Process Synchronization **Due: 17 April**

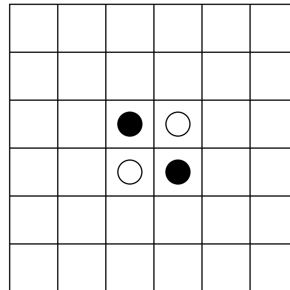
1 Introduction

Synchronization between multiple processes is fundamental to building modern software. We see situations where synchronization is needed in client/server software, multi-threaded implementations of programs, and programs implemented on parallel processing machines.

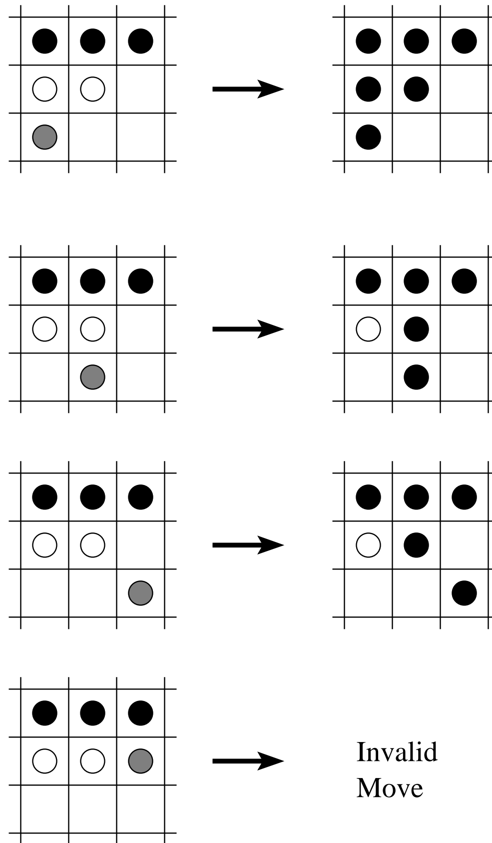
In this assignment, you will use the SYSV implementations of semaphores and shared memory to implement a game similar to Othello or the Iagno game that is available on some Linux installations. The SYSV semaphores and shared memory implementations are part of the SYSV IPC (InterProcess Communication) suite available on most modern versions of Unix, including Linux and FreeBSD.

2 The Game

The game is played on a board that is an 6×6 grid. The players move by placing pieces in the squares on the board. The initial configuration of the board looks like the following where the pieces are shown as black and white circles.



The player whose turn it is must place their piece in a location that is adjacent to a piece of the opponent such that along in the direction of the opponent's piece there is a sequence of one or more pieces such that a piece of the player occurs. A move turns all opponent's pieces along such a path into the player's piece. The following diagram shows some possible moves for the black player. On the left is the board when the move is made, and the grey circle shows the black player's move.



Notice in the first scenario, several pieces are changed from white to black. A player must make a valid move, and so, if no valid move exists the player must skip its turn.

3 Specification

You are to implement a game using two programs, which you will run as three processes.

- `play` — the primary program, which sets up shared memory for maintaining the game state, semaphores for controlling access to the game state, creates the player processes and controls the play of the game. Executing this program starts the game.
- `player` — a player program, chooses the next play based on the current board and the strategies described below.

3.1 Play

You are to use `fork()` and some form of `exec()` to create child `player` processes. The program should use semaphores to signal the turn for each player. When the child is done with its play it will signal the `play` process. The `play` process will then check whether the play is valid. If the play is valid, the process will update the board, and signal the next

`player` process. If the play is invalid, the last player will be signaled to choose another play. The process should also check if the player has a valid move, and if not it loses its turn. The game is over when both players no longer have a valid move.

The `play` program should take two command line arguments that identify the strategies for the two players. These arguments should be passed on to the two `player` processes. The tokens for the strategies are given in the description of the `player` program below.

When the game is over, the `play` program should print a message indicating the number of pieces belonging to each player.

3.2 Player

The `player` program should wait for its turn to be signalled through a semaphore. The process should then choose its next move based on the history of its previous valid moves and the board in shared memory. When the move has been chosen, the move should be printed to standard output, and communicated through shared memory to the `play` process. You should come up with a scheme to make sure the `player` process ends when the game is over.

