Practical 1

11/1/2024

- **Aim :** Identify Hardware requirements for modern operating system
- **Theory :**
- **What is an Operating system ? :**

Operating System lies in the category of system software. It basically manages all the resources of the computer. An operating system acts as a interface between the hardware and software of computer

- **Kernel :**

a kernal is a small program located in the core of operating system. which manages all the operations of computer. it helps operating system to communicate to computer. it's also called the heart of operating system.

- kernal loads first into memory when an operating system is loaded and remains into memory until OS is shut down. it is responsible for various task like Disk management etc.
- There are different types of operating system available like window, linux, MacOS etc. let's see hardware requirements of them.

- System requirements of windows 11 :
- Processor : windows 11 requires 1 gigahertz (GHz) or faster processor with 2 or more cores on a compatible 64-bit processor or system on a chip.
- RAM : windows 11 requires minimum 4 giga-bytes (GB) of ram to operate.
- Storage : windows 11 requires minimum 64 GB of storage.
- Graphics card : windows 11 is compatible with DirectX 12 or later with WDDM 2.0 driver.
- Display : requires HD (720p) display that is greater than 9" digonally.

- System requirements of linux 2.1.2 :
- Processor : requires 64 bit single core with 2 GHz speed or more.
- RAM : requires minimum 4GB RAM
- Storage : requires 100 GB of hard drive space.
- display : requires 1440 x 900 resolution minimum

- Difference between configuration of windows and Linux OS.

	Windows	Linux
1) Kernel:	it uses windows NT kernel which is a ~~higher level~~ components. hybrid kernal Combines features of microke -rnal and monolithic kernal architecture.	1) uses the Linux kernal, which is monolithic kernel that manages hardware resources and provides services to Higher level components.
file syste -m	2) includes file systems like NTFS, FAT32	2) includes file systems like XFS, ext4
UI	3) Has a GUI provided by the windows Desktop environment	3) often serves without GUI using only command line.
License	4) windows is licensed OS which is not open source	4) Linux is open source operating system
Security	5) Administrator user has all administrative privileges of computer	5) Root user is the super user and has all administ -rative privileges.

- ~~Result~~ : Hence, we successfully identified Hard ware requirements for modern OS.

Practical 2

- Aim: To implement the concept of process Creation

- Theory:

What is process?

i) Process is a program in execution

1) when a new process is created, the operating system assigns a unique process identifier (PID) to it and insert a new entry in the primary process table.

2) Then required memory space for all the elements of the process such as program, data and stack is allocated including space for process control block.

3) process creation is achieved through the fork() command system call.

4) After the fork() system call now we have two processes - parent and child process.

ii) getppid():

Returns the process ID of the parent of the calling process. If the calling process was created by the fork() function and the parent process still exists at the time of the getppid function call, this function returns the process id of the parent process otherwise, this function returns a value of 1 which is the process id for init process.

program:-

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
 printf("Hello World!\n");
 fork( );
 printf("I am after forking\n");
 printf("\tI am process %d.\n", getpid( ));
}
```

output:-

```
Hello World!
I am after forking
    I am process 12345.
I am after forking
    I am process 12346.
```

program:-

```
#include <stdio.h>
#include <unistd.h> // for getpid(), getppid(), and fork()
#include <stdlib.h> // for exit()

int main() {
    int pid;

    printf("I'm the original process with PID %d and PPID %d.\n",
           getpid(), getppid());

    pid = fork(); /* Duplicate. Child and parent continue from here */

    if (pid != 0) /* pid is non-zero, so I must be the parent */
    {
        printf("I'm the parent with PID %d and PPID %d.\n",
               getpid(), getppid());
        printf("My child's PID is %d\n", pid);
    }
    else /* pid is zero, so I must be the child */
    {
        sleep(4); // make sure that the parent terminates first
        printf("I'm the child with PID %d and PPID %d.\n",
               getpid(), getppid());
    }

