**Topic No. 5: Process Control and Scheduling**

1. Explain the algorithm for the exit() system call.

algorithm exit

input: return code for parent process

output: none

{

ignore all signals;

if (process group leader with associated control terminal)

{

send hangup signal to all members of process group;

reset process group for all members to 0;

close all open files (internal version of algorithm close);

release current directory (algorithm iput);

relea$e current (changed) root, if exists (algorithm iput);

free regions, memory associated with process (algorithm freereg);

write accounting record;

make process state zombie

assign parent process ID of all child processes to be init proc115s (1);

if any children were zombie, send death of child signal to init;

send death of child signal to parent process;

context switch;

}

### Algorithm for exit() System Call

The exit() system call in UNIX is responsible for terminating a process and releasing its associated resources. Here's the algorithm for the exit() system call:

1. \*\*Disable Signal Handling:\*\*

- The kernel disables signal handling for the exiting process as it no longer makes sense to handle signals.

2. \*\*Handle Process Group Leader:\*\*

- If the exiting process is a process group leader associated with a control terminal:

- The kernel sends a "hangup" signal to all processes in the process group.

- It resets the process group number to 0 for processes in the group to prevent future confusion.

3. \*\*Close Open File Descriptors:\*\*

- The kernel iterates through all open file descriptors of the process and closes each one internally using the close algorithm.

4. \*\*Release Inodes:\*\*

- It releases the inodes accessed for the current directory and the changed root directory (if it exists) using the iput algorithm.

5. \*\*Release Memory:\*\*

- The kernel frees all user memory regions associated with the process using the detachreg algorithm.

6. \*\*Set Process State to Zombie:\*\*

- The kernel changes the process state to "zombie" and saves the exit status code, as well as the accumulated user and kernel execution time, in the process table.

7. \*\*Write Accounting Record:\*\*

- An accounting record containing various runtime statistics is written to a global accounting file.

8. \*\*Adopt Child Processes:\*\*

- Process 1 (init) adopts all child processes of the exiting process to ensure their proper handling.

- If any child processes were zombies, a "death of child" signal is sent to init.

9. \*\*Signal Parent Process:\*\*

- The exiting process sends a "death of child" signal to its parent process.

- The parent process typically synchronizes with the exiting child using a wait system call.

10. \*\*Context Switch:\*\*

- The kernel performs a context switch to schedule another process for execution, as zombie processes are never scheduled to execute.

In summary, the exit() system call efficiently terminates a process, releases its resources, and ensures proper handling of child processes and communication with the parent process.

2. Explain the different functions of the clock interrupt handler.  
algorithm clock

input:

none

output: none

{

restart clock;

/\* so that it will interrupt again \*/

if (callout table not empty)

{

adjust callout times;

schedule callout function if time elapsed;

if (kernel profiling on)

note program counter at time of interrupt;

if (user profiling on)

' 'note program counter at time of interrupt;

gather system statistics;

gather statistics per process;

adjust measure of propess CPU utilitization;

for (all processes in the system)

{

if (1 second or more since last here and interrupt not in critical

region of code)

adjust alarm time if active;

adjust measure of CPU utilization;

if (process to execute in user mode)

adjust process priority;

}

wakeup swapper process is necessary;

}

**Functions of the Clock Interrupt Handler**

The clock interrupt handler in UNIX performs various essential functions to manage system operations efficiently. Here's an explanation of its different functions:

1. **Restarting the Clock:**
   * Ensures that the system clock continues to function properly by restarting it for subsequent interrupts.
2. **Scheduling Internal Kernel Functions:**
   * Utilizes internal timers to schedule invocation of kernel functions, ensuring timely execution of essential tasks.
3. **Execution Profiling:**
   * Provides execution profiling capability for both the kernel and user processes, allowing for performance analysis and optimization.
4. **Gathering Accounting Statistics:**
   * Gathers system and process accounting statistics, which are crucial for monitoring system performance and resource utilization.
5. **Timekeeping:**
   * Keeps track of time, maintaining accurate timestamps for various system operations and events.
6. **Sending Alarm Signals:**
   * Sends alarm signals to processes upon request, enabling timely notifications for specific events or conditions.
7. **Waking up the Swapper Process:**
   * Periodically wakes up the swapper process, which is responsible for managing process scheduling and memory allocation.
8. **Controlling Process Scheduling:**
   * Adjusts process priorities and scheduling parameters based on system conditions and resource utilization.

