Assignment 5: Semaphores

The Producer-Consumer Problem

In class we covered the standard producer-consumer problem and saw how conflicts can arise when trying to access a common buffer that is common to several processes. Using if statements to determine whether or not a given buffer is full or empty is not sufficient and can result in a race condition in which all of the processes end up sleeping. This is remedied by using semaphores, which allow for exclusive access to the buffer by a given processor at a given time, while ensuring atomic action when determining whether or not a buffer is empty or full. While semaphores help, it is still possible for a group of processors to come to a stand still in a dead lock condition if semaphores are not used carefully.

The use of semaphores was shown in class using the following pseudocode, assuming that the type `semaphore` allows the given variables to be visible as semaphores by all the processes that have access to the producer-consumer buffer, and that the variable `N` is the maximum number of items allowed in the buffer at any time:

Producer

```
semaphore mutex=1, full=0, empty=N;

while(true) {
  produce(&item);
  down(empty);
  down(mutex);
  enteritem(item);
  up(mutex);
  up(full);
}
```

Consumer

```
semaphore mutex=1, full=0, empty=N;

while(true) {
  down(full);
  down(mutex);
  removeitem(&item);
  up(mutex);
  up(empty);
```

```
  consumeitem(item);
}
```

The Producer-Consumer functions

You should get started with the program `main.c`, which uses the producer-consumer functions defined in `prod_cons.c` to initialize a buffer with the number of processors defined at the command line, as in

```
$ ./prodcons 10 15 20
```

which will fork 15 processes with a buffer size of 10 and a maximum time of 20 (which is not used in this program but you will need to use when you implement the producer-consumer problem). You can find out what the functions are and what they do by reading the file `prod_cons.h`. In the `main.c` program, each forked process adds 1 item to the buffer and subsequently removes it, after waiting for `TSLEEP` microseconds. The parent process then waits for its children to exit and prints the number of items remaining in the buffer.

Your assignment

Part 1

When you compile and run the sample program `main.c`, you can change the amount of time each process waits between production and consumption events by changing the `#define TSLEEP 1e3` line. When `Nitems=10`, `Nprocs=15`, and `maxtime=20` (redundant for the test program), run the program as

```
$ ./prodcons 10 15 20
```

the program will fail with either a segmentation fault or something like

```
Error in RemoveItem: Empty buffer!
```

Run the program a few times with `TSLEEP` set to `1e3` (microseconds) and then run it again with `TSLEEP` set to `1e6` (microseconds) and explain when and why it fails for each case. Then discuss why increasing the sleep time decreases the likelihood of a failure, and discuss why not all of the processes get a chance to print the results to the screen, regardless of whether or not the program exits prematurely due to an error. Write your answers in a file called `README`.

Part 2

Implement the producer-consumer problem using the functions defined in `prod_cons.c` but use semaphores to decrease the likelihood of a conflict when trying to write to/read from the buffer. You should create a file called `sem.c` in which you define the following functions

- `int NewSemaphoreSet(int proj_id, int nsems);` This function should take the `proj_id` supplied to the function `ftok` used to create a key for the semaphore set and should return the semaphore id that corresponds to a new semaphore set with `nsems` semaphores. You can use the parent process id returned in the buffer after the call to

  ```
  pid=InitializeBuffer(&buffer,Nitems,Nprocs);
  ```

  as the project id, as in

  ```
  semid=NewSemaphoreSet(buffer->pids[0],nsems);
  ```

  since `buffer->pids[0]` stores the pid of the parent process. You can create additional unique semaphore ids by using the other pids in the array. For example, to create an additional semaphore set with `Nprocs` elements, you would use

  ```
  semidp=NewSemaphoreSet(buffer->pids[1],Nprocs);
  ```

- `int DestroySemaphore(int semid);` This should remove the semaphore set associated with the semaphore id `int semid` and should return the result from the return of using the function `semctl`.

- `int up(int semid, int semnum);` This should add 1 to the semaphore number `semnum` in the semaphore set defined by the id `semid` and should return the value returned by using the `semop` function. For the semaphore operation flag `sem_flg`, use `!SEM_UNDO`.

- `int down(int semid, int semnum);` This should subtract 1 from the semaphore number `semnum` in the semaphore set defined by the id `semid` and should return the value returned by using the `semop` function. For the semaphore operation flag `sem_flg`, use `!SEM_UNDO`.