    printf("PID %d terminates.\n", getpid());
    return 0;
}
```

output:-
I'm the original process with PID 1234 and PPID 5678.
I'm the parent with PID 1234 and PPID 5678.
My child's PID is 1235
PID 1234 terminates.
I'm the child with PID 1235 and PPID 1234.
PID 1235 terminates.

program:-

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

int main() {
    pid_t pid;

    // fork a child process
    pid = fork();

    if (pid < 0) { // error occurred
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { // child process
        printf("I am the child, PID = %d, PPID = %d\n", getpid(), getppid());
    }
    else { // parent process
        printf("I am the parent, PID = %d\n", getpid());
    }

    return 0;
}
```

output:-

```
I am the parent, PID = 7046
I am the child, PID = 7047, PPID = 7046
```

Syntax : ~~pid_t~~ getppid (void);

getppid() returns the process ID of the parent of the current process. It never throws any error therefor is always successful.

iii) getpid():

getpid() returns the process ID of calling process. This is often used by routines that generate unique temporary filenames

Syntax :- ~~pid_t~~ getpid (void);

getpid() returns the process ID of the current process, It never throws any error therefore is always successful.

Result : Thus we successfully implemented the concept of process creation

P	T	D	K	Total
3M	3M	3M	3M	15M
Sign With date				

Practical 3

- Aim : Implement program for FCFS CPU scheduling algorithm.
- Theory :

First come first serve (FCFS) is also known as first in first out (FIFO). FCFS is the simplest scheduling algorithm. FCFS simply queues processes in the order that they arrive in the ready queue.

In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed Here we are considering that arrival time for all process is 0.

Given n processes with their burst times the task is to find average waiting time and average turn around time using FCFS scheduling algorithm.

Key points :

1) Arrival time : At what time process will enter into ready queue.

2) Burst time : Time required to execution of process.

3) Completion time :

Time At which process completes it's execution.

4) Turn Around time :
The Difference between completion time and Arrival time.

Turn Around Time = completion time - Arrival time

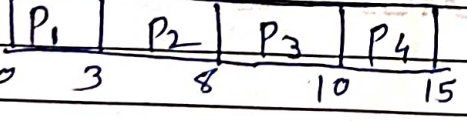
5) Waiting Time :-
Time difference between Turn Around time and burst time

Waiting time = Turn Around time - Burst time

eg:

Processes	Burst time	Arrival time
P_1	3	0
P_2	5	1
P_3	2	2
P_4	5	3

Gantt chart



Turn-Around Time = Completion time - Arrival time

Process
$P_1 = 3-0 = 3$
$P_2 = 8-1 = 7$
$P_3 = 10-2 = 8$
$P_4 = 15-3 = 12$

Average Turn Around Time = $\frac{3+7+8+12}{4}$

$= \frac{30}{4} = 7.22$

```
#include<stdio.h>
void findWaitingTime(int processes[], int n,
                                        . int bt[], int wt[])
{
wt[0] = 0;
for (int i = 1; i < n ; i++ )
wt[i] = bt[i-1] + wt[i-1] ;
}
void findTurnAroundTime( int processes[], int n,
                                int bt[], int wt[], int tat[])
{
for (int i = 0; i < n ; i++)
tat[i] = bt[i] + wt[i];
}
void findavgTime( int processes[], int n, int bt[])
{
        int wt[n], tat[n], total_wt = 0, total_tat = 0;
        findWaitingTime(processes, n, bt, wt);
        findTurnAroundTime(processes, n, bt, wt, tat);
        printf("Processes Burst time Waiting time Turn around time\n");
        for (int i=0; i<n; i++)
        {
                total_wt = total_wt + wt[i];
                total_tat = total_tat + tat[i];
                printf(" %d ",(i+1));
                printf("        %d ", bt[i] );
                printf("        %d",wt[i] );
                printf("        %d\n",tat[i] );
```

```
    }
    float s=(float)total_wt / (float)n;
    float t=(float)total_tat / (float)n;
    printf("Average waiting time = %f",s);
    printf("\n");
    printf("Average turn around time = %f ",t);
}
int main()
{   int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    int burst_time[] = {10, 5, 8};

    findavgTime(processes, n, burst_time);
    return 0;
}
```

