Ian Mallarino

z23369342

COP4610

Process scheduler analysis

Table of Contents

Introduction

With the exception of single-program operating systems, almost all modern operating systems use some type of scheduler. A scheduler is the part of the operating system that decides the order and length of processes' running times, allowing different processes to share a single set of hardware. A scheduler usually does this by breaking up a process into smaller slices, switching to other processes and allowing them to run intermittently before switching back to the original process for another set length of time. It can also just schedule whole processes to go one at a time until they need to wait for an I/O in some methods. The process is put in to this cycle until the time comes that the process finally terminates. Different schedulers can vary in multiple ways including how they choose the processes to be run, how long the process is allowed to run for, which processes take priority over others and a variety of other decisions. Different schedulers can be more or less useful than others in different situations, depending on the needs of the situation.

One such decision that can take multiple forms is preemption. Generally, this takes the form of prematurely stopping what a process is doing in favor of another, newer, generally more optimal process. This is not limited to processes, though, and can be implemented among multiple queues. The process does not necessarily need to be stopped in this case for the preemption to occur. The scheduler just switches back to the more important queue regardless of what exists in the currently selected queue. The idea behind preemption is to increase the efficiency of the scheduler by allowing processes that are known to be shorter to take priority over those that are longer, regardless of what is currently happening. If the longer processes were allowed to run when there are shorter processes waiting, there could be a case that the processor runs through all the processes in order of decreasing length, leaving the processor without anything to process, thus increasing the total running time of all the programs.

Preemption in this case is used on a queue-by-queue basis for the simulation of the final scheduling algorithm of the three being analyzed.

The first simulation involves the First Come First Served (FCFS) algorithm. This algorithm only runs processes as they arrive with no other decisions made about which of these processes get run. This approach is uncommon in modern operating systems as it is considered to be a naive approach. This is because while it gets the job done in most respects, it is rarely very efficient. A long process can starve all other processes, causing everything else to run slower. It also doesn't account for if processes might need to be re-run after its initial run, and thus might empty out processes that might take a longer time to come back later than other processes. Overall, this algorithm is rarely expected to produce the best running time with any given set of processes.

The next simulation is for the Shortest Job First (SJF) algorithm. This algorithm is expected to produce the best possible outcome out of all other scheduling algorithm in terms of efficiency. However, the issue with this algorithm is that in reality the length of the job is unknown prior to execution, making implementation almost impossible. This is slightly circumvented by using a formula that allows an approximate prediction to be made based on prior jobs, but it may always be wrong. We call this algorithm instead the Approximate Shortest Job First (ASJF). In this case the implementation of this algorithm is non-preemptive, making it almost exactly like the first come first served algorithm. The only difference is the order in which the jobs are chosen. A preemptive implementation of this algorithm would instead swap processes mid-execution if a process became available which had an expected time shorter than that of the remaining time of the current running process.

Finally, we have the Multi-level Feedback Queue (MLFQ) algorithm. This algorithm involves many different ready queues, each with their own method of running their processes and each with their