  Note: in the `down` and `up` functions, you should be sure to use the `usleep` function to make sure other processes have time to catch the semaphore change before proceeding. You may assume a sleep time of `usleep(SLEEPTIME);`, with `#define SLEEPTIME 1000`, which causes the `up` and `down` functions to sleep for 1 millisecond after the calls to `semop`. (use `#include<unistd.h>` as well)

You should design your producer-consumer problem so that the only producer is the master or parent process, and the consumers are the children. The parent process should produce a fixed number of items, which is given by `maxtime` and is an input argument at the command line. Since the `InitializeBuffer` function returns a `0` for the parent thread of execution, you can design the producer-consumer loops as

```
int time=0;
if(pid) {
  while(true) {
   ...
```

```
  }
} else {
  while(time++<maxtime) {
  ...
  }
}
```

When the parent is done, you should send the `KILL` signal to the child processes using the `pids` array only **after** you are sure that the number of items in the buffer is empty (use `semctl`, not `GetCount`). After doing this, you should then print out the statistics of how many items each child consumed and how many items the parent produced, as well as the total number of items consumed by all of the children. As an example, a call to your program with

```
$ ./prodcons 10 15 100
```

should output **nothing but**

```
Access stats:
Parent (16292) Produced: 100
Child 1 (16293) Consumed: 8
Child 2 (16293) Consumed: 8
Child 3 (16293) Consumed: 7
Child 4 (16293) Consumed: 7
Child 5 (16293) Consumed: 7
Child 6 (16293) Consumed: 7
Child 7 (16293) Consumed: 7
Child 8 (16293) Consumed: 7
Child 9 (16293) Consumed: 7
Child 10 (16293) Consumed: 7
Child 11 (16293) Consumed: 7
Child 12 (16293) Consumed: 7
Child 13 (16293) Consumed: 7
Child 14 (16293) Consumed: 7
Total Consumed: 100
```

This way if the total consumed is equal to the total produced then you know your program is working. This will require that you create two semaphore sets. The first one stores the `mutex`, `empty`, and `full` semaphores, which you should access using an enumerated type. The second semaphore set should contain `Nprocs` semaphores and each should store the number of items consumed (or produced) by each child (or parent).

Part 3

Once you have your program working, run it with `Nitems=10`, `Nprocs=20`, and `maxtime=100` and study how the value of `SLEEPTIME` used with `usleep` in your `up` and `down` functions affects the likelihood that your program gives the correct results or causes a race or deadlock

condition to occur. Be very specific about describing where and why these conditions might occur in your program. Discuss whether or not these conditions are ever completely avoidable in this example program. Write your answers in the `README` file.

Getting started

You should copy the following files to your directory:

```
/home/cos315/assignments/assignment5/main.c
/home/cos315/assignments/assignment5/prod_cons.c
/home/cos315/assignments/assignment5/prod_cons.h
```

Your job is to edit the file `main.c` and create the files `sem.h` and `sem.c` which contain definitions of the semaphore functions used in your producer-consumer implementation. You need to create your own `Makefile`. You can answer part 1 by compiling the program as it is with

```
$ gcc -o prodcons main.c prod_cons.c
```

As an example of the output you should get for part 2, you can run the program

```
/home/cos315/bin/prodcons
```

Deliverables, due date, and grading

You will be changing the `main.c` file and creating the files `sem.c` and `sem.h` which contain your semaphore functions. You will also be creating a `README` file and your own `Makefile`. Place these into a directory and submit that directory using the `submit` command by **5:00 pm** on **Monday, 02 June, 2003**.

Grading

Out of 100 points, grading will be as follows:

1. 0 if your program does not compile without errors or warnings.
2. 10 if it does compile and link error free but crashes upon running.
3. 20 if it runs and exits cleanly but does not give the right answer.
4. 30 if it runs correctly.
5. 30 points for your answers to parts 1 and 3.
6. If it runs correctly, the remaining 10 points will be assigned according to your programming style. Your style will not be graded if your program does not run correctly.

A note about stray message queues and semaphore sets

When you are debugging it is common for your program to leave unwanted message queues and semaphore sets in memory. Please remove them periodically with the commands

```
/home/cos315/bin/removeallsems
/home/cos315/bin/removeallqueues
```

Don't worry about the errors these functions return. These result when you try to remove semaphore sets or message queues that don't belong to you. You should also remove the temporary buffer files in your code directories with

```
$ rm -f .prodcons*
```

but make sure you don't have any files called `.prodcons*` first!