OS All Assignment Report

Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Class Test - 29-06-24

1.1 Problem - 1(Threading)

Below given the code :

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <pthread.h>

int maximum,minimum;
float average;
int n;

void* averaget(void* a){
    int *x = (int*)a;
    int sum=0;
    for(int i = 0; i<n; i++){
        sum +=x[i];
    }
    printf("Sum: %d\n",sum);
    average = (float)sum/n;
}

void* maximumt(void* a){
    int *x = (int*)a;
    maximum = x[0];
    for(int i=0; i<n; i++){
        if(maximum<x[i])
            maximum = x[i];
    }
```

```c
}

void* minimumt(void*a){
    int *x = (int*)a;
    minimum = x[0];
    for(int i=1; i<n; i++)
        if(minimum>x[i])
        minimum = x[i];

}

int main(){
    printf("Enter the number of data; \n");
    scanf("%d",&n);
    int x[n];
    printf("Enter the data: ");
    for(int i=0; i<n; i++)
        scanf("%d",&x[i]);

    pthread_t t1,t2,t3;
    pthread_create(&t1,NULL, averaget,&x);
    pthread_create(&t2,NULL, maximumt,&x);
    pthread_create(&t3,NULL,minimumt,&x);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    pthread_join(t3,NULL);

    printf("Average: %f\n",average);
    printf("Minimum : %d\n",minimum);
    printf("Maximum : %d\n",maximum);
}
```

The output are showing below:

```
┌──(miju_chowdhury㉿miju)-[/mnt/…/Operating_System/Code/Practice/Labt]
└─$ ./a.out
Enter the number of data;
7
Enter the data: 90 81 78 95 79 72 85
Average: 82.857140
Minimum : 72
Maximum : 95
```

Figure 1:

OS All Assignment Report

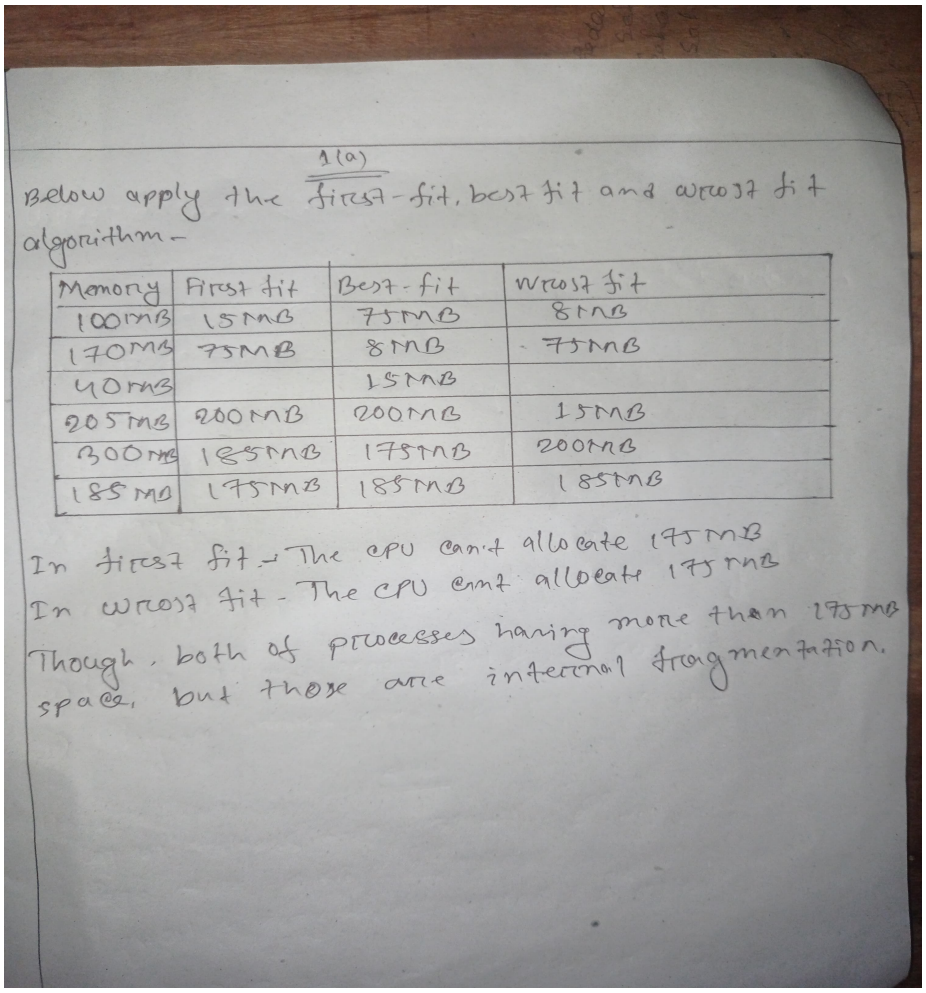
Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Class Test - 26-06-24

1.1 Problem - 1(Algorithm)

A. Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB , 15 MB , 185 MB , 75 MB , 175 MB , and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.
Answer:


1(a)

Below apply the first-fit, best fit and worst fit algorithm -

Memory	First fit	Best-fit	Worst fit
100MB	15MB	75MB	85MB
170MB	75MB	85MB	75MB
40MB		15MB	
205MB	200MB	200MB	15MB
300MB	185MB	175MB	200MB
185 MB	175MB	185MB	185MB

In first fit - The CPU can't allocate 175MB
In worst fit - The CPU can't allocate 175MB
Though both of processes having more than 175MB space, but those are internal fragmentation.

Figure 1:

C. Consider a system where free space is kept in a free-space list. Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.

Answer:
If the pointer to the free-space list is lost, the system cannot directly reconstruct the list. However, it can scan the entire disk to identify free blocks (e.g., by checking file system metadata or block allocation information) and rebuild the list manually. This is time-consuming but possible.

1.2 Problem - 3

QA: How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

Answer:
Caches improve performance by storing frequently accessed data closer to the CPU, reducing the time needed to fetch data from slower main memory. However, systems don't use larger caches because they are expensive (costly in terms of both space and power), and diminishing returns occur as cache size increases—larger caches have lower hit rates and increased latency for accessing the cache itself.

QB: Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

Answer:

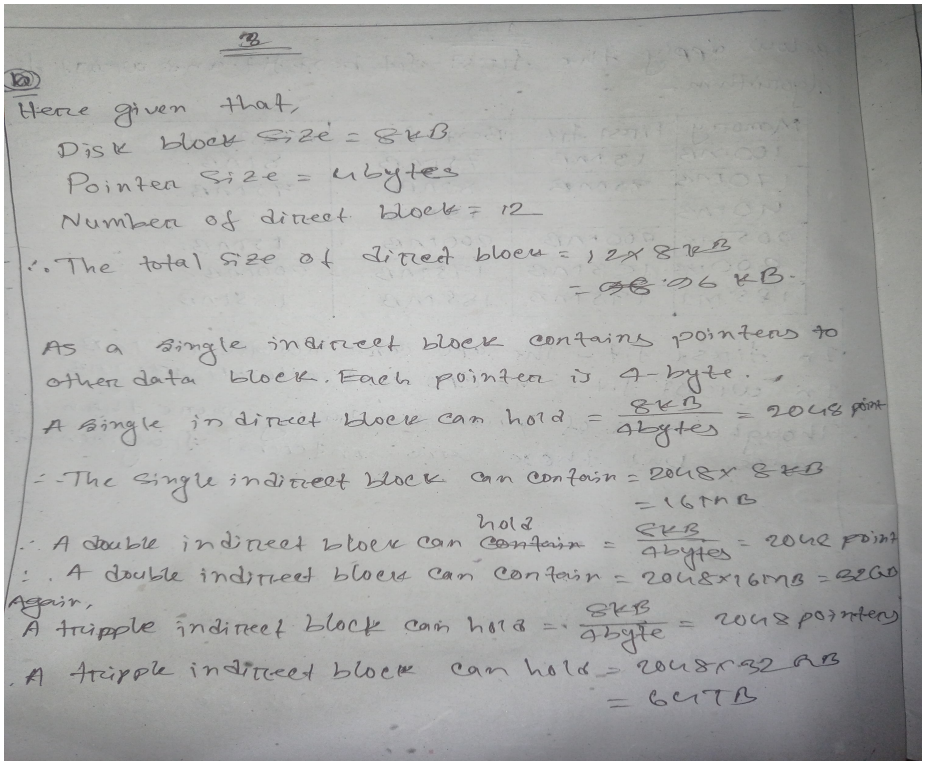


Figure 2:

1.3 Problem - 5

A. Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from "bad" to "perfect" according to their page-fault rate. Separate those algorithms that suffer from Belady's anomaly from those that do not.
a. LRU replacement
b. FIFO replacement
c. Optimal replacement
d. Second-chance replacement

Answer:
Below ranking those algorithms on five-point scale:
1. Optimal Replacement: Perfect (minimizes page faults, doesn't suffer from Belady's anomaly).
2. LRU (Least Recently Used): Good (approximates optimal, no Belady's anomaly).
3. Second-Chance: Fair (a better version of FIFO, no Belady's anomaly).
4. FIFO (First-In, First-Out): Bad (can increase page faults when more frames are added, suffers from Belady's anomaly).

B. Consider the following page reference string:
7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1.
Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

Answer:

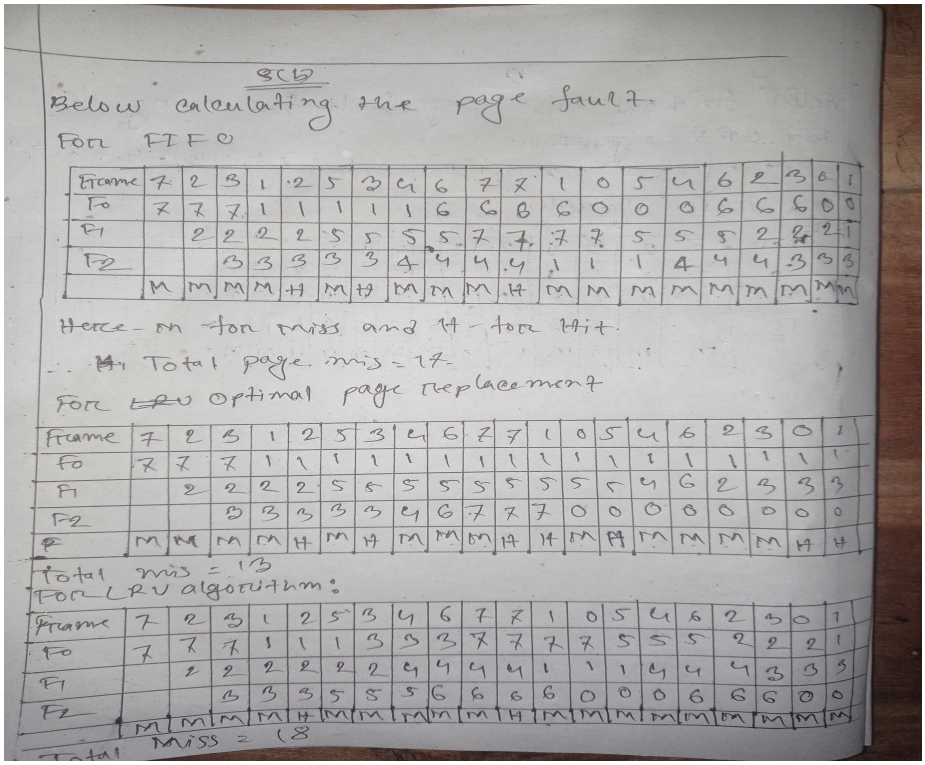


Figure 3:

C. Explain why minor page faults take less time to resolve than major page faults.

Answer:
Minor page faults take less time to resolve than major page faults because:
1. Minor page faults occur when the required page is already loaded in memory but is not in the correct location in the page table (e.g., it's swapped out or in a different process's address space). The system only needs to update the page table and may not require disk access. The time is mostly spent in accessing

the page table and possibly moving the page to the correct spot.
2. Major page faults occur when the required page is not in memory at all and must be loaded from disk. This involves disk I/O, which is significantly slower than memory access. The system must read the page from disk into physical memory, update the page table, and may need to swap out other pages to make room, making the process more time-consuming.
Thus, minor page faults involve only memory operations, while major page faults involve disk access, which is much slower.

1.4 Problem - 6(LRU,FIFO,OPT)

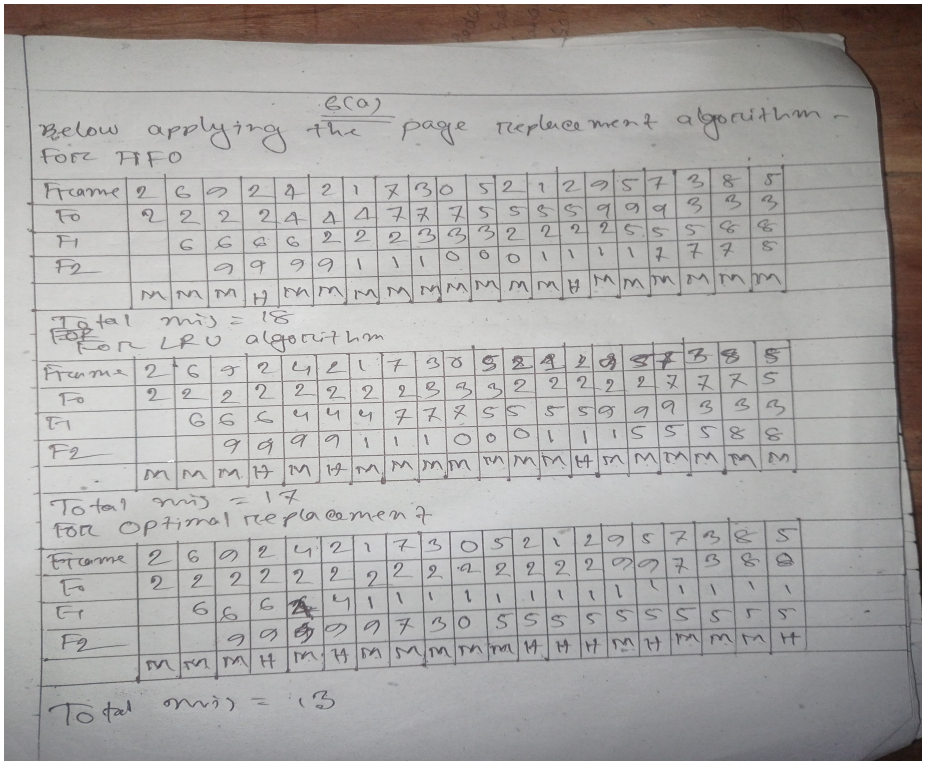


Figure 4:

OS All Assignment Report

Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Class Test - 25-06-24

1.1 Problem - 1

A. Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base –limit register pairs are provided: one for instructions and one for data. The instruction base –limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

Answer:
This scheme of using separate base-limit register pairs for code (instructions) and data is beneficial in a number of ways, particularly for protection, sharing, and efficiency. However, it also introduces potential complexities. Let's break down the advantages and disadvantages of such a scheme.
Advantages:
1. Memory Protection:
Since the instruction segment is read-only, this prevents programs from modifying their own code, which enhances security and stability. Bugs that attempt to alter the instruction section would be caught, reducing the risk of accidental or malicious tampering. Separating code and data also minimizes accidental overwrites, ensuring that data manipulations do not accidentally interfere with the program's instructions. 2. Efficient Sharing of Code:
Shared Code: With a read-only instruction segment, the same program code can be shared across multiple processes without duplication in memory. Only one instance of the code is needed, and each process can have its own data segment. This saves memory and allows many users to run the same program simultaneously. This also improves performance, as fewer resources are required for loading and maintaining duplicate copies of the program code. 3. Clear Differentiation of Responsibilities:
By separating code and data, the CPU can fetch instructions and data with

clearly defined boundaries. This can help improve cache utilization and make instruction pipelines more predictable, as code and data fetches are handled independently.

Disadvantade:
1. Inflexibility in Self-Modifying Code:
Programs that require self-modifying code (code that generates or alters itself) cannot be supported in this model, as the instruction segment is read-only. Self-modifying code can be used in optimizations or advanced functionalities (like just-in-time (JIT) compilation), which would not be feasible in this scheme. Additionally, certain languages and runtimes that rely on code modification would require special handling to execute properly in such an environment.
2. Complexity in Memory Management:
Managing two separate base-limit register pairs adds complexity to the system's memory management, as the operating system must track and update multiple base-limit registers for each process. This overhead can make context switching slower and add to the CPU's workload. Additional complexity arises when managing different permissions (read-only for instructions, read-write for data), which can be challenging in certain architectures.
3. Increased Hardware Requirements:
To implement separate base-limit register pairs, additional hardware support is necessary, increasing system costs and potentially making context switching more resource-intensive. This may not be justified for simple systems or systems with limited resources.
4. Fragmentation:
This separation can lead to internal fragmentation if the code and data segments are of different sizes, as they are not combined into a single segment. Memory may be wasted when allocating space for each segment individually. If a program's code or data needs grow unexpectedly, resizing or relocating segments may be required, which adds to overhead.

B. Why are page sizes always powers of 2?
Page sizes are almost always powers of 2 because it simplifies both the address translation process and memory management. It represents the binary representation.

C.

1.2 Problem - 2(Page and offset number)

Assuming a 1- KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
a. 3085
b. 42095

c. 215201
d. 650000
e. 2000001
Answer:

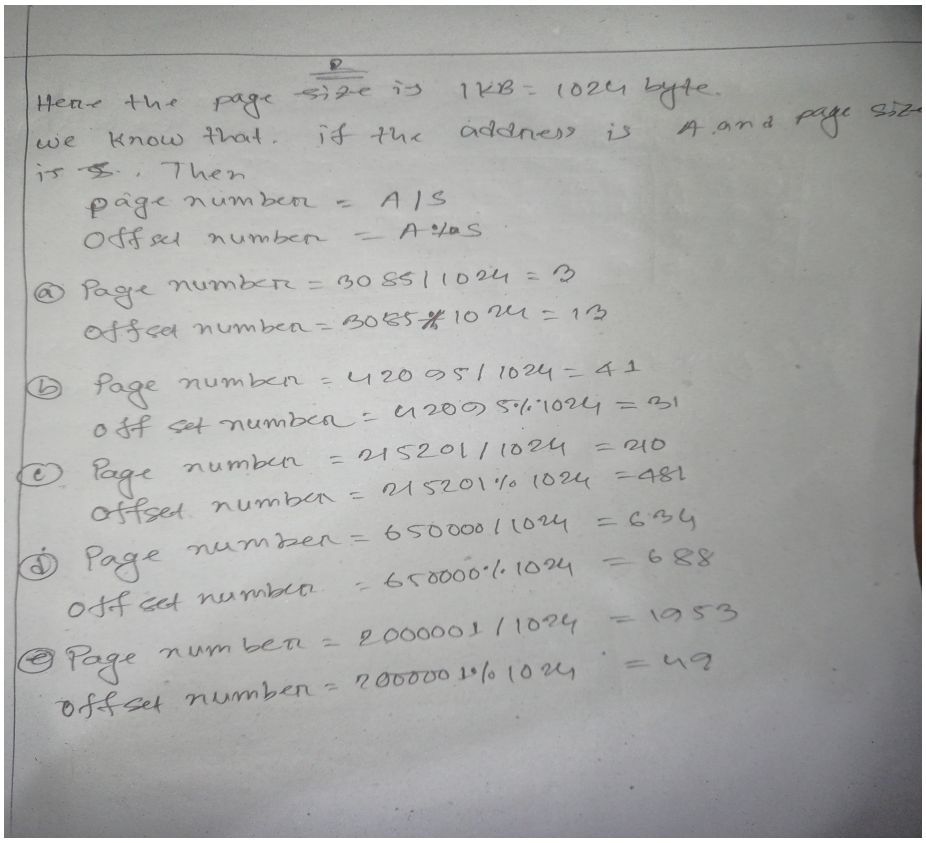


Figure 1:

1.3 Problem - 3(Paging)

A. The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2- KB page size. How many entries are there in each of the following?
a. A conventional, single-level page table
b. An inverted page table
What is the maximum amount of physical memory in the BTV operating system?

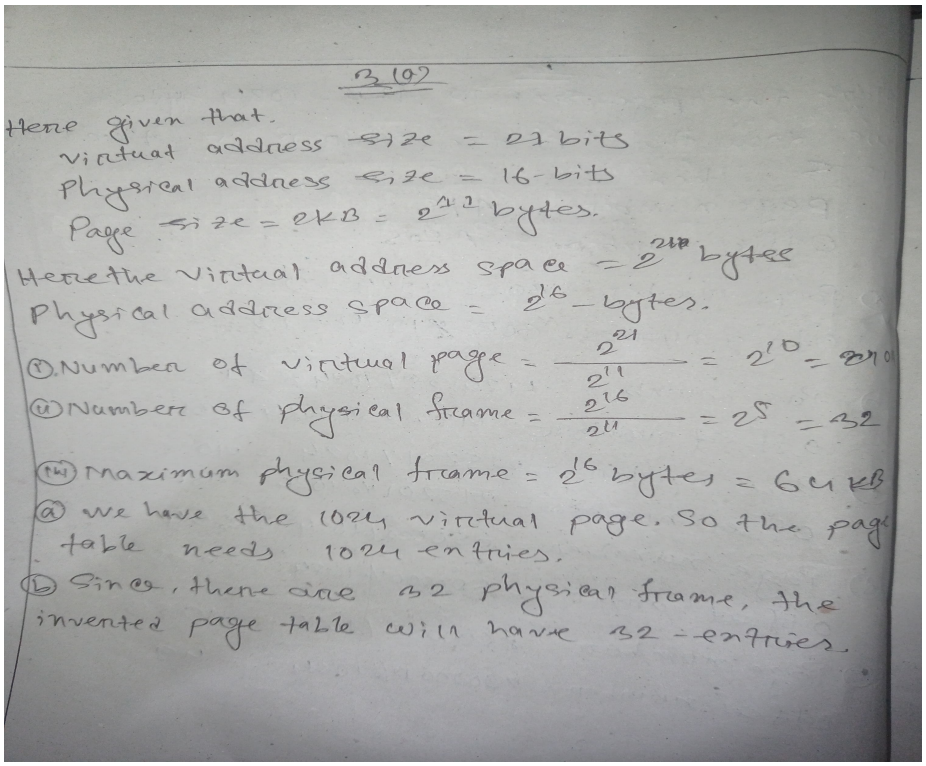


Figure 2:

B. Can we avoid internal fragmentation and external fragmentation? How?
To address internal and external fragmentation effectively, different strategies are used depending on the specific memory allocation needs and constraints of the operating system or application. Here's how both types of fragmentation can be managed or minimized:
1. Avoiding Internal Fragmentation:
Internal fragmentation occurs when fixed-size memory blocks are allocated, and the allocated memory block is larger than what the process requires, leading to wasted space within each block. This problem can be reduced or avoided by:
Using Variable-Size Allocation: Instead of using fixed-size memory blocks, variable-size memory allocation (like dynamic memory allocation) allows blocks to be tailored to the exact size required by each process, reducing wasted space.
2. Avoiding External Fragmentation:
External fragmentation happens when free memory is scattered in small blocks between allocated segments, making it difficult to allocate a large contiguous block when needed. Techniques to mitigate or avoid external fragmentation include:
a. Paging:

Paging eliminates the need for contiguous memory by dividing both physical and virtual memory into fixed-size blocks called "pages." Processes are allocated pages as needed, so even if memory is fragmented, pages can be mapped in any order. This avoids external fragmentation because memory doesn't need to be contiguous.

b. Segmentation with Paging:
This hybrid approach divides memory logically (by segment) but maps segments onto pages, combining the benefits of both systems and preventing external fragmentation. Segments are only mapped to pages as needed, allowing non-contiguous allocation.

1.4 Problem - 4(Paging)

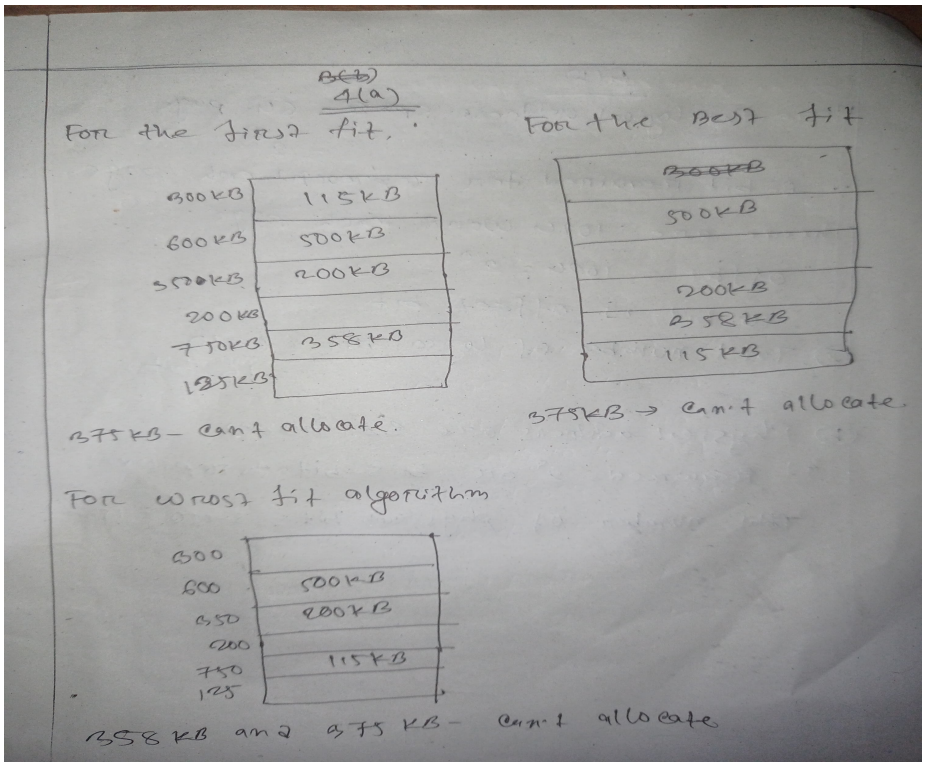


Figure 3:

1.5 Problem - 5(Paging)

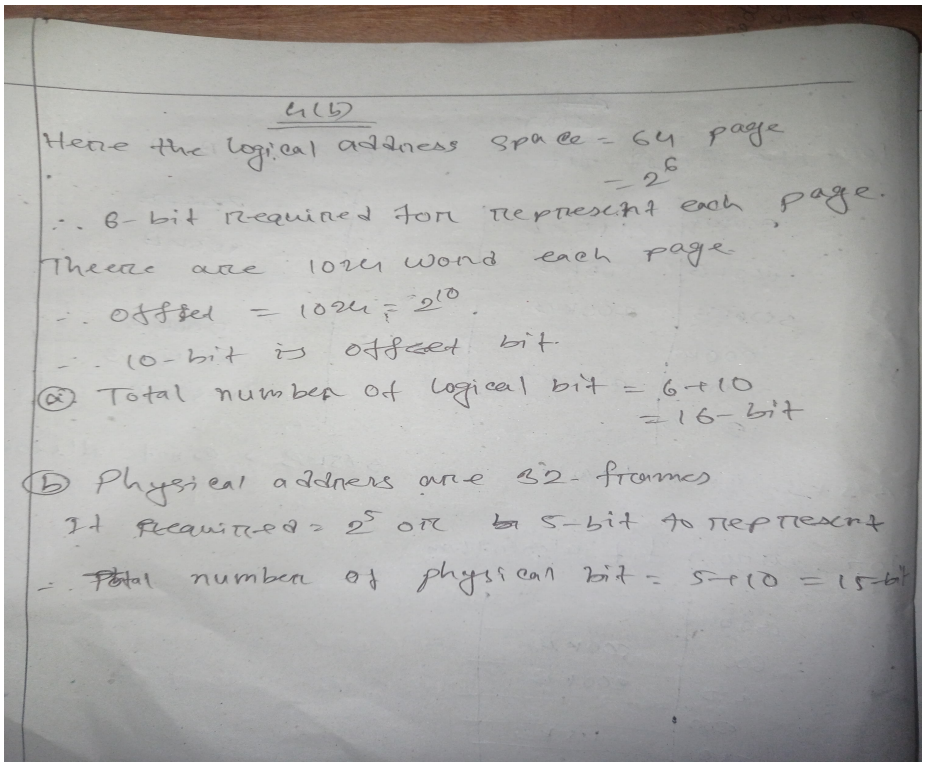

4(b)
Here the logical address space = 64 page
= 2^6
∴ 6-bit required for representing each page.
There are 1024 word each page
∴ Offset = 1024 = 2^{10}.
∴ 10-bit is offset bit.
(a) Total number of logical bit = 6+10
= 16-bit
(b) Physical address are 32-frames
It required = 2^5 or 5-bit to represent
∴ Total number of physical bit = 5+10 = 15-bit

Figure 4:

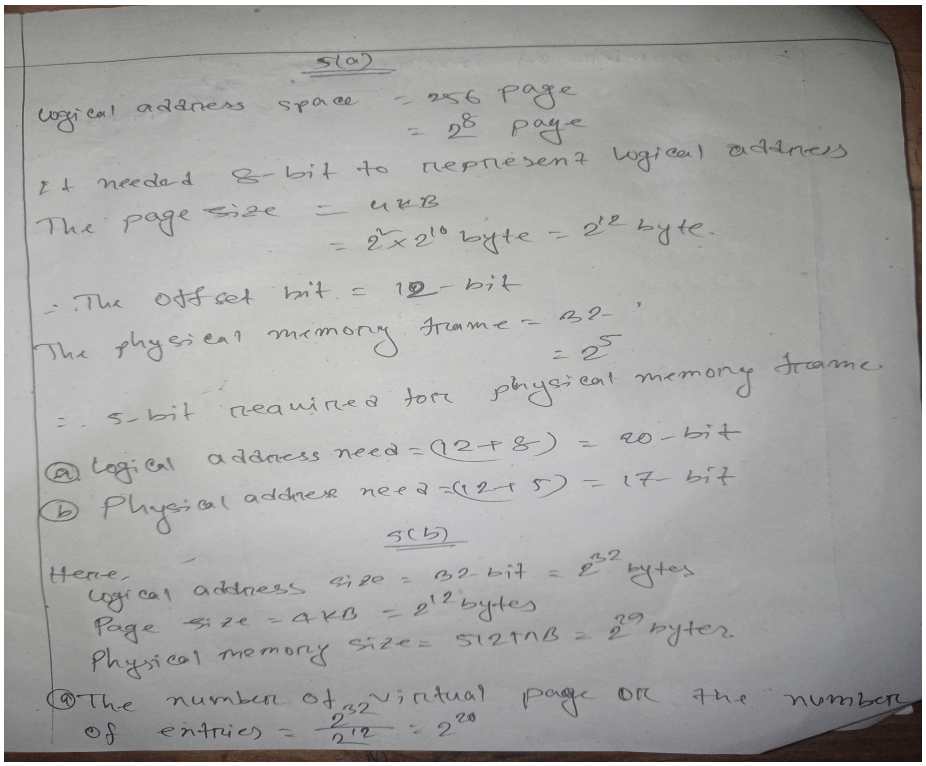

5(a)
Logical address space = 256 page
= 2^8 page
It needed 8-bit to represent logical address
The page size = 4KB
= $2^2 \times 2^{10}$ byte = 2^{12} byte.
∴ The offset bit = 12-bit
The physical memory frame = 32-
= 2^5
∴ 5-bit required for physical memory frame.
(a) Logical address need = (12+8) = 20-bit
(b) Physical address need = (12+5) = 17-bit
5(b)
Here,
Logical address size = 32-bit = 2^{32} bytes
Page size = 4KB = 2^{12} bytes
Physical memory size = 512MB = 2^{29} byte.
(a) The number of virtual page or the number of entries = $\frac{2^{32}}{2^{12}}$ = 2^{20}

Figure 5:

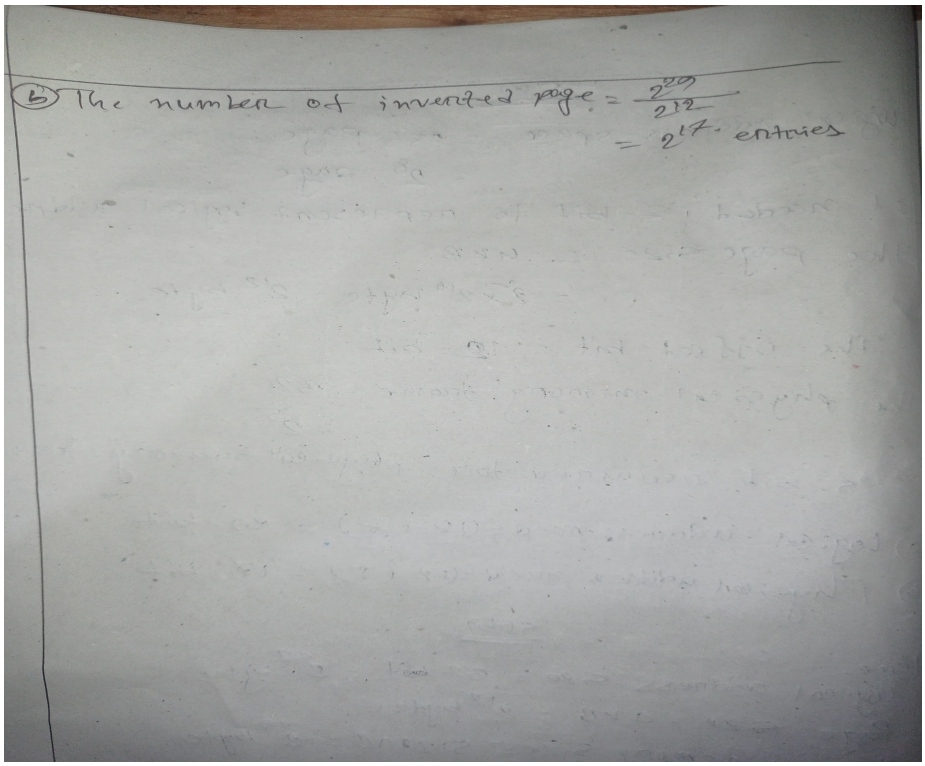


Figure 6:

OS All Assignment Report

Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Class Test - 09-06-24

1.1 Problem - 1

1.1.1 a. Atomic value

Assume val is an atomic integer in a Linux system. What is the value of val after the following operations have been completed?
atomic
atomic
atomic
atomic
atomic
atomic
set(val, 10);
sub(8, val);
inc(val);
inc(val);
add(6, val);
sub(3, val);

Answer:
The value sequence given below:
set : val = 10
sub : val = 10-8 = 2
int : val = 2+1 = 3
inc : val = 3+1 = 4
add : val = 4+6 = 10
sub : val = 10-3 = 7
As a result, the final value of val is 7

1.1.2 b. Describe how deadlock is possible with the dining philosophers problem.

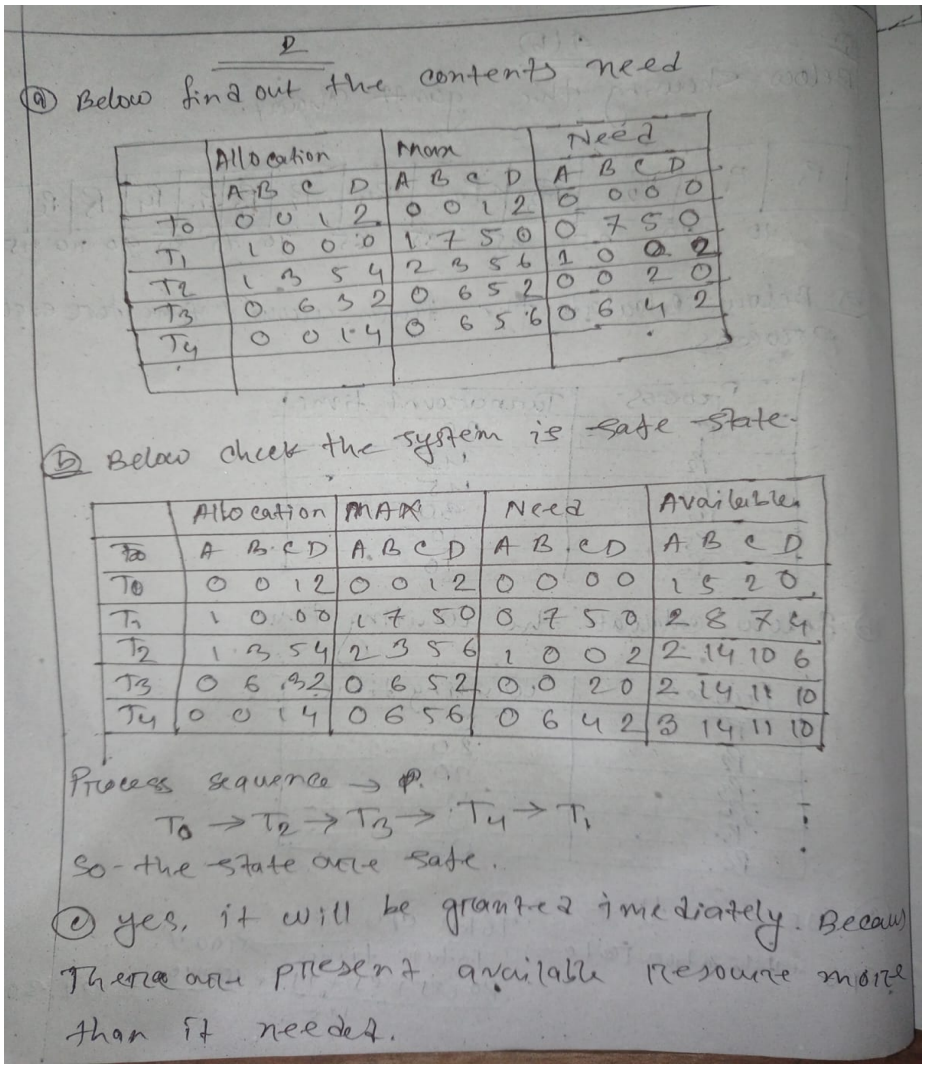
The Dining Philosophers Problem is a classic synchronization problem used to illustrate issues related to resource sharing and deadlock in concurrent programming. The problem involves a certain number of philosophers sitting around a table, each with a plate of food and a fork placed between each pair of philosophers. The philosophers alternate between thinking and eating, but to eat, each philosopher must acquire two forks: one on their left and one on their right.

1.1.3 c. Needed of lock

Explain why Windows and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, and condition variables. In each case, explain why the mechanism is needed.

Answer:
Operating systems like Windows and Linux implement multiple locking mechanisms to provide various ways of synchronizing concurrent processes or threads, depending on the specific needs of the situation. Different types of synchronization primitives are optimized for different scenarios to balance performance, fairness, and deadlock prevention.
Here's an explanation of the most common synchronization primitives — spinlocks, mutexes, semaphores, and condition variables — and the circumstances under which Windows and Linux use them.
Spinlock:
A spinlock is a type of lock where the thread continuously checks whether the lock is available and "spins" (or loops) while waiting for the lock to become available. Spinlocks are typically implemented using a busy-waiting loop.
It is uses when we needed short duration and low latency requirements.
It also uses for low overhead and quick lock aquisition.
Mutex Lock:
A mutex is a synchronization primitive used to ensure that only one thread can access a critical section at any given time. If a thread attempts to lock a mutex that is already locked by another thread, the requesting thread is blocked until the mutex is released.
It is uses when we needed long duration, wait and sleep neede.
It uses for avoid race condition and fairness.
Semaphore:
A semaphore is a synchronization primitive that is used to control access to a common resource by multiple threads in a concurrent system.
It is uses when we needed resource pool management and Producer-consumer problem.
It uses for Controlling Access to Shared Resources and counting and signaling.

1.2 Problem - 2(Bankers Algorithm)



D

(a) Below find out the contents need

	Allocation	Max	Need
	A B C D	A B C D	A B C D
T_0	0 0 1 2	0 0 1 2	0 0 0 0
T_1	1 0 0 0	1 7 5 0	0 7 5 0
T_2	1 3 5 4	2 3 5 6	1 0 0 2
T_3	0 6 3 2	0 6 5 2	0 0 2 0
T_4	0 0 1 4	0 6 5 6	0 6 4 2

(b) Below check the system is ~~safe~~ state.

	Allocation	MAX	Need	Available
~~T0~~	A B C D	A B C D	A B C D	A B C D
T_0	0 0 1 2	0 0 1 2	0 0 0 0	1 5 2 0
T_1	1 0 0 0	1 7 5 0	0 7 5 0	2 8 7 4
T_2	1 3 5 4	2 3 5 6	1 0 0 2	2 14 10 6
T_3	0 6 3 2	0 6 5 2	0 0 2 0	2 14 11 10
T_4	0 0 1 4	0 6 5 6	0 6 4 2	3 14 11 10

Process sequence → P.

$T_0 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$

So the state are safe.

(c) yes, it will be granted immediately. Because There are present available resource more than it needed.

Figure 1:

1.3 Problem - 3

1.3.1 A. Starvation Check

Can a system detect that some of its threads are starving? If you answer “yes,” explain how it can. If you answer “no,” explain how the system can deal with the starvation problem.

Answer:
Yes, a system can detect that some of its threads are starving, but the detection and handling of starvation are complex and depend on the underlying operating system's scheduling mechanism and policies. Here's an explanation of how a system can detect and manage starvation:
Detect thread starvation:
System can deted starvation by checking monitoring waiting time, Tracking CPU time allocation, priority invertion, fairness check, deadlock detection.
Deal with the starvation:
It can deal with the starvation with aging, fairness scheduling algorithm, preemption, resource reservation, timeout, multithreaded environments.

1.3.2 B. Deadlock

As an OS developer what do you prefer: deadlock avoidance, deadlock prevention or recovery from deadlock? Justify your answer.

Answer:
As an OS developer I prefer deadlock avoidance, deadlock prevention or deadlock recovery. Because, if we avoid the deadlock we can get next processes output, else we have to be stopped at the deadlock place. Sometimes we also needed the deadlock prevention, because of if we needed those processes result, we have to prevernt and recovery the deadlock.

1.3.3 C. Difference between

Spinlock and Mutex Lock

Spinlock:
1. A spinlock is a type of lock where the thread repeatedly checks (or ”spins”) to see if the lock is available. If the lock is not available, the thread keeps busy-waiting in a tight loop until it can acquire the lock.
2. The thread continuously ”spins” in a loop, checking the lock. This can waste CPU cycles if the lock is held for a long time.
3. High – Spinlocks consume CPU cycles even when the lock is not available, which can lead to inefficient CPU utilization, especially when the lock is held for a long time.
4. Spinlocks themselves do not inherently prevent deadlock, and improper use

can lead to deadlock.

Mutex:
1. A mutex (short for mutual exclusion) lock ensures that only one thread can access a critical section at a time. If a thread cannot acquire the mutex because it's locked, it will sleep and not consume CPU cycles.
2. The thread is put to sleep or blocked until the lock becomes available, which is more efficient in terms of CPU utilization.
3. Low – Mutex locks allow the system to schedule other threads when the lock is unavailable, making better use of the CPU.
4. Mutex locks are typically designed to prevent deadlock if used correctly, as long as careful programming practices (like lock ordering) are followed.

Deadlock and Livelock

Deadlock:

Livelock:

1.4 Problem - 4(Bankers Algorithm)

A. Dead lock check

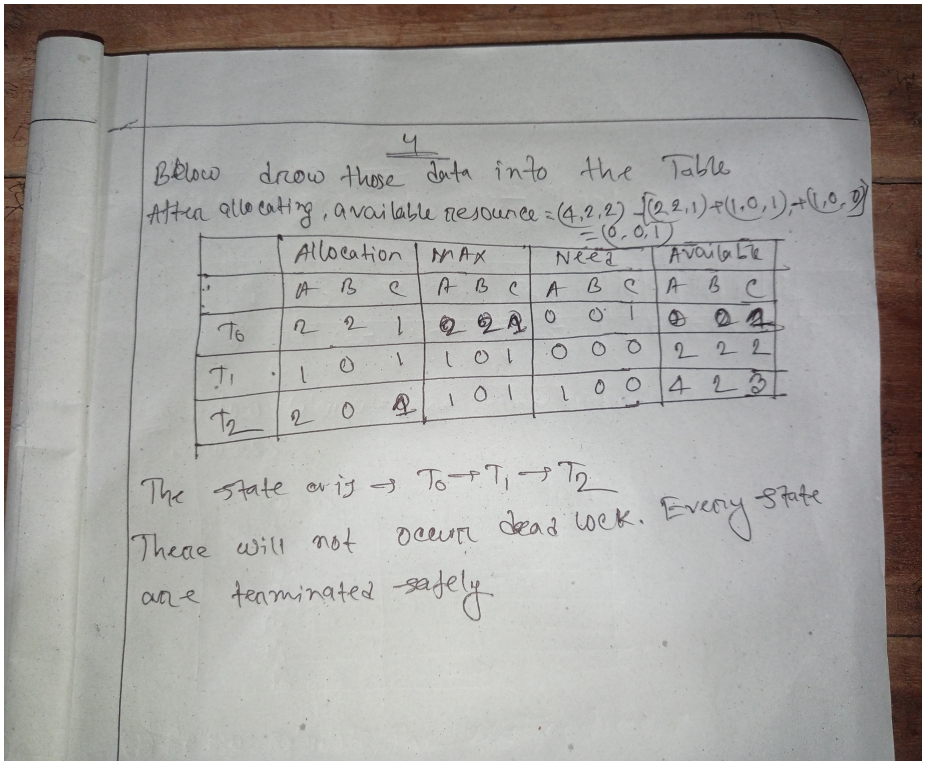


Figure 2:

B. Can indefinite blocking occur? Explain your answer.
Answer:
Yes, indefinite blocking (also known as starvation) can occur even if a system is in a safe state and deadlock is avoided. Indefinite blocking happens when a process is never able to acquire the resources it needs to complete, despite the system being in a safe state. This can occur when other processes continually receive priority over the blocked process, and the blocked process is never granted the resources it requires to proceed.

1.5 Problem - 5(Bankers Algorithm)

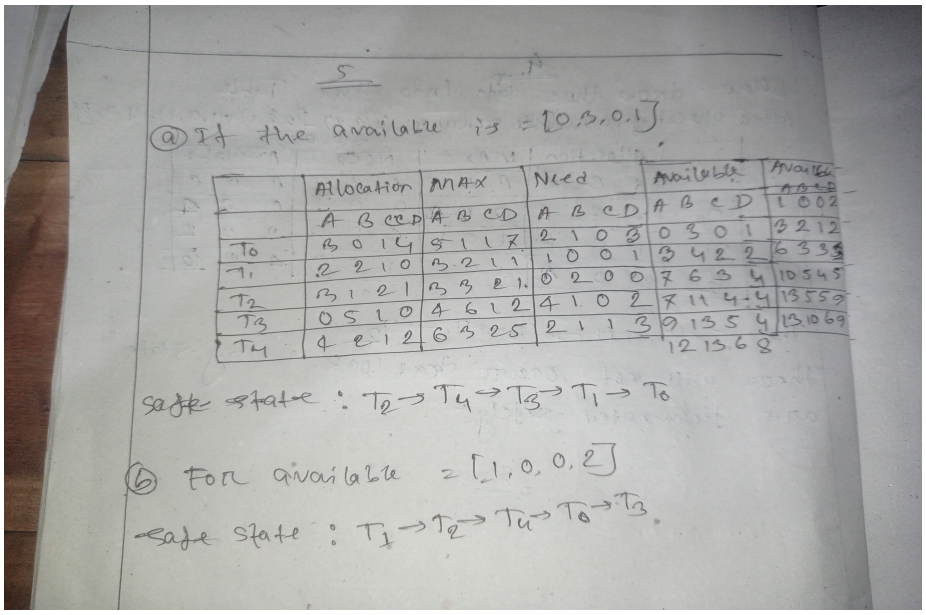

5
(a) If the available is = [0,3,0,1]
Allocation
MAX
Need
Available
A B C D
T0
T1
T2
T3
T4
safe state : T2 → T4 → T3 → T1 → T0
(b) For available = [1,0,0,2]
safe state : T1 → T2 → T4 → T0 → T3.

Figure 3:

OS All Assignment Report

Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Class Test - 04-06-24

1.1 Problem - 1

1.1.1 Advantage of preemptive kernel

What advantages does Linux have of being a preemptive kernel?

A preemptive kernel means that the operating system kernel can interrupt (or preempt) the currently running process at any time to allow another process to run. This is a feature of Linux and other modern operating systems. There are several advantages to Linux being a preemptive kernel:
1. It improve the responsiveness like as it make the faster response and better user experience.
2. It also make fairer to the process scheduling such as process prioritization and prevent the process starvation.
3. It also improve the multi-tasking.
4. Improving the interrupt handlin.

1.1.2 Share memory problem

Explain when executing the following two instructions belonging to two processes running in parallel is problematic? i++; and i–; when i is in shared memory.

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

int r = 1;
```

```c
int f1(){
    for(int i=0; i<10000; i++){
        r++;
    }
}
int f2(){
    for(int i=0; i<10000; i++){
        r--;
    }
}

int main(){
    pthread_t t1,t2;
    pthread_create(&t1, NULL, (void*)f1, NULL);
    pthread_create(&t2, NULL, (void*)f2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Result : %d\n",r);

    return 0;
}
```