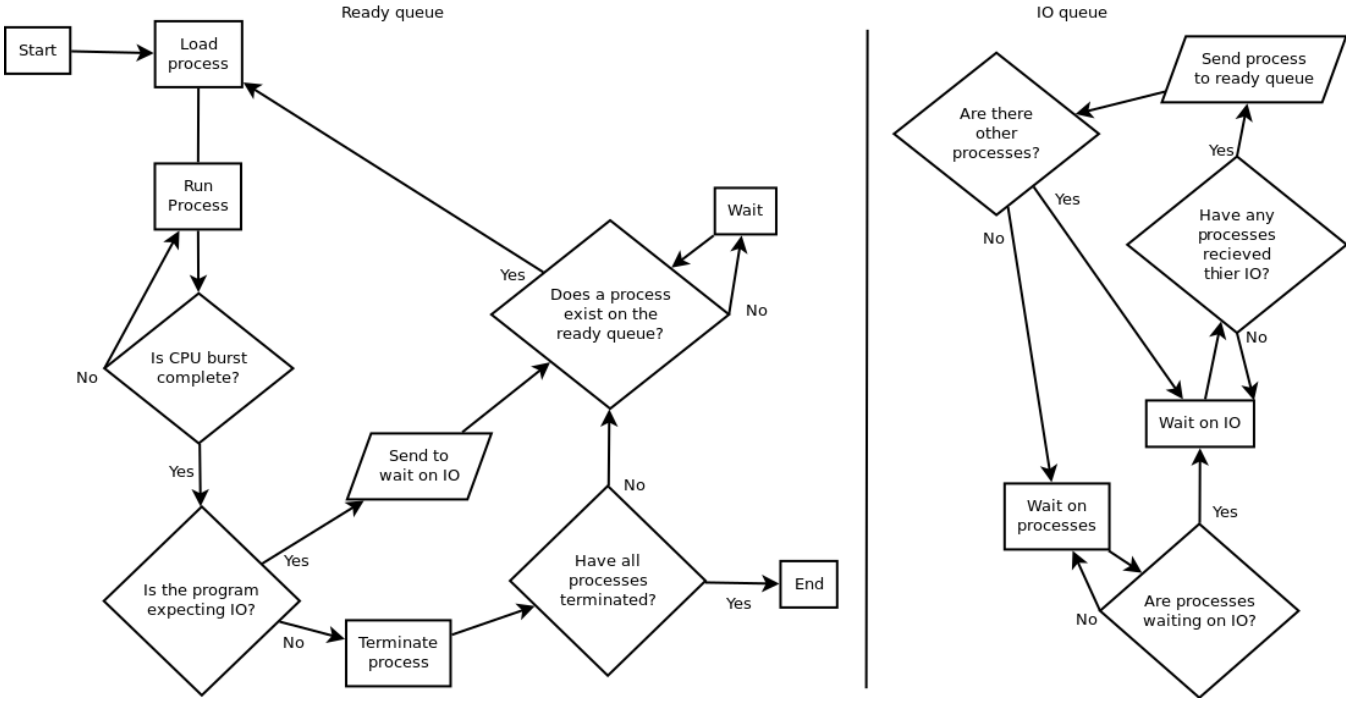
own priorities. The highest priority queue feeds in to the next highest, and this repeats until the last queue. This MLFQ in particular involves using Round Robin (RR) for the first two, which is another type of scheduling algorithm, followed by a FCFS for the last queue. In RR, all processes get equal time on the CPU, but they all get cut off by a time quantum and pushed to the back of the queue if the length of their job would otherwise exceed that limit. The process in this case is pushed to the back of the next lowest priority queue. MLFQ is considered to be much more efficient than FCFS due to its prioritization of processes that have shown to contain smaller execution times. Jobs which are longer don't get run until all the shorter jobs have gotten a chance to run. The simulation uses preemption for this algorithm in that the scheduler will switch to higher priority queues if a process shows up on any of them, regardless of what queue the scheduler is currently picking from.

With the framework for the basics of schedulers laid down, there is finally the simulation. The simulation is actually three separate simulations of three different scheduling algorithms, all contained in their own part of the program using the same processes. The program involves four of the five aforementioned schedulers, mainly using FCFS, SJF and MLFQ, but by extension of MLFQ, RR. Each simulation starts off in a similar way, resetting each of the queues and setting up the timers for the simulations to use. Each process has a value to record various important details about its execution and termination. During the simulation, queues are displayed every time a context switch occurs, showing the processes contained within each one, and the remaining length of their current job. Details are updated through the length of the simulation and displayed for the user at the end. The important details of each process are its turnaround time, waiting time, and response time. The important values used by the simulation as a whole are the total time and execution time. At the end of the simulation, these details are displayed for the user and other derived details are calculated and displayed. The results of

