
55
```

```c
void* collatz_odd(void* nothing) {

    for(;;) {                // Infinite loop
        // MUTEX - LOCK
        pthread_mutex_lock( &mutex );
        printf("[ODD THREAD] LOCK\n");


        // Wait for signal
        printf("[ODD THREAD] WAITING SIGNAL\n");
        pthread_cond_wait(&cond_var, &mutex);
        printf("[ODD THREAD] SIGNAL RECEIVED\n");

        // Avoid going in infinite loop (1, 4, 2, 1, 4, 2, 1....)
        if (value != 1) {
            // Do the operation
            printf("[ODD THREAD] value: %2d  => %2d\n",
                value, value*3 + 1);
            value = value*3 + 1;
            printf("%d\n", value);
        }
        // MUTEX - UNLOCK
        printf("[ODD THREAD] UNLOCK\n");
        pthread_mutex_unlock( &mutex );

        // End condition
        if (value == 1) return NULL;
    }
}

// Main function
int main(int argc, char* argv[]) {

    // Getting the starting value from the prompt
    if (argc != 2) {
        fprintf(stderr, "Usage: %s starting_val\n", argv[0]);
        return -1;
    }
    value = atoi(argv[1]);

    printf("Starting Collatz Conjecture with value:\n");
    printf("%d\n", value);

    // Threads declaration
    pthread_t thread1;
    pthread_t thread2;

    // Creating threads
    pthread_create(&thread1, NULL, collatz_even, NULL);
    pthread_create(&thread2, NULL, collatz_odd, NULL);

    // Wait the end of all threads
    pthread_join(thread1, NULL);
```

```
109     pthread_join(thread2, NULL);
110
111     return 0;
112 }
```

Codes/snippet–condvar.c

```
user@user-machine:~$ ./a.out 5
Starting Collatz Conjecture with value:
5
[EVEN THREAD] LOCK
[EVEN THREAD] value:  5 => SIGNAL SENT
[EVEN THREAD] UNLOCK
[ODD THREAD] LOCK
[ODD THREAD] WAITING SIGNAL
[EVEN THREAD] LOCK
[EVEN THREAD] value:  5 => SIGNAL SENT
[EVEN THREAD] UNLOCK
[ODD THREAD] SIGNAL RECEIVED
[ODD THREAD] value:  5  => 16
16
[ODD THREAD] UNLOCK
[ODD THREAD] LOCK
[ODD THREAD] WAITING SIGNAL
[EVEN THREAD] LOCK
[EVEN THREAD] value: 16 =>  8
8
[EVEN THREAD] UNLOCK
[EVEN THREAD] LOCK
[EVEN THREAD] value:  8 =>  4
4
[EVEN THREAD] UNLOCK
[EVEN THREAD] LOCK
[EVEN THREAD] value:  4 =>  2
2
[EVEN THREAD] UNLOCK
[EVEN THREAD] LOCK
[EVEN THREAD] value:  2 =>  1
1
[EVEN THREAD] UNLOCK
[ODD THREAD] SIGNAL RECEIVED
[ODD THREAD] UNLOCK
```

1.2.2 Exercice

Énoncé

Créer 3 threads ; 2 d'entre eux incrémenteront un compteur partagé pendant que le troisième attends que le compteur atteigne un montant donné. Implémenter ce simple exercice en utilisant les threads avec les *conditions variables*

Solution Codes/syncthreads.c

```c
/** Lab Subject :
  * 2 threads to increment a shared counter
  * 1 thread to wait that a given amount is reached
  */

#include <stdio.h>      // Input/Output
#include <unistd.h>     // Useful for usleep() function
#include <pthread.h>    // Threads management

pthread_cond_t  cond_var = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t id_mutex = PTHREAD_MUTEX_INITIALIZER;

int counter = 0;    // Global counter

#define LIMIT 10    // Global LIMIT constant

// Simple function to get a comprehensive unique ID for threads
int getId() {
    static int id = 0;
    pthread_mutex_lock(&id_mutex);      // MUTEX - LOCK
    id++;
    pthread_mutex_unlock(&id_mutex);    // MUTEX - UNLOCK
    return id;
}

//
void* increment(void* nothing) {
    int exit = 0;   // Loop condition
    int shortId = getId();   // Compute a short ID

    while(!exit) {      // Infinite loop

        pthread_mutex_lock(&mutex);         // MUTEX - LOCK

        if (counter >= LIMIT) {     // CONDITION VARIABLE CHECK
            printf("[INCREMENT] %d : SIGNAL SENT\n", shortId);
            pthread_cond_signal(&cond_var);     // Send signal
            exit = 1;       // Break the loop and terminate thread
        } else {
            // Print a trace of the thread activity
            printf("[INCREMENT] %d : Counter from '%d' to '%d'\n",
            shortId, counter, counter + 1);
```

