**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.**

**Ans**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdlib.h>

int main() {

pid\_t pid;

int child\_status;

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

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(1);

} else if (pid == 0) {

// Child process

printf("Child process (PID: %d) started with default priority.\n", getpid());

// Increase the "niceness" of the child process by 10

int new\_priority = nice(10);

if (new\_priority == -1) {

perror("nice() failed");

} else {

printf("Child process priority increased, new nice value: %d\n", new\_priority);

}

// Simulate some work in child

for (int i = 0; i < 5; i++) {

printf("Child process working...\n");

sleep(1);

}

exit(0);

} else {

// Parent process

printf("Parent process (PID: %d) waiting for child to complete.\n", getpid());

// Wait for child process to finish

waitpid(pid, &child\_status, 0);

printf("Child process completed with status: %d\n", child\_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 :3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6 Implement FIFO**

**Ans**

#include <stdio.h>

#include <stdbool.h>

int fifo\_page\_replacement(int reference\_string[], int num\_references, int num\_frames) {

int frames[num\_frames]; // Array to store the current pages in memory

int page\_faults = 0; // Counter for page faults

int index = 0; // FIFO index for the next page to be replaced

// Initialize frames with -1 (indicating empty)

for (int i = 0; i < num\_frames; i++) {

frames[i] = -1;

}

// Iterate through each page in the reference string

for (int i = 0; i < num\_references; i++) {

int page = reference\_string[i];

bool page\_in\_memory = false;

// Check if the page is already in the frames (no page fault)

for (int j = 0; j < num\_frames; j++) {

if (frames[j] == page) {

page\_in\_memory = true;

break;

}

}

// If page is not in memory, we have a page fault

if (!page\_in\_memory) {

page\_faults++;

// Insert page at FIFO index and update the index

frames[index] = page;

index = (index + 1) % num\_frames; // Move to the next FIFO position

}

// Print the current state of frames

printf("Page: %d -> Frames: ", page);

for (int j = 0; j < num\_frames; j++) {

if (frames[j] != -1)

printf("%d ", frames[j]);

else

printf("- ");

}

printf("\n");

}

return page\_faults;

}

int main() {

int reference\_string[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};

int num\_references = sizeof(reference\_string) / sizeof(reference\_string[0]);

int num\_frames = 3;

// Run the FIFO page replacement simulation

int total\_page\_faults = fifo\_page\_replacement(reference\_string, num\_references, num\_frames);

printf("\nTotal Page Faults: %d\n", total\_page\_faults);

return 0;

}

**SLIP 2 .**

**Q1. 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”. [10 marks]**

**ANS**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

int main() {

pid\_t pid;

// Create a child process

pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(1);

} else if (pid == 0) {

// Child process

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

printf("Hello World\n");

} else {

// Parent process

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

printf("Hi\n");

}

return 0;

}

**Q2. 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]**

**ANS**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_PROCESSES 10

// Structure to hold information about each process

typedef struct {

int process\_id;

int arrival\_time;

int cpu\_burst\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

} Process;

// Function to compare processes based on arrival time and CPU burst time for sorting

int compare(const void \*a, const void \*b) {

Process \*p1 = (Process \*)a;

Process \*p2 = (Process \*)b;

if (p1->arrival\_time == p2->arrival\_time)

return p1->cpu\_burst\_time - p2->cpu\_burst\_time;

return p1->arrival\_time - p2->arrival\_time;

}

int main() {

int n, i;

int total\_turnaround\_time = 0, total\_waiting\_time = 0;

Process processes[MAX\_PROCESSES];

srand(time(0)); // Seed for random burst generation

// Input: Number of processes

printf("Enter the number of processes (up to %d): ", MAX\_PROCESSES);

scanf("%d", &n);

// Input: Arrival time and initial CPU burst time for each process

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

processes[i].process\_id = i + 1;

printf("Enter arrival time and CPU burst time for Process %d: ", processes[i].process\_id);

scanf("%d %d", &processes[i].arrival\_time, &processes[i].cpu\_burst\_time);

