Operating Systems - Monsoon 2022
Final Examinations

Arani Bhattacharya, Sambuddho Chakravarty

Date: Dec 12, 2022 (1500hrs. – 1800hrs.)

Student's Details

Name of the Student:

Roll Number:

Stream:

Question Structure and Instructions

This paper consists of two sections:

1. The first section consists of 10 multiple choice questions, each bearing 2 points.
2. The second section consists of 10 short conceptual questions each carrying 3 points.
3. The third section consists of 2 longer questions each carrying 5 points.

You are required to answer the questions in the question paper ONLY for Section A. You will be given additional answer sheets to write the answers of Sections B and C.

1 Multiple Choice Questions (2x10 = 20 points)

`2 points for correct answer; no points for wrong answer`

A1 Suppose we require a type of interprocess communication (IPC) where fast performance is crucial, but consistency is not so important. What is the best IPC mechanism that should be used and why?

(a) Non-blocking sockets, because they the sender or reciever doesn't have to block for the task to complete.

(b) FIFOs because they are more efficient than both file reads and writes as well as sockets, while still using the same set of system calls.

(c) Non-blocking sockets, because processes check if the system call blocks or not and this gives the calling process the ability to yield, unlike blocking sockets.

(d) Shared memory because no systems calls are needed to read or write, while the kernel also does not enforce any consistency.

(e) None of the above.
Option: `(d)`

Roll Number:

A2 Suppose we have a machine on which no hardware support is available for interprocess synchronization. We are required to ensure mutual exclusion to a shared variable on this machine. Can we use a single lock variable in C to ensure mutual exclusion?

(a) Yes it can be by disabling interrupts while that shared variable is being updated. This is in fact a common mechanism used in modern operating systems with low-end CPUs, that have no such hardware lock operations available.

(b) Yes, there are some architecture's, especially the embedded ones that are resource contrained and implement atomic operations using temporary processor to suspend execution, until the critical section execution is over.

(c) No, there is no way to do that in such cases, and high-level primitives must be used.

(d) It depends. It may work out in cases where the critical section isn't long and the execution ends soon and has no sleeps and blocks. However, in other cases it may not work.

(e) Only (a).

(f) Only (b).

(g) Only (c).

(h) Both (a) and (b)

(i) Both (a) and (c)

(j) (a), (b) and (c) but not (d).
Option: `(a/e)`

A3 Consider a case of writing data file to a disk. Which of the following parts of the hardware or OS does it need to go through?

(a) System call API → syscall interrupt handler → filesystem → persistent disk.

(b) System call API → syscall interrupt handler → filesystem → VFS layer → disk.

(c) System call API → syscall interrupt handler → device driver → filesystem → VFS layer → disk.

(d) System call API → syscall interrupt handler → VFS layer → filesystem → device driver → disk.

(e) Depends on the type of storage, *i.e.* magnetic disks or SSDs. (a) for magnetic disks and (c) for SSDs.

(f) Depends on the type of storage, *i.e.* magnetic disks or SSDs. (b) for magnetic disks and (d) for SSDs.

(g) None of the above.
Option: `(d)`

A4 Which of the following is true about the interplay between virtual memory subsystem (memory management unit+kernel data structures) and VFS:

(a) MMU (and the kernel) translates virtual memory to real memory block which finally points to the data structures represented by VFS (*e.g.* inodes).

(b) MMU (and the kernel) takes care of page replacement policies while ignoring the data blocks represented by VFS structures and thus potentially causes starvation.

(c) MMU (and the kernel) relies on page faults to identify pages to import from the swap space and replace pages back to it (so as to make room for new pages). VFS uses this principle to also import file data blocks from the on disk file system and places them on pages allocated.

(d) None of the above
Option: `(c)`

A5 The following is true about the "dirty" bit in page descriptors:

(a) It is used to keep track of pages that may be replaced by data corresponding to any other process (that may have page faulted).

(b) It is used to keep track of pages that may be *exchanged* with pages in the swap memory.

(c) It is used to keep track of pages for a corresponding frame is allocated in the RAM.

(d) None of the above.
Option:(a)

A6 The following is the difference between pthread_mutex_lock() and advisory locks (e.g. flock()):

(a) Pthread_mutex_lock() can be both blocking and non-blocking while flock() is always blocking.

(b) Advisory locks like flock() and pthread_mutex() work exactly the same way, they differ only in the arguments they use.

(c) Pthread_mutex_lock() can only be used in multithreaded programs, while flock() can be used in all cases.

(d) Pthread_mutex_lock() causes the calling thread to sleep, if some other process has acquired the lock, while flock() does not.

(e) None of the above.
Option:(e)

OR

Which of the following problems does the Linus Elevator scheduler of disk scheduler suffer from?

(a) Low throughput

(b) Starvation

(c) Unnecessary movement of the disk head

(d) All of the above

(e) None of the above
Option: (b)

A7 The main reason to store attributes of a file with the inodes (instead of managing centrally with a common data structures) is:

(a) To be able to maintain consistency across all files. The OS need not worry about the attributes.

