CS2106: Operating Systems
Lab 3 – Process Communication & Synchronization in UNIX

Important:

- The deadline of submission through IVLE is **12th October 5pm**
- The total weightage is **6% + [Bonus 1%]:**
    - Exercise 1: 1% **[Lab Demo Exercise]**
    - Exercise 2: 2%
    - Exercise 3: 3% + **[Bonus 1%]**

Section 1. Exercises in Lab 3

There are **three exercises** in this lab. The purpose of this lab is to learn about process communication and synchronization in Unix-based OS. Due to the difficulty of debugging concurrent programs, the complexity of this lab has been adjusted accordingly. Hence, you should find the required amount of coding "minimal". Most likely you will spend more time on understanding the various mechanisms than on the actual programming effort.

General outline of the exercises:

- Exercise 1: Understanding POSIX shared memory.
- Exercise 2: Understanding POSIX semaphores.
- Exercise 3: Using shared memory and semaphore to solve a synchronization problem.

Note: the last 3 pages of this document contain short descriptions on the essential library calls. You can consider not to print them to save paper.

2.1 Exercise 1 [Lab Demo Exercise]

In this exercise, we are going to use **POSIX shared memory** mechanism to solve a simple problem. As a recap of lecture 5, the basic steps of using shared memory mechanism in general:

1. Create a shared memory region
2. All process intended to communicate with each other **attach** the shared memory region to their own memory space
3. Read/Write to the shared memory region for communication

The major shared memory library calls are listed in the appendix A for your reference.

Shared Memory Mechanism Demonstration

Under the "**`demo`**" subdirectory in "**`ex1"`** directory, use the "**`make`**" command to compile the three sample programs: **`shdMemAndFork.c`**, **`shdMemServer.c`** and **`shdMemClient.c`**.

The program **`shdMemAndFork`**.c shows one way of connecting a parent process with its child process. You can run it by using the "**`shdMem`**" executable. Here's a general description of the program:

Parent Process:

1. Create and attach the shared memory region.
2. Spawn a child using **`fork()`**
   - Useful trick: Shared memory region created and attached **before `fork()`** will remain functional after **`fork()`**. Hence, the parent and child still share the memory region after **`fork()`**.
3. The shared memory region is treated as an **integer array** in this program. The line
   ```
   int* Array = (int*) shdMemRegion;
   ```
   points **`Array`** to the starting address of the shared memory region. Afterwards, we can treat **`Array`** as a normal integer array. The **`Array[]`** is used as follows:
   - **`Array[0]`**: used for the parent process to indicate the values are ready for the child.
   - **`Array[1]`** and **`Array[2]:`** 2 values "passed" to the child.
   - **`Array[3]:`** used by the child process to indicate that the values have be overwritten and ready for the parent process to read.
4. Parent writes **`1234`** and **`5678`** to **`Array[1]`** and **`[2]`** respectively.
5. Parent writes **`9999`** to **`Array[0]`** to indicate the values are ready.
6. Parent repeatedly checks **`Array[3`**] for "**`1111`**" to see whether the child process is done
   - This is one crude way to synchronize parent with child process.
7. Parent found **`1111`** in **`Array[3]`** and reads **`Array[1]`** and **`[2]`** to see the updated values.

Child Process:

1. Spawned by the **`fork()`** system call at Step 2 of Parent Process
2. Repeatedly checks **`Array[0]`** for the value **`9999`**, which indicates the parent is done with writing.
   - See Step 4 and 5 of Parent
3. Reads the value of **`Array[1]`** and **`[2].`**
4. Change the **`Array[1]`** and **`[2]`** to **`4321`** and **`8765`** respectively
5. Change **`Array[3]`** to **`1111`** to indicate the writing is done.
   - See Step 6 of Parent

As an alternative, the two files **`shdMemServer.c`** and **`shdMemClient.c`** are provided to demonstrate communication between two independent processes with no parent-child relationship. The two files contain essentially the same code as the parent and child process in the previous code with the omission of **`fork()`** system call.

The key difference is that the client (which plays the child process role) has to attach the shared memory region "manually" using the memory region id. To run the program, you can do the following:

Use two terminal windows (as shown during week 5 lecture):

- In terminal A, run server program by using the executable "**`shdMemServer`**".
- Take note of the memory region id printed on screen.
- In terminal B, run client program by using the executable "**`shdMemClient`**".
- Enter the memory region id when prompted.
- Observe the results.

