G22.3250: Honors Operating Systems
Handout #2: Programming Project 1 (8 points)
Due: March 02, 1999

This project exercises concepts from CPU scheduling, interprocess communication (IPC), process synchronization, and language support for concurrency.

Particulars

For this assignment, you will write programs that perform the necessary synchronization for the following problem (Problem 6.8, Silberschatz and Galvin):

The Cigarette-Smokers Problem: Consider a system with three *smoker* processes and one *agent* process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table (using a random albeit fair selection procedure). The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the ingredients, and the cycle repeats.

You will express the necessary synchronization using primitives that you will first develop for monitor-based synchronization and message-passing based IPC. Your programs will be written in the C programming language (C++ is fine) and can use any machine where the operating system provides support for shared memory segments and semaphores. I would recommend, however, that you do this project using the department SUNs. **Note that the only machines you should use are** `courses1.cs.nyu.edu` **and** `courses2.cs.nyu.edu`**.**

To help you get started with the assignment, I have provided a file, *os.h*, in my home directory (`~vijayk/OS/homework1/`) accessible from the department SUNs. This file contains definitions for the following routines (valid only on Solaris):

- `void initenv(int size)`
  initializes the "environment". Here `size` gives the maximum amount of shared memory that will be required.
- `void *shmalloc(int size)`
  allocates a region of `size` bytes from shared memory.
- `struct semaphore_t;`
  `void seminit( struct semaphore_t **s, int i );`
  allocates the memory (in shared space) required to support a semaphore, initializes it to the value `i`, and stores the pointer to it in the variable `s`.
- `void semwait( struct semaphore_t * );`
  does a **P** operation on a semaphore.
- `void semsignal( struct semaphore_t * );`
  does a **V** operation on a semaphore.
- `void createprocess( void (*f)(void) );`
  creates a process which executes the function `f`. Note that all shared-memory allocations and semaphore initializations (as well as the monitor and message port initializations below) should be completed

before any `createprocess` call. Note also that anything that needs to be visible in the created processes must be allocated in shared memory. One way of doing this is defining a structure containing all the "global" variables, and then allocating this structure using `shmalloc`.

These definitions make use of Solaris support for shared memory segments and message queues to implement the shared memory and semaphore abstractions. Note that an abnormal termination of your processes may not free up the allocated shared memory segments and message queues. To ensure that you do not run out of these scarce system resources, please make sure that these are explicitly freed. The program `os-cleanup.c`, also available in the above subdirectory, does the needful.

Given this background, there are three parts to the assigment which are described below.

1. **Monitor-based Synchronization**
   You will first implement the following primitives for monitor-based synchronization using the provided shared memory and semaphore primitives. The following API defines a Brinch-Hansen monitor with one or more delay queues (these are equivalent to condition variables):
   - `struct monitor_t;`
     `void moninit( struct monitor_t **m, int i );`
     allocates the memory required to support a monitor with `i` queues, initializes it, and stores the pointer to it in the variable `m`.
     I would suggest using the following definition for the monitor structure:
     ```
     struct monitor_t {
       struct semaphore_t mutex;
       struct queue_t *qs;
     };
     struct queue_t {
       int count;
       struct semaphore_t delay;
     };
     ```
     The idea is that the monitor semaphore, `mutex`, would regulate entry and exit into the monitor, and each of the delay semaphores would be used to suspend processes waiting on different condition variables. See the notes from Lecture 9 for a discussion of how this is achieved.
   - `void menter( struct monitor_t * );`
     each external procedure executes this upon entry into the monitor.
   - `void mexit( struct monitor_t * );`
     each external procedure executes this upon exit from the monitor.
   - `void monwait( struct monitor_t *, int i );`
     suspends the process in queue `i` of the monitor. The logic for this function is as follows:
     ```
     increment count of processes in the queue
     signal the mutex semaphore
     wait on the delay semaphore
     ```
   - `void monsignal( struct monitor_t *, int i );`
     checks to determine whether or not queue `i` is empty. If it is, then the monitor is just exited, otherwise one process is released from queue `i`. In either case, the process executing the `monsignal` is assumed to have exited the monitor. The pseudocode for the function is:
     ```
     if ( the queue is empty )
       signal the mutex semaphore
     else
       decrement the number of delayed processes
       signal the delay semaphore
     ```

Now implement the Cigarette-Smokers Problem using these monitor primitives. Implement the action of smoking using `sleep` calls. Plot a graph showing the number of occasions each of the smokers gets to smoke a cigarette as a ratio to the total number of occasions. One would expect that as the program runs for a long duration of time, and the agent process picks each of the 3 combinations in a fair fashion, then each smoker would asymptotically smoke on one-third of the occasions.

