Lab5: Interprocess Synchronization

Subject

- Running a file : `gcc -pthread -o semaphore semaphore.c`
- Then use `./execName` to display results.

1- Concurrent Access To Shared Memory : Race Problems

If a memory variable is shared by different processes and these processes modify it concurrently, then this might lead to a final erroneous result ! The goal in the following exercise is to show these possible errors.

1. Using two tasks, create a shared variable ‘i’ and initialize it 65; one task should increment the variable and the other one should decrement it

I started by trying multiple tasks with processes but i didn't seem to had any issue with this code however i realize that when i start this program a high number of times i get strange result `65`, `66` and even `64`. We can see that there is a race problem, we can't be sure wich thread is going to finish his task first leading to unexpected results.

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

void *increment(void *received_data)
{
  int *data = (int*) received_data;
  (*data) = 66;
  pthread_exit(NULL);
}

void *decrement(void *received_data)
{
  int *data = (int*) received_data;
  (*data) = 65;
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  pthread_t thread[2];
  int i = 65;
  pthread_create( &(thread[0]), NULL, increment, &i);
  pthread_create( &(thread[1]), NULL, decrement, &i);
  printf("%d\n",i);
  pthread_join(thread[0], NULL);
  pthread_join(thread[1], NULL);
}
```

2. Explain why the following code could lead to an error.

```
Reg = i;
sleep(for_some_time);
// your choice Reg++ (or Reg--);
// depending on the task
i = Reg;
```

This code could lead to an error because if the data is accessed by 2 process/thread at the same time the result would not be good.

2- Solving the Problem : Synchronizing access using semaphores

1. Use semaphores to enforce mutual exclusion and solve the race problem in the first exercise (`sem_init` or `sem_open` , `sem_wait` , `sem_post`).

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

//global semaphore variable
sem_t semaphore;

void *increment(void *received_data)
{
  sem_wait(semaphore);
  int *data = (int*) received_data;
  (*data)++;
  sem_post(semaphore);
  pthread_exit(EXIT_SUCCESS);
}

void *decrement(void *received_data)
{
  sem_wait(&semaphore);
  int *data = (int*) received_data;
  (*data)--;
  sem_post(&semaphore);
  pthread_exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[]) {
  pthread_t thread[2];

  //Creating semaphore
  sem_init(&semaphore,0,1);

  int i = 65;
  pthread_create( &(thread[0]), NULL, increment, &i);
  pthread_create( &(thread[1]), NULL, decrement, &i);
  printf("%d\n",i);
  pthread_join(thread[0], NULL);
  printf("%d\n",i);
  pthread_join(thread[1], NULL);
}
```

With this version of threads, the race problem has been solved with semaphores. The increment will access to the data and close before the decrement turn.

What if we had more than two processes ? Is there something else to do to enforce mutual exclusion ? Explain and experiment using three processes.

If there are more than two processes we can't be sure which one of the three we can enforce mutual exclusion by adding empty/full.

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


sem_t *empty;
sem_t *full;

void *increment(void *received_data)
```

```
{
  sem_wait(&empty);
  int *data = (int*) received_data;
  (*data)++;
  sem_post(&full);
  pthread_exit(EXIT_SUCCESS);
}

void *decrement(void *received_data)
{
  sem_wait(&full);
  int *data = (int*) received_data;
  (*data)++;
  sem_post(&full);
  pthread_exit(EXIT_SUCCESS);
}
void *multiple(void *received_data)
{
  sem_wait(&full);
  int *data = (int*) received_data;
  (*data)*2;
  sem_post(&full);
  pthread_exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[]) {
  pthread_t thread[3];
  //Creating semaphore
  sem_init(&empty,0,1);
  sem_init(&full,0,0);

  int i = 65;
  pthread_create( &(thread[0]), NULL, increment, &i);
  pthread_create( &(thread[1]), NULL, decrement, &i);
  pthread_create( &(thread[2]), NULL, decrement, &i);
  printf("%d\n",i);
  pthread_join(thread[0], NULL);
  printf("%d\n",i);
  pthread_join(thread[1], NULL);
  pthread_join(thread[2], NULL);
}
```

2. A deadlock is a situation in which a process is waiting for some resource held by another process waiting for it to release another resource, thereby forming a loop of blocked processes ! Use semaphores to force a deadlock situation using three processes.

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

sem_t first;
sem_t second;
sem_t third;

void *increment(void *received_data)
{
  sem_wait(&second);
  int *data = (int*) received_data;
  (*data)++;
  sem_post(&first);
  pthread_exit(EXIT_SUCCESS);
}

void *decrement(void *received_data)
{
  sem_wait(&third);
  int *data = (int*) received_data;
  (*data)++;
  sem_post(&second);
```

```
  pthread_exit(EXIT_SUCCESS);
}
void *multiple(void *received_data)
{
  sem_wait(&first);
  int *data = (int*) received_data;
  (*data)*2;
  sem_post(&third);
  pthread_exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[]) {
  pthread_t thread[3];
  //Creating semaphore
  sem_init(&first,0,0);
  sem_init(&second,0,0);
  sem_init(&third,0,0);
  int i = 65;
  pthread_create( &(thread[0]), NULL, increment, &i);
  pthread_create( &(thread[1]), NULL, decrement, &i);
  pthread_create( &(thread[2]), NULL, multiple, &i);
  printf("%d\n",i);
  pthread_join(thread[0], NULL);
  printf("%d\n",i);
  pthread_join(thread[1], NULL);
  pthread_join(thread[2], NULL);
}
```