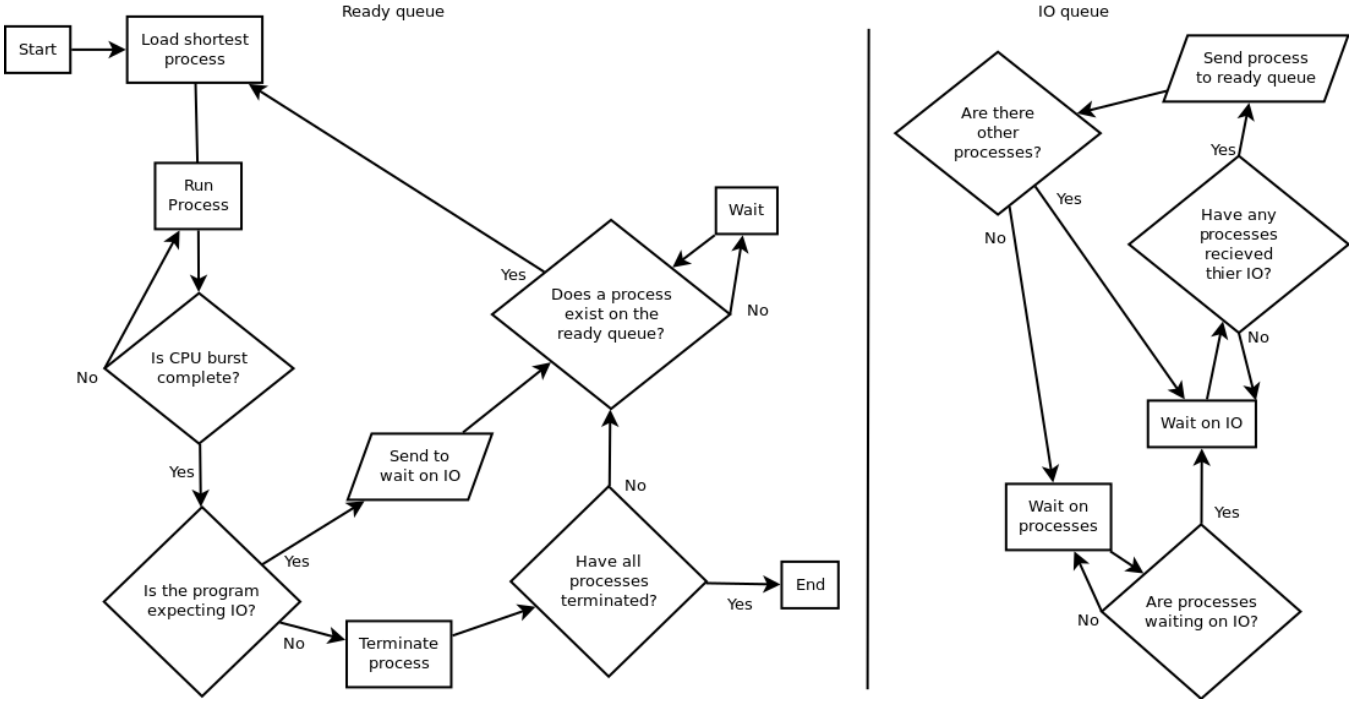
the output are based on the process' initial jobs and starting times which, for the sake of the current measurement, is assumed to be zero. This all leads to getting the results to test the efficiency of each algorithm in question with a side-by-side comparison of the individual processes, averages and times.

