Lab Worksheet 04 - Semaphores

Objectives

1. Program basic data exchanges via shared memory.
2. Program synchronized exchanges with semaphores.

Remarks

- **Always** clean up your shared memory objects -- segments and semaphores! It is critical because the kernel ignores them until it needs to reallocate the memory they occupy. By default, reopening a semaphore that already exists does not set its counter to the initialization value, so your code might not behave as expected if you run the same program multiple times without cleaning up at the end of each run.
- This is the point where you reach the limits of standard specifications: OS X and Linux/Unix/Cygwin handle shared memory objects in different ways.
  - OS X does not support unnamed semaphores. If you're using an Apple computer, either limit your code to named semaphores or use the Linux VM.
  - Linux/Unix/Cygwin provides a location in the file system namespace to manipulate shared memory objects from the terminal: they are kept in `/dev/shm/` and, given the right permissions, can be deleted with the `rm` command. The philosophy of OS X is to leave no such location directly exposed, and expects your code to perform all the proper `unlink` calls.
  - OS X links the POSIX real-time (RT) library *by default*, but Linux/Unix/Cygwin does not. In the latter case, since shared memory is part of the RT specification, you need to add option `-lrt` at the end of your compilation command.

    eg. `$gcc -o test -Wall my-code-with-shared-objects.c -lrt`

Exercise 1: First steps with shared memory

Write a program where the initial parent process creates a segment of shared memory that can hold N integer values, then creates N child processes and waits for their termination. Each child generates a random value random_val, stores it in shared memory and displays it before exiting. The random value is generated thus:

```
random_val = (int) (10 * (float) rand () / RAND_MAX);
```

Once all its children have terminated, the parent process reads all the values in shared memory, sums them up and then displays the result.

Exercise 2: Synchronizing accesses to shared memory

Extend the program written for exercise 1 so that, instead of generating only one value, every child process generates N values. The parent process now sums up N * N values. Since the segment of shared memory can hold at most N integers, you now need to synchronize accesses.

Exercise 3: Barrier

The program below creates N-1 child processes. All N processes carry out `calc1()` and then `calc2()` concurrently.

```
void calc1 () {
     int i;
     for (i = 0; i <1E8; i ++);
}

void calc2 () {
     int i;
     for (i = 0; i <1E8; i ++);
}

int main (int argc, char * argv []) {
     int i = 0;
     pid_t pid_child [2];

     while ((i < N-1) && ((pid_child [i] = fork ())! = 0))
          i ++;

     calc1 ();
     calc2 ();
     printf ( "End Process %d \n", i);

     EXIT_SUCESS return;
}
```

We want to change the program by using semaphores to make sure that no process begins `calc2` before all other processes have also finished `calc1`.
Write the program `barrier` that implements this new specification.