Project 2

Course: CMPT 300

Section: D100

Name:

Lvyu Ye	lvyuy@sfu.ca
Liwei Jin	liweij@sfu.ca
Di Fu	dif@sfu.ca

Date: Nov.16th, 2013

Lvyu Ye	301239914	lvyuy@sfu.ca
Liwei Jin	301239922	liweij@sfu.ca
Di Fu	301239912	dif@sfu.ca

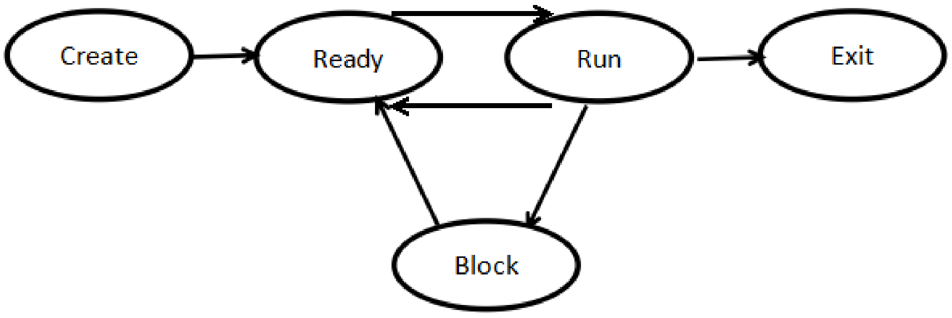
1. Introduction

This is an Operating System scheduling simulation presented in C++. It successfully combined object oriented programming and pthread.h API, and thus there are multiple threads involved, executing concurrently. To tackle the IPC issues, this project “fakes” a way to implement a monitor mechanism very like what in Java. Besides, the ready queue follows the rules of a Multilevel Feedback Queue (MLFQ) design. This implementation has turned out to be the key making this simulation, as it shows in this report, a great success.

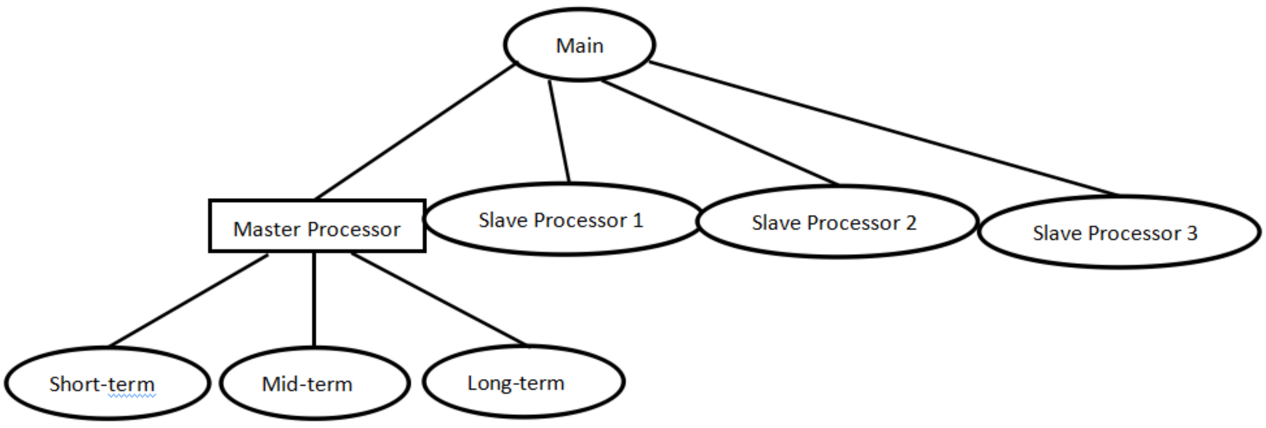
This report contains five sections, besides introduction, there are: Section 2 with high-level architectural design descriptions, comprised of three subsections contributing to depict the architecture of the program and the process of conducting the simulation with it; Section 3 gives an overview on how we fake the “monitor” mechanism in C++; Section 4 gives an overview on how we designed and implemented the Multilevel-Feedback Queue (MLFQ) for this simulation; Section 5 provides a module dependency diagram for all the .h and .cpp files; Section 6 focuses on analyzing the simulation feedbacks on a common running example; Section 7 provides the code and explain the correctness of them.

