**Santa Clara University**

Department of Computer Engineering

Advanced Operating Systems (COEN383) – Midterm Preview (30 points) Time: 75 minutes

**Name: Tejas Jayesh Dedhiya**

**SCU ID: W1605246**

**Grades**

|  |  |  |
| --- | --- | --- |
| Problem 1 | 6 |  |
| Problem 2 | 6 |  |
| Problem 3 | 6 |  |
| Problem 4 | 6 |  |
| Problem 5 | 6 |  |
| Total | 30 |  |

1. Understand process scheduling and understand how long time it would take to execute jobs sequentially vs. in parallel? – 6pts?

# Answer)

* Process scheduling is an essential part of a Multiprogramming operating systems.
* It is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process based on a particular strategy.
* An operating system uses a program scheduler to schedules the processes of computer system. The schedulers are of following types:

**Long term scheduler:** The long-term scheduler basically decides the priority in which processes must be placed in main memory. Processes of long-term scheduler are placed in the ready state because in this state the process is ready to execute waiting for calls of execution from CPU which takes time that’s why this is known as long term scheduler

**Mid - term scheduler:** It places the blocked and suspended processes in the secondary memory of a computer system.

**Short term scheduler:** It decides the priority in which processes is in the ready queue are allocated the central processing unit (CPU) time for their execution.

* An operating system uses two types of scheduling processes execution:

1. **Preemptive process:**

In preemptive scheduling policy, a low priority process must suspend its execution if high priority process is waiting in the same queue for its execution.

**2. Non - Preemptive process:**  
 In non - preemptive scheduling policy, processes are executed in first come first serve basis, which means the next process is executed only when currently running process finishes its execution.

* Operating system performs the task of scheduling processes based on priorities using these following algorithms:

**1)**[**First come first serve (FCFS)**](https://www.includehelp.com/operating-systems/fcfs-first-come-first-serve-scheduling-algorithm.aspx)  
 In this scheduling algorithm the first process entered in queue is processed first.

**2)**[**Shortest job first (SJF)**](https://www.includehelp.com/operating-systems/sjf-shortest-job-first-scheduling-algorithm.aspx)In this scheduling algorithm the process which requires shortest CPU time to execute is processed first.

**3) Shortest Remaining Time First (SRTF) scheduling**  
This scheduling Algorithm is the preemptive version of the SJF scheduling algorithm. In this, the process which is left with the least processing time is executed first.

**4) Longest Job First (LJF)**  
In this type of scheduling algorithm, the process with the maximum time required to execute is scheduled first. In this type of scheduling is not widely used because it is not a very effective way of scheduling, as the average turn-around time and the average waiting time are maximum in this case.

**5) Longest Remaining Time First (LRTF)**  
As SRTF is to SJF, LRTF is the preemptive version of the LJF scheduling algorithm.

**6) Priority scheduling**  
In this scheduling algorithm the priority is assigned to all the processes and the process with highest priority executed first. Priority assignment of processes is done based on internal factor such as CPU and memory requirements or external factor such as user’s choice. The priority scheduling algorithm supports preemptive and non - preemptive scheduling policy.

**7) Round Robin (RR) scheduling**  
In this algorithm the process is allocated the CPU for the specific time period called **time slice**, which is normally of 10 to 100 milliseconds. If the process completes its execution within this time slice, then it is removed from the queue otherwise it must wait for another time slice.

**Time to execute jobs sequentially vs parallel:**

"If each job has 50% I/O wait, then it will take 20 minutes to complete in the absence of competition. If run sequentially, the second one will finish 40 minutes after the first one starts. With two jobs, the approximate CPU utilization is 1 − 0.52. Thus, each one gets 0.375 CPU minute per minute of real time. To accumulate 10 minutes of CPU time, a job must run for 10/0.375 minutes, or about 26.67 minutes. Thus, running sequentially, the jobs finish after 40 minutes, but running in parallel they finish after 26.67 minutes."

50% I/O wait time means that a process is not in execution (i.e., CPU is sitting idle) for 50% of the **total time a process requires from CPU to complete itself (its execution)**. Thus, CPU Utilization turns out to be //whereas 50% I/O time means it needs 50% of total execution time (10 minutes) to complete its I/O.

