* 1. Write the simulation program to implement demand paging and show the

page scheduling and total number of page faults according to the LFU page replacement algorithm. Assume the memory of n frames.

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

write in linux code

🡺 #include <stdio.h>

#include <stdlib.h>

#include <limits.h>

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

#define MAX\_PAGES 100 // Maximum number of pages

int search(int page, int memory[], int n\_frames, int start\_index) {

int i;

for (i = start\_index; i < n\_frames; ++i) {

if (memory[i] == page) {

return i;

}

}

return -1;

}

int pageFaultsLFU(int reference\_string[], int n, int n\_frames) {

int memory[MAX\_FRAMES];

int page\_faults = 0;

int page\_frequency[MAX\_PAGES] = {0};

int i, j;

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

memory[i] = -1;

}

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

int page = reference\_string[i];

// Check if the page is already in memory

int page\_found = search(page, memory, n\_frames, 0);

if (page\_found == -1) {

page\_faults++;

int empty\_frame\_index = search(-1, memory, n\_frames, 0);

if (empty\_frame\_index != -1) {

memory[empty\_frame\_index] = page;

page\_frequency[page]++;

} else {

int min\_freq\_page = memory[0];

int min\_freq = page\_frequency[memory[0]];

for (j = 1; j < n\_frames; ++j) {

if (page\_frequency[memory[j]] < min\_freq) {

min\_freq\_page = memory[j];

min\_freq = page\_frequency[memory[j]];

}

}

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

if (memory[j] == min\_freq\_page) {

memory[j] = page;

page\_frequency[page]++;

page\_frequency[min\_freq\_page] = 0;

break;

}

}

}

} else {

page\_frequency[page]++;

}

printf("Page reference: %d - Memory: ", page);

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

if (memory[j] == -1) {

printf(" - ");

} else {

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

}

}

printf("\n");

}

return page\_faults;

}

int main() {

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

int n\_frames = 3; // Number of frames

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

int total\_page\_faults = pageFaultsLFU(reference\_string, n, n\_frames);

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

return 0;

}

* 1. Write a C program to implement the shell which displays the command prompt “myshell$”. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command ‘typeline’ as

typeline +n filename :- To print first n lines in the file.

typeline -a filename :- To print all lines in the file.

write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

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

#define MAX\_PAGES 100 // Maximum number of pages

int search(int page, int memory[], int n\_frames, int start\_index) {

int i;

for (i = start\_index; i < n\_frames; ++i) {

if (memory[i] == page) {

return i;

}

}

return -1;

}

int pageFaultsLFU(int reference\_string[], int n, int n\_frames) {

int memory[MAX\_FRAMES];

int page\_faults = 0;

int page\_frequency[MAX\_PAGES] = {0};

int i, j;

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

memory[i] = -1;

}

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

int page = reference\_string[i];

// Check if the page is already in memory

int page\_found = search(page, memory, n\_frames, 0);

if (page\_found == -1) {

page\_faults++;

int empty\_frame\_index = search(-1, memory, n\_frames, 0);

if (empty\_frame\_index != -1) {

memory[empty\_frame\_index] = page;

page\_frequency[page]++;

} else {

int min\_freq\_page = memory[0];

int min\_freq = page\_frequency[memory[0]];

for (j = 1; j < n\_frames; ++j) {

if (page\_frequency[memory[j]] < min\_freq) {

min\_freq\_page = memory[j];

min\_freq = page\_frequency[memory[j]];

}

}

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

if (memory[j] == min\_freq\_page) {

memory[j] = page;

page\_frequency[page]++;

page\_frequency[min\_freq\_page] = 0;

break;

}

}

}

} else {

page\_frequency[page]++;

}

printf("Page reference: %d - Memory: ", page);

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

if (memory[j] == -1) {

printf(" - ");

} else {

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

}

}

printf("\n");

}

return page\_faults;

}

int main() {

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

int n\_frames = 3; // Number of frames

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

int total\_page\_faults = pageFaultsLFU(reference\_string, n, n\_frames);

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

return 0;

}

* 1. Write the simulation program for demand paging and show the page scheduling and total number of page faults according the FIFO page replacement algorithm. Assume the memory of n frames.

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

write in linux code

#include <stdio.h>

#include <stdlib.h>

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

#define MAX\_PAGES 100 // Maximum number of pages

int search(int page, int memory[], int n\_frames) {

int i;

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

if (memory[i] == page) {

return i;

}

}

return -1;

}

int pageFaultsFIFO(int reference\_string[], int n, int n\_frames) {

int memory[MAX\_FRAMES];

int page\_faults = 0;

int frame\_index = 0;

int i, j;

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

memory[i] = -1;

}

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

int page = reference\_string[i];

// Check if the page is already in memory

int page\_found = search(page, memory, n\_frames);

if (page\_found == -1) {

page\_faults++;

memory[frame\_index] = page;

frame\_index = (frame\_index + 1) % n\_frames;

printf("Page reference: %d - Memory: ", page);

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

if (memory[j] == -1) {

printf(" - ");

} else {

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

}

}

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 n\_frames = 3; // Number of frames

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

int total\_page\_faults = pageFaultsFIFO(reference\_string, n, n\_frames);

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

return 0;

}

* 1. Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following ‘list’ commands as

myshell$ list f dirname :- To print names of all the files in current

directory.