2. High-level Architectural Design Description

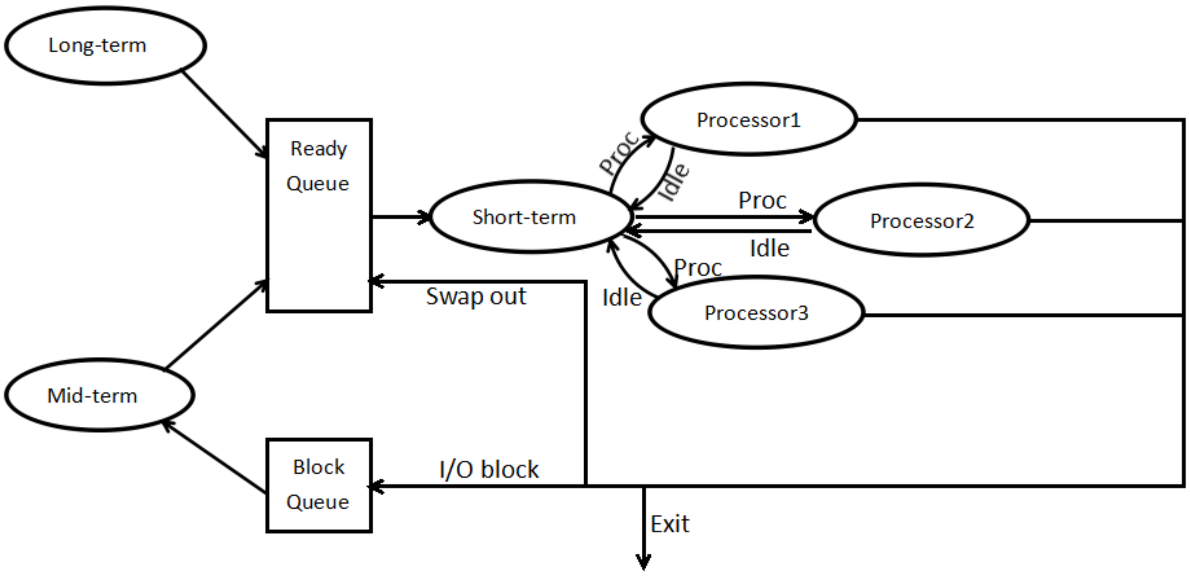
Scheduling Environment



Each process has **five states** as shown above. The transitions are made by the schedulers.



Seven threads coexist in this scheduling environment. Main thread creates an object “Master Processor”, and when this Master Processor is on create, it spawns out three threads. Then the Main threads create three Slave Processors, which will run the “codes” of a program afterwards.



This graph depicts how the schedulers schedule the processes around different queues and processors. The UNIX "pipe" facility is used to achieve communication between short-term scheduler and three processors (six pipes as shown in the graph, those arrows with "Proc" on it and those with "Idle" below it).

Simulation Experiment

The experiment, in general, is designated to simulate the CPU scheduling in reality. The developing process can be divided into several stages as below.

Firstly, our team spent about a week figuring out the design issues. Those issues includes, but may not include all, the programming paradigm of this program (or **using C or C++**); how to **fake the "monitor"** mechanism and make it really useful (it has been proved that this "monitor" indeed useful); the design issues of the Multilevel-Feedback Queue; how to achieve the communication between threads; how to simulate a process; how to simulate a block queue etc. We also consulted our teaching assistant Jiten for quite a few helpful suggestions.

Next, a prototype was coded according to the design in about 3 days; the workflow was maintained by Git and Github. Another week was spent on testing, debugging and bettering off this prototype. The design was modified hugely, yet more reasonably along the developing process.

The simulation was conducted at the same time. Results were recorded into separate files and analyzed, according to which the design was modified. And finally, a simulation report was formed as presented here

Main Program Logic

The logic of running thread structure:

There are 1 mother thread and 6 children threads running.
The mother thread creates children threads.
6 children threads stand for:

1. Long-term scheduler
2. Mid-term scheduler
3. Short-term scheduler
4. Processor 1
5. Processor 2
6. Processor 3

Those 6 children threads act together to form the basic functions of operating system.

---
The logic of data structure:

1. Ready queue is an individual object, which was first initialized in main function.
2. Block queue is an individual object, which was first initialized in main function.
3. Each processor is an individual object, which was first initialized in main function.
4. Master Processor is an individual object, including short-term, mid-term and long-term schedulers, which was first initialized in main function.
5. Each process is an individual object, which was first initialized in long-term scheduler and deleted in long-term scheduler after finishing executing.

