OPERATING SYSTEM PROJECT

A black sign with white text

Description automatically generated

**Course Name**: Operating System

**Course Code**: Cse316

**Submitted To**: Amandeep Kaur

**Submitted By**: Kuwar Singh

**Reg No.**: 11803893

**Email**:kuwarsingh123a@gmail.com

**Question**:

35. Write a four level priority process dispatcher operating within the constraints of finite available memory and I/O resources. Use different dispatchers and memory allocation.

The dispatcher controls the following resources:

• 2 Printers

• 1 Scanner

• I Modem

• 2 CD Drives

• 1024 Mbyte Memory available for processes

The dispatcher is presented with a list of processes along with their arrival times, priority, and requested resources. The dispatcher ensures that each requested resource is solely available to that process throughout its lifetime in the 'ready-to-run' dispatch queues: from the initial transfer from the job queue through to process completion, including any intervening idle time quanta. —\*The executing process is emulated by a supplied program that reports any signals sent to it and `ticks' once a second while it is running.

**CODE**

Code for hostd.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <fcntl.h>

#include <signal.h>

#include "pcb.h"

#include "mab.h"

#include "rsrc.h"

#include "utility.h"

// Setting the avaiable resources

int available\_printers = 2;

int available\_scanners = 1;

int available\_modems = 1;

int available\_CDs = 2;

void HOSTDispatcher(PcbPtr queue) {

// Initialize dispatcher queues, real time queue and memory

MabPtr memory = createnullMab();

PcbPtr inputQueue = queue;

PcbPtr running = NULL;

int timer = 0; // Set dispatcher timer to 0

PcbPtr priorityQueue1 = NULL;

PcbPtr priorityQueue2 = NULL;

PcbPtr priorityQueue3 = NULL;

PcbPtr userQueue = NULL;

PcbPtr realTimeQueue = NULL;

int ready; // used to determine if we're ready to start/restart processes, i.e. queue management has been completed

int r = 0;

// While there's anything in any of the queues or there is a currently running process:

while(inputQueue != NULL || realTimeQueue != NULL || userQueue != NULL || priorityQueue1 != NULL || priorityQueue2 != NULL || priorityQueue3 != NULL || running != NULL) {

// Unload pending processes from the input queue:

while(inputQueue != NULL && inputQueue->arrivaltime <= timer) {

// Check priority of process - not real time so put into user queue

if (inputQueue->priority != 0){

if(userQueue != NULL)

userQueue = enqPcb(userQueue,deqPcb(&inputQueue));

else

userQueue = deqPcb(&inputQueue);

}

// Process is real time so put into real time queue

else{

if(realTimeQueue != NULL)

realTimeQueue = enqPcb(realTimeQueue,deqPcb(&inputQueue));

else

realTimeQueue = deqPcb(&inputQueue);

}

}

// user queue processes wait for memory and resources to be allocated then put into priority queue

while(userQueue != NULL && memChk(memory, userQueue->memoryalloc) != NULL && rsrcChk(userQueue) == 1) {

// if priority is 1 then put into priorityQueue 1

if (userQueue->priority == 1){

if(priorityQueue1 != NULL) {

PcbPtr process = deqPcb(&userQueue);

priorityQueue1 = enqPcb(priorityQueue1,process);

memAlloc(memory,process->memoryalloc,process->id);

rsrcAlloc(process);

}

else {

PcbPtr process = deqPcb(&userQueue);

priorityQueue1 = process;

memAlloc(memory,process->memoryalloc,process->id);

rsrcAlloc(process);

}

}

// if priority is 2 then put into priorityQueue 2

else if (userQueue->priority == 2){

if(priorityQueue2 != NULL) {

PcbPtr process = deqPcb(&userQueue);

priorityQueue2 = enqPcb(priorityQueue2,process);

memAlloc(memory,process->memoryalloc,process->id);

rsrcAlloc(process);

}

else {

PcbPtr process = deqPcb(&userQueue);

priorityQueue2 = process;

memAlloc(memory,process->memoryalloc,process->id);

rsrcAlloc(process);

}

}

// if priority is 3 then put into priorityQueue 3

else if (userQueue->priority == 3){

if(priorityQueue3 != NULL) {

PcbPtr process = deqPcb(&userQueue);

priorityQueue3 = enqPcb(priorityQueue3,process);

memAlloc(memory,process->memoryalloc,process->id);

rsrcAlloc(process);