Here we have a deadlock situation, each thread is waiting for another thread to complete his task (1 waiting for 2 ; 2 waiting for 3 ; 3 waiting for 1).

3. Use semaphores to run 3 different applications (firefox, emacs, vi) in a predefined sequence no matter in which order they are launched.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <fcntl.h>
#include <semaphore.h>
#include <unistd.h>

sem_t first;
sem_t second;
sem_t third;

void *vim()
{
  system("/usr/bin/vim");
  sem_post(&first);
  pthread_exit(EXIT_SUCCESS);
}

void *firefox()
{
  sem_wait(&first);
  system("/usr/bin/firefox");
  sem_post(&second);
  pthread_exit(EXIT_SUCCESS);
}

void *emacs()
{
  sem_wait(&second);
  system("/usr/bin/emacs");
  pthread_exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[]) {
  pthread_t thread[3];
  //Creating semaphore
```

```
  sem_init(&first,0,0);
  sem_init(&second,0,0);
  sem_init(&third,0,0);

  pthread_create( &(thread[0]), NULL, vim, NULL);
  pthread_create( &(thread[1]), NULL, firefox, NULL);
  pthread_create( &(thread[2]), NULL, emacs, NULL);
  pthread_join(thread[0], NULL);
  pthread_join(thread[1], NULL);
  pthread_join(thread[2], NULL);
}
```

4. Use sempahores to implement the following parallelized calculation `(a+b)*(c-d)*(e+f)`

- T1 runs `(a+b)` and stores the result in a shared table (1st available spot)
- T2 runs `(c+d)` and stores the result in a shared table (1st available spot)
- T3 runs `(e+f)` and stores the result in a shared table (1st available spot)
- T4 waits for two tasks to end and does the corresponding calculation
- T4 waits for the remaining task to end and does the final calculation then displays the result

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <fcntl.h>
#include <semaphore.h>
#include <unistd.h>

sem_t finish;
sem_t order;

typedef struct
{
  int a, b, c, d, e, f;
  int order;
  int tab[3];
  int result;
} thread_data;

void *T1(void *received_struct)
{
  thread_data *data = (thread_data *)received_struct;
  int temp = data->a + data->b;
  sem_wait(&order);              //Part of code that can't be accessed  by multiple threads at the same time
  data->tab[data->order] = temp; //The first available spot in the array is filled by the result
  (data->order)++;               //Increase to know the new first available spot
  sem_post(&order);              //Release of this part
  sem_post(&finish);             //Tell others threads that the calculation is complete
  pthread_exit(EXIT_SUCCESS);
}

void *T2(void *received_struct)
{
  thread_data *data = (thread_data *)received_struct;
  int temp = data->c + data->d;
  sem_wait(&order);
  data->tab[data->order] = temp;
  (data->order)++;
  sem_post(&order);
  sem_post(&finish);
  pthread_exit(EXIT_SUCCESS);
}

void *T3(void *received_struct)
{
  thread_data *data = (thread_data *)received_struct;
  int temp = data->e + data->f;
  sem_wait(&order);
  data->tab[data->order] = temp;
  (data->order)++;
```

```
  sem_post(&order);
  sem_post(&finish);
  pthread_exit(EXIT_SUCCESS);
}

void *T4(void *received_struct)
{
  thread_data *data = (thread_data *)received_struct;
  int temp;
  sem_wait(&finish);                  //Wait for the first thread to finish
  sem_wait(&finish);                  //wait for the second thread to finish
  temp = data->tab[0] * data->tab[1]; //Calculation
  sem_wait(&finish);                  //Wait for the last thread to finish
  data->result = temp * data->tab[2]; //Complete calculation
  pthread_exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
  pthread_t thread[4];
  thread_data data;
  data.order = 0;
  //Creating semaphore
  sem_init(&order, 0, 0);
  sem_init(&finish, 0, 0);
  pthread_create(&(thread[0]), NULL, T1, &data);
  pthread_create(&(thread[1]), NULL, T2, &data);
  pthread_create(&(thread[2]), NULL, T3, &data);
  pthread_create(&(thread[3]), NULL, T4, &data);
  pthread_join(thread[0], NULL);
  pthread_join(thread[1], NULL);
  pthread_join(thread[2], NULL);
  pthread_join(thread[3], NULL);
}
```