Output

Clear

```
/tmp/EZIbVh0Z89.o
Processes Burst time Waiting time Turn around time
 1    10         0    10
 2    5    10   15
 3    8    15   23
Average waiting time = 8.333333
Average turn around time = 16.000000
```

Waiting time = Turn Around time − Burst time

Process $P_1 = 3 - 3 = 0$

$P_2 = 7 - 5 = 2$

$P_3 = 8 - 2 = 6$

$P_4 = 12 - 5 = 7$

Average waiting time $= \frac{0+2+6+7}{4} = \frac{15}{4}$

$= \boxed{3.75}$

Result : Hence, we successfully implement a program for FCFS CPU scheduling Algorithm

P	T	D	K	Total
3M		3M		15M
Sign with date				

Practical 4

Aim : Implement the Concept of semaphore

Theory : Semaphore are fundamental synchron -ization primitives used in operating systems to control access to shared resources. They help in preventing race conditions and ensuring that critical sections of code are executed automatically. In this practical, ~~sephms~~ semaphore concept is implement -ed.

~~Sepha~~ semaphore is a integer variable used for controlling access to a common resource by multiple processes in concurrent system. It provides two fundamental operations.

1) Wait (P)
2) Signal (V)

Semaphore maintain a count of that represents the availability of resources.

- Wait (P) operation :
When a process wishes to enter a critical section. It executes the wait operation on the ~~sepha~~ semaphore If the semaphore's value is greater than zero. It decrements the semaphore and proceeds. Otherwise, if the value is zero, the process is blocked until semaphore's value becomes positive.

- Signal (V) operation :
when a process exits the critical section, it executes the signal operation on the semaphore, incrementing its value. If there are processes :

waiting (blocked) on the semaphore, one of them is woken up It first locks semaphore mutex, increments the count to indicate that resource is available, and signals thread to wakeup any waiting threads.

- Destroy semaphore:
  It is used to clean up the semaphore resources by destroying its mutex & its variables.

- Types of semaphore:

  - Binary semaphore:
    Can take only two values typically 0 and 1. used for mutex locks

  - Counting semaphore:
    Can take multiple values used for resource counting and synchroniza-tion

- Procedure:

1) Initialize semaphore: Initialize semaphore section variable with an initial value using appropriate system calls or programming language concepts.

