*** SLIP 1**

Q.1 Write a program that demonstrates the use of nice() system call. After a child process is started using fork(), assign higher priority to the child using nice() system call.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
pid_t child_pid;
int nice_value = -10;
child_pid = fork();
if (child_pid == 0) {
printf("Child process (PID %d) is running with normal priority.\n", getpid());
} else if (child_pid > 0) {
printf("Parent process (PID %d) is running with normal priority.\n", getpid());
if (nice(nice_value) == -1) {
perror("nice");
exit(EXIT_FAILURE);
}
printf("Parent process has adjusted the child process's priority to be higher.\n");
} else {
perror("fork");
exit(EXIT_FAILURE);
}
sleep(5);
return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page schedulingand total number of page faults for the following given page reference string. Give input n=3 as

the number of memory frames.

Reference String :3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

Implement FIFO

```
#include <stdio.h>
#define MAX_FRAMES 3
int main() {
int referenceString[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
int n = sizeof(referenceString) / sizeof(referenceString[0]);
int memoryFrames[MAX_FRAMES];
int pageFaults = 0;
int frameIndex = 0;
for (int i = 0; i < n; i++) {
int page = referenceString[i];
int pageFound = 0;
for (int j = 0; j < MAX_FRAMES; j++) {
if (memoryFrames[j] == page) {
pageFound = 1;
break;
}
}
if (!pageFound) {
pageFaults++;
if (frameIndex < MAX_FRAMES) {
memoryFrames[frameIndex] = page;
frameIndex++;
} else {
for (int j = 0; j < MAX_FRAMES - 1; j++) {
memoryFrames[j] = memoryFrames[j + 1];
}
```

```
memoryFrames[MAX_FRAMES - 1] = page;
}
}
printf("Page %d: [", page);
for (int j = 0; j < MAX_FRAMES; j++) {
printf("%d", memoryFrames[j]);
if (j < MAX_FRAMES - 1) {
printf(", ");
}
}
printf("]\n");
}
printf("Total page faults: %d\n", pageFaults);
return 0;
}
```

*** SLIP 2**

Q.1 Create a child process using fork(), display parent and child process id. Child process will display the message “Hello World” and the parent process should display “Hi”.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
pid_t child_pid;
child_pid = fork();
if (child_pid == 0) {
printf("Child Process ID: %d\n", getpid());
printf("Hello World\n");
} else if (child_pid > 0) {
printf("Parent Process ID: %d\n", getpid());
printf("Hi\n");
} else {
perror("fork");
exit(EXIT_FAILURE);
}
return 0;
}
```

Q.2 Write the simulation program using SJF (non-preemptive). The arrival time and first CPU

bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units).

The next CPU burst should be generated using random function. The output should give the Gantt

chart, Turnaround Time and Waiting time for each process and average times.[20

marks]

program not working properly

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
struct Process {
int id;
int arrivalTime;
int initialCpuBurst;
int remainingCpuBurst;
int turnaroundTime;
int waitingTime;
};
void sortByArrivalTime(struct Process processes[], int n) {
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (processes[i].arrivalTime > processes[j].arrivalTime) {
struct Process temp = processes[i];
processes[i] = processes[j];
processes[j] = temp;
}
}
}
}
int main() {
srand(time(NULL));
int n;
printf("Enter the number of processes: ");
```

```
scanf("%d", &n);
struct Process processes[n];
for (int i = 0; i < n; i++) {
processes[i].id = i;
printf("Enter arrival time for process P%d: ", i);
scanf("%d", &processes[i].arrivalTime);
printf("Enter initial CPU burst for process P%d: ", i);
scanf("%d", &processes[i].initialCpuBurst);
processes[i].remainingCpuBurst = processes[i].initialCpuBurst;
}

sortByArrivalTime(processes, n);
int currentTime = 0;
int totalTurnaroundTime = 0;
int totalWaitingTime = 0;
printf("\nGantt Chart:\n");
while (1) {
int shortestJobIndex = -1;
int shortestJobBurst = INT_MAX;
for (int i = 0; i < n; i++) {
if (processes[i].arrivalTime <= currentTime && processes[i].remainingCpuBurst <
shortestJobBurst && processes[i].remainingCpuBurst > 0) {
shortestJobIndex = i;
shortestJobBurst = processes[i].remainingCpuBurst;
}
}
if (shortestJobIndex == -1) {
currentTime++;
continue;
}
```

```
processes[shortestJobIndex].remainingCpuBurst--;

printf("P%d ", processes[shortestJobIndex].id);

if (processes[shortestJobIndex].remainingCpuBurst == 0) {

int finishTime = currentTime + 1;

processes[shortestJobIndex].turnaroundTime = finishTime -
processes[shortestJobIndex].arrivalTime;

processes[shortestJobIndex].waitingTime = processes[shortestJobIndex].turnaroundTime -
processes[shortestJobIndex].initialCpuBurst;

totalTurnaroundTime += processes[shortestJobIndex].turnaroundTime;

totalWaitingTime += processes[shortestJobIndex].waitingTime;

}

currentTime++;

}


printf("\n");

printf("\nProcess\tTurnaround Time\tWaiting Time\n");

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

printf("P%d\t%d\t\t%d\n", i, processes[i].turnaroundTime, processes[i].waitingTime);

}

float averageTurnaroundTime = (float)totalTurnaroundTime / n;

float averageWaitingTime = (float)totalWaitingTime / n;

printf("\nAverage Turnaround Time: %.2f\n", averageTurnaroundTime);

printf("Average Waiting Time: %.2f\n", averageWaitingTime);

return 0;

}
```

*** SLIP 3**

Q. 1 Creating a child process using the command exec(). Note down process ids of the parent

and the child processes, check whether the control is given back to the parent after the child