}

else {

PcbPtr process = deqPcb(&userQueue);

priorityQueue3 = process;

memAlloc(memory,process->memoryalloc,process->id);

rsrcAlloc(process);

}

}

}

// If a process is currently running:

if(running != NULL && r == 0) {

// Decrement process remainingcputime;

running->remainingcputime--;

// If times up:

if(running->remainingcputime == 0) { //stop process if it has finished running

// Send SIGINT to the process to terminate it;

terminatePcb(running);

// Free up process structure memory;

memFree(memory, running->id);

rsrcFree(running);

running = NULL;

ready = 1;

}

else {

// Send SIGTSTP to suspend it;

suspendPcb(running);

// Reduce the priority of the process (if possible)

if (running->priority < 3)

running->priority++;

//enqueue it on the appropriate feedback queue

if(running->priority == 2) {

if(priorityQueue2 != NULL)

priorityQueue2 = enqPcb(priorityQueue2, running);

else

priorityQueue2 = running;

}

if(running->priority == 3) {

if(priorityQueue3 != NULL)

priorityQueue3 = enqPcb(priorityQueue3, running);

else

priorityQueue3 = running;

}

running = NULL;

ready = 1;

}

}

else if (r == 1){

running->remainingcputime--;

if(running->remainingcputime == 0) { //stop process if it has finished running

terminatePcb(running);

running = NULL;

ready = 1;

}

}

else {

ready = 1;

}

if(ready == 1) {

// If there is a real time process waiting then run it

if (realTimeQueue != NULL){

running = deqPcb(&realTimeQueue);

startPcb(running);

r = 1;

}

// If no process currently running && feedback queues are not all empty:

else if(priorityQueue1 != NULL || priorityQueue2 != NULL || priorityQueue3 != NULL){

// Dequeue a process from the highest priority feedback queue that is not empty and Set it as currently running process;

if(priorityQueue1 != NULL)

running = deqPcb(&priorityQueue1);

else if(priorityQueue2 != NULL)

running = deqPcb(&priorityQueue2);

else if(priorityQueue3 != NULL)

running = deqPcb(&priorityQueue3);

// If already started but suspended, restart it (send SIGCONT to it) else start it (fork & exec)

if(running->suspended == 1){

r = 0;

restartPcb(running);

}

else {

r = 0;

startPcb(running);

}

}

ready = 0;

}

// sleep for one second;

sleep(1);

// Increment dispatcher timer;

timer++;

}

}

int main(int argc, char \*\* argv)

{

PcbPtr newPcb = (PcbPtr) malloc(sizeof(Pcb));

newPcb = readInput(argv[1]);

HOSTDispatcher(newPcb);

free(newPcb);

return 0;

}

Algorithm

The Hypothetical Operating System Testbed (HOST) is a multiprogramming system with a four level priority process dispatcher operating within the constraints of finite available resources.

Structures used by the dispatcher

● Process control block (PCB) The process control block or PCB is the process structure used by the dispatcher. The structure contains variables such as id, arrival time, remaining CPU time, priority, memory and resource requirements and it’s current status. The PCB structure is one of a doubly linked list which allows queuing of processes which is an essential part of the dispatcher.

● Memory structure (MAB) The memory structure or MAB is used to allocate memory for processes in the dispatcher. The structure contains variables such as offset, size, allocated, process. MAB is also a doubly linked list when the initial block, size of 960, is split up into smaller blocks.

● Resource manager (RSRC) The resource manager checks, allocates and frees resources that processes require to run. In this system, we have 2 printers, 1 scanner, 1 modem and 2 CDs which are simply global variables initialized on execution of the program.