The choice of the next move is made according to a strategy that is determined by a parameter to the program. The strategies that you should implement include:

- `first` — search for a move starting from the upper left and moving to the right, and starting at the left of each row.
- `last` — search for a move starting from the lower right, moving left and up.
- `max` — choose the location that has the most opponent pieces.

The program should take one argument which must be one of `first`, `last`, `max`.

4 Output

All output should go to standard output. Messages should be printed when:

- a player's turn is started (indicate which player),
- a move is chosen (indicate location),
- a move is invalid (indicate player and location), and
- the game is over (indicate number of pieces for each player).

You should feel free to print other messages.

To collect your sample output you can use the `script` utility instead of redirecting to a file.

5 Semaphores

The actual implementation of semaphores in Linux, SYSV semaphores, differ from what was covered in class. The following sections contain information on using SYSV semaphores and some guidance on how to map them to the implementations discussed in class.

5.1 Preliminaries

In order to use semaphores in your code, the following compiler directives are needed:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

These header files define the types and contain function prototypes for using semaphores. In the SYSV semaphore implementation, each function call manipulates a **set** of semaphores that can be used together in flexible and complicated ways. Instead of trying to use this functionality, it is suggested that you make the set contain only one semaphore, and write functions to implement the operations we discussed in class.

5.2 Creation and Accessing

The `semget` function is used to obtain a semaphore (set) from the operating system. The function prototype is

```
int semget ( key_t key, int nsems, int semflg );
```

where `key` is an identifying key that you provide for the semaphore, `nsems` is the number of semaphores in the semaphore set, and `semflg` is flags for permission and creation of the semaphores. The following only discusses `semget` as relevant to this assignment; if you want more, manual page discusses each parameter in more detail.

The `key` parameter is an integer you provide to identify your semaphore. You will need to pass this key to any processes that needs to access to the semaphore. The `nsem` parameter defines the number of semaphores in the set, which should be 1 for this project (unless you want more of a challenge). The `semflg` controls the access to the semaphore by other processes. This access control is similar to file access control in Unix: you can allow read and/or write access to processes owned by the same user, users in the same group, or all users. In addition to this control, other flags determine whether or not the semaphore should be created if it does not exist or whether the existence of the semaphore with the key is an error. The recommendation for this assignment, is to use `IPC_CREAT | 0600` for the `semflg` parameter, which creates the semaphore if it does not already exist and only allows processes owned by you to access the semaphore.

The return value of the `semget` function is a numeric identifier for the semaphore. This value will be -1 if there is an error, and you should check for resulting errors in your program. Primarily the error checking is just to identify any problems you may run into along the way.

So, putting this altogether, to create a semaphore set with one element and key 123, use the following code:

```
int semaphore;

semaphore = semget(123, 1, IPC_CREAT | 0600);
if (semaphore == -1) {
    /* error occurred so print an error message and exit */
```

```
    perror("semget");
    exit(1);
}
```

Using the above code will also return an already existing semaphore, so you use it to access a semaphore in child processes after the semaphore already exists.

5.3 Initialization

To set the initial value of the semaphore, use the `semctl` function. This function controls several aspects of semaphores, including the ability to destroy them. In order to use `semctl` the following union is needed in your own code to pass arguments to the semaphore control function:

```
#if !defined(_SEM_SEMUN_UNDEFINED) || _SEM_SEMUN_UNDEFINED
/* according to X/OPEN we have to define it ourselves */
union semun {
   int val;                    /* value for SETVAL */
   struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
   unsigned short int *array;  /* array for GETALL, SETALL */
   struct seminfo *__buf;      /* buffer for IPC_INFO */
};
#endif
```

The prototype for the `semctl` function is

```
int semctl(int  semid, int semnum, int cmd, union semun arg);
```

where `semid` is the semaphore identifier (returned earlier), `semnum` is the semaphore number in the semaphore set to modify (and should be 0 in your programs), `cmd` is the command to execute, and `arg` is the argument for the command. Each of these are described in detail in the manual page. The return value is not important, except that -1 represents that an error occurred.

