Assignments 4 and 5:

Monitors, Semaphores, and Shared Memory
Recitation for ECES 338
February 22, 1999

Additional Notes About Monitors

In Assignment #4, you are asked to complete three different synchronization problems, using *Monitors*. Monitors are covered in the Dinosaur book, on pages 181 through 189. Unfortunately, I don't feel that Sliberschatz and Galvin provide the best explanation possible, so I am going to elaborate a bit upon what they have written. Tekin has a book that you may borrow, entitled "Concurrent Programming: Principles and Practice" by Gregory Andrews. It provides a more thorough evaluation of Monitors (although their syntax is at times somewhat weird). I've looked through their material, and going to provide the highlights here:

- Monitors are program modules similar to *critical regions*, with several key differences. A monitor is like an *object*, in that it contains local variables and functions. The local variables are shared throughout all instances of the given monitor, but may only be accessed by the functions within the monitor. Furthermore, all functions within the monitor operate in a mutually-exclusive manner. Thus, only one process can be active within the monitor at time.

- However, further synchronization might be necessary, which is why *condition variables* have been provided. To quote Andrews: "A *condition variable* is used to delay a process that cannot safely continue executing until the monitor's state satisfies some boolean condition. It is also used to awaken a delayed process when the condition becomes true" (Andrews, 266). As described in the *Dinosaur* text, the signal and wait operations are supported on condition variables. However, Andrews enlightens us to several differences that exist between condition variables and semaphores:

  - "**Signal** has no effect if no process is delayed on the condition variable; the fact that it was executed is not remembered" (Andrews, 267).
  - "**Wait** *always* delays a process until a later **signal** is executed" (Andrews, 267).
  - "Third, the process executing **signal** always executes (at least in the monitor) before a process awakened as a result of the **signal**" (Andrews, 267).

- Finally, on page 183, the *Dinosaur* text describes an inherent conflict between condition variables and the mutual-exclusive nature of the monitor. Basically, a problem occurs once a condition is signaled – if the waiting processes awakens immediately, then two processes would be operating concurrently in the monitor.

There are many different ways to solve this problem – choose one, and state choice in your assumptions.

Semaphores

- `int semget (key_t key, int nsems, int semflg)`
  - This function allocates an array of system-wide semaphores. The `key` parameter is used by `semget()` in order to allocate a unique identifier for the set of semaphores. Using this same `key` in subsequent calls to `semget()` will allow other processes to use the same set of semaphores. The `nsems` parameter specifies the number of semaphores to allocate. Finally, the `semflg` parameter specifies the permissions that the new set of semaphores should have after creation. The flags are documented in the *"Monkey book"*.

    Example:

```
key_t key;
int semid;

semid = semget (key, 3, IPC_CREAT | IPC_EXCL | 0600);
if (semid == -1)
        {
        perror ("Semget() | Process x");
        exit (50);
        }
```

    This call to `semget()` will create an array of three semaphores. The `IPC_CREAT` flag will cause `semget()` to create the semaphores if they do not already exist. The `IPC_EXCL` flag will cause `semget()` to fail if the value specified in `key` already exists. Thus, these two flags will ensure that an *"entirely new"* set of semaphores is created. The final part of the flag is the permission bits. A successful call will return a semaphore identifier, which you will need to use later on in further semaphore operations.

- `key_t ftok(const char *path, int id)`
  - Use the `ftok()` function in order to generate a key for any of the IPC functions. You don't really need to understand how this function works, and it is fine to just use the exact same call that the book uses.

- `int semctl (int semid, int semnum, int cmd, /* union semun arg */)`
  - This function implements a variety of operations on a set of semaphores. This first parameter is an identifier (which you saved from your call to `semget()`, right?). The second parameter is a number, which represents the number of semaphores in the set upon which to carry out the given operation. Finally, the third parameter specifies a command. Depending upon this command, further parameters may be needed. In general, this function can be used in order to access the internal data that accompanies UNIX System V semaphores (as described in the `semget()` section of the "Monkey book"), or to carry out specific operations on the set of semaphores. It is important to note that the definition for the `semun` union **IS**

NOT included in the Solaris header files. This means that in order to commands that require this data structure, you must define it in yourself as follows:

```
union semun {
        int val;
        struct semid_ds *buf;
        ushort *array;
        };
```

Now, there are really only two different operations that you will need to use for your programming assignments. The first one, `SETALL`, is used to initialize your semaphores. The second command, `IPC_RMID`, is used to remove your set of semaphores from the system. I will present an example of each command.