If we use this code segment like as i++ and i– for the shared memory, the output should be 1. But we get the output given below:

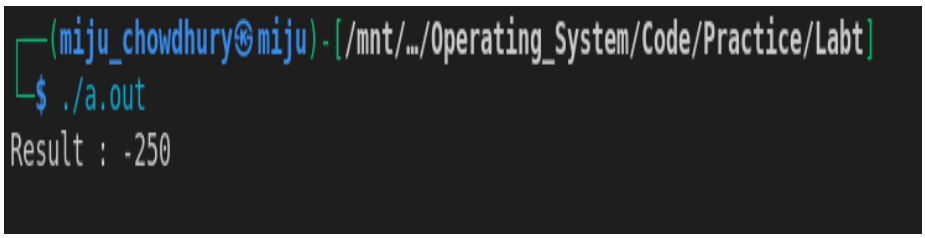


Figure 1:

Here are happened data inconsistency, because of the are used the shared variable are without synchronizely, those thread can't update the data sequentially. Therefore, there happened problematic output.

1.2 Illustrate Three case of race condition

Below showing the three cases of race condition.
Case 1: Increment and decrement the shared variable:

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

int r = 1;

int f1(){
    for(int i=0; i<10000; i++){
        r++;
    }
}
int f2(){
    for(int i=0; i<10000; i++){
        r--;
    }
}

int main(){
    pthread_t t1,t2;
    pthread_create(&t1, NULL, (void*)f1, NULL);
    pthread_create(&t2, NULL, (void*)f2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Result : %d\n",r);

    return 0;
}
```

```
┌──(miju_chowdhury㉿miju)-[/mnt/…/Operating_System/Code/Practice/Labt]
└─$ ./a.out
Result : -250
```

Figure 2:

Case 2: Bank accounts withdraw and deposit:

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

int balance = 1000000;

int deposit(){
    for(int i=0; i<10000; i++){
        balance++;
    }
}
int withdraw(){
    for(int i=0; i<10000; i++){
        balance--;
    }
}

int main(){
    pthread_t t1,t2;
    pthread_create(&t1, NULL, (void*)deposit, NULL);
    pthread_create(&t2, NULL, (void*)withdraw, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Result : %d\n",balance);

    return 0;
}
```

The output should be 1000000, but we find given below:

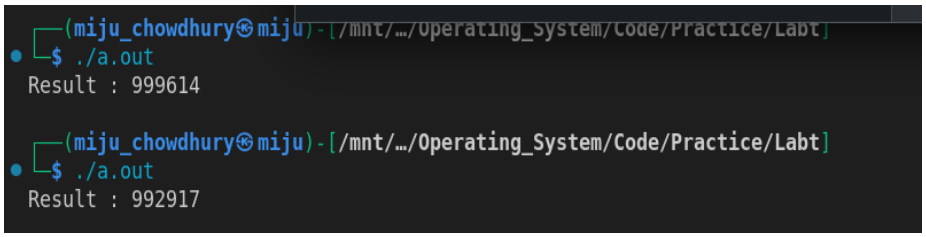


Figure 3:

Case 3:

1.3 Problem - 3

1.4 A. Semaphore Problem

Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

Below given a code exmaple for semaphore :

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


sem_t s;
int r = 1;

int f1(){
    sem_wait(&s);
    for(int i=0; i<1000; i++)
        r++;
    sem_post(&s);
}

int f2(){
    sem_wait(&s);
    for(int i=0; i<1000; i++)
        r--;
    sem_post(&s);
}

int main(){
    sem_init(&s, 0,1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,(void*)f1,NULL);
    pthread_create(&t2,NULL,(void*)f2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    printf("Result : %d\n",r);

