Jagan Institute of Management Studies

3, Institutional Area, Sector-5, Rohini, Delhi-110085

MCA- 163 Lab: Operating System with Linux

Assignment 3,4

--Kartik Sharma--

1. Write a program that creates a child process and displays the process id and parent process ids of both the parent and the child process.

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

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

  if (child_pid < 0) {
    // Fork failed
    printf("Fork failed!\n");
    return 1;
  }
  else if (child_pid == 0) {
    // This code runs in the child process
    printf("\nChild Process:\n");
    printf("Process ID: %d\n", getpid());
    printf("Parent Process ID: %d\n", getppid());
  }
  else {
    // This code runs in the parent process
    printf("\nParent Process:\n");
    printf("Process ID: %d\n", getpid());
    printf("Parent Process ID: %d\n", getppid());
    printf("Child Process ID: %d\n", child_pid);
  }

  return 0;
}
```

output

```
$ gcc -o process_demo process_demo.c
$ ./process_demo

Parent Process:
Process ID: 12345
Parent Process ID: 12344
Child Process ID: 12346

Child Process:
Process ID: 12346
Parent Process ID: 12345
$
```

2. Write a program to demonstrate thread creation. The thread created should simply print a fixed message as many times as required by the main program (pass data to the thread). (Hint: To compile this program, gcc progfile.c –lpthread)

```
#include <stdio.h>
#include <pthread.h>
void* printMessage(void* arg) {
   int count = *(int*)arg;
   for (int i = 0; i < count; i++) {
      printf("Hello from Kartik!\n");
   }
   return NULL;
}
int main() {
   pthread_t thread;
   int repeatCount = 5;
  pthread_create(&thread, NULL,
printMessage, &repeatCount);

   pthread_join(thread, NULL);

  printf("Thread has finished executing.\n");
  return 0;
}
```

Output:

Hello from Kartik!
Hello from Kartik!
Hello from Kartik!
Hello from Kartik!
Hello from Kartik!
Thread has finished executing.

3. Write a program to display parent and child relationship with fork command.

```
# include <unistd.h> #
include <sys/types.h> #
include <stdio.h> int
main() {

    int pid ; pid
    = fork( ) ; if
    ( pid > 0 )

  printf ( "\n Child : Hello I am the child
  process\n" ) ;

 else
  printf ( "\n Parent : Hello I am the parent
  process\n" ) ;

 }
```

Output:

```
Child : Hello I am the child process

Parent : Hello I am the parent process
```

4. Write a program to demonstrate process synchronization.

```
# include <unistd.h>
# include
<sys/types.h> #
include <stdio.h> int
main() {

      int pid,i ; pid
      = fork( ) ; if (
      pid > 0 ) {

          for(
          i=0;i<=20;i++) {
                sleep(1);
                printf("Parent %d",
                i);
                }
        }
```

```
        else
        {
            for( i=0;i<=20;i++)
            {
                sleep(1);
                printf("child %d", i);
            }
        }
    }
```

Output:

```
Parent 0Parent 1Parent 2Parent 3Parent 4Parent 5Parent 6Parent 7Parent 8Parent 9Parent 10Parent 11Parent 12Parent 13Parent 14Parent 15Paren
t 16Parent 17Parent 18Parent 19Parent 20child 0child 1child 2child 3child 4child 5child 6child 7child 8child 9child 10child 11child 12child
 13child 14child 15child 16child 17child 18child 19child 20~$
```

5. Write a C program to calculate total waiting and turnaround time of n processes with FCFS CPU Scheduling algorithm.

