**ASS1**

**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 = -10;

pid = fork();

if (pid < 0) {

perror("Fork failed");

exit(1);

} else if (pid == 0) {

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

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

perror("Failed to change nice value in child");

exit(1);

}

printf("Child Process: New nice value = %d\n", nice(0));

for (volatile long i = 0; i < 1e8; i++);

printf("Child Process: Work done.\n");

exit(0);

} else {

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

wait(NULL);

printf("Parent Process: Child has completed.\n");

}

return 0;

}

**gcc -o a1nice a1nice.c**

**$ ./a1nice**

**Parent Process: PID = 4572**

**Child Process: PID = 4573**

**Failed to change nice value in child: Operation not permitted**

**Parent Process: Child has completed.**

**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 <stdlib.h>

#define MAX\_FRAMES 10

void fifoPageReplacement(int referenceString[], int n, int nFrames) {

int memory[MAX\_FRAMES];

int pageFaults = 0;

int memorySize = 0;

int i, j;

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

memory[i] = -1;

}

printf("Page Reference String: ");

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

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

}

printf("\n");

printf("Page Scheduling (Memory States):\n");

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

int page = referenceString[i];

int found = 0;

for (j = 0; j < memorySize; j++) {

if (memory[j] == page) {

found = 1;

break;

}

}

if (!found) {

pageFaults++;

if (memorySize < nFrames) {

memory[memorySize] = page;

memorySize++;

} else {

for (j = 0; j < nFrames - 1; j++) {

memory[j] = memory[j + 1];

}

memory[nFrames - 1] = page;

}

}

printf("Step %d: ", i + 1);

for (j = 0; j < memorySize; j++) {

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

}

printf("\n");

}

printf("Total Number of Page Faults: %d\n", pageFaults);

}

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 nFrames = 3;

fifoPageReplacement(referenceString, n, nFrames);

return 0;

}

**$ touch implement\_FIFO.c**

**$ gcc -o fifo1 implement\_FIFO.c**

**$ ./fifo1**

**Page Reference String: 3 4 5 6 3 4 7 3 4 5 6 7 2 4 6**

**Page Scheduling (Memory States):**

**Step 1: 3**

**Step 2: 3 4**

**Step 3: 3 4 5**

**Step 4: 4 5 6**

**Step 5: 5 6 3**

**Step 6: 6 3 4**

**Step 7: 3 4 7**

**Step 8: 3 4 7**

**Step 9: 3 4 7**

**Step 10: 4 7 5**

**Step 11: 7 5 6**

**Step 12: 7 5 6**

**Step 13: 5 6 2**

**Step 14: 6 2 4**

**Step 15: 6 2 4**

**Total Number of Page Faults: 11**

**Ass 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”.**

**Ans**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

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

printf("Hello World\n");

} else {

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

printf("Hi\n");

}

return 0;

}

**gcc -o a2fork a2fork.c**

**$ ./a2fork**

**Parent Process ID: 5123**

**Hi**

**Child Process ID: 5124**

**Hello World**

**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.**

**Ans**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

} Process;

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

return ((Process \*)a)->arrival\_time - ((Process \*)b)->arrival\_time;

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[n];

srand(time(0));

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

processes[i].pid = i + 1;

printf("Enter arrival time and first CPU burst for process %d: ", i + 1);

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

}

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

int current\_time = 0;

int completed = 0;

while (completed < n) {

int idx = -1;

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

if (processes[i].arrival\_time <= current\_time && processes[i].completion\_time == 0) {

if (idx == -1 || processes[i].burst\_time < processes[idx].burst\_time) {

idx = i;

}

}

}

if (idx != -1) {

current\_time += processes[idx].burst\_time + 2; // I/O wait time

processes[idx].completion\_time = current\_time;

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

processes[idx].waiting\_time = processes[idx].turnaround\_time - processes[idx].burst\_time;

printf("P%d ", processes[idx].pid);

processes[idx].burst\_time = rand() % 10 + 1; // Random next CPU burst

completed++;

} else {

current\_time++;

}

}

printf("\nGantt Chart: ");

current\_time = 0;

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

printf("%d ", current\_time);

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

printf("%d ", current\_time);

}

printf("\n");

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

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

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

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

}

printf("Average Turnaround Time: %.2f\n", total\_turnaround\_time / n);