2. **Message Passing-based IPC**
   In this part of the assignment, you will implement the synchronization required for the Cigarette-Smokers Problem using message-passing IPC. Recall from Lecture 4 that in the message-passing IPC style, each process executes a dispatch loop where it sends messages, waits for incoming messages, and then acts upon the received message. Since message receives are blocking, they can serve as an effective synchronization mechanism.

   In order to do this part of the assignment, you will first implement the following primitives for message ports using the monitor primitives you have implemented in the first part. For this assignment, you only need to statically assign ports to processes; i.e., you will allocate a certain number of ports and each of your processes would know which port it needs to send to (or receive from).

   - `struct msg_t;`
     `struct messageport_t;`
     `msgportinit( struct messageport_t **mp );`
     Given definitions of `msg_t` and `messageport_t` structures, `msgportinit` allocates the memory required to support a message port with unbounded capacity, initializes it, and stores the pointer to it in the variable `mp`.

     I would suggest implementing the message port structure as a monitor with the following structure definitions:

     ```
     struct msg_t {
       struct msg_t *link;
       int length;
       char data[100];
     };
     struct messageport_t {
       struct msg_t *head, *tail;
       struct monitor_t *m;
     };
     ```

     The initialization routine would then initialize `m` to be a monitor with 1 queue, with `head` initially being NULL. Processes receiving from an empty message port would then block on the delay semaphore associated with the monitor queue.

   - `void sendtoport( struct messageport_t *, struct msg_t * );`
     A non-blocking primitive which stores a message into the specified message port. You can assume that the message need not be copied (of course, your application program should then not try to use or free this message structure). The logic for this function is as follows:

     ```
     menter( monitor )
     add msg to end of message queue
     monsignal( monitor, 0 )
     ```

- `void recvfromport( struct messageport_t *, struct msg_t ** );`
  A blocking primitive to receive a message from the specified message port. You can assume that the message need not be copied. The logic for this function is as follows:

```
menter( monitor )
if ( no messages in the queue )
  monwait( monitor, 0 )
take the first message from the queue
mexit( monitor )
```

Now implement the Cigarette-Smokers Problem using message-passing. Note that you may need to create an additional process that models the table (more specifically, the constraint that the agent places two items on the table and cannot proceed unless these items have been picked up by one of the smoker processes).

3. **Scheduling**
   In this part of the assignment, you will implement a proportional-share scheduler to control the queuing of processes in the monitor implementation of Part 1.

   Assume that you want to solve a modified Cigarette-Smokers Problem. The new problem has *three* agent processes, each of which are capable of producing a particular configuration of two items (tobacco and matches, matches and paper, and paper and tobacco). The additional constraint that is imposed is that one of the smoker processes (say, the one who holds the matches) should get twice as many chances to smoke as either of the other two smoker processes.

   Since each agent process is assumed to originally have identical behavior (modulo the different items they produce), the only way to enforce the above constraint is to control the way in which these processes are allowed to enter and exit the monitor. One way of controlling this is to use proportional-share scheduling (e.g., lottery scheduling) in the delay queue structure, changing it so that a specific process (or a class of processes) is more likely to be awakened as compared to the others. To do this, change the implementation of the queue structure in Part 1 to be the following:

```
struct queue_t {
  int count;
  int *weights_array;
  struct semaphore_t *delay_array;
};
```

   The idea is to delay processes belonging to different proportional-share classes on their own semaphores instead of on the same semaphore. Given this structure, the constraint can be implemented by modifying the `monwait` and `monsignal` functions to implement a policy similar to lottery scheduling. You will also need to come up with a simple API for specifying different shares, and associating processes with a share class. One possibility is that the `monwait` and `monsignal` functions both take an additional argument that denotes the share class of the calling process.

   I would recommend that you implement the *simplest* policy that allows you to solve the modified Cigarette-Smokers Problem. You should be able to verify your solution by measuring the number of turns for each smoker and drawing plots similar to the ones for Part 1. You should observe that the distinguished smoker process does actually get twice as many occasions to smoke as the others.

Guidelines

Although the project may appear to be very involved, in reality the amount of code you have to write is very small and relatively straightforward. The part that will need the most work is developing monitor- and message-passing based solutions to the Cigarette-Smokers Problem, and figuring out how proportional-share scheduling can help solve the modified Cigarette-Smokers problem. All of this is something you can do without writing a single line of code, and my suggestion would be to first work out these solutions on paper.

I would also advice you to start this project as early as possible. Although I have tried to provide virtually all the logic required to implement the primitives, I am sure that you will run into some problems that I have not foreseen.

You are free to discuss problem clarifications and possible solution approaches with each other; however, I will expect that the programs and solutions you hand-in will be your own.

Please hand in source code listings for the primitives and your solution to the synchronization problem. Please also include the plots (for Parts 1 and 3) showing the relative rates of progress for the three smoker processes.