**Program structure and individual modules**

For simplicity and modularity, the dispatcher program was split up into different parts; process structure, memory structure, resource management and a utility helper. These modules are all implemented as header files. The utility helper only consists of a function that reads each line of an input file and put the job parameters into process structures (PCB) and create a list of processes. The process structure is implemented as a doubly linked list which allows the dispatcher to queue jobs which is an essential part of this assignment. This structure has the following functions:

● create new process

● start process

● terminate process

● enqueue process

● dequeue process

● suspend process

● restart process

During the reading of the inputs, the dispatcher would create new processes and store the parameters inputted and queue up the processes with the enqueue function. When the process is ready, it would then call the dequeue function and the start process function which fork and execs the process. If the processes needs to be suspended since it’s in round robin, then it would call the suspend and restart functions accordingly. The memory allocation structure is also a doubly linked list. It has the following functions:

● create new memory block ● check memory for allocation ● allocate memory block ● free memory block ● split memory block ● merge memory blocks Memory allocation of this dispatcher uses the first fit algorithm as mentioned previously.

On executions, the dispatcher will call the create new memory block function. After the processes are taken from the input queue, it then needs to be allocated memory, so it will call the check memory function which would search through the list to find the first block that can cater for the process requirements. It would split the memory block and form a linked list if needed. When the process is completed, the free memory function is called and it would merge any adjacent blocks that are unallocated. The resource manager manages the global variables which represent the resources of the system. After memory is allocated, the resource check function is called to check if there are enough resources available for a process, if there is then it would continue to call the resource allocation function which decrements the global variables accordingly to it’s needs. When the process is completed, the free resource allocation is called which increment the respective resource variables.

**Shortcomings of this dispatcher:**

● There is possible starvation for user priority processes which may find it hard to get memory and resources or even get at turn to be executed since the real time processes will always have higher priority. An improvement for this would be to set some condition that would raise the priority of the process if it’s been in waiting for too long.

● Since the first fit algorithm was implemented there may be some fragmentation at the start of the memory structure, and therefore the next fit algorithm would have been more effective

**Dispatching scheme (including memory and resource allocation)**

The Hypothetical Operating System Testbed (HOST) is a multiprogramming system with a four level priority process dispatcher operating within the constraints of finite available resources.

1.On execution of the dispatcher, the read input function is called to obtain the list of jobs from the input file, create process structures (PCB) for each process and queue them up. That queue is then sent to the dispatching function which on execution initialises the user queue, real time queue, the three priority queues and a memory block of 1024MB but since 64MB is reserved for real time processes, the size is only 960MB.

2. After the input queue is ready, the dispatcher initialises the dispatcher timer which increments every second, if the process in the input arrives then it’s priority will be checked. If the priority is 0 then it is a real time process and is sent to the real time queue which is essentially a first come first serve queue and is given priority over the other queues.

3.On the other hand, if the priority is 1, 2 or 3 then it will go into the user queue to await memory and resource allocation. This is where the first fit memory algorithm comes into play. The process will stay in the queue until it has been allocated both the required memory and resources. Once this happens, it is then put into the appropriate priority queue where the first two queues are in round robin and the third is a feedback queue. When the queue management part is completed then the dispatcher will start to start processes that are in either the real time queue or the priority queue. The real time queue will take priority and the real time process will run until it is completed (non­preemptive) and then it is terminated. If there are no real time processes queued up then the dispatcher will dispatch the first job it finds in the priority queues starting at the first one. The processes that are dispatched from the priority queues only run for 1 second and then it is suspended and then put into the next lower priority queue, awaiting for it’s turn to be restarted. If it already in the third priority queue, then it just gets fed back into the queue and put at the end. The timer is then incremented and the queue management and dispatching of processes will continue until there no more jobs left in any of the queues.

CONSTRAINTS

The HOST has the following resources:

• 2 Printers

• 1 Scanner

• 1 Modem

• 2 CD drives

• 1024 Mbyte memory available for processes

Low-priority processes can use any or all of these resources, but the HOST dispatcher is notified of which resources the process will use when the process is submitted. The dispatcher ensures that each requested resource is solely available to that process throughout its lifetime in the "ready-torun" dispatch queues: from the initial transfer from the job queue to the Priority 1-3 queues through to process completion, including intervening idle time quanta. Real-Time processes will not need any I/O resources (Printer, Scanner, Modem, CD), but will obviously require memory allocation - this memory requirement will always be 64 Mbytes or less for Real-Time jobs.

**Processes**

Processes on HOST are simulated by the hostd creating a new process for each dispatched process. This process is a generic process (supplied as process — source: sigtrap.c) that can be used for any priority process. It actually runs itself at very low priority, sleeping for one-second periods and displaying the following:

1. A message displaying the process ID when the process starts;

2. A regular message every second the process is executed; and 3

. A message when the process is Suspended, Continued, or Terminated.