Conceptual visualizations

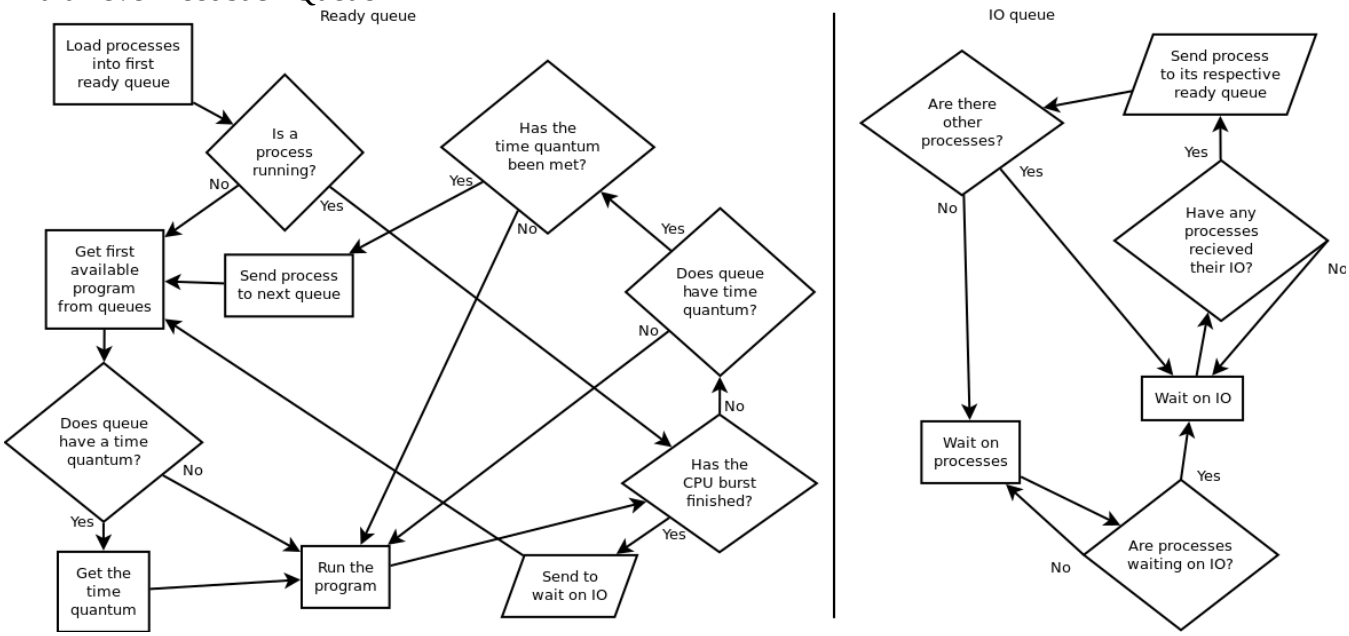
First Come First Served



Shortest Job First



Multi-level Feedback Queue



Sample of scheduling

(first nine context switches)

First Come First Served

Shortest Job First

Multi-level Feedback Queue

Results

Processes

	SJF CPU utilization: 89.7934%			FCFS CPU utilization: 82.39%			MLFQ CPU utilization: 89.4%		
	Tw	Ttr	Tr	Tw	Ttr	Tr	Tw	Ttr	Tr
P1	67	425	0	313	668	0	74	431	0
P2	249	728	81	274	756	4	291	773	4
P3	74	298	9	340	562	22	90	313	9
P4	279	823	45	198	746	28	282	827	15
P5	79	718	4	260	897	45	59	698	20
P6	56	362	15	307	609	50	300	600	25
P7	405	776	403	220	598	60	244	620	30
P8	146	519	25	317	685	81	343	713	36
Avg	169.375	581.125	72.75	278.625	690.125	36.25	210.375	621.875	17.375

Averages

	SJF	FCFS	MLFQ
CPU utilization	89.7934%	82.3857%	89.3591%
Avg Wait time (WT)	169.375	278.625	210.375
Avg Turnaround time (TT)	581.125	690.125	621.875
Avg Response time (RT)	72.75	36.25	17.375

Conclusion

As expected, SJF was the quickest to finish and FCFS took the longest, but MLFQ followed fairly closely behind SJF with only a fraction of a percent of CPU utilization off. Response time has been extended a lot longer due to process seven being starved of resources before it was finally given a chance to go. In all respects MLFQ has gotten better average results than FCFS as well as CPU utilization.