```
// C program for implementation of FCFS scheduling

#include<stdio.h>
// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
    // waiting time for first process is 0
    wt[0] = 0;

    // calculating waiting time
    for (int i = 1; i < n ; i++ )
        wt[i] = bt[i-1] + wt[i-1] ;
}
// Function to calculate turn around time
void findTurnAroundTime( int processes[], int n,

                        int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

//Function to calculate average time
void findavgTime( int processes[], int n, int bt[])
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    //Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt);
    //Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);
    //Display processes along with all details
    printf("Processes Burst time Waiting time Turn around time\n");
    // Calculate total waiting time and total turn
    // around time
    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] );
    }
    int s=(float)total_wt / (float)n;
    int t=(float)total_tat / (float)n;
    printf("Average waiting time = %d",s);
    printf("\n");
    printf("Average turn around time = %d ",t);
}

// Driver code
int main()
{
    //process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];
    //Burst time of all processes
    int burst_time[] = {10, 5, 8};
    findavgTime(processes, n, burst_time);
    return 0;

}
```

Output:

```
~$ ./a.out
Processes Burst time Waiting time Turn around time
 1        10        0        10
 2        5         10       15
 3        8         15       23
Average waiting time = 8
Average turn around time = 16 ~$
```

6. Write a C++ Program for Priority CPU Scheduling.

```
// C++ program for implementation of FCFS scheduling
#include<bits/stdc++.h>
using namespace std;
struct Process
{
        int pid; // Process ID
        int bt; // CPU Burst time required
        int priority; // Priority of this process
};
// Function to sort the Process acc. to priority
bool comparison(Process a, Process b)
{
        return (a.priority > b.priority);
}
// Function to find the waiting time for all processes
void findWaitingTime(Process proc[], int n,
                                        int wt[])
{
        // waiting time for first process is 0
        wt[0] = 0;

        // calculating waiting time
        for (int i = 1; i < n ; i++ )
                wt[i] = proc[i-1].bt + wt[i-1] ;
```

```
}

// Function to calculate turn around time
void findTurnAroundTime( Process proc[], int n, int wt[], int tat[])
{
	// calculating turnaround time by adding
	// bt[i] + wt[i]
	for (int i = 0; i < n ; i++)
		tat[i] = proc[i].bt + wt[i];
}

//Function to calculate average time
void findavgTime(Process proc[], int n)
{
	int wt[n], tat[n], total_wt = 0, total_tat = 0;

	//Function to find waiting time of all processes
	findWaitingTime(proc, n, wt);

	//Function to find turn around time for all processes
	findTurnAroundTime(proc, n, wt, tat);

	//Display processes along with all details
	cout << "\nProcesses "<< " Burst time "
		<< " Waiting time " << " Turn around time\n";

	// Calculate total waiting time and total turn
	// around time
	for (int i=0; i<n; i++)
	{
		total_wt = total_wt + wt[i];
		total_tat = total_tat + tat[i];
		cout << " " << proc[i].pid << "\t\t"
			<< proc[i].bt << "\t " << wt[i]
			<< "\t\t " << tat[i] <<endl;
	}

	cout << "\nAverage waiting time = "
		<< (float)total_wt / (float)n;
	cout << "\nAverage turn around time = "
		<< (float)total_tat / (float)n;
}

void priorityScheduling(Process proc[], int n)
{
	// Sort processes by priority
	sort(proc, proc + n, comparison);

	cout<< "Order in which processes gets executed \n";
	for (int i = 0 ; i < n; i++)
		cout << proc[i].pid <<" " ;

	findavgTime(proc, n);
}
// Driver code
int main()
{
	Process proc[] = {{1, 10, 2}, {2, 5, 0}, {3, 8, 1}};
```

```
        int n = sizeof proc / sizeof proc[0];
        priorityScheduling(proc, n);
        return 0;
    }
```

Output:

```
Order in which processes get executed (based on priority):
1 3 2

Processes  Priority  Burst time  Waiting time  Turn around time
 1            2            10        0                 10
 3            1            8         10                18
 2            0            5         18                23

