Operating System

IIIWC09 PRACTICAL RECORD FILE

2021-2022

Name:Adosh Singh

Roll No.: 2020UII7403 B.Tech (IIOT)



**Netaji Subhas University of Technology**

|  |  |  |  |
| --- | --- | --- | --- |
| S.no. | INDEX | PAGE NO. | DATE |
| 1 | Process creation and termination using fork() and exit() | 1 | 22/02/22 |
| 2 | Process handling using exec() and wait() | 2-4 | 08/03/22 |
| 3 | CPU scheduling using First Come First Serve algorithm | 5-6 | 15/03/22 |
| 4 | CPU scheduling using Shortest job first algorithm | 6-7 | 22/03/22 |
| 5 | CPU scheduling using “Priority based”(using assigned ranks/priority) algorithm | 8-9 | 29/03/22 |
| 6 | CPU scheduling using “Round Robin” algorithm | 9-10 | 05/04/22 |
| 7 | Inter Process Communication using shared memory | 13 | 12/04/22 |

1. **Process creation and termination using**

**fork() and exit()**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

int main(int argc, char \*\*argv)

{

    pid\_t pid;

    pid = fork();

    if (pid == 0)

    {

        printf("It is the child process and pid is %d\n", getpid());

        exit(0);

    }

    else if (pid > 0)

    {

        printf("It is the parent process and pid is %d\n", getpid());

    }

    else

    {

        printf("Error while forking\n");

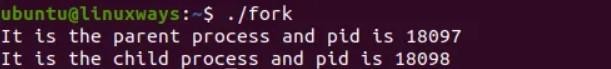
        exit(EXIT\_FAILURE);

    }

    return 0;

}

Output:



1. **Process handling using exec() and wait()**

# exec():

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

main(void)

{

    pid\_t pid = 0;

    int status;

    pid = fork();

    if (pid == 0)

    {

        printf("I am the child.");

        execl("/bin/ls", "ls", "-l", "/home/ubuntu/", (char \*)0);

        perror("In exec(): ");

    }

    if (pid > 0)

    {

        printf("I am the parent, and the child is %d.\n", pid);

        pid = wait(&status);

        printf("End of process %d: ", pid);

        if (WIFEXITED(status))

        {

            printf("The process ended with exit(%d).\n", WEXITSTATUS(status));

        }

        if (WIFSIGNALED(status))

        {

            printf("The process ended with kill -%d.\n", WTERMSIG(status));

        }

    }

    if (pid < 0)

    {

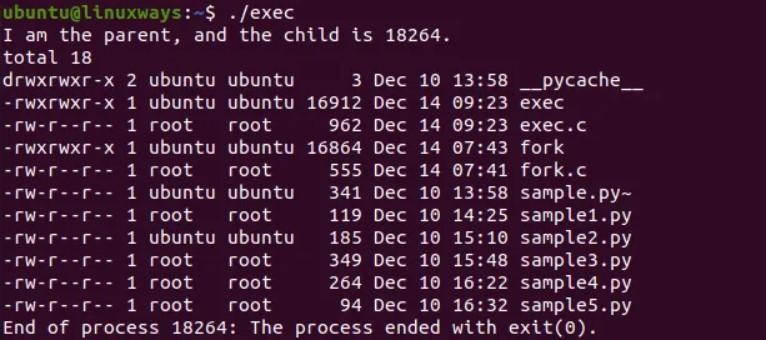
        perror("In fork():");

    }

    exit(0);

}

# Output:



wait():

#include <stdio.h> // printf() #include <stdlib.h> // exit() #include <sys/types.h> // pid\_t #include <sys/wait.h> // wait() #include <unistd.h> // fork

int main(int argc, char \*\*argv)

{

pid\_t pid; pid = fork();

if (pid == 0)

{

printf("It is the child process and pid is %d\n", getpid());

int i = 0;

for (i = 0; i < 8; i++)

{

printf("%d\n", i);

}

exit(0);

}

else if (pid > 0)

{

printf("It is the parent process and pid is %d\n", getpid());