```
44
45              counter = counter + 1;       // Increment
46          }
47
48          pthread_mutex_unlock(&mutex);      // MUTEX - UNLOCK
49
50          // Wait 1ms to avoid that only one thread appear in the output
51          usleep(1000);
52      }
53
54      // "Natural" termination of the thread
55      printf("\tTermination of [INCREMENT] %d\n", shortId);
56      return NULL;
57  }
58
59  void* isReached(void* nothing) {
60
61      int shortId = getId();
62
63      pthread_mutex_lock(&mutex);          // MUTEX - LOCK
64
65      // Wait for signal from increment threads
66      if (counter < LIMIT) { // Avoid missing signals skipping cond_wait
67          printf("[ISREACHED] %d : WAITING SIGNAL\n", shortId);
68          pthread_cond_wait(&cond_var, &mutex);
69      }
70      // Trace of the received signal
71      printf("[ISREACHED] %d : SIGNAL RECEIVED\n", shortId);
72      printf("[ISREACHED] %d : Counter = %d\n", shortId, counter);
73
74      pthread_mutex_unlock(&mutex);      // MUTEX - UNLOCK
75
76      // "Natural" termination of the thread
77      printf("\tTermination of [ISREACHED] %d\n", shortId);
78      return NULL;
79  }
80
81  // Main function
82  int main(int argc, char* argv[]) {
83      printf("Synchronized Threads Lab\n");
84
85      // Iterator
86      int i;
87
88      // Array of threads
89      pthread_t thread[3];
90
91      // Array of thread return values
92      int iret[3];
93
94      // Creating threads
95      iret[0] = pthread_create(&thread[0], NULL, increment, NULL);
96      iret[1] = pthread_create(&thread[1], NULL, increment, NULL);
```

```c
    iret[2] = pthread_create(&thread[2], NULL, isReached, NULL);

    // Wait until threads are all complete
    for (i = 0; i < 3; i++)
        pthread_join(thread[i], NULL);

    // Print the result (return value) of all threads
    for (i = 0; i < 3; i++)
        printf("Thread %d returns: %d\n", i, iret[i]);

    return 0;
}
```

Codes/syncthreads.c

```
user@user-machine:~$ ./a.out
Synchronized Threads Lab
[INCREMENT] 1 : Counter from '0' to '1'
[INCREMENT] 2 : Counter from '1' to '2'
[ISREACHED] 3 : WAITING SIGNAL
[INCREMENT] 1 : Counter from '2' to '3'
[INCREMENT] 2 : Counter from '3' to '4'
[INCREMENT] 2 : Counter from '4' to '5'
[INCREMENT] 1 : Counter from '5' to '6'
[INCREMENT] 2 : Counter from '6' to '7'
[INCREMENT] 1 : Counter from '7' to '8'
[INCREMENT] 2 : Counter from '8' to '9'
[INCREMENT] 1 : Counter from '9' to '10'
[INCREMENT] 2 : SIGNAL SENT
[ISREACHED] 3 : SIGNAL RECEIVED
Termination of [INCREMENT] 2
[INCREMENT] 1 : SIGNAL SENT
[ISREACHED] 3 : Counter = 10
Termination of [ISREACH] 3
Termination of [INCREMENT] 1
Thread 0 returns: 0
Thread 1 returns: 0
Thread 2 returns: 0
```

2 Interprocess Communication

2.1 Shared Memory and Race Problems

2.1.1 Théorie

Mémoires partagées

Nous avons déjà vu dans la partie 1.1.2 un code exploitant les différentes techniques de création d'espaces mémoire partagés pour les processus. A titre de rappel, voici les quelques fonctions utiles à connaître :

Codes/snippet–shared–memory.c

