IT5002 Comp Sys and Appl 2023 Sem I Final Assessment

Course Name: - **# of Questions:** 15
Categories Used in Exam:

Categories are not used in this exam

Question #: 1

Which ONE of the following statements about kernels is true?

Question #: 2

Which ONE of the following statements about bootstrapping is FALSE?

Question #: 3

Which ONE of the following is NOT a potential source for race conditions between processes or threads?

Question #: 4

Which of the following statements listed in i) to v) is/are TRUE about processes?

i) A program and a process are the same thing.
ii) A process is responsible for finding its own memory when starting up.
iii) The memory allocated to a process can grow or shrink over time.
iv) Processes have their own memory spaces.
v) When a process forks a child process, the child process's variables are either re-initialized or contain unknown values.

Question #: 5

Which of the following statements i) to v) is/are TRUE about process scheduling?

i) A set of processes will finish faster under Shortest Job First versus First-In-First-Out.
ii) The scheduling algorithm used in LINUX ensures as far as possible that a higher priority process will not starve a lower priority process.
iii) Given a set of P of processes p_i each with a deadline d_i, a fixed priority scheduling policy ensures that all processes meet their respective deadlines.
iv) It makes sense to boost processes with large amounts of I/O activity because they are always background processes.
v) Processes that have used significant amounts of CPU time should be upgraded so that they can finish faster.

Question #: 6

During context switching between processes, which of the following i) to v) are saved/restored on the stack?

i) General purpose registers.
ii) Hardware configuration registers.
iii) Program counter.
iv) Status register.
v) Open file descriptors.

Question #: 7

Which of the following statements i) to v) is/are TRUE about synchronization mechanisms?
i) Counting semaphores can cause incorrect behavior if used for mutual exclusion.
ii) Conditional variables prevent the “lost wakes” problem (i.e. the “producer/consumer problem”) even without mutual exclusion.
iii) The “lost wakes” problem found in sleep/wake mechanisms can be solved if used with mutexes.
iv) Barriers are useful to wait for all processes to complete a stage before all progress to the

next stage.
v) Semaphores can be used to implement barriers.

Question #: 8

Which of the following statements i) to v) is/are TRUE about contiguous memory management, with allocation units of 1 byte?

i) The best-fit strategy may increase the risk of external fragmentation.
ii) The worst-fit strategy may increase the risk of internal fragmentation.
iii) It is possible to make the best-fit and worst-fit strategies run in O(1).
iv) The buddy allocation algorithm runs in $O(N^2)$.
v) In the buddy allocation algorithm, any two blocks of free memory of size 2^N can be merged to form a new block of size 2^{N+1}.

Question #: 9

Assuming a FAT32 filesystem with a logical block size of 4KB. What is the largest partition/disk size that this file system can support? (1 GB = 1024 MB, 1 MB = 1024 KB, 1 KB = 1024 bytes)

Question #: 10

Which ONE of the following statements about file systems is TRUE?

Question #: 11

Suppose that we have two processes A and B with a single global variable x in shared memory. x has an initial value of 2. Each process modifies x as shown below:

Process A: x = x + 2;
Process B: x = x * 2;

a. Assume that both processes can execute their statements atomically and there are no race conditions, how many possible values of x are there after both processes execute their respective statements? (For example, If you find that the possible values of x are 3, 4, 4, 6, –these counts as 4 possible values)

Answer: 1

b. Assume that it is no longer possible to execute the statements in each process atomically and there are now race conditions. How many possible values of x are there after both processes execute their respective statements? (For example, If you find that the possible values of x are 3, 4, 4, 6, –these counts as 4 possible values)

Answer: 2

Question #: 12

We have three processes P1, P2 and P3 shown below running on a single CPU which must be shared amongst them.

Process	Ready to run time	CPU time	Priority (part b)
P1	1	5	Lowest
P2	0	8	Middle
P3	3	4	Highest

a. In a system with a round-robin scheduling policy where processes get to run for **3 cycles** each time, how many cycles would each process take, in total, to run? (If a process was ready to run at time 2, and ended running at time 17, it has taken 17 –2 + 1 = 16 cycles).

P1: 1 cycles
P2: 2 cycles
P3: 3 cycles

b. In a system with a fixed-priority preemptive scheduling policy with P1 having the LOWEST priority and P3 the HIGHEST, how many cycles would each process take, in total, to run? (If a process was ready to run at time 2, and ended running at time 17, it has taken 17 –2 + 1 = 16 cycles).

P1: 4 cycles
P2: 5 cycles

P3: __6__ cycles

C. In a system with a shortest-remaining-time non-preemptive scheduling policy where each process gets to run for **2 cycles** each time it is picked, how many cycles would each process take, in total, to run? (If a process was ready to run at time 2, and ended running at time 17, it has taken 17 –2 + 1 = 16 cycles).