Average waiting time = 9.33333
Average turn around time = 17~$
```

7. Write a C program to implement Banker's Algorithm.

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here
    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                        { 2, 0, 0 }, // P1 {
                        3, 0, 2 }, // P2 { 2,
                        1, 1 }, // P3 { 0, 0,
                        2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 9, 0, 2 }, // P2
                      { 2, 2, 2 }, // P3
                      { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available Resources

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {

                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }
```

```
                                if (flag == 0) {
                                        ans[ind++] = i;
                                        for (y = 0; y < m; y++)
                                                avail[y] += alloc[i][y];
                                        f[i] = 1;
                                }
                        }
                }
        }
        printf("Following is the SAFE Sequence\n");
        for (i = 0; i < n - 1; i++)
                printf(" P%d ->", ans[i]);
        printf(" P%d", ans[n - 1]);
        return (0);


}
```

Output:

```
~$ ./a.out
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2~$ 
```

8. Write a program to simulate the following contiguous memory allocation Techniques
   a) Worst fit
   b) Best fit
   c) First fit

a) Worst fit

```
// C++ implementation of worst - Fit algorithm
#include<bits/stdc++.h>
using namespace std;

// Function to allocate memory to blocks as per worst fit
// algorithm
void worstFit(int blockSize[], int m, int processSize[],
                                                                                                                        int
n)
{
        // Stores block id of the block allocated to a
        // process
        int allocation[n];

        // Initially no block is assigned to any process
        memset(allocation, -1, sizeof(allocation));
        // pick each process and find suitable blocks
        // according to its size ad assign to it
        for (int i=0; i<n; i++)
        {
                // Find the best fit block for current process
                int wstIdx = -1;
                for (int j=0; j<m; j++)
                {
                        if (blockSize[j] >= processSize[i])
                        {
                                if (wstIdx == -1)
                                        wstIdx = j;
                                else if (blockSize[wstIdx] < blockSize[j])
                                        wstIdx = j;
                        }
                }
```

```
                // If we could find a block for current process
                if (wstIdx != -1)
                {
                        // allocate block j to p[i] process
                        allocation[i] = wstIdx;

                        // Reduce available memory in this block.
                        blockSize[wstIdx] -= processSize[i];
                }
        }
        cout << "\nProcess No.\tProcess Size\tBlock no.\n";
        for (int i = 0; i < n; i++)
        {

                cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
                if (allocation[i] != -1)
                        cout << allocation[i] + 1;
                else
                        cout << "Not Allocated";

                cout << endl;
        }
}

// Driver code
int main()
{
        int blockSize[] = {100, 500, 200, 300, 600};
        int processSize[] = {212, 417, 112, 426};
        int m = sizeof(blockSize)/sizeof(blockSize[0]);
        int n = sizeof(processSize)/sizeof(processSize[0]);

        worstFit(blockSize, m, processSize, n);

        return 0 ;
}
```

Output:

```
Process No.     Process Size    Block no.
  1             212             5
  2             417             2
  3             112             5
  4             426             Not Allocated
~$
```

b) Best Fit

```
// C++ implementation of Best - Fit algorithm
#include<bits/stdc++.h>
using namespace std;
// Function to allocate memory to blocks as per Best fit
// algorithm
void bestFit(int blockSize[], int m, int processSize[], int n)
{

        // Stores block id of the block allocated to a
        // process
        int allocation[n];
        // Initially no block is assigned to any process
        memset(allocation, -1, sizeof(allocation));
        // pick each process and find suitable blocks
        // according to its size ad assign to it
        for (int i=0; i<n; i++)
```

```
	{
		// Find the best fit block for current process
		int bestIdx = -1;
		for (int j=0; j<m; j++)
		{
			if (blockSize[j] >= processSize[i])
			{
				if (bestIdx == -1)
					bestIdx = j;
				else if (blockSize[bestIdx] > blockSize[j])
					bestIdx = j;
			}
		}

		// If we could find a block for current process
		if (bestIdx != -1)
		{
			// allocate block j to p[i] process
			allocation[i] = bestIdx;

			// Reduce available memory in this block.
			blockSize[bestIdx] -= processSize[i];
		}
	}
	cout << "\nProcess No.\tProcess Size\tBlock no.\n";
	for (int i = 0; i < n; i++)
	{
		cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
		if (allocation[i] != -1)
			cout << allocation[i] + 1;
		else
			cout << "Not Allocated";
		cout << endl;
	}
}

// Driver code
int main()
{
	int blockSize[] = {100, 500, 200, 300, 600};
	int processSize[] = {212, 417, 112, 426};
	int m = sizeof(blockSize)/sizeof(blockSize[0]);
	int n = sizeof(processSize)/sizeof(processSize[0]);

	bestFit(blockSize, m, processSize, n);

	return 0 ;
}
```
Output:

```
Process No.     Process Size    Block no.
 1              212             4
 2              417             2
 3              112             3
 4              426             5
~$
```