You can also use one single terminal window:

- Run the server program in background by "**`shdMemServer &`**".
- Take note of the memory region id printed on screen.
- Run client program by using the executable "**`shdMemClient`**".
- Enter the memory region id when prompted.

Your task for exercise 1:

Write a **single program `ex1.c`** (i.e. use the **`shmAndFork.c`** approach) to spawn a single child process then perform the task below:

Parent Process:
1. Ask the user for the size of integer array, **`S`**.
2. Ask the user for the starting value to be placed in this array, **`V`**.
3. Store **`V`** in **`A[0]`**, **`V+1`** in **`A[1]`**, ……, **`V+S-1`** in **`A[S-1]`** • The values in the array **`A[]`** should be communicated to the child process via shared memory region.
4. Let the child process to sum up the first half of the shared array, i.e. **`A[0]`** to **`A[(S/2)-1`**].
5. Sum up the second halves of the shared array, **`A[(S/2)]`** to **`A[S-1]`**
6. Wait for the child to finish summing and report the following: • Partial sums from the parent and child processes. • The final sum.

Child Process:
1. Wait for the parent to finish populating the shared array **`A.`**
2. Sum up the first half of the shared array, i.e. **`A[0]`** to **`A[(S/2)-1].`**
3. Report the partial sum to parent process.

Sample Execution Session: (User input in **bold** font)

```
Enter Array Size: 10
Enter Start Value: 1         //Shared Array = {1, 2, 3, …, 8, 9, 10}
Parent Sum = 40              //6+7+8+9+10
Child Sum = 15               //1+2+3+4+5
Total = 55
```

Important Note:

Please ensure you use shared memory to pass the values between parent and child. **Any other mechanism will not be accepted.** ☺

2.2 Exercise 2

In this exercise, we are going to learn about **POSIX semaphore**. As discussed in lecture, semaphore is a simple yet powerful synchronization mechanism. The POSIX semaphore is one possible implementations under Unix, other popular choice include **`phtread_mutex`** (semaphore implementation for **`pthread`** library).

The major semaphore library calls are listed in the ***appendix B*** for your reference.

Semaphore Mechanism Experiments

Under the "**`demo`**" subdirectory in "**`ex2`**" directory, there are 2 sample programs: **`semaphore_example.c`** and **`semaphore_exampleImproved.c`**. You can similarly use the "**`make`**" command to compile them.

In **`semaphore_example.c`**, pay special attention to semaphore allocation in the shared memory region. Note the step where the shared memory region is type casted as (**`sem_t*`**), which allows the region to be used as a semaphore structure.

The program consists of a very simple interaction between a parent and child process. The parent and child process will prints out three occurrences of character '**`p`**' and '**`c`**' respectively. Without any synchronization, you can see interleaving patterns, e.g.

Examples of interleaving pattern in output	

`p`
`c`
`c`
`p`
`p`
`c`
	`p`
	`c`
	`p`
	`c`
	`p`
	`c`

Suppose we want to force the processes to print out the character consecutively, i.e.

The only two correct consecutive patterns in output	

`p`
`p`
`p`
`c`
`c`
`c`
	`c`
	`c`
	`c`
	`p`
	`p`
	`p`

This can be easily accomplished by using a binary semaphore to "protect" the whole for-loop in the parent or child process. In the same program, the semaphore is already allocated and initialized. Place the **`sem_wait()`** and **`sem_post()`** at appropriate locations to get the desired behavior.

The **`semaphore_exampleImproved.c`** is simply a modularized version of the **`semaphore_example.c`**.

Your task for exercise 2:

Firstly, we have repackaged the semaphore functions seen in the demo programs into a set of function wrappers. This allows us to use the semaphore without worrying about how to allocate them in the share memory region etc. To be more general, the functions **allocate array of semaphores instead of a single semaphore**. The functions can be found at the top of **`ex2.c`**.