Source

main.cpp

```
/*****************************************************************************
Ian Mallarino
z23369342
COP4610 - programming project
Simulation, analysis and comparison of three process scheduling algorithms:
    First Come First Served(FCFS)
    Shortest Job First(SJF)
    Multi-level Feedback Queue(MLFQ)
Collects and displays information relevant to their efficiencies such as
    turn around times, waiting times, and response times of each process
    as well as CPU utilization for any given set of processes' io and cpu
    bursts. Also displays process states at context switches.
*****************************************************************************/

#include<iostream>
#include<vector>    //container for processes and their bursts

#include"Process.h"
#include"scheduler.h"

using namespace std;

int main(){
    vector<Process> refList={   //Processes in their initial states for reference
only; Do not change.
        Process({4,24,5,73,3,31,5,27,4,33,6,43,4,64,5,19,2}),
        Process({18,31,19,35,11,42,18,43,19,47,18,43,17,51,19,32,10}),
        Process({6,18,4,21,7,19,4,16,5,29,7,21,8,22,6,24,5}),
        Process({17,42,19,55,20,54,17,52,15,67,12,72,15,66,14}),
        Process({5,81,4,82,5,71,3,61,5,62,4,51,3,77,4,61,3,42,5}),
        Process({10,35,12,41,14,33,11,32,15,41,13,29,11}),
        Process({21,51,23,53,24,61,22,31,21,43,20}),
        Process({11,52,14,42,15,31,17,21,16,43,12,31,13,32,15})
    };
    vector<int> TQ={6,11,-1};

    for(unsigned a=0;a<refList.size();a++)  //assign each process its own
identifier to be recalled later
        refList[a].procNo=a+1;

    printResults(FCFS(refList,true));
    printResults(SJF(refList,true));
    printResults(MLFQ(refList,TQ,true));

    return 0;
}
```

Process.h

```
#ifndef PROCESS_H
#define PROCESS_H

#include<vector>

class Process{
    public:
        Process();
        Process(const Process&);
        Process(std::vector<int>,int=0);
        virtual ~Process();

        int procNo;

        int start;  //starting time
        int turnAround; //turnaround time
        int waiting;    //waiting time
        int response;   //response time

        int cpuTotal;   //cumulative cpu time
        int ioTotal;    //cumulative io time

        int currentTime;    //current burst timer
        unsigned currentBurst;   //counter for the cpu/io bursts
        unsigned currentQueue;   //for mlfq; decides the feedback queue it belongs
in

        bool activated; //flag for response time

        std::vector<int> bursts;    //list of cpu/io bursts

        void operator=(Process);    //copys details of another process
        void operator=(std::vector<int>);   //sets bursts to new set of times

        int getCPU(int);    //returns the value of the nth CPU burst
        int getIO(int); //returns the value of the nth IO burst
        int getBurst(); //returns the value of the current burst

        void nextBurst();   //resets the current time and increments the burst
counter
        void addBurst(int,int); //adds io and cpu bursts to set of bursts
        void reset();   //deletes the bursts and sets everything to zero

        bool isFinished();  //checks if current burst is finished
};

#endif // PROCESS_H
```

Process.cpp

```
#include "Process.h"

Process::Process(){
    reset();
}

Process::Process(const Process& p){
    procNo=p.procNo;

    start=p.start;
    turnAround=p.turnAround;
    waiting=p.waiting;
    response=p.response;

    currentTime=p.currentTime;
    currentBurst=p.currentBurst;
    currentQueue=p.currentQueue;

    activated=p.activated;

    bursts=p.bursts;
    cpuTotal=p.cpuTotal;
    ioTotal=p.ioTotal;
}

Process::Process(std::vector<int> burstList,int startTime){
    reset();
    *this=burstList;
    start=startTime;
}

void Process::operator=(const Process p){
    procNo=p.procNo;

    start=p.start;
    turnAround=p.turnAround;
    waiting=p.waiting;
    response=p.response;

    currentTime=p.currentTime;
    currentBurst=p.currentBurst;
    currentQueue=p.currentQueue;

    activated=p.activated;

    bursts=p.bursts;

    cpuTotal=p.cpuTotal;
    ioTotal=p.ioTotal;
}
```

```
void Process::operator=(std::vector<int> newBursts){
    bursts=newBursts;
    cpuTotal=0;
    ioTotal=0;
    for(unsigned a=0;a<bursts.size()-1;a+=2){
        cpuTotal+=bursts[a];
        ioTotal+=bursts[a+1];
    }
    cpuTotal+=bursts[bursts.size()-1];
}

int Process::getCPU(int index){
    return bursts[2*index];
}

int Process::getIO(int index){
    return bursts[2*index+1];
}

int Process::getBurst(){
    return bursts[currentBurst];
}

void Process::nextBurst(){
    currentBurst++;
    currentTime=0;
}

void Process::addBurst(int io,int cpu){
    bursts.push_back(io);
    bursts.push_back(cpu);
    cpuTotal+=cpu;
    ioTotal+=io;
}

void Process::reset(){
    procNo=0;

    start=0;
    turnAround=0;
    waiting=0;
    response=-1;

    currentTime=0;
    currentBurst=0;
    currentQueue=0;

    activated=false;

    bursts.clear();

    cpuTotal=0;
    ioTotal=0;
```

```
}

bool Process::isFinished(){
    return getBurst()==currentTime;
}

Process::~Process(){
    //dtor
}
```