(b) The attributes are actually stored in different data structures within the filesytems first few sectors, that makes it easy to retrieve them whenever required.

(c) This is a security step, done so as to protect the attributes and permissions from being changed by malicious actors. The inode structure stores a cryptographic checksum that can be verified if the attributes and permissions have been tampered with or not.

(d) (a) and (c)

(e) (b) and (c)

(f) None of the above.
Option: (f)

OR

Consider a inode-based file system that does not allow any indirection to be used. Which of the following is true about such a file system?

(a) There would be excessive fragmentation of files

(b) Accessing files would become slower

(c) The maximum size of files would be relatively small

(d) Relatively small number of files can be stored

(e) Both (a) and (b)

(f) Both (a) and (c)
Option: (b)

A8 The memory ranges visible to Interrupt handlers are:

(a) Virtual address ranges.

(b) Corresponding to both kernel and user processes, since interrupt handlers run on supervisor (OS) mode.

(c) The real addresses and not the virtual address, as the latter is only for application programs.

(d) Both (b) and (c).

(e) None of the above.
Option: (a)

OR

Suppose we have a 24-bit virtual memory address, with the most significant 12 bits representing the virtual page number. What is the size of a page frame, assuming the memory is byte-addressable and there is a single-level page table?

(a) 1 KB

(b) 2 KB

(c) 4 KB

(d) 512 bytes

(e) Cannot be calculated from the given information
Option: (c)

A9 Which of the following is true about file system mounting:

(a) File system partitions can be mounted at specific directories called mount points.

(b) Devices that do not have file systems can also be mounted to the filesystem, e.g. devices like raspberry pi that have inbuilt storage, like what is done for every filesystems.

(c) Files can also be mounted as partitions on specific mount points.

(d) (a) and (b).

(e) (a) and (c).

(f) All three of them (a), (b) and (c).

(g) None of the above.
`Option: (e)`

A10 Which of the following is not true about on-demand paging and subsequent page faults:

(a) The `'segmentation fault'` error that one encounters is an example of a page fault, occurs when a process access memory location not allocated to it.

(b) They have significance only in the context of a memory allocation and its access, wrt a process.

(c) They could also occur in the context of various block and character devices, *i.e.* when a process tries to read or write, to or from, such devices.

(d) None of the above.
Option: `(a)`

2 Short Conceptual Questions (3x10=30 points)

There are 10 questions in this section each worth 3 points. You may write the answers to these questions succintly.

Roll Number:

B1 Consider a situation where we have three global variables x, y and z used by three programs A, B and C as shown below:

```
Program A:              Program B:              Program C:
    sem_wait(semX);         sem_wait(semY);         sem_wait(semZ);
    x = x + 1;              y = y + 1;              z = z + 1;
    sem_wait(semY);         sem_wait(semZ);         sem_wait(semX);
    y = y + x;              z = z + y;              x = x + z;
    sem_signal(semX);       sem_signal(semY);       sem_signal(semZ);
    sem_signal(semY);       sem_signal(semZ);       sem_signal(semX);
```

Does the above code ensure mutual exclusion? Is there any bug in the above code, and if so what type of bug and why?

Yes, The above code ensures mutual exclusion. Yes, it has a bug in it as it can lead to deadlock because the semaphores are used in a way that it can lead to circular wait.

(a) Yes, ensures mutual exclusion (either one) – 1 mark

(b) Yes, it has a bug (either one) – 1 mark

(c) Circular wait OR the semaphores used in a way/manner – 1 mark

B2 Current x86-64 systems allow only 48-bit virtual addresses, even though allowing 64-bit virtual addresses would allow a larger virtual address space. What needs to be changed in the memory management system to accommodate 64-bit virtual addresses, and what would be its disadvantage?

A 64-bit virtual address would require have larger virtual address space, so it would require more levels of paging. This would have the disadvantage of increasing the number of memory accesses while accessing any data.

(a) Larger virtual address space requirement OR more level of paging – 1.5 mark

(b) Increase in the number of memory accesses – 1.5 marks

B3 Suppose you have a set of programs which does not satisfy the principle of locality, but refer to a diverse set of page ranges. Is utilizing demand paging the right way of allocating pages to programs, or should pages be allocated proactively for each process? You may assume that proactive allocation follows an algorithm of fetching pages close to the page for which page fault occurred.

Demand paging is the right way. This is because proactive allocation would fetch the wrong pages as principle of locality is not satisfied, leading to unnecessary fetches of pages.

(a) Demand paging is the right way – 1.5 marks

(b) Proactive allocation would fetch wrong pages/unnecessary fetches of pages – 1.5 marks

B4 Suppose there is a hard disk which has a large number of files, whose sizes keep changing frequently due to frequent writes. Would it be beneficial to allocate blocks using inodes, or can you suggest an alternative strategy?

No, inodes are not a good idea as they do not lead to continuous allocation of data blocks. A better idea is to use any system that ensures continuous allocation, such as extants.

(a) No / inodes are not a good idea – 1.5 marks

(b) Ensure continuous allocation / use extants – 1.5 marks

B5 Assume that it is 1990s and you are working as the chief designer for Intel corporation. What if you are were given the task of re-designing the 80386 CPU. Suggest one change for the existing MMU that would ensure that the processor could work equally well for varied kinds of OSes and applications. Justify your responses.