Consider this problem: Two processes (parent and its child) cooperate to fill a shared array of size **`500,000`**. The parent process will fill the array using the value **`1111`**, while the child process uses **`9999`**. Each process should produce **`250,000`** values individually. However, the order in which the process fill in the shared array is not relevant (i.e. the **`9999`** and **`1111`** in the array need not follow any specific pattern).

To facilitate the production, a shared array **`sharedArray`** of **`500,001`** values is allocated. The first location **`sharedArray[0]`** stores the index of the next free location in the array. So, the parent and child process should perform the following for **`250,000`** times:

1. Read the index from **`sharedArray[0].`**
2. Produce the value (**`9999`** or **`1111`**) to the free slot.
3. Increase the index in **`sharedArray[0].`**

The last part of the parent process consists of a simple auditing code, which goes through the shared array and checks the number of occurrences of **`1111`** and **`9999`**. In a correctly coded program, you should see this result at the end of execution:

```
Audit Result: P = 250000, C = 250000, N = 0
```

`P` and **`C`** represent the occurrences of **`1111`** and **`9999`** respectively, which should be exactly **250,000** each in a correct program. **`N`** represents occurrences of any other value, which **should be zero** in a correct program.

Please note that:

- You should not change the allocation or layout of the shared array in anyway.
- **You should leave the auditing code untouched** to facilitate our testing.
- You must use semaphore to solve this task:
  - Semaphore should be used to protect critical code only. If your critical section contains more operations than absolutely necessary, you may be penalized.

2.3Exercise 3

This exercise makes use of both share memory region and semaphore to solve a synchronization problem.

Consider a modern age implementation of the classic **mancala game:**

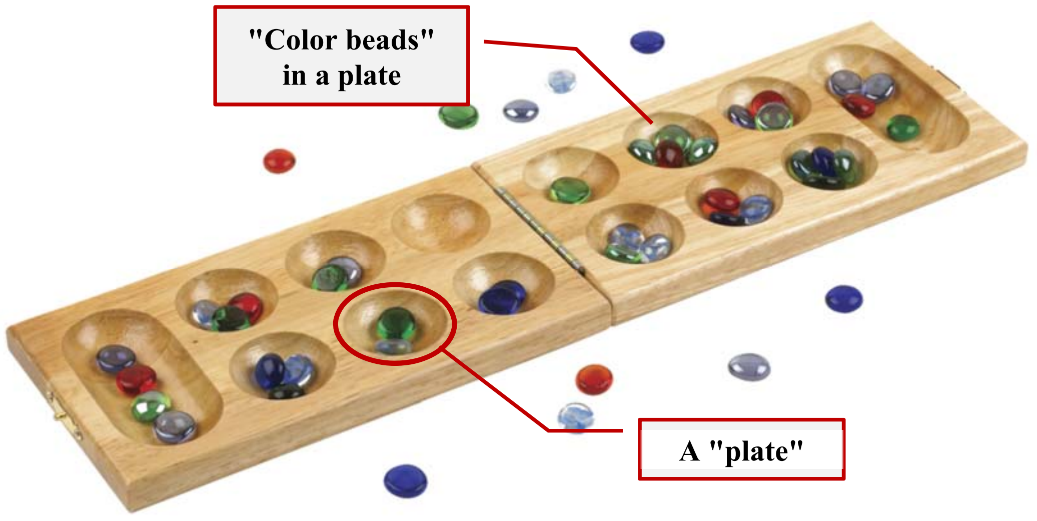


Image taken from Walmart webpage

The new mancala game supports **5 players or more** and has as many "plates" as the number of players. In each of the plates, there are a number of randomly placed **red, green** and **blue** color beads. Each player has an id [0 to number of players -1] and each plate is similarly numbered. A player has **only one** simple move:

- Choose beads of a particular color in his/her plate (i.e. plate with the same id as his/her id).
- Move 1 to 5 beads to **next plate** (i.e. plate with the id+1, wraps around). As each plate can have at most 99 of each of the color beads, the player will always respect the limit (i.e. move less or none if the target plate is almost full / full).

The entire game logic has already implemented for you ☺. Your job is just to make sure the players don't mess up each other's plate.....

Key notes of the game logic implementation:

- Each "plate" is simulated as a 6-digit integer. As each type of color beads cannot exceed 99, we store the blue color beads as the last two digits, green as the middle two digits and green as the first two digits. e.g. "**123456**" means 12 red color beads, 34 green color beads and 56 blue color beads.