}
```

Result is : 1

The wait() and signal() operations are the foundation of semaphores used for

synchronization. These operations are typically used to ensure mutual exclusion by controlling access to critical sections of code in a concurrent system. If wait() and signal() are not executed atomically, race conditions may arise, violating mutual exclusion and leading to inconsistent or incorrect behavior.
Semaphore Operations are occurs like this:
1. wait() (also called P or down):
1.1 Decreases the semaphore value.
1.2 If the semaphore value is greater than 0, the process continues; otherwise, it is blocked.
2. signal() (also called V or up):
2.1 Increases the semaphore value.
2.2 If there are any blocked processes waiting on the semaphore, one of them is woken up.

1.5 Spinlock and mutex lock

Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:
• The lock is to be held for a short duration.
• The lock is to be held for a long duration.
• A thread may be put to sleep while holding the lock

Answer:
When choosing between a spinlock and a mutex lock, the decision depends on various factors, including how long the lock is held, the system's characteristics, and whether threads may sleep while holding the lock.
Spin lock are generally uses for short duration and Mutex lock are uses for long time duration. Spin lock dosen't allow the wait and sleep but the Mutex lock allows the wait and sleep for the thread creation.
As a result, 1. For the lock is to be held for a short duration we have to use Spin lock
2. The lock is to be held for a long duration we have to use Mutex lock. 3. A thread may be put to sleep while holding the lock we have to use Mutex lock.

1.6 Problem - 4

1.7 A. Disabling Interrupt

Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

Answer:
Disabling interrupts is a technique often used in low-level operating system code to implement synchronization primitives. Disabling interrupts for implementing synchronization primitives in a single-processor system is not appropriate for user-level programs because:
1. It affects the entire system, blocking all processes and external events, leading to inefficiency.
2. It requires kernel privileges, which user-space programs do not have.
3. It can cause starvation, deadlocks, and poor system responsiveness.
3. There are better alternatives, such as using atomic operations and user-space synchronization primitives, that provide more efficient, scalable, and safe ways to manage concurrency.

1.7.1 4(b) Multiprocessing systems

Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

Answer:
Disabling interrupts is not appropriate for implementing synchronization primitives in a multiprocessor system for the following reasons:

1. Interrupts are per-processor, not system-wide, so disabling interrupts on one processor does not provide mutual exclusion across multiple processors.
2. Cache coherence issues arise because disabling interrupts does not handle synchronization across caches in different processors.
3. It can lead to deadlocks and starvation, where processors are blocked or prevented from making progress.
4. Scalability becomes a problem, as interrupt disabling is not a scalable solution for large, multi-core systems.
5. Disabling interrupts can result in inefficient resource usage, as it prevents processors from handling interrupts and performing other tasks.

OS All Assignment Report

Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Class Test - 30.5

1.1 Problem -1

1.1.1 a

Figure out how often context switches occur in your system and describe it with a screen shot.

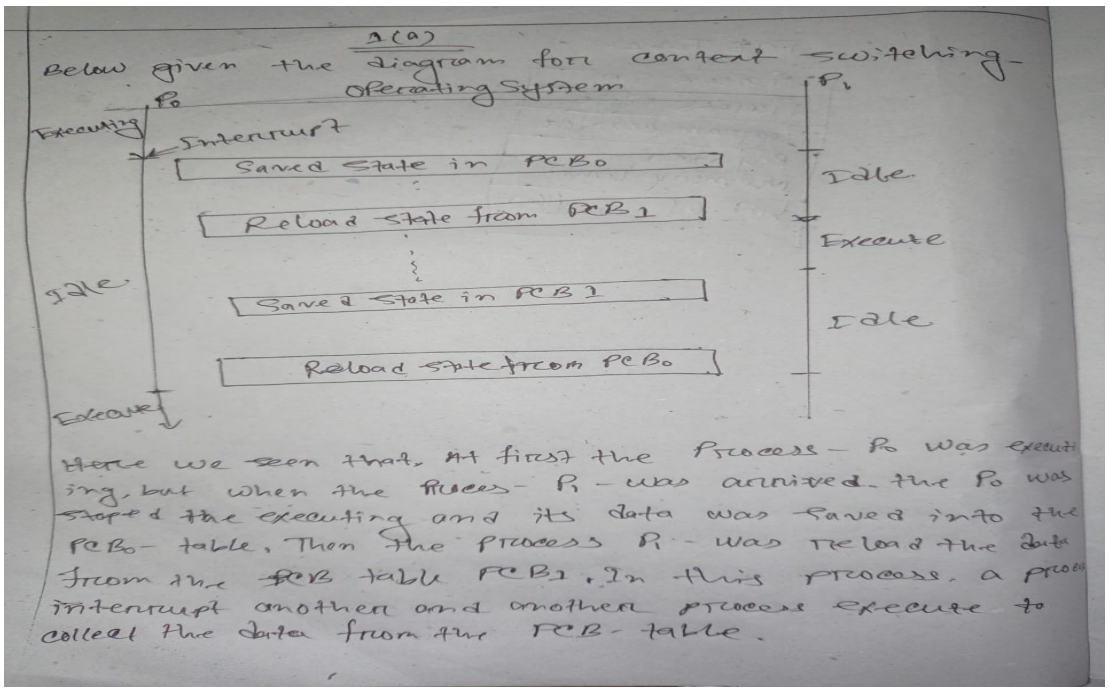


Figure 1:

1.1.2 b

Figure out the number of context switches occurred for a specific process in your system. Describe investigation steps in detail.

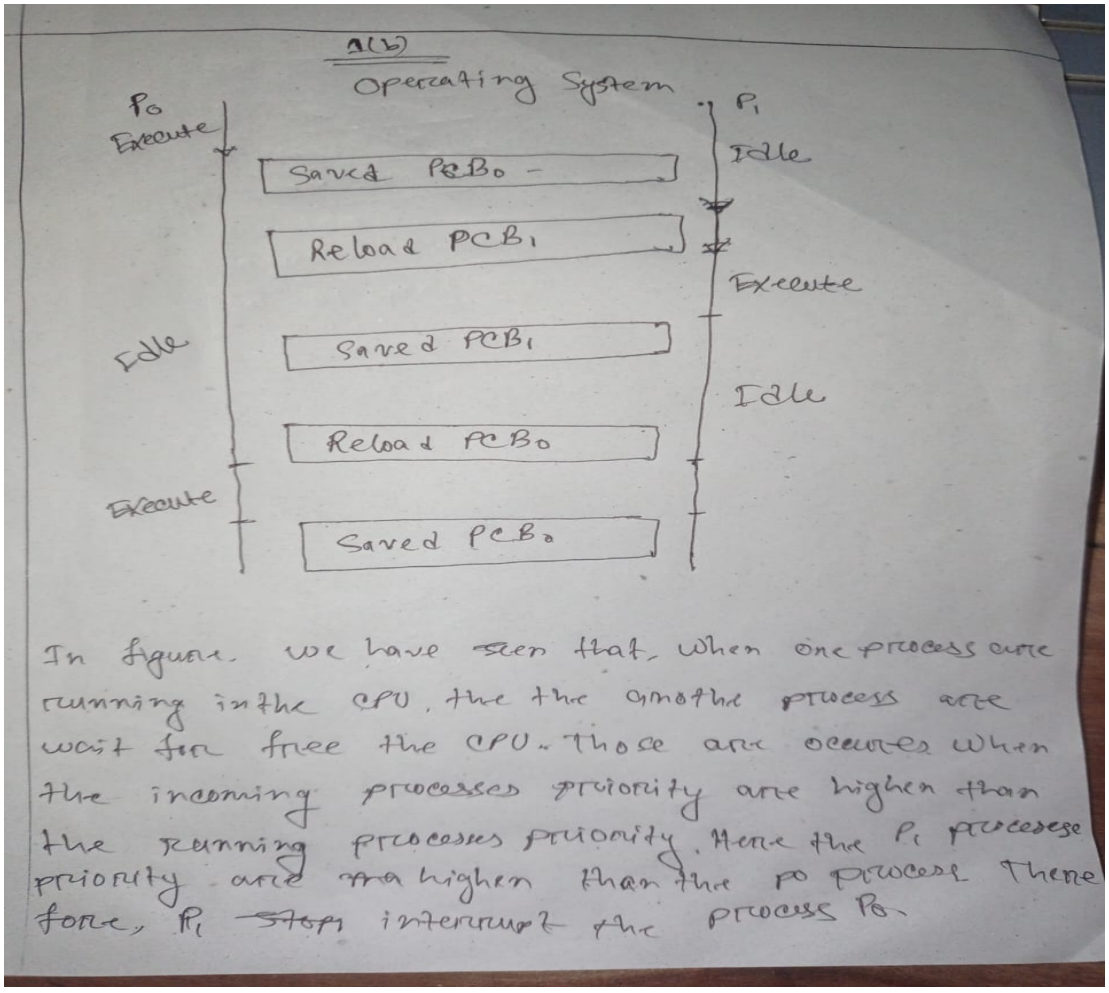


Figure 2:

1.1.3 c. What are the catchy things about:

A. Processor affinity:

Processor affinity refers to the concept of binding or pining a thread or process to a specific CPU core or a set of cores in a multi-core system. This can help

optimize performance by reducing cache misses and improving CPU cache locality, among other benefits. Here are some of the catchy things about processor affinity:
1. Improve cache locality
2. Reduced Context Switching Overhead
3. Avoiding CPU cache contention
4. Optimizing Multi-threaded Applications
5. Predictable performance
6. Reducing Power Consumption

B. Dispatcher

A dispatcher in an operating system, particularly in process management, is an essential component that manages the handoff between processes and the CPU. Here are some catchy of a dispatcher:
1. The Timekeeper of the CPU
2. Swift switcher
3. Balancing Act
4. The refere
5. Quick Decision-Maker
6. System protector.

C. Starvation of a process

Starvation in process scheduling is a condition where a process waits indefinitely without getting necessary CPU resources, often due to other higher-priority tasks continuously taking precedence. Below given some catchy of dispatcher :
1. The Waiting Game
2. The priority track
3. The domino effect
4. Hogging Resources
5. The silent killer
6. Fairness over speed

1.2 Problem - 2

1.2.1 A. Scheduling Algorithm :

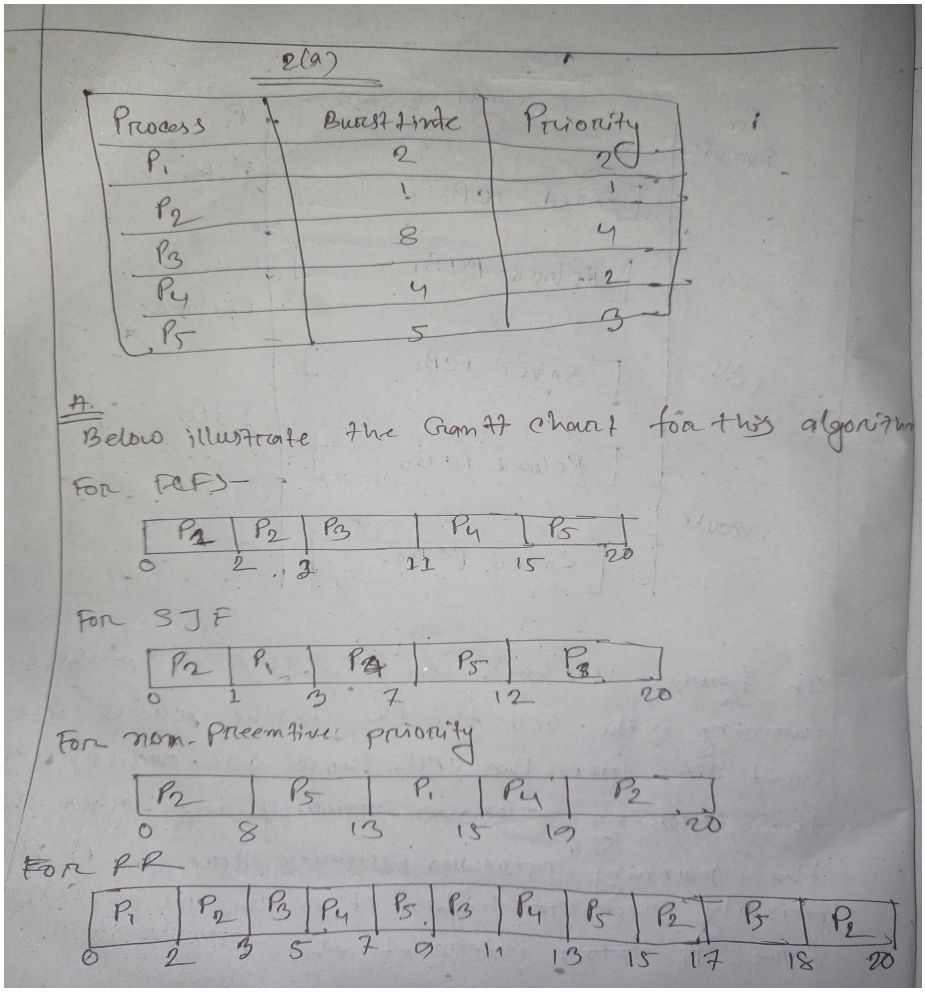


Figure 3:

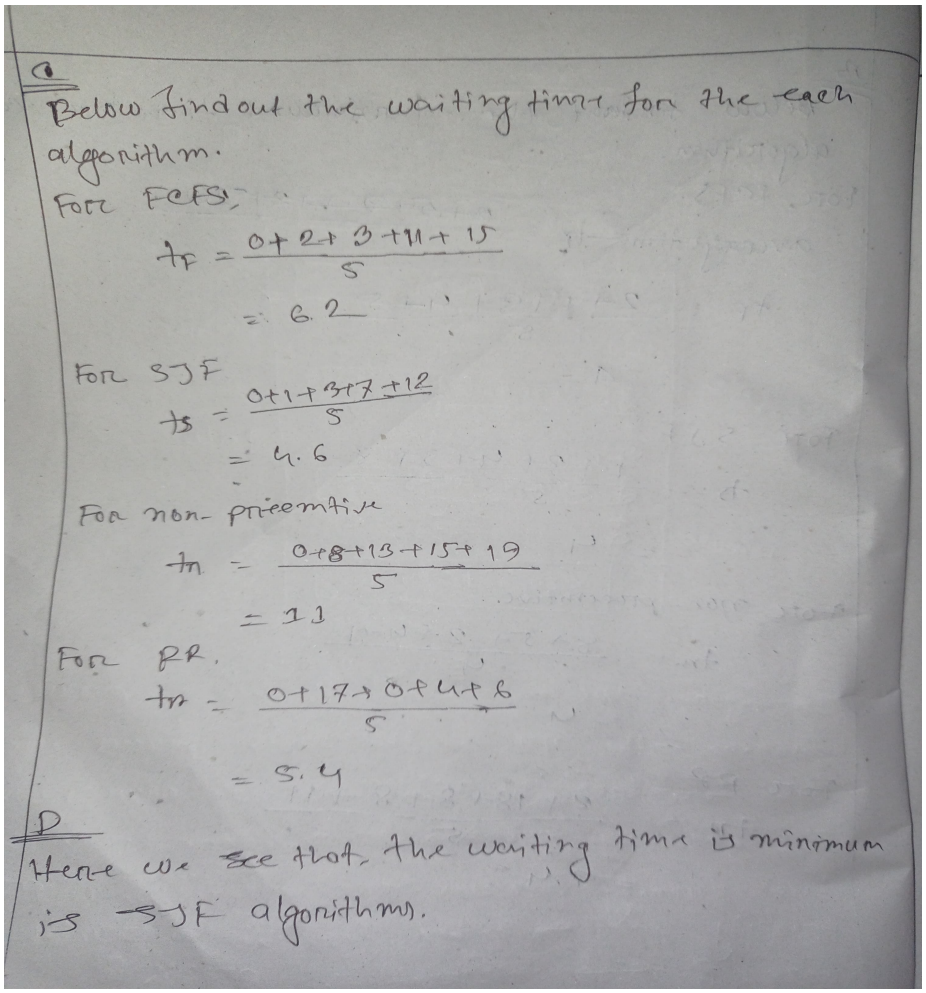

c
Below find out the waiting time for the each algorithm.
For FCFS;
$t_f = \frac{0+2+3+11+15}{5}$
$= 6.2$
For SJF
$t_s = \frac{0+1+3+7+12}{5}$
$= 4.6$
For non-preemtive
$t_n = \frac{0+8+13+15+19}{5}$
$= 11$
For RR.
$t_r = \frac{0+17+0+4+6}{5}$
$= 5.4$
D
Here we see that the waiting time is minimum is SJF algorithms.

Figure 4:

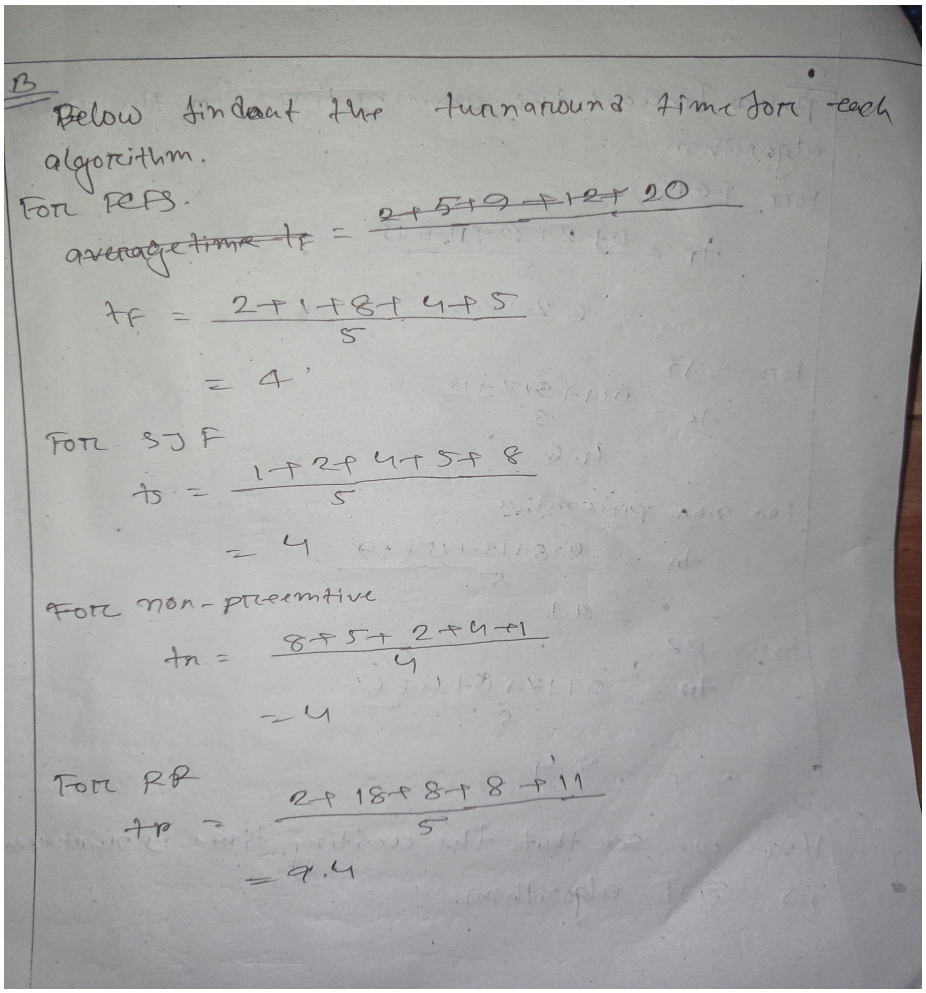

B
Below find out the turnaround time for each algorithm.
FOR RFS.
average time tf = 2+5+9+12+20
tf = 2+1+8+4+5 / 5
= 4
FOR SJF
ts = 1+2+4+5+8 / 5
= 4
FOR non-preemtive
tn = 8+5+2+4+1 / 4
= 4
FOR RR
tp = 2+18+8+8+11 / 5
= 9.4

Figure 5:

1.3 Problem -3

1.3.1 Calculate priority

The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:
Priority = (recent CPU usage / 2) + base where base = 60 and recent CPU usage refers to a value indicating how often a process has used the CPU since priorities were last recalculated. Assume that recent CPU usage for process P 1 is 40, for process P 2 is 18, and for process P 3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU -bound process?

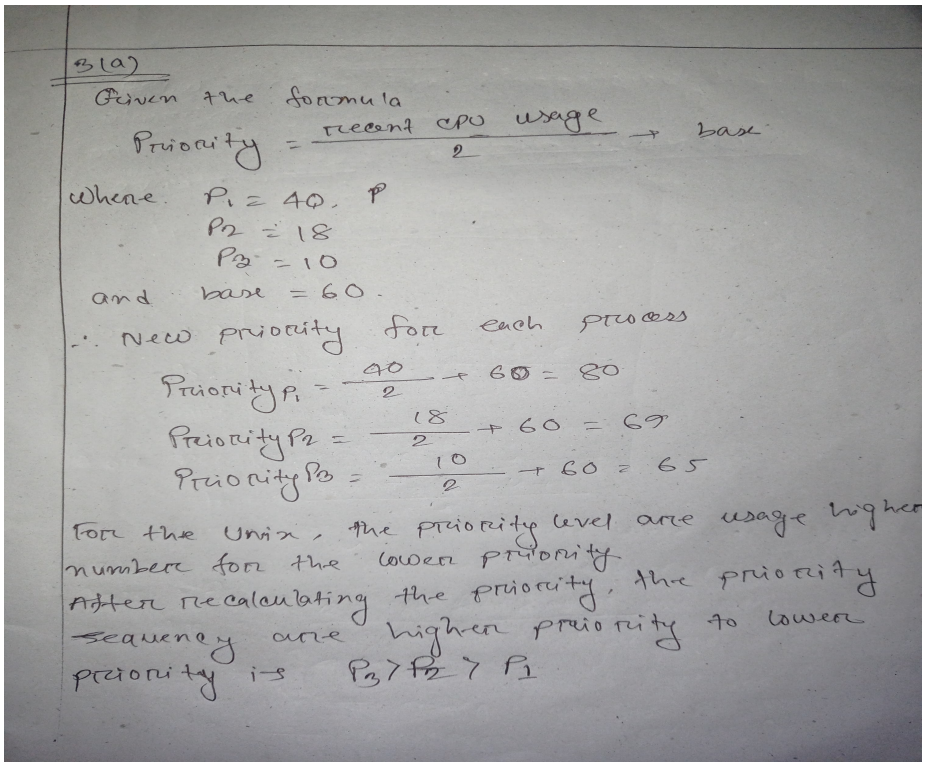


Figure 6:

1.4 Problem - 4

1.4.1 A. Ideal thread and ideal processor

The terms "ideal thread" and "ideal processor" are used to describe an optimal relationship between threads and processors to achieve maximum efficiency and performance.

1.4.2 B. Scheduling Algorithm

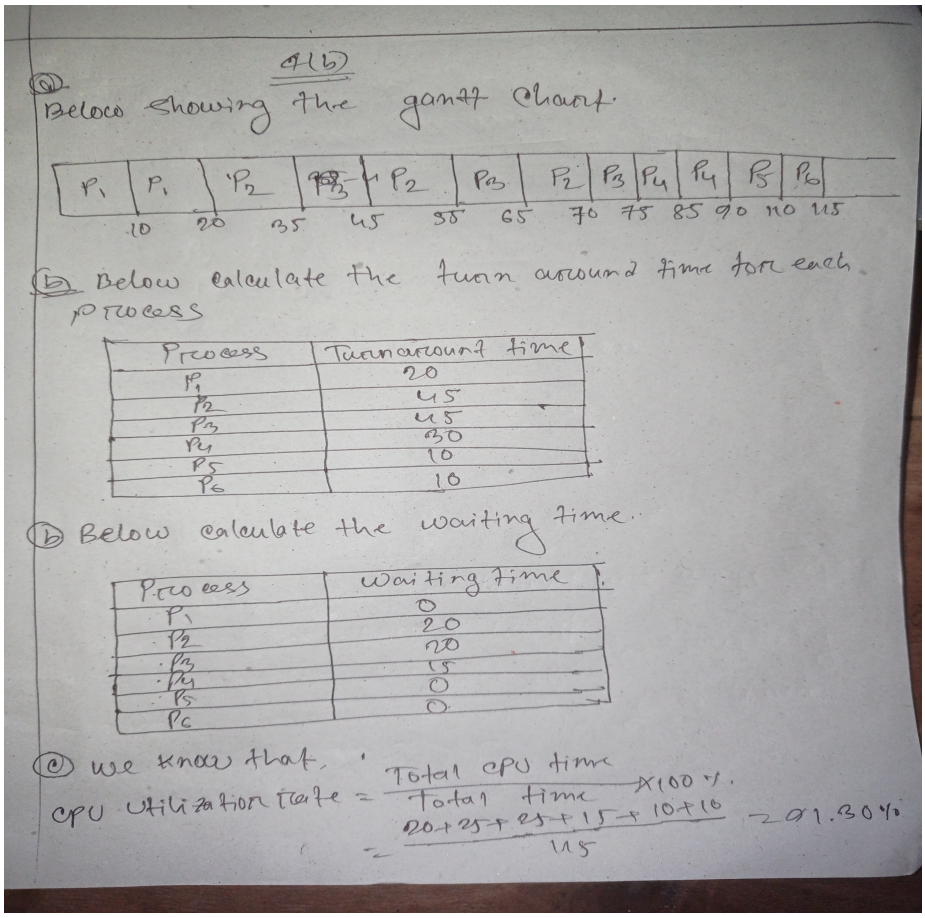


Figure 7:

OS All Assignment Report

Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Class Test - 26.5

1.1 Problem - 1

1.1.1 fork()

A. If one thread in the process calls fork() , does the new process duplicate all threads, or is the new process single-threaded?

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


int f1(){
    pid_t c = fork();
    int a=4,b=5;
    printf("sum = %d\n",a+b);
}
int f2(){
    int a=4,b=5;
    printf("Sub = %d\n",a-b);
}
int main(){
    pthread_t t1,t2;
    pthread_create(&t1,NULL,(void*)f1,NULL);
    pthread_create(&t2,NULL,(void*)f2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
}
```

The output for the upper code is:
There will not effect to any other thread, If a single thread the fork() function.

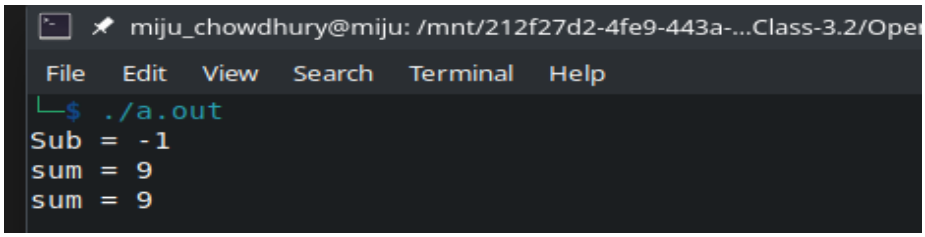


Figure 1: Output

Because, every threads are running into different stack pointer, different register. So, there will not effect any other if we call the fork() in a single thread.

1.1.2 exclp()

B. If a thread invokes the exec() system call, does it replace the entire code of the process?

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


int f1(){
    execlp("/bin/ls","ls",NULL);
    printf("Hello world.\n");
}

int f2(){
    int a=4,b=5;
    printf("Sub = %d\n",a-b);
}

int main(){
    pthread_t t1,t2;
    pthread_create(&t1,NULL,(void*)f1,NULL);
    pthread_create(&t2,NULL,(void*)f2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

}
```

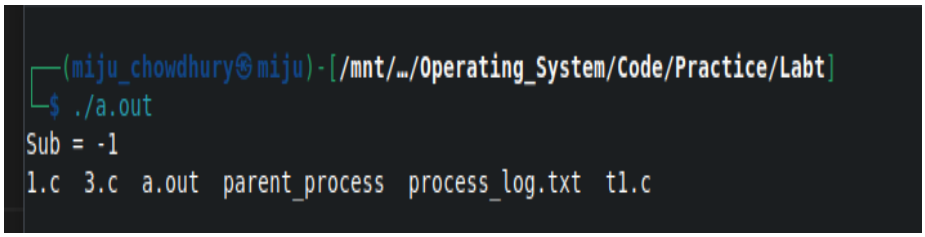


Figure 2: For calling exclp

The output are given below if we call exclp() in one thread: If we call the exclp(), another thread are not effected for calling exclp().

1.1.3 exclp() and fork()

C. If exec() is called immediately after forking, will all threads be duplicated?

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
int f1(){
    printf("Hello world.\n");
}
int f2(){
    int a=4,b=5;
    printf("Sub = %d\n",a-b);
}

int main(){
    pthread_t t1,t2;
    pthread_create(&t1,NULL,(void*)f1,NULL);
    pthread_create(&t2,NULL,(void*)f2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    fork();
    execlp("/bin/ls","ls",NULL);

}
```

Below showing the output for the upper code:
If we use execlp() after calling the fork() and after the create the thread, then thread will not stopped and but the execlp() give the output for main process and child process.

