2602071846 – Anselyus Patrick Siswanto

Operating System Assignment 1

1. Explain in detail what happens when a process gets interrupted. You should explain the relationship between interrupts and the process counter.
   Answer:
   When a process is interrupted, the CPU will momentarily stop its execution to handle a higher-priority task, often triggered by hardware or software events. The operating system saves the current process's state, including the program counter (which stores the address of the next program to be executed), so the process can resume later. This saved state is crucial to restore the process after the interruption. The CPU will then execute an ISR to handle the interruption.
   After the ISR completes, the CPU restores the saved process state, including the program counter, allowing the interrupted process to continue from where it left off. The program counter plays a vital role in ensuring that processes resume correctly after an interrupt.

2. What is a process control block?
   Answer:
   A process control block (PCB) is a data structure maintained by the operating system for every active process. PCB contains information about a process's state, program counter, register values, memory usage, and I/O device allocation. The operating system creates a PCB when a process is created, initialized, or installed. This structure of data collection allows the operating system to manage processes more efficiently, making it capable of doing important functions such as multitasking and context switching. The PCB ensures that processes can be paused, resumed, or terminated without any data loss by keeping all necessary process-specific information centralized.

3. In threading, in which case and why should we use pthread_join() ?
   Answer:
   In threaded programming, pthread_join() is used for pausing the main thread until another thread completes/terminates. Utilizing this function can prevent threads from persisting as inactive while still consuming system resources despite already finishing their tasks. It ensures sequential completion of thread tasks, helps in preventing resource wasting, and allows for precise control over the flow of the program by synchronizing the end of child threads with the main thread.

4. Considering the following process with arrival time and CPU burst time as follow:

Process	CPU Burst Time	Arrival Time
P1	8	0
P2	3	2
P3	6	4
P4	4	5

Calculate the Average waiting time and average turnaround time using the following scheduling algorithm:

a. Shortest Job First

Process	Completion Time	Turnaround Time	Waiting Time
P1	8	8	0
P2	11	9	6
P3	21	17	11
P4	15	10	6

Average Turnaround Time = 11
Average Waiting Time = 5.75

b. Shortest Job Remaining Time Next

Process	Completion Time	Turnaround Time	Waiting Time
P1	15	15	7
P2	5	3	0
P3	21	17	11
P4	9	4	0

Average Turnaround Time = 4.5
Average Waiting Time = 9.75

c. Round Robin with time slice=2

Process	Completion Time	Turnaround Time	Waiting Time
P1	21	21	13
P2	11	9	6
P3	19	15	9
P4	15	10	6

Average Turnaround Time = 13.75
Average Waiting Time = 8.5

d. Highest Response Ratio Next

Process	Completion Time	Turnaround Time	Waiting Time
P1	8	8	0
P2	11	9	6
P3	21	17	11
P4	15	10	6

Average Turnaround Time = 11
Average Waiting Time = 5.75

5. What are the issues involving process scheduling in a multiprocessor environment?
   Answer:
   In a multiprocessor environment, process scheduling becomes more complex due to several challenges and issues. Such as:
   - Load Balancing
     Issue: If one processor is heavily loaded while others are idle or underutilized. Overall system performance can degrade. This is an issue as load balancing wants to ensure that no single processor becomes a bottleneck.
   - Processor Affinity
     Issue: Moving processes between processors can cause cache misses, as the process cache in the original processor. This can negatively affect performance. Maintaining processor affinity helps to reduce the overhead of cache reloading, but it may conflict with the need for load balancing.
   - Synchronization Overhead
     Issue: Improper synchronization can lead to issues like race conditions or deadlocks. Additionally, synchronization mechanisms can introduce performance penalties if not managed effectively.
   - Context Switching Overhead
     Issue: In a multiprocessor environment, frequent context switching between processors can lead to high overhead, especially if synchronization and inter-processor communication are involved. This reduces the overall efficiency of the system.

6. What is the difference between an embedded system and a deeply embedded system?
   Answer:
   a. Embedded System:
   An embedded system is a computer system that is part of a larger device or system, designed to perform specific tasks. It typically has a dedicated function and operates in real-time environments.
   Characteristics:
   - Can be visible to the user or may interact with external systems.
   - Programmable, with the possibility of updating or reconfiguring its software.
   - Often has external interfaces.

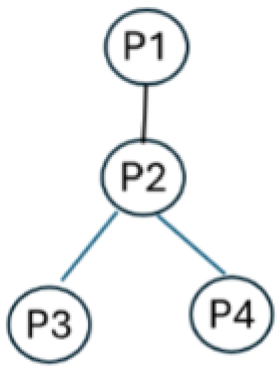
   Examples: Smart TVs, washing machines, and automotive control systems
   b. **Deeply Embedded System:**
   A deeply embedded system is a subset of embedded systems, but it is typically more tightly integrated, with minimal or no external interfaces. These systems are usually designed to perform very specific, low-level tasks within the device.
   Characteristics:
   - Invisible to the end-user, often running without direct interaction or display.
   - Usually non-programmable or very limited in terms of external reconfiguration.
   - Often has no external interface, running autonomously within the hardware.

   Examples: Microcontrollers in appliances sensors in industrial equipment, or low-level firmware inside hardware components.
   Key Differences:
   - Visibility: Embedded systems may be visible and user-interactive, whereas deeply embedded systems are typically invisible and fully integrated.

- Programmability: Embedded systems are often reprogrammable, while deeply embedded systems are usually fixed-function and non-programmable.
- Complexity: Embedded systems can perform more complex tasks with external interfaces, while deeply embedded systems are focused on specific, low-level tasks with minimal external interfaces.

Programming

Creating processes using the fork() system call in a certain way, will result in a hierarchy of processes. Consider the following hierarchy:



Write a C program to create such a hierarchy. Each child simply display its process id.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void* child_function(void* arg) {
    printf("Child   process   simulation   with   thread:   Process   ID   =   %ld\n",
(long)pthread_self());
    return NULL;
}

int main() {
    pthread_t thread1, thread2, thread3;

    printf("Parent process (P1): Process ID = %ld\n", (long)pthread_self());

    // Create child P2
    pthread_create(&thread1, NULL, child_function, NULL);
    pthread_join(thread1, NULL);

    // Inside "P2", create P3 and P4
    pthread_create(&thread2, NULL, child_function, NULL);
    pthread_create(&thread3, NULL, child_function, NULL);

    // Wait for both P3 and P4 to finish
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);

    return 0;
}
```