The process will terminate of its own accord after 20 seconds if it is not terminated by your dispatcher. The process prints out using a randomly generated color scheme for each unique process, so that individual "slices" of processes can be easily distinguishable. Use this process rather than your own. The life cycle of a process is as follows: –5–

1. The process is submitted to the dispatcher input queues via an initial process list that designates the arrival time, priority, processor time required (in seconds), memory block size, and other resources requested.

2. A process is "ready-to-run" when it has "arrived" and all required resources are available.

3. Any pending Real-Time jobs are submitted for execution on a firstcome-first-served basis.

4. If enough resources and memory are available for a lower priority User process, the process is transferred to the appropriate priority queue within the feedback dispatcher unit, and the remaining resource indicators (memory list and i/o devices) updated. 5. When a job is started (fork and exec("process",...)), the dispatcher will display the job parameters (Process ID, priority, processor time remaining (in seconds), memory location and block size, and resources requested) before performing the exec.

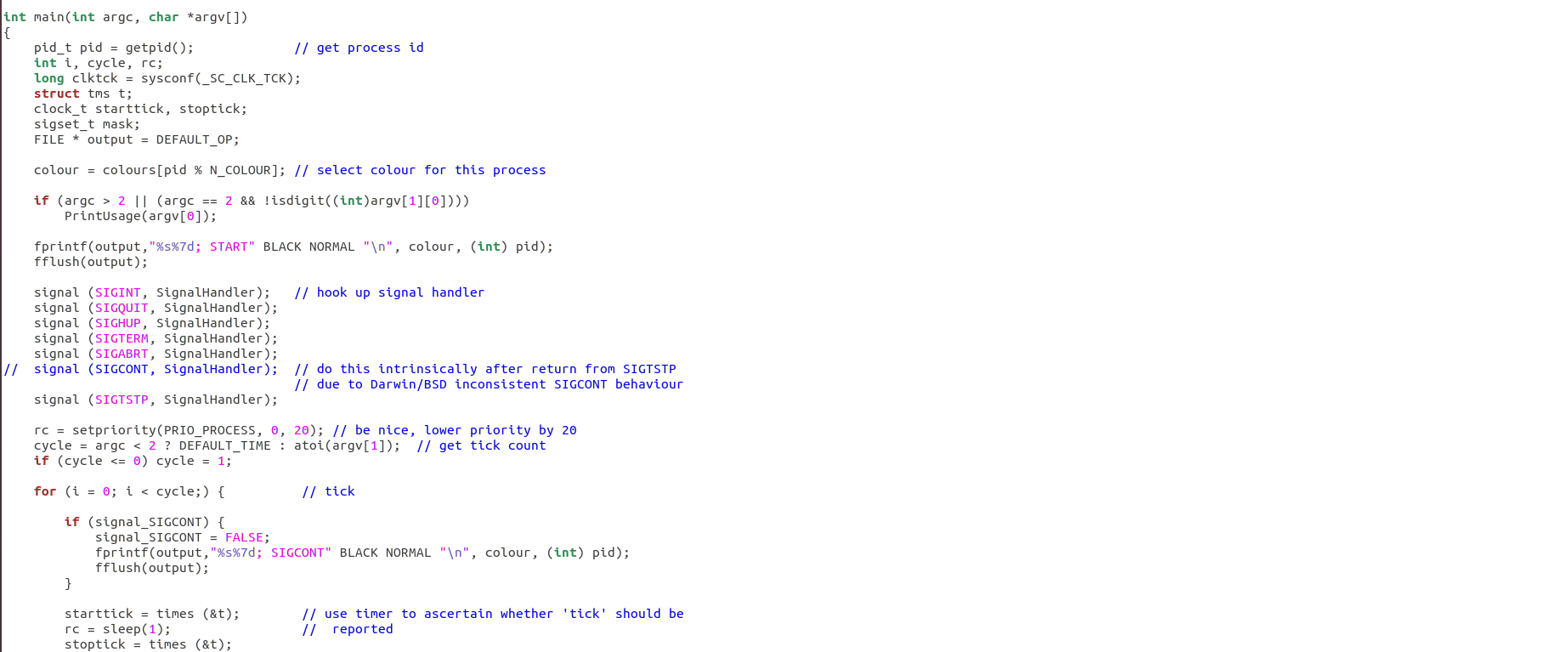
6. A Real-Time process is allowed to run until its time has expired when the dispatcher kills it by sending a SIGINT signal to it.