}

// Sort processes based on arrival time, and CPU burst time (for SJF non-preemptive)

qsort(processes, n, sizeof(Process), compare);

int current\_time = 0;

printf("\nGantt Chart:\n");

// Simulation of SJF (non-preemptive) scheduling

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

// Calculate start time, completion time, turnaround time, and waiting time

if (current\_time < processes[i].arrival\_time) {

current\_time = processes[i].arrival\_time;

}

processes[i].waiting\_time = current\_time - processes[i].arrival\_time;

processes[i].completion\_time = current\_time + processes[i].cpu\_burst\_time;

processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

total\_turnaround\_time += processes[i].turnaround\_time;

total\_waiting\_time += processes[i].waiting\_time;

// Display the process in the Gantt chart

printf("| P%d (%d) ", processes[i].process\_id, current\_time);

// Update current time to next CPU burst plus fixed I/O waiting time (2 units)

current\_time = processes[i].completion\_time + 2;

// Generate the next CPU burst time randomly (assuming range 1-10)

processes[i].cpu\_burst\_time = rand() % 10 + 1;

}

printf("|\n\n");

// Output the Turnaround Time and Waiting Time for each process

printf("Process\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");

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

printf("P%d\t%d\t%d\t%d\t\t%d\t\t%d\n",

processes[i].process\_id,

processes[i].arrival\_time,

processes[i].cpu\_burst\_time,

processes[i].completion\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

// Calculate and display the average Turnaround Time and Waiting Time

double average\_turnaround\_time = (double)total\_turnaround\_time / n;

double average\_waiting\_time = (double)total\_waiting\_time / n;

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

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

return 0;

}

**Slip 3 .**

**Q1.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.**

**ANS**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdlib.h>

int main() {

pid\_t pid;

// Create a child process

pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(1);

} else if (pid == 0) {

// Child process

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

// Using exec to run 'ls' command

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

// If exec fails, print an error message

perror("exec failed");

exit(1);

} else {

// Parent process waits for the child to complete

wait(NULL);

printf("Child process has terminated.\n");

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

}

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**

**ANS:** #include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_PROCESSES 10

#define IO\_WAIT\_TIME 2 // Fixed I/O waiting time between CPU bursts

// Structure to hold information about each process

typedef struct {

int process\_id;

int arrival\_time;

int cpu\_burst\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

} Process;

// Function to generate the next CPU burst time randomly

int generate\_random\_burst() {

return rand() % 10 + 1; // Generates a burst between 1 and 10

}

int main() {

int n, i;

int total\_turnaround\_time = 0, total\_waiting\_time = 0;

Process processes[MAX\_PROCESSES];

srand(time(0)); // Seed for random burst generation

// Input: Number of processes

printf("Enter the number of processes (up to %d): ", MAX\_PROCESSES);

scanf("%d", &n);

// Input: Arrival time and initial CPU burst time for each process

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

processes[i].process\_id = i + 1;

printf("Enter arrival time and CPU burst time for Process %d: ", processes[i].process\_id);

scanf("%d %d", &processes[i].arrival\_time, &processes[i].cpu\_burst\_time);

}

int current\_time = 0;

printf("\nGantt Chart:\n");

// FCFS Scheduling simulation

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

// Calculate the start time for each process

if (current\_time < processes[i].arrival\_time) {

current\_time = processes[i].arrival\_time;

}

processes[i].waiting\_time = current\_time - processes[i].arrival\_time;

processes[i].completion\_time = current\_time + processes[i].cpu\_burst\_time;

processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

// Update total times for averages

total\_turnaround\_time += processes[i].turnaround\_time;

total\_waiting\_time += processes[i].waiting\_time;

// Display the process in the Gantt chart

printf("| P%d (%d) ", processes[i].process\_id, current\_time);

// Update the current time to reflect the next CPU burst and I/O waiting time

current\_time = processes[i].completion\_time + IO\_WAIT\_TIME;

// Generate the next CPU burst randomly

processes[i].cpu\_burst\_time = generate\_random\_burst();