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

return 0;

}

**touch a2sjf.c**

**gcc -o a2sjf a2sjf.c**

**./a2sjf**

**Enter number of processes: 2**

**Enter arrival time and first CPU burst for process 1: 2**

**3**

**Enter arrival time and first CPU burst for process 2: 4**

**2**

**P1 P2**

**Gantt Chart: 0 6 6 10**

**Average Turnaround Time: 6.00**

**Average Waiting Time: 3.50**

**Q3**

**Partially implement the Menu driven Banker's algorithm for accepting Allocation, Max from**

**user.**

**a) Accept Available**

**b) Display Allocation, Max**

**c) Find Need and display It,**

**d) Display Available Consider the system with 3 resources types A,B, and C with 7,2,6**

**instances respectively.**

**Consider the following snapshot:**

**Answer the following questions:**

**a) Display the contents of Available array?**

**b) Is there any deadlock? Print the message**

**Ans**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 3

int allocation[MAX\_PROCESSES][MAX\_RESOURCES];

int max[MAX\_PROCESSES][MAX\_RESOURCES];

int need[MAX\_PROCESSES][MAX\_RESOURCES];

int available[MAX\_RESOURCES];

int n, m;

void calculateNeed() {

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

for (int j = 0; j < m; j++) {

need[i][j] = max[i][j] - allocation[i][j];

}

}

}

void displayAllocationMax() {

printf("Allocation Matrix:\n");

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

for (int j = 0; j < m; j++) {

printf("%d ", allocation[i][j]);

}

printf("\n");

}

printf("Max Matrix:\n");

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

for (int j = 0; j < m; j++) {

printf("%d ", max[i][j]);

}

printf("\n");

}

}

void displayAvailable() {

printf("Available Resources:\n");

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

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

}

printf("\n");

}

bool checkDeadlock() {

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES] = { false };

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

work[i] = available[i];

}

while (1) {

bool progressMade = false;

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

if (!finish[i]) {

bool canAllocate = true;

for (int j = 0; j < m; j++) {

if (need[i][j] > work[j]) {

canAllocate = false;

break;

}

}

if (canAllocate) {

for (int j = 0; j < m; j++) {

work[j] += allocation[i][j];

}

finish[i] = true;

progressMade = true;

}

}

}

if (!progressMade) break;

}

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

if (!finish[i]) {

printf("Deadlock detected! Process %d is in deadlock.\n", i + 1);

return true;

}

}

printf("No deadlock detected.\n");

return false;

}

int main() {

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resources (A, B, C): ");

scanf("%d", &m);

printf("Enter Available resources (A, B, C): ");

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

scanf("%d", &available[i]);

}

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

printf("Enter Allocation for process %d: ", i + 1);

for (int j = 0; j < m; j++) {

scanf("%d", &allocation[i][j]);

}

printf("Enter Max for process %d: ", i + 1);

for (int j = 0; j < m; j++) {

scanf("%d", &max[i][j]);

}

}

int choice;

do {

printf("\nMenu:\n");

printf("1. Display Allocation and Max\n");

printf("2. Display Available\n");

printf("3. Calculate and Display Need\n");

printf("4. Check for Deadlock\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

displayAllocationMax();

break;

case 2:

displayAvailable();

break;

case 3:

calculateNeed();

printf("Need Matrix:\n");

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

for (int j = 0; j < m; j++) {

printf("%d ", need[i][j]);

}

printf("\n");

}

break;

case 4:

checkDeadlock();

break;

case 5:

printf("Exiting...\n");

break;

default:

printf("Invalid choice. Please try again.\n");

}

} while (choice != 5);

return 0;