<u>**Your task in exercise 3:**</u>

Compile the given `ex3.c` and try it out. Remember that you need a "`-lpthread`" flag for the semaphores library. The program expects 3 command line arguments:

```
a.out [random seed] [no. of players] [no. of rounds]
```

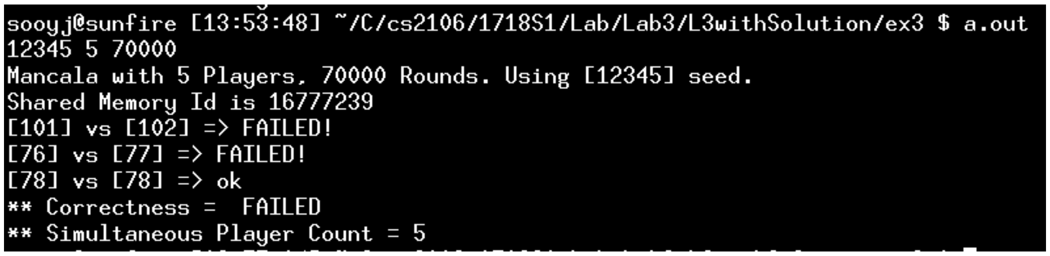
- `[Random seed]` is used to setup the random number generator. Using the same seed should give you the same setup between runs.
- `[No. of Players]` should be larger or equal to 5.
- `[No. of Rounds]` indicates how many moves per player are to be performed.

The program creates `[No. of Players]` concurrent processes and let them access a shared mancala. Each player process will perform the movement of color beads `[No. of Rounds]` times. At the end of the program, two important checks are performed:

1. Whether the total number of {red, green, blue} beads in the mancala remains the same. This should be intuitively true, as beads are move around but never removed / added.
2. The maximum number of simultaneous players at any point in time. This shows you how many process played "at the same moment in time".

If you run the program now with random seed **12345** and **5** players, you can see that the verification actually fails when the number of rounds is large.
[Note: round number is system dependent, around 200 and above in the lab machine, more than 70000 rounds on Solaris ☺]
[Note^2: Timing dependent! The same parameters may give different behavior!].

Below is an example of a failed run:



```
sooyj@sunfire [13:53:48] ~/C/cs2106/1718S1/Lab/Lab3/L3withSolution/ex3 $ a.out
12345 5 70000
Mancala with 5 Players, 70000 Rounds. Using [12345] seed.
Shared Memory Id is 16777239
[101] vs [102] => FAILED!
[76] vs [77] => FAILED!
[78] vs [78] => ok
** Correctness =  FAILED
** Simultaneous Player Count = 5
```

You can see that we started with **`[101]`** red beads and somehow ended up with **`[102]`** red beads! Similar issue plagues the green color beads too. In some runs, color beads can disappear into thin air!

So, your job is very simple, **fix the synchronization** between the processes so that we will always have consistent game state before and after. Note that you **can only modify the program at the places marked with a "//TODO" comment.**

The marking of this question follows a tiered system:

1	**Correctness [1%]**	The game end verification passes regardless of the number of rounds played for **5 players**.
2	**Parallelism [1%]**	You allowed the **maximum possible** players to play at the same time **for 5 players**.
3	**Scalability [1%]**	Your solution works for player count beyond 5 [i.e. both **correct** and **allow maximum parallelism**]. To rein in system resource usage, we will not test with more than 10 players.

Please note that we only consider higher tier scoring if your code satisfy ALL lower tier(s) requirements. For example, an incorrect program receives **0 regardless of how many simultaneous players supported.** Similarly, if your code does not allow the maximum parallelism (tier 2), we will not test beyond 5 players (tier 3).

Hints:
- Identify **the type of synchronization problem first**.
- The solution is very short, you need around 10 lines of new code.
- Try the **simplest solution first**.

Bonus Round!