3. Explain the system calls for time.  
  
 **System Calls for Time Management**

Time management is crucial in UNIX systems, facilitated by several system calls. These include stime, time, times, and alarm, each serving different purposes in managing system and process time. Let's delve into each of these system calls:

1. **stime:**
   * Allows the superuser to set a global kernel variable representing the current system time.
   * Syntax: **stime(pvalue);**
   * The kernel increments this variable once per second.
2. **time:**
   * Retrieves the system time previously set by stime.
   * Syntax: **time(tloc);**
   * Returns the time value to the user process, often used by commands like **date** to display the current time.
3. **times:**
   * Retrieves cumulative execution times for the calling process and its zombie children.
   * Syntax: **times(tbuffer);**
   * Uses a **struct tms** structure to store user and kernel times for both the process and its children.
   * Returns elapsed time from a specific reference point, usually system boot time.
4. **alarm:**
   * Allows user processes to schedule alarm signals.
   * Example usage involves checking file access time periodically.
   * Syntax: **alarm(seconds);**
   * Sets an alarm to trigger after a specified number of seconds, causing the process to receive an alarm signal.
   * Often used in conjunction with **signal** to handle the alarm signal and perform specific actions.

These system calls rely heavily on the system clock, with the kernel managing various time counters during clock interrupts. This ensures accurate timekeeping and facilitates timely actions such as scheduling alarms and gathering process execution statistics.

4.What is the use of the fork() system call? Explain the sequence of operations the kernel executes for fork.

**Explanation of fork() System Call**

The **fork()** system call in UNIX is essential for creating new processes. Let's break down its usage and the sequence of operations the kernel executes:

1. **Usage:**
   * The **fork()** system call is invoked by a parent process to create a new child process.
   * Syntax: **pid\_t fork();**
   * Upon successful execution, it returns the child process ID (PID) to the parent process and 0 to the child process.
2. **Sequence of Operations:**
   * **Check for Available Resources:**
     + The kernel first ensures it has the necessary resources to complete the fork successfully. This includes memory space for the child process.
   * **Allocate Process Table Slot and Assign PID:**
     + The kernel allocates a slot in the process table for the new child process.
     + It assigns a unique ID number (PID) to the child process.
   * **Check Process Limit:**
     + It verifies that the user invoking **fork()** is not already running too many processes, adhering to configurable process limits.
   * **Initialize Child Process State:**
     + Marks the child process state as "being created."
   * **Copy Parent Process Context:**
     + The kernel makes a logical copy of the parent process context, including user-level context such as text, data, and stack segments.
     + Certain shared regions, like the text region, may have their reference counts incremented instead of being physically copied.
   * **Increment File and Inode Counters:**
     + Increments file and inode table counters for files associated with the child process.
   * **Initialize Child Process Attributes:**
     + Copies various fields from the parent process slot to the child process slot, including user and group IDs, process group, and scheduling parameters.
     + Assigns the parent process ID to the child process.
   * **Adjust Reference Counts for Associated Resources:**
     + Increments reference counts for resources associated with the parent process, such as the current directory and changed root (if applicable).
     + Increments file table reference counts for open files, ensuring proper resource sharing between parent and child.
   * **Create User-Level Context for Child Process:**
     + Allocates memory for the child process user area, regions, and auxiliary page tables.
     + Duplicates every region from the parent process to the child process and attaches them accordingly.
   * **Create Dynamic Portion of Child Context:**
     + Copies the parent context layer 1, containing user saved register context and kernel stack frame, to the child process.
     + Creates a dummy context layer for the child process, containing data allowing the child to recognize itself and start running when scheduled.
3. **Return Values:**
   * If the executing process is the parent process:
     + Changes the child process state to "ready to run" and returns the child PID to the parent process.
   * If the executing process is the child process:
     + Initializes timing fields and returns 0 to the child process.

This sequence of operations ensures the successful creation of a child process with an identical initial state to its parent, enabling parallel execution and multitasking in UNIX systems.

5. What is the use of a signal? Explain the types of signals.

6. Explain System boot and the init process.

The process of system booting involves initializing the system from an inactive state, commonly referred to as "bootstrapping." This sequence varies based on machine type but generally involves getting a copy of the operating system into machine memory and starting its execution. On UNIX systems, this process typically occurs in multiple stages.