P1: __7__ cycles
P2: __8__ cycles
P3: __9__ cycles

Question #: 13

We consider a system that allocates memory in units of 4 bytes (i.e. each allocation unit or AU is 4 bytes) and byte addresses. The system uses a bitmap in an array called freeMemory, where each free AU is marked with a "1" bit, and each used AU is marked with a "0" bit. The freeMemory array is 16 bytes long. The first bit is for the AU starting at address 0, the next bit is for the AU starting at address 4, etc.

a. What is the maximum amount of memory in bytes that this system can manage? State your answer in decimal.

Answer: __1__ bytes

Suppose now that the freeMemory array contains the following values in hexadecimal:

0x3F 0xF0 0x0E 0x78
0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

We have the following memory requests (remember that we are now allocating in units of 4 bytes).

1. a =malloc(14) // Allocate 14 bytes
2. b =malloc(20) // Allocate 20 bytes

3. free(a) // Free request in 1.
4. c =malloc(9) // Allocate 9 bytes
5. d =malloc(8) // Allocate 8 bytes
6. free(b) // Free request in 2
7. e =malloc(18) // Allocate 18 byte

Suppose we allocate using a worst-fit policy:

b. What are the starting addresses of the memory blocks requested in steps 1, 2, 4, 5 and 7? State your answer in decimal.

Step 1 starting address: 2
Step 2 starting address: 3
Step 4 starting address: 4
Step 5 starting address: 5
Step 7 starting address: 6

c. What are the new values in freeMemory after step 7? State just the first four bytes answers in hexadecimal, leaving out the 0x prefix (e.g. if your answer is 0x08, simply fill in 08)

Byte 0: 0x 7
Byte 1: 0x 8
Byte 2: 0x 9
Byte 3: 0x 10

d. What is the total internal fragmentation in bytes after step 7? State your answer in decimal.

Answer: 11 bytes

Question #: 14

We now have a buddy allocation system with a total of 1024 bytes of memory. We repeat the same requests from Question 13, shown here again for your convenience. As before we are using allocation units (AUs) of 4 bytes. Thus free memory entries at entry *avail[i]* indicate free memory blocks of 2^i AUs, and not 2^i bytes.

Where there are two suitable blocks of the same size, the block with the lower starting address is always chosen.

1. a =malloc(14) // Allocate 14 bytes
2. b =malloc(20) // Allocate 20 bytes
3. free(a) // Free request in 1.
4. c =malloc(9) // Allocate 9 bytes
5. d =malloc(8) // Allocate 8 bytes
6. free(b) // Free request in 2
7. e =malloc(18) // Allocate 18 bytes

a. How many AUs are there in this system? State your answer in decimal.

Answer: __1__ AUs

b. What are the starting addresses of the memory blocks in steps 1, 2, 4, 5 and 7? State your answers in decimal.

Step 1 starting address: __2__
Step 2 starting address: __3__
Step 4 starting address: __4__
Step 5 starting address: __5__
Step 7 starting address: __6__

At the end of step 7, what is the total internal fragmentation in bytes? State your answer in decimal.

Answer: __7__ bytes

Question #: 15

We have an inode-based file system where the logical block size is 2048 bytes (2 KB). The first 32 blocks of the file system are reserved for various data structures like the boot block, directory, etc. In particular blocks 16 to 31 hold the bitmap that indicates free data blocks with a “1”, and occupied data blocks with a “0”. Data blocks begin at block 32, so the very first bit of the bitmap indicates the status of block 32, the second bit indicates the status of block 33, etc.

Consider the following example of the **first 8 bits** of this bitmap:

0	0	0	1	1	1	1	0

As data blocks begin at block 32, this means that blocks 32 to 34 and 39 are occupied, while blocks 35 to 38 are free.

a. What is the maximum disk or partition size possible on this file system excluding the blocks reserved by the file system? Express your answer in decimal in MB and in decimal (1 MB = 1024 KB, 1 KB = 1024 bytes)

Answer: __1__ GB

b. What are the first and last data block numbers of the data blocks represented by this bitmap? State your answers as decimal numbers without any punctuation. For example if your answer is 1,234,567, write it simply as 1234567

First data block number: __2__
Last data block number: __3__

The first 16 bytes of the free blocks bitmap is shown below in binary:

0b00000000 0b00000011 0b11111100 0b00001100
0b00011111 0b11111111 0b00001100 0b00000000
0b11001111 0b00000000 0b11110000 0b00001111
0b11111111 0b10101010 0b00001111 0b10101111

Our OS provides the following calls to create/open, read, write, and close a file, as well as to move to a particular byte in the file.

fd = open(filename): Opens a file for reading and writing, returning a file descriptor to fd
lseek(fd, bytenum, from): Moves the file pointer to byte number bytenum, which can be negative. The "from" argument can take three values:

START: Move to bytenum bytes from the start of the file
CURR: Move to bytenum bytes from the current file pointer position.

read(fd, buffer, count): Read count bytes from file into buffer.
write(fd, buffer, count): Write count bytes from buffer to file.
close(fd): Close the file

The inode and index and data blocks are allocated using the free blocks bitmap on a "first available block" basis. So if we are creating a file, we need to:

I) Find the first free data block for the inode.

ii) Create an entry in the directory with the filename and inode block number.
iii) Find further free blocks for the data, and for lower level index blocks if necessary.