process terminates.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
pid_t child_pid;
child_pid = fork();
if (child_pid < 0) {
perror("Fork failed");
exit(1);
}
if (child_pid == 0) {
printf("Child Process: My PID is %d\n", getpid());
printf("Child Process: Executing 'ls' command...\n");
execlp("ls", "ls", (char *)NULL);
perror("execlp");
exit(1);
} else {
printf("Parent Process: My PID is %d\n", getpid());
printf("Parent Process: Waiting for the child to finish...\n");
wait(NULL);
printf("Parent Process: Child has finished.\n");
```

}

return 0;

}

_____

Q.2 Write the simulation program using FCFS. The arrival time and first CPU bursts of different

jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst

should be generated using random function. The output should give the Gantt chart,Turnaround

Time and Waiting time for each process and average times. [20 marks]

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct Process {
int id;
int arrivalTime;
int initialCpuBurst;
int remainingCpuBurst;
int turnaroundTime;
int waitingTime;
};
void sortByArrivalTime(struct Process processes[], int n) {
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (processes[i].arrivalTime > processes[j].arrivalTime) {
struct Process temp = processes[i];
processes[i] = processes[j];
```

```
processes[j] = temp;
}
}
}
}
int main() {
srand(time(NULL));
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
struct Process processes[n];
for (int i = 0; i < n; i++) {
processes[i].id = i;
printf("Enter arrival time for process P%d: ", i);
scanf("%d", &processes[i].arrivalTime);
printf("Enter initial CPU burst for process P%d: ", i);
scanf("%d", &processes[i].initialCpuBurst);
processes[i].remainingCpuBurst = processes[i].initialCpuBurst;
}
sortByArrivalTime(processes, n);
int currentTime = 0;
int totalTurnaroundTime = 0;
int totalWaitingTime = 0;
printf("\nGantt Chart:\n");
printf("0 "); // Print the initial time
for (int i = 0; i < n; i++) {
if (processes[i].arrivalTime > currentTime) {
currentTime = processes[i].arrivalTime;
}
```

```
printf("| P%d ", processes[i].id);
processes[i].waitingTime = currentTime - processes[i].arrivalTime;
totalWaitingTime += processes[i].waitingTime;
currentTime += processes[i].remainingCpuBurst;
processes[i].turnaroundTime = currentTime - processes[i].arrivalTime;
totalTurnaroundTime += processes[i].turnaroundTime;
}
printf("|\n");
printf("\nProcess\tTurnaround Time\tWaiting Time\n");
for (int i = 0; i < n; i++) {
printf("P%d\t%d\t\t%d\n", i, processes[i].turnaroundTime, processes[i].waitingTime);
}
float averageTurnaroundTime = (float)totalTurnaroundTime / n;
float averageWaitingTime = (float)totalWaitingTime / n;
printf("\nAverage Turnaround Time: %.2f\n", averageTurnaroundTime);
printf("Average Waiting Time: %.2f\n", averageWaitingTime);
return 0;
```

*** SLIP 4**

Q.1 Write a program to illustrate the concept of orphan process (Using fork() and sleep()) [10 marks]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
pid_t child_pid;
printf("Parent Process (PID %d) is starting...\n", getpid());
child_pid = fork();
if (child_pid < 0) {
perror("Fork failed");
exit(1);
}
if (child_pid == 0) {
printf("Child Process (PID %d) is running...\n", getpid());
sleep(5);
printf("Child Process (PID %d) has finished.\n", getpid());
} else {
printf("Parent Process (PID %d) is running...\n", getpid());
sleep(2);
printf("Parent Process (PID %d) has finished.\n", getpid());
}
return 0;
}
```

Q.2 Write the program to simulate Non-preemptive Priority scheduling. The arrival time and first

CPU burst and priority for different n number of processes should be input to the algorithm.

Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly.

The output should give Gantt chart, turnaround time and waiting time for each process. Also find

the average waiting time and turnaround time..

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct Process {
int id;
int arrivalTime;
int initialCpuBurst;
int remainingCpuBurst;
int priority;
int turnaroundTime;
int waitingTime;
};
void sortByPriority(struct Process processes[], int n) {
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (processes[i].priority > processes[j].priority) {
struct Process temp = processes[i];
processes[i] = processes[j];
processes[j] = temp;
}
}
}
```

```
}

int main() {
srand(time(NULL));
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
struct Process processes[n];
for (int i = 0; i < n; i++) {
processes[i].id = i;
printf("Enter arrival time for process P%d: ", i);
scanf("%d", &processes[i].arrivalTime);
printf("Enter initial CPU burst for process P%d: ", i);
scanf("%d", &processes[i].initialCpuBurst);
printf("Enter priority for process P%d: ", i);
scanf("%d", &processes[i].priority);
processes[i].remainingCpuBurst = processes[i].initialCpuBurst;
}
sortByPriority(processes, n);
int currentTime = 0;
int totalTurnaroundTime = 0;
int totalWaitingTime = 0;
printf("\nGantt Chart:\n");
printf("0 "); // Print the initial time
for (int i = 0; i < n; i++) {
if (processes[i].arrivalTime > currentTime) {
currentTime = processes[i].arrivalTime;
}
printf("| P%d ", processes[i].id);
```

```
processes[i].waitingTime = currentTime - processes[i].arrivalTime;
totalWaitingTime += processes[i].waitingTime;
int randomBurst = (rand() % 5) + 1;
processes[i].remainingCpuBurst -= randomBurst;
currentTime += randomBurst;
processes[i].turnaroundTime = currentTime - processes[i].arrivalTime;
totalTurnaroundTime += processes[i].turnaroundTime;
}
printf("|\n");
printf("\nProcess\tTurnaround Time\tWaiting Time\n");
for (int i = 0; i < n; i++) {
printf("P%d\t%d\t\t%d\n", i, processes[i].turnaroundTime, processes[i].waitingTime);
float averageTurnaroundTime = (float)totalTurnaroundTime / n;
float averageWaitingTime = (float)totalWaitingTime / n;
printf("\nAverage Turnaround Time: %.2f\n", averageTurnaroundTime);
printf("Average Waiting Time: %.2f\n", averageWaitingTime);
return 0;
}
```

*** SLIP 5**

Q.1 Write a program that demonstrates the use of nice () system call. After a child process is started using fork (), assign higher priority to the child using nice () system call.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
int main() {
pid_t child_pid;
int nice_value = -10;
printf("Parent Process (PID %d) is starting...\n", getpid());
child_pid = fork();
if (child_pid < 0) {
perror("Fork failed");
exit(1);
}
if (child_pid == 0) {
printf("Child Process (PID %d) is running...\n", getpid());
if (nice(nice_value) == -1) {
perror("Nice failed");
exit(1);
}
printf("Child Process (PID %d) has a higher priority (nice value: %d).\n", getpid(),
nice_value);
} else {
printf("Parent Process (PID %d) is running...\n", getpid());
sleep(2);
printf("Parent Process (PID %d) has finished.\n", getpid());
```