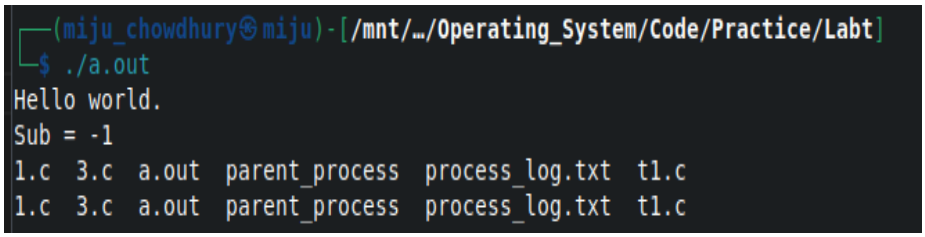


Figure 3: execlp() after calling thread

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


int f1(){
    printf("Hello world.\n");
}

int f2(){
    int a=4,b=5;
    printf("Sub = %d\n",a-b);
}

int main(){
    fork();
    execlp("/bin/ls","ls",NULL);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,(void*)f1,NULL);
    pthread_create(&t2,NULL,(void*)f2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

}
```

The output if we use the execlp() before the calling thread:

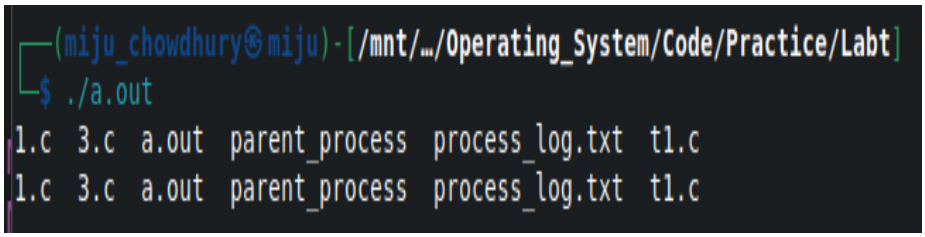


Figure 4: execlp() before calling thread

Then execlp() give the output two times and that is for parent process and child process. But there will not execute any thread.

1.2 Problem - 2

Write a C program to show how two threads can communicate by the help of 'signal'.

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

#define SIGM SIGQUIT

sigset_t s;

void* reciver(void* arg){
    printf("Signal Reciver\n");
    int sig;
    sigwait(&s,&sig);
    printf("Recived Signal: %d\n",sig);

}
void* sender(void *arg){
    sleep(2);
    printf("Sending signal %d\n",SIGM);
    pthread_kill(*(pthread_t*)arg, SIGM);
}

int main(){
    pthread_t t1,t2;
    sigemptyset(&s);
```

```
    sigaddset(&s, SIGM);
    pthread_sigmask(SIG_BLOCK, &s,NULL);
    pthread_create(&t1,NULL,reciver,NULL);
    pthread_create(&t2,NULL,sender,&t1);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

}
```

The output of the upper code is given below:

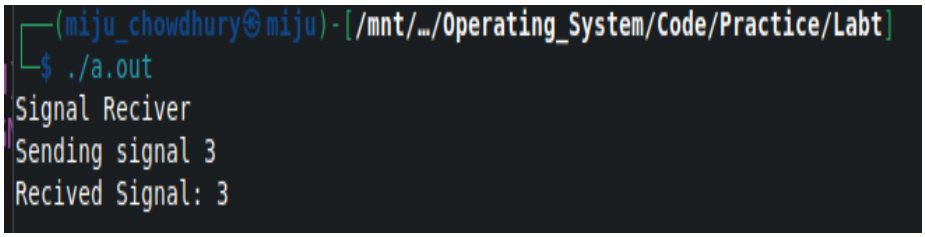


Figure 5: Thread Communication

1.3 Probelm-3

Write a C program to show how data inconsistency arises in a multi-threaded process.

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

int r = 1;

int f1(){
    for(int i=0; i<10000; i++){
        r++;
    }
}
int f2(){
    for(int i=0; i<10000; i++){
        r--;
    }
}
```

```c
int main(){
    pthread_t t1,t2;
    pthread_create(&t1, NULL, (void*)f1, NULL);
    pthread_create(&t2, NULL, (void*)f2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Result : %d\n",r);

    return 0;
}
```

Below showing the output for the upper code :

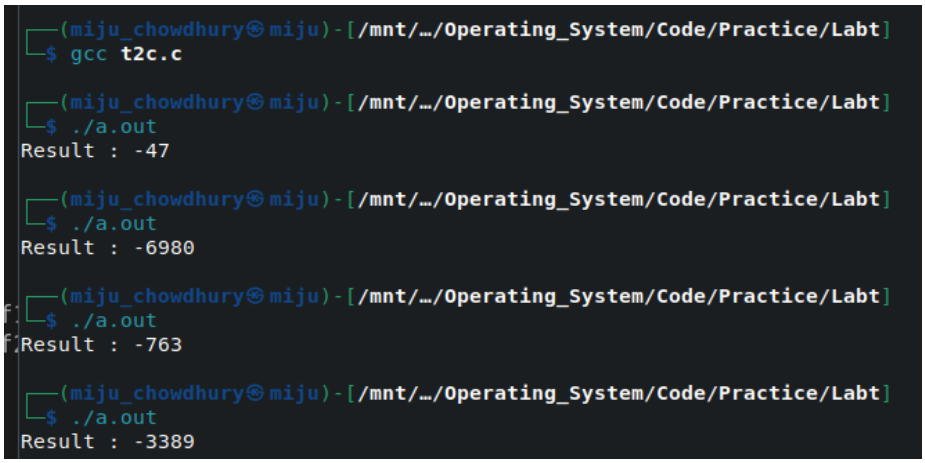


Figure 6: Data inconsistancy

The output should be given 1. But if we use the shared variable for the multithreading, then the same variables are uses in every thread. therefore, there will be happend the race condition. As a result, it given the different output for every time run the program.

6.OS

Tuesday, October 22, 2024 01:14 PM

1. Sending Signals

1a. How can a process send a signal to:

- **A. Another Process**:
  - A process can send a signal to another process using the kill() system call, which takes the process ID (PID) of the target process and the signal to be sent as arguments.

    c
    Copy code
    kill(target_pid, SIGTERM);
- **B. A Specific Thread of the Same Process**:
  - In a multi-threaded program, a process can send a signal to a specific thread by using the pthread_kill() function, passing the thread ID (TID) and the desired signal.

    c
    Copy code
    pthread_kill(thread_id, SIGUSR1);
- **C. A Specific Thread of a Different Process**:
  - To send a signal to a specific thread in another process, the tgkill() system call is used, which takes the PID of the target process, the TID of the target thread, and the signal as arguments.

    c
    Copy code
    tgkill(target_pid, target_tid, SIGUSR1);

1b. What happens if a parent process and its child process try to kill each other by sending a SIGKILL signal?

- **Scenario 1**:
  - The parent sends SIGKILL to the child. The child process is immediately terminated.
- **Scenario 2**:
  - The child sends SIGKILL to the parent. The parent process is immediately terminated.
- **Effect**:
  - If the parent kills the child, the child is removed from the process table and will not be able to send signals or perform any actions afterward.
  - If the child kills the parent, it will become an orphan, and the init process will adopt it.
  - In both scenarios, the processes are terminated, and any resources they hold will be released back to the system.

1c. In Linux, how many signals can be caught and cannot be caught by a user-defined signal handler?

- **Signals that can be caught by a user-defined signal handler**:
  - SIGINT
  - SIGQUIT
  - SIGTERM
  - SIGUSR1
  - SIGUSR2
  - SIGPIPE
  - SIGALRM

- SIGCONT
- SIGHUP
- SIGTSTP
- SIGTTIN
- SIGTTOU
- SIGSEGV
- (and others, depending on the system)

- **Signals that cannot be caught by a user-defined signal handler**:
  - SIGKILL
  - SIGSTOP
  - SIGCONT (can be caught but generally has a different handling)
  - SIGQUIT
  - SIGILL
  - SIGTRAP
  - SIGABRT
  - SIGBUS
  - SIGFPE
  - SIGSEGV

    The total number of signals can vary by system, but typically there are around 31 standard signals.

2. Differences

2a. Write down the differences between:

- **A. User Thread and Kernel Thread**:

Feature	User Thread	Kernel Thread
Management	Managed by user-level libraries	Managed by the operating system kernel
Scheduling	Cannot take advantage of multiple processors	Can take advantage of multiple processors
Overhead	Lower overhead due to no kernel context switches	Higher overhead due to kernel involvement
Visibility	The kernel is unaware of user threads	The kernel is aware of kernel threads

- **B. Child Process and Thread**:

Feature	Child Process	Thread
Resource Allocation	Has its own memory space	Shares memory space with its parent
Overhead	More overhead due to separate memory	Lower overhead since threads share memory
Communication	Inter-process communication needed	Direct communication through shared variables
Creation	Created using fork()	Created using pthread_create() or similar

2b. Performance Implications in Many-to-Many Model

- **A. Number of kernel threads < Number of processing cores**:
  - **Implication**: Underutilization of CPU resources; some cores may be idle while user threads are waiting for kernel threads, leading to poor performance.

- **B. Number of kernel threads = Number of processing cores**:
  - **Implication**: Optimal performance; each kernel thread can be scheduled on a separate core, maximizing CPU utilization and minimizing context switching.
- **C. Number of kernel threads > Number of processing cores but < Number of user-level threads**:
  - **Implication**: Some kernel threads will be waiting for CPU time, potentially leading to increased context switching and overhead, but still allowing for parallel execution of user threads.

3a. Behavior of fork() in Linux with Threads

- If one thread in a program calls fork(), the new process is single-threaded. Only the calling thread is duplicated in the new process; the other threads of the original process do not exist in the child process.

3b. Is the Linux Kernel Multithreaded?

- **Justification**: Yes, the Linux kernel is multithreaded. It can manage multiple threads of execution concurrently. The kernel can schedule and manage threads independently, allowing for parallel execution of tasks, such as handling multiple I/O operations simultaneously. This design enhances the performance and responsiveness of the operating system.

3c. Serving 1000 Clients: Best Approach

- **A. Service Instructions in Main Thread**:
  - **Disadvantage**: Blocking operations can lead to poor responsiveness, as the main thread will be busy serving clients.
- **B. Separate Child Process for Each Client**:
  - **Disadvantage**: High overhead due to the cost of creating and managing many processes. This approach can lead to resource exhaustion.
- **C. Separate Thread for Each Client**:
  - **Advantage**: Lower overhead compared to processes, allowing for efficient handling of many clients. Threads can share resources, leading to better performance.
- **D. Hybrid Structure**:
  - **Advantage**: This structure provides a balance of resource management and performance. It reduces overhead by limiting the number of processes while leveraging threads to handle multiple clients efficiently within each process.

4a. Reasons for Separate Registers, Stack, and Program Counter in Multi-threaded Processes

1. **Independent Execution**: Each thread needs its own register set to maintain the context of execution. This ensures that when switching between threads, the CPU can restore the correct state without interference.
2. **Stack Isolation**: Each thread requires a separate stack to manage function calls, local variables, and return addresses. This isolation prevents stack corruption that could occur if multiple threads share the same stack.
3. **Program Counter Tracking**: Each thread maintains its own program counter, allowing it to track its position in the execution sequence independently of other threads. This is crucial for correct execution flow and handling of function calls and returns.

1. Problems Faced by Multiple Child Processes

b. Do multiple child processes face any problems if they try to update:

- **Local Variables**:
  - **Answer**: No, multiple child processes do not face problems when updating local

variables. Each child process has its own separate memory space, meaning local variables are not shared. Changes made to local variables in one child process do not affect local variables in another process. This isolation ensures that each process operates independently without interference from others.

- **Global Variables**:
  - **Answer**: Yes, multiple child processes can face problems when updating global variables. Since global variables are shared across all processes, if multiple child processes modify a global variable simultaneously, it can lead to race conditions and unpredictable behavior. For example, if two processes read a global variable, increment it, and then write it back, the final value may not reflect all increments. To manage access to global variables, synchronization mechanisms (like mutexes) are necessary to ensure safe updates.

2. Problems Faced by Multiple Threads

c. Do multiple threads face any problems if they try to update:

- **Local Variables**:
  - **Answer**: No, multiple threads do not face problems when updating local variables. Each thread maintains its own stack, and local variables are stored in that stack. This means that changes made by one thread do not affect local variables in other threads, allowing independent execution without interference.
- **Global Variables**:
  - **Answer**: Yes, multiple threads can face problems when updating global variables. Since all threads within a process share the same memory space, accessing and modifying global variables concurrently can lead to race conditions. If two or more threads attempt to read and write to a global variable at the same time, it may result in inconsistent data or unexpected behavior. To prevent this, synchronization mechanisms (like mutexes, spinlocks, or semaphores) must be used to ensure that only one thread can access or modify the global variable at a time.

3. Main Thread Termination

5a. What happens when the main thread terminates before termination of:

- **Child Processes**:
  - **Answer**: If the main thread terminates before its child processes, those child processes do not automatically terminate. In a typical multi-threaded program, the child processes will continue to run independently even if the main thread ends. However, they may become orphaned processes, which will be adopted by the init process (or its equivalent), ensuring they are managed by the operating system.
- **Subordinate Threads**:
  - **Answer**: When the main thread terminates, any subordinate (or worker) threads may also be terminated depending on the thread management model used. In many implementations, if the main thread exits while other threads are still running, the process may terminate entirely, which includes all threads within that process. This behavior may lead to abrupt termination of any ongoing work performed by the threads, potentially resulting in resource leaks or incomplete operations.

4. Examples of When Multithreading is Beneficial

b. Five Examples When Multithreading is Beneficial:

1. **Web Servers**:
   - Web servers handle multiple client requests simultaneously. Each request can be

processed in a separate thread, allowing for improved responsiveness and reduced latency.

2. **Responsive User Interfaces**:
    - In applications with graphical user interfaces (GUIs), multithreading can keep the interface responsive while performing background tasks (like file downloads or data processing) in separate threads.
3. **Parallel Processing**:
    - Multithreading is useful in computationally intensive tasks where a problem can be divided into smaller sub-problems that can be processed simultaneously, such as image processing or scientific simulations.
4. **I/O Bound Applications**:
    - Applications that spend a significant amount of time waiting for I/O operations (like file reading/writing or network communication) can benefit from multithreading, as one thread can handle I/O while others continue to perform computations.
5. **Real-time Data Processing**:
    - In systems requiring real-time data processing (like video streaming or real-time analytics), threads can be used to handle incoming data streams while simultaneously processing and displaying the data to users.

OS All Assignment Report

Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Lab Test - 25.5

1.1

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include<sys/wait.h>

void addition(int a, int b) {
    printf("Child Process (Addition): %d + %d = %d\n", a, b, a + b);
}
void subtraction(int a, int b) {
    printf("Child Process (Subtraction): %d - %d = %d\n", a, b, a - b);
}
void multiplication(int a, int b) {
    printf("Child Process (Multiplication): %d * %d = %d\n", a, b, a * b);
}
int main() {
    pid_t pid1, pid2, pid3;
    int a = 10, b = 5;

    pid1 = fork();
    if (pid1 == 0) {
        printf("Child 1 pid is %d\n",getpid());
        addition(a, b);
    }
    pid2 = fork();
    if (pid2 == 0) {
        printf("Child 2 pid is %d\n",getpid());
        subtraction(a, b);
```

```
    }
    pid3 = fork();
    if (pid3 == 0) {
        printf("Child 3 pid is %d\n",getpid());
        multiplication(a, b);
    }
    if (pid1 > 0 && pid2 > 0 && pid3 > 0) {
        wait(NULL);
        wait(NULL);
        wait(NULL);

        printf("Parent Process (Parent Process ID: %d)\n", getpid());
    }

    return 0;
}
```

1.1.1 Create a orphan process

```
#include<unistd.h>
#include<sys/wait.h>
#include<stdio.h>

int main(){
    pid_t c = fork();

    if(c==0){
        sleep(20);
        printf("Child Process\n Child PID : %d\n", getpid());
    }
    else{
        // wait(NULL);
        sleep(10);
        printf("Parent process\n Parent PID : %d\n", getpid());
    }
}
```

1.1.2 Create a Zoombie process

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

```c
#include <sys/wait.h>

int main(){
    pid_t c = fork();
    if(c==0){
        printf("Child process pid %d\n", getpid());
    }
    else{
        sleep(10);
        printf("Parent process %d\n",getpid());
    }
}
```

1.2 Probelem - 3

Write a C program to create a main process named 'parentprocess' having 3 child processes without any grandchildren processes. Child Processes' names are child1, child2, child3. Trace the position in the process tree.

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main(){
    pid_t child_1 = fork();
    if(child_1 == 0){
        sleep(5);
        printf("Child 1 pid is %d\n",getpid());
    }
    pid_t child_2 = fork();
    if(child_2 == 0){
        sleep(10);
        printf("Child 2 pid is %d\n",getpid());
    }
    pid_t child_3 = fork();
    if(child_3 == 0){
        sleep(15);
        printf("Child 3 pid is %d\n",getpid());
    }
    else{
        sleep(20);
        printf("Parent pid is %d\n",getpid());
    }
```

```
}
```

The output of the the process in the pstree is given below

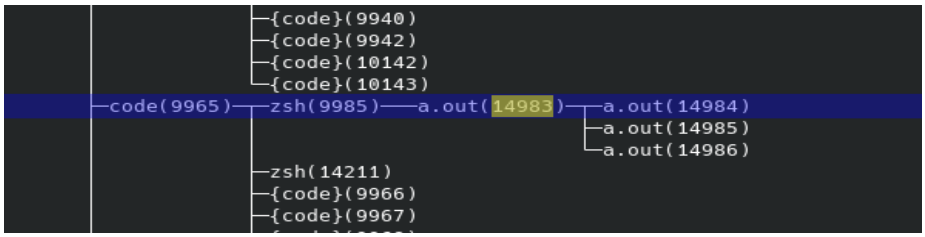


Figure 1: Parent Child in pstree

1.2.1 Problem - 4

Write a C program to create a main process named 'parentprocess' having 'n' child processes without any grandchildren processes. Child Processes' names are child1, child2, child3,....., childn. Trace the position in the process tree. Number of child processes (n) and name of child processes will be given in the CLI of Linux based systems.
Example:
./parentprocess 3 child1 child2 child3
Hint: fork, exec, fopen, system

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
void create_child_process(const char *child_name) {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }
    if (pid == 0) {
        char *args[] = {"sh", "-c", "echo $PPID", NULL};
        FILE *file = fopen("process_log.txt", "a");
        if (file != NULL) {
            fprintf(file, "Child process %s with PID %d created.\n", child_name, getpid()
            fclose(file);
```

```
        }
        printf("%s (PID: %d) is created.\n", child_name, getpid());
        sleep(5);

        exit(0);
    }
}
int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <number of children> <child names>\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    if (n <= 0 || argc != n + 2) {
        printf("Invalid number of child processes or incorrect arguments.\n");
        return 1;
    }
    printf("Parent process PID: %d\n", getpid());
    for (int i = 2; i < argc; i++) {
        create_child_process(argv[i]);
    }
    for (int i = 2; i < argc; i++) {
        sleep(5);
    }
    printf("\nTracing process tree:\n");
    system("pstree -p");
    return 0;
}
```

The output of the the process in the pstree is given below

```
├─gnome-keyring-d(2239)─┬─{gnome-keyring-d}(2243)
│                       ├─{gnome-keyring-d}(2244)
│                       ├─{gnome-keyring-d}(2245)
│                       └─{gnome-keyring-d}(2247)
├─gnome-terminal-(16282)─┬─zsh(16293)───parent_process(16984)─┬─parent_process(16985)
│                        │                                   ├─parent_process(16986)
│                        │                                   ├─parent_process(16987)
│                        │                                   └─sh(17008)───pstree(17009)
│                        ├─{gnome-terminal-}(16283)
│                        ├─{gnome-terminal-}(16284)
```

Figure 2: Parent Child in pstree

2

Monday, October 21, 2024 09:44 PM

1. a. Can you kill processes having PID 0 and PID 1?