}

else

{

}

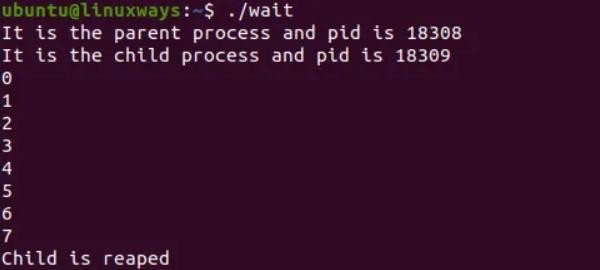
int status; wait(&status);

printf("Child is reaped\n");

printf("Error in forking..\n"); exit(EXIT\_FAILURE);

return 0;}

Output:



1. **CPU scheduling using First Come First**

**Serve algorithm**

#include<iostream> using namespace std;

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);

cout << "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];

cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n; cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

int main()

{

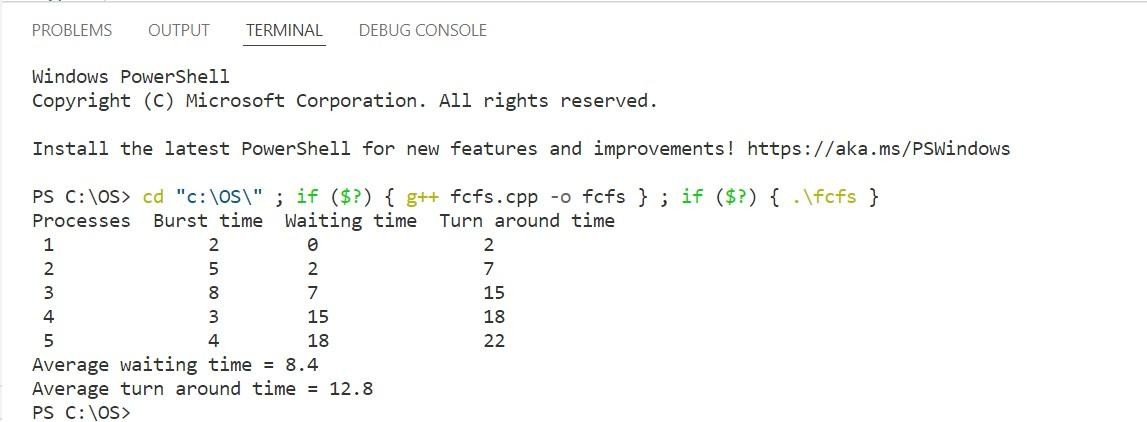
int processes[] = { 1, 2, 3, 4, 5};

int n = sizeof processes / sizeof processes[0]; int burst\_time[] = {2, 5, 8, 3, 4};

findavgTime(processes, n, burst\_time);

return 0;

}



1. **CPU scheduling using Shortest Job First**

**algorithm**

#include<bits/stdc++.h> using namespace std;

void findWaitingTime(int processes[], int n,

int bt[], int wt[])

{

sort(bt, bt+n);

wt[0]=0;

for(int i=1; i<n; i++){ wt[i]=0;

for(int j=0; j<i; j++){ wt[i]+=bt[j];

}

}

}

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); cout << " 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];

cout <<"\t "<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n; cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

int main (){

int processes[] = { 1, 2, 3, 4, 5};

int n = sizeof processes / sizeof processes[0]; int burst\_time[] = {2, 5, 8, 3, 4};

findavgTime(processes, n, burst\_time);

return 0;

}



1. **CPU scheduling using Priority based**

**algorithm**

#include <bits/stdc++.h> using namespace std;

void findWaitingTime(int processes[], int n,

int bt[], int wt[], int pr[])

{

for (int i = 0; i < n - 1; i++)

{

for (int j = 1; j <= n - i - 1; j++)

{

if (pr[j - 1] > pr[j])

{

int t = pr[j - 1]; pr[j - 1] = pr[j]; pr[j] = t;

t = bt[j - 1]; bt[j - 1] = bt[j]; bt[j] = t;

t = processes[j - 1]; processes[j - 1] = processes[j]; processes[j] = t;

}

}

}

wt[0] = 0;

for (int i = 1; i < n; i++)

{

wt[i] = 0;

for (int j = 0; j < i; j++)

{

wt[i] += bt[j];