```
1  /** Important functions to be used while creating a shared memory space
2    * Most requires : <sys/shm.h>, <sys/ipc.h> and <sys/types.h>
3    *
4    */
5
6  #define KEY 4567
7  #define PERMS 0660       // Define an rw- permission for user and group.
8
9  int     id;              // Id of the shared memory segment
10 int* shared_data;   // Pointer to the shared memory
11
12 // SHMGET - Allocate a shared memory segment
13 //
14 //      int shmget(key_t key, size_t size, int shmflg);
15 //
16 // Example: Allocate a new int sized shared memory segment
17 id = shmget(KEY, sizeof(int), IPC_CREAT | PERMS);
18
19 // SHMAT - Attaches the shared memory segment to the address space
20 // of the calling process.
21 //
22 //      void* shmat(int shmid, const void* shmaddr, int shmflg);
23 //
24 // Example: Get the access to the memory allocated previously
25 shared_data = (int*) shmat(id, NULL, 0);
26
27 // SHMDT - Detaches the shared memory segment from the adress space
28 // of the calling process.
29 //
30 //      int shmdt(const void* shmaddr);
31 //
32
33 // Access to the shared data
34 *(shared_data) = 42;
35
36 // SHMCTL - Shared memory control. Perform the <ctl> control on memory.
37 //
38 //      int shmctl(int shmid, int cmd, struct shmid_ds* buf);
39 //
40 // Example: Mark the segment to be destroyed, Free the shared memory