First thing to be done would be to remove the support for the segmentation. Back in the 1990s most OSes, barring few exceptions were not using segmentation hardware. The segmentation registers and their reference was an artifact from the earlier 8086 architecture. Segmentation, by the very design doesn't allow for a truly flat address space (like the memory model corresponding to what C language believes it to be). The contiguity can be maintained, while still keeping separation of different memory ranges for the individual processes. The memory access times either ways remain constant. Further, fewer physical memory is required to manage the memory management subsystems (*i.e.* page and segment tables)

(a) Remove segmentation – 1.5 marks

(b) Any of the following : breaks the flat memory model of C language; contiguity is not maintained ; more space required by the memory management subsystem – 1.5 marks

OR

Consider a case where a program is writing to a file backwards. Is Linus Elevator scheduler capable of optimizing such writes, and what is the strategy used to optimize them?

Yes, Linus Elevator is capable of optimizing such writes via use of front merge.

(a) Yes, Linus Elevator is capable – 1.5 marks

(b) Use of front merge – 1.5 marks

B6 Is it possible for a process to remain in running state even though it has requested for input/output event, without spinning in a busy wait loop? If it is possible, then explain what is the technique. If it is not possible, then is there an alternative technique to optimize input/output?

Yes, it is possible as once input/output is requested, the process needs to go to the waiting state, unless if asynchronous input/output is used. OR No, it is not possible in general, but the alternative technique possible is asynchronous input/output.

(a) Asynchronous input/output is used. Signals are a classic example of the same – 3 marks

B7 Suppose you have two processes A and B, which are currently in ready and running state respectively. Process B tries to access some memory and faces a page fault. Now which states do the processes A and B go into? Would your answer change if B had faced a cache miss instead of a page fault?

Process B goes to waiting state and Process A goes to running state. In case of cache miss, Process A stays in ready state and Process B stays in running state.

(a) Process B goes to waiting state – 1 mark

(b) Process A goes to running state – 1 mark

(c) Process A and B both stay in the same state in case of cache miss – 1 mark

B8 Suppose we have a two-level data cache called L1 and L2, and a two-level TLB called T1 and T2. What will be the sequence of access requests to them, if the addresses and the data are in none of the caches. Does the above sequence change if the processor is in 16-bit (real) mode?

First to T1, then to T2, then to page table in main memory, then to L1, then to L2, then to physical address in main memory. In case of 16-bit mode, the page tables do not work, so it is only through L1 and L2.

(a) T1, then T2 – 1 mark

(b) L1, then L2 – 1 mark

(c) For 16-bit, only L1 then L2 – 1 mark. Mentioning main memory is optional.

B9 Consider a case of writing to a pipe. Can a user seek to any position of the pipe using the `seek()` function? Why or why not?

No, it is not possible to do a seek, because it is a special type of file on which seeks are disabled, and reading has to be done character-by-character.

(a) No, it is not possible – 1.5 marks

(b) EITHER seek disabled OR reading character-by-character/character-wise/single character – 1.5 marks

B10 Suppose we create a new process using `fork()` and then make changes to a global variable within the process in both the child and parent processes. Will there be a race condition? Justify.

No, there will be no race condition. This is because as soon as a change is made to the global variable, a copy-on-write which would make a copy of the variable. So the changes would happen on different copies.

(a) No race condition – 1.5 marks

(b) Copy of global variable made /copy-on-write – 1.5 marks

3 Long Questions (2x5 = 10 points)

There are 2 questions in this section each worth 5 points.

C1 (a) Suppose you want to transmit data from one system to another, over a local network. One way of doing it is to write this data to a file, and then use a socket to send the data. Is this the right way to do it, or is a better/faster technique available?

A faster technique is available. Write directly to the socket without first writing to the file.

i. Write directly / without writing to file – 2.5 marks

(b) Suppose you want two processes on the same system to communicate using messages, but the sequence in which the messages are received is not important. What is the right IPC mechanism to use, assuming that you do not want the messages to be fragmented?

Datagram sockets – 2.5 marks

C2 (a) Is it possible that a power failure after `write()` system call has returned, results in data loss? If so, why does it happen and is there any change possible in the filesystem to ensure that it does not happen?

Yes, it is possible because often returns are completed after writing to cache. In those cases, data losses can happen. Journaling can be used to avoid such a problem.

Yes, possible – 1 mark

ii. Returns after writing to cache – 1 mark

iii. Journaling/keeping track of writes in separate block – 0.5 mark

Assume that the above proposed change is NOT utilized in a specific filesystem. What do you think is the right way to recover the filesystem to a state where reading and writing are again possible? Is there a disadvantage of using such a recovery technique?

Checking of disk/filesystem for any inconsistencies (fsck) is needed. It can take a lot of time as all the inodes need to be checked one-by-one.

Checking of disk/filesystem for inconsistencies/fsck – 1.5 marks

(b) ii. Can take a lot of time – 1 mark