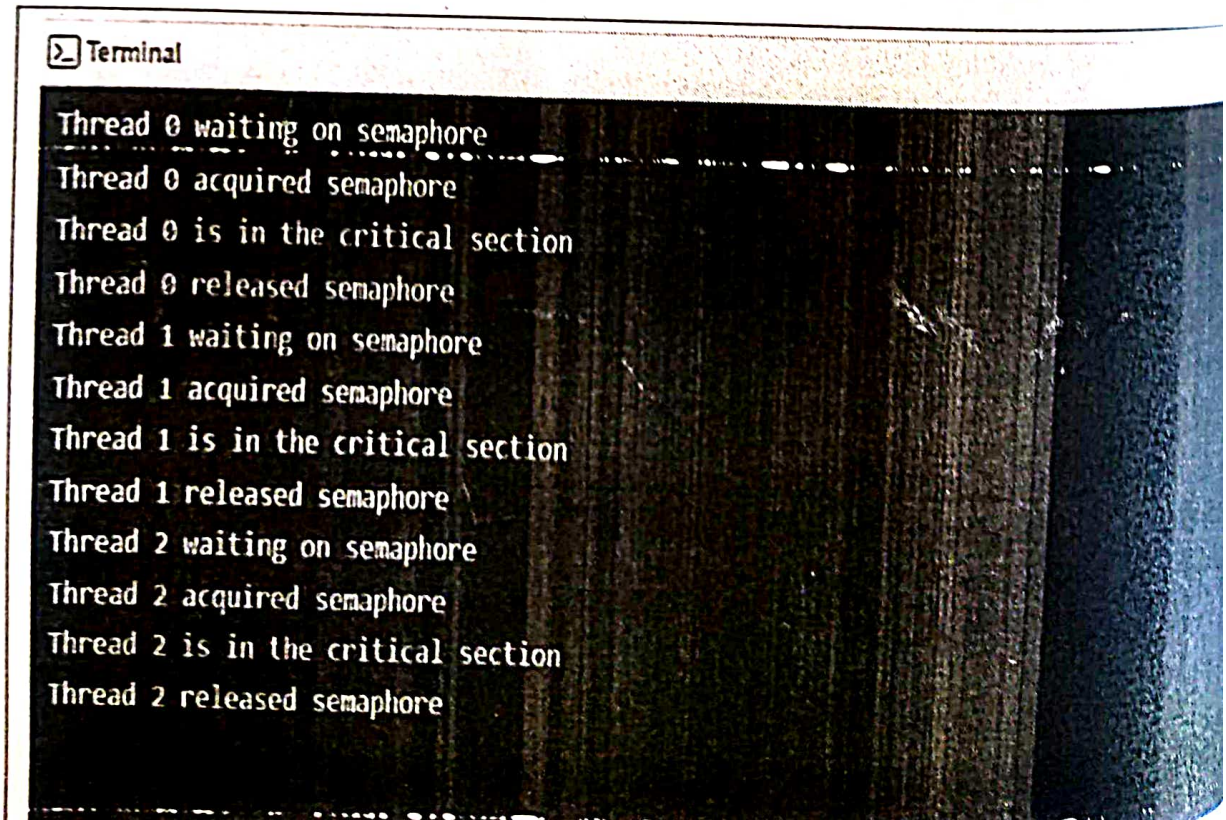
2) Implement Critical sections:
   Identity critical section of code where access to shared resources needs to be synchronized.

CODE :

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

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int count;
} semaphore_t;

void init_semaphore(semaphore_t *sem, int initial_count) {
    pthread_mutex_init(&sem->mutex, NULL);
    pthread_cond_init(&sem->cond, NULL);
    sem->count = initial_count;
}

void wait_semaphore(semaphore_t *sem) {
    pthread_mutex_lock(&sem->mutex);
    while (sem->count <= 0) {
        pthread_cond_wait(&sem->cond, &sem->mutex);
    }
    sem->count--;
    pthread_mutex_unlock(&sem->mutex);
}

void signal_semaphore(semaphore_t *sem) {
    pthread_mutex_lock(&sem->mutex);
    sem->count++;
    pthread_cond_signal(&sem->cond);
    pthread_mutex_unlock(&sem->mutex);
}

void destroy_semaphore(semaphore_t *sem) {
    pthread_mutex_destroy(&sem->mutex);
    pthread_cond_destroy(&sem->cond);
}

// Example usage
semaphore_t my_semaphore;

void* thread_function(void *arg) {
    int thread_id = *((int*)arg);
    printf("Thread %d waiting on semaphore\n", thread_id);

    wait_semaphore(&my_semaphore);
    printf("Thread %d acquired semaphore\n", thread_id);
    // Critical Section
    printf("Thread %d is in the critical section\n", thread_id);
    signal_semaphore(&my_semaphore);
    printf("Thread %d released semaphore\n", thread_id);
```

```
    pthread_exit(NULL);
}

int main() {
    int num_threads = 3;
    pthread_t threads[num_threads];
    int thread_ids[num_threads];

    init_semaphore(&my_semaphore, 1);

    for (int i = 0; i < num_threads; ++i) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
    }

    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], NULL);
    }

    destroy_semaphore(&my_semaphore);
    return 0;
}
```

OUTPUT :



3) Semaphore wait (P):
Before entring critical section, execute the wait operation on semaphore.

4) semaphore signal (V):
After exiting initial section, executing the signal operation on the semaphore.

- Result: Hence, we successfully implement the concept of semaphore.