- **PID 0 (Swapper Process)**: You cannot kill the process with PID 0 because it's a kernel process responsible for system idle time.
- **PID 1 (Init/Systemd Process)**: Killing the process with PID 1 is possible (but extremely dangerous) because it's the parent of all other processes. Doing so can crash or hang the system. Only the root user can attempt this using:

```
sudo kill -9 1
```

2. Consider the following code segment:

```
pid_t pid;
pid = fork();
if (pid == 0) {
  fork();
  pthread_create(&tid, NULL, (void)thread_handler, NULL);
}
fork();
```

A. How many unique processes will be created?

- After analyzing the forks:
  - Total of **5 unique processes** will be created: 1 original process and 4 child processes.

B. How many unique threads will be created?

- **1 unique thread** will be created using pthread_create in the child process.

1. b. What will be the output for the following code segment?

```
fork();
printf('Bangladesh');
fork();
printf('Bangladesh');
fork();
```

Output Analysis:

1. **First fork()**: Creates one child process.
   - Total Processes: 2 (Parent and Child)
2. **First printf("Bangladesh")**: This will be executed by both processes, producing the output "Bangladesh" twice.
3. **Second fork()**: Each of the 2 processes will now create a child process.
   - Total Processes: 4 (2 existing + 2 new)
4. **Second printf("Bangladesh")**: This will be executed by all 4 processes, producing the output "Bangladesh" four more times.
5. **Third fork()**: Each of the 4 processes will create another child process.
   - Total Processes: 8 (4 existing + 4 new)

Total Output:

- The string "Bangladesh" will be printed **8 times** in total, but the order is not guaranteed due to concurrent execution.

Summary of Output:

- **Total prints of "Bangladesh"**: 8

1. c. How many child processes will be created if the following loop is in a program?

```
for (i = 0; i < n; i++)
  fork();
```

Explanation:

- The loop runs n times, and each iteration of fork() creates a new process. Each existing process can also create additional processes in the subsequent iterations.
- The number of child processes created follows a pattern of exponential growth. Specifically, for each iteration of the loop, the number of processes doubles:

1. **Iteration 0 (i = 0)**: 1 process creates 1 new child → 2 total processes.
2. **Iteration 1 (i = 1)**: 2 processes (existing) create 2 new children → 4 total processes.
3. **Iteration 2 (i = 2)**: 4 processes create 4 new children → 8 total processes.
4. **Iteration 3 (i = 3)**: 8 processes create 8 new children → 16 total processes.
5. ...
6. **Iteration n-1 (i = n-1)**: The total number of processes is 2n2^n2n.

Total Child Processes:

- The total number of child processes created will be 2n−12^n - 12n−1 (the original process is not counted as a child).

Process Tree for n = 4:

```
      P0
     /  \
    P1   P2
   /  \  /  \
  P3  P4 P5  P6
```

- **Level 0**: P0 (original)
- **Level 1**: P1 and P2 (2 processes from 1 fork)
- **Level 2**: P3 and P4 (forked from P1) and P5 and P6 (forked from P2)

Summary for n = 4:

- **Total Child Processes Created**: 2^4 - 1 =15 child processes.
- **Total Processes (including the original)**: 2^4 = 16.

2. a. Give two reasons why caches are useful.

1. **Speed Improvement**: Caches store frequently accessed data closer to the CPU, significantly reducing the time it takes to retrieve that data compared to accessing slower memory (like RAM or disk).
2. **Bandwidth Reduction**: By serving data from the cache instead of fetching it from slower storage, caches help reduce the overall data traffic on the memory bus and other communication pathways, improving system efficiency.

2. b. What problems do they solve?

1. **Latency Reduction**: Caches solve the problem of high latency associated with accessing slower storage devices, allowing for faster data retrieval and improved application performance.
2. **Data Locality Exploitation**: Caches take advantage of the principle of temporal and spatial locality, where programs tend to access the same data or nearby data locations repeatedly, enhancing overall system efficiency.

2. c. What problems do they cause?

1. **Cache Coherency Issues**: In multiprocessor systems, maintaining consistency between multiple caches can lead to complexity, as changes in one cache must be reflected across others, potentially leading to stale data.
2. **Cache Pollution**: Caches can become filled with less useful data (cache pollution), displacing

frequently used data and leading to decreased performance. This happens when the cache fills up with data that isn't accessed frequently.

2. d. If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

1. **Cost**: High-speed memory (like SRAM used in caches) is significantly more expensive to produce than slower memory (like HDDs or SSDs). Making a cache the size of a disk would be prohibitively expensive.
2. **Power Consumption**: Large caches would consume more power, which can be critical in portable or battery-operated devices. Smaller, efficient caches provide a balance between performance and power usage.
3. **Diminishing Returns**: Beyond a certain size, increasing cache size yields diminishing returns on performance, as the likelihood of cache hits decreases relative to access patterns, making larger caches less efficient.
4. **Complexity**: Larger caches would introduce additional complexity in terms of management, organization, and replacement policies, which could lead to increased overhead and latency.

Thus, while larger caches might theoretically improve performance, practical limitations in cost, power, and efficiency prevent them from entirely replacing slower storage devices.

3. a. What do you know about:

Orphan Process

An **orphan process** is a process that continues to run after its parent process has terminated. In Unix-like operating systems, when a parent process exits before its child processes, those child processes become orphaned.

- **How it occurs**: When a parent process ends unexpectedly (e.g., crashes or is killed), its child processes do not automatically terminate. Instead, the child processes are re-assigned to the **init process** (PID 1), which becomes their new parent. This ensures that orphan processes can still run and be managed by the system.
- **Impact**: Orphan processes are generally not a problem as long as the system manages them properly. The init process can clean them up, ensuring that system resources are not wasted.

Zombie Process

A **zombie process** (or defunct process) is a process that has completed execution but still has an entry in the process table. This occurs when a process has finished running but its parent process has not yet read its exit status.

- **How it occurs**: When a child process terminates, it sends a signal (SIGCHLD) to its parent process, indicating that it has exited. The parent must then call wait() or waitpid() to read the exit status. If the parent process fails to do this, the child process remains in a "zombie" state.
- **Impact**: Zombie processes occupy an entry in the process table, which can lead to resource exhaustion if many zombies accumulate. However, they do not consume CPU or memory resources actively. Once the parent process reads the exit status, the zombie process is removed from the process table.

Summary

- **Orphan Process**: A child process that continues running after its parent has terminated, typically re-parented to the init process.
- **Zombie Process**: A terminated process that still has an entry in the process table because its parent has not yet acknowledged its termination.

3. b. When can you call a process an orphan process or a zombie

process?

- **Orphan Process**: A process is called an orphan process when its parent process has terminated or exited, leaving it without a parent. In this situation, the orphaned process is usually adopted by the init process (PID 1) to ensure it can continue executing and be properly managed by the operating system.
- **Zombie Process**: A process is called a zombie process when it has completed execution but still has an entry in the process table because its parent process has not yet read its exit status. This happens when the child process terminates, but the parent fails to call wait() or waitpid() to collect the child's exit status.

.

3. c. Can you create them? If yes, describe your steps with code.

- Yes, both can be created.
  - **Zombie Process**: You can create a zombie by having the child exit without the parent calling wait():

```
if (fork() == 0) {
   exit(0);  // Child exits immediately, becomes a zombie
} else {
   sleep(100);  // Parent waits without calling wait()
}
```

Orphan Process: You can create an orphan by making the parent exit before the child finishes:

```
if (fork() == 0) {
   sleep(10);  // Child runs for a while, becomes orphan
   printf("I am orphaned\n");
} else {
   exit(0);  // Parent exits
}
```

3. d. How can we trace an orphan and zombie process in a Linux-based OS?

- **Orphan Process**: Use ps -ef | grep init to check if the process is adopted by init (PID 1).
- **Zombie Process**: Use ps aux | grep Z to list zombie processes. They appear with a "Z" status.

3. e. Explain how a Linux-based OS treats them.

- **Orphan Process**: The init process adopts orphan processes, ensuring they are managed and eventually terminated properly.
- **Zombie Process**: The OS keeps a zombie's entry in the process table until the parent calls wait() to collect its exit status.

3. f. What are the advantages and disadvantages of having orphan and zombie processes?

- **Advantages**:
  - **Orphan**: The system can still ensure resource cleanup as init adopts them.
  - **Zombie**: Retains information about the process's exit status until the parent retrieves it.
- **Disadvantages**:
  - **Orphan**: May consume resources unnecessarily if not managed by the parent process.
  - **Zombie**: Can lead to resource leakage if not cleared, as they still occupy process table entries.

4. What happens when you interact with an "Addition" executable file:

A. A single user double-clicks on its file icon once:

- **Program vs. Process**: The **program** "Addition" is loaded into memory by the OS, becoming a **process**.

- **Memory Layout**: The OS allocates memory for the process in the form of text (code), data, heap, and stack segments.
- **OS Loader**: The OS loader loads the executable into memory, initializing it as a process, setting up its memory layout, and starting execution at the entry point (main function).

B. A single user double-clicks on its file icon 'n' times:

- **Program vs. Process**: Each double-click creates a separate **process** from the same program. Thus, 'n' clicks create 'n' **processes** of the "Addition" executable.
- **Memory Layout**: Each process will have its own memory layout (text, data, heap, and stack), separate from the others.
- **OS Loader**: For each double-click, the OS loader creates a new instance of the process by loading the executable into memory for each click.

C. Multiple users double-click on its file icon at the same time:

- **Program vs. Process**: Each user creates a separate **process**. If multiple users double-click the icon, separate processes will be created for each user.
- **Memory Layout**: Each user's process will have its own memory layout. There will be no sharing of memory space between the processes unless they are explicitly designed to do so (e.g., through shared memory).
- **OS Loader**: The OS loader will handle each user request, loading the "Addition" program into memory for each process, independent of other users.

D. A single user presses ENTER after typing ./Addition in a terminal:

- **Program vs. Process**: The **program** is executed as a single **process**, similar to the double-click scenario, but triggered through the terminal.
- **Memory Layout**: The memory layout (text, data, heap, stack) is allocated for the process by the OS, as usual.
- **OS Loader**: The OS loader loads the "Addition" program into memory, initializes the process, and starts execution. Since it is launched from the terminal, you can see the output and interact with the process in the terminal window.

Summary:

- Each interaction with the executable (whether single or multiple clicks or commands) results in the creation of separate processes, each with its own memory layout.
- The OS loader plays a key role in loading the executable into memory, setting up the memory space, and managing the process lifecycle.

5. Do we need an operating system for all the cases? Justify your answer:

A. A single user wants to do a single task using a computer system having a finite set of computer hardware:

- **Answer**: **No**, an operating system is not strictly required.
- **Justification**: If the user is running a single program that directly interacts with hardware, an OS is unnecessary. Embedded systems or microcontroller-based systems often run a single task without a full-fledged OS.

B. A single user wants to do multiple tasks simultaneously using a computer system having a finite set of computer hardware:

- **Answer**: **Yes**, an operating system is needed.
- **Justification**: An OS is required to handle **multitasking** and to manage resources like CPU, memory, and I/O devices efficiently. The OS provides process scheduling, memory management, and task switching, allowing multiple tasks to run simultaneously.

C. A single user wants to do multiple tasks consecutively using a computer system having a finite set of computer hardware:

- **Answer**: **Yes**, an operating system is needed.
- **Justification**: Even though the tasks are executed one after another, the OS is needed to manage **task switching** and ensure efficient resource allocation, memory management, and file system

management. The OS coordinates the execution of consecutive tasks.

D. A single user wants to do a single task using a single hardware:

- **Answer**: **No**, an operating system is not strictly required.
- **Justification**: Similar to embedded systems, when only a single task is needed, the program can directly interact with the hardware without an OS. A real-time system or firmware may run the task without the overhead of a general-purpose OS.

E. Multiple users want to do multiple tasks simultaneously using a computer system having a finite set of computer hardware:

- **Answer**: **Yes**, an operating system is essential.
- **Justification**: In this case, an OS is required for **multi-user** support and **multi-tasking**. It manages users, schedules their tasks, isolates processes, handles permissions, and allocates system resources fairly and efficiently to ensure simultaneous operation without interference between users.

6. a. What can be done to increase the degree of multiprogramming?

- **Increase the number of processes** in memory by:
  1. **Adding more RAM**: This allows the OS to load more processes into memory simultaneously.
  2. **Using virtual memory**: The OS can use disk space as an extension of RAM, allowing more processes to reside in memory.
  3. **Efficient memory management**: Techniques like paging and segmentation can optimize memory usage and allow more processes to run concurrently.

6. b. Journey of a source code (Addition.c) to turn into a process:

1. **Source Code (Addition.c)**:
   - The source code is written in C, stored in a file.
2. **Compilation**:
   - The C compiler (e.g., gcc) translates the source code into machine-readable **object code** (Addition.o).
   - **Compilation stages**:
     1. **Preprocessing**: Expands macros and includes header files.
     2. **Compilation**: Converts C code to assembly code.
     3. **Assembly**: Translates assembly code into object code.
     4. **Linking**: Links object code with libraries to create an **executable file** (Addition).
3. **Execution**:
   - When the executable is run, the **OS loader** loads it into memory.
   - The loader sets up the **process memory layout** (text, data, heap, stack).
4. **Process Execution**:
   - The OS schedules the process for execution by assigning it CPU time.

Figure:

Source Code (Addition.c)
↓
Preprocessing
↓
Compilation
↓
Assembly
↓
Linking
↓
Executable (Addition)
↓
OS Loader (loads executable into memory)
↓
Process Execution (process gets CPU time)

6. c. Positive and Negative Sides of Linux OS Architecture vs. UNIX OS Architecture:

Positive Sides:

1. **Open Source**: Linux is open-source, allowing for more community-driven development and flexibility compared to UNIX, which is traditionally proprietary.
2. **Modularity**: Linux has a more modular architecture, making it easier to customize and optimize for specific use cases.
3. **Wider hardware support**: Linux supports a broad range of hardware, while UNIX systems tend to be more limited to specific hardware platforms.
4. **Modern features**: Linux often includes modern features like better networking support, more file systems, and containerization (e.g., Docker).

Negative Sides:

1. **Fragmentation**: The wide variety of Linux distributions can lead to fragmentation and compatibility issues.
2. **Less standardized**: While UNIX systems follow strict standards, Linux distributions can differ significantly, leading to inconsistencies.
3. **Potential instability**: Due to frequent updates and community-driven development, some Linux distributions may be less stable than traditional UNIX systems.

6. d. Memory Layout for the C Program:

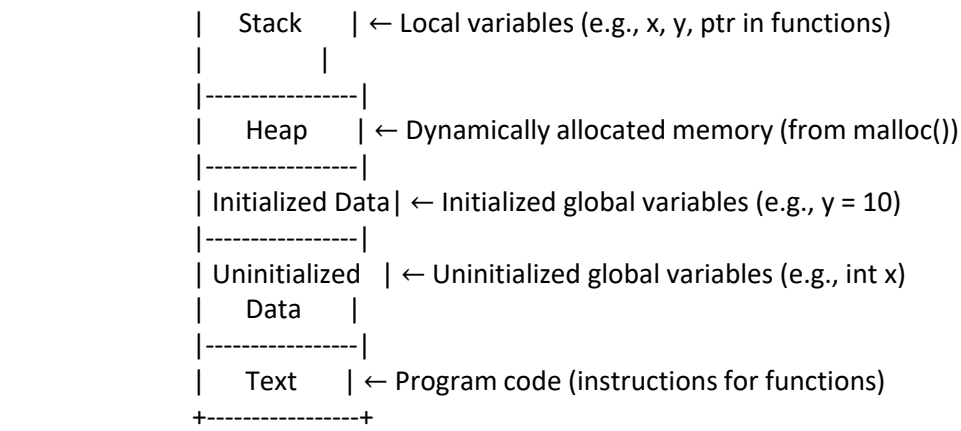
```
#include <stdio.h>
#include <stdlib.h>
int x, y = 10;
void f2(){
   int x, y = 10;
   int *ptr;
   ptr = (int *) malloc(sizeof(int) * 5);
}
void f1(){
   int x, y = 10;
   int *ptr;
   ptr = (int *) malloc(sizeof(int) * 5);
   f2();
}
int main(int argc, char *argv[]){
   int x, y = 10;
   int *ptr;
   ptr = (int *) malloc(sizeof(int) * 5);
   return 0;
}
```

Memory Layout:

1. **Text Segment**:
   - Contains the compiled code (instructions) for main(), f1(), f2(), and other functions.
2. **Data Segment**:
   - **Initialized data**: Variables with values defined at compile time (e.g., y = 10 in global scope).
   - **Uninitialized data (BSS)**: Global variables like x that are not explicitly initialized.
3. **Heap Segment**:
   - Dynamically allocated memory for the malloc() calls in main(), f1(), and f2().
4. **Stack Segment**:
   - Stores local variables for each function (x, y, ptr in main(), f1(), f2()).
   - Each function call creates a stack frame containing local variables and return addresses.

Memory Layout Structure:

```
+-----------------+
```



7. a. How many processes can be in a running state, waiting state, and ready state at a time?

- **Running State**: **1 process** can be in the running state on a single-core CPU. On a multi-core CPU, the number of running processes equals the number of cores.
- **Waiting State**: There can be **many processes** in the waiting state, as processes wait for I/O or other events.
- **Ready State**: There can be **many processes** in the ready state, waiting for CPU time to execute.

7. b. Who can have access to the Process Control Block (PCB) of a process?

- **Answer**: **The operating system (OS)** has access to the PCB.
- **Justification**: The OS needs to control process management, scheduling, resource allocation, and tracking the state of processes. Direct access by user programs could lead to security risks and system instability.

7. c. Where are PCBs stored?

- **Answer**: PCBs are stored in the **kernel space** of memory.
- **Justification**: Kernel space is protected from user access, ensuring that only the OS can modify or access PCBs. This prevents user-level programs from corrupting critical process information.

7. d. Without PCBs, is it possible to handle multiple processes?

- **Answer**: **No**, it is not possible.
- **Justification**: The PCB is essential for the OS to manage multiple processes. It stores the state of each process (e.g., program counter, memory allocation, I/O status), allowing context switching and scheduling. Without PCBs, the OS would not be able to track or switch between processes.

8. a. Write down the difference between:

i. Unnamed Pipe vs. Named Pipe:

- **Unnamed Pipe**:
  - Used for communication between **related processes** (parent-child processes).
  - Exists only during the lifetime of the communicating processes.
- **Named Pipe (FIFO)**:
  - Used for communication between **unrelated processes**.
  - Can persist beyond the lifetime of the processes and is identified by a name in the file system.

ii. Named Pipe vs. Regular File:

- **Named Pipe (FIFO)**:
  - Provides **inter-process communication** (IPC) by transferring data between processes.
  - Data flows in a **one-way** or **two-way** communication stream.
- **Regular File**:
  - Used for **data storage** on disk.
  - Data is stored persistently and can be accessed randomly (not limited to sequential access

like a pipe).

iii. Program vs. Process:

- **Program**:
  - A **passive** entity, a set of instructions stored in a file.
  - Does not execute until loaded into memory and initiated.
- **Process**:
  - An **active** entity, an instance of a program in execution.
  - Consumes system resources (CPU, memory) and has a state (e.g., running, waiting).

iv. User Mode vs. Kernel Mode:

- **User Mode**:
  - The processor operates with **restricted access** to system resources. It is used for executing user applications.
  - System calls are required to access hardware and sensitive operations.
- **Kernel Mode**:
  - The processor operates with **full access** to system resources and hardware.
  - The OS runs in kernel mode to perform privileged operations like managing memory, processes, and I/O devices.

8. b. What happens when a child process updates its local and global variables?

- **Local Variables**: The child process has its **own copy** of local variables, which are created during the fork. Any changes to local variables in the child process **do not affect** the parent process, as the local variables exist in the child's separate memory space.
- **Global Variables**: Similarly, the child process gets its **own copy** of global variables when the fork occurs. Any updates to global variables in the child process **do not affect** the parent process. Both parent and child have independent copies of global variables due to the memory separation after fork().