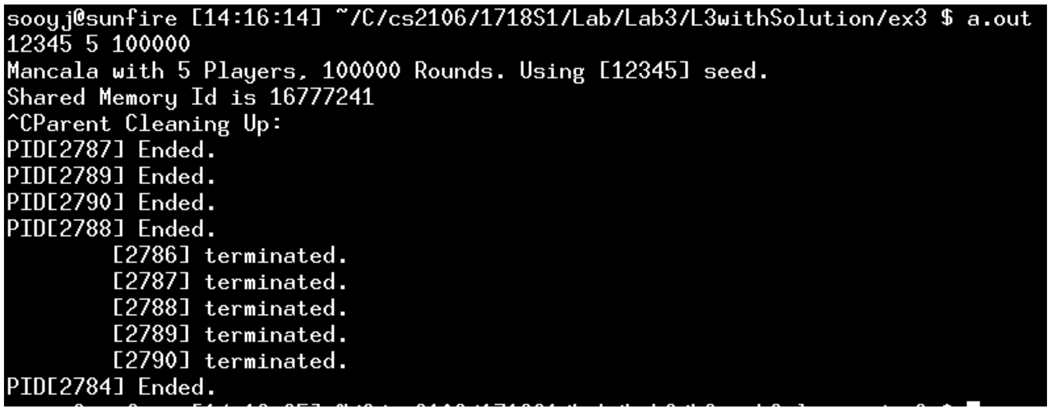
When working with multi-process programs (like lab 2 and this lab), there are two irritating problems:

a. When the parent process is terminated with **`[ctrl-c]`**, child processes may be still running around.
b. Similarly, resources allocated (e.g. shared memory, semaphores etc) may not be cleared up properly if the program dies a "unnatural death".

Fortunately, (a) is already handled automatically by terminal. When the [ctrl-c] signal (i.e. SIGINT) is send to the foreground process, the signal will be relayed to the **entire process group (i.e. all child spawned).** So, we only need to worry about (b).

Let's utilize what you have learned about **unix signal** and add the following functionality:
- When the parent process received the SIGINT signal, it will monitors the termination of all child processes and exit **only when all child processes are terminated.**

Use the same reporting format as the following sample run:



```
sooyj@sunfire [14:16:14] ~/C/cs2106/1718S1/Lab/Lab3/L3withSolution/ex3 $ a.out
12345 5 100000
Mancala with 5 Players, 100000 Rounds. Using [12345] seed.
Shared Memory Id is 16777241
^CParent Cleaning Up:
PID[2787] Ended.
PID[2789] Ended.
PID[2790] Ended.
PID[2788] Ended.
        [2786] terminated.
        [2787] terminated.
        [2788] terminated.
        [2789] terminated.
        [2790] terminated.
PID[2784] Ended.
```

The **`[Ctrl-C]`** caused a "**`^C`**" be printed on screen. The "**`PID[XXXXX] Ended`**" is printed by each of the child process, while the "**`\t[XXXX] terminated`**." is reported by the parent process. The last "**`PID[2784] Ended`**." belongs to the parent process.

Notes:

- The bonus score is only considered **if you meet at least the tier 1 correctness requirement.**
- You can use this functionality to clean up the system resources properly, e.g. detach and delete share memory regions etc. However, due to the many ways in which you can allocate system resources, we **will not check** on the cleanup code.
- Make sure the bonus segment does not affect your main solution in anyway. It is probably easier to experiment and learn in a separate program before you place it in the main solution code.
- You need to read up on the "**`signal()`**" system calls.

Section 3. Submission

Zip the following files as A/E0123456.zip (**use your NUSNET user id**):

a. **`ex2.c`**
b. **`ex3.c`**

Upload the zip file to the "Student Submission→Lab 3" workbin folder on IVLE. Note the deadline for the submission is **12th October, 5pm**.

Again, please ensure you follow the instructions carefully (output format, how to zip the files etc). Deviations will cause unnecessary delays in the marking of your submission.

**~~~ Have Fun! ~~~**
**~~~ Have Fun! ~~~**
**~~~ Have Fun! ~~~** ;-)

Appendix A. Share Memory Library Calls

```
1. shmget( name, size, settings );
```

Brief Description
Stands for **Sh**ared **M**emory **Get**. Each POSIX shared memory region has a name (also known as **`key`**) and a unique **`id`**. This library call can be used to create or to get an existing shared memory region with key **`name`**. If a shared memory region with the same key already exists, then the call will failed. As it is sometimes very hard to find an unused key, we can use a special value **`IPC_PRIVATE`**, which asks the system for any unused key to create the new shared memory region.