41 shmctl(id, IPC_RMID, NULL);
```

Codes/snippet–shared–memory.c

Accès concurrent

Une *race condition* peut arriver quand l'on tente d'accéder en écriture (ou en lecture) simultanément à une même ressource. Une des modifications peut alors ne pas être prise en compte ou résulter en comportement incohérent et imprévisible (Voir *Figure 1*). Dans l'exercice suivant, on simulera un tel accès concurrent pour mettre en évidence le problème.

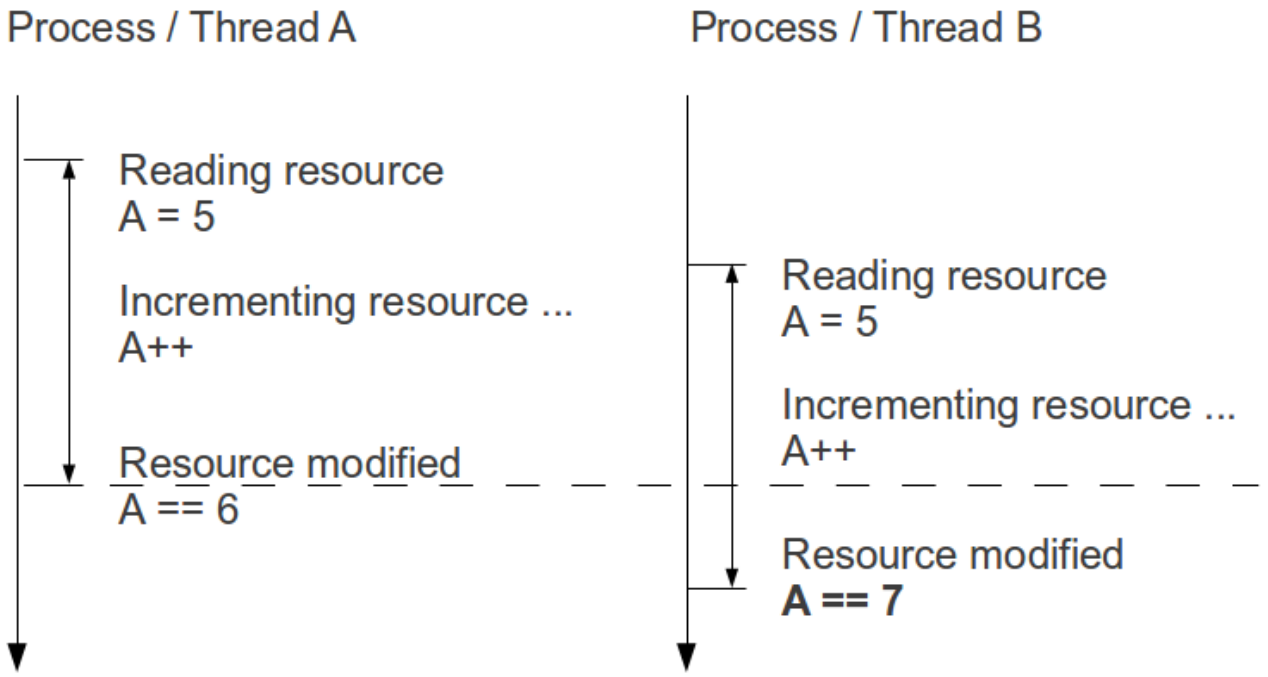

FIGURE 1 – Exemple d'incohérence dû à un accès concurrent

Il est possible d'éviter ce genre de problèmes en utilisant les techniques de synchronisation vues précédemment, notamment les *mutexes*. Cependant, ces techniques peuvent engendrées d'autres problèmes...

Inter-blocage

Un problème lié à l'utilisation des techniques de synchronisation (mutex, join et condition variable) est l'apparition de situation d'*inter-blocage* (deadlock). Ceci peut arriver dans plusieurs situations notamment :

1. Quand un thread/process tente de verrouiller une ressource déjà verrouillé par un second thread/process et que le premier thread/process est déjà en train de verrouiller une ressource requise par le second. Le premier attendra indéfiniment que le second libère le verrou mais comme celui-ci requière une ressource elle-même verrouiller par le premier, cela n'arrivera jamais (Voir *Figure 2*. On corrige ce type de problèmes en :

