**Total POINTS 100**

**TRUE/FALSE QUESTIONs – 1pt each**

1. Shared memory IPC comes with built-in (kernel provided) synchronization False
2. FIFOs persist without any processes connected to them True
3. Shared memory and memory mapped files require 3 times the memory overhead compared to FIFO and MQ False
4. Pipes are supported by a First-In-First-Out bounded buffer given by the Kernel True
5. POSIX message queues support separate priority levels for the messages True
6. In POSIX message queues, the order of the messages is always FIFO without any exception False
7. A unnamed pipe can be established only between processes in the same family tree True
8. A unnamed pipe does not exist without processes connected to both ends True
9. In POSIX message queue, you can configure message size and number of messages between the 2 processes True
10. In shared memory IPC, the Virtual Memory manager maps the same piece of physical memory to the address space of each sharing process True
11. After creating a shared memory segment with shm\_open() function, the default size of the segment is 0 True
12. POSIX IPC objects (message queues, shared memory, semaphores) can be found under**/dev/mqueue** and **/dev/shm** directories True
13. You can set/change the length of the shared memory segment using **ftruncate()** function True
14. In POSIX, names for message queue, shared memory and kernel semaphores must start with a "/" True
15. **sem\_unlink()** function permanently removes a semaphore from the kernel True
16. You must use **ftruncate()** before using a shared memory segment True
17. You must call **mmap()** before using (i.e., read/write) a shared memory segment True

**File SYSTEMS**

1. [20 pts] Assume that a file system has each disk block of size 4KB and each block pointer of 4 bytes. In addition, the each inode in this system has 14 direct pointers, 2 single indirect pointers, 1 double indirect and 1 triple indirect pointer. Ignoring the space for inode, answer the following questions for this file system:
2. What is the maximum possible file size?

14\*4KB + 2\*1024\*4KB + 1024\*1024\*4KB + 1024\*1024\*1024\*4KB = 56KB + 8MB + 4GB + 4TB = 4.004 TB

1. How much overhead (amount of non-data information) for the maximum file size derived in (a)?

4.004 GB

1. How much overhead for a file of size 6GB?

4KB\*1024 + 1024\*1024\*4\*1.5 = 6GB

Overhead = 4 + 1024\*4\*1.5 = 6.004 MB

**SIGNALS**

1. [13 pts] The following code will create a Zombie child process because the child process is terminated and the parent process is busy in a loop without calling wait() function. Now, modify this program by handling SIGCHLD signal so that no Zombie process is created. The parent process cannot call wait()directly in the main(). However, calling wait() from inside the signal handler is fine. The main still must go to the infinite while loop. You can add helper functions.

int main**(){**

**if** **(**fork**()==** 0**)** // child process

exit**(**0**);**

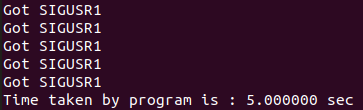
**else** // parent process

**while** **(true);**

**}**

1. **[20 pts]** Consider the program below and answer the following questions with proper explanation.
2. What is the output? How much time does the program take to run?[10 points]

The output for part a prints “Got SIGUSR1” 5 times. The total process take 5 seconds as the process runs through a for loop 5 times and sleeps for 1 second each.



1. What is the output with line 5 commented? How much time will it take now? [10 points]



SIGUSR1 is sending a user defined signal to the child process. There needs to be a signal handler to process this signal on the receiving end. Since there is no signal handler, the child processes the signal only once and then exits. The parent however waits for 5 seconds to completely exit. Therefore, the child exits in 1 second and the parent takes 5 seconds to exit.

|  |
| --- |
| 1 void signal\_handler **(**int signo**){**  2 printf **(**"Got SIGUSR1\n"**);**  3 **}**  4 int main **(){**  5  **signal (SIGUSR1, signal\_handler);** //comment out for b)  6 int pid **=** fork **();**  7 **if** **(**pid **==** 0**){**// chilld process  8 **for** **(**int i**=**0**;** i**<**5**;** i**++){**  9 kill**(**getppid**(),** SIGUSR1**);**  10 sleep **(**1**);**  11 **}**  12 **}else{** // parent process  13 wait**(**0**);**  14 **}**  15**}** |

1. **[30 pts]** Write a wrapper class KernelSemaphore on top of POSIX kernel semaphore. See sem\_overview(7) in man pages or linux.die.net to learn about kernel semaphores. Test your KernelSemaphore class by by setting the initial value to 0. Then write 2 programs – one waits for the semaphore and the other one releases (i.e., V()’) it. The header for the KernelSemaphore and the 2 programs in questions are provided in the below. You should make sure that the consumer program can only print out its prompt after the producer program has released the semaphore.

class KernelSemaphore{

    string name;

public:

    KernelSemaphore (string \_name, int \_init\_value);

    void P();

    void V();

    ~KernelSemaphore ();

};

// producer.cpp (Run this first in a terminal)

int main (){

    cout << "This program will create the semaphore, initialize it to 0, ";

    cout << "then produce some data and finally V() the semaphore" << endl;

    KernelSemaphore ks ("/my\_kernel\_sema", 0);

    sleep (rand () % 10); // sleep a random amount of seconds

    ks.V();

}

// consumer.cpp (Run this second in another terminal)

int main (){

    KernelSemaphore ks ("/my\_kernel\_sema", 0);

    ks.P();

    cout << "I can tell the producer is done"<< endl;

}