You will write code to simulate a running operation system’s scheduling of processes.  Your code will be one process/thread only and the ‘processes’ are just data structures used in your simulation.

**Make sure you read this and understand completely what you are doing before writing your design and code!**

Rules for code:

A.     Must be C language supported by Ubuntu without installing any new compilers or libraries, as in standard C.

B.     You must write your own priority queue code not use built in data structures so we can see how you chose to implement the algorithm.

C.      You must credit where you got the algorithms and code you use any part of.

D.  Use good modules (functions), your main loop should call functions and fit on one screen.

Do not try to do everything in one function but break it up into logical modules.

Write the hardest part first so you know what it needs.

Write the lowest level, test then write what calls it, test.  Do not try to write whole thing at once unless you really enjoy wasting your own time.

Here are the rules for all programs:

**Rules:**

**If your code does not follow these rules you will have to rewrite it before I accept.**

        Test for an error return from ALL system calls, that includes **all**semaphore calls, fork, etc.

o   If error call perror() unless timer is up for timed wait. (not really an error)

        NO break or continue statements in any loop unless part of a switch() statement and do not change the loop behavior.

        No global variables, you declare, outside of a function.

        All loops must stop only when false, the Boolean expression in the ( ) after while, or for.

        All code must be properly formatted with indentation for all blocks { }

* Functions that are not void have one and only one return statement the last line in the function.
* Must have your name, class name, and assignment name in comments at top of file.
* Have comments that explain how things work if not completely obvious on first glance.  x—; // No decrement x comment  which is obvious

This simulation will assume one processor with one core.  It will assume all user processes with kernel taking no time to do its work.

Time will be the **main loop one clock tick per iteration**.  One time through the loop one clock tick.

Priorities should be 0 to 15 with 0 lowest and 15 highest.

You must use the priority queue algorithm adjusting priorities each time you add a process to the ready queue.

You will also use the Round Robin algorithm so the process in the CPU is forced to leave when quantum time is up.

You will have 3 \* 16 processes, 3 for each priority from 0 to 15.  One that is CPU bound, one that is even CPU and I/O and 1 that is I/O bound, one between CPU and even and one between even and I/O bound. CPU bound must want more time than normally allowed in CPU before fast I/O spending most time in CPU.  I/O bound should stay in CPU much less time than normal CPU time and spend most time waiting for I/O.  Even should spend about the same amount of time in CPU as doing I/O.

**I will provide:**

        the struct definition for a process

        the struct definition for all process

o   time quantum 70

o   time to increase priority 30

        values for all processes

        an input file with starting values

o   space separated unsinged

o   priority cpu io runtime

        code to print the results of the stimulation

**There should be several data structures.**

A priority queue of processes ready for the CPU.  Must be an Array constantly sorted by priority.

Each time click (one time around the main loop) you adjust priorities and move processes in this queue.

A list of processes waiting for I/O

A struct of OS parameters common to all processes

Max time in CPU before being bumped to ready queue if no I/O, the time quantum.

Max wait time user processes in ready queue before increasing priority

**Each process should keep track of at least the following information.**

-Process id (the index to the array.)

-Starting priority, reset current priority to this each time process moved into wait queue, does not change.

-Current priority of the process which may change with aging.

-Time in CPU needed before/between I/O (set once)

-Time I/O takes (set once)

-Total time in machine not counting time in wait queue (set at startup counts down).  How much time process spends doing something before it exits.  Sum of total time in CPU and total time in I/O.

-Time in CPU currently.  Set to 0 when moved into CPU, when reaches Time in CPU or quantum is up moves out of CPU.

-Time left waiting for current I/O

-Time process has been waiting in ready queue

-Total time in CPU

-Total time in I/O

-Total time in ready queue

-Count of # of times in ready queue

-Smallest time in ready queue

-Longest time in ready queue

You should have modules (functions() ) to

-Swap process in CPU (how hard would it be to simulate this taking time as it does in real life?  Not required but think about changing your simulator for latency.  Context switch between kernel and user takes even longer in real system)

-Add process to ready queue

**Program:**

Define at least 3 \* 16 processes with different starting times from input file into array

Add one to the ready queue, run the loop once then add the next.

loop until all processes exit

Each iteration is one (1) time unit.

Adjust time counters in each process

swap processes as necessary from CPU to either the I/O or ready queue

swap processes from I/O to ready queues when I/O time is up

Adjust priority of processes in ready queue so no processes wait too long and so higher priorities run first.

After the loop stops and all process have exited call function provided to print the statistics for

               the OS setup for time limits

each process’s stats including total, min and max times

Document your program:

You can read about real simulations online.  Your simulation is more to show how schedulers work than evaluation of algorithms.

Explanation of the Round Robin Priority Queue with ageing algorithm, the required one.

Priority Scheduling

Priority scheduling is when which each job is assigned a priority and the job with the highest priority gets scheduled first.

Priority scheduling can suffer from a major problem known as indefinite blocking, or starvation, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.

If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m. ), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years. )

One common solution to this problem is aging, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

Round Robin Scheduling

CPU bursts are assigned with limits called time quantum.

When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.

If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU and either exits or is waiting for I/O

If the timer goes off first, then the process is swapped out of the CPU and moved to the ready queue.

typedef unsigned short ui;

struct process\_struct {

ui priority; // never changes

ui cpu;  // time in cpu before I/O

ui io; // time I/O takes

ui runTime; // how long process runs

ui curCpu; // count of current time in CPU

ui curIo;  // count of time waiting for I/O

ui wait; // current count of time in wait queue

ui curPrior; // adjusted for starvation

ui cpuTotal; // sum of time in cpu

ui ioTotal; // sum of time doing io

// statistics

ui waitSum; // total time in wait queue

ui waitCount; // how many times in wait queue (for average)

ui waitMin; // smallest time in wait queue

ui waitMax; // longet time in wait queu

};

typedef struct process\_struct process;

struct os\_struct { ui quantum; ui wait; };

typedef struct os\_struct os;

examples of main variables

process a[48];

// process id is array index

// variables below hold index to process in this array (process id)

       ui queue[48]; ui queueCount = 0;

       ui io[48];  ui ioCount = 0;

       ui cpu;