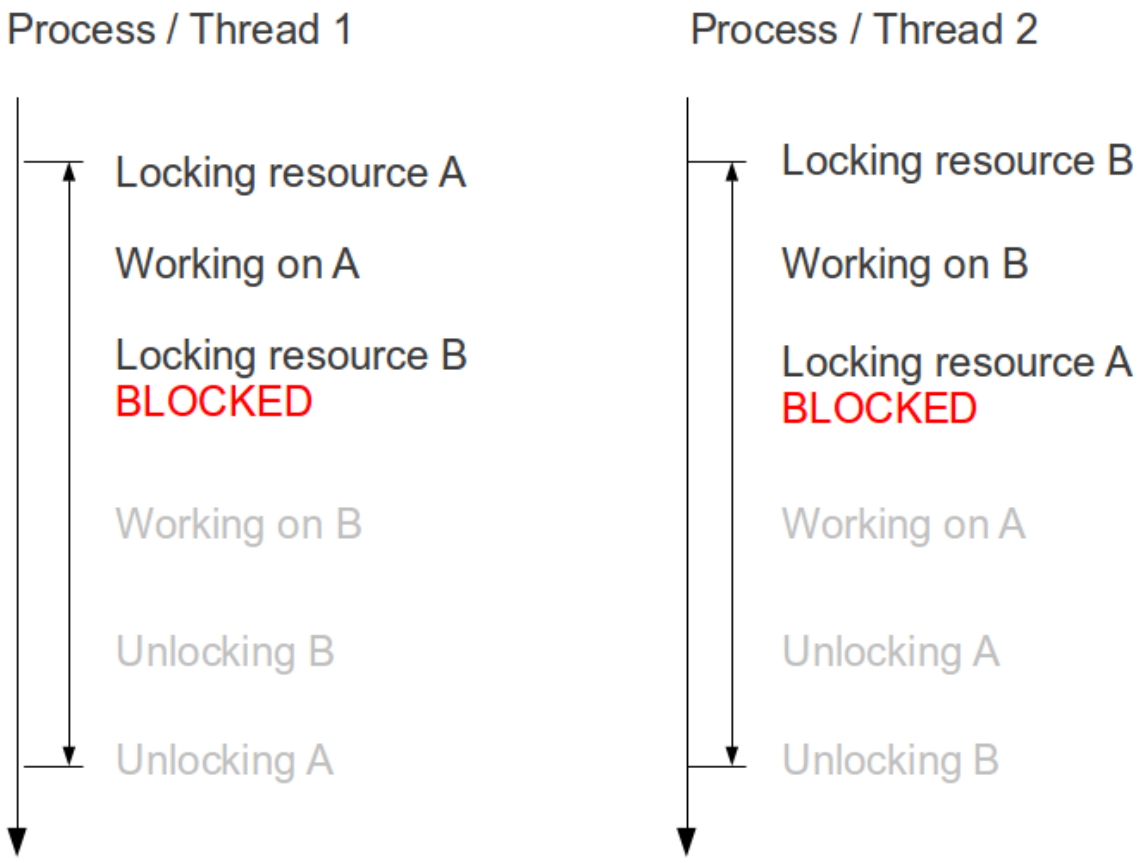

FIGURE 2 – Problème d'inter-blocage dû aux verrous exclusifs

– Définissant un ordre fixe d'utilisation des ressources quels que soient les threads/processes utilisés. Ceci assure que les ressources seront verrouillées dans cet ordre et déverrouillées dans l'ordre inverse (Voir *Figure 3*).
– Vérifiant que le verrou peut être mis grâce à `pthread_mutex_trylock()`. Le thread/-process n'est alors pas "bloqué" et l'on peut éventuellement prendre une décision pour gérer l'erreur (attendre un temps aléatoirement long par exemple). Reste à ne pas tomber dans l'autre extrême qui serait de définir tout code parallélisé comme étant verrouillé par un verrou exclusif. Cela casserait l'intérêt de la parallélisation. Les verrous ne devrait être utilisés que sur des sections critiques aussi petites que possible.

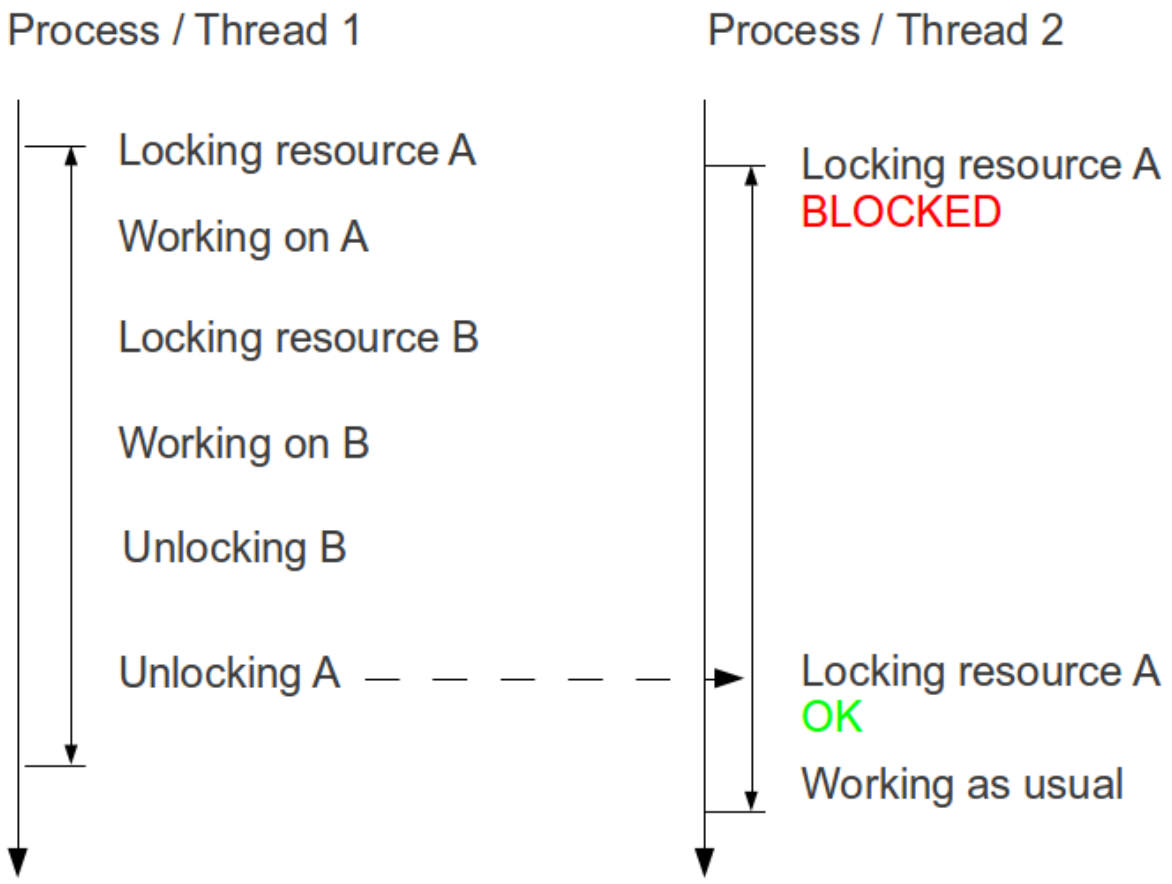

FIGURE 3 – Exemple de résolution du problème des *deadlocks* dû aux mutexes

2. Quand un thread/process attend sur une *condition variable* un signal qui soit a déjà été envoyé avant que le thread/process puisse être en position d'attente, soit que la modification de la condition variable entraîne une *race condition* (Voir *Figure 4*. On corrige ce type de problèmes en :