funcs.h

```
#ifndef FUNCS_H_INCLUDED
#define FUNCS_H_INCLUDED

#include<vector>

#include"Process.h"

using namespace std;

//supplementary for quicksort
int partition(vector<Process>& vec,int first,int last){
    int i=first-1;

    for(int a=first;a<=last;a++)
        if(vec[a].procNo<=vec[last].procNo){
            i++;
            exchange(vec[a],vec[i]);
        }
    return i;
}

//sorts processes by their process number using quicksort
void sortProcesses(vector<Process>& vec,int first=0,int last=-1){
    if(last==-1)
        last=vec.size()-1;
    if(first<last){
        int div=partition(vec,first,last);
        sortProcesses(vec,first,div-1);
        sortProcesses(vec,div+1,last);
    }
}

//return whether or not a program exists on the ready queue(s)
bool isReady(vector<vector<Process> > ready){
    for(unsigned a=0;a<ready.size();a++)
        if(ready[a].size())
            return true;
    return false;
}

//select a program to run from the ready queues from highest to lowest priority
bool run(vector<vector<Process> >& ready,vector<Process>& run,int time){
    for(unsigned a=0;a<ready.size();a++)
        if(ready[a].size()){
            transfer(ready[a],run);
            if(!run[0].activated){  //check if the program has responded before
                run[0].response=time-run[0].start;  //record the response time
after the first burst starts
                run[0].activated=true;  //flag the program as having been started
            }
            return true;
```

```
        }
    return false;
}

//display the processes and their current bursts contained within a given list
void listProcesses(vector<Process> pList){
    for(unsigned a=0;a<pList.size();a++){
        cout<<"P"<<pList[a].procNo<<": "<<pList[a].getBurst();
        if(a<pList.size()-1)
            cout<<", ";
        else
            cout<<'\n';
    }
}

//take a sample of the current activities of all the processes and current time
void sampleProcesses(vector<Process> run,vector<Process> io,vector<vector<Process>
> waiting,int time){
    cout<<"Current time: "<<time<<endl;
    if(run.size()){
        cout<<"Running:"<<endl;
        listProcesses(run);
    }
    if(io.size()){
        cout<<"Waiting on IO:"<<endl;
        listProcesses(io);
    }
    for(unsigned a=0;a<waiting.size();a++)
        if(waiting[a].size()){
            cout<<"Ready queue "<<a+1<<":"<<endl;
            listProcesses(waiting[a]);
        }
    cout<<"----------------------------"<<endl;
}

//prints the values of the desired times, their averages, and the utilization of
the CPU for the set of processes given
void printResults(vector<Process> pList,int totalTime,int CPUTime){
    sortProcesses(pList);

    cout<<endl<<"Results:"<<endl;
    cout<<"CPU utilization: "<<100.0*CPUTime/totalTime<<"%"<<endl;
    cout<<"Process"<<"\t"
        <<"Tw"<<"\t"
        <<"Ttr"<<"\t"
        <<"Tr"<<endl;
    double waitingSum=0,turnAroundSum=0,responseSum=0;
    for(unsigned a=0;a<pList.size();a++){
        cout<<pList[a].procNo<<"\t"
            <<pList[a].waiting<<"\t"
            <<pList[a].turnAround<<"\t"
            <<pList[a].response<<"\t"<<endl;
        waitingSum+=pList[a].waiting;
```

```
        turnAroundSum+=pList[a].turnAround;
        responseSum+=pList[a].response;
    }
    cout<<"Avg:"<<"\t"
        <<waitingSum/pList.size()<<"\t"
        <<turnAroundSum/pList.size()<<"\t"
        <<responseSum/pList.size()<<endl;
    cout<<"Total time: "<<totalTime<<endl<<endl;
}

void printResults(vector<Process> pList){
    int CPUTime=0,totalTime=0;
    for(unsigned a=0;a<pList.size();a++){
        if(totalTime<pList[a].turnAround)
            totalTime=pList[a].turnAround;
        CPUTime+=pList[a].cpuTotal;
    }

    printResults(pList,totalTime,CPUTime);
}

//starts processes at the time they are set to start
void loadReady(vector<Process>& starting,vector<Process>& ready,int time){
    for(unsigned a=0;a<starting.size();)
        if(time>=starting[a].start)
            transfer(starting,ready,a);
        else
            a++;
}

//increases the process' waiting times
void updateReady(vector<vector<Process> >& ready,int time=1){
    for(unsigned a=0;a<ready.size();a++)
        for(unsigned b=0;b<ready[a].size();b++)
            ready[a][b].waiting+=time;
}

//updates the list of processes waiting on io and puts them back on the ready list
when they are done
void updateIO(vector<Process>& io,vector<vector<Process> >& ready,int time=1){
    for(unsigned a=0;a<io.size();a++){
        if(time>io[a].getBurst()-io[a].currentTime){
            time-=io[a].getBurst()-io[a].currentTime;
            io[a].currentTime=io[a].getBurst();
            io[a].waiting+=time;
        }else
            io[a].currentTime+=time;
        if(io[a].isFinished()){
            io[a].nextBurst();
            transfer(io,ready[io[a].currentQueue],a);
        }
    }
}
```

```
//returns the index of the processor with the shortest burst in a list of processes
int getShortest(vector<Process> p){
    int shortest=0;
    for(unsigned a=1;a<p.size();a++)
        if(p[shortest].getBurst()>p[a].getBurst())
            shortest=a;
    return shortest;
}

#endif // FUNCS_H_INCLUDED
```