myshell$ list n dirname :- To print the number of all entries in the current

directory

write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <dirent.h>

#include <sys/wait.h>

#define MAX\_INPUT\_SIZE 1024

#define MAX\_TOKEN\_SIZE 64

#define MAX\_NUM\_TOKENS 64

void executeCommand(char \*tokens[]) {

pid\_t pid = fork();

if (pid == -1) {

printf("Failed to fork.\n");

return;

} else if (pid == 0) {

if (execvp(tokens[0], tokens) == -1) {

printf("Command not found.\n");

}

exit(EXIT\_FAILURE);

} else {

wait(NULL);

}

}

void listFiles(char \*dirname, char \*option) {

DIR \*directory;

struct dirent \*entry;

directory = opendir(dirname);

if (directory == NULL) {

printf("Error opening directory.\n");

return;

}

if (strcmp(option, "f") == 0) {

while ((entry = readdir(directory)) != NULL) {

printf("%s\n", entry->d\_name);

}

} else if (strcmp(option, "n") == 0) {

int count = 0;

while ((entry = readdir(directory)) != NULL) {

count++;

}

printf("Number of entries in directory: %d\n", count);

} else {

printf("Invalid 'list' command option.\n");

}

closedir(directory);

}

int main() {

char input[MAX\_INPUT\_SIZE];

char \*tokens[MAX\_NUM\_TOKENS];

char \*token;

const char \*delimiter = " \n";

int num\_tokens;

while (1) {

printf("myshell$ ");

if (!fgets(input, MAX\_INPUT\_SIZE, stdin)) {

break;

}

num\_tokens = 0;

token = strtok(input, delimiter);

while (token != NULL && num\_tokens < MAX\_NUM\_TOKENS - 1) {

tokens[num\_tokens] = token;

num\_tokens++;

token = strtok(NULL, delimiter);

}

tokens[num\_tokens] = NULL;

if (strcmp(tokens[0], "list") == 0) {

if (num\_tokens != 3) {

printf("Invalid 'list' command format.\n");

} else {

char \*option = tokens[1];

char \*dirname = tokens[2];

listFiles(dirname, option);

}

} else {

executeCommand(tokens);

}

}

return 0;

}

* 1. rite the simulation program to implement demand paging and show the page scheduling and total number of page faults according to the LRU (using counter method) page replacement algorithm. Assume the memory of n frames.

Reference String : 3,5,7,2,5,1,2,3,1,3,5,3,1,6,2

write in linux code

#include <stdio.h>

#include <stdlib.h>

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

#define MAX\_PAGES 100 // Maximum number of pages

int search(int page, int memory[], int n\_frames) {

int i;

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

if (memory[i] == page) {

return i;

}

}

return -1;

}

int pageFaultsLRU(int reference\_string[], int n, int n\_frames) {

int memory[MAX\_FRAMES];

int page\_faults = 0;

int counter[MAX\_FRAMES] = {0};

int i, j, min\_counter;

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

memory[i] = -1;

}

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

int page = reference\_string[i];

// Check if the page is already in memory

int page\_found = search(page, memory, n\_frames);

if (page\_found == -1) {

page\_faults++;

int empty\_frame\_index = search(-1, memory, n\_frames);

if (empty\_frame\_index != -1) {

memory[empty\_frame\_index] = page;

counter[empty\_frame\_index] = 1;

} else {

min\_counter = counter[0];

int replace\_page\_index = 0;

for (j = 1; j < n\_frames; ++j) {

if (counter[j] < min\_counter) {

min\_counter = counter[j];

replace\_page\_index = j;

}

}

memory[replace\_page\_index] = page;

counter[replace\_page\_index] = 1;

}

} else {

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

if (memory[j] == page) {

counter[j]++;

}

}

}

printf("Page reference: %d - Memory: ", page);

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

if (memory[j] == -1) {

printf(" - ");

} else {

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

}

}

printf("\n");

}

return page\_faults;

}

int main() {

int reference\_string[] = {3, 5, 7, 2, 5, 1, 2, 3, 1, 3, 5, 3, 1, 6, 2};

int n\_frames = 3; // Number of frames

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

int total\_page\_faults = pageFaultsLRU(reference\_string, n, n\_frames);

printf("Total Page Faults using LRU (counter method): %d\n", total\_page\_faults);

return 0;

}

* 1. Write a programto implement the toy shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.

count c filename :- To print number of characters in the file. count w filename :- To print number of words in the file.

count l filename :- To print number of lines in the file.

write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#define MAX\_INPUT\_SIZE 1024

#define MAX\_TOKEN\_SIZE 64

#define MAX\_NUM\_TOKENS 64

void executeCommand(char \*tokens[]) {

pid\_t pid = fork();

if (pid == -1) {

printf("Failed to fork.\n");

return;

} else if (pid == 0) {

if (execvp(tokens[0], tokens) == -1) {

printf("Command not found.\n");

}

exit(EXIT\_FAILURE);

} else {

wait(NULL);

}

}

void countCharacters(char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

printf("Error opening file.\n");

return;

}

int count = 0;

int ch;

while ((ch = fgetc(file)) != EOF) {

count++;

}

printf("Number of characters in the file: %d\n", count);

fclose(file);

}

void countWords(char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

printf("Error opening file.\n");

return;

}

int count = 0;

char word[100];

while (fscanf(file, "%s", word) != EOF) {

count++;