```
}
return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling

and total number of page faults for the following given page reference string. Give input n as the number of memory frames. Reference String: 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

i. Implement FIFO

```
#include <stdio.h>
int main() {
int page_reference_string[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
int n, i;
printf("Enter the number of memory frames: ");
scanf("%d", &n);
int memory_frames[n];
int page_queue[n];
int page_faults = 0;
for (i = 0; i < n; i++) {
memory_frames[i] = -1;
page_queue[i] = -1;
}
for (i = 0; i < sizeof(page_reference_string) / sizeof(page_reference_string[0]); i++) {
int page = page_reference_string[i];
int found = 0;
for (int j = 0; j < n; j++) {
if (memory_frames[j] == page) {
found = 1;
```

```
break;
}
}

if (!found) {
page_faults++;
if (page_queue[0] != -1) {
int replaced_page = page_queue[0];
for (int j = 0; j < n; j++) {
if (memory_frames[j] == replaced_page) {
memory_frames[j] = page;
break;
}
}
for (int j = 0; j < n - 1; j++) {
page_queue[j] = page_queue[j + 1];
}
} else {
for (int j = 0; j < n; j++) {
if (memory_frames[j] == -1) {
memory_frames[j] = page;
break;
}
}
}
for (int j = 0; j < n - 1; j++) {
page_queue[j] = page_queue[j + 1];
}
page_queue[n - 1] = page;
```

```
}
printf("Page Reference: %d\n", page);
printf("Memory Frames: ");
for (int j = 0; j < n; j++) {
printf("%d ", memory_frames[j]);
}
printf("\nPage Queue: ");
for (int j = 0; j < n; j++) {
printf("%d ", page_queue[j]);
}
printf("\n\n");
}
printf("Total Page Faults: %d\n", page_faults);

return 0;
}
```

*** SLIP 6**

Q.1 Write a program to find the execution time taken for execution of a given set of instructions (use clock() function

```
#include <stdio.h>
#include <time.h>
int main() {
clock_t start_time = clock();
for (int i = 0; i < 1000000; i++) {
}
clock_t end_time = clock();
double execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("Execution time: %lf seconds\n", execution_time);
return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.

Reference String :3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

Implement FIFO

```
#include <stdio.h>
#include <stdbool.h>
bool isPagePresent(int memoryFrames[], int n, int page) {
for (int i = 0; i < n; i++) {
if (memoryFrames[i] == page) {
return true;
}
}
```

```
return false;
}
int main() {
int pageReferenceString[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
int n;
printf("Enter the number of memory frames: ");
scanf("%d", &n);
int memoryFrames[n];
int pageQueue[n];
int pageFaults = 0;
int queueIndex = 0;
for (int i = 0; i < n; i++) {
memoryFrames[i] = -1;
pageQueue[i] = -1;
}
for (int i = 0; i < sizeof(pageReferenceString) / sizeof(pageReferenceString[0]); i++) {
int page = pageReferenceString[i];
if (!isPagePresent(memoryFrames, n, page)) {
pageFaults++;
int victimPage = pageQueue[queueIndex];
for (int j = 0; j < n; j++) {
if (memoryFrames[j] == victimPage) {
memoryFrames[j] = page;
break;
}
}
pageQueue[queueIndex] = page;
queueIndex = (queueIndex + 1) % n;
}
```

```
printf("Page Reference: %d\n", page);
printf("Memory Frames: ");
for (int j = 0; j < n; j++) {
printf("%d ", memoryFrames[j]);
}
printf("\nPage Queue: ");
for (int j = 0; j < n; j++) {
printf("%d ", pageQueue[j]);
}
printf("\n\n");
}
printf("Total Page Faults: %d\n", pageFaults);
return 0;
}
```

*** SLIP 7**

Q.1 Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

```
#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid_t pid = fork();

if (pid == -1) {

perror("Fork failed");

return 1;

}

if (pid == 0) {

printf("Child process is executing the 'ls' command:\n");

execl("/bin/ls", "ls", NULL);

perror("execl");

} else {

printf("Parent process is going to sleep...\n");

sleep(2);

printf("Parent process woke up from sleep.\n");

int status;

wait(&status);

if (WIFEXITED(status)) {

printf("Child process exited with status %d\n", WEXITSTATUS(status));

}

}

return 0;

}
```

Q.2 Write the simulation program using FCFS. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef struct {
int arrival_time;
int first_cpu_burst;
int next_cpu_burst;
int turnaround_time;
int waiting_time;
} Job;
void calculateAverages(Job jobs[], int n, float *avg_turnaround, float *avg_waiting) {
int total_turnaround = 0;
int total_waiting = 0;
for (int i = 0; i < n; i++) {
total_turnaround += jobs[i].turnaround_time;
total_waiting += jobs[i].waiting_time;
}
*avg_turnaround = (float)total_turnaround / n;
*avg_waiting = (float)total_waiting / n;
}
int main() {
int n;
srand(time(NULL));
printf("Enter the number of jobs: ");
```

```
scanf("%d", &n);

Job jobs[n];
for (int i = 0; i < n; i++) {
printf("Job %d\n", i + 1);
printf("Arrival Time: ");
scanf("%d", &jobs[i].arrival_time);
printf("First CPU Burst Time: ");
scanf("%d", &jobs[i].first_cpu_burst);
jobs[i].next_cpu_burst = 0;
jobs[i].turnaround_time = 0;
jobs[i].waiting_time = 0;
}
int current_time = 0;
for (int i = 0; i < n; i++) {
if (current_time < jobs[i].arrival_time) {
current_time = jobs[i].arrival_time;
}
jobs[i].next_cpu_burst = rand() % 10 + 1; // Generating random burst time between 1 and 10 units

jobs[i].turnaround_time = current_time + jobs[i].first_cpu_burst + jobs[i].next_cpu_burst - jobs[i].arrival_time;
jobs[i].waiting_time = jobs[i].turnaround_time - jobs[i].first_cpu_burst - jobs[i].next_cpu_burst;
current_time += jobs[i].first_cpu_burst + jobs[i].next_cpu_burst;
}
printf("\nGantt Chart:\n");
printf("0");
for (int i = 0; i < n; i++) {
```

```
printf(" -> Job%d -> %d", i + 1, current_time);
}
printf("\n");
printf("\nTurnaround Times and Waiting Times:\n");
for (int i = 0; i < n; i++) {
printf("Job%d - Turnaround Time: %d, Waiting Time: %d\n", i + 1, jobs[i].turnaround_time, jobs[i].waiting_time);
}
float avg_turnaround, avg_waiting;
calculateAverages(jobs, n, &avg_turnaround, &avg_waiting);
printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround);
printf("Average Waiting Time: %.2f\n", avg_waiting);
return 0;
}
```

*** SLIP 8**

Q.1 Write a C program to accept the number of process and resources and find the need matrix content and display it

```
#include <stdio.h>
void calculateNeedMatrix(int need[][10], int max[][10], int allocation[][10], int n, int m) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++) {
need[i][j] = max[i][j] - allocation[i][j];
}
}
}
void displayMatrix(int matrix[][10], int n, int m) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++) {
printf("%d\t", matrix[i][j]);
}
printf("\n");
}
}
int main() {
int n, m;
printf("Enter the number of processes: ");
scanf("%d", &n);
printf("Enter the number of resources: ");
scanf("%d", &m);
int allocation[10][10], max[10][10], need[10][10];
printf("Enter the Allocation Matrix:\n");
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++) {
```

```
scanf("%d", &allocation[i][j]);
}
}
printf("Enter the Maximum Matrix:\n");
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++) {
scanf("%d", &max[i][j]);
}
}
calculateNeedMatrix(need, max, allocation, n, m);
printf("Need Matrix:\n");
displayMatrix(need, n, m);
return 0;
}
```

---

or Q.2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n =3 as the number of memory frames.

Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8

Implement OPT

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_FRAMES 3
void initialize(int frames[], int n) {
for (int i = 0; i < n; i++) {
frames[i] = -1;
}
```

```
}
bool isPageInMemory(int page, int frames[], int n) {
for (int i = 0; i < n; i++) {
if (frames[i] == page) {
return true;
}
}
return false;
}
int findOptimalPage(int pageReference[], int frames[], int n, int currentPageIndex) {
int farthestIndex = -1;
int farthestPage = -1;
for (int i = 0; i < n; i++) {
int page = frames[i];
bool isFuture = false;
for (int j = currentPageIndex; j < MAX_FRAMES; j++) {
if (page == pageReference[j]) {
isFuture = true;
break;
}
}
if (!isFuture) {
return i;
}
}
return (farthestIndex == -1) ? 0 : farthestIndex;
}
int main() {
int pageReference[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
```

```
int frames[MAX_FRAMES];
int pageFaults = 0;
initialize(frames, MAX_FRAMES);
for (int i = 0; i < MAX_FRAMES; i++) {
printf("Frame %d: ", i);
for (int j = 0; j < MAX_FRAMES; j++) {
if (frames[j] != -1) {
printf("%d ", frames[j]);
}
}
printf("\n");
}
for (int i = 0; i < MAX_FRAMES; i++) {
printf("-------");
}
printf("\n");
for (int i = 0; i < MAX_FRAMES; i++) {
printf("  Page ");
}
printf("\n");
for (int i = 0; i < MAX_FRAMES; i++) {
printf("-------");
}
printf("\n");
for (int i = 0; i < MAX_FRAMES; i++) {
printf("       ");
}
printf("\n");
int currentPageIndex = MAX_FRAMES;
```

```
for (int i = MAX_FRAMES; i < sizeof(pageReference) / sizeof(pageReference[0]); i++) {
int currentPage = pageReference[i];
if (!isPageInMemory(currentPage, frames, MAX_FRAMES)) {
int pageToReplace = findOptimalPage(pageReference, frames, MAX_FRAMES, i);
frames[pageToReplace] = currentPage;
pageFaults++;
}
for (int j = 0; j < MAX_FRAMES; j++) {
printf("  %d    ", frames[j]);
}
printf("\n");
currentPageIndex++;
if (i < sizeof(pageReference) / sizeof(pageReference[0]) - 1) {
for (int j = 0; j < MAX_FRAMES; j++) {
if (frames[j] == pageReference[currentPageIndex]) {
frames[j] = -1;
}
}
}
}
printf("Total Page Faults: %d\n", pageFaults);
return 0;
}
```

*** SLIP 9**

Q.1 Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
pid_t pid = fork(); // Create a child process
if (pid == -1) {
perror("Fork failed");
exit(1);
}
if (pid == 0) {
printf("Child process is executing the 'ls' command:\n");
execl("/bin/ls", "ls", NULL);
perror("execl"); // This will be printed if execl() fails
exit(1); // Terminate the child process
} else {
printf("Parent process is going to sleep...\n");
sleep(2); // Sleep for 2 seconds
printf("Parent process woke up from sleep.\n");
int status;
wait(&status);
if (WIFEXITED(status)) {
printf("Child process exited with status %d\n", WEXITSTATUS(status));
}
}
```

```
return 0;
}
```

Q.2 Write the program to simulate Round Robin (RR) scheduling. The arrival time and first CPUburst for different n number of processes should be input to the algorithm. Also give the time quantum as input. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct Process {
int arrival_time;
int first_cpu_burst;
int next_cpu_burst;
int remaining_time;
int turnaround_time;
int waiting_time;
int executed;
};
int main() {
int n, time_quantum;
srand(time(NULL));
printf("Enter the number of processes: ");
scanf("%d", &n);
printf("Enter the time quantum: ");
scanf("%d", &time_quantum);
struct Process processes[n];
for (int i = 0; i < n; i++) {
printf("Process %d\n", i + 1);
```

```
printf("Arrival Time: ");
scanf("%d", &processes[i].arrival_time);
printf("First CPU Burst Time: ");
scanf("%d", &processes[i].first_cpu_burst);
processes[i].next_cpu_burst = 0;
processes[i].remaining_time = processes[i].first_cpu_burst;
processes[i].turnaround_time = 0;
processes[i].waiting_time = 0;
processes[i].executed = 0;
}
int current_time = 0;
int total_turnaround_time = 0;
int total_waiting_time = 0;
int executed_processes = 0;
printf("\nGantt Chart:\n");
while (executed_processes < n) {
for (int i = 0; i < n; i++) {
if (processes[i].arrival_time <= current_time && processes[i].remaining_time > 0) {
int execute_time = (processes[i].remaining_time > time_quantum) ? time_quantum :
processes[i].remaining_time;

processes[i].next_cpu_burst = rand() % 10 + 1;
printf("P%d (%d-%d) ", i + 1, current_time, current_time + execute_time);
processes[i].remaining_time -= execute_time;
current_time += execute_time;
if (processes[i].remaining_time == 0) {
processes[i].turnaround_time = current_time - processes[i].arrival_time;
processes[i].waiting_time = processes[i].turnaround_time - processes[i].first_cpu_burst;
total_turnaround_time += processes[i].turnaround_time;
total_waiting_time += processes[i].waiting_time;
```

```
executed_processes++;
}
}
}
}
printf("\n\nTurnaround Time and Waiting Time:\n");
for (int i = 0; i < n; i++) {
printf("Process %d - Turnaround Time: %d, Waiting Time: %d\n", i + 1,
processes[i].turnaround_time, processes[i].waiting_time);
}
printf("\nAverage Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
return 0;
}
```

*** SLIP 10**

Q.1 Write a program to illustrate the concept of orphan process (Using fork() and sleep())

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
pid_t child_pid = fork(); // Create a child process
if (child_pid == -1) {
perror("Fork failed");
exit(1);
}
if (child_pid == 0) {
printf("Child process (PID: %d) is running.\n", getpid());
sleep(5);
printf("Child process (PID: %d) is done.\n", getpid());
} else {
printf("Parent process (PID: %d) is running.\n", getpid());
printf("Parent process is terminating.\n");
exit(0);
}
return 0;
}
```

Q.2 Write the simulation program using FCFS. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times.

