FIFO PAGE REPLACEMENT QUESTION

#include <stdio.h>

#define MAX\_PAGES 50

#define MAX\_FRAMES 10

int main() {

    int pages[MAX\_PAGES], frames[MAX\_FRAMES];

    int num\_pages, num\_frames;

    int i, j, k;

    int page\_faults = 0;

    int is\_page\_in\_memory;

    // Input number of pages

    printf("Enter the number of pages: ");

    scanf("%d", &num\_pages);

    // Input page numbers

    printf("Enter the page numbers:\n");

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

        scanf("%d", &pages[i]); }

    // Input number of frames

    printf("Enter the number of frames: ");

    scanf("%d", &num\_frames);

    // Initialize frame array to -1 (indicating empty)

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

        frames[i] = -1;}

    // Print the header for the output

    printf("Ref String\tPage Frames\n");

    // Process each page

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

       // printf("%d\t\t", pages[i]);

        is\_page\_in\_memory = 0;

        // Check if the page is already in one of the frames

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

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

                is\_page\_in\_memory = 1;

                break; } }

        // If page is not in memory, replace the oldest page

        if (!is\_page\_in\_memory) {

            frames[j] = pages[i];

            j = (j + 1) % num\_frames;

            page\_faults++;}

        // Print the current state of the frames

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

            printf("%d\t", frames[k]);}

        printf("\n");}

    // Output the total number of page faults

    printf("Total page faults: %d\n", page\_faults);

    return 0;

}

2)BEST FIT

#include <stdio.h>

struct Process {

    int pid, size;

};

struct Partition {

    int pid, size, isFree;

};

void bestFit(struct Process p[], struct Partition f[], int n, int m) {

    for (int i = 0; i < n; i++) {

        int best = -1;

        for (int j = 0; j < m; j++) {

            if (f[j].isFree && f[j].size >= p[i].size &&

                (best == -1 || f[j].size < f[best].size))

                best = j;

        }

        if (best != -1) {

            f[best].isFree = 0;

            f[best].pid = p[i].pid;

            printf("P%d -> F%d\n", p[i].pid, best + 1);

        } else {

            printf("P%d not allocated\n", p[i].pid);

        }

    }

}

int main() {

    struct Process p[] = {{1, 10}, {2, 20}, {3, 15}, {4, 5}, {5, 18}};

    struct Partition f[] = {{1, 25,1}, {2, 15,1}, {3, 10,1}, {4, 20,1}, {5, 18,1}};

    bestFit(p, f, sizeof(p) / sizeof(p[0]), sizeof(f) / sizeof(f[0]));

    return 0;

}

FIRST FIT

#include <stdio.h>

int main() {

    // Array of process sizes

    int p[5] = {212, 417, 112, 426, 211};

    // Array of partition sizes

    int f[4] = {100, 500, 200, 300};

    // Array to store allocation status of processes

    // -1 means the process is not allocated to any partition

    int a[5] = {-1};

    // Loop through each process

    for (int i = 0; i < 5; i++) {

        // Loop through each partition to find the first suitable one

        for (int j = 0; j < 4; j++) {

            // Check if partition size is enough for the current process

            if (f[j] >= p[i]) {

                // Allocate the process to the partition

                a[i] = j;

                // Reduce the size of the partition

                f[j] -= p[i];

                // Break the loop after allocation

                break;

            }

        }

    }

    // Print the allocation results

    printf("Process No.\tProcess Size\tPartition No.\n");

    for (int i = 0; i < 5; i++) {

        // Print process number and its size

        printf("%d\t\t\t%d\t\t\t", i + 1, p[i]);

        // If allocated, print the partition number

        if (a[i] != -1)

            printf("%d\n", a[i] + 1);

        else

            // If not allocated, print "Not Allocated"

            printf("Not Allocated\n");

    }

    return 0;

}

4)))))PRIORITY

#include <stdio.h>

int main() {

int n, i, j, temp, bt[10], wt[10], tat[10], pr[10], avwt = 0, avtat = 0;

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

scanf("%d", &n);

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

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

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

}

// Sorting processes by priority (ascending order)

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

for (j = i + 1; j < n; j++) {

if (pr[i] > pr[j]) { // Correct comparison for sorting by priority

// Swap priority

temp = pr[i];

pr[i] = pr[j];

pr[j] = temp;

// Swap burst time

temp = bt[i];

bt[i] = bt[j];

bt[j] = temp;

}

}

}

wt[0] = 0; // Waiting time for the first process is 0

// Calculate waiting time and turnaround time

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

if (i > 0) {

wt[i] = wt[i - 1] + bt[i - 1];

}

tat[i] = bt[i] + wt[i];

avwt += wt[i];

avtat += tat[i];

}

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

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, bt[i], pr[i], wt[i], tat[i]);

