**Total POINTS 100**

**TRUE/FALSE QUESTIONs – 1pt each [30 pts]**

1. Shared memory IPC comes with built-in (kernel provided) synchronization

**False**

1. FIFOs persist without any processes connected to them

**True**

1. Shared memory and memory mapped files require 3 times memory overhead compared to FIFO and MQ

**False**

1. Pipes are supported by a First-In-First-Out bounded buffer given by the Kernel

**True**

1. POSIX message queues support separate priority levels for the messages

**True**

1. In POSIX message queues, the order of the messages is always FIFO without any exception

**False**

1. A unnamed pipe can be established only between processes in the same family tree

**True**

1. A unnamed pipe does not exist without processes connected to both ends

**True**

1. In POSIX message queue, you can configure message size and number of messages

**True**

1. In shared memory IPC, the Virtual Memory manager maps the same piece of physical memory to the address space of each sharing process

**True**

1. After creating a shared memory segment with shm\_open() function, the default size of the segment is 0

**True**

1. POSIX IPC objects (message queues, shared memory, semaphores) can be found under**/dev/mqueue** and **/dev/shm** directories

**True**

1. You can set/change the length of the shared memory segment using **ftruncate()** function

**True**

1. In POSIX, names for message queue, shared memory and kernel semaphores must start with a "/"

**True**

1. **sem\_unlink()** function permanently removes a semaphore from the kernel

**True**

1. You must use **ftruncate()** before using a shared memory segment

**True**

1. You must call **mmap()** before using (i.e., read/write) a shared memory segment

**True**

1. UDP protocol deals with retransmitting packets in case they are lost in route

**False**

1. TCP protocol is more heavy-weight because it maintains state information about the connection

**False**

1. Routing protocols are dynamic: they reconfigure under network topology change or outage automatically

**True**

1. Domain Naming System (DNS) can be used for load balancing and faster content delivery

**True**

1. The ping command is used to test whether you can make TCP connections with a remote host

**True**

1. The accept() function needs to be called the same number of times as the number of client-side connect() calls to accept all of them

**False**

1. HTTP protocol uses TCP underneath
2. Ports [0,1023] are reserved for well-known services (e.g., HTTP, SMTP, DNS, TELNET)

**True**

1. The master socket in a TCP server is used only to accept connections, not to run conversations with clients

**True**

1. A socket is a pair of IP-address and port number combination in both client and server side

**True**

1. For making a socket on the client side, the port number is chosen by the OS randomly from the available pool

**False**

1. A socket is like other file descriptors and is added to the Descriptor Table

**True**

1. UDP is connectionless while TCP is connection oriented

**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?

**Size of direct pointer = 14**

**Size of indirect pointer = 2 ^ 10**

**Size of double indirect pointer = 2 ^ 20**

**Size of tribble indirect pointer = 2^30**

**Maximum Size = 4\*(14 + 2\* 2^10+ 2^20 + 2^30) = 56KB + 8MB + 4GB + 4TB = 4 \* 1074792462 (KB)**

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

**OverHead = (2^10+2^10+2^20+2^30)4(B)**

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

**OverHead=(6\*(2^30))/(4\*(2^10)) \* 4 (B)**

**SIGNALS**

1. [10 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){**

**signal(SIGCHLD,SIG\_IGN);**

**};**

**}**

1. **[15 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]

**Got SIGUSR1**

**Got SIGUSR1**

**Got SIGUSR1**

**Got SIGUSR1**

**Got SIGUSR1**

**It takes 5 seconds to run**

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

**No output**

**5 seconds to finish**

|  |
| --- |
| 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. **[15 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 (){

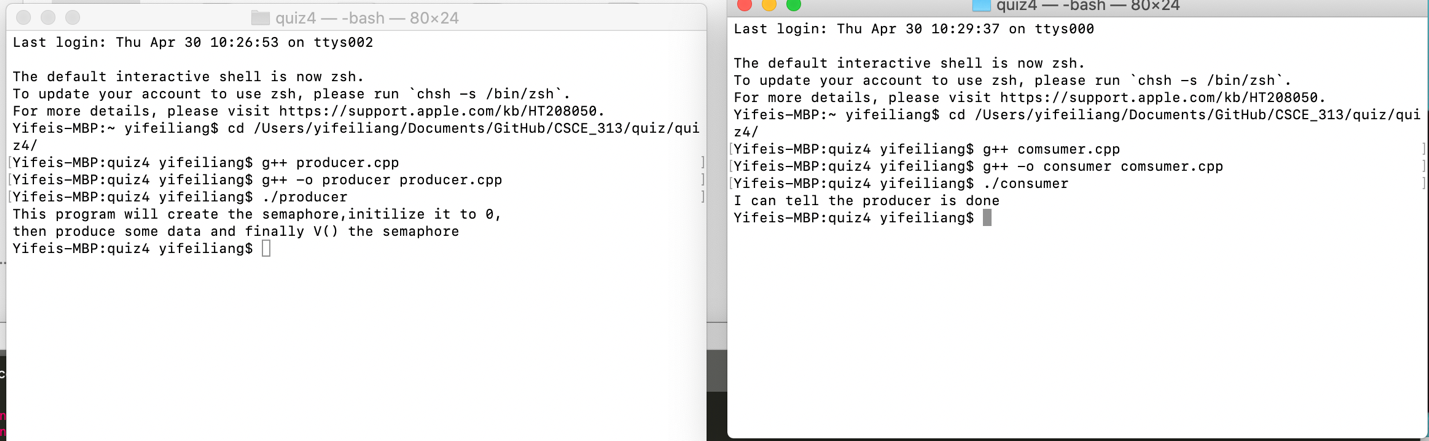
KernelSemaphore ks ("/my\_kernel\_sema", 0);

    ks.P();

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

}

**The .cpp file is in the folder**



1. **[10 pts]** Assume a very computer system from a company that used a very old legacy device whose path is /dev/legacy/specialdevice and it is about to be decomissioned. However, there are several important pieces of software who use this legacy device to log their output and you cannot change those. That means, the path /dev/legacy/specialdevice must continue to exist although the underlying physical device must be replaced. Now, what can you do to make sure all legacy tools and software continues running w/o problem without putting a new physical device in the above path? Note: you may forward all traffic to the legacy device to let’s say /sys/logfile path.

**Suppose we forward all the traffic from the legacy device to /sys/logfile. We can make a hard link to link every file in/dev/legacy/specialdevide to /sys/logfile. Thus even if we delete all the file in /dev/legacy/specialdeivce, the link in the /sys/logfile will not be affected and will functioning properly as deleted one.**