7. A low priority User job is allowed to run for one dispatcher tick (one second) before it is suspended (SIGTSTP) or terminated (SIGINT) if its time has expired. If suspended, its priority level is lowered (if possible) and it is requeued on the appropriate priority queue . To retain synchronization of output between your dispatcher and the child process, your dispatcher should wait for the process to respond to a SIGTSTP or SIGINT signal before continuing ( waitpid(p->pid, &status, WUNTRACED)). To match the performance sequence indicated in the comparison of scheduling policies , the User job should not be –6– suspended and moved to a lower priority level unless another process is waiting to be (re)started.

8. Provided no higher-priority Real-Time jobs are pending in the submission queue, the highest priority pending process in the feedback queues is started or restarted (SIGCONT).

9. When a process is terminated, the resources it used are returned to the dispatcher for reallocation to further processes. 10. When there are no more processes in the dispatch list, the input queues and the feedback queues, the dispatcher exits.

**Snapshot for sigtrap.c**

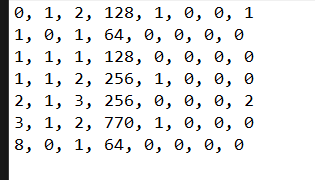


**For more code refer to sigtrap.c file as uploaded**



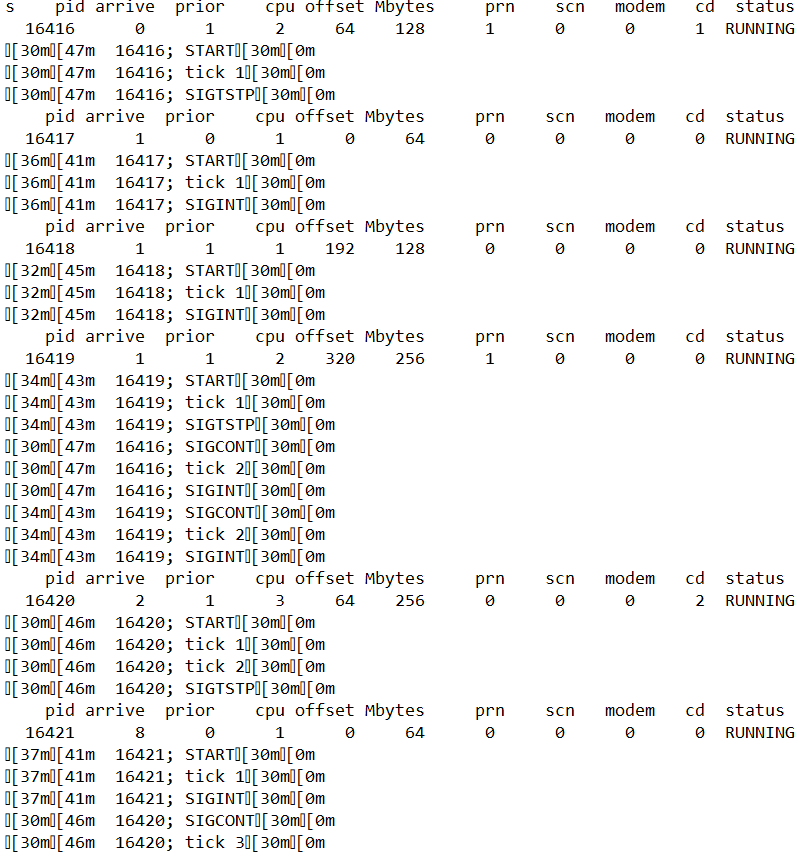
INPUT

Suppose our input is like this



**OUTPUT**

Then the output will be



In the similar way you can try fir other inputs and outputs by running the code files named hostd.c and sigtrap.c and both of them can be combined in a single file named as makefile.

**Have you made minimum 5 revisions of solution on GitHub?**

Yes, I did

GitHub Link: <https://github.com/Kuwar123/priority-dispatcher>