50%=50/100 = .5

Thus, the time needed to complete a process which requires 10 minutes of CPU will be = *CPU time required by process/CPU utilization*=

10/CPU utilization= 10/0.5 = 20 minutes.

when two processes run sequentially (one after the other) then the total time required will be = 10/0.5 + 10/0.5 = 20+20=40 minutes (ANSWER)

**in case of parallel execution, we again find the CPU utilization**. since two processes are in parallel thus the formulae become--> ***{1-(I/O time) ^no. of processes in parallel execution}*** =1-(0.5) ^2=1-.25=0.75 now the CPU utilization for 1 process will be 0.75/2=0.375 Therefore the time required will be = CPU time required by process/CPU utilization= 10/0.375=26.67 minutes.

Since the two processes are running in parallel thus the time required by 1 process will be the total time required by 2 processes=26.67 minutes (ANSWER)

**2.** Understand memory mapping and the different levels of mapping that can occur [6 pts]

# Answer)

Memory mapping is a process or command in computer programming that **requests that files, code, or objects be brought into system memory**. It allows files or data to be processed temporarily as main memory by a central processing unit.

Memory mapping is the translation between the logical address space and the physical memory. The objectives of memory mapping are **(1) to translate from logical to physical address**, (2) to aid in memory protection (q.v.), and (3) to enable better management of memory resources.

Different levels of Mapping:

1. **Direct Mapping**

* In Direct mapped cache memory, each block mapped to exactly one location in cache memory.
* A particular block of main memory can map the line number of cache is given by - Cache line number = (Block Address of Main Memory) modulo (Number of lines in Cache).
* The direct-mapped cache is like rows in a table with three columns' main memory address are bits for Offset, Index, and Tag. The size of the fields depends on the capacity of memory and size of the block in the cache.
* The least significant w bits are used to identify a word within a block of main memory. Tag corresponds to the remaining bits are used to determine the proper block of main memory. Line off-set or block is used to select a block to be accessed out of total blocks are available according to the capacity of the cache.
* The data block or cache line that contains the actual data fetched and stored, a tag with all or part of the address of the data that was fetched, and a flag bit that shows the presence in the row entry of a valid bit of data.

2**) Associative Mapping**

* In this type of mapping, any main memory block can go in any line of the cache. So, we have to use proper replacement policy to replace a block from the cache if the required block of main memory is not present in the cache. Here, the main memory is divided into two fields: word field identifies which word in the block is needed and the tag field identifies the block. It is the fastest and the most flexible mapping form of cache mapping.

3) **Set-associative Mapping**

* In this mapping technique, blocks of cache are grouped to form a set and a block of main memory can go into any block of a specific set.
* This form of mapping removes the drawbacks of direct mapping. In Set-associative mapping, each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping is a combination of direct and associative cache mapping techniques
* This also reduces searching overhead present in the associative mapping. Here, searching is restricted to the number of sets instead of the number of blocks.

3. How do you recover disk metadata in memory if you have system crash? [6 pts]

# Answer)

## **Approach #1: check consistency during reboot, repair problems**

* Example: **Unix fsck ("file system check")**
  + During every system boot fsck is executed.
  + Checks to see if disk was shut down cleanly; if so, no more work to do.
  + If disk didn't shut down cleanly (e.g., system crash, power failure, etc.), then scan disk contents, identify inconsistencies, repair them.
  + Example: block in file and in free list
  + Example: reference count for a file descriptor doesn't match the number of links in directories
  + Example: block in two different files
  + Example: file descriptor has a reference count > 0 but is not referenced in any directory.
* Limitations of fsck:
  + Restores disk to consistency but doesn't prevent loss of information; system could end up unusable.
  + Security issues: a block could migrate from the password file to some other random file.
  + Can take a long time: 1.5 hours to read every block in a medium-size disk today. Can't restart system until fsck completes. As disks get larger, recovery time increases.

## **Approach #2: ordered writes**

* Prevent certain kinds of inconsistencies by making updates in a particular order.
  + For example, when adding a block to a file, first write back the free list so that it no longer contains the file's new block.
  + Then write the file descriptor, referring to the new block.
  + What can we say about the system state after a crash?
  + In general:
    - Never write a pointer before initializing the block it points to (e.g., indirect block).
    - Never reuse a resource (inode, disk block, etc.) before nullifying all existing pointers to it.
    - Never clear last pointer to a live resource before setting new pointer (e.g., mv).
* Result: no need to wait for fsck when rebooting
* Problems:
  + Can leak resources (run fsck in background to reclaim leaked resources).
  + Requires lots of synchronous metadata writes, which slows down file operations.
* Improvement:
  + Don't write the blocks synchronously but record dependencies in the buffer cache.
  + For example, after adding a block to a file add dependency between file descriptor block and free list block.
    - When it's time to write the file descriptor back to disk, make sure that the free list block has been written first.
  + Tricky to get right: potentially end up with circular dependencies between blocks.