3. Overview of Monitor Implementation

This project provides a way to achieve mutual exclusion, with the public interface:

```
synchronized(M) { ..codes }
```

where M is a mutex.

The idea to this implementation comes from the observation to the nature of "for" statement in C/C++. In C/C++, "for" statement is used in this manner:

```
for(statement 1; statement 2; statement3){..codes}
```

where statement1 is executed **at the beginning of** the for loop, then statement 2 is executed right after; statement 3 is executed **at the end of** the for loop. Here comes the inspiration: why not use this nature to fake a monitor, which also has the nature of "obtaining the lock **at the beginning of** a chunk of code" and "releasing the lock **at the end of** the code".

So, the implementation of this "monitor" comes naturally: statement 1 will try to obtain the lock/mutex; statement 2 will test the lock, block current thread if the lock is already in possession of other threads; statement 3 will release the lock/mutex when the critical section ends.

In order to make this looks more like a real monitor (i.e. not wring the for loop explicitly), we encapsulate this implementation into a class "Lock" in "monitor.h" file; using a macro:

```
#define synchronized(M) for(Lock M##_lock=M; M##_lock; M##_lock.setUnlock())
```

to fully achieve such a synchronized() fashion as a monitor.

Also worth mentioning here, since there are issues similar to the "consumer/producer", **conditional variable** is also introduced to coordinate the "consumer" and "producer" in this simulation context.