}

}

}

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 pr[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0; findWaitingTime(processes, n, bt, wt, pr); findTurnAroundTime(processes, n, bt, wt, tat); cout <<"Priority " << "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];

cout <<" "<< pr[i] <<"\t" << " " << processes[i] << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n; cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

int main()

{

int processes[] = { 1, 2, 3, 4};

int n = sizeof processes / sizeof processes[0];

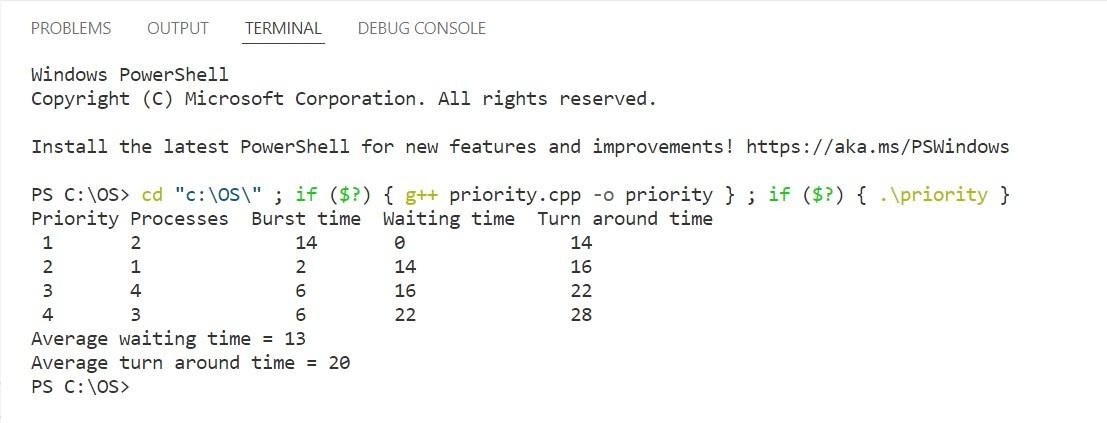
int burst\_time[] = {2, 14, 6, 6};

int pr[] = {2, 1, 4, 3};

findavgTime(processes, n, burst\_time, pr);

return 0;

}



1. **CPU scheduling using Round Robin algorithm**

#include<iostream> using namespace std;

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum)

{

int rem\_bt[n];

for (int i = 0 ; i < n ; i++) rem\_bt[i] = bt[i];

int t = 0; while (1)

{

bool done = true;

for (int i = 0 ; i < n; i++)

{

if (rem\_bt[i] > 0)

{

done = false;

if (rem\_bt[i] > quantum)

{

}

else

{

}

}

}

t += quantum; rem\_bt[i] -= quantum;

t = t + rem\_bt[i]; wt[i] = t - bt[i]; rem\_bt[i] = 0;

if (done == true) break;

}

}

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 quantum)

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0; findWaitingTime(processes, n, bt, wt, quantum);

findTurnAroundTime(processes, n, bt, wt, tat);

cout << "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];

cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n; cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

int main()

{

int processes[] = { 1, 2, 3, 4, 5};

int n = sizeof processes / sizeof processes[0]; int burst\_time[] = {2, 5, 8, 3, 4};

int quantum = 2;

findavgTime(processes, n, burst\_time, quantum); return 0;

}



1. **Interprocess Communication using shared**

**memory**

# For writer process:

#include <iostream> #include <sys/ipc.h> #include <sys/shm.h #include <stdio.h> using namespace std;

int main()

{

key\_t key = ftok("shmfile", 65);

int shmid = shmget(key, 1024, 0666 | IPC\_CREAT); char \*str = (char \*)shmat(shmid, (void \*)0, 0); cout << "Write Data : ";

gets(str);

printf("Data written in memory: %s\n", str); shmdt(str);

return 0;

}

# For reader process:

#include <iostream> #include <sys/ipc.h> #include <sys/shm.h> #include <stdio.h> using namespace std; int main()

{

key\_t key = ftok("shmfile", 65);

int shmid = shmget(key, 1024, 0666 | IPC\_CREAT); char \*str = (char \*)shmat(shmid, (void \*)0, 0);

printf("Data read from memory: %s\n", str); shmdt(str);

shmctl(shmid, IPC\_RMID, NULL); return 0;

}

# Output:

