CSCE 313 # QUIZ 4 - KEY SPRING 2020

TOTAL POINTS 100

TRUE/FALSE QUESTIONS – 1PT EACH [30 PTS] (ONLY THE FALSE ONES ARE CIRCLED)

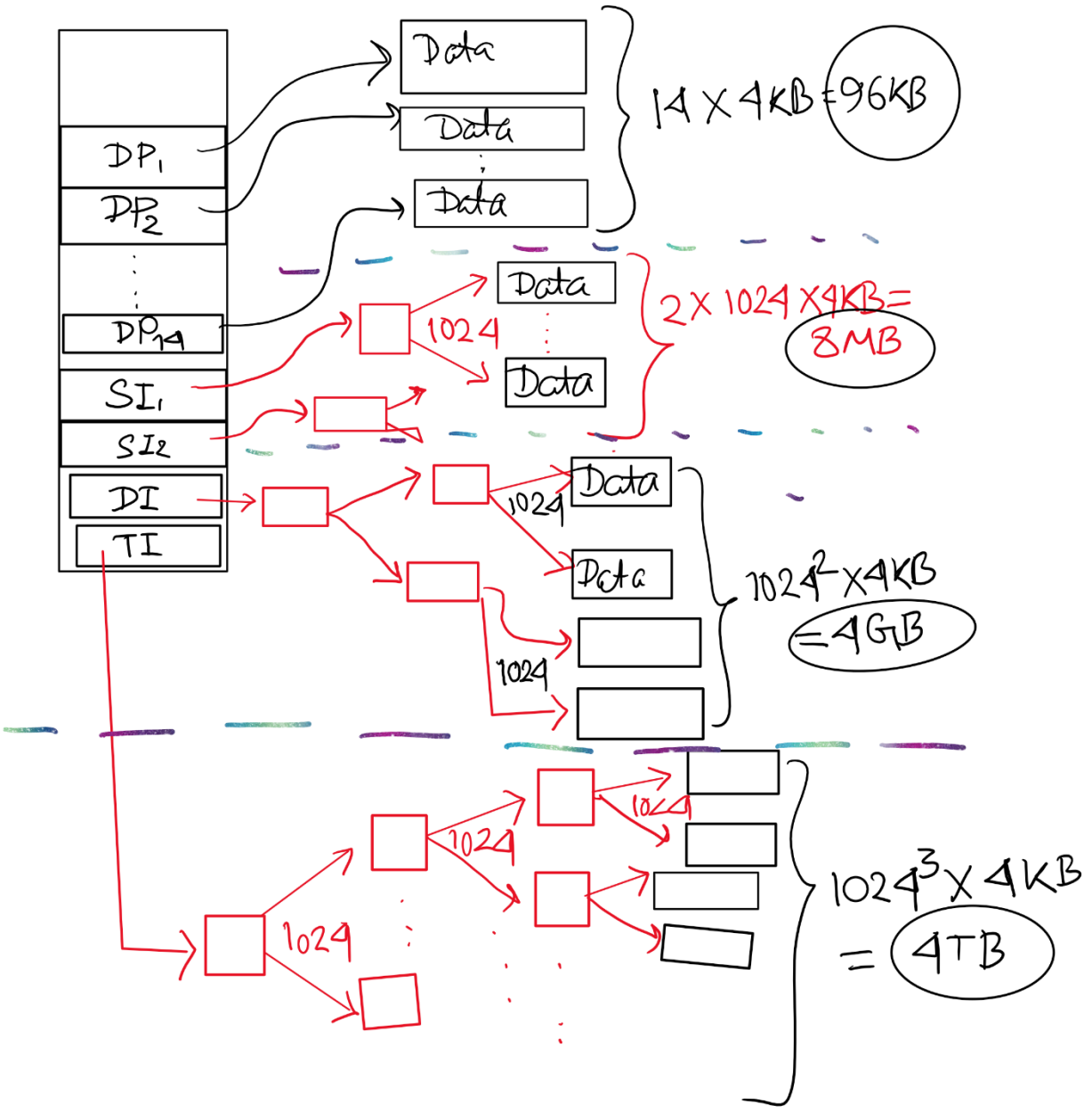
1. Shared memory IPC comes with built-in (kernel provided) synchronization
2. FIFOs persist without any processes connected to them
3. Shared memory and memory mapped files require 3 times memory overhead compared to FIFO and MQ
4. Pipes are supported by a First-In-First-Out bounded buffer given by the Kernel
5. POSIX message queues support separate priority levels for the messages
6. In POSIX message queues, the order of the messages is always FIFO without any exception
7. A unnamed pipe can be established only between processes in the same family tree
8. A unnamed pipe does not exist without processes connected to both ends
9. In POSIX message queue, you can configure message size and number of messages
10. In shared memory IPC, the Virtual Memory manager maps the same piece of physical memory to the address space of each sharing process
11. After creating a shared memory segment with shm_open() function, the default size of the segment is 0
12. POSIX IPC objects (message queues, shared memory, semaphores) can be found under **/dev/mqueue** and **/dev/shm** directories
13. You can set/change the length of the shared memory segment using **ftruncate()** function
14. In POSIX, names for message queue, shared memory and kernel semaphores must start with a "/"
15. **sem_unlink()** function permanently removes a semaphore from the kernel
16. You must use **ftruncate()** before using a shared memory segment
17. You must call **mmap()** before using (i.e., read/write) a shared memory segment
18. UDP protocol deals with retransmitting packets in case they are lost in route
19. TCP protocol is more heavy-weight because it maintains state information about the connection
20. Routing protocols are dynamic: they reconfigure under network topology change or outage automatically
21. Domain Naming System (DNS) can be used for load balancing and faster content delivery
22. The ping command is used to test whether you can make TCP connections with a remote host
23. 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
24. HTTP protocol uses TCP underneath
25. Ports [0,1023] are reserved for well-known services (e.g., HTTP, SMTP, DNS, TELNET)
26. The master socket in a TCP server is used only to accept connections, not to run conversations with clients
27. A socket is a pair of IP-address and port number combination in both client and server side
28. For making a socket on the client side, the port number is chosen by the OS randomly from the available pool
29. A socket is like other file descriptors and is added to the Descriptor Table
30. UDP is connectionless while TCP is connection oriented

FILE SYSTEMS

31. [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:
    (a) What is the maximum possible file size? [6 pts]
    (b) How much overhead (amount of non-data information) for the maximum file size derived in (a)? [7 pts}
    (c) How much overhead for a file of size 6GB? [7 pts]

CSCE 313 QUIZ 4 - KEY SPRING 2020

ANSWER



(a)

In the above, the far-left box is the inode that contains pointers (direct or indirect) to data blocks. Now, the according to specification, there 14 direct pointers to disk blocks. Since each block is 4KB, these direct pointers lead us to 14x4KB = 96KB.

Now, each single indirect (SI) pointer first points to a pointer block, which is full of pointers to disk blocks. A disk block, irrespective of data or pointer, is always 4KB in size. When it contains pointers, it can contain 4KB/4 = 1024 of those. As a result, we get a tree where the root node is indirect block and then the tree branches out to 1024 leaf nodes that are data blocks. [Notice that I am using red color for pointer blocks and black for data blocks. We will use this drawing for parts (b) and (c)]. That means, each SI pointer can lead us to: 1024*4KB = 4MB worth of data. Since there are 2 SI pointers, the capacity at this level is = 8MB.

The same rationale applies to the DI pointers, except that now we have a deeper tree with 2 levels of internal nodes and 1 level of leaf nodes. Using our knowledge of tree, we can calculate that the leaf level now has 1024^2 nodes giving us: 1024^2 * 4KB = 4GB capacity.

The TI indirect level, in the same way, gives us: 1024^3 * 4KB = 4TB data.

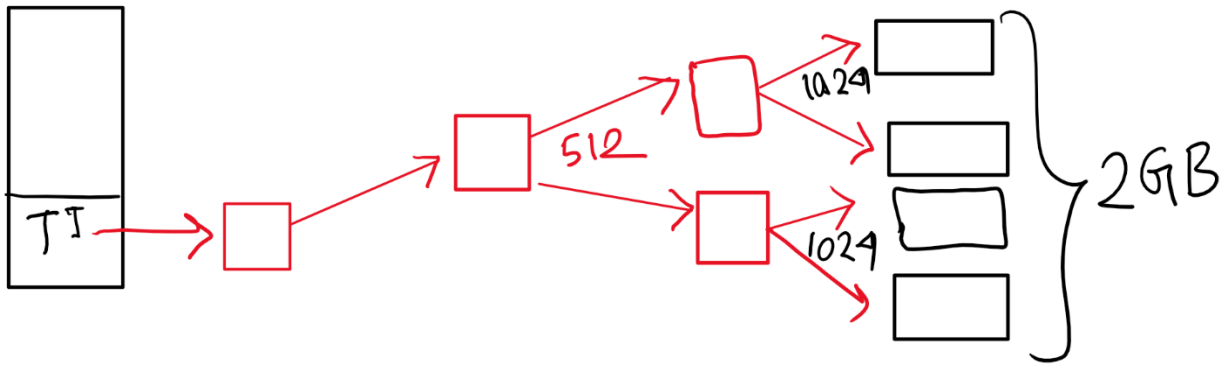
Therefore, the max file size is just all the above added together: 96KB + 8MB + 4GB + 4TB. [In exam, for similar questions, just stop here – no need to calculate any further]