   – Entourant le `pthread_cond_wait()` par un verrou exclusif (mutex) entre le code "attendant" et celui "signalant". Ainsi on a l'assurance qu'une fois le thread lancé celui-ci aura le temps d'arriver à la condition d'attente sans recevoir de manière prématurée le signal (voir *Figure 5*).
   – Vérifiant avant de déclencher l'attente du signal que la condition n'est pas déjà remplie.

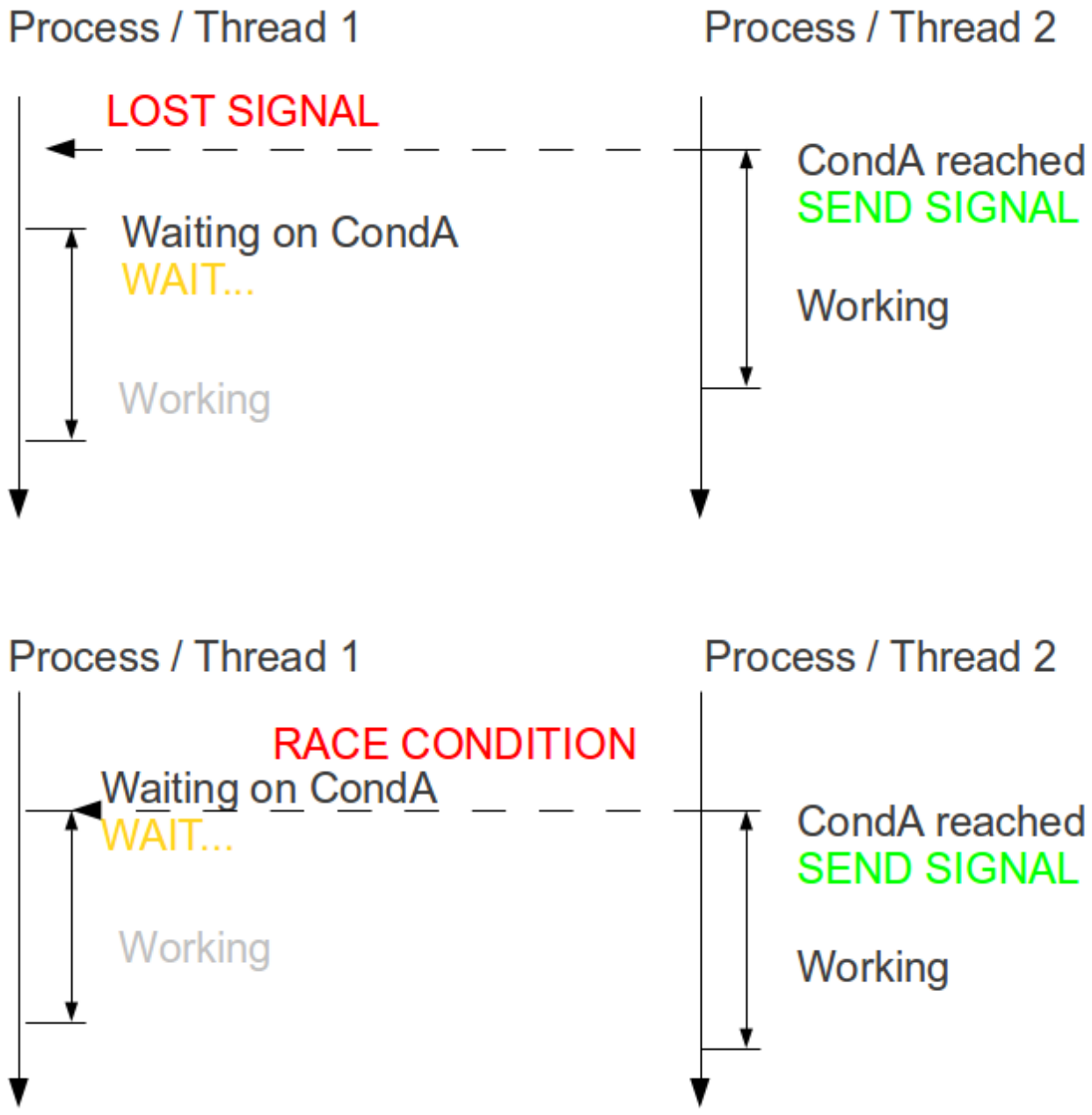

FIGURE 4 – Problème d'inter-blocage dû aux conditions variables

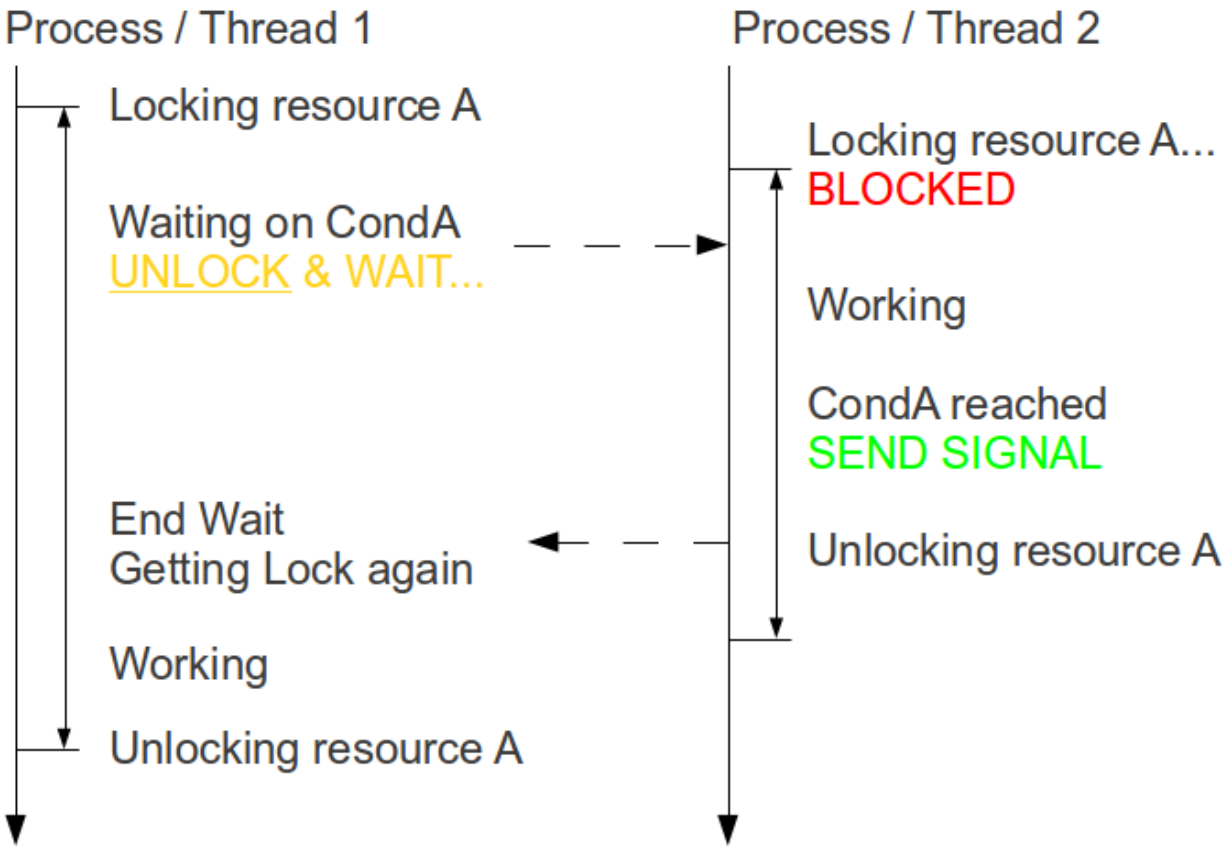

FIGURE 5 – Exemple de résolution du problème des *deadlocks* dû aux conditions variables

2.1.2 Exercice

Énoncé

Mettez en évidence par la simulation une situation d'accès concurrent.

Solution

Dans un but de clarté du code, nous utiliserons ici des threads plutôt que des processus. En effet, avec des threads nous n'aurons pas besoin de réaliser la syntaxe lourde de gestion des processus enfants (niveaux d'indentation) ainsi que la gestion des segments de mémoires partagées déjà traités auparavant.

Le code suivant simule un accès concurrent de N threads de gestion bancaire, chacun augmentant le solde d'un compte par tranche de 10€ jusqu'à atteindre 100€. Nous devrions donc obtenir à la fin un solde de $N*100$€ sans compter sur les incohérences liés aux accès concurrents.

Codes/simul–race–condition.c