1. **Bootstrap Sequence:**
   * An administrator initiates the boot procedure by setting switches on the computer console or pushing a button to load a bootstrap program. This program, often consisting of a few instructions, instructs the machine to execute another program.
   * Eventually, the bootstrap procedure reads the boot block (block 0) from a disk and loads it into memory. The boot block program then loads the kernel from the file system (e.g., "/unix") into memory.
2. **Kernel Initialization:**
   * Once loaded, the kernel initializes its internal data structures, such as linked lists of free buffers and inodes, hash queues for buffers and inodes, region structures, and page table entries.
   * After initialization, the kernel mounts the root file system ("/") and sets up the environment for process 0 (init process).
3. **Process 0 (Init Process):**
   * Process 0 forks, creating a new process called process 1 (init process). This fork operation is invoked directly from the kernel because process 0 is executing in kernel mode.
   * Process 1, running in kernel mode, creates its user-level context by allocating a data region and attaching it to its address space. It copies code from the kernel address space to this new region to form its user-level context.
   * Process 1 sets up the saved user register context, transitions from kernel to user mode, and executes the copied code. It is now a user-level process.
   * The copied code includes a call to the **exec** system call to execute the program "/etc/init."
4. **Init Process:**
   * Process 1, commonly known as **init**, executes the program "/etc/init" as part of the user-level context setup.
   * Init process is responsible for further initialization of new processes and managing system startup procedures.

The reason for copying the code for the **exec** system call to the user address space of process 1 is to facilitate the execution of the init program. Instead of directly invoking an internal version of **exec**, the kernel allows process 1 to execute the **exec** system call as a regular user-level process. This approach follows the UNIX philosophy of using system calls for various operations, ensuring consistency and compatibility across different processes.

7. Draw and explain user level and kernel level priority.

8. Explain a simple process scheduling algorithm with an example.

The provided algorithm, **schedule\_process**, outlines a simple process scheduling approach used in UNIX systems, particularly employing a round-robin with multilevel feedback strategy. Here's an explanation of how this algorithm works:

1. **Initialization**:
   * The algorithm initializes without any specific input.
2. **Process Selection**:
   * The algorithm iterates through all processes in the run queue.
   * For each process, it selects the highest priority process that is loaded in memory. Priority is determined based on recent CPU usage and other factors, with lower numerical values indicating higher priority.
   * If no eligible process is found to execute (i.e., all processes are waiting for I/O or other events), the algorithm idles the machine until the next interrupt occurs.
3. **Idle Handling**:
   * When no process is eligible to execute, the algorithm enters an idle state, waiting for the next interrupt.
   * Upon receiving an interrupt, the machine is taken out of the idle state, and the scheduling process resumes.
4. **Context Switch**:
   * Once a process is chosen for execution, it is removed from the run queue.
   * The algorithm then performs a context switch, transitioning the execution context from the previously running process to the chosen process.
   * The chosen process's execution is resumed from the point where it was suspended.
5. **Execution Continuation**:
   * After the context switch, the chosen process continues its execution until it completes its time quantum or is preempted by another process.

This algorithm ensures that processes are fairly scheduled for execution based on their priority levels. The round-robin approach ensures that each process gets a chance to execute, while the multilevel feedback mechanism adjusts process priorities dynamically based on CPU usage, preventing any single process from monopolizing the CPU for an extended period.

Overall, this scheduling algorithm aims to maintain system responsiveness and fairness by efficiently utilizing available CPU resources and providing a balanced execution environment for all processes.

9. Explain how the kernel prevents a process from monopolizing the use of the CPU in Unix System V

In UNIX System V, the kernel employs several mechanisms to prevent a single process from monopolizing the CPU and ensure fair resource allocation among multiple processes. One key strategy involves adjusting process priorities and CPU usage dynamically. Here's how the kernel accomplishes this:

1. **Process Priorities**:
   * Each process in the system is assigned a priority value, which determines its scheduling preference. Lower numerical priority values indicate higher priority.
   * The kernel periodically recalculates the priority of each process based on its recent CPU usage and other factors. This ensures that processes with excessive CPU usage are given lower priorities, allowing other processes to have a chance to execute.
2. **Round-Robin Scheduling**:
   * The kernel employs a round-robin scheduling algorithm, ensuring that all processes in the system get a fair share of CPU time.
   * If multiple processes are ready to execute and have the same priority, the kernel uses a round-robin policy to schedule them. This policy ensures that each process gets an equal opportunity to execute, preventing any single process from monopolizing the CPU.
3. **Multilevel Feedback**:
   * The kernel utilizes a multilevel feedback mechanism to adjust process priorities based on their recent behavior.
   * Processes that have been using the CPU excessively are assigned lower priorities over time, while processes that have been waiting patiently or have had less CPU usage may receive higher priorities.
   * This feedback mechanism ensures that processes adapt to changing system conditions and prevents long-running processes from starving other processes of CPU time.
4. **Idle Handling**:
   * If no process is eligible to execute (e.g., all processes are waiting for I/O), the kernel enters an idle state, waiting for the next interrupt or event.
   * This idle handling mechanism prevents the CPU from being continuously occupied by a single process when there are no other runnable processes.
5. **Interrupt-driven Behavior**:
   * The kernel operates in an interrupt-driven manner, responding promptly to external events and interrupts.
   * When an interrupt occurs, the kernel may preempt the currently executing process and schedule a different process to run, ensuring that CPU time is fairly distributed among competing processes.