}

printf("Number of words in the file: %d\n", count);

fclose(file);

}

void countLines(char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

printf("Error opening file.\n");

return;

}

int count = 0;

char line[MAX\_INPUT\_SIZE];

while (fgets(line, sizeof(line), file) != NULL) {

count++;

}

printf("Number of lines in the file: %d\n", count);

fclose(file);

}

int main() {

char input[MAX\_INPUT\_SIZE];

char \*tokens[MAX\_NUM\_TOKENS];

char \*token;

const char \*delimiter = " \n";

int num\_tokens;

while (1) {

printf("myshell$ ");

if (!fgets(input, MAX\_INPUT\_SIZE, stdin)) {

break;

}

num\_tokens = 0;

token = strtok(input, delimiter);

while (token != NULL && num\_tokens < MAX\_NUM\_TOKENS - 1) {

tokens[num\_tokens] = token;

num\_tokens++;

token = strtok(NULL, delimiter);

}

tokens[num\_tokens] = NULL;

if (strcmp(tokens[0], "count") == 0) {

if (num\_tokens != 3) {

printf("Invalid 'count' command format.\n");

} else {

char \*option = tokens[1];

char \*filename = tokens[2];

if (strcmp(option, "c") == 0) {

countCharacters(filename);

} else if (strcmp(option, "w") == 0) {

countWords(filename);

} else if (strcmp(option, "l") == 0) {

countLines(filename);

} else {

printf("Invalid 'count' command.\n");

}

}

} else {

executeCommand(tokens);

}

}

return 0;

}

* 1. Write the simulation program for demand paging and show the page scheduling and total number of page faults according the Optimal page replacement algorithm. Assume the memory of n frames.

Reference String : 7, 5, 4, 8, 5, 7, 2, 3, 1, 3, 5, 9, 4, 6, 2

write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

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

#define MAX\_PAGES 100 // Maximum number of pages

int search(int page, int memory[], int n\_frames, int start\_index) {

int i;

for (i = start\_index; i < n\_frames; ++i) {

if (memory[i] == page) {

return i;

}

}

return -1;

}

int predictOptimal(int reference\_string[], int n, int memory[], int n\_frames, int current\_index) {

int res = -1, farthest = current\_index;

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

int j;

for (j = current\_index; j < n; ++j) {

if (memory[i] == reference\_string[j]) {

if (j > farthest) {

farthest = j;

res = i;

}

break;

}

}

if (j == n)

return i;

}

return (res == -1) ? 0 : res;

}

int pageFaultsOptimal(int reference\_string[], int n, int n\_frames) {

int memory[MAX\_FRAMES];

int page\_faults = 0;

int i, j;

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

memory[i] = -1;

}

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

int page = reference\_string[i];

// Check if the page is already in memory

int page\_found = search(page, memory, n\_frames, 0);

if (page\_found == -1) {

page\_faults++;

int empty\_frame\_index = search(-1, memory, n\_frames, 0);

if (empty\_frame\_index != -1) {

memory[empty\_frame\_index] = page;

} else {

int index = predictOptimal(reference\_string, n, memory, n\_frames, i + 1);

memory[index] = page;

}

}

printf("Page reference: %d - Memory: ", page);

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

if (memory[j] == -1) {

printf(" - ");

} else {

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

}

}

printf("\n");

}

return page\_faults;

}

int main() {

int reference\_string[] = {7, 5, 4, 8, 5, 7, 2, 3, 1, 3, 5, 9, 4, 6, 2};

int n\_frames = 3; // Number of frames

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

int total\_page\_faults = pageFaultsOptimal(reference\_string, n, n\_frames);

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

return 0;

}

* 1. Write a program to implement shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.

myshell$ search a filename pattern :- To search all the occurrence of

pattern in the file.

myshell$ search c filename pattern :- To count the number of occurrence

of pattern in the file.

write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#define MAX\_INPUT\_SIZE 1024

#define MAX\_TOKEN\_SIZE 64

#define MAX\_NUM\_TOKENS 64

void executeCommand(char \*tokens[]) {

pid\_t pid = fork();

if (pid == -1) {

printf("Failed to fork.\n");

return;

} else if (pid == 0) {

if (execvp(tokens[0], tokens) == -1) {

printf("Command not found.\n");

}

exit(EXIT\_FAILURE);

} else {

wait(NULL);

}

}

void searchAllOccurrences(char \*filename, char \*pattern) {

char command[MAX\_INPUT\_SIZE];

snprintf(command, sizeof(command), "grep -o %s %s | wc -l", pattern, filename);

printf("Occurrences of pattern '%s' in file '%s':\n", pattern, filename);

system(command);

}

void countOccurrences(char \*filename, char \*pattern) {

char command[MAX\_INPUT\_SIZE];

snprintf(command, sizeof(command), "grep -c %s %s", pattern, filename);

printf("Count of occurrences of pattern '%s' in file '%s':\n", pattern, filename);

system(command);

}

int main() {

char input[MAX\_INPUT\_SIZE];

char \*tokens[MAX\_NUM\_TOKENS];

char \*token;

const char \*delimiter = " \n";

int num\_tokens;

while (1) {

printf("myshell$ ");

if (!fgets(input, MAX\_INPUT\_SIZE, stdin)) {

break;