In summary, after a fork, changes in the child process's local or global variables **do not impact** the parent process due to the process memory isolation.

8. c. How many ways can two processes work on shared data?

i. Child-Parent Relationship:

1. **Shared Memory**:
   - The parent and child can use **shared memory** to work on the same data. Shared memory allows multiple processes to access the same memory region.
   - Requires explicit setup using system calls like shmget, shmat, and shmdt.
2. **Pipes (Unnamed Pipes)**:
   - The parent and child can communicate via **pipes**, where data is passed from one process to another.
   - Pipes provide one-way or two-way communication but do not share the same memory directly.

ii. Non Child-Parent Relationship:

1. **Named Pipes (FIFOs)**:
   - Two unrelated processes can communicate via **named pipes** (FIFOs), which allow them to read/write data in a common stream.
2. **Message Queues**:
   - Two processes can use **message queues** for exchanging messages without sharing memory directly. This is a form of IPC where messages are passed between processes using system calls like msgget and msgsnd.
3. **Shared Memory**:
   - Even unrelated processes can use **shared memory** for working on the same data. Shared memory segments must be explicitly created and accessed using the appropriate system

calls.

4. **Sockets**:
    - **Sockets** can be used for communication between processes, especially when the processes are running on different machines. Even on the same machine, they can communicate using local sockets (IPC sockets).

8. d. Code for inter-process communication between two terminals (one counting down from n, one terminating on Bye)

The following example uses **named pipes (FIFO)** for inter-process communication. Two terminals will use a named pipe to communicate: one terminal sends a number to the other, which counts down from n to 1. When "Bye" is entered, the receiving process terminates.

1. **Terminal 1 (Sender)**:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
   char input[100];
   int fd;

   // Create named pipe
   mkfifo("/tmp/myfifo", 0666);

   while (1) {
      // Open FIFO in write mode
      fd = open("/tmp/myfifo", O_WRONLY);

      // Get input from the user
      printf("Enter a number or 'Bye': ");
      fgets(input, 100, stdin);
      input[strcspn(input, "\n")] = '\0'; // Remove newline character

      // Write input to the FIFO
      write(fd, input, strlen(input) + 1);
      close(fd);

      // If input is "Bye", exit loop
      if (strcmp(input, "Bye") == 0) {
         break;
      }
   }

   // Remove the named pipe
   unlink("/tmp/myfifo");
   return 0;
}
```

Terminal 2 (Receiver):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <unistd.h>
#include <fcntl.h>

int main() {
   char input[100];
   int fd, num;

   // Create named pipe (same path as sender)
   mkfifo("/tmp/myfifo", 0666);

   while (1) {
      // Open FIFO in read mode
      fd = open("/tmp/myfifo", O_RDONLY);

      // Read input from the FIFO
      read(fd, input, sizeof(input));
      close(fd);

      // Check if input is "Bye"
      if (strcmp(input, "Bye") == 0) {
         printf("Terminating...\n");
         break;
      }

      // Convert input to integer
      num = atoi(input);

      // Print countdown from n to 1
      for (int i = num; i > 0; i--) {
         printf("%d\n", i);
         sleep(1);  // Adding sleep to simulate delay
      }
   }

   // Remove the named pipe
   unlink("/tmp/myfifo");
   return 0;
}
```

Steps:

1. Run the **sender** program in one terminal.
2. Run the **receiver** program in a separate terminal.
3. The sender can send numbers to the receiver, which will count down.
4. Sending "Bye" from the sender will terminate the receiver process.

Explanation:

- **Named Pipe (FIFO)** is used to communicate between two unrelated processes.
- **Terminal 1** (sender) writes data (numbers or "Bye") to the FIFO.
- **Terminal 2** (receiver) reads from the FIFO and performs actions based on the input (n for countdown, "Bye" for termination).

9. a. What happens when two processes want to do read and write operations through a pipe in opposite order?

- **Deadlock Possibility**: If both processes attempt to read and write through the pipe simultaneously

in opposite order (one tries to read first while the other tries to write first), this can lead to a **deadlock**. Both processes might wait indefinitely because each is expecting the other to either produce or consume data.

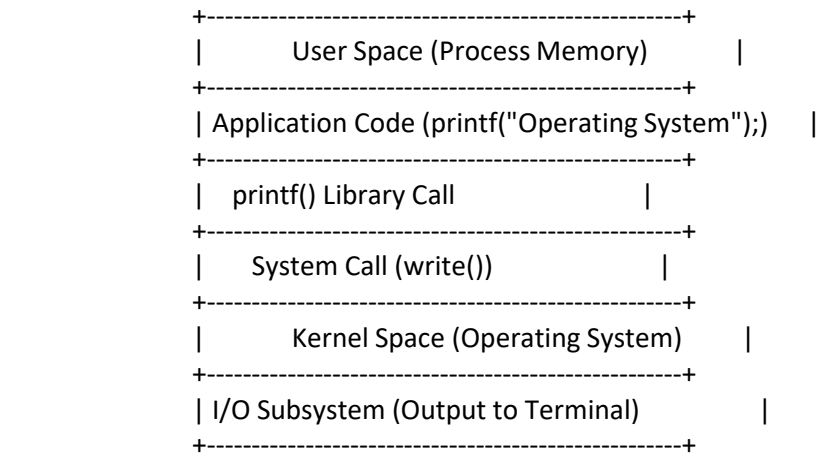
- Example:
  - Process A: waiting to **read** from the pipe, but there's no data because Process B hasn't written yet.
  - Process B: waiting to **write** to the pipe, but the pipe might be full because Process A hasn't read any data yet.

9. b. Explain with a figure what happens if there is an instruction printf("Operating System"); in a C program.

When a printf() call is made, the following steps happen:

1. **System Call Invocation**:
   - printf() is a library function that internally invokes a system call like write() to send data to the standard output (usually the terminal).
2. **Data Handling**:
   - The string "Operating System" is passed to the write() system call, which interacts with the OS.
3. **OS Interaction**:
   - The OS kernel handles the system call, writing the string to the output device (screen).

Figure:



9. c. Does the CPU remain idle when context switching happens? Justify your answer.

- **Yes, the CPU is temporarily idle during context switching**.
  - **Justification**: During a context switch, the CPU is not performing any user-level computation. Instead, it is saving the current process's state (e.g., registers, program counter) and loading the next process's state. This is managed by the OS and involves kernel-level operations. However, these operations are short, so the CPU is only idle for a minimal amount of time during the switch.

9. d. What can the operating system do to keep the CPU busy all the time?

- **Multiprogramming**: The OS can keep multiple processes in memory, allowing the CPU to switch to another ready process whenever the current one is waiting for I/O or blocked.
- **Context Switching**: The OS can quickly switch between processes that are ready to execute, ensuring that the CPU never remains idle.
- **Scheduling Algorithms**: Efficient scheduling algorithms (e.g., Round Robin, Shortest Job First) can

be used to maximize CPU utilization, ensuring that whenever a process is blocked or waiting, another process is scheduled to run.
- **Load Balancing**: In multi-core systems, the OS can balance the workload between multiple cores, ensuring that all CPU cores are kept busy with processes.

10. a. What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?

The main difficulty in writing an operating system for a real-time environment is **ensuring deterministic behavior**. This includes:

1. **Predictable Response Times**: The OS must guarantee that tasks are executed within strict time constraints, which requires precise scheduling.
2. **Concurrency Control**: Managing multiple tasks that may need to access shared resources without causing delays or violating deadlines.
3. **Resource Allocation**: Ensuring that critical resources are available when needed without causing starvation of real-time tasks.
4. **Latency Minimization**: Reducing delays caused by context switching, interrupts, and I/O operations.
5. **Testing and Validation**: Rigorous testing is required to ensure that timing constraints are met under all conditions, making debugging more challenging.

10. b. Is it possible to construct a secure operating system for computer systems without a privileged mode of operation?

Arguments for Possibility:

1. **Software-Based Security**: It is possible to implement security measures purely in software, such as user access control, process isolation, and sandboxing techniques.
2. **Minimal Trust Environment**: By designing the system to operate in a minimal trust environment and leveraging user-level checks, security can still be maintained.
3. **Controlled Execution**: Implementing strict protocols for inter-process communication and resource access can mitigate some risks.

Arguments Against Possibility:

1. **Lack of Protection**: Without a privileged mode, the OS cannot enforce separation between user processes and critical system resources, making it difficult to prevent malicious or erroneous code from affecting the system's integrity.
2. **Vulnerability to Attacks**: Without hardware-level enforcement of security policies, it becomes easier for malicious software to exploit vulnerabilities, leading to system compromises.
3. **Resource Management**: Effective resource management is hindered, as there are no privileged operations to control access to sensitive resources, leading to potential security breaches.

10. c. What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?

Purpose of Interrupts:

- **Event Handling**: Interrupts are used to signal the CPU that an external event needs attention, allowing the OS to respond to I/O requests, timers, or hardware failures without continuous polling.
- **Asynchronous Processing**: They allow asynchronous execution of tasks, enabling more efficient CPU usage by allowing the OS to process high-priority tasks when events occur.

Difference Between Interrupts and Traps:

- **Interrupts**: Generated by external hardware events (e.g., keyboard input, mouse movement) and are handled asynchronously. They can interrupt the current execution of a program and require immediate attention.

- **Traps**: Generated by the CPU in response to certain conditions during program execution (e.g., division by zero, invalid memory access) and are synchronous. They occur as a direct result of executing a particular instruction and typically indicate an error or exception condition.

Can Traps be Generated Intentionally by a User Program?

- **Yes**, traps can be intentionally generated by user programs, often through:
    - **System Calls**: Programs can invoke traps to request services from the OS, such as file operations or network communication.
    - **Debugging**: Developers can use traps to trigger breakpoints or exceptions for debugging purposes, allowing them to inspect the program state when certain conditions are met.

In summary, while interrupts and traps both signal the CPU to perform certain actions, they arise from different sources and serve different purposes in the execution flow.

OS Final Assignment

MD: Miju Ahmed
ID: 2010676104

March 2024

1 What do you know about

1.0.1 Process

A process is a program in execution and a process is an active entity. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is executing.

Again a process is also the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code.

A program by itself is not a process. A program becomes a process when an executable file is loaded into memory.

1.0.2 Process Control Block(PCB)

Each process is represented in the operating system by a process control block (PCB)—also called a task control block.It contains many pieces of information associated with a specific process,including these:

• Process state: The state may be new, ready, running, waiting, halted, and so on.

• Program counter: The counter indicates the address of the next instruction to be executed for this process.

• CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.

• CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

• Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

• I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

1.0.3 PCB Table

Each process is represented in the operating system by a process control block(PCB) also called a task control block. Below showing the figure of the PCB table. It contains Process state, Program counter, CPU register, CPU scheduling information, Memory management information, Accounting information and I/O status information.

Process State
Process Number
Program Counter
Registers
Memory limits
List of open files
. . .

Table 1: PCB Table

1.0.4 Process ID(PID)

Process ID is a unique numerical identifier assigned by the operating system to each process that is running on the system. These IDs are used to track and manage processes, allowing the operating system to differentiate between them and perform various operations, such as resource allocation, scheduling, and termination.

1.0.5 Process Tree

A process tree is a hierarchical representation of processes in an operating system. It visualizes the relationships between parent and child processes, showing how processes are created and organized.

1.0.6 Processes having PID 0 and PID 1

Process having PID 0 and PID 1. But those having different significant in most operating system. PID 0 represents the kernel's idle process, while PID 1 typically represents the init process, responsible for system initialization and management. Below given the some description about PID 0 and PID 1.

PID 0 is used in Unix like Operating system such as Linux. This is reserved for the swapper or scheduler process. It also known as idle process or kernel process. init process in Unix like operating systems. PID 0 is typically a system process and it runs in kernel mode and manages system tasks when no other processes are running.

On the other hand, PID 1 is typically assigned to the init process in Unix-like systems. The init process is the first user-space process started by the kernel during the boot process. It remains active throughout the entire lifetime of the system and is crucial for managing system shutdowns and reboots.

1.1 Write down the differences between the following terms:

1.1.1 Difference between Program and Process

The terms "program" and "process" are related but refer to different concepts in computing. Below given the difference between Program and Process

Program	Process
A program is a set of instructions or code written by a programmer to perform a specific task or function.	A process is an active instance of a program that is being executed by the operating system.
It is a passive entity, typically stored on disk as an executable file or source code.	When a program is executed, it becomes a process.
A program is a static entity and does not perform any actions until it is executed by the operating system or another program.	Each process has its own memory space, resources, and state.
Programs can range from simple scripts to complex applications.	Multiple instances of the same program can run simultaneously, each as a separate process, with each process having its own execution context and data.

Table 2: Difference between Program and Process

1.1.2 Difference between Process and Light-Weight Process

Below given the difference between Process and Light-Weight Process

Process	Light-Weight Process
A process is a heavyweight unit of execution in an operating system.	A lightweight process is a concurrency primitive that exists within the context of a traditional process.
It includes the program code, data, and resources required for execution, such as memory space, file descriptors, and environment variables.	LWPs are sometimes referred to as "threads" or "kernel threads" in certain operating systems.
Each process has its own memory space, address space, and resources, isolated from other processes.	LWPs share the same address space and resources within a process.
Processes are managed by the operating system's kernel and typically require significant overhead to create, switch between, and manage.	They are lighter in terms of memory and resource overhead compared to processes.
Inter-process communication (IPC) mechanisms are often required for processes to communicate with each other.	LWPs are managed by the operating system's kernel but have less overhead compared to processes because they share resources with other LWPs within the same process.

Table 3: Difference between Process and Light-Weight Process

1.2 Can you kill processes having PID 0 and PID 1? If yes, write down the steps of killing processes having PID 0 and PID 1.

In most of the Operating systems, we can't directly kill processes with PID 0 and PID 1 using standard methods because processes are critical for the functioning of the system. However, there are certain scenarios where we may need to force shutdown or reboot the system, effectively terminating these processes indirectly. Here are the steps for forcing a shutdown or reboot.

1. If we want to shutdown the system process right now we need to run the command
sudo shutdown now
2. If we restart the system immedietly the we should run
sudo shutdown -r now

3. For reboot the system we should run
sudo systemctl reboot
or
sudo reboot

4. We also can perform some operation by the SysRq Key. Those are given

below,
At first we have the ensure that the SysRq Key is enabled or nor in the kernel by the below command,
echo 1 — sudo tee /proc/sys/kernel/sysrq
Then we can perform SysRq key command
R: Switch the keyboard from raw mode.
E: Send the SIGTERM signal to all processes except init.
I: Send the SIGKILL signal to all processes except init.
S: Sync all mounted filesystems.
U: Remount all mounted filesystems read-only.
B: Immediately reboot the system.

2 Write down the differences between the following terms:

2.1 Difference between Preemptive and Non-Preemptive Scheduling Algorithms

Preemptive and non-preemptive scheduling algorithms are two different approaches used by operating systems to manage the execution of processes in a multi-tasking environment. Below given the difference between Preemptive and Non-Preemptive Scheduling Algorithms

Preemptive	Non-Preemptive
The operating system can interrupt a running process and allocate the CPU to another process.	A running process keeps the CPU until it voluntarily relinquishes it, typically by blocking on I/O or terminating.
The decision to switch processes is made by the operating system based on priorities, time slices, or other criteria.	The operating system doesn't forcibly interrupt the running process. Instead, it waits for the current process to finish its execution or block itself.
Preemptive scheduling often employs priority-based scheduling algorithms, where higher priority processes can preempt lower priority ones.	Priority of processes is only considered when selecting the next process to run, not during execution. Once a process starts, it continues until it completes or blocks.
The Round-Robin scheduling algorithm is a preemptive algorithm.	The FCFS scheduling algorithm is a non-preemptive algorithm.

Table 4: Difference between Preemptive and Non-Preemptive Scheduling Algorithms

2.2 Difference between User mode and kernel mode

User mode and kernel mode are two distinct privilege levels or modes of operation in a computer's CPU. They define the level of access and control that software running on the CPU has over hardware and system resources. Below given the difference between User mode and Kernel mode

User mode	Kernel mode
Restricted access to resources.	Full access to resources
Limited privileges	Full privileges
Executes user-level code	Executes kernel-level code
Provides protection from errors and malicious software	Requires trust and stability for system stability
Most application software runs here	Operating system components run here

Table 5: Difference between User mode and Kernel mode

2.3 Difference between Named pipe and Unnamed pipe

Named pipes and unnamed pipes are both mechanisms for inter-process communication (IPC) in Unix-like operating systems, but they have some key differences. Below given the difference between Named pipe and Unnamed pipe

Named pipe	Unnamed pipe
Named pipes have a unique name associated with them, located in the file system. They are created using the mkfifo command or the mkfifo() system call.	Unnamed pipes have no associated name or representation in the file system. They are created using the pipe() system call.
It's allow communication between unrelated processes, even those running on different machines, as long as they have permission to access the named pipe file.	It's allow communication between related processes only, typically between a parent process and its child processes.
It is typically used for long-term communication between processes or for communication between processes started at different times.	It is commonly used for short-lived communication between related processes, often in situations where one process needs to pass data to another process it has spawned.

Table 6: Difference between Named pipe and Unnamed pipe

2.4 Difference between Conventional File and Named Pipe

Below given the difference between Conventional File and Named Pipe

Conventional File	Named Pipe
Conventional files are used for storing data persistently on disk.	Named pipes are used for inter-process communication (IPC) and do not store data persistently. They operate purely in memory.
They are accessed via file system calls like open(), read(), write(), and close().	Named pipes are accessed using file system calls similar to conventional files, but they are unidirectional and cannot be opened for both reading and writing by the same process.
Conventional files have a well-defined structure with a beginning, end, and usually fixed size.	Named pipes have a FIFO (First-In-First-Out) structure. Data written to a named pipe is read in the same order it was written.
They persist even after the processes that created them terminate, until they are explicitly deleted.	Named pipes exist only as long as processes are actively using them. Once all processes close their ends of the pipe, it is removed from memory.
Text files, executable binaries, images, videos, etc are here.	Communication between a producer process generating data and a consumer process reading and processing that data in real-time are here

Table 7: Difference between Conventional File and Named Pipe

2.5 Difference between Logical core and Physical core

Below given the difference between Logical Core and Physical Core

Physical Core	Logical Core
A physical core is a physical processing unit within the CPU.	A logical core, also known as a hardware thread, represents a virtual processing unit within a physical core.
Executes instructions independently.	Shares resources with other logical cores in the same physical core.
Has dedicated resources.	Utilizes simultaneous multithreading (SMT) for concurrent execution.
Provides true parallelism.	Improves multitasking performance.
Limited by the number of physical cores on the CPU.	Increases the number of available threads without adding physical cores.

Table 8: Difference between Logical Core and Physical Core

2.6 Difference between Default Signal Handler and User-Defined Signal Handler

Default signal handlers and user-defined signal handlers are both mechanisms for handling signals in Unix-like operating systems, but they differ in terms of their behavior and how they are implemented. Below given the difference between Default Signal Handler and User-Defined Signal Handler

Default Signal Handler	User-Defined Signal Handler
Default signal handlers are predefined actions taken by the operating system when a signal is received by a process.	User-defined signal handlers are custom functions written by the programmer to handle signals in a specific way.
Default signal handlers are automatically invoked when a signal is received	User-defined signal handlers must be explicitly installed by the programmer.
Programmers have limited control over default signal handlers	Programmers have full control over user-defined signal handlers.
Default handlers have fixed behaviors.	User-defined signal handlers can be customized to perform specific actions based on the requirements of the program.

Table 9: Difference between Default Signal Handler and User-Defined Signal Handler

3 Write down the positive and negative sides of the following scheduling algorithms:

3.1 First Come, First-Served scheduling algorithm:

The First Come, First Served (FCFS) scheduling algorithm is one of the simplest scheduling algorithms used in operating systems. Here are the positive and negative aspects of FCFS:

The positive side of FCFS scheduling algorithm is it can be simple implemented, it also provides fairness among process. This algorithm has no starvation. It has minimal overhead.
The negative side of this algorithm is it may result in poor average turnaround time. It also has inefficiency use of CPU. It does not consider the priority of process.

