**2017 os hi**

**(a) Fill in the blanks:**

1. From the system’s perspective, the role of an operating system is to act as **a resource manager**.
2. ‘.’ is the relative path of **current** directory.
3. **Environment variables** are not inherited by a child process from its parent.
4. An un-named pipe can be destroyed by **closing all its file descriptors**.
5. Because of **Belady's** anomaly sometimes page faults increase instead of decreasing.

**(b) Names of Two:**

1. **System calls of Linux relevant for devices but not for files:** ioctl, mmap.
2. **APIs of Windows O/S related to process management:** CreateProcess, TerminateProcess.
3. **Techniques of memory management which allocate contiguous memory:** Contiguous Memory Allocation.
4. **Non-shareable resources:** Printer, CPU.

**(c) Differentiation:**

1. **Privileged user, super user, and ordinary user:**
   * A **privileged user** typically refers to a user with more permissions than a regular user but fewer than the superuser. They might have permissions to access certain resources or perform specific tasks that ordinary users cannot.
   * A **superuser** (often called root in Unix/Linux systems) has unrestricted access to all resources and commands on a system.
   * An **ordinary user** has limited permissions and cannot perform tasks that could potentially harm the system or other users.
2. **Process and thread:**
   * A **process** is an executing instance of a program, consisting of its code, data, and resources. It has its own address space and resources.
   * A **thread** is a lightweight process within a process. Threads share the same memory space and resources as the process they belong to but have their own execution context.

**(d) Calculation of Average Turn Around Time and Average Waiting Time:**

1. **SRTF (SRJF):**
   * Average Turn Around Time: 0+14+30+404=21.040+14+30+40​=21.0
   * Average Waiting Time: 0+6+10+174=8.2540+6+10+17​=8.25
2. **Pre-emptive priority-based:**
   * Average Turn Around Time: 0+25+36+404=25.2540+25+36+40​=25.25
   * Average Waiting Time: 0+6+15+174=9.540+6+15+17​=9.5
3. **Round Robin (time quantum = 5):**
   * Average Turn Around Time: 14+25+29+204=22.0414+25+29+20​=22.0
   * Average Waiting Time: 0+6+8+94=5.7540+6+8+9​=5.75

**(e) Implementation in C:**

cCopy code

#include <stdio.h> #include <stdlib.h> #include <unistd.h> int main() { int pipefd[2]; pid\_t pid; // Create pipe if (pipe(pipefd) == -1) { perror("pipe"); exit(EXIT\_FAILURE); } // Fork child process pid = fork(); if (pid == -1) { perror("fork"); exit(EXIT\_FAILURE); } else if (pid == 0) { // Child process // Duplicate standard output to pipe write end dup2(pipefd[1], STDOUT\_FILENO); close(pipefd[0]); // Close unused read end of pipe close(pipefd[1]); // Close pipe write end // Execute sort command execlp("sort", "sort", (char \*)NULL); perror("execlp"); exit(EXIT\_FAILURE); } else { // Parent process // Wait for child process to complete wait(NULL); // Duplicate pipe read end to standard input dup2(pipefd[0], STDIN\_FILENO); close(pipefd[1]); // Close unused write end of pipe close(pipefd[0]); // Close pipe read end // Redirect standard error to errorop file int fd = open("errorop", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644); if (fd == -1) { perror("open"); exit(EXIT\_FAILURE); } dup2(fd, STDERR\_FILENO); close(fd); // Close file descriptor // Execute sort command again execlp("sort", "sort", (char \*)NULL); perror("execlp"); exit(EXIT\_FAILURE); } return 0; }

**Q.2 (a) Examples:**

1. **Dedicated O/S:**
   * Real-time operating systems used in industrial control systems.
   * Operating systems embedded in specific devices like microwave ovens or digital cameras.
2. **Multiuser O/S:**
   * Linux
   * Unix
3. **Boot loaders:**
   * GRUB (Grand Unified Bootloader)
   * LILO (LInux LOader)
4. **Shells of Linux:**
   * Bash (Bourne Again SHell)
   * Zsh (Z Shell)
5. **Services provided by O/S for system administrators:**
   * Process management
   * Memory management
   * File system management
   * User management

**(b) True or False:**

1. **(i) An operating system is an event-driven program:** False. Operating systems are not inherently event-driven programs. They manage system resources and provide an interface for user programs to interact with hardware.
2. **(ii) A multiprogrammed batch O/S gives the best possible utilization of system resources:** False. While multiprogramming improves resource utilization by allowing multiple programs to execute concurrently, it may not always result in the best possible utilization. Resource allocation policies and program characteristics also influence resource utilization.
3. **(iii) Size of an object file is larger than the size of the corresponding executable file:** True. Object files contain relocatable machine code and symbols needed for linking, which can make them larger than the corresponding executable file without these symbols and relocation information.