}

num\_tokens = 0;

token = strtok(input, delimiter);

while (token != NULL && num\_tokens < MAX\_NUM\_TOKENS - 1) {

tokens[num\_tokens] = token;

num\_tokens++;

token = strtok(NULL, delimiter);

}

tokens[num\_tokens] = NULL;

if (strcmp(tokens[0], "search") == 0) {

if (num\_tokens != 5) {

printf("Invalid 'search' command format.\n");

} else {

char \*option = tokens[1];

char \*type = tokens[2];

char \*filename = tokens[3];

char \*pattern = tokens[4];

if (strcmp(option, "a") == 0 && strcmp(type, "filename") == 0) {

searchAllOccurrences(filename, pattern);

} else if (strcmp(option, "c") == 0 && strcmp(type, "filename") == 0) {

countOccurrences(filename, pattern);

} else {

printf("Invalid 'search' command.\n");

}

}

} else {

executeCommand(tokens);

}

}

return 0;

}

Q.1. Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LRU page replacement algorithm. Assume the memory of n frames.

Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

write in linux code

#include <stdio.h>

#include <stdlib.h>

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

#define MAX\_PAGES 100 // Maximum number of pages

int search(int page, int memory[], int n\_frames) {

int i;

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

if (memory[i] == page) {

return i;

}

}

return -1;

}

void updateLRU(int indexes[], int n\_frames, int index) {

int i;

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

if (indexes[i] != -1 && indexes[i] < indexes[index]) {

indexes[i]++;

}

}

indexes[index] = 0;

}

int pageFaultsLRU(int reference\_string[], int n, int n\_frames) {

int memory[MAX\_FRAMES];

int indexes[MAX\_FRAMES];

int page\_faults = 0;

int i, j;

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

memory[i] = -1;

indexes[i] = -1;

}

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

int page = reference\_string[i];

// Check if the page is already in memory

int page\_found = search(page, memory, n\_frames);

if (page\_found == -1) {

int empty\_frame\_index = search(-1, memory, n\_frames);

if (empty\_frame\_index != -1) {

memory[empty\_frame\_index] = page;

updateLRU(indexes, n\_frames, empty\_frame\_index);

} else {

int lru\_index = 0;

for (j = 1; j < n\_frames; ++j) {

if (indexes[j] > indexes[lru\_index]) {

lru\_index = j;

}

}

memory[lru\_index] = page;

updateLRU(indexes, n\_frames, lru\_index);

}

page\_faults++;

} else {

updateLRU(indexes, n\_frames, page\_found);

}

printf("Page reference: %d - Memory: ", page);

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

if (memory[j] == -1) {

printf(" - ");

} else {

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

}

}

printf("\n");

}

return page\_faults;

}

int main() {

int reference\_string[] = {8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2};

int n\_frames = 3; // Number of frames

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

int total\_page\_faults = pageFaultsLRU(reference\_string, n, n\_frames);

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

return 0;

}

* 1. Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LFU page replacement algorithm. Assume the memory of n frames. Reference String : 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

write in linux code

#include <stdio.h>

#include <stdlib.h>

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

#define MAX\_PAGES 100 // Maximum number of pages

int search(int page, int memory[], int n\_frames) {

int i;

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

if (memory[i] == page) {

return i;

}

}

return -1;

}

void updateLFU(int frequencies[], int n\_frames, int index) {

frequencies[index]++;

}

int pageFaultsLFU(int reference\_string[], int n, int n\_frames) {

int memory[MAX\_FRAMES];

int frequencies[MAX\_FRAMES];

int page\_faults = 0;

int i, j;

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

memory[i] = -1;

frequencies[i] = 0;

}

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

int page = reference\_string[i];

// Check if the page is already in memory

int page\_found = search(page, memory, n\_frames);

if (page\_found == -1) {

int empty\_frame\_index = search(-1, memory, n\_frames);

if (empty\_frame\_index != -1) {

memory[empty\_frame\_index] = page;

updateLFU(frequencies, n\_frames, empty\_frame\_index);

} else {

int lfu\_index = 0;

for (j = 1; j < n\_frames; ++j) {

if (frequencies[j] < frequencies[lfu\_index]) {

lfu\_index = j;

}

}

memory[lfu\_index] = page;

frequencies[lfu\_index] = 1;

}

page\_faults++;

} else {

updateLFU(frequencies, n\_frames, page\_found);

}

printf("Page reference: %d - Memory: ", page);

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

if (memory[j] == -1) {

printf(" - ");

} else {

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

}

}

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 n\_frames = 3; // Number of frames

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

int total\_page\_faults = pageFaultsLFU(reference\_string, n, n\_frames);

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

return 0;

}

* 1. Write the simulation program for Round Robin scheduling for given time quantum. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give the Gantt chart, turnaround time and waiting time for each process. Also display the average turnaround time and average waiting time.

write in linux code

#include <stdio.h>

#include <stdlib.h>

// Process structure

struct Process {

int process\_id;

int arrival\_time;

int burst\_time;

int remaining\_burst;

int turnaround\_time;

int waiting\_time;

};

void roundRobinScheduling(struct Process processes[], int n, int time\_quantum) {

int remaining\_processes = n;

int current\_time = 0;

while (remaining\_processes > 0) {

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

if (processes[i].remaining\_burst > 0) {

if (processes[i].remaining\_burst > time\_quantum) {

current\_time += time\_quantum;

processes[i].remaining\_burst -= time\_quantum;

