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

Important:

- The deadline of submission through IVLE is **9th October 5pm**
- The total weightage is **6%:**
  - Exercise 1: 1%
  - Exercise 2: 1%
  - Exercise 3: 4%
- System/OS restriction: Unix/Linux

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 coming midterm and 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 as IPC mechanism.
- Exercise 2: Understanding POSIX semaphores as Inter-Process synchronization mechanism.
- Exercise 3: Using shared memory and semaphore to understand the consumer/producer 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 understand the POSIX shared memory mechanism and use it 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

Use the "**make**" command to compile the three sample programs: **shdMemAndFork.c**, **server.c** and **client.c** given under the **ex1/** directory.

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 **`server.c`** and **`client.c`** are provided to demonstrate communication between two independent processes with no parent-child relationship. The **`server.c`** is the same as the parent process in the previous code with the omission of **`fork()`** system call. The **`client.c`** contains the same code used by the child process discussed.

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 recording of week 5 lecture):

- In terminal A, run server program by using the executable "**`server`**".
- Take note of the memory region id printed on screen.
- Go to terminal B, run client program by using the executable "**`client`**".
- 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 "**`server &`**".
- Take note of the memory region id printed on screen.
- Run client program by using the executable "**`client`**".
- Enter the memory region id when prompted.

If you are adventurous enough, you can ask you friend to run the server program and see whether your client program can communicate with it.

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 halve 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 halve 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 take a look at the 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 Experimentation

In the **`ex2/`** directory, there are 2 sample programs: **`semaphore_example.c`** and **`semaphore_exampleImproved.c`**.

In **`semaphore_example.c`**, pay attention to how we set a shared memory region for a semaphore. Note the step where we typecast the shared memory region to a (**`sem_t*`**), which allows us to treat the region as a dynamically allocated 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 an interleaving pattern, 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 just a modularized version of the **semaphore_example.c**.

Your task for exercise 2:

Modify **ex2.c** to perform the following task:

Two processes (parent and its child) are going to fill a shared array of size **50,000** with values. The parent process will fill the array using the value **1111**, while the child process uses **9999**. This is to distinguish between values "produced" by different process. Each process should produce **25,000** values. However, the order in which the process fill in the shared array is not relevant (i.e. the **9999** and **1111** need not be in an interleaving or any specific pattern).

To facilitate the production, a shared array **sharedArray** of **50,001** 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 **25,000** 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 = 25000, C = 25000, N = 0
```

P and **C** represent the occurrences of **1111** and **9999** respectively. **N** represents occurrences of any other value, which should be zero.

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

It is time to put what we have learned so far into action. This exercise makes use of both share memory region and semaphore to solve a producer/consumer problem.

There are N_p producer processes and N_c consumer processes in total. All processes share a bounded circular array of size `100`. The producer processes have to produce a total of `M` values, while the consumers will consume the same amount of values. Please note that `M` is the **total** production/consumption count (i.e. not per producer or consumer counts).

Basic outline of a producer process P_i
(`i` is the id of the producer, ranges from `0` to N_p `-1`)

- The total production count is initialized to **M.**

1. Read the total production count:
   - If the total production count == `0`, end process.
2. Read the index of the next free slot, `OUT:`
   - As `M` may be larger than the available size of the shared array, you should make sure `OUT` wraps back to `0` when it reaches the end of the shared array.
3. Produce the value ***i*** in the free slot in shared array.
4. Decrement the total production count.
5. Increment the individual production count.
   - Each process has its own individual production count for auditing purpose.
   - In a correct program:
     Sum of all individual production count == Total production count.

Basic outline of the job of a consumer process C_i
(`i` is the id of the consumer ranges from `0` to N_c `-1`):

- Total consumption count is initialized to **M**.

1. Read the total consumption count
   - If the total consumption count == `0`, end process.
2. Read the index of the next available item, `IN.`
3. Read the value `V` from the shared array.
4. Decrement the total consumption count.
5. Increment the occurrences of `V` for auditing purpose:
   - As `V` is produced by P_v, the individual count of P_v should matches the occurrences of `V` being consumed.

To facilitate checking and bookkeeping purpose, the shared array is allocated with **`100 + 4 + 2*Np`** integers, instead of only **`100`** integers. The array is used as follows:

Array Index	Array Content
0	Total Production Count
1	Total Consumption Count
2	OUT index
3	IN index
4 (For P_0) . . . . . . **4+N_p-1** (For P_{Np-1})	Individual production count for each producer process
4+N_p (For Value **0**) . . . . . . **(4+N_p) + N_p – 1** (For Value N_p)	Number of consumption for each value
4+2*N_p . . . . . . **4+2*N_p + 100 – 1**	The shared buffer of 100 integers

Your task in exercise 3:

A substantial portion of the program is already coded for you in **`ex3.c`**, which includes:

- Checking of command line arguments and converting them to number of producers/consumers and total production counts.
- Allocation of shared array.
- Spawning of processes
  - To avoid the famous fork-bomb ☺
- Assigning Producer/Consumer processes.
- Auditing code for checking program correctness.

Your coding effort is largely focused on the two functions ***`Producer()`*** and ***`Consumer()`***. The parameters of these functions are designed to ease your coding effort:

For producer process:

```
void Producer( int id, int* countLoc, int* index,
                       int* auditArea, int* bufferArea )
```

Brief description:
`id` = Identification of the Producer. Also used as the value produced by this producer into shared buffer area.

countLoc = A pointer to the shared Total Production Count in shared memory region. You can simply dereference this pointer to access/update the total production count.

index = For the **OUT** index, which indicate the free slot for producing new item. A pointer to the **OUT** index in shared memory region. Use dereferencing to access/update the **OUT** index.

auditArea = For the Individual Production Counts. Points to the "array" of individual production count in shared memory region. E.g. **auditArea[0]** gives you the individual production count of producer **0**.

bufferArea = The **100** elements shared buffer. Produced value should be place in this array. E.g. **bufferArea[0]** refers to the first location in the shared buffer, **bufferArea[99]** refers to the last location.

Note that you can add more parameters to the `producer()` function, e.g. to pass in semaphores etc. Please be reminded you should pass semaphore around using _pointer_ rather than pass by value.

For consumer process:

```
void Consumer( int id, int* countLoc, int* index,
                       int* auditArea, int* bufferArea )
```

Brief description:

id = Identification of the Consumer.

countLoc = A pointer to the shared Total Consumer Count in shared memory region. Dereference this pointer to access/update the total consumption count.

index = For the **IN** index, which indicates the next available item in shared buffer. A pointer to the **IN** index in shared memory region. Use dereferencing to access/update the **IN** index.

auditArea = For the counting the occurrences of each value read. E.g. **auditArea[0]** gives you the occurrences count of value **0**. Please note that although both **Producer()** and **Consumer()** has an **auditArea** array, they are pointed to different locations in the shared memory region.

bufferArea = The **100** elements shared buffer. Value to be consumed should be read from this array. E.g. **bufferArea[0]** refers to the first location in the shared buffer, **bufferArea[99]** refers to the last location.

Similar to `Producer()`, you are free to add parameter to the function call.

Requirements of a correct program:

- You should ensure correct production and consumption
  - The program must meet overall production/consumption count.
  - However, the production/consumption count for individual producer/consumer is not restricted. E.g. it is possible for `Producer 0` to produce more than other process due to scheduling, etc.
- You should ensure **maximum** parallelism
  - Only protect necessary code in critical section.
- You should use only semaphore(s) for synchronization.
- **You should not modify the auditing code, shared memory allocation and shared array layout.**

Assumption:

- Number of producers is lesser than 100.

Suggestions/Hints:

- Debugging concurrent process can be a nightmare. Restrict the "freedom" of your program to have a better chance to pinpoint the problem, e.g. use limited number of producer/consumer initially to test. A good start would be 1 producer and 1 consumer to test the basic structure, then vary the number of producer while keeping a single consumer (and vice versa).
- Use a very small **M** in initial testing. Ensure the producers and consumers are working fine before letting them run freely.
- Print out debug message for each producer/consumer (remember to use id to distinguish them from each other).
- You can print out the shared memory region after each update to make sure you have updated the array correctly. Just remember to remove unnecessary printout before you submit.

Section 3. Submission

Zip the following files as A0123456.zip (**use your student id!**):

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

Upload the zip file to the "Student Submission→Lab 2" workbin folder on IVLE. Note the deadline for the submission is **9th 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! ~~~

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, initial_value, sharing_setting );`

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 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`**.

The 3rd 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.

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 "**`-lrt`**" flags (stands for **R**eal **T**ime library), e.g.

```
gcc sempahoreTest.c -lrt
```