For initialization, the proper command is `SETVAL`. From the comments for the argument union, the correct field in the union to fill in is `val`. So, suppose we want to initialize the value of our previously created semaphore to 1. Notice that the second parameter is 0 to modify the first (and only) semaphore in the set. The following code is used:

```
int rc; /* return code */
union semun sem_val;

/* Initialize the value to 1*/
sem_val.val = 1;
rc = semctl(semaphore, 0, SETVAL, sem_val);
if (rc == -1) {
    /* error occurred */
    perror("semctl");
    exit(1);
}
```

5.4 Updating

This is where the behavior (or at least description of the behavior) changes from semaphores in class. In class, the `P` and `V` operations subtract and add one to the value of the semaphore respectively. In SYSV semaphores, you can modify the behavior by any amount and even check if the value is precisely zero.

If you add a positive number to the semaphore value, the value is updated and the updating process continues to run without blocking. If you add a negative number n to the semaphore value v then several possibilities arise:

- if $v >= |n|$, then n is added to v and the process continues running.
- if $v < |n|$, then the process is blocked until $v >= |n|$, at which time n is added to v and the process continues running.

Given this description, there is no guarantee that a queue is being used to maintain a list of waiting processes. So you cannot be guaranteed of that behavior. Still, the `P` and `V` operations can be effectively modeled by only using -1 and 1 as the modification values.

To modify the semaphore value, the `semop` function is used. The prototype for `semop` is

```
int semop ( int semid, struct sembuf *sops, unsigned nsops );
```

where `semid` is the semaphore identifier, `sops` is an array of modifications (semaphore operations) to make to the semaphores in the set, and `nsops` is the number of modifications in the array `sops`. The last value should be 1 in your program. The return value is generally not important unless it is -1, signifying an error.

The `sembuf` structure is defined as

```
struct sembuf
{
  short int sem_num;           /* semaphore number */
  short int sem_op;            /* semaphore operation */
  short int sem_flg;           /* operation flag */
};
```

where `sem_num` is the semaphore in the set to modify (should be 0), `sem_op` is the modification number to be made (1 or -1), and `sem_flg` are flags determining whether or not the process should really block, and whether or not the operation is undone upon process termination. This last field should be 0 in general.

So, to add one to the semaphore value, use the following code:

```
struct sembuf sem_op; /* semaphore operation buffer */
int rc; /* return code */

/* Increase the value of the semaphore by 1 */
sem_op.sem_num = 0; /* always one semaphore per set */
sem_op.sem_op = 1;  /* increase value by one */
```

```
sem_op.sem_flg = 0; /* no special flags */
rc = semop(semaphore, &sem_op, 1);
if (rc == -1) {
    perror("semop");
}
```

6 Shared Memory

Linux also provides mechanisms for constructing and using a block of memory that is shared among several processes. The programmer's interface to the shared block of memory is the SYSV shared memory API.

6.1 Preliminaries

To use shared memory, you must have the following compiler directives in your source code:

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

These include constants and function prototypes for using shared memory.

In addition, it's an extremely good idea to define a structure that contains all of the shared variables for your programs. For example, the following structure will be used to share two integers and a character array:

```
typedef struct {
    int x;
    int y;
    char location[20];
} SharedData;
```

Be careful if you try to share classes in this way.

6.2 Creation

To create a shared memory segment, the `shmget` function is used. The prototype for `shmget` is

```
int shmget(key_t key, int size, int shmflg);
```

where `key` is the key to use for identifying the shared memory, `size` is the number of bytes to allocate in a shared memory segment, and `shmflg` determines the permissions and other properties related to the shared memory segment.

The key and flag parameters are essentially the same as for semaphores, so refer to that section for more information. Again, the key needs to be communicated to all processes needing access to the shared memory segment. The `size` field is used to specify the number of bytes needed in the shared memory segment. The return value is the shared memory segment identifier that is used in subsequent calls, or -1 if there was an error.

To allocate shared space for our variables in the `SharedData` structure in a shared memory segment with key 200, use the following code:

```
int shmid;
shmid = shmget(200, sizeof(SharedData), IPC_CREAT | 0600);
if (shmid == -1) {
    perror("shmget");
    exit(1);
}
```

The same code accesses an already existing shared memory segment (with key 200), so you can use this code in child processes.