Overall, these mechanisms work together to prevent any single process from monopolizing the CPU in UNIX System V. By dynamically adjusting priorities, employing round-robin scheduling, utilizing multilevel feedback, handling idle states effectively, and responding to interrupts, the kernel maintains a balanced and fair execution environment for all processes in the system.

10. Explain profiling in detail.

Profiling in UNIX System V involves analyzing the performance of the kernel or user-level processes by monitoring their execution behavior. Here's how profiling works in both the kernel and user levels:

1. **Kernel Profiling**:
   * Kernel profiling measures the time spent executing in user mode versus kernel mode and tracks the execution time of individual kernel routines.
   * The kernel profile driver samples system activity at clock interrupts and records the mode of execution (user or kernel) at each interrupt.
   * The profile driver maintains a list of kernel addresses to sample, usually corresponding to kernel functions, which are downloaded by a process.
   * At each clock interrupt, the profile driver updates counters based on the execution mode and program counter values.
   * User processes can read these counters to obtain statistical measurements of kernel execution time and identify performance bottlenecks.
2. **User-Level Profiling**:
   * Users can profile the execution of their processes at the user level using the **profil** system call.
   * The **profil** system call takes parameters such as the address of an array in user space, the size of the array, the virtual address of a user subroutine, and a scaling factor.
   * When a process is interrupted by the clock while in user mode, the clock handler examines the program counter and increments the corresponding location in the array based on the scaling factor.
   * By analyzing the contents of the array, users can determine the frequency of execution of different parts of their code.
3. **Example**:
   * For instance, consider a program that continuously calls two functions **f** and **g** in an infinite loop.
   * The process sets up profiling using the **profil** system call, specifying the range of text addresses it wishes to profile.
   * After running the program for a certain duration, users can analyze the profiling data to understand the execution behavior of their code.
   * The profiling output provides information about the frequency of execution of different functions and code segments, helping users identify performance issues and optimize their code.

Overall, profiling allows users to gain insights into the behavior and performance of their code at both the kernel and user levels. By analyzing profiling data, users can optimize their programs, improve resource utilization, and enhance overall system performance.

Top of Form

**Topic No. 6: Memory management and I/O Subsystem**

1. What is demand paging? Explain the data structures used for it.

2. Explain the working of the page stealer process.

The page stealer, a kernel process in UNIX systems, is integral for managing memory pages to optimize system performance. Here's how it operates:

During system initialization, the kernel creates the page stealer. It activates whenever the system faces low free memory conditions. Its primary task is to swap out memory pages that are no longer actively used by processes. The page stealer examines active, unlocked memory regions, skipping locked ones to avoid interfering with ongoing processes. It increments the age field of all valid pages, marking their usage over time.

Pages in memory are classified into two states: aging and eligible for swapping. Initially, pages are in the aging state, indicating recent usage by processes. The page stealer tracks page references and marks pages as eligible for swapping after a certain number of examinations without references. Processes accessing pages influence their aging process. If a page is referenced by any process, its age is reset. Pages not part of any process's working set become eligible for swapping.

The kernel sets high and low-water marks to regulate page stealing and prevent thrashing. When free memory falls below the low-water mark, the page stealer activates to swap out pages until free memory exceeds the high-water mark. The page stealer decides whether to swap out pages based on their current state and modifications. Pages may be swapped out if they have no copy on the swap device, if they've been modified in memory, or if they're scheduled for swapping due to low memory conditions.

Swap operations involve scheduling pages for swapping, writing them to the swap device, and updating page table entries accordingly. Pages are swapped in blocks to optimize disk operations. The kernel manages swap space fragmentation by swapping out blocks of pages but swapping in individual pages. It allocates swap space dynamically and maintains swap-use tables to track page locations.

The page stealer prioritizes swapping based on process activity and available space on the swap device. It efficiently utilizes swap space to minimize disk I/O overhead.

As processes continue execution and free memory fluctuates, the page stealer adapts its swapping strategy to maintain system performance. In essence, the page stealer process plays a vital role in managing memory resources in UNIX systems, ensuring efficient utilization and preventing memory-related performance degradation.

3. Explain page fault. Explain the handling of validity page fault.

4. Explain in detail the allocation of space on the swap device.

5. Explain the swapping of processes between swap space and main memory.

6. Write a short note on: Streams.