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

```
#include <stdlib.h>
#include <time.h>
struct Process {
int id;
int arrivalTime;
int burstTime;
int turnaroundTime;
int waitingTime;
};
int main() {
srand(time(NULL));
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
struct Process processes[n];
for (int i = 0; i < n; i++) {
processes[i].id = i + 1;
printf("Enter Arrival Time for Process %d: ", i + 1);
scanf("%d", &processes[i].arrivalTime);
printf("Enter Initial Burst Time for Process %d: ", i + 1);
scanf("%d", &processes[i].burstTime);
}
int currentTime = 0;
int totalTurnaroundTime = 0;
int totalWaitingTime = 0;
printf("\nGantt Chart:\n");
printf("0");
for (int i = 0; i < n; i++) {
printf(" --> P%d", processes[i].id);
```

```
currentTime += 2;
processes[i].waitingTime = currentTime - processes[i].arrivalTime;
processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;
totalTurnaroundTime += processes[i].turnaroundTime;
totalWaitingTime += processes[i].waitingTime;
int nextBurst = rand() % 10 + 1;
processes[i].burstTime = nextBurst;
currentTime += processes[i].burstTime;
}
printf("\n\nProcess\tTurnaround Time\tWaiting Time\n");
for (int i = 0; i < n; i++) {
printf("P%d\t%d\t\t%d\n", processes[i].id, processes[i].turnaroundTime,
processes[i].waitingTime);
}
double avgTurnaroundTime = (double)totalTurnaroundTime / n;
double avgWaitingTime = (double)totalWaitingTime / n;
printf("\nAverage Turnaround Time: %.2lf\n", avgTurnaroundTime);
printf("Average Waiting Time: %.2lf\n", avgWaitingTime);
return 0;
}
```

*** SLIP 11**

Q.1 Create a child process using fork(), display parent and child process id. Child process will display the message “Hello World” and the parent process should display “Hi”.

```
#include <stdio.h>
#include <unistd.h>
int main() {
pid_t child_pid = fork(); // Create a child process
if (child_pid < 0) {
perror("Fork failed");
return 1;
}
if (child_pid == 0) {
printf("Child Process (PID: %d) says: Hello World\n", getpid());
} else {
printf("Parent Process (PID: %d) says: Hi\n", getpid());
}
return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling

and total number of page faults for the following given page reference string. Give input n as the

number of memory frames.

Reference String: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Implement FIFO

output:

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

```
#include <stdlib.h>
int isPageInMemory(int page, int* memory, int n) {
for (int i = 0; i < n; i++) {
if (memory[i] == page) {
return 1;
}
}
return 0;
}
void displayMemory(int* memory, int n) {
for (int i = 0; i < n; i++) {
printf("%d ", memory[i]);
}
printf("\n");
}
int main() {
int n;
printf("Enter the number of memory frames: ");
scanf("%d", &n);
int* memory = (int*)malloc(n * sizeof(int)); // Memory frames
int* pageReferenceString;
printf("Enter the number of pages in the reference string: ");
int numPages;
scanf("%d", &numPages);
pageReferenceString = (int*)malloc(numPages * sizeof(int));
printf("Enter the page reference string: ");
for (int i = 0; i < numPages; i++) {
scanf("%d", &pageReferenceString[i]);
}
```

```
int pageFaults = 0;
int index = 0;
for (int i = 0; i < n; i++) {
memory[i] = -1;
}
printf("\nPage Scheduling using FIFO:\n");
for (int i = 0; i < numPages; i++) {
printf("Page Reference: %d\n", pageReferenceString[i]);
if (!isPageInMemory(pageReferenceString[i], memory, n)) {
pageFaults++;
memory[index] = pageReferenceString[i];
index = (index + 1) % n;
displayMemory(memory, n);
} else {
printf("Page %d is already in memory.\n", pageReferenceString[i]);
}
}
printf("\nTotal Page Faults: %d\n", pageFaults);
free(memory);
free(pageReferenceString);
return 0;
}
```

*** SLIP 12**

Q.1 [10] Write a program to illustrate the concept of orphan process (Using fork() and sleep()) .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
pid_t child_pid = fork(); // Create a child process
if (child_pid == -1) {
perror("Fork failed");
exit(1);
}
if (child_pid == 0) {
printf("Child process (PID: %d) is running.\n", getpid());
sleep(5);
printf("Child process (PID: %d) is done.\n", getpid());
} else {
printf("Parent process (PID: %d) is running.\n", getpid());
printf("Parent process is terminating.\n");
exit(0);
}
return 0;
}
```

---

or Q2. Write the simulation program using FCFS. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times

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

```
#include <stdlib.h>
#include <time.h>
struct Process {
int id;
int arrivalTime;
int cpuBurstTime;
int ioBurstTime;
int turnaroundTime;
int waitingTime;
};
int generateRandomCPUBurstTime() {
return (rand() % 10) + 1;
}
int generateRandomIOBurstTime() {
return (rand() % 5) + 1;
}
int main() {
srand(time(NULL));
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
struct Process processes[n];
for (int i = 0; i < n; i++) {
processes[i].id = i + 1;
printf("Enter Arrival Time for Process %d: ", i + 1);
scanf("%d", &processes[i].arrivalTime);
processes[i].cpuBurstTime = generateRandomCPUBurstTime();
processes[i].ioBurstTime = generateRandomIOBurstTime();
}
```

```
int currentTime = 0;
int totalTurnaroundTime = 0;
int totalWaitingTime = 0;
printf("\nGantt Chart:\n");
for (int i = 0; i < n; i++) {
if (processes[i].arrivalTime > currentTime) {
currentTime = processes[i].arrivalTime;
}
printf("P%d ", processes[i].id);
currentTime += processes[i].cpuBurstTime;
processes[i].waitingTime = currentTime - processes[i].arrivalTime -
processes[i].cpuBurstTime;
processes[i].turnaroundTime = processes[i].waitingTime + processes[i].cpuBurstTime;
totalTurnaroundTime += processes[i].turnaroundTime;
totalWaitingTime += processes[i].waitingTime;
currentTime += processes[i].ioBurstTime;
}
printf("\n\nProcess\tTurnaround Time\tWaiting Time\n");
for (int i = 0; i < n; i++) {
printf("P%d\t%d\t\t%d\n", processes[i].id, processes[i].turnaroundTime,
processes[i].waitingTime);
}
double avgTurnaroundTime = (double)totalTurnaroundTime / n;
double avgWaitingTime = (double)totalWaitingTime / n;
printf("\nAverage Turnaround Time: %.2lf\n", avgTurnaroundTime);
printf("Average Waiting Time: %.2lf\n", avgWaitingTime);
return 0;
}
```