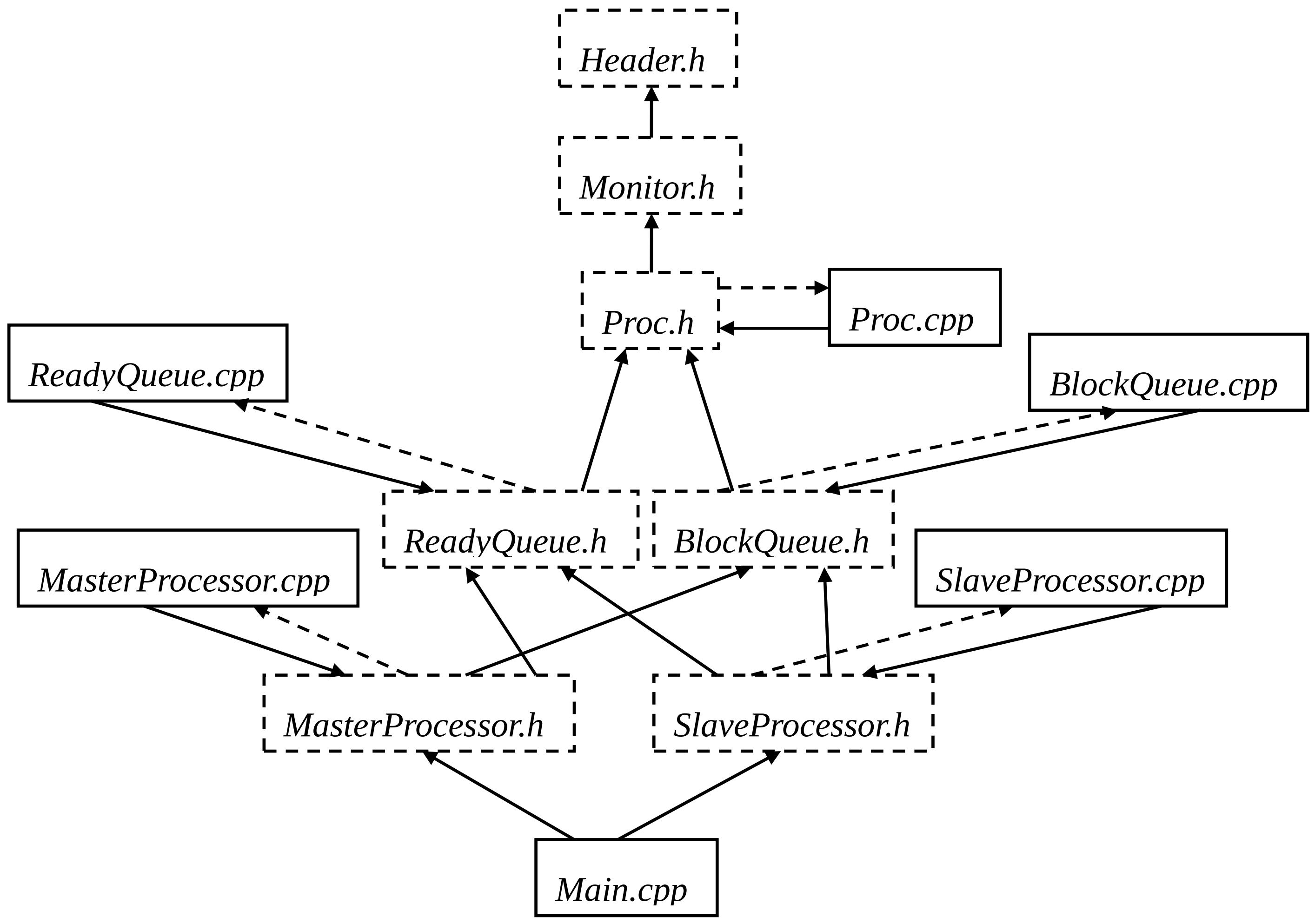
4. Overview of MLFQ Implementation

We designed a *Multilevel-Feedback Queue*, with multiple rules to determine the running sequence of processes waiting in the ready queue. The rules are:

"Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
*Rule 2: If Priority(A) = Priority(B), A & B run in **Round Robin**.*
Rule 3: When a process enters the system, it is placed at the highest priority.
Rule 4: Once a process uses up its time allotment at a given level (regardless of how many time it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
Rule 5: Once a process gives up the CPU because of IO, when it is moved back to the ready queue, its priority will not be reduced (i.e. put into the same queue as last time)
Rule 6: After some time period S, move all the jobs in the system to the topmost queue."

Rule 1 is implemented by **dividing the ready queue into 3 sub-queues**, each sub-queue will contain those processes with corresponding priority; Rule 2 is achieved by **making the 3 sub-queues FCFS queues;** Rule 3 is implemented in such a way when a process is created, **it will always be put into the queue with the highest priority**; Rule 4 is implemented in a manner that if a process used up its time quanta, it will be moved back to the ready queue with **a reduced priority** and put into the corresponding sub-queue; Rule 5 is achieved by **not** reducing the priority when moving the process from a processor to block queue; Rule 6 is implemented in a way that the processes will be granted a **periodically priority boost**.

5. Module Dependency Diagram



6. Test Analysis

Overview of Performance Metrics

A short snap of a small part of an example of running result:

```
#################################################################################################
                                                              (3) Process(PID=169) executing
                                                                  Commands to be run:77
## Long-Term-Scheduler ##:
  Process(PID=259) is created
  Put in ReadyQueue
## Short-Term-Scheduler ##:
  Process(PID=171,Priority=3)
  Moved out from ReadyQueue
  with timeQuanta=20
  Assigned to Processor(2)
                    (1) Process(PID=170) executing
                        Commands to be run:30
                                                              (3) Process(PID=169) swapped out
                                                                  Changed priority to 2
                                                                  Back to ReadyQueue
                                                                  Commands to be run:57
## Long-Term-Scheduler ##:
  Process(PID=260) is created
  Put in ReadyQueue
                    (1) Process(PID=170) IO-Block
                        Put in BlockQueue
                        Commands to be run:29
                                          (2) Process(PID=171) executing
                                              Commands to be run:76
## Short-Term-Scheduler ##:
  Process(PID=172,Priority=3)
  Moved out from ReadyQueue
  with timeQuanta=20
  Assigned to Processor(3)
## Long-Term-Scheduler ##:
  Process(PID=261) is created
  Put in ReadyQueue
## Mid-Term-Scheduler ##:
  Process(PID=170) IO-Block ends
  Moved out from BlockQueue
  Put in ReadyQueue
                                          (2) Process(PID=171) IO-Block
                                              Put in BlockQueue
```

```
                                   Commands to be run:56
##########################################################################################
```

In this example, the **PID, rest commands to be run, priority, running time quanta and status of each process are collected as metrics.**

Analysis of Simulation Results

In the result above, at first:
|| process-PID=170 is running on processor 1;
|| process-PID=169 is running on processor 3;
|| long-term scheduler creates a process-PID=259 and short-term scheduler gets a process-PID=171 with priority=3 out of ready queue and assigns it to the idle processor 2 with time-quanta=20;

And then:
|| process-PID=170 was IO blocked on processor 1;
|| process-PID=169 is running out of time quanta (back to ready queue) on processor 3;
|| process-PID=171 starts executing on processor 2;
|| long-term scheduler creates a process-PID=260;
At this time, processor 1 and processor 3 are both idle.