generic.h

```
#ifndef GENERIC_H_INCLUDED
#define GENERIC_H_INCLUDED

#include<vector>

//moves an element from one vector to another
template<class c>
void transfer(std::vector<c>& srcList,std::vector<c>& destList,int index=0){
    destList.push_back(srcList[index]);
    srcList.erase(srcList.begin()+index);
}

//exchanges two variables
template<class c>
void exchange(c& e1,c& e2){
    c tempElem=e1;
    e1=e2;
    e2=tempElem;
}

#endif // GENERIC_H_INCLUDED
```

scheduler.h

```
#ifndef SCHEDULER_H_INCLUDED
#define SCHEDULER_H_INCLUDED

#include"generic.h"
#include"funcs.h"

/************************************************
First Come First Served
Executes each process as they become available to
    completion.
************************************************/
vector<Process> FCFS(vector<Process> newList,bool print=false){
    vector<Process> running,IOList,terminatedList;  //each of the lists needed in
the simulation
    vector<vector<Process> > readyList; //ready list for the multiple ready queues
there will be with MLFQ
    readyList.resize(1);
    int time=0,CPUTime=0;

    if(print)
        cout<<"FCFS:"<<endl;

    while(newList.size()||running.size()||IOList.size()||isReady(readyList)){
//check that any of the queues contain processes
        updateReady(readyList);
        updateIO(IOList,readyList);
        loadReady(newList,readyList[0],time);   //load processes on to the ready
queue as they reach their starting times

        if(running.empty()) //check if there are any programs currently running
            if(run(readyList,running,time)&&print) //check and pull a process from
the ready queue
                sampleProcesses(running,IOList,readyList,time);

        if(running.size()){ //if a program is running
            CPUTime++;  //tick up the time for how long the cpu is being used
            running[0].currentTime++;   //tick down the cpu burst
            if(running[0].isFinished()){   //check if the cpu burst is completed
                running[0].nextBurst(); //move to next burst
                if(running[0].currentBurst==running[0].bursts.size()){  //check if
all bursts are finished
                    running[0].turnAround=time-running[0].start;    //record the
turnaround time once the program terminates
                    transfer(running,terminatedList);   //terminate the program
                    if(print)
                        cout<<"Program terminated."<<endl;
                }else
                    transfer(running,IOList);   //send the program to wait on io
            }
        }
        time++; //tick up the overall time
```

```
    }
    return terminatedList;
}

/************************************************
Shortest Job First
Chooses the process with the shortest cpu burst to
    be run if any are available.
************************************************/
//Modified version of the FCFS algorithm that changes the order processes are
chosen to run based on burst size
vector<Process> SJF(vector<Process> newList,bool print=false){
    vector<Process> running,IOList,terminatedList;
    vector<vector<Process> > readyList;
    readyList.resize(1);
    int time=0,CPUTime=0;

    if(print)
        cout<<"SJF:"<<endl;

    while(newList.size()||running.size()||IOList.size()||isReady(readyList)){
        updateReady(readyList);
        updateIO(IOList,readyList);
        loadReady(newList,readyList[0],time);

        if(running.empty())
            if(readyList[0].size()){
                transfer(readyList[0],running,getShortest(readyList[0]));   //chose
the lowest burst on the ready queue
                if(!running[0].activated){  //check if the program has responded