} else {

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

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

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

processes[i].remaining\_burst = 0;

remaining\_processes--;

}

}

}

}

}

void displayProcesses(struct Process processes[], int n) {

float avg\_turnaround = 0, avg\_waiting = 0;

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

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].arrival\_time,

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

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

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

}

avg\_turnaround /= n;

avg\_waiting /= n;

printf("\nAverage Turnaround Time: %.2f\nAverage Waiting Time: %.2f\n", avg\_turnaround, avg\_waiting);

}

int main() {

int n, time\_quantum;

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

scanf("%d", &n);

struct Process processes[n];

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

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

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

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

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

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

processes[i].remaining\_burst = processes[i].burst\_time;

}

printf("Enter time quantum: ");

scanf("%d", &time\_quantum);

roundRobinScheduling(processes, n, time\_quantum);

displayProcesses(processes, n);

return 0;

}

* 1. Write a C program to simulate Non-preemptive Shortest Job First (SJF) – scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time

write in linux code

#include <stdio.h>

#include <stdlib.h>

// Process structure

struct Process {

int process\_id;

int arrival\_time;

int burst\_time;

int turnaround\_time;

int waiting\_time;

int completion\_time;

int executed;

};

// Function to calculate average turnaround time and average waiting time

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround, float \*avg\_waiting) {

\*avg\_turnaround = 0;

\*avg\_waiting = 0;

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

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

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

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

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

}

\*avg\_turnaround /= n;

\*avg\_waiting /= n;

}

// Function to perform SJF scheduling

void SJFScheduling(struct Process processes[], int n) {

int currentTime = 0, smallest, count = 0;

while (count < n) {

smallest = -1;

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

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

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

smallest = i;

}

}

}

if (smallest != -1) {

currentTime += processes[smallest].burst\_time;

processes[smallest].completion\_time = currentTime;

processes[smallest].executed = 1;

count++;

} else {

currentTime++;

}

}

}

// Function to display Gantt chart, turnaround time, and waiting time for each process

void displayProcesses(struct Process processes[], int n, float avg\_turnaround, float avg\_waiting) {

printf("Gantt Chart:\n");

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

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

}

printf("|\n");

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

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

printf("%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].turnaround\_time, processes[i].waiting\_time);

}

printf("\nAverage Turnaround Time: %.2f\nAverage Waiting Time: %.2f\n", avg\_turnaround, avg\_waiting);

}

int main() {

int n;

float avg\_turnaround, avg\_waiting;

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

scanf("%d", &n);

struct Process processes[n];

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

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

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

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

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

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

processes[i].executed = 0;

}

SJFScheduling(processes, n);

calculateAverages(processes, n, &avg\_turnaround, &avg\_waiting);

displayProcesses(processes, n, avg\_turnaround, avg\_waiting);

return 0;

}

* 1. Write the program to simulate preemptive Shortest Job First (SJF) – scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time

1. write in linux code

#include <stdio.h>

#include <stdlib.h>

// Process structure

struct Process {

int process\_id;

int arrival\_time;

int burst\_time;

int remaining\_burst;

int turnaround\_time;

int waiting\_time;

int completion\_time;

int executed;

};

// Function to calculate average turnaround time and average waiting time

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround, float \*avg\_waiting) {

\*avg\_turnaround = 0;

\*avg\_waiting = 0;

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

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

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

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

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

}

\*avg\_turnaround /= n;

\*avg\_waiting /= n;

}

// Function to perform SJF scheduling

void SJFScheduling(struct Process processes[], int n) {

int currentTime = 0, smallest, count = 0;

while (count < n) {

smallest = -1;

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

if (processes[i].arrival\_time <= currentTime && processes[i].remaining\_burst > 0) {

if (smallest == -1 || processes[i].remaining\_burst < processes[smallest].remaining\_burst) {

smallest = i;

}

}

}

if (smallest != -1) {

processes[smallest].remaining\_burst--;

if (processes[smallest].remaining\_burst == 0) {

currentTime++;

processes[smallest].completion\_time = currentTime;

processes[smallest].executed = 1;

count++;

}

} else {

currentTime++;

}

}

}

// Function to display Gantt chart, turnaround time, and waiting time for each process

void displayProcesses(struct Process processes[], int n, float avg\_turnaround, float avg\_waiting) {

printf("Gantt Chart:\n");

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

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

}

printf("|\n");

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

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

printf("%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].turnaround\_time, processes[i].waiting\_time);

}

printf("\nAverage Turnaround Time: %.2f\nAverage Waiting Time: %.2f\n", avg\_turnaround, avg\_waiting);

}

int main() {

int n;

float avg\_turnaround, avg\_waiting;

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

scanf("%d", &n);

struct Process processes[n];

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

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

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

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

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

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

processes[i].remaining\_burst = processes[i].burst\_time;

processes[i].executed = 0;

}

SJFScheduling(processes, n);

calculateAverages(processes, n, &avg\_turnaround, &avg\_waiting);

displayProcesses(processes, n, avg\_turnaround, avg\_waiting);

return 0;

}

* 1. Write the program to simulate Non preemptive priority scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. 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>

// Process structure

struct Process {

int process\_id;

int arrival\_time;

int burst\_time;

int priority;

int turnaround\_time;

int waiting\_time;

};

// Function to calculate average turnaround time and average waiting time

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround, float \*avg\_waiting) {