After a while:
|| process-PID=171 was IO blocked on processor 2;
|| short-term scheduler gets a process-PID=172 with priority=3 out of ready queue and assigns it to the idle processor 3 with time-quanta=20, long-term scheduler creates a process-PID=261 and mid-term scheduler founds that the IO of process-PID=170 ends so mid-term scheduler gets the process-PID=170 out of block queue and put it into ready queue.

It is clear that the result is correct and matches the design.

Performance Characteristics

In the hypothesis of design, the followings are the crucial characteristics:
1. The process should atomically be passed from ready queue to running state by short-term .
2. The process should atomically be passed from block queue to ready queue.
3. Time quanta should be calculated according to the priority of each process by short-term scheduler.
4. After IO-blocking or swapping out happened, the respective processor should be idle.
5. long-term scheduler can create new process with the PID collected from old-already-exit process.
6. All critical sections are protected by mutex.
7. Using pthread_cond_wait() and pthread_cond_signal() to make short-term scheduler block when it asks ready queue for prcess and there is no process in ready queue.

Another test result is provided as reference:

```
## Short-Term-Scheduler ##:
  Process(PID=2,Priority=3)
  Moved out from ReadyQueue
  with timeQuanta=20
  Assigned to Processor(2)
                            (1) Process(PID=1) executing
                                Commands to be run:3
## Short-Term-Scheduler ##:
  Process(PID=3,Priority=3)
  Moved out from ReadyQueue
  with timeQuanta=20
  Assigned to Processor(3)
## Long-Term-Scheduler ##:
  Process(PID=5) is created
  Put in ReadyQueue
                            (1) Process(PID=1) exits
                                                        (2) Process(PID=2) executing
                                                            Commands to be run:27
## Long-Term-Scheduler ##:
  Process(PID=6) is created
  Put in ReadyQueue
                                                                                    (3)      Process(PID=3)
executing
                                                                                        Commands to be run:49
## Short-Term-Scheduler ##:
  Process(PID=4,Priority=3)
  Moved out from ReadyQueue
  with timeQuanta=20
  Assigned to Processor(1)
                                                        (2) Process(PID=2) IO-Block
                                                            Put in BlockQueue
                                                            Commands to be run:19
## Long-Term-Scheduler ##:
  Process(PID=1) is deleted
                                                                                    (3)      Process(PID=3)
IO-Block
                                                                                        Put in BlockQueue
                                                                                        Commands to be run:33
```

7. Correctness Section

This section includes all the code of this simulation program, and comments in detail for each of the files.

header.h:

```
#ifndef HEADER_H
#define HEADER_H
#include <iostream>
#include <string>
#include <vector>
#include <pthread.h>
#include <unistd.h>
#include <fcntl.h>
#include <queue>
#include <list>
#include <cstdlib>
#include <cstdio>
#include <ctime>

using namespace std;

#define PROC_EXIT -1
#define PROC_BLOCK 0
#define PROC_RUN 1
#define PROC_READY 2
#define PROC_CREATE 3

int const LEVEL=3;
int const SLAVES_NUMBER = 3;
int const SLAVE_INDENT_WIDTH = 31;
int const TIME_UNIT = 10;
int const MAX_PROCESS_NUMBER = 1024;
int const IO_WAIT_TIME = 8;
int const MIN_NUM_OF_CODE = 1;
int const MAX_NUM_OF_CODE = 100;
int const BOOST_TRIGGER = 30;
double const CREATE_PROC_FREQUENCY = 10;
double const FOR_A_WHILE = 2;

#endif
```

Comments:
This code is the header of the whole project. It includes all the system library needed, and all the constant parameters of the program (Such as maximum number of processes, levels of MLFQ, etc.).

