|  |  |
| --- | --- |
| **NAME:** Kanko Ghosh  **CLASS:** BCSE III  **SEM:** FIRST | **CLASS ROLL NO.:** 002010501035  **SESSION:** 2022-2023  **ASSIGNMENT SET:**  2 |

**Question 1**:

Design a CPU scheduler for jobs whose execution profiles will be in a file that is to be read and appropriate scheduling algorithm to be chosen by the scheduler.

Format of the profile:

<Job id> <priority> <arrival time> <CPU burst(1) I/O burst(1) CPU burst(2) ……. >-1

(Each information is separated by blank space and each job profile ends with -1. Lesser priority number denotes higher priority process with priority number 1 being the process with highest priority.)

Example: 2 3 0 100 2 200 3 25 -1 1 1 4 60 10 ….. -1 etc.

Testing:

a. Create job profiles for 20 jobs and use three different scheduling algorithms (FCFS,

preemptive Priority and Round Robin (time slice:20)).

b. Compare the average waiting time, turnaround time of each process for the different scheduling algorithms

**Code: FCFS**

#include<stdio.h>

#include<stdlib.h>

typedef struct Queue {

int front, rear, size;

unsigned capacity;

int\* array;

} Queue;

Queue\* createQueue(unsigned capacity) {

Queue\* queue = (Queue \*)malloc(sizeof(Queue));

queue->capacity = capacity;

queue->front = queue->size = 0;

queue->rear = capacity - 1;

queue->array = (int \*)malloc(queue->capacity\*sizeof(int));

return queue;

}

int isFull(Queue\* queue) {

return (queue->size == queue->capacity);

}

int isEmpty(Queue\* queue) {

return (queue->size == 0);

}

void enqueue(Queue\* queue, int item) {

if (isFull(queue)) return;

queue->rear = (queue->rear + 1) % queue->capacity;

queue->array[queue->rear] = item;

queue->size = queue->size + 1;

}

int dequeue(Queue\* queue) {

if (isEmpty(queue))

return -1;

int item = queue->array[queue->front];

queue->front = (queue->front + 1)

% queue->capacity;

queue->size = queue->size - 1;

return item;

}

int front(Queue\* queue) {

if (isEmpty(queue))

return -1;

return queue->array[queue->front];

}

int rear(Queue\* queue) {

if (isEmpty(queue))

return -1;

return queue->array[queue->rear];

}

typedef struct Process {

int id;

int priority;

int AT;

int bt[100];

int ptr;

} Process;

int main(){

int sum\_wt = 0;

int sum\_tt = 0;

freopen("output.txt", "r", stdin);

freopen("output\_fcfs.txt", "w", stdout);

int no\_of\_processes;

scanf("%d", &no\_of\_processes);

Process ps[no\_of\_processes];

for (int i = 0; i < no\_of\_processes; i++){

scanf("%d", &ps[i].id);

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

scanf("%d", &ps[i].AT);

sum\_tt -= ps[i].AT;

int cnt = -1;

do {

cnt++;

int temp;

scanf("%d", &temp);

ps[i].bt[cnt] = temp;

sum\_wt -= temp;

} while (ps[i].bt[cnt] != -1);

ps[i].ptr = 0;

}

////// input done

////// make gant chart now!!

Queue \* FCFS = createQueue(no\_of\_processes);

for (int i = 0; i < no\_of\_processes; i++){

if (ps[i].AT == 0) {

enqueue(FCFS, i);

}

}

int completed = 0;

int time = 0;

while (completed != no\_of\_processes){

if (isEmpty(FCFS)){

printf("Time Stamp: %d: Idle CPU\n", time);

time++;

continue;

}

int pid = front(FCFS);

ps[pid].bt[ps[pid].ptr] --;

printf("Time Stamp: %d, executed: %d\n", time, pid);

if (ps[pid].bt[ps[pid].ptr] == 0){

dequeue(FCFS);

if (ps[pid].bt[(ps[pid].ptr)+1] == -1 || ps[pid].bt[(ps[pid].ptr)+2] == -1){

completed++;

printf("Completed %d\n", completed);

sum\_tt += time;

ps[pid].AT = \_\_INT\_MAX\_\_;

} else {

int io\_time = ps[pid].bt[(ps[pid].ptr)+1];

ps[pid].AT = time + io\_time;

ps[pid].ptr+=2;

}

}

time++;

for (int i = 0; i < no\_of\_processes; i++){

if (ps[i].AT == time) {

enqueue(FCFS, i);

}

}

}

printf("Average Waiting Time: %f\n", ((float)(sum\_tt-sum\_wt))/no\_of\_processes);

printf("Average Turnaround Time: %f\n", ((float)sum\_tt)/no\_of\_processes);

}

|  |  |
| --- | --- |
| Input | Output |
| 20  0 10 17 192 215 231 344 370 -1  1 8 144 320 376 408 427 482 -1  2 7 75 137 185 368 452 488 -1  3 2 9 96 126 302 357 452 -1  4 9 59 80 151 253 446 494 -1  5 1 0 184 236 255 387 395 -1  6 2 42 167 201 217 255 258 -1  7 0 249 292 337 417 454 481 -1  8 5 119 205 237 353 456 492 -1  9 5 63 219 255 326 406 414 -1  10 8 63 82 263 340 365 429 -1  11 3 162 253 255 314 402 468 -1  12 0 49 158 235 307 314 431 -1  13 5 15 146 171 178 200 302 -1  14 0 87 114 124 219 469 494 -1  15 8 7 65 202 286 344 385 -1  16 0 0 12 307 422 422 441 -1  17 2 404 407 420 422 444 486 -1  18 3 23 118 139 163 344 407 -1  19 6 6 42 145 198 299 489 -1 | …… …… Time Stamp: 17914, executed: 17  Time Stamp: 17915, executed: 17  Time Stamp: 17916, executed: 17  Time Stamp: 17917, executed: 17  Time Stamp: 17918, executed: 17  Time Stamp: 17919, executed: 17  Time Stamp: 17920, executed: 17  Time Stamp: 17921, executed: 17  Time Stamp: 17922, executed: 17  Time Stamp: 17923, executed: 17  Time Stamp: 17924, executed: 17  Time Stamp: 17925, executed: 17  Completed 20  Average Waiting Time: 15050.849609  Average Turnaround Time: 13547.200195 |

**PseudoCode: Premptive Priority**

int main(){

int sum\_wt = 0;

int sum\_tt = 0;

freopen("output\_pp.txt", "r", stdin);

freopen("output\_preemptive.txt", "w", stdout);

// ------------------------

// input same as previous

// ------------------------

////// make gant chart now!!

int completed = 0;

int time = 0;

while (completed != no\_of\_processes){

int i = 0;

for (; i < no\_of\_processes; i++){

if (ps[i].AT <= time){

ps[i].bt[ps[i].ptr] --;

printf("Time Stamp: %d, executed: %d\n", time, ps[i].id);

if (ps[i].bt[ps[i].ptr] == 0){

if (ps[i].bt[(ps[i].ptr)+1] == -1 || ps[i].bt[(ps[i].ptr)+2] == -1){

completed++;

sum\_tt += time;

ps[i].AT = \_\_INT\_MAX\_\_;

} else {

int io\_time = ps[i].bt[(ps[i].ptr)+1];

ps[i].AT = time + io\_time;

ps[i].ptr += 2;

}

}

break;

}

}

if (i == no\_of\_processes){

printf("Time Stamp: %d: Idle CPU\n", time);

}

time++;

}

printf("Average Waiting Time: %f\n", ((float)(sum\_tt-sum\_wt))/no\_of\_processes);

printf("Average Turnaround Time: %f\n", ((float)sum\_tt)/no\_of\_processes);

}

|  |  |
| --- | --- |
| Input | Output |
| 20  7 0 249 292 337 417 454 481 -1  12 0 49 158 235 307 314 431 -1  14 0 87 114 124 219 469 494 -1  16 0 0 12 307 422 422 441 -1  5 1 0 184 236 255 387 395 -1  3 2 9 96 126 302 357 452 -1  6 2 42 167 201 217 255 258 -1  17 2 404 407 420 422 444 486 -1  18 3 23 118 139 163 344 407 -1  11 3 162 253 255 314 402 468 -1  8 5 119 205 237 353 456 492 -1  9 5 63 219 255 326 406 414 -1  13 5 15 146 171 178 200 302 -1  19 6 6 42 145 198 299 489 -1  2 7 75 137 185 368 452 488 -1  10 8 63 82 263 340 365 429 -1  15 8 7 65 202 286 344 385 -1  1 8 144 320 376 408 427 482 -1  4 9 59 80 151 253 446 494 -1  0 10 17 192 215 231 344 370 -1 | …… …… Time Stamp: 18467, executed: 0  Time Stamp: 18468, executed: 0  Time Stamp: 18469, executed: 0  Time Stamp: 18470, executed: 0  Time Stamp: 18471, executed: 0  Time Stamp: 18472, executed: 0  Time Stamp: 18473, executed: 0  Time Stamp: 18474, executed: 0  Time Stamp: 18475, executed: 0  Time Stamp: 18476, executed: 0  Time Stamp: 18477, executed: 0  Time Stamp: 18478, executed: 0  Time Stamp: 18479, executed: 0  Time Stamp: 18480, executed: 0  Time Stamp: 18481, executed: 0  Time Stamp: 18482, executed: 0  Average Waiting Time: 11413.299805  Average Turnaround Time: 9909.650391 |

**Code: FCFS**

// Queue Data structure import

#define TIME\_QUANTUM 20

int main(){

int sum\_wt = 0;

int sum\_tt = 0;

// ------------------------

// input same as previous

// ------------------------

////// input done

////// make gant chart now!!

Queue \* FCFS = createQueue(no\_of\_processes);

for (int i = 0; i < no\_of\_processes; i++){

if (ps[i].AT == 0) {

enqueue(FCFS, i);

}

}

int completed = 0;

int time = 0;

while (completed != no\_of\_processes){

if (isEmpty(FCFS)){

printf("Time Stamp: %d: Idle CPU\n", time);

time++;

for (int j = 0; j < no\_of\_processes; j++){

if (ps[j].AT == time) {

enqueue(FCFS, j);

}

}

continue;

}

int i = 0;

for (; i < TIME\_QUANTUM; i++){

int pid = front(FCFS);

ps[pid].bt[ps[pid].ptr] --;

printf("Time Stamp: %d, executed: %d\n", time, ps[pid].id);

if (ps[pid].bt[ps[pid].ptr] == 0){

dequeue(FCFS);

if (ps[pid].bt[(ps[pid].ptr)+1] == -1 || ps[pid].bt[(ps[pid].ptr)+2] == -1){

completed++;

sum\_tt += time;

printf("Completed %d\n", completed);

ps[pid].AT = \_\_INT\_MAX\_\_;

} else {

int io\_time = ps[pid].bt[(ps[pid].ptr)+1];

ps[pid].AT = time + io\_time;

ps[pid].ptr+=2;

}

time++;

for (int j = 0; j < no\_of\_processes; j++){

if (ps[j].AT == time) {

printf("added!");

enqueue(FCFS, j);

}

}

break;

}

time++;

for (int j = 0; j < no\_of\_processes; j++){

if (ps[j].AT == time) {

printf("added!");

enqueue(FCFS, j);

}

}

}

// if (time == 900) return 1;

if (i == TIME\_QUANTUM){

int temp = dequeue(FCFS);

enqueue(FCFS, temp);

}

getchar();

}

printf("Average Waiting Time: %f\n", ((float)(sum\_tt-sum\_wt))/no\_of\_processes);

printf("Average Turnaround Time: %f\n", ((float)sum\_tt)/no\_of\_processes);

}

|  |  |
| --- | --- |
| Input | Output |
| 20  0 10 17 192 215 231 344 370 -1  1 8 144 320 376 408 427 482 -1  2 7 75 137 185 368 452 488 -1  3 2 9 96 126 302 357 452 -1  4 9 59 80 151 253 446 494 -1  5 1 0 184 236 255 387 395 -1  6 2 42 167 201 217 255 258 -1  7 0 249 292 337 417 454 481 -1  8 5 119 205 237 353 456 492 -1  9 5 63 219 255 326 406 414 -1  10 8 63 82 263 340 365 429 -1  11 3 162 253 255 314 402 468 -1  12 0 49 158 235 307 314 431 -1  13 5 15 146 171 178 200 302 -1  14 0 87 114 124 219 469 494 -1  15 8 7 65 202 286 344 385 -1  16 0 0 12 307 422 422 441 -1  17 2 404 407 420 422 444 486 -1  18 3 23 118 139 163 344 407 -1  19 6 6 42 145 198 299 489 -1 | …… …… Time Stamp: 17914, executed: 17  Time Stamp: 17915, executed: 17  Time Stamp: 17916, executed: 17  Time Stamp: 17917, executed: 17  Time Stamp: 17918, executed: 17  Time Stamp: 17919, executed: 17  Time Stamp: 17920, executed: 17  Time Stamp: 17921, executed: 17  Time Stamp: 17922, executed: 17  Time Stamp: 17923, executed: 17  Time Stamp: 17924, executed: 17  Time Stamp: 17925, executed: 17  Completed 20  Average Waiting Time: 17137.150391  Average Turnaround Time: 15633.500000 |

**Observation:** waiting and turnaround time (average) for the 3 schedules are as follows: Preemptive < FCFS < Round Robin. Preemptive Priority has least, since convoy effect is not visible as in FCFS. Since the burst time consists of IO and CPU, Round robin performed worse, since idle times were visible in case of round robin, but not in FCFS.