*** SLIP 13**

Q.1 Write a program that demonstrates the use of nice() system call. After a child process is started using fork(), assign higher priority to the child using nice() system call.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>
int main() {
pid_t child_pid = fork();
if (child_pid == -1) {
perror("Fork failed");
exit(1);
}
if (child_pid == 0) {
printf("Child process (PID: %d) is running with default priority.\n", getpid());
int nice_value = nice(10);
if (nice_value == -1) {
perror("Nice failed");
} else {
printf("Child process (PID: %d) has a higher priority (lower nice value: %d).\n", getpid(), nice_value);
}
} else {
printf("Parent process (PID: %d) is running with default priority.\n", getpid());
printf("Parent process is not changing its priority.\n");
}
return 0;
}
```

---

or Write the simulation program using SJF(non-preemptive). The arrival time and first CPU bursts of different jobsshould be input to the system. The Assume the fixed I/O waiting time (2 units).Thenext CPU burst should be generated using random function. The output should give the Gantt chart,Turnaround Time and Waiting time for each process and average times.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct Process {
int id;         // Process ID
int arrivalTime; // Arrival Time
int cpuBurstTime; // CPU Burst Time
int ioBurstTime;  // I/O Burst Time
int turnaroundTime;
int waitingTime;
};
int generateRandomCPUBurstTime() {
return (rand() % 10) + 1;
}
int main() {
srand(time(NULL));
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
struct Process processes[n];
for (int i = 0; i < n; i++) {
processes[i].id = i + 1;
printf("Enter Arrival Time for Process %d: ", i + 1);
scanf("%d", &processes[i].arrivalTime);
processes[i].cpuBurstTime = generateRandomCPUBurstTime();
```

```
processes[i].ioBurstTime = 2; // Fixed I/O burst time (2 units)
}
int currentTime = 0;
int totalTurnaroundTime = 0;
int totalWaitingTime = 0;
printf("\nGantt Chart:\n");
for (int i = 0; i < n; i++) {
if (processes[i].arrivalTime > currentTime) {
currentTime = processes[i].arrivalTime;
}
printf("P%d ", processes[i].id);
currentTime += processes[i].cpuBurstTime;
processes[i].waitingTime = currentTime - processes[i].arrivalTime -
processes[i].cpuBurstTime;
processes[i].turnaroundTime = processes[i].waitingTime + processes[i].cpuBurstTime;
totalTurnaroundTime += processes[i].turnaroundTime;
totalWaitingTime += processes[i].waitingTime;
}
printf("\n\nProcess\tTurnaround Time\tWaiting Time\n");
for (int i = 0; i < n; i++) {
printf("P%d\t%d\t\t%d\n", processes[i].id, processes[i].turnaroundTime,
processes[i].waitingTime);
}
double avgTurnaroundTime = (double)totalTurnaroundTime / n;
double avgWaitingTime = (double)totalWaitingTime / n;
printf("\nAverage Turnaround Time: %.2lf\n", avgTurnaroundTime);
printf("Average Waiting Time: %.2lf\n", avgWaitingTime);
return 0;
}
```

_____

SLIP 14

Q.1 Write a program to find the execution time taken for execution of a given set of instructions (use clock() function

```
#include <stdio.h>
#include <time.h>
int main() {
clock_t start, end;
double cpu_time_used;
start = clock();
for (int i = 0; i < 1000000; i++) {
}
end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Execution time: %f seconds\n", cpu_time_used);
return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n =3 as the number of memory frames.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Implement FIFO

```
#include <stdio.h>
#include <stdlib.h>
int isPageInMemory(int page, int* memory, int n) {
for (int i = 0; i < n; i++) {
if (memory[i] == page) {
return 1;
}
```

```
}
return 0;
}
int main() {
int n = 3; // Number of memory frames
int pageReferenceString[] = {0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1};
int numPages = sizeof(pageReferenceString) / sizeof(pageReferenceString[0]);
int memory[n];
int pageFaults = 0;
int currentIndex = 0;
for (int i = 0; i < n; i++) {
memory[i] = -1;
}
printf("Page Scheduling using FIFO:\n");
for (int i = 0; i < numPages; i++) {
int currentPage = pageReferenceString[i];
if (!isPageInMemory(currentPage, memory, n)) {
pageFaults++;
memory[currentIndex] = currentPage;
currentIndex = (currentIndex + 1) % n;
for (int j = 0; j < n; j++) {
printf("%d ", memory[j]);
}
printf("\n");
}
}
printf("\nTotal Page Faults: %d\n", pageFaults);
return 0;
}
```

_____

SLIP 15

Q.1 Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
pid_t child_pid = fork(); // Create a child process
if (child_pid == -1) {
perror("Fork failed");
exit(1);
}
if (child_pid == 0) {
printf("Child process (PID: %d) is running.\n", getpid());
if (execl("/bin/ls", "ls", (char *)NULL) == -1) {
perror("Execl failed");
exit(1);
}
} else {
printf("Parent process (PID: %d) is going to sleep for 5 seconds.\n", getpid());
sleep(5);
printf("Parent process (PID: %d) woke up from sleep.\n", getpid());
}
return 0;
}
```

_____

Q.2 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames. Reference String :7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 : Implement LRU

```
#include <stdio.h>
#define MAX_PAGES 100
int isPageInMemory(int page, int* memory, int n) {
for (int i = 0; i < n; i++) {
if (memory[i] == page) {
return 1;
}
}
return 0;
}
int findLRUIndex(int* memory, int* pageUsage, int n) {
int minIndex = 0;
int minUsage = pageUsage[0];
for (int i = 1; i < n; i++) {
if (pageUsage[i] < minUsage) {
minIndex = i;
minUsage = pageUsage[i];
}
}
return minIndex;
}
int main() {
int n = 3; // Number of memory frames
int pageReferenceString[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
int numPages = sizeof(pageReferenceString) / sizeof(pageReferenceString[0]);
```

```
int memory[n];
int pageUsage[n];
int pageFaults = 0;
for (int i = 0; i < n; i++) {
memory[i] = -1;
pageUsage[i] = 0;
}
printf("Page Scheduling using LRU:\n");
for (int i = 0; i < numPages; i++) {
int currentPage = pageReferenceString[i];
if (!isPageInMemory(currentPage, memory, n)) {
pageFaults++;
int lruIndex = findLRUIndex(memory, pageUsage, n);
memory[lruIndex] = currentPage;
pageUsage[lruIndex] = i;
for (int j = 0; j < n; j++) {
printf("%d ", memory[j]);
} printf("\n");
} else {
for (int j = 0; j < n; j++) {
if (memory[j] == currentPage) {
pageUsage[j] = i;
break;
}
}
}
}
printf("\nTotal Page Faults: %d\n", pageFaults);
return 0; }
```