## **Approach #3: write-ahead logging**

* Also called ***journaling file systems***
* Implemented in Linux ext3 and NTFS (Windows).
* Similar in function to logs in database systems; allows inconsistencies to be corrected quickly during reboots
  + Before performing an operation, record information about the operation in a special append-only log file; flush this info to disk before modifying any other blocks.
  + Example: adding a block to a file
    - Log entry: "I'm about to add block 99421 to file descriptor 862 at block index 93"
  + Then the actual block updates can be carried out later.
  + If a crash occurs, replay the log to make sure all updates are completed on disk.
  + Guarantees that once an operation is started, it will eventually complete.
  + Problem: log grows over time, so recovery could be slow.
  + Solution #1: checkpoint
    - Occasionally stop and flush all dirty blocks to disk.
    - Once this is done, the log can be cleared.
  + Solution #2: keep track of which parts of the log correspond to which unwritten blocks; as blocks get written to disk, can gradually delete old portions of the log that are no longer needed.
  + Typically, the log is used only for metadata (free list, file descriptors, indirect blocks), not for actual file data.
* Logging advantages:
  + Recovery much faster.
  + Eliminate inconsistencies such as blocks confused between files.
  + Log can be localized in one area of disk, so writes are faster (no seeks).
  + Metadata writes can be delayed a long time, for better performance.

4. How UNIX can add new device dynamically without any need for recompiling the OS? [6 pts]

# Answer)

* This is done via a technique known as a loadable driver
* The actual hardware driver is loaded into the system dynamically. Since devices can be loaded and unloaded, they do not have to be compiled into the system kernel.
* Linux operating systems contain a table, called the device table, which is indexed according to the number of devices.
* Each device is represented in the table by a device data structure; for each entry in the table, there is a pointer for all functions of the device.
* These pointers maintain the entry for read, write, open, close, etc., so that entries can be updated, added, and removed. When a new device is added and installed on the operating system, a new entry is made in the table. Then, pointers are filled in for the drivers of the new device.

Installing a new device in Unix is just a matter of creating a special file in the /dev/ directory using mknod (1).

If the device belongs to a new device class, you need a new device driver. In older Unix systems you had to relink your kernel and reboot. This was a version that was closed source, so all the drivers were precompiled. You might have to recompile your kernel in the oldest Unix versions.

Linux will detect you plugged in a new device and load the appropriate kernel module and create the necessary device node automagically. If you install a new kernel version, it may recompile some device drivers first to create a module for the new kernel.

5. Understand how the OS performs IO on disk? [6 pts]

# Answer)

The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU. While using memory mapped IO, OS **allocates buffer in memory and informs** I/O device to use that buffer to send data to the CPU.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

* **Block devices** − A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
* **Character devices** − A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc.

## **Device Controllers**

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.

## **Synchronous vs asynchronous I/O**

* **Synchronous I/O** − In this scheme CPU execution waits while I/O proceeds
* **Asynchronous I/O** − I/O proceeds concurrently with CPU execution

## **Communication to I/O Devices**

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

* Special Instruction I/O
* Memory-mapped I/O
* Direct memory access (DMA)

### **Special Instruction I/O**

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

### **Memory-mapped I/O**

When using memory mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.

While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

## **Direct Memory Access (DMA)**

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So, a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.

The operating system uses the DMA hardware as follows −

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Device driver is instructed to transfer disk data to a buffer address X. |
| 2 | Device driver then instruct disk controller to transfer data to buffer. |
| 3 | Disk controller starts DMA transfer. |
| 4 | Disk controller sends each byte to DMA controller. |
| 5 | DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero. |
| 6 | When C becomes zero, DMA interrupts CPU to signal transfer completion. |

## **Polling vs Interrupts I/O**

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both techniques allow the processor to deal with events that can happen at any time and that are not related to the process it is currently running.

### **Polling I/O**

Polling is the simplest way for an I/O device to communicate with the processor. The process of periodically checking status of the device to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

Most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program. This is an inefficient method and much of the processors time is wasted on unnecessary polls.

Compare this method to a teacher continually asking every student in a class, one after another, if they need help. Obviously the more efficient method would be for a student to inform the teacher whenever they require assistance.

### **Interrupts I/O**

An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention.

A device controller puts an interrupt signal on the bus when it needs CPU’s attention when CPU receives an interrupt, it saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.