Eg: if time quantum for round robin is 1 unit, CPU and IO burst of 5 unit for 2 processes each, then in case of FCFS, CPU bursts of P1 is executed then P2 CPU bursts, along with IO cycles of P1. But in Round robin, both P1 and P2 performs IO bursts together, and then CPU remains idle

**Considerations**:

* IO cycle time = CPU cycle time.
* For preemptive priority, the input process list is sorted according to the priority

**Question 2**

2. Create child processes: X and Y.

a. Each child process performs 10 iterations. The child process displays its name/id and the current iteration number, and sleeps for some random amount of time. Adjust the sleeping duration of the processes to have different outputs (i.e. another interleaving of processes’ traces).

b. Modify the program so that X is not allowed to start iteration *i* before process Y has terminated its own iteration *i-1*. Use semaphore to implement this synchronization.

c. Modify the program so that X and Y now perform in lockstep [both perform iteration I, then iteration i+1, and so on] with the condition mentioned in Q (2b) above.

d. Add another child process Z.

Perform the operations as mentioned in Q (2a) for all three children.

Then perform the operations as mentioned in Q (2c) [that is, 3 children in lockstep].

**Code: (a + d)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <time.h>

#include <semaphore.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <sys/wait.h>

#define SLEEP 0.1

#define NO\_OF\_PROCESS 3

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

{

    int parent\_pid = getpid();

    //2 processes

    pid\_t \*child = (pid\_t \*)malloc(NO\_OF\_PROCESS \* sizeof(pid\_t));

    for (int i = 0; i < NO\_OF\_PROCESS; i++){

        child[i] = fork();

        if (child[i] < 0) {

            //error

            printf("Forking failed\n");

            printf("Child no: %d", i);

            return 1;

        } else if (child[i] == 0){

            /\*\*

             \* @brief

             \* child ->

             \* do the loop here

             \* 10 iterations

             \*/

            srand(getpid());

            for (int j = 0; j < 10; j++){

                printf("Process PID: %d\tParent PID: %d\tIteration count: %d\n", getpid(), getppid(), j);

                sleep(rand()%10);

            }

            break;

        } else {

            /\*\*

             \* @brief

             \* parent ->

             \* print details

             \*/

            printf("Child process: %d, pid: %d \n", i, child[i]);

        }

    }

    if (getpid() == parent\_pid){

        int v;

        for (int i = 0; i < 2; i++){

            waitpid(child[i], &v, 0);

        }

        free(child);

    }

    return 0;

}

**Output: (a+d)**

Child process: 0, pid: 114

Process PID: 114 Parent PID: 113 Iteration count: 0

Child process: 1, pid: 115

Process PID: 115 Parent PID: 113 Iteration count: 0

Child process: 2, pid: 116

Process PID: 116 Parent PID: 113 Iteration count: 0

Process PID: 114 Parent PID: 113 Iteration count: 1

Process PID: 114 Parent PID: 113 Iteration count: 2

Process PID: 114 Parent PID: 113 Iteration count: 3

Process PID: 116 Parent PID: 113 Iteration count: 1

Process PID: 116 Parent PID: 113 Iteration count: 2

Process PID: 115 Parent PID: 113 Iteration count: 1

Process PID: 114 Parent PID: 113 Iteration count: 4

Process PID: 116 Parent PID: 113 Iteration count: 3

Process PID: 115 Parent PID: 113 Iteration count: 2

Process PID: 115 Parent PID: 113 Iteration count: 3

Process PID: 114 Parent PID: 113 Iteration count: 5

Process PID: 116 Parent PID: 113 Iteration count: 4

Process PID: 115 Parent PID: 113 Iteration count: 4

Process PID: 115 Parent PID: 113 Iteration count: 5

Process PID: 115 Parent PID: 113 Iteration count: 6

Process PID: 115 Parent PID: 113 Iteration count: 7

Process PID: 115 Parent PID: 113 Iteration count: 8

Process PID: 114 Parent PID: 113 Iteration count: 6

Process PID: 114 Parent PID: 113 Iteration count: 7

Process PID: 116 Parent PID: 113 Iteration count: 5

Process PID: 116 Parent PID: 113 Iteration count: 6

Process PID: 115 Parent PID: 113 Iteration count: 9

Process PID: 114 Parent PID: 113 Iteration count: 8

Process PID: 116 Parent PID: 113 Iteration count: 7

Process PID: 116 Parent PID: 113 Iteration count: 8

Process PID: 114 Parent PID: 113 Iteration count: 9

**Explanation and PseudoCode: (b+d):** For p number of processes and n number of iterations,

(p-1)\*n number of semaphores are used (all initialised with zero value). Let the processes be P1 to Pn. P1 will go on without any restriction, so it will not be associated to any other semaphores. When it executes the xth iteration, it signals the semaphore [0][x], which allows the next process. Thus, on execution of xth iteration by Py, it signals [y-1][x], which allows the Py+1 to execute (x+1)th iteration

#define SLEEP 0.1

#define NO\_OF\_PROCESS 3

#define NO\_OF\_ITERATION 10

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

{

    int parent\_pid = getpid();

    pid\_t \*child;

    sem\_t \*\*\*sem;

    for (int i = 0; i < NO\_OF\_PROCESS-1; i++){

        for (int j = 0; j < NO\_OF\_ITERATION; j++){

            sem[i][j] = sem\_open(strcat(pcnt, itcnt), O\_CREAT, 0660, 0);

        }

    }

    for (int i = 0; i < NO\_OF\_PROCESS; i++){

        child[i] = fork();

        if (child[i] < 0) {

            printf("Forking failed\n");

            printf("Child no: %d", i);

            return 1;

        } else if (child[i] == 0){

            srand(getpid());

            for (int j = 0; j < 10; j++){

                if (i != 0){

                    sem\_wait(sem[i-1][j]);

                }

                    sleep(rand()%10);

                printf("Process PID: %d\tParent PID: %d\tIteration count: %d\n", getpid(), getppid(), j);

                if (i != NO\_OF\_PROCESS - 1){

                    sem\_post(sem[i][j]);

                }

            }

            break;

        } else {

            printf("Child process: %d, pid: %d \n", i, child[i]);