**(c) Conditions for portability:**

1. **Source code:**
   * Written in a high-level programming language (like C or Java) that is supported on multiple platforms.
   * Avoiding platform-specific dependencies and using standard libraries or APIs.
   * Adhering to portability guidelines and standards.
2. **Executable code:**
   * Compiled with a cross-compiler targeting multiple platforms.
   * Does not contain platform-specific optimizations or dependencies.
   * Uses standard libraries or dynamically links libraries at runtime.

**(d) Accessing I/O device:**

[Diagram illustrating different ways to access I/O device through program]

In a single-user system, a programmer can access an I/O device through direct I/O instructions, memory-mapped I/O, or system calls provided by the operating system. These methods enable communication between the program and the device driver, which manages the I/O device. The device driver interacts with the hardware to perform I/O operations on behalf of the program.

**Q.4 (a) File Permissions Analysis:**

1. **If user cs1 executes program p1 which tries to write in file f1, whether this will be allowed?**
   * This will be allowed because the **setuid** bit (**s**) is set on program **p1**, which means it will execute with the privileges of its owner (**root**). Since **root** has write permissions on **f1**, the write operation will be allowed.
2. **If program p2 tries to read file f1, whether this will be allowed?**
   * This will be allowed if **p2** has read permissions on **f1**. However, based on the provided information, it cannot be determined whether **p2** has read permissions on **f1**.
3. **Will user cs2 be able to view the listing of files of directory d1?**
   * No, user **cs2** will not be able to view the listing of files in directory **d1** because they do not have read or execute permissions on the directory.
4. **Will user cs1 be able to copy file f2 to directory d1?**
   * No, user **cs1** will not be able to copy file **f2** to directory **d1** if they do not have write permissions on **d1**.
5. **What will happen when user cs1 gives a command to delete file f1?**
   * If user **cs1** has write permissions on the directory containing **f1**, they will be able to delete **f1**. Otherwise, the operation will fail due to lack of permissions.

**(b) Finding Cluster Addresses:**

* For file **f1**: The first cluster is 4. Following the FAT entries: 4 -> 11 -> 6 -> 9 -> 3 -> 5
* For file **f2**: The first cluster is 2. Following the FAT entries: 2 -> 7 -> 1

**(c) Conditions for Linked Allocation:**

1. When files are expected to grow dynamically, and contiguous allocation is not suitable.
2. In systems where efficient space utilization is crucial, as linked allocation avoids external fragmentation.
3. When quick allocation and deallocation of disk space are required, as linked allocation only requires updating pointers.

**(d) Decision Making:**

1. **Size of logical block in a hard disk:** The size of the logical block is usually decided during the formatting of the disk or when creating the file system. It is often determined based on factors such as the disk capacity, performance requirements, and compatibility with existing systems.
2. **The 'format' of a track:** The format of a track is typically determined during disk formatting, where the disk's surface is divided into tracks, and each track is divided into sectors. The formatting process determines the number of tracks, sectors per track, and the size of each sector.
3. **The size of a sector:** The size of a sector is usually predefined by hardware standards and is not typically configurable by the user. It is determined by the disk controller and is often fixed at 512 bytes or 4 KB.

Top of Form

**Q.5 (a) Process State Diagram:**

Here is a simplified process state diagram showing various possible states of a process, types of schedulers, and different scheduling queues:

sqlCopy code

+------------+ | New Process| +------------+ | v +------------+ | Ready | +------------+ | | v | +------------+ +----------------+ | Running | <------------> | CPU Scheduler| +------------+ +----------------+ | | | | v v v | +------------+ +------------+ | | Blocked | | Waiting | | +------------+ +------------+ | | | | v v | +------------------------+ +----------------+ | I/O Scheduler | | Disk Queue | +------------------------+ +----------------+

In this diagram:

* Processes transition from the "New" state to the "Ready" state when they are ready to be executed.
* The "Running" state represents the process currently using the CPU.
* Processes may transition from "Running" to "Blocked" if they require I/O operations.
* The "Ready" state contains processes waiting to run, managed by the CPU scheduler.
* The "I/O Scheduler" manages I/O operations and transitions processes between "Blocked" and "Waiting" states.
* The "Disk Queue" manages disk operations, including requests from the "Waiting" state.

**(b) Design Decisions for Scheduling Algorithms:**

(i) **Multilevel Feedback Queue Algorithm:**

* Decision: Determine the number of priority levels and the criteria for priority promotion and demotion.
* How: Implement multiple queues with varying priorities, adjust priorities based on process behavior (e.g., aging), and define rules for promotion and demotion.

(ii) **Round Robin Algorithm:**

* Decision: Determine the time quantum for each process and the method for selecting the next process.
* How: Set a fixed time quantum, implement a circular queue to manage processes, and use a scheduler to select the next process based on the queue's order.