```c
/** Lab Subject :
  * Simulate a race condition using threads
  */

#include <stdio.h>      // Input/Output
#include <pthread.h>    // Threads management

#define N_THREAD 10

int account = 0;

void* deposit(void* nothing) {
    int i;
    for (i = 0 ; i < 10 ; i++) {
        account = account + 10;
        usleep(50000);  // Wait 50 ms
    }
    return NULL;
}

// Main function
int main(int argc, char* argv[]) {
    int i;          // Iterator

    pthread_t thread[N_THREAD];     // Array of threads

    int iret[N_THREAD];             // Array of thread return values

    // Creating threads
    for (i = 0 ; i < N_THREAD ; i++) {
        iret[i] = pthread_create(&thread[i], NULL, deposit, NULL);
    }

    // Wait until threads are all complete
    for (i = 0 ; i < N_THREAD ; i++) {
        pthread_join(thread[i], NULL);
```

```
37      }
38      printf("Account/N_THREAD(%d): %d\n", N_THREAD, account/N_THREAD);
39      return 0;
40  }
```

Codes/simul–race–condition.c

```
user@user-machine:~$ ./a.out
Account/N_THREAD(10): 84
```

Après de multiples exécution de ce programme, on constate que les valeurs de retours sont complètement inconsistantes (oscillant entre 75 et 99).

2.2 Synchronizing access using semaphores

Dans cette partie, nous allons essayer d'implémenter le concept de verrou d'exclusion mutuelle à l'aide de *sémaphores*. Il est très intéressant d'utiliser les sémaphores pour un tel usage car le concept de mutex ne s'applique pas directement au cas des processus (chaque processus ayant ses propres ressources, mutex inclus, ceux-ci sont sans effet sur les accès concurrents sur un segment de mémoire partagé entre processus).

Notez que nous implémenterons à l'aide de sémaphores mais sur des threads plutôt que des processus dans le même soucis de lisibilité que précédemment.

2.2.1 Théorie

Définition

Nous utiliserons ici les *sémaphores Système V*, permettant - au contraire des sémaphores classiques - de fonctionner entre deux processus différents.

Un sémaphore est une variable et constitue une méthode couramment utilisée pour synchroniser la mémoire lors d'une exécution parallélisée.

Leur utilisation logicielle est permise par une implémentation matérielle (au niveau du microprocesseur), permettant de tester et modifier la variable protégée au cours d'un *cycle insécable*.

Opérations P et V

P et V du néerlandais Proberen et Verhogen signifient tester et incrémenter. En français, on s'en souviendra plutôt par "Puis-je ?"/"Prendre" et "Vas-y !"/"Vendre".

Concrètement, une sémaphore contient une valeur correspondant au nombre de ressources disponibles. Dans le cas d'une ressource unique, cette valeur peut être :

- 0 : Ressource non-disponible
- 1 : Ressource disponible

L'opération P va tenter de décrémenter la valeur de la sémaphore ("Puis-je ?"). Si cette valeur ne peut pas être décrémentée sans tomber sous 0 alors P attend qu'une augmentation de la valeur de la sémaphore déclenche un nouveau test de sa part.

L'opération V quant à elle va incrémenter la valeur de la sémaphore ("Vas-y !"). En augmentant le nombre de ressources disponibles, elle indique du même coup à toute opération P en attente que ce qui était en attente peut désormais reprendre.

2.2.2 Exercice

Énoncé

1. Utilisez les sémaphores pour assurer une exclusion mutuelle.
   (a) Implémentez les fonctions suivantes :
      i. `initSem` initialisant une sémaphore
      ii. `P` acquérant une ressource
      iii. `V` libérant une ressource
   (b) Utilisez `P` et `V` pour assurer une exclusion mutuelle et résoudre le problème d'accès concurrentiel dans l'exercice précédent.
2. Utilisez les sémaphores pour créer une situation d'inter-blocage.

Solution

Sémaphores - Verrous d'exclusion mutuelle

Ce code permet de corriger une situation similaire à celle exposée en *Figure 1.*

Codes/semaphore–mutual–exclusion.c

```c
/** Lab Subject :
  * Mutual exclusion enforced by semaphores fixing race condition
  */

#include <stdio.h>      // Input/Output
#include <unistd.h>     // Time management (usleep)
#include <pthread.h>    // Threads management
#include <sys/sem.h>    // Semaphores management

#define N_THREAD 10
#define PERMS 0660  // -rw permissions for group and user

int account = 0;

int semId;

int initSem(int semId, int semNum, int initValue) {
    return semctl(semId, semNum, SETVAL, initValue);
}

/* An operation list is structured like this :
*   { semphore index, operation, flags }
* The operation is an integer value interpreted like this :
```

```
24 *   >= 0 : Rise the semaphore value by this value.
25 *        This trigger the awakening of semaphores waiting for a rise.
26 *   == 0 : Wait for the semaphore to be at value 0.
27 *    < 0 : Substract abs(value) to the semaphore.
28 *        If then the semaphore is negative, wait for a rise.
29 */
30
31 // Try to take a resource, wait if not available
32 int P(int semId, int semNum) {
33     // Operation list of 1 operation, taking resource, no flag
34     struct sembuf operationList[1];
35     operationList[0].sem_num = semNum;
36     operationList[0].sem_op = -1;
37     operationList[0].sem_flg = 0;
38
39     return semop(semId, operationList, 1);
40 }
41
42 // Release a resource
43 int V(int semId, int semNum) {
44     // Operation list of 1 operation, releasing resource, no flag
45     struct sembuf operationList[1];
46     operationList[0].sem_num = semNum;
47     operationList[0].sem_op = 1;
48     operationList[0].sem_flg = 0;
49
50     return semop(semId, operationList, 1);
51 }
52
53 void* deposit(void* nothing) {
54     int i;
55     for (i = 0 ; i < 10 ; i++) {
56         P(semId, 0);          // Take resource/semaphore 0 of semID
57         account = account + 10;
58         V(semId, 0);          // Release resource/semaphore 0 of semID
59         usleep(50*1000);   // Wait 50 ms
60     }
61     return NULL;
62 }
63
64 // Main function
65 int main(int argc, char* argv[]) {
66     int i;           // Iterator
67
68     // We create a set of 1 semaphore
69     // ftok generates a key based on the program name and a char value
70     // This avoid to pick an arbitrary key already existing
71     semId = semget(ftok(argv[0], 'A'), 1, IPC_CREAT | PERMS);
72
73     // Set the semaphore at index 0 to value 1 (= available for use)
74     initSem(semId, 0, 1);
75
76     pthread_t thread[N_THREAD];     // Array of threads
```