        }

    }

}

**Output: (b+d)**

Child process: 0, pid: 141

Process PID: 141 Parent PID: 140 Iteration count: 0

Child process: 1, pid: 142

Process PID: 142 Parent PID: 140 Iteration count: 0

Child process: 2, pid: 143

Process PID: 143 Parent PID: 140 Iteration count: 0

Process PID: 141 Parent PID: 140 Iteration count: 1

Process PID: 142 Parent PID: 140 Iteration count: 1

Process PID: 143 Parent PID: 140 Iteration count: 1

Process PID: 141 Parent PID: 140 Iteration count: 2

Process PID: 142 Parent PID: 140 Iteration count: 2

Process PID: 143 Parent PID: 140 Iteration count: 2

Process PID: 141 Parent PID: 140 Iteration count: 3

Process PID: 142 Parent PID: 140 Iteration count: 3

Process PID: 143 Parent PID: 140 Iteration count: 3

Process PID: 141 Parent PID: 140 Iteration count: 4

Process PID: 142 Parent PID: 140 Iteration count: 4

Process PID: 143 Parent PID: 140 Iteration count: 4

Process PID: 141 Parent PID: 140 Iteration count: 5

Process PID: 142 Parent PID: 140 Iteration count: 5

Process PID: 143 Parent PID: 140 Iteration count: 5

Process PID: 141 Parent PID: 140 Iteration count: 6

Process PID: 141 Parent PID: 140 Iteration count: 7

Process PID: 142 Parent PID: 140 Iteration count: 6

Process PID: 143 Parent PID: 140 Iteration count: 6

Process PID: 141 Parent PID: 140 Iteration count: 8

Process PID: 141 Parent PID: 140 Iteration count: 9

Process PID: 142 Parent PID: 140 Iteration count: 7

Process PID: 143 Parent PID: 140 Iteration count: 7

Process PID: 142 Parent PID: 140 Iteration count: 8

Process PID: 143 Parent PID: 140 Iteration count: 8

Process PID: 142 Parent PID: 140 Iteration count: 9

Process PID: 143 Parent PID: 140 Iteration count: 9

**Explanation and Code: (c+d)**

After execting an iteration, a process goes to wait state. If all the process goes to wait state, then all the semaphores will have value 0. So on checking the values, all the semaphores are signalled so that the processes can move to next iteration

#define NO\_OF\_PROCESS 3

#define NO\_OF\_ITERATION 10

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

    int parent\_pid = getpid();

    pid\_t \*child;

    sem\_t \*\*sem;

    for (int i = 0; i < NO\_OF\_PROCESS; i++){

        sem[i] = sem\_open(itcnt, O\_CREAT, 0660, 1);

    }

    sem\_t \*mutex = sem\_open("mutex", O\_CREAT, 0660, 1);

    sem\_unlink("count");

    sem\_t \*count = sem\_open("count", O\_CREAT, 0660, NO\_OF\_PROCESS);

    for (int i = 0; i < NO\_OF\_PROCESS; i++){

        child[i] = fork();

        if (child[i] < 0) {

            //error

            printf("Forking failed\n");

            printf("Child no: %d", i);

            return 1;

        } else if (child[i] == 0){

            pid\_t temp = getpid();

            srand(temp);

            for (int j = 0; j < NO\_OF\_ITERATION; j++){

                // sem\_wait(mutex2);

                sem\_wait(sem[i]);

                sem\_wait(count);

                printf("Process PID: %d\tIteration count: %d\n", i, j);

                sleep(sleep(rand()%10));

                sem\_wait(mutex);

                int val;

                int t = sem\_getvalue(count, &val);

                if (!val){

                    for (int k = 0; k < NO\_OF\_PROCESS; k++){

                        sem\_post(sem[k]);

                        sem\_post(count);

                    }

                }

                sem\_post(mutex);

            }

            break;

        } else {

            printf("Child process: %d, pid: %d \n", i, child[i]);

        }

    }

**Output: (c+d)**

Child process: 0, pid: 159

Process PID: 0 Iteration count: 0

Child process: 1, pid: 160

Process PID: 1 Iteration count: 0

Child process: 2, pid: 161

Process PID: 2 Iteration count: 0

Process PID: 2 Iteration count: 1

Process PID: 1 Iteration count: 1

Process PID: 0 Iteration count: 1

Process PID: 1 Iteration count: 2

Process PID: 2 Iteration count: 2

Process PID: 0 Iteration count: 2

Process PID: 2 Iteration count: 3

Process PID: 1 Iteration count: 3

Process PID: 0 Iteration count: 3

Process PID: 0 Iteration count: 4

Process PID: 2 Iteration count: 4

Process PID: 1 Iteration count: 4

Process PID: 1 Iteration count: 5

Process PID: 0 Iteration count: 5

Process PID: 2 Iteration count: 5

Process PID: 0 Iteration count: 6

Process PID: 2 Iteration count: 6

Process PID: 1 Iteration count: 6

Process PID: 2 Iteration count: 7

Process PID: 0 Iteration count: 7

Process PID: 1 Iteration count: 7

Process PID: 0 Iteration count: 8

Process PID: 1 Iteration count: 8

Process PID: 2 Iteration count: 8

Process PID: 1 Iteration count: 9

Process PID: 2 Iteration count: 9

Process PID: 0 Iteration count: 9

**Question 3**

3. Implement the following applications using different IPC mechanisms. Your choice is restricted to Pipe, FIFO:

a. Broadcasting weather information (one broadcasting process and more than one listeners)

b. Telephonic conversation (between a caller and a receiver)

**Explanation:** file handling is done for pipes. The broadcaster will write the message n number of times, if there are n receivers. Each receiver will read the buffer once to get the information. Semaphores used are:

* **notify\_on\_sent**: Signalled by the receiver on receiving. It helps the broadcaster to get the count of number of receivers the broadcaster could sent the data
* **received:** used so that one receiver can read only once, otherwise one receiver may read the buffer more than once, so others will not get the data.

**Code**: Broadcaster

///same includes

int main(){

int fifo\_file;

char \*fifo\_name = "/tmp/myFifo";

mkfifo(fifo\_name, 0666);

char buffer[80] = "";

int r\_cnt;

printf("Enter number of receivers: ");

scanf("%d", &r\_cnt);

sem\_unlink("notify\_on\_sent");

sem\_unlink("get\_value");

sem\_t \*notify\_on\_sent = sem\_open("notify\_on\_sent", O\_CREAT, 0660, 0);

sem\_t \*get\_value = sem\_open("get\_value", O\_CREAT, 0660, 0);

sem\_t \*sent[r\_cnt];

for (int i = 0; i < r\_cnt; i++){

char \* temp = strcat(uitoa(i), "sent");

sem\_unlink(temp);

sent[i] = sem\_open(temp, O\_CREAT, 0660, 0);

}

while(strcmp(buffer, "!end")){

printf("Enter message: \n");

scanf("%s", buffer);

printf("lol\n");

fifo\_file = open(fifo\_name, O\_WRONLY);

for (int i = 0; i < r\_cnt; i++){

write(fifo\_file, buffer, 80);

sem\_post(sent[i]);

}

close(fifo\_file);

// for (int i = 0; i < r\_cnt; i++){

// }

while (1){

sem\_wait(notify\_on\_sent);

int val;

sem\_getvalue(get\_value, &val);

if (val == r\_cnt){

while (val--){

sem\_wait(get\_value);