}

**gcc -o a2banker a2banker.c**

**./a2banker**

**Enter number of processes: 5**

**Enter number of resources (A, B, C): 3**

**Enter Available resources (A, B, C): 7 2 6**

**Enter Allocation for process 1: 0 1 0**

**Enter Max for process 1: 0 0 0**

**Enter Allocation for process 2: 2 0 0**

**Enter Max for process 2: 2 0 0**

**Enter Allocation for process 3: 3 0 3**

**Enter Max for process 3: 0 0 1**

**Enter Allocation for process 4: 2 1 1**

**Enter Max for process 4: 1 0 0**

**Enter Allocation for process 5: 0 0 2**

**Enter Max for process 5: 0 0 2**

**Menu:**

**1. Display Allocation and Max**

**2. Display Available**

**3. Calculate and Display Need**

**4. Check for Deadlock**

**5. Exit**

**Enter your choice: 1**

**Allocation Matrix:**

**0 1 0**

**2 0 0**

**3 0 3**

**2 1 1**

**0 0 2**

**Max Matrix:**

**0 0 0**

**2 0 0**

**0 0 1**

**1 0 0**

**0 0 2**

**Menu:**

**1. Display Allocation and Max**

**2. Display Available**

**3. Calculate and Display Need**

**4. Check for Deadlock**

**5. Exit**

**Enter your choice: 2**

**Available Resources:**

**7 2 6**

**Menu:**

**1. Display Allocation and Max**

**2. Display Available**

**3. Calculate and Display Need**

**4. Check for Deadlock**

**5. Exit**

**Enter your choice: 3**

**Need Matrix:**

**0 -1 0**

**0 0 0**

**-3 0 -2**

**-1 -1 -1**

**0 0 0**

**Menu:**

**1. Display Allocation and Max**

**2. Display Available**

**3. Calculate and Display Need**

**4. Check for Deadlock**

**5. Exit**

**Enter your choice: 4**

**No deadlock detected.**

**Menu:**

**1. Display Allocation and Max**

**2. Display Available**

**3. Calculate and Display Need**

**4. Check for Deadlock**

**5. Exit**

**Enter your choice: 5**

**Exiting...**

**Ass 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.**

**Ans**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

// This is the child process

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

char \*args[] = {"/bin/echo", "Hello from the child process!", NULL};

execvp(args[0], args);

perror("Exec failed"); // Exec only returns on failure

exit(EXIT\_FAILURE);

} else {

// This is the parent process

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

wait(NULL); // Wait for the child process to terminate

printf("Child process terminated. Control back to parent.\n");

}

return 0;

}

**touch a3exec.c**

**gcc -o a3exec a3exec.c**

**./a3exec**

**Parent Process ID: 5563**

**Child Process ID: 5564**

**Hello from the child process!**

**Child process terminated. Control back to parent.**

**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.**

**Ans**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

} Process;

void calculateTimes(Process processes[], int n) {

int current\_time = 0;

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

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

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

}

current\_time += processes[i].burst\_time + 2; // Add I/O wait time

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

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

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

// Generate next CPU burst randomly for the next execution

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

}

}

void displayGanttChart(Process processes[], int n) {

int current\_time = 0;

printf("Gantt Chart:\n");

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

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

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

}

printf("| P%d ", processes[i].pid);

current\_time += processes[i].burst\_time + 2; // Add I/O wait time

}

printf("|\n");

current\_time = 0;

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

current\_time += processes[i].burst\_time + 2; // Add I/O wait time

printf("%d ", current\_time);

}

printf("\n");

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[n];

srand(time(0));

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

processes[i].pid = i + 1;

printf("Enter arrival time and first CPU burst for process %d: ", i + 1);

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

}

calculateTimes(processes, n);

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

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

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

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

printf("Process P%d: Turnaround Time = %d, Waiting Time = %d\n",

processes[i].pid, processes[i].turnaround\_time, processes[i].waiting\_time);

}

printf("Average Turnaround Time: %.2f\n", total\_turnaround\_time / n);

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

displayGanttChart(processes, n);

return 0;

}

**gcc -o a3fcfs a3fcfs\_simulation.c**

**$ ./a3fcfs**

**Enter number of processes: 3**

**Enter arrival time and first CPU burst for process 1: 2**

**1**

**Enter arrival time and first CPU burst for process 2: 2**

**2**

**Enter arrival time and first CPU burst for process 3: 3**

**2**

**Process P1: Turnaround Time = 3, Waiting Time = 2**

**Process P2: Turnaround Time = 7, Waiting Time = 5**

**Process P3: Turnaround Time = 10, Waiting Time = 8**

**Average Turnaround Time: 6.67**

**Average Waiting Time: 5.00**

**Gantt Chart:**

**| P1 | P2 | P3 |**

**6 11 20**

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

**Ans**