SLIP 16

Q.1 Write a program to find the execution time taken for execution of a given set of instructions(use clock()function)

```
#include <stdio.h>
#include <time.h>
void yourInstructions() {
for (int i = 0; i < 1000000; i++) {
}
}
int main() {
clock_t start, end;
double cpu_time_used;
start = clock();
yourInstructions();
end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Execution time: %f seconds\n", cpu_time_used);
return 0;
}
```

_____

Q.2 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n =3 as the number of memory frames. Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8 : Implement OPT

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
bool isPageInMemory(int page, int* memory, int n) {
for (int i = 0; i < n; i++) {
```

```
if (memory[i] == page) {
return true;
}
}
return false;
}
int findPageToReplace(int* memory, int* pageReferenceString, int n, int currentIndex) {
int pageToReplace = -1;
int farthestDistance = -1;
for (int i = 0; i < n; i++) {
int page = memory[i];
int distance = INT_MAX;
for (int j = currentIndex; j < n; j++) {
if (pageReferenceString[j] == page) {
distance = j;
break;
}
}
if (distance > farthestDistance) {
farthestDistance = distance;
pageToReplace = i;
}
}
return pageToReplace;
}
int main() {
int n = 3;
int pageReferenceString[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
int numPages = sizeof(pageReferenceString) / sizeof(pageReferenceString[0]);
```

```
int memory[n];
int pageFaults = 0;

for (int i = 0; i < n; i++) {
memory[i] = -1;
}
printf("Page Scheduling using OPT:\n");
for (int i = 0; i < numPages; i++) {
int currentPage = pageReferenceString[i];
if (!isPageInMemory(currentPage, memory, n)) {
pageFaults++;
int pageToReplace = findPageToReplace(memory, pageReferenceString, n, i);
memory[pageToReplace] = currentPage;
for (int j = 0; j < n; j++) {
printf("%d ", memory[j]);
}
printf("\n");
}
}
printf("\nTotal Page Faults: %d\n", pageFaults);
return 0;
}
```

_____

SLIP 17

Q.1 Write the program to calculate minimum number of resources needed to avoid deadlock.

```
#include <stdio.h>
void calculateMinimumResources(int processes, int resources, int available[resources], int allocated[processes][resources], int maximum[processes][resources]) {
int need[processes][resources];
int work[resources];
int finish[processes];
int safe_sequence[processes];
int work_copy[resources];
for (int i = 0; i < resources; i++) {
work[i] = available[i];
}
for (int i = 0; i < processes; i++) {
finish[i] = 0;
}
for (int i = 0; i < processes; i++) {
for (int j = 0; j < resources; j++) {
need[i][j] = maximum[i][j] - allocated[i][j];
}
}
int count = 0;
while (count < processes) {
int found = 0;
for (int i = 0; i < processes; i++) {
if (finish[i] == 0) {
int j;
for (j = 0; j < resources; j++) {
```

```
if (need[i][j] > work[j]) {
break;
}
}
if (j == resources) {
for (int k = 0; k < resources; k++) {
work[k] += allocated[i][k];
}
safe_sequence[count] = i;
finish[i] = 1;
count++;
found = 1;
}
}
}
if (found == 0) {
printf("System is not in a safe state. Deadlock detected!\n");
return;
}
}
printf("System is in a safe state. Safe sequence: ");
for (int i = 0; i < processes; i++) {
printf("P%d ", safe_sequence[i]);
}
printf("\n");
printf("Minimum resources needed to avoid deadlock: ");
for (int i = 0; i < resources; i++) {
work_copy[i] = available[i];
for (int j = 0; j < processes; j++) {
```

```
work_copy[i] -= allocated[j][i];
}
printf("%d ", work_copy[i]);
}
printf("\n");
}
int main() {
int processes = 5;
int resources = 3;
int available[] = {3, 3, 2};
int allocated[5][3] = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};
int maximum[5][3] = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};
calculateMinimumResources(processes, resources, available, allocated, maximum);
return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n=3 as

the number of memory frames.

Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8

Implement OPT

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int isPageInMemory(int page, int* memory, int num_frames) {
for (int i = 0; i < num_frames; i++) {
if (memory[i] == page) {
return 1;
```

```
}
}
return 0;
}
int findOptimalReplacement(int* memory, int* page_references, int num_frames, int current_index, int num_references) {
int page_to_replace = -1;
int farthest_distance = -1;
for (int i = 0; i < num_frames; i++) {
int page = memory[i];
int distance = num_references;
for (int j = current_index + 1; j < num_references; j++) {
if (page_references[j] == page) {
distance = j - current_index;
break;
}
}
if (distance > farthest_distance) {
farthest_distance = distance;
page_to_replace = i;
}
}
return page_to_replace;
}
int main() {
char reference_string[] = "12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8";
int num_frames = 3;
int page_references[100];
int memory[3] = {-1, -1, -1};
int page_faults = 0;
```

```
int current_index = 0;
int num_references = 0;
char* token = strtok(reference_string, ",");
while (token != NULL) {
page_references[num_references] = atoi(token);
num_references++;
token = strtok(NULL, ",");
}
for (int i = 0; i < num_references; i++) {
if (!isPageInMemory(page_references[i], memory, num_frames)) {
page_faults++;
if (memory[current_index] == -1) {
memory[current_index] = page_references[i];
} else {
int page_to_replace = findOptimalReplacement(memory, page_references, num_frames, i,
num_references);
memory[page_to_replace] = page_references[i];
}
current_index = (current_index + 1) % num_frames;
printf("Page %d -> Frames: %d, %d, %d\n", page_references[i], memory[0], memory[1],
memory[2]);
}
}
printf("Total Page Faults: %d\n", page_faults);
return 0;
}
```

SLIP 18

Q. 1 Write a C program to accept the number of process and resources and find the need matrix content and display it.

```
#include <stdio.h>
int main() {
int num_processes, num_resources;
printf("Enter the number of processes: ");
scanf("%d", &num_processes);
printf("Enter the number of resources: ");
scanf("%d", &num_resources);
int maximum[num_processes][num_resources];
int allocation[num_processes][num_resources];
int need[num_processes][num_resources];
printf("Enter the Maximum Allocation Matrix:\n");
for (int i = 0; i < num_processes; i++) {
for (int j = 0; j < num_resources; j++) {
scanf("%d", &maximum[i][j]);
}
}
printf("Enter the Allocation Matrix:\n");
for (int i = 0; i < num_processes; i++) {
for (int j = 0; j < num_resources; j++) {
scanf("%d", &allocation[i][j]);
}
}
printf("Need Matrix:\n");
for (int i = 0; i < num_processes; i++) {
for (int j = 0; j < num_resources; j++) {
need[i][j] = maximum[i][j] - allocation[i][j];
```

```
printf("%d ", need[i][j]);
}
printf("\n");
}
return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.

Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8

Implement OPT

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int isPageInMemory(int page, int* memory, int num_frames) {
for (int i = 0; i < num_frames; i++) {
if (memory[i] == page) {
return 1;
}
}
return 0;
}
int findOptimalReplacement(int* memory, int* page_references, int num_frames, int current_index, int num_references) {
int page_to_replace = -1;
int farthest_distance = -1;
for (int i = 0; i < num_frames; i++) {
int page = memory[i];
```

```
int distance = num_references;
for (int j = current_index + 1; j < num_references; j++) {
if (page_references[j] == page) {
distance = j - current_index;
break;
}
}
if (distance > farthest_distance) {
farthest_distance = distance;
page_to_replace = i;
}
}
return page_to_replace;
}
int main() {
char reference_string[] = "12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8";
int num_frames;
int page_references[100];
int memory[20]; // Max memory frames
int page_faults = 0;
int current_index = 0;
int num_references = 0;
printf("Enter the number of memory frames: ");
scanf("%d", &num_frames);
char* token = strtok(reference_string, ",");
while (token != NULL) {
page_references[num_references] = atoi(token);
num_references++;
token = strtok(NULL, ",");
```

```
}
for (int i = 0; i < num_references; i++) {
if (!isPageInMemory(page_references[i], memory, num_frames)) {
page_faults++;
if (i < num_frames) {
memory[i] = page_references[i];
} else {
int page_to_replace = findOptimalReplacement(memory, page_references, num_frames, i,
num_references);
memory[page_to_replace] = page_references[i];
}
printf("Page %d -> Frames: ", page_references[i]);
for (int j = 0; j < num_frames; j++) {
printf("%d ", memory[j]);
}
printf("\n");
}
}
printf("Total Page Faults: %d\n", page_faults);
return 0;
}
```

SLIP 19

Q.1 Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
pid_t child_pid;
child_pid = fork();
if (child_pid < 0) {
// Fork failed
perror("Fork failed");
exit(1);
} else if (child_pid == 0) {
printf("Child process ID: %d\n", getpid());
execl("/bin/ls", "ls", (char *)NULL);
perror("Execl failed");
exit(1);
} else {
printf("Parent process ID: %d\n", getpid());
sleep(2);
int status;
wait(&status);
if (WIFEXITED(status)) {
printf("Child process exited with status: %d\n", WEXITSTATUS(status));
}
```

```
}
return 0;
}
```

Q.2 Write the program to simulate Non-preemptive Priority scheduling. The arrival time and first CPU burst and priority for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
struct Process {
int id;
int arrival_time;
int burst_time;
int priority;
int waiting_time;
int turnaround_time;
};
void swap(struct Process* a, struct Process* b) {
struct Process temp = *a;
*a = *b;
*b = temp;
}
int main() {
int n, i, j;
srand(time(NULL));

printf("Enter the number of processes: ");
```

```
scanf("%d", &n);
struct Process processes[n];
for (i = 0; i < n; i++) {
processes[i].id = i + 1;
processes[i].arrival_time = 0;
processes[i].burst_time = rand() % 10 + 1;
processes[i].priority = rand() % 5;
}
for (i = 0; i < n - 1; i++) {
for (j = 0; j < n - i - 1; j++) {
if (processes[j].priority > processes[j + 1].priority) {
swap(&processes[j], &processes[j + 1]);
}
}
}
int total_waiting_time = 0;
int total_turnaround_time = 0;
int completion_time = 0;
printf("Gantt Chart:\n");
for (i = 0; i < n; i++) {
printf("P%d (%d) -> ", processes[i].id, completion_time);
processes[i].waiting_time = completion_time - processes[i].arrival_time;
processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
completion_time += processes[i].burst_time;
total_waiting_time += processes[i].waiting_time;
total_turnaround_time += processes[i].turnaround_time;
}
printf("\n");
```

```
printf("Process\tWaiting Time\tTurnaround Time\n");

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

printf("P%d\t%d\t\t%d\n", processes[i].id, processes[i].waiting_time,
processes[i].turnaround_time);

}

float average_waiting_time = (float)total_waiting_time / n;

float average_turnaround_time = (float)total_turnaround_time / n;

printf("Average Waiting Time: %.2f\n", average_waiting_time);

printf("Average Turnaround Time: %.2f\n", average_turnaround_time);

return 0;

}
```

SLIP 20

Q.1 Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
pid_t child_pid;
child_pid = fork();
if (child_pid < 0) {
perror("Fork failed");
exit(1);
} else if (child_pid == 0) {
printf("Child process ID: %d\n", getpid());
execl("/bin/ls", "ls", (char *)NULL);
perror("Execl failed");
exit(1);
} else {
printf("Parent process ID: %d\n", getpid());
sleep(2);
int status;
wait(&status);
if (WIFEXITED(status)) {
printf("Child process exited with status: %d\n", WEXITSTATUS(status));
}
}
```

```
return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n=3 as

the number of memory frames.

Reference String : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2

i. Implement LRU

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int isPageInMemory(int page, int* memory, int num_frames) {
for (int i = 0; i < num_frames; i++) {
if (memory[i] == page) {
return 1;
}
}
return 0;
}
int findLRUPage(int* memory, int* page_references, int num_frames, int current_index) {
int page_to_replace = -1;
int min_index = current_index;
for (int i = 0; i < num_frames; i++) {
int page = memory[i];
for (int j = current_index - 1; j >= 0; j--) {
if (page_references[j] == page) {
if (j < min_index) {
```

```
min_index = j;
page_to_replace = i;
}
break;
}
}
}
return page_to_replace;
}
int main() {
char reference_string[] = "7,0,1,2,0,3,0,4,2,3,0,3,2";
int num_frames = 3;
int page_references[100];
int memory[3];
int page_faults = 0;
int current_index = 0;
int num_references = 0;
char* token = strtok(reference_string, ",");
while (token != NULL) {
page_references[num_references] = atoi(token);
num_references++;
token = strtok(NULL, ",");
}
for (int i = 0; i < num_references; i++) {
if (!isPageInMemory(page_references[i], memory, num_frames)) {
page_faults++;

if (i < num_frames) {
memory[i] = page_references[i];
```

```
} else {
int page_to_replace = findLRUPage(memory, page_references, num_frames, i);
memory[page_to_replace] = page_references[i];
}
printf("Page %d -> Frames: %d, %d, %d\n", page_references[i], memory[0], memory[1], memory[2]);
}
}
printf("Total Page Faults: %d\n", page_faults);
return 0;
}
```