Example of `SETALL`:

```
ushort start_val[3];    /* Initial values for Semaphore. */
union semun ctl_arg;    /* Argument union for semctl(). */

start_val[0] = 1;
start_val[1] = 1;
start_val[2] = 0;
ctl_arg.array = start_val;      /* Set address of array */
                                /* in the semun union. */

if (semctl (semid, 0, SETALL, ctl_arg) == -1)
        {
        perror ("Semctl | Process x");
        exit (51);
        }
```

This example is fairly straight-forward, with a few exceptions. Firstly, it is very important that your array of values (to be used to initialize your semaphores) be of type `ushort`. The proper values will not be set if you use any other type. Finally, the `nsem` parameter to `semctl()` can be set to zero, so that all of the semaphores in the array are initialized. If you only want to initialize a subset of your semaphores (starting from the first one), I believe that you can specify this behavior by putting a non-zero value in for this parameter.

Example of `IPC_RMID`:

```
if (semctl (semid, 0, IPC_RMID, 0) == -1)
        {
        perror ("Semctl | Process x");
        exit (52);
        }
```

- `int semop (int semid, struct sembuf *sops, size_t nsops)`
  - Finally, we come to the function that allows the basic *wait* and *signal* operations to be performed on a set of semaphores. Unfortunately, UNIX System V semaphores are very generalized, so this function is rather cumbersome to use. All access to a semaphore is done via a `sembuf` structure. In general, operations that decrement a semaphore attempt to *gain access to a resource* (wait). Operations that increment a semaphore attempt to *release access to a resource* (signal). In

order to facilitate these operations, the book defines the following instances of the `sembuf` structure:

```
struct sembuf acquire = {0, -1, SEM_UNDO},
              release = {0,  1, SEM_UNDO};
```

Using these variables in order to implement *wait* (`acquire`) and *signal* (`release`) is fairly easy. I will present two examples, demonstrating both operations.

Example of "Wait":

```
acquire.sem_num = 0;
if (semop (semid, &acquire, 1) == -1)
        {
        perror ("Semop | Process x | Acquire resource 0");
        exit (53);
        }
```

Example of "Signal":

```
release.sem_num = 0;
if (semop (semid, &release, 1) == -1)
        {
        perror ("Semop | Process x | Release resource 0");
        exit (54);
        }
```

These examples demonstrate how the *ugliness* of `semop()` can be abstracted to some extent. You may also find it useful to create `wait()` and `signal()` functions, in order to reduce this complexity (and code reuse) even further.

Shared Memory

In general, once you implement semaphores, the functions necessary in order to implement shared memory will seem very easy. I will present a brief overview of these functions in the following section.

- `int shmget (key_t key, int size, int shmflg)`
  - This function is practically identical to `semget()`. Basically, you give `shmget()` a `key` value, a `size`, and the requisite `flags`, and it will allocate said number of bytes in the system. It will return an identifier to this area of memory, that you will need to save for use with other system calls.

- `int shmctl (int shmid, int cmd, struct shmid_ds *buf)`
  - Again, this function is used to carry out a variety of operations on a given shared memory segment. The only command that you will need for your assignment is `IPC_RMID`, which removes the shared memory segment from the system.

- `int shmat (int shmid, void *shmaddr, int shmflg)`
  - This function is unique to shared memory. Basically, once a shared memory region has been created, each process that wishes to use the shared memory must first *attach* to it. Whereas with semaphores this attachment could be facilitated through the `semget()` system call, in shared memory the `shmat()` system call

must be used instead. The first parameter is the identifier for the shared memory region, and the other two parameters control how the process attaches the shared area of memory to its address space. It is always best to leave these two parameters as zero.

Example:

```
int *shrd_var;   /* Affords local access to the shared */
                 /* memory region. */

if ((shrd_var = (int *) shmat (shmid, 0, 0)) == (int *) -1)
        {
        perror ("Shmat | Process x");
        exit (3);
        }
```

- `int shmdt (void *shmaddr)`
  - This function detaches a process from a shared memory region. It is fairly simple to use – the only parameter is the variable that you wish to detach.

Example:

```
shmdt ((void *)shrd_var);
```