6.3 Mapping to Address Space

Once the shared memory segment has been created it needs to be mapped into the virtual address space of the process (question to ponder: why?). The `shmat` function is used to map the segment into the address space of the process. The function prototype for `shmat` is

```
void *shmat ( int shmid, const void *shmaddr, int shmflg );
```

where `shmid` is the shared memory segment identifier returned by `shmget`, `shmaddr` is the requested virtual memory address to place the shared memory segment, and `shmflg` are flags modifying how the memory segment is loaded. For this project, you should specify an address of `NULL` and flags of `0` to allow the function to place the shared memory segment wherever it can. The return value of the function is the virtual address for accessing the shared memory segment, or `NULL` if there was an error.

To map our previous shared memory segment into the process' address space, use the code

```
SharedData *sharedData;
sharedData = (SharedData *)shmat(shmid, NULL, 0);
if (sharedData == NULL) {
    perror("shmat");
    exit(1);
}
```

Notice the cast of the address to point to a value of type `SharedData`. This is a useful technique so that the contents of the shared memory segment can be accessed by field names of `sharedData`.

6.4 Accessing Shared Variables

Once you have mapped the shared memory segment into the process' address space, you can access the shared variables through the pointer returned by `shmat`. For example, the following code sets the shared variable `x`, to the value 25.

```
sharedData->x = 25;
```

Now, every process that has accessed and mapped the same shared memory segment will see the value 25 when accessing `x`,.

7 Viewing SYSV IPC Remnants

You can see what SYSV IPC resources are currently in use by using the `ipcs` command. Some sample output is below.

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x000003e9 8248833    joeuser    600        24         0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status
0x00000001 1024       joeuser    600        1
0x00000002 1025       joeuser    600        1
0x00000003 1026       joeuser    600        1
0x00000004 1027       joeuser    600        1
0x00000005 1028       joeuser    600        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes  messages
```

It's generally a good idea to clean up after your program executes, which you can do through proper programming, but you are not required to do so. Instead you can use the `ipcrm` command. For example, to remove the semaphore with semid 1027, the command (and response) is

```
% ipcrm sem 1027
resource deleted
```

Use `shm` as the first argument to remove shared memory segments.

8 Hints

- To succeed in this project, it will be important to write functions that map the SYSV semaphores to semaphores like we described in class. Implementing `P` and `V` operations for a semaphore (and defining just what a semaphore is) is vital.
- There is a useful tutorial on SYSV IPC at the URL `http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html`
- Finally, work out your solution on paper. You may get a false belief that your program works by running it over and over again, but unless you have it all worked out in your head, you won't be sure it actually works.

9 Submission

We will use the Curator located at `http://spasm.cs.vt.edu:8080/curator01/` to collect program submissions. No grading will be done by the Curator.

You are to submit a single tarred (`man tar`) and gzipped (`man gzip`) archive containing

- A text file named `README` describing the program, describing the contents of the archive, providing building instructions (including the platform you used for development), and a user's guide (including how to start the program);
- The source code for your programs;
- Sample output for 3 executions of your programs;
- A script named `build` or a suitable `Makefile` for building your programs.

Your files must be a directory named `project3-pid` (using your `pid` in the name) of the archive. Be sure to include only the files listed above. Do not include extra files from an integrated development environment such as `configure` scripts, automake related files, etc. This is primarily an issue if you are using an IDE such as KDevelop.

Be sure to include your name in all files submitted. **DO NOT** include executables or object files of any type in the archive. **Submissions that do not gunzip and/or untar will not be graded. Be careful to FTP in binary mode if you are transferring your file to a Windows machine before submitting to the Curator.**

Failure to follow the submission rules will result in a grade of zero (0) for this assignment. There will be no exceptions.

10 Programming Environment

As stated in the syllabus, you must use Linux and `gcc/g++` to implement this project. Your program must compile and run properly in the lab. You may use any STL classes, but no third party data structure libraries. You may also use code from textbooks, but you must properly cite the source.

11 Programming Style

Your solution may either use an object-oriented or procedural design. Either way your solution should employ reasonable decomposition.