```
77
78      // Creating threads
79      for (i = 0 ; i < N_THREAD ; i++) {
80          pthread_create(&thread[i], NULL, deposit, NULL);
81      }
82
83      // Wait until threads are all complete
84      for (i = 0 ; i < N_THREAD ; i++) {
85          pthread_join(thread[i], NULL);
86      }
87      printf("Account/N_THREAD(%d): %d\n", N_THREAD, account/N_THREAD);
88      semctl(semId, 0, IPC_RMID, 0);      // Free the semaphore
89      return 0;
90  }
```

Codes/semaphore–mutual–exclusion.c

```
user@user-machine:~$ ./a.out
Account/N_THREAD(10): 100
```

Le problème d'accès concurrent a été réglé.

Sémaphores - Simulation d'inter-blocage

Le code suivant simule une situation d'inter-blocage similaire à celle exposée en *Figure 2* mais avec des sémaphores plutôt que des mutexes.

Codes/semaphore–deadlock.c

```
1  /** Lab Subject :
2    * Simulate a deadlock situation
3    */
4
5  #include <stdio.h>       // Input/Output
6  #include <unistd.h>      // Time management (usleep)
7  #include <pthread.h>     // Threads management
8  #include <sys/sem.h>     // Semaphores management
9
10 #define PERMS 0660  // -rw permissions for group and user
11
12 int semId;
13
14 int initSem(int semId, int semNum, int initValue) {
15     return semctl(semId, semNum, SETVAL, initValue);
16 }
17
18 /* An operation list is structured like this :
19 *    { semphore index, operation, flags }
20 * The operation is an integer value interpreted like this :
21 *    >= 0 : Rise the semaphore value by this value.
22 *         This trigger the awakening of semaphores waiting for a rise.
```

```
23 *    == 0 : Wait for the semaphore to be at value 0.
24 *     < 0 : Substract abs(value) to the semaphore.
25 *         If then the semaphore is negative, wait for a rise.
26 */
27
28 // Try to take a resource, wait if not available
29 int P(int semId, int semNum) {
30     // Operation list of 1 operation, taking resource, no flag
31     struct sembuf operationList[1];
32     operationList[0].sem_num = semNum;
33     operationList[0].sem_op = -1;
34     operationList[0].sem_flg = 0;
35     return semop(semId, operationList, 1);
36 }
37
38 // Release a resource
39 int V(int semId, int semNum) {
40     // Operation list of 1 operation, releasing resource, no flag
41     struct sembuf operationList[1];
42     operationList[0].sem_num = semNum;
43     operationList[0].sem_op = 1;
44     operationList[0].sem_flg = 0;
45
46     return semop(semId, operationList, 1);
47 }
48
49 void* funcA(void* nothing) {
50     printf("Thread A try to lock 0...\n");
51     P(semId, 0);          // Take resource/semaphore 0 of semID
52     printf("Thread A locked 0.\n");
53
54     usleep(50*1000);  // Wait 50 ms
55
56     printf("Thread A try to lock 1...\n");
57     P(semId, 1);          // Take resource/semaphore 1 of semID
58     printf("Thread A locked 1.\n");
59
60     V(semId, 0);          // Release resource/semaphore 0 of semID
61     V(semId, 1);          // Release resource/semaphore 1 of semID
62     return NULL;
63 }
64
65 void* funcB(void* nothing) {
66     printf("Thread B try to lock 1...\n");
67     P(semId, 1);          // Take resource/semaphore 0 of semID
68     printf("Thread B locked 1.\n");
69
70     usleep(5*1000);  // Wait 50 ms
71
72     printf("Thread B try to lock 0...\n");
73     P(semId, 0);          // Take resource/semaphore 1 of semID
74     printf("Thread B locked 0.\n");
75
```

```c
    V(semId, 0);          // Release resource/semaphore 0 of semID
    V(semId, 1);          // Release resource/semaphore 1 of semID
    return NULL;
}

// Main function
int main(int argc, char* argv[]) {
    int i;            // Iterator

    // We create a set of 2 semaphores
    // ftok generates a key based on the program name and a char value
    // This avoid to pick an arbitrary key already existing
    semId = semget(ftok(argv[0], 'A'), 2, IPC_CREAT | PERMS);

    // Set the semaphore at index 0 to value 1 (= available for use)
    initSem(semId, 0, 1);
    // Set the semaphore at index 1 to value 1 (= available for use)
    initSem(semId, 1, 1);

    pthread_t thread[2];     // Array of threads

    pthread_create(&thread[0], NULL, funcA, NULL);
    pthread_create(&thread[1], NULL, funcB, NULL);

    // Wait until threads are all complete
    for (i = 0 ; i < 2 ; i++) {
        pthread_join(thread[i], NULL);
    }
    printf("This is not printed in case of deadlock\n");
    // Free the semaphores
    semctl(semId, 0, IPC_RMID, 0);
    semctl(semId, 1, IPC_RMID, 0);
    return 0;
}
```

Codes/semaphore–deadlock.c

```
user@user-machine:~$ ./a.out
Thread B try to lock 1...
Thread B locked 1.
Thread A try to lock 0...
Thread A locked 0.
Thread B try to lock 0...
Thread A try to lock 1...
^C
```

Nous sommes obligé d'interrompre le programme pour l'empêcher d'attendre indéfiniment.