3.2 Shortest Job First Scheduling Algorithm:

For Shortest Job First Scheduling algorithm also have some positive and negative aspects. Those are given below:

The positive aspects of SJF algorithm is it minimize the average waiting time. It also optimal when all the process burst times are known in advance. This algorithm improves turnaround time and increses CPU utilization.
The negative aspects of SJF algorithm is it requires accurate predictions of the CPU burst time for each process. This information is often not available in practice, making it difficult to implement accurately. It's long jobs can suffer from starvation. It is not suitable for interactive systems.

4 Which scheduling algorithms could result in starvation? How can the OS solve it?

Several scheduling algorithms can potentially lead to starvation, where certain processes are indefinitely delayed or do not receive the CPU time they need to execute. These algorithms include:
1. First-Come, First-Served(FCFS)
2. Shortest Job First(SJF)
3. Priority Scheduling
4. Round Robin(RR) with Large Time Quantum.
To solve the problem of starvation, the operating system can implement various techniques:
1. Aging: Aging involves gradually increasing the priority of waiting processes. As a process waits in the ready queue, its priority is increased over time. This

ensures that every process eventually gets CPU time, even if they have lower priorities.
2. Priority Aging: Similar to aging, priority aging involves gradually increasing the priority of waiting processes. This prevents low-priority processes from being starved by higher-priority ones.
3. Fair Share Scheduling: Implementing fair share scheduling policies that allocate CPU time based on the proportional share of resources each process is entitled to. This ensures that all processes receive their fair share of CPU time.
4. Priority Ceiling Protocol: In priority-based scheduling, the OS can use the Priority Ceiling Protocol to prevent priority inversion, a situation where a low-priority process holds a resource needed by a high-priority process. This protocol helps avoid priority inversion, which can lead to starvation.

5 Write a C program having initialized and uninitialized local and global variables and describe the memory layout of its process.

Below writing the C program demonstrating initialized and uninitialized local and global variables:

```
#include <stdio.h>
int global_initialized_variable = 5;
int global_uninitialized_variable;

int main() {

    int local_initialized_variable = 6;
    int local_uninitialized_variable;
    printf("Global Initialized Variable is :%d\n", global_initialized_variable)
    printf("Global UnInitialized Variable is :%d\n", global_uninitialized_varia
    printf("Local Initialized Variable is :%d\n", local_initialized_variable);
    printf("Local UnInitialized Variable is :%d\n", local_initialized_variable)
    return 0;
}
```

Below describing the memory layout for this program process

Global Initialized Variables are stored in the initialized data segment of the process's memory. Memory is allocated and initialized during program startup. Accessible to all functions within the program.

Global Uninitialized Variables are stored in the uninitialized data segment of the process's memory. Memory is allocated but not initialized during program

startup. Contains default values (typically zero) before initialization. Accessible to all functions within the program.

Local Initialized Variables are stored on the stack. Memory is allocated and initialized when the function is called. Memory is deallocated when the function exits. Accessible only within the scope of the function where it is declared.

Local Uninitialized Variables are stored on the stack. Memory is allocated but not initialized when the function is called. Contains garbage values (values that were previously stored in that memory location). Accessible only within the scope of the function where it is declared.

6 When instructions after execlp() are executed? Why?

The execlp() function is used to replace the current process image with a new one. When execlp() is called, the process that invoked it is replaced by the new process specified in the function call. Therefore, any instructions written after execlp() in the code will not be executed.
The reason is given below
Process Replacement:
When execlp() is called, it loads and executes a new program file in the current process. The new program file replaces the memory image of the current process, including both code and data segments.
Control Transfer:
Control is transferred directly to the new program specified in the execlp() function call. All subsequent instructions in the code after execlp() are replaced by the instructions of the new program.
No Return:
execlp() does not return unless an error occurs. If the function successfully executes, it replaces the current process with the new program, and the current program flow is terminated. If there's an error in executing the new program, execlp() returns, but in this case, the process usually exits, so subsequent instructions are not executed either way.
For example,

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

int main() {
    printf("This line is printed before execlp()\n");
    execlp("ls", "ls", "-l", NULL);
    printf("This line is printed after execlp()\n");
```

```
    return 0;
}
```

For this code,

```
printf("This line is printed after execlp()\n");
```

will not be executed because the process is replaced by the ls command executed by execlp(). Once execlp() is called, the current process is effectively gone, and any subsequent code is no longer part of the executing program.

7 Process Creation

7.1 The code segment

```
pid_t pid;
pid = fork();
if (pid == 0) {
        fork();
        pthread_create(&tid, NULL, (void)thread_handler, NULL);
}
fork();
```

7.1.1 Process number

There are total 7 process is created. Below showing the creation of process:
Initial fork(): 2 processes (parent and child)
Inside the if statement:
First fork(): 2 processes (parent's child and child's child)
pthreadcreate(): 1 thread
Outside the if statement:
Second fork(): 2 processes (parent and child's child)
Total unique processes: $2 + 2 + 1 + 2 = 7$
Therefore, a total of 7 unique processes are created.

7.1.2 Thread number

There are total 1 thread is created. Below showing the creation of thread:
Initial fork(): 0 threads (only processes)
Inside the if statement:
pthreadcreate(): 1 thread
Total unique threads: $0 + 1 = 1$
Therefore, only 1 unique thread is created.

7.2 Code segment:

```
    fork();
    printf("Bangladesh");
----fork();
----printf("Bangladesh");
    fork();
```

The output for this code segment is given below:
Bangladesh
Bangladesh
Bangladesh
Bangladesh
Bangladesh
Bangladesh
Bangladesh
Bangladesh

7.3 Child Process

```
for (i = 0; i < n; i++)
        fork();
```

Below given some explanation how many child processes will be created when the given loop is executed:
Each iteration of the loop creates a new child process by forking the current process. So, for each value of i, the number of child processes created will be doubled. After n iterations, there will be a total of $2^n childprocesses$.

When n = 4, the loop will execute 4 iterations:

1. Iteration 1: Initially, there is one process (the parent process). After the first fork, there are two processes.
2. Iteration 2: Each of the two processes from the previous iteration forks, resulting in a total of four processes.
3. Iteration 3: Each of the four processes from the previous iteration forks again, resulting in a total of eight processes.
4. Iteration 4: Each of the eight processes from the previous iteration forks once more, resulting in a total of sixteen processes.
So, when n = 4, the total number of child processes created will be $2^4 = 16$.

8 Signal

8.1 How can a process send a signal to

8.1.1 Another process

A process can send a signal to another process using the kill() system call or by using other inter-process communication mechanisms.
Using kill() System Call:The kill() system call allows a process to send a signal to another process. Its syntax is as follows:

```
int kill(pid_t pid, int sig);
```

Here 'pid' is the process ID of the target process and 'sig' is the signal to send.

```
#include <signal.h>

int main() {
    kill(12345, SIGTERM);
    return 0;
}
```

We can also use the 'raise()' function. The raise() function allows a process to send a signal to itself. Its syntax is:

```
int raise(int sig);
```

Here the 'sig' is signal to send.

```
#include <signal.h>

int main() {
    raise(SIGINT);
    return 0;
}
```

8.1.2 A specific thread of the same process?

In a multithreaded program, a process can send a signal to a specific thread within the same process using the pthreadkill() function.

8.2 What happens if a parent process and its child process try to kill each other by sending a SIGKILL signal? Consider as many scenarios as possible.

When a parent process and its child process attempt to kill each other by sending a SIGKILL signal (signal number 9), several scenarios can occur, depending

on the timing and order of signal delivery. Let's consider various possibilities:
1. Parent Kills Child First:
a. If the parent process sends a SIGKILL signal to the child process before the child has a chance to act, the child process will immediately terminate without any further action.
2. Child Kills Parent First:
a. If the child process sends a SIGKILL signal to the parent process before the parent has a chance to act, the parent process will immediately terminate without any further action.
3. Simultaneous Kill Attempt:
a. If both processes attempt to send a SIGKILL signal to each other simultaneously, it depends on the race condition (which signal is delivered first).
b. If the parent's signal reaches the child first, the child terminates, and the parent continues executing.
c. If the child's signal reaches the parent first, the parent terminates, and the child becomes an orphan process.
d. The orphan child process will then be adopted by the init process and continue executing, unless the parent process is a session leader and the child process is in a different session, in which case the child will also receive a SIGHUP signal.
4. Child Ignores Signal:
a. If the child process has blocked or ignored the SIGKILL signal, it will not terminate immediately. However, if the parent process terminates, the child will become an orphan process and be adopted by the init process, which will then kill the child process with SIGKILL.
5. Parent and Child Terminate Each Other:
a. In rare cases, if both the parent and child processes successfully send a SIGKILL signal to each other at the exact same moment, they will both terminate simultaneously.
6. Signals Blocked or Ignored:
a. If both processes have blocked or ignored the SIGKILL signal, they will not terminate. However, if the parent process terminates, the child process will become an orphan process and may be terminated by the init process.

8.3 In Linux, how many signals can be caught and cannot be caught by a user-defined signal handler? Make lists of these two categories signals.

In Linux, there are a total of 64 signals. Here's the list of all signals:
1. 'SIGHUP' (1) - Hangup,
2. 'SIGINT' (2) - Interrupt from keyboard (Ctrl+C),
3. 'SIGQUIT' (3) - Quit from keyboard (Ctrl+,
4. 'SIGILL' (4) - Illegal instruction,
5. 'SIGTRAP' (5) - Trace/breakpoint trap,

6. ‘SIGABRT‘ (6) - Aborted,
7. ‘SIGBUS‘ (7) - Bus error,
8. ‘SIGFPE‘ (8) - Floating point exception,
9. ‘SIGKILL‘ (9) - Kill (cannot be caught or ignored),
10. ‘SIGUSR1‘ (10) - User-defined signal 1,
11. ‘SIGSEGV‘ (11) - Segmentation fault,
12. ‘SIGUSR2‘ (12) - User-defined signal 2,
13. ‘SIGPIPE‘ (13) - Broken pipe,
14. ‘SIGALRM‘ (14) - Alarm clock,
15. ‘SIGTERM‘ (15) - Termination,
16. ‘SIGSTKFLT‘ (16) - Stack fault,
17. ‘SIGCHLD‘ (17) - Child status has changed,
18. ‘SIGCONT‘ (18) - Continue (sent by ‘kill -CONT‘),
19. ‘SIGSTOP‘ (19) - Stop (cannot be caught or ignored),
20. ‘SIGTSTP‘ (20) - Terminal stop signal (Ctrl+Z),
21. ‘SIGTTIN‘ (21) - Background read from TTY,
22. ‘SIGTTOU‘ (22) - Background write to TTY,
23. ‘SIGURG‘ (23) - Urgent condition on socket,
24. ‘SIGXCPU‘ (24) - CPU time limit exceeded,
25. ‘SIGXFSZ‘ (25) - File size limit exceeded,
26. ‘SIGVTALRM‘ (26) - Virtual timer expired,
27. ‘SIGPROF‘ (27) - Profiling timer expired,
28. ‘SIGWINCH‘ (28) - Window size change,
29. ‘SIGIO‘ (29) - I/O now possible,
30. ‘SIGPWR‘ (30) - Power failure restart,
31. ‘SIGSYS‘ (31) - Bad system call,
32. ‘SIGUNUSED‘ (31) - Unused signal (alias for ‘SIGSYS‘),
33. ‘SIGRTMIN‘ (34) - First Real-time signal,
34. ‘SIGRTMIN+1‘ (35) - Second Real-time signal,
35. ‘SIGRTMIN+2‘ (36) - Third Real-time signal,
36. ‘SIGRTMIN+3‘ (37) - Fourth Real-time signal,
37. ‘SIGRTMIN+4‘ (38) - Fifth Real-time signal,
38. ‘SIGRTMIN+5‘ (39) - Sixth Real-time signal,
39. ‘SIGRTMIN+6‘ (40) - Seventh Real-time signal,
40. ‘SIGRTMIN+7‘ (41) - Eighth Real-time signal,
41. ‘SIGRTMIN+8‘ (42) - Ninth Real-time signal,
42. ‘SIGRTMIN+9‘ (43) - Tenth Real-time signal,
43. ‘SIGRTMIN+10‘ (44) - Eleventh Real-time signal,
44. ‘SIGRTMIN+11‘ (45) - Twelfth Real-time signal,
45. ‘SIGRTMIN+12‘ (46) - Thirteenth Real-time signal,
46. ‘SIGRTMIN+13‘ (47) - Fourteenth Real-time signal,
47. ‘SIGRTMIN+14‘ (48) - Fifteenth Real-time signal,
48. ‘SIGRTMIN+15‘ (49) - Sixteenth Real-time signal,
49. ‘SIGRTMAX-14‘ (50) - Fifteenth from the top Real-time signal,
50. ‘SIGRTMAX-13‘ (51) - Fourteenth from the top Real-time signal,
51. ‘SIGRTMAX-12‘ (52) - Thirteenth from the top Real-time signal,

52. ‘SIGRTMAX-11‘ (53) - Twelfth from the top Real-time signal,
53. ‘SIGRTMAX-10‘ (54) - Eleventh from the top Real-time signal,
54. ‘SIGRTMAX-9‘ (55) - Tenth from the top Real-time signal,
55. ‘SIGRTMAX-8‘ (56) - Ninth from the top Real-time signal,
56. ‘SIGRTMAX-7‘ (57) - Eighth from the top Real-time signal,
57. ‘SIGRTMAX-6‘ (58) - Seventh from the top Real-time signal,
58. ‘SIGRTMAX-5‘ (59) - Sixth from the top Real-time signal,
59. ‘SIGRTMAX-4‘ (60) - Fifth from the top Real-time signal,
60. ‘SIGRTMAX-3‘ (61) - Fourth from the top Real-time signal,
61. ‘SIGRTMAX-2‘ (62) - Third from the top Real-time signal,
62. ‘SIGRTMAX-1‘ (63) - Second from the top Real-time signal,
63. ‘SIGRTMAX‘ (64) - Highest priority Real-time signal

These signals are defined in ‘signal.h‘ and are standardized across most Unix-like systems, including Linux.

9 What do you know about

1. Orpha Process: An orphan process is a process whose parent process has terminated or finished execution before the child process. In other words, an orphan process is a child process that still exists in the system's process table but no longer has a parent process to which it is linked. Orphan processes are typically adopted by the init process (with PID 1) in Unix-like operating systems.

2. Zoombie Process: A zombie process, also known as a defunct process, is a process that has completed execution but still has an entry in the process table. This entry contains its process ID (PID) and exit status, but the process's resources, such as memory and open file descriptors, have been released.

9.1 When can you call a process an orphan process or a zombie process?

An orphan process and a zombie process are two different states of a process in a Unix-like operating system, and they occur in different situations.

An orphan process is a process whose parent process has terminated or finished before the child process. This situation typically occurs when the parent process exits or terminates unexpectedly before its child processes. Orphan processes are adopted by the init process (process with PID 1 in Unix systems), which becomes their new parent process. The init process periodically calls wait() to collect the exit statuses of orphaned child processes and prevent them from becoming zombies.

A zombie process is a process that has completed execution but still has an entry in the process table. This situation typically occurs when a child process finishes execution, but its parent process fails to call wait() or waitpid() to collect its exit status. The process table entry for the zombie process remains, including its process ID (PID) and exit status, but the process's resources have been released. Zombie processes consume very few system resources but can accumulate if not reaped by their parent process.

9.2 Can you create them? If yes, describe your steps with code.

Yes, I can create them. Below create both orpha process and zoombie process:

To create an orphan process, we'll fork a child process and then exit the parent process before the child has a chance to complete.

To create a zombie process, we'll fork a child process and then let the parent process continue without waiting for the child to complete, thus preventing it from collecting the child's exit status.

The code is given below

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t cpid;
    cpid = fork();

    if (cpid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (cpid == 0) {
        printf("Child process %d is sleeping...\n", getpid());
        sleep(10);
        printf("Child process %d completed.\n", getpid());
    } else {
        printf("Parent PID %d, child PID=%d\n", getpid(), child_pid);
        sleep(5);
        printf("Parent PID %d exiting without waiting for child to complete.
    }
```

```
    return 0;
}
```

In this code, The parent process creates a child process using fork(). In the child process, it goes to sleep for 10 seconds, simulating a long-running task. In the parent process, it doesn't wait for the child to complete and exits after sleeping for 5 seconds.

9.3 How can we trace an orphan and zombie process in a Linux based OS?

In Linux-based operating systems, we can trace orphan and zombie processes using various system monitoring and debugging tools. Those debugging tools are pstree, htop, top, etc. command.

9.4 Explain how a Linux based OS treats them.

In a Linux-based operating system, orphan and zombie processes are treated differently in terms of their management and system behavior:

Orphan processes are automatically adopted by the init process. The init process periodically calls wait() to collect the exit statuses of orphaned child processes. init reaps orphan processes, removing their entries from the process table and releasing associated system resources..

Zombie processes remain in the process table until their parent process collects their exit status. If a parent process fails to collect the exit status of its child, the zombie process remains in the process table until the parent process terminates. When a parent process terminates, the init process automatically inherits any zombie children and reaps them periodically.

9.5 What are the advantages and disadvantages of having orphan and zombie processes?

Orphan and zombie processes have distinct advantages and disadvantages in a Linux-based operating system:

Advantages:
Orphan Processes:
1. Resource Cleanup: Orphan processes allow the parent process to continue executing without waiting for the child process to finish. This can be advantageous if the parent process has other tasks to perform.

2. Fault Tolerance: In case the parent process crashes or terminates unexpectedly, orphan processes are adopted by the init process, preventing resource leakage and ensuring that the system continues to function smoothly.

Zombie Processes:
1. Minimal Resource Consumption: Zombie processes consume minimal system resources, usually just a process table entry and a PID. This allows the system to maintain information about terminated processes without consuming significant memory or CPU resources.
2. Exit Status Storage: Zombie processes retain their exit status until their parent processes collect them. This can be useful for debugging purposes, as the parent process can later retrieve information about the child process's termination.
3. Fault Detection: The presence of zombie processes can indicate a fault in the parent process or an issue with process management. Monitoring the number of zombie processes can help detect and diagnose system problems.

Disadvantages:
Orphan Processes:
1. Resource Leakage: If the parent process terminates without properly handling its child processes, orphan processes can accumulate, leading to resource leakage and potential system instability.
2. Loss of Control: Orphan processes may not be properly monitored or managed by the parent process, leading to situations where critical tasks are left unfinished or resources are not properly released.

Zombie Processes:
1. Process Table Saturation: Accumulation of zombie processes can saturate the process table, preventing new processes from being created and potentially causing system slowdowns or failures.
2. Indication of Poor Programming Practices: The presence of zombie processes often indicates poor programming practices, such as failing to call wait() or waitpid() to collect the exit status of child processes. This can lead to difficult-to-diagnose bugs and system instability.
3. Potential for Misinterpretation: In some cases, the presence of zombie processes may be misinterpreted as a system problem or a sign of malicious activity, leading to unnecessary concern or investigation.

OS All Assignment Report

Md. Miju Ahmed
ID: 2010676104
Session: 2019-2020

November 2024

1 Assignment - 1

1.1 What is PID? Why is it necessary?

PID is a number assigned by the OS to manage and identify each running process.
PID in an important part for managing the process. It can track the process by keeping the data into the PCB table. It is necessary because it can manage Process management, Resource allocation, Inte-process communication, Process control and termination, Debugging and monitoring.

1.2 Is PID necessary only in Ubuntu? In Windows or other operating system, is there any similar concept?

Its not necessary only Ubuntu. It exists in almost every operating system.
Yes, every major operating system, including Windows, macOS, and Unix-based systems, uses the concept of a PID to manage processes. Though the methods of managing, displaying, and interacting with PIDs vary across operating systems,

1.3 Is there any process having PID = 0? If yes, write down the tasks or responsibilities of that process.

Yes, there are having PID = 0 for the operating system. This processes having some specific roles.
The task for this PID = 0 is PID 0 is assigned to the swapper or idle process and It's responsible for managing CPU idle time and performing basic system functions when no other processes are ready to execute.
The responsibilites of PID 0 is CPU management, Context switching, Initialization.

1.4 Write down the task the tasks or responsibilities of that process having PID 1.

The PID 1 indicates the init process or systemd. It is one of the most critical system process. This process is the first user-space process started by the kernel during boot, and it acts as the ancestor of all other processes in the system.
The responsibilities of PID-1 process is System initialize, process management, reaping orphan processes, runlevel or target management, shutdown or restart control.