**(c) Analysis of Program p1:** (i) The **fork()** system call is called three times, resulting in the creation of 2^3 = 8 child processes. So, there will be 8 child processes of the shell. (ii) Each child process will print "hello" once. So, "hello" will be displayed 8 times. (iii) Each child process will have its process group. So, there will be 8 process groups in total. (iv) The **kill** command can be used to terminate all processes in the shell's process group. Assuming the process group ID of the shell is **PID**, the command would be: **kill -9 -PID**

Top of Form

**Q.6 (a)**

**I. First Fit Policy:**

1. For the process of size 456 KB:
   * It will fit into the 700 KB partition.
2. For the process of size 320 KB:
   * It will fit into the 400 KB partition.
3. For the process of size 587 KB:
   * It will not fit into any partition, as there is no single partition large enough to accommodate it.
4. For the process of size 498 KB:
   * It will fit into the 600 KB partition.

**Unallocated Partitions:**

* The 300 KB and 250 KB partitions will remain unallocated.

**II. Worst Fit Policy:**

1. For the process of size 456 KB:
   * It will fit into the 500 KB partition.
2. For the process of size 320 KB:
   * It will fit into the 400 KB partition.
3. For the process of size 587 KB:
   * It will not fit into any partition, as there is no single partition large enough to accommodate it.
4. For the process of size 498 KB:
   * It will fit into the 600 KB partition.

**Processes Not Getting Any Partition:**

* The process of size 587 KB will not get any partition.

**(b)**

**i) Page No. and Offset:**

* Given Logical Address: 1101100010100111
* Page Size = 1 KB = 2^10 bytes
* Logical Address is 16 bits long, so 10 bits for page number and 6 bits for offset.
* Page No.: 1101100010 (First 10 bits)
* Offset: 10100111 (Last 6 bits)

**ii) Max. Possible No. of Frames:**

* Frame Size = Page Size = 1 KB = 2^10 bytes
* Main Memory Capacity = Logical Address Space = 2^16 bytes
* Max. Possible No. of Frames = Main Memory Capacity / Frame Size = 2^16 / 2^10 = 2^6 = 64 frames

**(c)**

*[Diagram not available]*

**(d) Effective Memory Access Time:**

* Given:
  + Access Time of Associative Memory (Cache): 15 ns
  + Access Time of Main Memory: 60 ns
  + Time to Serve Page Fault: 100 ms = 100,000,000 ns
  + Probability of Finding Page Table Entry in Associative Memory: 90% = 0.9
  + Probability of Page Fault: 30% = 0.3
* Effective Memory Access Time = (Probability of Finding Page Table Entry \* Access Time of Associative Memory) + (Probability of Page Fault \* (Time to Serve Page Fault + Access Time of Main Memory)) = (0.9 \* 15 ns) + (0.3 \* (100,000,000 ns + 60 ns)) = (13.5 ns) + (30,000,000 ns + 60 ns) = 30,013,073.5 ns

**(e) Graphs:** *[Explanation not available]*

Top of Form

**Q.7 (a) Hardware Support for Implementing:**

(i) **Paging Technique of Memory Management:**

* Memory Management Unit (MMU): It translates virtual addresses to physical addresses.
* Page Table: To store the mapping of virtual pages to physical frames.

(ii) **SPOOLing Technique:**

* Disk Storage: To hold the spooled data temporarily.
* Printers or Other Output Devices: To receive data from the spool for printing or processing.
* Memory Buffer: To temporarily hold data being spooled.

(iii) **Round Robin Algorithm for CPU Scheduling:**

* Timer: To enforce time slices or time quanta for each process.
* Scheduler: To manage the queue of processes and decide their order of execution.
* Context Switching Mechanism: To switch between processes when their time slice expires.

**(b) Observations for Virtual Memory:**

* Programs often do not use their entire address space.
* Programs frequently access only a portion of their code and data.
* There's a large variation in the priority and frequency of data access.

**(c) Use of Un-named Pipes for Synchronization:**

* Yes, un-named pipes can be used for synchronizing two processes.
* For example, one process can write a special value into the pipe to signal the other process to proceed. The other process waits until it reads this special value from the pipe before continuing execution.

**(d) Identifying Reasons for Non-Completion:**

* Yes, it may be possible to identify the reasons for non-completion by examining the states of the processes and analyzing system logs or debugging information.
* Possible reasons could include resource contention, deadlocks, infinite loops, or system errors.

**(e) Rules for Deadlock Prevention:**

1. **Mutual Exclusion:** Ensure that at least one resource is non-sharable, allowing only one process to use it at a time.
2. **Hold and Wait:** Require processes to request all resources at once and release them before requesting new ones, preventing a process from holding one resource while waiting for another.
3. **No Preemption:** Do not allow resources to be forcibly taken away from a process; instead, require a process to release all its resources and then request them again if necessary.
4. **Circular Wait:** Enforce a policy where resources are allocated in a specific order, preventing circular dependencies among processes.

Top of Form