(b)
Now, to represent a max size file, you need to just count the red blocks in the above picture, because they are the ones that do not hold any data. That is my definition of overhead for this type of question. [From that perspective, technically speaking, even the inode itself is an overhead. But we are not considering the inode overhead here, because: (1) I want to keep things simple, (2) an inode does not take an entire disk block.] In other words, you just have to count only the internal nodes for all the pointer trees.

Total overhead = 2 blocks for 2 SI + (1 + 1024) blocks for 1 DI + (1 + 1024 + 1024^2) blocks for 1 TI
= (2 + 1 + 1024 + 1 + 1024 + 1024^2) * 4KB

(c) The total overhead for a 6GB file: Using up to DI level, we can reach a file capacity of (96KB + 8MB + 4GB). That means, about 2GB is yet to be represented using the TI level (note that I am simplifying it a little bit. For exact calculation, you have to subtract the whole underlined amount from 6GB and represent the rest using TI. But we do not want to use the calculator, so it is OK to do such simplification).

Now, to represent the remaining 2GB, you do not need the entire TI level, only part of it. Therefore, the overhead will be less than that of the maximum sized file. In other words, you will only need part of the TI tree and we will only calculate for that part. To get to 2GB, the tree you need looks like the following:



This shows that we are using 1 pointer out of the TI root (the rest 1023 pointers are not needed) and 512 pointers out of the block pointed by that. The remaining 512 pointers are not needed, because we already reach 2GB by fully expanding 1024 ways the last level (i.e., before leaf level) internal nodes. Also, even the partially used pointer blocks are fully overhead (i.e., non-data) because you cannot keep any data there. This leads to the overhead of: (1 + 1 + 512)*4KB worth of disk being “wasted” in the TI

level. You also have to add the overhead from SI and DI levels who total to (2 + 1+ 1024)*4KB. Putting all together, we get an overhead of = (2 + 1 + 1024 + 1 + 1 + 512) * 4KB, whatever that number is.

SIGNALS

32. [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);
}
```

ANSWER:

```
int handler(int s){
   wait (NULL);
}

int main(){
   signal (SIGCHLD, handler);  // install the signal handler here
   if (fork()== 0) // child process
         exit(0);
   else {// parent process
         // or here: signal (SIGCHLD, handler);
         while (true);
   }
}
```

33. [15 pts] Consider the program below and answer the following questions with proper explanation.

a) What is the output? How much time does the program take to run?[10 points]

b) What is the output with line 5 commented? How much time will it take now? [5 points]

```
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 }
```

Answer:

Here, we have a child process that is supposed to runs for ~5 seconds (because of the sleep(1) 5 times from the loop, everything else takes minimal time) and the parent process, if alive, will wait for the child process, because of the wait(0) statement in line 13. Now, whether the parent process will live or not, will depend on the its ability to handle the SIGUSR1. Note that the child always runs for 5 seconds and that does not change. It is the parent process's lifetime that we are trying to determine.

(a) 5 seconds, because you have a signal handler. Output is the prompt 5 times:

```
Got SIGUSR1
Got SIGUSR1
Got SIGUSR1
Got SIGUSR1
Got SIGUSR1
```

(b) ~0 seconds. Because there is no handler for the SIGUSR1, which kills the parent process in the first instance.

34. [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. **[Answer is provided in the below, you only need to provide a full definition of kernel semaphore class]**

```
class KernelSemaphore{
    string name;
    sem_t* sema;
public:
    KernelSemaphore (string _name, int _init_value){
         name = _name; //retain name because the destructor will need it
         sema = sem_open (name.c_str(), O_CREAT, 0644, _init_value);

    }
 void P(){sem_wait (sema);}

 void V(){sem_post (sema);}
    ~KernelSemaphore (){
         sem_close (sema); //removes the process's handle to the sema, but
does not remove from the kernel, which is done in the next step
         sem_unlink(name.c_str()); //note you refer to the name here. That is
why the constructor needs to retain the name of the kernel semaphore
   }
};
// 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;
}
```

35. **[10 pts]** Assume a very old computer system from a company that used a very old legacy device whose path is /dev/legacy/specialdevice and it is about to be decommissioned. 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.

Answer: Make /dev/legacy/specialdevice a symbolic link of the actual path /sys/logfile