}

avwt /= n;

avtat /= n;

printf("\nAverage Waiting Time: %d\n", avwt);

printf("Average Turnaround Time: %d\n", avtat);

return 0;

}

PRODUCER CONSUMER

#include <stdio.h>

#include <stdlib.h>

int mutex = 1;

int full = 0;

int empty = 3;

int x = 0;

int wait(int s) {

        return --s;}

int signal(int s) {

        return ++s;}

void producer() {

        mutex = wait(mutex);

        full = signal(full);

        empty = wait(empty);

        x++;

        printf("\nProducer produces item %d\n", x);

        mutex = signal(mutex);}

void consumer() {

        mutex = wait(mutex);

        full = wait(full);

        empty = signal(empty);

        printf("\nConsumer consumes item %d\n", x);

        x--;

        mutex = signal(mutex);}

int main() {

        int n;

        printf("\n1. PRODUCER\n2. CONSUMER\n3. EXIT\n");

        while (1) {

                printf("\nENTER YOUR CHOICE\n");

                scanf("%d", &n);

                switch (n) {

                        case 1:

                                if ((mutex == 1) && (empty != 0)) {

                                        producer();

                                } else {

                                        printf("BUFFER IS FULL\n");

                                }

                                break;

                        case 2:

                                if ((mutex == 1) && (full != 0)) {

                                        consumer();

                                } else {

                                        printf("BUFFER IS EMPTY\n");

                                }

                                break;

                        case 3:

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

                                exit(0);

                        default:

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

                }

        }

        return 0;}}

SJF

#include <stdio.h>

void main() {

    int i, j, temp, bt[10], n, wt[10], tt[10], w\_total = 0, t\_total = 0;

    float aw, at;

    // Input number of processes

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

    scanf("%d", &n);

    // Input burst times of processes

    printf("Enter the burst times of processes:\n");

    for (i = 0; i < n; i++) {

        scanf("%d", &bt[i]);

    }

    // Sort burst times in ascending order using Bubble Sort

    for (i = 0; i < n - 1; i++) {

        for (j = 0; j < n - i - 1; j++) {  // Corrected sorting logic

            if (bt[j] > bt[j + 1]) {

                temp = bt[j];

                bt[j] = bt[j + 1];

                bt[j + 1] = temp;

            }

        }

    }

    // Initialize waiting time and turnaround time for the first process

    wt[0] = 0;               // First process has no waiting time

    tt[0] = bt[0];           // Turnaround time is equal to its burst time

    w\_total = wt[0];

    t\_total = tt[0];

    // Calculate waiting time and turnaround time for remaining processes

    for (i = 1; i < n; i++) {

        wt[i] = wt[i - 1] + bt[i - 1]; // Waiting time

        tt[i] = wt[i] + bt[i];         // Turnaround time

        w\_total += wt[i];

        t\_total += tt[i];

    }

    // Calculate average waiting time and turnaround time

    aw = (float)w\_total / n;

    at = (float)t\_total / n;

    // Printing results

    printf("\nProcess\tBT\tWT\tTT\n");

    for (i = 0; i < n; i++) {

        printf("P%d\t%d\t%d\t%d\n", i + 1, bt[i], wt[i], tt[i]);

    }

    // Print average times

    printf("\nAverage Waiting Time: %.2f\n", aw);

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

}

FCFS

#include<stdio.h>

void main() {

int i,j,bt[10],n,wt[10],tt[10],w1=0,t1=0; float aw,at;

printf("enter no.of processes:\n");

scanf("%d",&n);

printf("enter the bursttime of processes:");

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

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

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

wt[0]=0;

tt[0]=bt[0];

wt[i+1]=bt[i]+wt[i];

tt[i+1]=tt[i]+bt[i+1];

w1=w1+wt[i];

t1=t1+tt[i];

}

aw=w1/n;

at=t1/n;

printf("\nbt\twt\ttt\n");

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

printf("%d\t%d\t%d\n",bt[i],wt[i],tt[i]);

 printf("aw=%f\n,at=%f\n",aw,at);

}

FORK

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

    pid\_t pid;

    pid = fork();

    if (pid < 0) {

        fprintf(stderr, "Fork Failed\n");

        return 1;

    }else if (pid == 0) {

        printf("This is the child process. My PID is %d\n", getpid());

    }else {

        printf("This is the parent process. My child's PID is %d and my PID is %d\n", pid, getpid());

    }

    return 0;

}

WAIT

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

int main(){

    pid\_t pid;

    pid=fork();

  if(pid<0){

    perror("fork failed");

  }

  else if(pid==0){

    printf("the pi id is %d",getpid());

    sleep(2);

  }

  else{

    printf("parent process is waitig for child ");

    wait(NULL);

    printf("parent process compeleted ");

  }

}