\*avg\_turnaround = 0;

\*avg\_waiting = 0;

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

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

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

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

}

\*avg\_turnaround /= n;

\*avg\_waiting /= n;

}

// Function to perform Non-preemptive Priority Scheduling

void priorityScheduling(struct Process processes[], int n) {

// Sort processes based on priority

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

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

if (processes[j].priority > processes[j + 1].priority) {

// Swap processes

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

// Calculate waiting time for each process

processes[0].waiting\_time = 0;

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

processes[i].waiting\_time = processes[i - 1].waiting\_time + processes[i - 1].burst\_time;

}

// Calculate turnaround time for each process

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

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

}

}

// Function to display Gantt chart, turnaround time, and waiting time for each process

void displayProcesses(struct Process processes[], int n, float avg\_turnaround, float avg\_waiting) {

printf("Gantt Chart:\n");

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

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

}

printf("|\n");

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

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

printf("%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].turnaround\_time, processes[i].waiting\_time);

}

printf("\nAverage Turnaround Time: %.2f\nAverage Waiting Time: %.2f\n", avg\_turnaround, avg\_waiting);

}

int main() {

int n;

float avg\_turnaround, avg\_waiting;

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

scanf("%d", &n);

struct Process processes[n];

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

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

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

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

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

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

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

scanf("%d", &processes[i].priority);

}

priorityScheduling(processes, n);

calculateAverages(processes, n, &avg\_turnaround, &avg\_waiting);

displayProcesses(processes, n, avg\_turnaround, avg\_waiting);

return 0;

}

* 1. Write the program to simulate FCFS CPU-scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

1. write in linux code

#include <stdio.h>

#include <stdlib.h>

// Process structure

struct Process {

int process\_id;

int arrival\_time;

int burst\_time;

int turnaround\_time;

int waiting\_time;

int completion\_time;

};

// Function to calculate average turnaround time and average waiting time

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround, float \*avg\_waiting) {

\*avg\_turnaround = 0;

\*avg\_waiting = 0;

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

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

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

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

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

}

\*avg\_turnaround /= n;

\*avg\_waiting /= n;

}

// Function to perform FCFS scheduling

void fcfsScheduling(struct Process processes[], int n) {

int currentTime = 0;

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

// Update completion time

processes[i].completion\_time = currentTime + processes[i].burst\_time;

// Update current time for the next process

currentTime = processes[i].completion\_time;

}

}

// Function to display Gantt chart, turnaround time, and waiting time for each process

void displayProcesses(struct Process processes[], int n, float avg\_turnaround, float avg\_waiting) {

printf("Gantt Chart:\n");

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

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

}

printf("|\n");

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

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

printf("%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].turnaround\_time, processes[i].waiting\_time);

}

printf("\nAverage Turnaround Time: %.2f\nAverage Waiting Time: %.2f\n", avg\_turnaround, avg\_waiting);

}

int main() {

int n;

float avg\_turnaround, avg\_waiting;

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

scanf("%d", &n);

struct Process processes[n];

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

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

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

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

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

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

}

fcfsScheduling(processes, n);

calculateAverages(processes, n, &avg\_turnaround, &avg\_waiting);

displayProcesses(processes, n, avg\_turnaround, avg\_waiting);

return 0;

}

* 1. Write a C program to simulate FCFS CPU-scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

1. write in linux code

#include <stdio.h>

#include <stdlib.h>

// Process structure

struct Process {

int process\_id;

int arrival\_time;

int burst\_time;

int turnaround\_time;

int waiting\_time;

int completion\_time;

};

// Function to calculate average turnaround time and average waiting time

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround, float \*avg\_waiting) {

\*avg\_turnaround = 0;

\*avg\_waiting = 0;

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

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

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

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

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

}

\*avg\_turnaround /= n;

\*avg\_waiting /= n;

}

// Function to perform FCFS scheduling

void fcfsScheduling(struct Process processes[], int n) {

int currentTime = 0;

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

// Update completion time

processes[i].completion\_time = currentTime + processes[i].burst\_time;

// Update current time for the next process

currentTime = processes[i].completion\_time;

}

}

// Function to display Gantt chart, turnaround time, and waiting time for each process

void displayProcesses(struct Process processes[], int n, float avg\_turnaround, float avg\_waiting) {

printf("Gantt Chart:\n");

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

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

}

printf("|\n");

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

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

printf("%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].turnaround\_time, processes[i].waiting\_time);

}

printf("\nAverage Turnaround Time: %.2f\nAverage Waiting Time: %.2f\n", avg\_turnaround, avg\_waiting);

}

int main() {

int n;

float avg\_turnaround, avg\_waiting;

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

scanf("%d", &n);

struct Process processes[n];

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

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

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

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

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

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

}

fcfsScheduling(processes, n);

calculateAverages(processes, n, &avg\_turnaround, &avg\_waiting);

displayProcesses(processes, n, avg\_turnaround, avg\_waiting);

return 0;

}

* 1. Write a C Program to create a child process using fork (), display parent and child process id. Child process will display the message “I am Child Process” and the parent process should display “I am Parent Process”.