}

printf("|\n\n");

// Output the Turnaround Time and Waiting Time for each process

printf("Process\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");

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

printf("P%d\t%d\t%d\t%d\t\t%d\t\t%d\n",

processes[i].process\_id,

processes[i].arrival\_time,

processes[i].cpu\_burst\_time,

processes[i].completion\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

// Calculate and display the average Turnaround Time and Waiting Time

double average\_turnaround\_time = (double)total\_turnaround\_time / n;

double average\_waiting\_time = (double)total\_waiting\_time / n;

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

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

return 0;

}

**SLIP 4 .**

**Q1 1 Write a program to illustrate the concept of orphan process ( Using fork() and sleep()) [10 marks]**

ANS #include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(1);

} else if (pid == 0) {

// Child process

printf("Child process (PID: %d) started and will sleep for 5 seconds...\n", getpid());

sleep(5); // Child process sleeps, parent terminates during this time

printf("Child process (PID: %d) is now orphaned and adopted by init/systemd.\n", getpid());

printf("Child process (PID: %d) finishing execution.\n", getpid());

} else {

// Parent process

printf("Parent process (PID: %d) will terminate immediately.\n", getpid());

exit(0); // Parent terminates, making the child an orphan

}

return 0;

}

**Q2 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..**

**ANS:**  #include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_PROCESSES 10

#define IO\_WAIT\_TIME 2 // Fixed I/O waiting time between CPU bursts

// Structure to hold information about each process

typedef struct {

int process\_id;

int arrival\_time;

int cpu\_burst\_time;

int priority;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int is\_completed;

} Process;

// Function to generate the next CPU burst time randomly

int generate\_random\_burst() {

return rand() % 10 + 1; // Generates a burst time between 1 and 10

}

// Function to find the next process based on priority and arrival time

int find\_next\_process(Process processes[], int n, int current\_time) {

int i, highest\_priority = -1, selected\_process = -1;

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

if (!processes[i].is\_completed && processes[i].arrival\_time <= current\_time) {

if (highest\_priority == -1 || processes[i].priority < highest\_priority) {

highest\_priority = processes[i].priority;

selected\_process = i;

}

}

}

return selected\_process;

}

int main() {

int n, i;

int total\_turnaround\_time = 0, total\_waiting\_time = 0;

Process processes[MAX\_PROCESSES];

srand(time(0)); // Seed for random burst generation

// Input: Number of processes

printf("Enter the number of processes (up to %d): ", MAX\_PROCESSES);

scanf("%d", &n);

// Input: Arrival time, CPU burst time, and priority for each process

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

processes[i].process\_id = i + 1;

printf("Enter arrival time, CPU burst time, and priority for Process %d: ", processes[i].process\_id);

scanf("%d %d %d", &processes[i].arrival\_time, &processes[i].cpu\_burst\_time, &processes[i].priority);

processes[i].is\_completed = 0; // Mark process as not completed initially

}

int current\_time = 0;

printf("\nGantt Chart:\n");

// Non-Preemptive Priority Scheduling simulation

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

// Find the next process to execute based on highest priority

int next\_process = find\_next\_process(processes, n, current\_time);

if (next\_process == -1) {

// If no process is ready, increment current time

current\_time++;

i--; // Decrement loop counter to retry finding a process

continue;

}

// Calculate start and completion times for the selected process

if (current\_time < processes[next\_process].arrival\_time) {

current\_time = processes[next\_process].arrival\_time;

}

processes[next\_process].waiting\_time = current\_time - processes[next\_process].arrival\_time;

processes[next\_process].completion\_time = current\_time + processes[next\_process].cpu\_burst\_time;

processes[next\_process].turnaround\_time = processes[next\_process].completion\_time - processes[next\_process].arrival\_time;

processes[next\_process].is\_completed = 1; // Mark process as completed

// Update total times for averages

total\_turnaround\_time += processes[next\_process].turnaround\_time;

total\_waiting\_time += processes[next\_process].waiting\_time;