The 2nd parameter specifies the size of the shared memory region in bytes.

The 3rd parameter specifies further settings associated with the shared memory region, e.g.

- Permission of access: who can read/write the shared memory region. It is the same as the permission flags for all files in unix. For simplicity, you can use the setting "**`0666`**" for this lab, which specifies all users can read/write to the shared memory region
- To create or to find existing memory region: Use the predefined constant **`IPC_CREAT`** to create a new region.

Return value
If successful, a unique memory region id (an integer) will be returned by this function call. This id will be used by most shared memory region function calls (see later). Unsuccessful call is indicated by negative return value.

```
2. shmat( memRegion_id, where, settings );
```

Brief Description
Stands for **Sh**ared **M**emory **At**tach. Attach an existing shared memory region to the process's memory space.

The 1st parameter is the unique shared memory region id (e.g. returned by shmget() call).

The 2nd parameter specifies where in the memory space to attach the shared memory region. Most of the time, we can just let the system to decide. Use a **`NULL`** as 2nd parameter to ask the system to find a suitable location.

The 3rd parameter specifies further setting which can be safely ignored in this lab. Use a **`0`** to indicate no further setting is required.

Return value
The starting address of the shared memory region is returned as a (**`void*`**). A pointer variable can be used to remember this address. Subsequent access through this pointer will access the actual shared memory region. Note that, similar to **`malloc()`**, we usually have to typecast it to the appropriate pointer type before using it.

If this function fails, a **`-1`** is returned.

```
3. shmdt( address_of_shared_memory_region );
```

Brief Description
Stands for **Sh**ared **M**emory **D**e**t**ach. Detach the shared memory region from process memory space. The shared memory region is specified by the starting address (e.g. the return value from the **`shmat()`** call). As a good practice, you should detach a shared memory region after use. A shared memory region with no process attaching to it can be safely destroyed (see the **`shmctl()`** call below).

```
4. shmctl( memRegion_id, operation, setting_for_operation);
```

Brief Description

Stands for **Sh**ared **M**emory **C**on**t**ro**l**. Perform control operations on the shared memory region specified by the shared memory region id, **`memRegion_id`**. In this lab, we use this call only to remove a shared memory region from the system. The removal operation can be specified by the special value **`IPC_RMID`**. The 3rd parameter is only valid for other operations (not used in this lab). So, you can safely place a **`0`** as the 3rd parameter.

Appendix B. POSIX Semaphore Library Calls

```
1. sem_init( sem__pointer, sharing_setting, initial_value );
```

Brief Description

POSIX semaphore is defined as a structure of type **`sem_t`**, i.e. to declare a new semaphore, you can use:

```
    sem_t S;            //S is a semaphore structure
```

The function call **`sem_init()`** can then be used to initialize the semaphore before use. This function call takes in a semaphore structure by pointer as the 1st parameter.

The 2nd parameter is to set the sharing status of a semaphore. If **`0`** is used, then this semaphore is shared only between threads of a single process. In this lab, we are synchronizing between processes, hence a non-zero value should be used instead.

The 3rd parameter is the initial value to be given to the semaphore. E.g. to setup a binary semaphore as **`mutex`**, we can use the initial value of **`1`**.

```
2. sem_wait( sem__pointer );
```

Brief Description

This is equivalent to the **`wait()`** semaphore operation. A semaphore structure is passed as a pointer into this function call. If the value associated with the semaphore is **`<= 0`**, the process calling this will be blocked until a corresponding **`signal()`** call is made (see below).

```
3.  sem_post(semPtr);
```

Brief Description

This is equivalent to the **`signal()`** semaphore operation. Similar to the **`sem_wait()`** function, a pointer of a semaphore structure is passed as the first parameter. The semaphore value will be incremented and a single blocked process, if available, will be unblocked.

Additional note:

- If inter-process synchronization is needed, the semaphore must be placed in a location that is accessible to the processes, i.e. **the semaphore should be allocated in a shared memory region.**
- To compile programs using POSIX semaphore, remember to add the "**`-lpthread`**" flags, e.g.

```
    gcc sempahoreTest.c –lpthread
```