1. write in linux code

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid;

// Fork a child process

pid = fork();

if (pid < 0) {

// Fork failed

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

return 1;

} else if (pid == 0) {

// Child process

printf("I am Child Process. My Process ID is %d\n", getpid());

} else {

// Parent process

printf("I am Parent Process. My Process ID is %d\n", getpid());

}

return 0;

}

Q1,Write a C program to simulate 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.

write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Process structure

struct Process {

int process\_id;

int arrival\_time;

int cpu\_burst;

int priority;

int remaining\_burst;

int turnaround\_time;

int waiting\_time;

int completion\_time;

};

// Function to generate a random burst time

int generateRandomBurst() {

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

}

// Function to calculate average turnaround time and average waiting time

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround, float \*avg\_waiting) {

\*avg\_turnaround = 0;

\*avg\_waiting = 0;

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

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

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

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

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

}

\*avg\_turnaround /= n;

\*avg\_waiting /= n;

}

// Function to perform Preemptive Priority Scheduling

void preemptivePriorityScheduling(struct Process processes[], int n) {

int currentTime = 0;

int completed = 0;

while (completed < n) {

int highestPriority = -1;

int chosenProcess = -1;

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

if (processes[i].arrival\_time <= currentTime && processes[i].remaining\_burst > 0) {

if (highestPriority == -1 || processes[i].priority < processes[highestPriority].priority) {

highestPriority = i;

chosenProcess = i;

}

}

}

if (chosenProcess != -1) {

// Execute the process for 1 time unit

processes[chosenProcess].remaining\_burst--;

// Check if the process is completed

if (processes[chosenProcess].remaining\_burst == 0) {

currentTime++;

processes[chosenProcess].completion\_time = currentTime;

processes[chosenProcess].waiting\_time -= 2; // Subtract fixed I/O waiting time

processes[chosenProcess].remaining\_burst = generateRandomBurst(); // Generate new random burst

completed++;

}

} else {

currentTime++;

}

}

}

// Function to display Gantt chart, turnaround time, and waiting time for each process

void displayProcesses(struct Process processes[], int n, float avg\_turnaround, float avg\_waiting) {

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

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

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

}

printf("|\n");

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

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

printf("%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].turnaround\_time, processes[i].waiting\_time);

}

printf("\nAverage Turnaround Time: %.2f\nAverage Waiting Time: %.2f\n", avg\_turnaround, avg\_waiting);

}

int main() {

srand(time(NULL)); // Seed for random number generation

int n;

float avg\_turnaround, avg\_waiting;

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

scanf("%d", &n);

struct Process processes[n];

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

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

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

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

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

scanf("%d", &processes[i].priority);

processes[i].cpu\_burst = generateRandomBurst();

processes[i].remaining\_burst = processes[i].cpu\_burst;

}

preemptivePriorityScheduling(processes, n);

calculateAverages(processes, n, &avg\_turnaround, &avg\_waiting);

displayProcesses(processes, n, avg\_turnaround, avg\_waiting);

return 0;

}

* 1. Write a C 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.

1. write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main() {

pid\_t pid;

// Fork a child process

pid = fork();

if (pid < 0) {

// Fork failed

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

return 1;

} else if (pid == 0) {

// Child process

printf("Child Process: My PID is %d. Nice value before: %d\n", getpid(), nice(0));

// Increase priority using nice

if (nice(10) == -1) {

perror("Nice failed");

exit(EXIT\_FAILURE);

}

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

} else {

// Parent process

printf("Parent Process: My PID is %d. Nice value before: %d\n", getpid(), nice(0));

// Wait for the child process to complete

wait(NULL);

printf("Parent Process: Nice value after: %d\n", nice(0));

}

return 0;

}

* 1. Write a C program to illustrate the concept of orphan process. Parent process creates a child and terminates before child has finished its task. So child process becomes orphan process. (Use fork(), sleep(), getpid(), getppid()).

1. write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main() {

pid\_t pid;

// Fork a child process

pid = fork();

if (pid < 0) {

// Fork failed

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

return 1;

} else if (pid == 0) {

// Child process

printf("Child Process: My PID is %d. Nice value before: %d\n", getpid(), nice(0));

// Increase priority using nice

if (nice(10) == -1) {

perror("Nice failed");

exit(EXIT\_FAILURE);

}

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

} else {

// Parent process

printf("Parent Process: My PID is %d. Nice value before: %d\n", getpid(), nice(0));

// Wait for the child process to complete

wait(NULL);

printf("Parent Process: Nice value after: %d\n", nice(0));

}

return 0;

}

* 1. Write a C program to accept n integers to be sorted. Main function

creates child process using fork system call. Parent process sorts the integers using bubble sort and waits for child process using wait system call. Child process sorts the integers using insertion sort.

write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

// Function to perform Bubble Sort

void bubbleSort(int arr[], int n) {

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

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

if (arr[j] > arr[j + 1]) {

// Swap elements if they are in the wrong order

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// Function to perform Insertion Sort

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1] that are greater than key to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

int main() {

int n;

// Accept the number of integers

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

scanf("%d", &n);

int arr[n];

// Accept the integers to be sorted

printf("Enter the integers to be sorted:\n");

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

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

}

// Fork a child process

pid\_t pid = fork();

if (pid < 0) {

// Fork failed

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

return 1;

} else if (pid == 0) {

// Child process

printf("Child Process: Sorting using Insertion Sort\n");

insertionSort(arr, n);

printf("Child Process: Sorted array: ");

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

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

}

printf("\n");

} else {

// Parent process

printf("Parent Process: Sorting using Bubble Sort\n");

bubbleSort(arr, n);

printf("Parent Process: Sorted array: ");

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

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

}

printf("\n");

// Wait for the child process to complete

wait(NULL);

}

return 0;

}

* 1. Write a C program to implement the toy shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.

count c filename :- To print number of characters in the file. count w filename :- To print number of words in the file. count l filename :- To print number of lines in the file.

write in linux code

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#define MAX\_COMMAND\_LENGTH 100

#define MAX\_TOKENS 10

// Function to count characters in a file

void countCharacters(char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file");

return;

}

int count = 0;

int ch;

while ((ch = fgetc(file)) != EOF) {

count++;

}

fclose(file);

printf("Number of characters in %s: %d\n", filename, count);

}

// Function to count words in a file

void countWords(char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file");

return;

}

int count = 0;

int inWord = 0;

int ch;

while ((ch = fgetc(file)) != EOF) {

if (ch == ' ' || ch == '\t' || ch == '\n') {

inWord = 0;

} else if (inWord == 0) {

inWord = 1;

count++;

}

}

fclose(file);

printf("Number of words in %s: %d\n", filename, count);

}

// Function to count lines in a file

void countLines(char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file");

return;

}

int count = 0;

int ch;

while ((ch = fgetc(file)) != EOF) {

if (ch == '\n') {

count++;

}

}

fclose(file);

printf("Number of lines in %s: %d\n", filename, count);

}

int main() {

char command[MAX\_COMMAND\_LENGTH];

char \*tokens[MAX\_TOKENS];

char \*token;

while (1) {

// Display command prompt

printf("myshell$ ");

fflush(stdout);

// Read the command

if (fgets(command, sizeof(command), stdin) == NULL) {

break;

}

// Remove newline character

command[strcspn(command, "\n")] = '\0';

// Tokenize the command

int token\_count = 0;

token = strtok(command, " ");

while (token != NULL && token\_count < MAX\_TOKENS - 1) {

tokens[token\_count] = token;

token\_count++;

token = strtok(NULL, " ");

}

tokens[token\_count] = NULL;

// Check for exit command

if (strcmp(tokens[0], "exit") == 0) {

break;

}

// Check for count command

if (strcmp(tokens[0], "count") == 0) {

if (token\_count >= 3) {

if (strcmp(tokens[1], "c") == 0) {

countCharacters(tokens[2]);

} else if (strcmp(tokens[1], "w") == 0) {

countWords(tokens[2]);

} else if (strcmp(tokens[1], "l") == 0) {

countLines(tokens[2]);

} else {

printf("Invalid count command format\n");

}

} else {

printf("Invalid count command format\n");

}

continue;

}

// Fork a child process

pid\_t pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

return 1;

} else if (pid == 0) {

// Child process

// Execute the given command

execvp(tokens[0], tokens);

// If execvp fails

perror("Command execution failed");

exit(EXIT\_FAILURE);

} else {

// Parent process

// Wait for the child process to complete

wait(NULL);

}

}

return 0;

}

* 1. Write a C program that accepts an integer array. Main function forks child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of execve() system call. The child process uses execve() system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.

1. write in linux code

//Parent class 1 part

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

// Function to perform binary search

void binarySearch(int arr[], int size, int key) {

int low = 0, high = size - 1, mid;

while (low <= high) {

mid = (low + high) / 2;

if (arr[mid] == key) {

printf("Item %d found at index %d\n", key, mid);

return;

} else if (arr[mid] < key) {

low = mid + 1;

} else {

high = mid - 1;

}

}

printf("Item %d not found in the array\n", key);

}

int main() {

int n;

// Accept the number of elements in the array

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

int arr[n];

// Accept the elements of the array

printf("Enter %d elements in the array:\n", n);

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

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

}

// Fork a child process

pid\_t pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

return 1;

} else if (pid == 0) {

// Child process

// Convert array elements to strings for execve arguments

char \*args[n + 3]; // +3 for program name, array size, and NULL

args[0] = "./child";

args[1] = (char \*)malloc(10); // To pass the size of the array

args[2] = NULL;

sprintf(args[1], "%d", n);

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

args[i + 2] = (char \*)malloc(10);

sprintf(args[i + 2], "%d", arr[i]);

}

// Execute the child program

execve(args[0], args, NULL);

// If execve fails

perror("Child: Execve failed");

exit(EXIT\_FAILURE);

} else {

// Parent process

// Wait for the child process to complete

wait(NULL);

}

return 0;

}

// Child Class 2 part

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[]) {

if (argc < 3) {

fprintf(stderr, "Usage: %s <array\_size> <sorted\_array\_elements...>\n", argv[0]);

return 1;

}

int size = atoi(argv[1]);

int sortedArr[size];

// Convert command line arguments to integers

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

sortedArr[i] = atoi(argv[i + 2]);

}

// Perform binary search on the sorted array

int key;

printf("Enter the item to search in the array: ");

scanf("%d", &key);

// Binary search

int low = 0, high = size - 1, mid;

while (low <= high) {

mid = (low + high) / 2;

if (sortedArr[mid] == key) {

printf("Child: Item %d found at index %d\n", key, mid);

return 0;

} else if (sortedArr[mid] < key) {

low = mid + 1;

} else {

high = mid - 1;

}

}

printf("Child: Item %d not found in the array\n", key);

return 0;

}