// Display the process in the Gantt chart

printf("| P%d (%d) ", processes[next\_process].process\_id, current\_time);

// Update the current time to reflect CPU burst and I/O waiting time

current\_time = processes[next\_process].completion\_time + IO\_WAIT\_TIME;

// Generate the next CPU burst randomly

processes[next\_process].cpu\_burst\_time = generate\_random\_burst();

}

printf("|\n\n");

// Output the Turnaround Time and Waiting Time for each process

printf("Process\tArrival\tBurst\tPriority\tCompletion\tTurnaround\tWaiting\n");

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

printf("P%d\t%d\t%d\t%d\t\t%d\t\t%d\t\t%d\n",

processes[i].process\_id,

processes[i].arrival\_time,

processes[i].cpu\_burst\_time,

processes[i].priority,

processes[i].completion\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

// Calculate and display the average Turnaround Time and Waiting Time

double average\_turnaround\_time = (double)total\_turnaround\_time / n;

double average\_waiting\_time = (double)total\_waiting\_time / n;

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

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

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.**

**Ans:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <errno.h>

int main() {

pid\_t pid;

int nice\_value;

// Forking a new process

pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

// Child process

printf("Child Process (PID: %d) - Before nice(): My nice value: %d\n", getpid(), nice(0));

// Assigning a higher priority to the child process using nice()

nice\_value = nice(-5); // Decrease nice value (increase priority)

if (nice\_value == -1 && errno != 0) {

perror("Failed to change nice value");

exit(EXIT\_FAILURE);

}

printf("Child Process (PID: %d) - After nice(): My new nice value: %d\n", getpid(), nice(0));

// Simulating some work

for (volatile int i = 0; i < 100000000; i++);

printf("Child Process (PID: %d) - Work completed!\n", getpid());

exit(EXIT\_SUCCESS);

} else {

// Parent process

printf("Parent Process (PID: %d) - My nice value: %d\n", getpid(), nice(0));

// Simulating some work in the parent process

for (volatile int i = 0; i < 100000000; i++);

// Waiting for child process to complete

wait(NULL);

printf("Parent Process (PID: %d) - Child has completed execution.\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**

**Ans:** #include <stdio.h>

#include <stdlib.h>

#define MAX\_FRAMES 10 // Maximum number of frames

// Function to implement FIFO page replacement algorithm

void fifo\_page\_replacement(int pages[], int n, int num\_frames) {

int frames[MAX\_FRAMES];

int page\_faults = 0;

int index = 0;

int is\_page\_present;

// Initialize frames to -1

for (int i = 0; i < num\_frames; i++) {

frames[i] = -1;

}

printf("Page Reference String: ");

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

printf("%d ", pages[i]);

}

printf("\n");

printf("Frame States:\n");

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

is\_page\_present = 0;

// Check if the page is already in the frames

for (int j = 0; j < num\_frames; j++) {

if (frames[j] == pages[i]) {

is\_page\_present = 1; // Page is already in frame

break;

}

}

if (!is\_page\_present) {

// Page fault occurs

page\_faults++;

frames[index] = pages[i]; // Insert the new page into the frame

index = (index + 1) % num\_frames; // Move to the next frame index

}

// Print the current state of frames

printf("Current Frame State: ");

for (int j = 0; j < num\_frames; j++) {

if (frames[j] != -1) {

printf("%d ", frames[j]);

} else {

printf("X "); // 'X' indicates an empty frame

}

}

printf("\n");

}

printf("\nTotal Page Faults: %d\n", page\_faults);

}

int main() {

int n = 15; // Number of pages in reference string

int pages[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};

int num\_frames;

// Input: Number of memory frames

printf("Enter the number of memory frames (1 to %d): ", MAX\_FRAMES);

scanf("%d", &num\_frames);

if (num\_frames < 1 || num\_frames > MAX\_FRAMES) {

printf("Invalid number of frames. Please enter a value between 1 and %d.\n", MAX\_FRAMES);

return 1;

}

fifo\_page\_replacement(pages, n, num\_frames);

return 0;

}