before
                    running[0].response=time-running[0].start;  //record the
response time after the first burst starts
                    running[0].activated=true;  //flag the program as having been
started
                }
                if(print)
                    sampleProcesses(running,IOList,readyList,time);
            }

        if(running.size()){
            CPUTime++;
            running[0].currentTime++;
            if(running[0].isFinished()){
                running[0].nextBurst();
                if(running[0].currentBurst==running[0].bursts.size()){
                    running[0].turnAround=time-running[0].start;
                    transfer(running,terminatedList);
                    if(print)
                        cout<<"Program terminated."<<endl;
                }else
                    transfer(running,IOList);
            }
```

```
        }
        time++;
    }
    return terminatedList;
}

/************************************************
Multi-level Feedback Queue
Sets up a time quantum for the first n-1 queues. If
    a process exceeds those time quantums it is sent
    to a lower priority queue. If a process exists on
    a higher priority queue the program will always
    choose a process from that queue. Once a process
    loses priority, it cannot be restored.
************************************************/
//Modified version of FCFS which adds a time quantum and multiple ready queues for
processes that are too long
vector<Process> MLFQ(vector<Process> newList,vector<int> TQ,bool print=false){
    vector<Process> running,IOList,terminatedList;
    vector<vector<Process> > readyList;
    readyList.resize(TQ.size());   //initialize the next two queues
    int time=0,CPUTime=0;
    int tqRemain;   //remaining time for current process

    if(print)
        cout<<"MLFQ:"<<endl;

    while(newList.size()||running.size()||IOList.size()||isReady(readyList)){
        updateReady(readyList);
        updateIO(IOList,readyList);
        loadReady(newList,readyList[0],time);

        if(running.empty())
            if(run(readyList,running,time)){
                tqRemain=TQ[running[0].currentQueue];
                if(print)
                    sampleProcesses(running,IOList,readyList,time);
            }

        if(running.size()){
            CPUTime++;
            running[0].currentTime++;
            if(running[0].isFinished()){
                running[0].nextBurst();
                if(running[0].currentBurst==running[0].bursts.size()){
                    running[0].turnAround=time-running[0].start;
                    transfer(running,terminatedList);
                    if(print)
                        cout<<"Program terminated."<<endl;
                }else
                    transfer(running,IOList);
                if(run(readyList,running,time)){
                    tqRemain=TQ[running[0].currentQueue];
```

```
                    if(print)
                        sampleProcesses(running,IOList,readyList,time);
                }
            }

            //checks if the time quantum has run out and downgrades the process to
the next queue if applicable
            tqRemain--; //ticks down the remaining time quantum
            if(tqRemain==0){    //if the program is still running but the time
quantum has run out
                running[0].currentQueue++;  //increase the program's queue
                transfer(running,readyList[running[0].currentQueue]);   //send the
program to that queue
            }
        }
        time++;
    }
    return terminatedList;
}

#endif // SCHEDULER_H_INCLUDED
```