c) First Fit

```
// C++ implementation of First - Fit
algorithm #include<bits/stdc++.h> using
namespace std;

// Function to allocate memory to
// blocks as per First fit algorithm
void firstFit(int blockSize[], int m,
                  int processSize[], int n)
{
    // Stores block id of the
    // block allocated to a process
    int allocation[n];
    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));
    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i = 0; i < n; i++)
    {

        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                // allocate block j to p[i] process
                allocation[i] = j;
                // Reduce available memory in this block.
                blockSize[j] -= processSize[i];
                break;

            }
        }
    }
    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {

        cout << " " << i+1 << "\t\t"
            << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";

        cout << endl;
    }
}
// Driver code
int main()
{

    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0 ;
}
```

Output:

```
Process No.     Process Size    Block no.
 1              212             2
 2              417             5
 3              112             2
 4              426             Not Allocated
~$
```

9. Work with Linux memory management commands Top, free, /proc/meminfo.

TOP: The top command is used to display real-time system processes and memory usage.

```
top - 11:02:19 up 16:04,  0 users,  load average: 0.90, 0.73, 0.80
Tasks:   6 total,   1 running,   5 sleeping,   0 stopped,   0 zombie
%Cpu(s): 13.0 us,  6.1 sy,  1.0 ni, 76.4 id,  1.6 wa,  0.0 hi,  1.9 si,  0.0 st
MiB Mem :  32090.3 total,   7257.5 free,   2965.9 used,  21866.9 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  27057.9 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    7 user      38  18  942132  92984  43528 S   0.7   0.3   0:02.65 node
    1 user      20   0    2780    960    868 S   0.0   0.0   0:00.01 tini
    6 user      38  18    2892    976    876 S   0.0   0.0   0:00.00 sh
  211 user      38  18   15440   8764   7240 S   0.0   0.0   0:00.01 sshd
  322 user      38  18    7696   6808   3512 S   0.0   0.0   0:00.05 bash
  357 user      38  18    7816   3764   3160 R   0.0   0.0   0:00.00 top
```

F REE: The free command provides a summary of system memory usage.

```
New File   Smaller   Bigger   Clear   Pause   Kick
~$ free
              total        used        free      shared  buff/cache   available
Mem:       32860440     2847180     7615648        5364    22397612    27897012
Swap:             0           0           0
~$ free -h
              total        used        free      shared  buff/cache   available
Mem:           31Gi       2.7Gi       7.3Gi       5.0Mi        21Gi        26Gi
Swap:            0B          0B          0B
~$
```

MEMINFO: The /proc/meminfo file contains detailed information about the system's memory usage.

```
~$ cat /proc/meminfo
MemTotal:        32860440 kB
MemFree:          7680552 kB
MemAvailable:    27964428 kB
Buffers:           312760 kB
Cached:          20666672 kB
SwapCached:             0 kB
Active:           4978056 kB
Inactive:        17709272 kB
Active(anon):        5264 kB
Inactive(anon):   1590456 kB
Active(file):     4972792 kB
Inactive(file):  16118816 kB
Unevictable:        18536 kB
Mlocked:            18536 kB
SwapTotal:              0 kB
SwapFree:               0 kB
Dirty:                 80 kB
Writeback:              0 kB
AnonPages:        1726552 kB
Mapped:            638064 kB
Shmem:               5364 kB
KReclaimable:     1420684 kB
Slab:             1757088 kB
SReclaimable:     1420684 kB
SUnreclaim:        336404 kB
```