}

break;

}

}

}

}

**Code**: Receiver

// same includes

int main(){

int fifo\_file;

char \*fifo\_name = "/tmp/myFifo";

mkfifo(fifo\_name, 0666);

char buffer[80] = "";

int r\_cnt;

sem\_t \*notify\_on\_sent = sem\_open("notify\_on\_sent", O\_CREAT, 0660, 0);

sem\_t \*get\_value = sem\_open("get\_value", O\_CREAT, 0660, 0);

char \* temp = strcat(uitoa(r\_cnt), "sent");

sem\_t \*received = sem\_open(temp, O\_CREAT, 0660, 0);

printf("Enter receiver number: ");

scanf("%d", &r\_cnt);

printf("lol\n");

while(strcmp(buffer, "!end")) {

sem\_wait(received);

fifo\_file = open(fifo\_name, O\_RDONLY); // read only mode

read(fifo\_file, buffer, 80);

printf("Message received: %s\n", buffer);

close(fifo\_file);

sem\_post(notify\_on\_sent);

sem\_post(get\_value);

}

}

**Output**

|  |  |  |
| --- | --- | --- |
| **Broadcaster** | **Receiver (1)** | **Receiver (2)** |
| Enter number of receivers: 2  Enter message:  hello  Enter message:  abc  Enter message:  !end | Enter receiver number: 0  Message received: hello  Message received: abc | Enter receiver number: 0  Message received: hello  Message received: abc |

**Question 4**

4. Write a program for *p*-producer *c*-consumer problem, *p, c* >= 1. A shared circular buffer that can hold 25 items is to be used. Each producer process stores any numbers between 1 to 80 (along with the producer id) in the buffer one by one and then exits. Each consumer process reads the numbers from the buffer and adds them to a shared variable TOTAL (initialized to 0). Though any consumer process can read any of the numbers in the buffer, the only constraint being that every number written by some producer should be read exactly once by exactly one of the consumers.

(a) The program reads in the value of *p* and *c* from the user, and forks *p* producers and *c* consumers.

(b) After all the producers and consumers have finished (the consumers exit after all the data produced by all producers have been read), the parent process prints the value of TOTAL.

Test the program with different values of *p* and *c*.

**Explanation**: bounded buffer is syncronised using the mutex. Other semaphores used are full (which is waited upon consuming, hence if empty is 0, consumer will wait and is signaled by producer on producing) and empty, which works in the reverse way)

**Code:**

#define BUFFER\_LENGTH 1000

typedef struct Data {

    int data;

    int pid;

} Data;

typedef struct Bounded\_Buffer {

    Data buffer[BUFFER\_LENGTH];

    int cur\_size;

    int max\_size;

    int front;

    int back;

} Bounded\_Buffer;

void init (Bounded\_Buffer \*b, int size){

    b->cur\_size = 0;

    b->front = 0;

    b->back = 0;

    b->max\_size = size;

}

int push (Bounded\_Buffer \*b, Data data){

    if(b->cur\_size >= b->max\_size)

        return -1;

    b->buffer[b->front % b->max\_size] = data;

    b->cur\_size++;

    b->front++;

    return 0;

}

Data pop(Bounded\_Buffer \*b){

    if(b->cur\_size <= 0) {

        Data Err;

        Err.pid = -1;

        Err.data = -1;

        return Err;

    }

    b->cur\_size--;

    b->back++;

    return b->buffer[(b->back-1) % b->max\_size];

}

int main () {

    int p\_cnt = 100;

    int c\_cnt = 100;

    printf("Enter number of producers: ");

    scanf("%d", &p\_cnt);

    printf("Enter number of consumers: ");

    scanf("%d", &c\_cnt);

    int buffer\_size = 25;

    sem\_unlink("mutex");

    sem\_unlink("empty");

    sem\_unlink("full");

    sem\_unlink("start");

    sem\_t \*mutex = sem\_open("mutex", O\_CREAT, 0660, 1);

    sem\_t \*empty = sem\_open("empty", O\_CREAT, 0660, buffer\_size);

    sem\_t \*full = sem\_open("full", O\_CREAT, 0660, 0);

    sem\_t \*start = sem\_open("start", O\_CREAT, 0660, 0);

    Bounded\_Buffer \*buffer = (Bounded\_Buffer \*)mmap(NULL , sizeof(Bounded\_Buffer), PROT\_READ | PROT\_WRITE , MAP\_SHARED | MAP\_ANONYMOUS , -1, 0);

    init(buffer, buffer\_size);

    pid\_t ps[p\_cnt];

    pid\_t cs[c\_cnt];

    for (int i = 0; i < p\_cnt; i++){

        ps[i] = fork();

        if (ps[i] < 0) {

            perror("Forking failed\n");

            return 1;

        } else if (ps[i] == 0) {

            sem\_wait(start);

            sem\_wait(empty);

            sem\_wait(mutex);

                srand(getpid());

                Data d;

                d.data = rand()%80;

                d.pid = getpid();

                int v = push(buffer, d);

                if (v == -1 ){

                    printf("P\_ID: %d\t failed, buffer full", d.pid);

                } else {

                    printf("P\_ID: %d\t number of items: %d\n", d.pid, d.data);

                }

            sem\_post(mutex);

            sem\_post(full);

            return 0;

        } else {

            printf("Producer No: %d created, pid: %d\n", i, ps[i]);

        }

    }

    for (int j = 0; j < c\_cnt; j++){

        cs[j] = fork();

        if (cs[j] < 0) {

            perror("Forking failed\n");

            return 1;

        } else if (cs[j] == 0) {

            sem\_wait(start);

            sem\_wait(full);

            sem\_wait(mutex);

            Data d = pop(buffer);

            if (d.pid == -1){

                printf("C\_ID: %d couldnt consume, buffer empty\n", getpid());

            } else {

                printf("C\_ID: %d\t consumed data: %d\t from P\_ID: %d\n", getpid(), d.data, d.pid);

            }

            sem\_post(mutex);

            sem\_post(empty);

            return 0;

        } else {

            printf("Consumer No: %d created\n", j);

        }

    }

    int v;

    for (int i = 0; i < p\_cnt+c\_cnt; i++){

        sem\_post(start);

    }

    for (int i = 0; i < p\_cnt; i++){

        waitpid(ps[i], &v, 0);

    }

    for (int i = 0; i < c\_cnt; i++){

        waitpid(cs[i], &v, 0);

    }

    sem\_unlink("full");

    sem\_unlink("mutex");

    sem\_unlink("empty");

    sem\_unlink("start");

    return 0;

}

**Output:**

Enter number of producers: 10

Enter number of consumers: 10

Producer No: 0 created, pid: 375

Producer No: 1 created, pid: 376

Producer No: 2 created, pid: 377

Producer No: 3 created, pid: 378

Producer No: 4 created, pid: 379

Producer No: 5 created, pid: 380

Producer No: 6 created, pid: 381

Producer No: 7 created, pid: 382

Producer No: 8 created, pid: 383

Producer No: 9 created, pid: 384

Consumer No: 0 created

Consumer No: 1 created

Consumer No: 2 created

Consumer No: 3 created

Consumer No: 4 created

Consumer No: 5 created

Consumer No: 6 created

Consumer No: 7 created

Consumer No: 8 created

Consumer No: 9 created

P\_ID: 375 number of items: 4

P\_ID: 377 number of items: 3

P\_ID: 383 number of items: 5

P\_ID: 381 number of items: 11

P\_ID: 379 number of items: 34

C\_ID: 388 consumed data: 4 from P\_ID: 375

C\_ID: 385 consumed data: 3 from P\_ID: 377

P\_ID: 376 number of items: 36

C\_ID: 387 consumed data: 5 from P\_ID: 383

C\_ID: 389 consumed data: 11 from P\_ID: 381

C\_ID: 394 consumed data: 34 from P\_ID: 379

P\_ID: 384 number of items: 40

P\_ID: 380 number of items: 29

P\_ID: 378 number of items: 36

C\_ID: 386 consumed data: 36 from P\_ID: 376

C\_ID: 393 consumed data: 40 from P\_ID: 384

P\_ID: 382 number of items: 4

C\_ID: 391 consumed data: 29 from P\_ID: 380

C\_ID: 392 consumed data: 36 from P\_ID: 378

C\_ID: 390 consumed data: 4 from P\_ID: 382

**Question 5**

Write a program for the Reader-Writer process for the following situations:

a) Multiple readers and one writer: writer gets to write whenever it is ready (reader/s wait)

b) Multiple readers and multiple writers: any writer gets to write whenever it is ready, provided no other swriter is currently writing (reader/s wait)

**Explanation**: original implementation.

For writer priority, a shared variable “write\_count” and a set of semaphores = number of readers are used. Write count is incremented on appearance of a writer. If a reader finds presence of a writer, it goes to blocked state by waiting the semaphore corresponding to it. The writer signals all the semaphores before exiting .

**Code:**

int main(){

    int r\_cnt = 10;

    int w\_cnt = 10;

    int read\_iteration\_count = 20;

    int \*buffer = (int \*)mmap(NULL, sizeof(int), PROT\_READ | PROT\_WRITE , MAP\_SHARED | MAP\_ANONYMOUS , -1, 0);

    int \*reader\_cnt = (int \*)mmap(NULL, sizeof(int), PROT\_READ | PROT\_WRITE , MAP\_SHARED | MAP\_ANONYMOUS , -1, 0);

    int \*writer\_cnt = (int \*)mmap(NULL, sizeof(int), PROT\_READ | PROT\_WRITE , MAP\_SHARED | MAP\_ANONYMOUS , -1, 0);

    \*buffer =  -1; //init

    \*reader\_cnt = 0;

    \*writer\_cnt = 0;

    pid\_t readers[r\_cnt];

    pid\_t writers[w\_cnt];

    int t;

    //semaphores

    sem\_unlink("wrt");

    sem\_unlink("mutex");

    sem\_unlink("w\_priority");

    sem\_unlink("start");

    sem\_t \*wrt = sem\_open("wrt", O\_CREAT, 0660, 1);

    sem\_t \*mutex = sem\_open("mutex", O\_CREAT, 0660, 1);

    sem\_t \*w\_priority = sem\_open("w\_priority", O\_CREAT, 0660, 1);

    sem\_t \*start = sem\_open("start", O\_CREAT, 0660, 0);

    sem\_t \*is\_writer\_done[r\_cnt];

    for (int i = 0; i < r\_cnt; i++){

        char \* temp = uitoa(i);

        char \* temp2 = strcat(temp, "is\_writer\_done");

        sem\_unlink(temp2);

        is\_writer\_done[i] = sem\_open(temp2, O\_CREAT, 0660, 0);

    }

    //reader processes

    for (int i = 0; i < r\_cnt; i++){

        readers[i] = fork();

        if (readers[i] < 0){

            perror("Forking failed\n");

            return 1;

        } else if (readers[i] == 0) {

            srand(getpid());

            sleep(sleep(rand()%10));

                sem\_wait(start);

                //read and print this many times!!

                while (\*writer\_cnt > 0)

                    sem\_wait(is\_writer\_done[i]);

                    //while since is\_writer\_done maybe > 0

                sem\_wait(mutex);

                  \*reader\_cnt = \*reader\_cnt + 1;

                  if(\*reader\_cnt == 1) sem\_wait(wrt);

                sem\_post(mutex);

                printf("Reader cnt: %d\tValue: %d \n", i, \*buffer);

                sem\_wait(mutex);

                  \*reader\_cnt = \*reader\_cnt - 1;

                  if(\*reader\_cnt == 0) sem\_post(wrt);

                sem\_post(mutex);

            return 0;

        } else {

            printf("Reader No: %d created\n", i);

        }

    }

    //writer processes

    for (int i = 0; i < w\_cnt; i++){

        writers[i] = fork();

        if (writers[i] < 0){

            perror("Forking failed\n");

            return 1;

        } else if (writers[i] == 0) {

            srand(getpid());

            sleep(sleep(rand()%10));

            sem\_wait(start);

            sem\_wait(w\_priority);

            \*writer\_cnt = \*writer\_cnt + 1;

            sem\_post(w\_priority);

            sem\_wait(wrt);

            \*buffer = i;

            printf("Writer #%d wrote data = %d\n", i, i);

            sem\_post(wrt);

            sem\_wait(w\_priority);

            \*writer\_cnt = \*writer\_cnt - 1;

            sem\_post(w\_priority);

            for(int j = 0; j < r\_cnt; j++) sem\_post(is\_writer\_done[j]);

            return 0;

        } else {

            printf("Writer No: %d created\n", i);

        }

    }

    int v;

    for (int i = 0; i < r\_cnt+w\_cnt; i++){

        sem\_post(start);

    }

    for (int i = 0; i < r\_cnt; i++){

        waitpid(readers[i], &v, 0);

    }

    for (int i = 0; i < w\_cnt; i++){

        waitpid(writers[i], &v, 0);

    }

    sem\_unlink("wrt");

    sem\_unlink("mutex");

    sem\_unlink("start");

    sem\_unlink("w\_priority");

    for (int i = 0; i < r\_cnt; i++){

        char \* temp = uitoa(i);

        char \* temp2 = strcat(temp, "is\_writer\_done");

        sem\_unlink(temp2);

    }

    return 0;

}

**Output:**

Reader No: 0 created

Reader No: 1 created

Reader No: 2 created

Reader No: 3 created

Reader No: 4 created

Reader No: 5 created

Reader No: 6 created

Reader No: 7 created

Reader No: 8 created

Reader No: 9 created

Writer No: 0 created

Writer No: 1 created

Writer No: 2 created

Writer No: 3 created

Writer No: 4 created

Writer No: 5 created

Writer No: 6 created

Writer No: 7 created

Writer No: 8 created

Writer No: 9 created

Writer #3 wrote data = 3

Writer #1 wrote data = 1

Writer #4 wrote data = 4

Writer #9 wrote data = 9

Reader cnt: 8 Value: 9

Writer #2 wrote data = 2

Reader cnt: 3 Value: 2

Reader cnt: 1 Value: 2

Reader cnt: 2 Value: 2

Reader cnt: 6 Value: 2

Writer #0 wrote data = 0

Writer #6 wrote data = 6

Writer #7 wrote data = 7

Reader cnt: 5 Value: 7

Reader cnt: 7 Value: 7

Reader cnt: 4 Value: 7

Reader cnt: 9 Value: 7

Writer #5 wrote data = 5

Reader cnt: 0 Value: 5

Writer #8 wrote data = 8

**Question 6:**  
Implement Dining Philosophers’ problem using Monitor. Test the program with (a) 5 philosophers and 5 chopsticks, (b) 6 philosophers and 6 chopsticks, and (c) 7 philosophers and 7 chopsticks

**Explanation:** Using Monitor, Dining philosopher’s problem is implemented. Condition is implemented using semaphore and count

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <time.h>

#include <semaphore.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <sys/wait.h>

#define NOP 7

#define THINKING 0

#define HUNGRY 1

#define EATING 2

char \* uitoa(int num){

char \* str;

str = (char \*)malloc(5 \* sizeof(char));

str[4] = 0;

for (int i = 3; i >= 0; i--){

str[i] = (num%10)+'0';

num/=10;

}

return str;

}

typedef struct condition {

sem\_t \*t;

int count;

} Condition;

typedef struct DP {

sem\_t \* mutex;

sem\_t \* next;

int next\_count;

int state[NOP];

Condition x[NOP];

// int turn[NOP];

} DP;

void cond\_wait(DP \* dp, int i){

dp->x[i].count++;

if (dp->next\_count > 0){

sem\_post(dp->next);

} else {

sem\_post(dp->mutex);

}

sem\_wait(dp->x[i].t);

dp->x[i].count--;

}

void cond\_signal (DP \* dp, int i){

if (dp->x[i].count > 0){

dp->next\_count++;

sem\_post(dp->x[i].t);

sem\_wait(dp->next);

dp->next\_count--;

}

}

void test (DP \* dp, int i){

if (dp->state[(i+NOP-1)%NOP] != EATING && dp->state[(i+1)% NOP] != EATING && dp->state[i] == HUNGRY){ // && dp->turn[i] == i && dp->turn[(i+NOP-1)%NOP] == i){

dp->state[i] = EATING;

cond\_signal(dp, i);

}

}

void pickup(DP \* dp, int i){

sem\_wait(dp->mutex);

dp->state[i] = HUNGRY;

test(dp, i);

if (dp->state[i] == HUNGRY){

cond\_wait(dp, i);

}

printf("%d got the chopsticks\n", i);

if (dp->next\_count > 0){

sem\_post(dp->next);

} else {

sem\_post(dp->mutex);

}

}

void putdown(DP \* dp, int i){

sem\_wait(dp->mutex);

dp->state[i] = THINKING;

// dp->turn[i] = (i+1)%NOP;

// dp->turn[(i+NOP-1)%NOP] = (i+NOP-1)%NOP;

printf("%d put the chopsticks down\n", i);

test(dp, (i+NOP-1)%NOP);

test(dp, (i+1)%NOP);

if (dp->next\_count > 0){

sem\_post(dp->next);

} else {

sem\_post(dp->mutex);

}

}

void init (DP \* dp){

/\*\*

\* sem\_t \* mutex;

\* sem\_t \* next;

\* int next\_count;

\* int state[NOP];

\* Condition x[NOP];

\*/

sem\_unlink("mutex");

dp->mutex = sem\_open("mutex", O\_CREAT, 0660, 1);

sem\_unlink("next");

dp->next = sem\_open("next", O\_CREAT, 0660, 0);

dp->next\_count = 0;

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

dp->state[i] = THINKING;

//condition

dp->x[i].count = 0;

char \* temp = uitoa(i);

char \* temp2 = strcat(temp, "cond");

sem\_unlink(temp2);

dp->x[i].t = sem\_open(temp2, O\_CREAT, 0660, 0);

//turn

// dp->turn[i] = ((i/2)\*2 + 2)%NOP;

}

}

int main(){

DP \* dp = (DP \*)mmap(NULL, sizeof(DP), PROT\_READ | PROT\_WRITE , MAP\_SHARED | MAP\_ANONYMOUS , -1, 0);

init (dp);

pid\_t child [NOP];

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

child[i] = fork();

if (child[i] < 0){

perror("Forking failed\n");

return 1;

} else if (child[i] == 0){

for (int j = 0; j < 100; j++){

srand(getpid());

sleep(rand()%10);

pickup(dp, i);

sleep(rand()%5);

putdown(dp, i);

}

return 0;

} else {

1;

}

}

int v;

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

waitpid(child[i], &v, 0);

}

}

**Output:**

2 got the chopsticks

2 put the chopsticks down

4 got the chopsticks

2 got the chopsticks

4 put the chopsticks down

2 put the chopsticks down

2 got the chopsticks

2 put the chopsticks down

4 got the chopsticks

4 put the chopsticks down

2 got the chopsticks

2 put the chopsticks down

2 got the chopsticks

2 put the chopsticks down

1 got the chopsticks

4 got the chopsticks

4 put the chopsticks down

3 got the chopsticks

5 got the chopsticks

1 put the chopsticks down

0 got the chopsticks

5 put the chopsticks down

3 put the chopsticks down

2 got the chopsticks

4 got the chopsticks

0 put the chopsticks down

6 got the chopsticks

2 put the chopsticks down

4 put the chopsticks down

6 put the chopsticks down

2 got the chopsticks

2 put the chopsticks down

4 got the chopsticks

4 put the chopsticks down

2 got the chopsticks

2 put the chopsticks down

2 got the chopsticks

2 put the chopsticks down

4 got the chopsticks

4 put the chopsticks down

2 got the chopsticks

2 put the chopsticks down

1 got the chopsticks

4 got the chopsticks

4 put the chopsticks down

5 got the chopsticks

3 got the chopsticks

1 put the chopsticks down

5 put the chopsticks down

0 got the chopsticks

0 put the chopsticks down

3 put the chopsticks down

2 got the chopsticks

4 got the chopsticks

2 put the chopsticks down

6 got the chopsticks

4 put the chopsticks down

6 put the chopsticks down

2 got the chopsticks

2 put the chopsticks down

**Question 7:**  
Write a program that will find out whether a system is in safe state or not with following

specifications:

Command line input: name of a file - The file contains the initial state of the system as given

below:

#no of resources 4

#no of instances of each resource 2 4 5 3

#no of processes 3

#no of instances of each resource that each process needs in its lifetime 1 1 1 1, 2 3 1 2, 2 2 1 3

The program waits to accept a resource allocation request to be supplied by the user or read from another file:

For example: 0 1 0 1 1 indicates that p0 has requested allocation of 1 instance of R0, R2 and R3 each.

Your program should declare the result:

(1) should this request be granted?

(2) if your answer is yes, print the safe sequence in which all remaining needs can be granted one by one and also grant the request. If the requesting process's need is NIL, the program internally releases all its resources. Go back to accept another request till all processes finish with all their needs.

Testing:

a. Generate possible request sequences of each process.

B. Each such sequence must satisfy the maximum requirements of the process.

**Explanation**: Deadlock avoidance algorithm is implemented. For finding all possible sequence of requests, permutations are created using backtracking algorithm. For checking if a process is completed after fulfilling a request, a frequency table is used.

#include<stdio.h>

#include<stdlib.h>

int no\_of\_types\_resource;

int no\_of\_processes;

int no\_of\_requests;

int \* max\_resource\_count; // total resource count

int \*\* process\_requirement; // process requirements

int \*\* orig; // used to keep data, to release resources.

int \*\* process\_requests; // request list

int \* permutate; // for permutation generation

int \* freq; // frequency table

int cnt = 0;

void swap(int a, int b){

int temp = permutate[a];

permutate[a] = permutate[b];

permutate[b] = temp;

}

void allocate (int loc){

int pid = process\_requests[loc][0];

freq[pid] --;

for (int i = 1; i<=no\_of\_types\_resource; i++){

process\_requirement[pid][i-1] -= process\_requests[loc][i];

max\_resource\_count[i-1] -= process\_requests[loc][i];

}

if (freq[pid] == 0){

for (int i = 0; i < no\_of\_types\_resource; i++){

max\_resource\_count[i] += orig[pid][i];

}

}

}

void deallocate (int loc){

int pid = process\_requests[loc][0];

freq[pid] ++;

for (int i = 1; i<=no\_of\_types\_resource; i++){

process\_requirement[pid][i-1] += process\_requests[loc][i];

max\_resource\_count[i-1] += process\_requests[loc][i];

}

if (freq[pid] == 1){

for (int i = 0; i < no\_of\_types\_resource; i++){

max\_resource\_count[i] -= orig[pid][i];

}

}

}

void solve(int index){

if (index == no\_of\_requests){

// printf("lol");

// getchar();

for (int i = 0; i < no\_of\_requests; i++){

printf("%d ", permutate[i]+1);

} printf("\n");

cnt++;

return;

}

/\*\*

\* when to return??

\* 1. at least one contains more!!

\* 2. next to be allocated contains more!!

\*/

//1

// not required??

int j = 0;

for (; j < no\_of\_processes; j++){

int k = 0;

for (; k < no\_of\_types\_resource; k++){

if (process\_requirement[j][k] > max\_resource\_count[k]) break; //break when fails

}

if (k == no\_of\_types\_resource) break;

}

if (j == no\_of\_processes) return;

for (int i = index; i < no\_of\_requests; i++){ // idx ..... i

swap(index, i);

int k;

for (k = 0; k < no\_of\_types\_resource; k++){

if (process\_requests[permutate[index]][k+1] > max\_resource\_count[k] )break;

}

if (k == no\_of\_types\_resource) {

int t = permutate[index];

allocate(t);

solve (index+1);

deallocate(t);

}

swap(index, i);

}

}

int main(){

FILE \* fpreq = fopen("prerequisite.txt", "r");

FILE \* falloc = fopen("allocation.txt", "r");

fscanf(fpreq, "%d", &no\_of\_types\_resource);

max\_resource\_count = (int \*)malloc (sizeof(int) \* no\_of\_types\_resource);

for (int i = 0; i < no\_of\_types\_resource; i++){

int temp;

fscanf(fpreq, "%d", max\_resource\_count+i);

}

fscanf(fpreq, "%d", &no\_of\_processes);

freq = (int \*) malloc (sizeof(int) \* no\_of\_processes);

process\_requirement = (int \*\*)malloc(sizeof(int \*) \* no\_of\_processes);

orig = (int \*\*)malloc(sizeof(int \*) \* no\_of\_processes);

for (int i = 0; i < no\_of\_processes; i++){

freq[i] = 0;

process\_requirement[i] = (int \*)malloc(sizeof(int) \* no\_of\_types\_resource);

orig[i] = (int \*)malloc(sizeof(int) \* no\_of\_types\_resource);

for (int j = 0; j < no\_of\_types\_resource; j++){

int temp;

fscanf(fpreq, "%d", &temp);

process\_requirement[i][j] = temp;

orig[i][j] = temp;

}

}

fscanf(falloc, "%d", &no\_of\_requests);

process\_requests = (int \*\*)malloc(sizeof(int\*)\*no\_of\_requests);

for (int i = 0; i < no\_of\_requests; i++){

process\_requests[i] = (int \*)malloc(sizeof(int)\*(no\_of\_types\_resource+1));

for (int j = 0; j < (no\_of\_types\_resource+1); j++){

int temp;

fscanf(falloc, "%d", &temp);

process\_requests[i][j] = temp;

if (j == 0){

freq[temp] ++;

}

}

}

permutate = (int \*)malloc(sizeof(int)\*no\_of\_requests);

for (int i = 0; i < no\_of\_requests; i++){

permutate[i] = i;

}

fclose(fpreq);

fclose(falloc);

printf("Valid output sequences are listed below: \n");

printf("-----------------------------------------\n");

solve (0);

printf("Total number of solutions: %d\n\n", cnt);

return 0;

}

**Input and Output:**

|  |  |  |
| --- | --- | --- |
| prerequisite.txt (processor and resource values) | Processor demands | Output |
| 4  2 4 5 3  3  1 1 1 1  2 3 1 2  2 2 1 3 | 8  0 1 0 1 1  1 0 3 0 0  2 1 1 1 1  0 0 1 0 0  1 1 0 1 1  2 1 1 0 0  1 1 0 0 1  2 0 0 0 2 | Valid output sequences are listed below:  -----------------------------------------  1 2 4 5 7 6 3 8  1 2 4 5 7 6 8 3  1 2 4 5 7 3 6 8  ……  ……  8 1 6 4 3 7 2 5  8 1 6 4 3 2 7 5  8 1 6 4 3 2 5 7  Total number of solutions: 2412 |