Consider the following program, where test.txt does not exist yet and is created by our program. Our free block bitmap is as shown earlier:

```
buffer=[0]
buffer1="Hello"
buffer2= "world."

fd = open("test.txt");

for i in range(1024):
      write(fd, buffer1, 5);

lseek(fd, 2141, CURR)

for i in range(128):
      write(fd, buffer2, 6);
```

c. The directory entry for the filesystem just consists of the filename and the inode block number. Fill in both fields below:

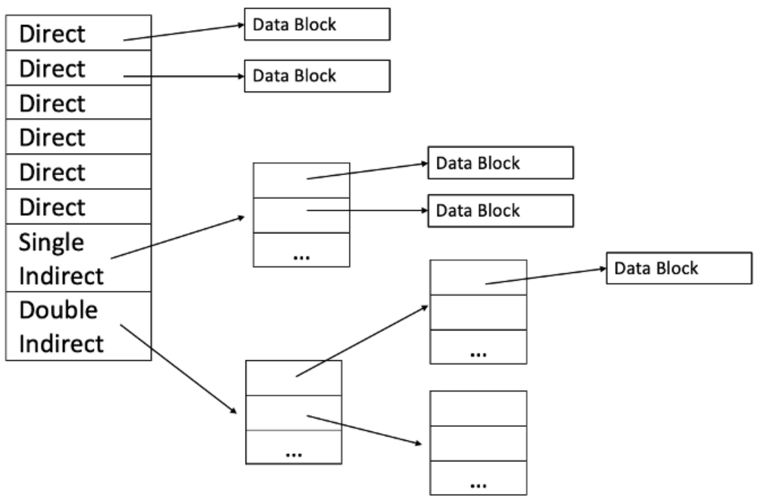
Filename: __4__

inode block number: __5__

d. Assuming that all block numbers in our filesystem is represented by a 24-bit number. Further, we assume that data blocks that are used as index blocks will only contain block numbers and nothing else. What is the maximum number of index/data blocks that a single index block can point to? State your answer in decimal.

Answer: __6__ pointers.

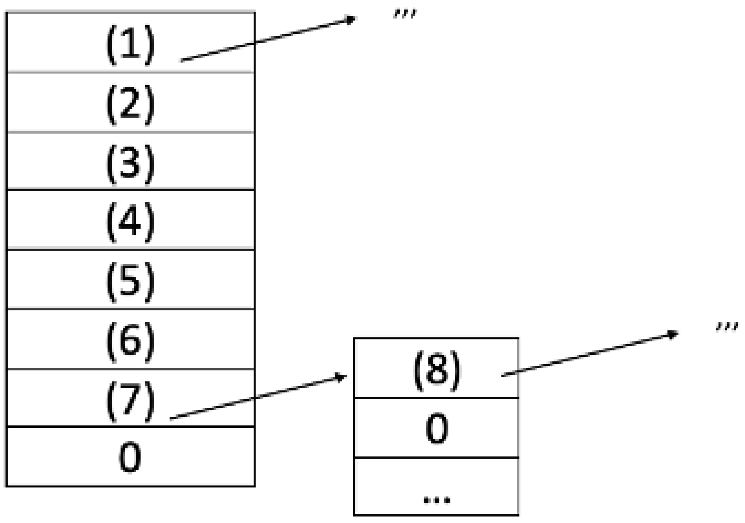
Our inode structure is illustrated below (not all data/index blocks are shown). There are six direct pointers, one single indirect pointer, and one double indirect pointer.



e What is the maximum file size in MB possible with this structure? State your answer correct to 2 decimal places. 1 MB = 1024 KB, 1 KB = 1024 bytes.

Answer: 7

f. The diagram below shows the inode structure. The (1), (2) etc. in the structures are labels that correspond to blanks that you need to fill in. We run the program above to completion. Fill in the block numbers of the data blocks used by the program in the blanks provided. If an entry in the inode is not pointing to any data block, fill in 0.



(1): 8

(2): 9

(3):	10
(4):	11
(5):	12
(6):	13
(7):	14
(8):	15