main.cpp:

```
#include "masterprocessor.h"
#include "slaveprocessor.h"

using namespace std;

int main() {
srand(time(NULL));
int proc_pip[SLAVES_NUMBER][2];
int idle_pip[SLAVES_NUMBER][2];
for (int i=0; i<SLAVES_NUMBER; i++) {
      pipe(proc_pip[i]);
      pipe(idle_pip[i]);
}
ReadyMLFQ *rq = new ReadyMLFQ();

BlockQueue *bq = new BlockQueue();

MasterProcessor *MsP = new MasterProcessor(rq, bq, proc_pip,
idle_pip);

SlaveProcessor *SlP[SLAVES_NUMBER];
for (int i=0; i<SLAVES_NUMBER; i++)
      SlP[i] = new SlaveProcessor(rq, bq, proc_pip[i],
      idle_pip[i], i+1);

pthread_exit(NULL);

delete rq;
delete bq;
delete MsP;
for (int i=0; i<SLAVES_NUMBER; i++) delete SlP[i];

return 0;
}
```

Comments:
This file is the entry of the whole project and the main thread of the program. All the

objects (such as Master Processor, Slave Processor, Ready Queue, Block Queue and pipes for communication between Slave processors and short-term schedulers) are initialized in the function "main". And the main function will hold to wait all its child threads.

monitor.h:

```
#ifndef MONITOR_H
#define MONITOR_H
#include "header.h"

#define   synchronized(M)   for(Lock   M##_lock=M;   M##_lock;
M##_lock.setUnlock())

class Lock
{
public:
Lock(pthread_mutex_t &mutex):m_mutex(mutex),m_locked(true)
{
    pthread_mutex_lock(&mutex);
}
~Lock()
{
    pthread_mutex_unlock(&m_mutex);
}
operator bool() const
{
    return m_locked;
}
void setUnlock()    //unlock the mutex
{
    m_locked = false;
}
private:
pthread_mutex_t &m_mutex;
    bool m_locked;
};
#endif
```

Comments:

This code is the header of the Monitor used in project. It includes the declaration and the definition of the Lock class (which is used in Monitor) and the usage of the macro of the monitor

proc.h/proc.cpp:
```
#ifndef PROC_H
#define PROC_H
#include "monitor.h"

class Proc{
public:

    Proc(int);
    Proc(int, int);
    ~Proc();
    bool isBlocked();
    bool isRunning();
    int getID();
    int getPriority();
    int proc_execute();
    int restCommands();
    void setState(int);
    void changePriority(int);
    int getBlocTime();
    void setBlocTime(int);
protected:
    Proc(){};

private:
    int priority;
    int procID;
    int procType; //cpu-bounded as 2, normal as 1, io-bounded as 0
    int state;  //IPC issue involved on this variable
    int blocTime;
    std::list<int> loc;   //"lines of codes"
    pthread_mutex_t state_mutex;
    void initialize_loc();
};

//This class wrap process and running time together, in order to pass
those info to slave processor
class ProcWrapper{
public:
    ProcWrapper(Proc *pro0, int timeQuanta0) {
        pro = pro0;
        timeQuanta = timeQuanta0;
    }
    Proc *pro;
    int timeQuanta;
protected:
    ProcWrapper(){};
};

#endif
```

```
#include "proc.h"

Proc::Proc(int id){
    priority=LEVEL;    //priority is initialized to LEVEL
    procID=id;
    state=PROC_RUN;
    blocTime=0;
    procType=1;    //normal as default
    /*generate the simulation of a process and IOs*/
    initialize_loc();
    pthread_mutex_init(&state_mutex, NULL);
};

Proc::Proc(int id, int pt){
    priority=LEVEL;
    procID=id;
    state=PROC_RUN;
    procType=pt;
    /*generate the simulation of a process and IOs*/
    initialize_loc();
    pthread_mutex_init(&state_mutex, NULL);
};

Proc::~Proc(){pthread_mutex_destroy(&state_mutex);}

int Proc::getBlocTime(){
    return blocTime;
}

void Proc::setBlocTime(int b){
    blocTime=b;
}

bool Proc::isBlocked(){
    return state == PROC_BLOCK;
}
```

```
bool Proc::isRunning() {
    bool isRun;
    synchronized(state_mutex){
       isRun=(state!=PROC_EXIT);
    }
    return isRun;
}

int Proc::getID(){
    return procID;
}

int Proc::getPriority(){
    return priority;
}

void Proc::setState(int s) {
    synchronized(state_mutex){
    state = s;
    }
}

void Proc::changePriority(int i){
    priority+=i;
};

void Proc::initialize_loc(){
    int     num_of_lines=MIN_NUM_OF_CODE+rand()%(MAX_NUM_OF_CODE     -
MIN_NUM_OF_CODE); //a process will have 10 to 110 lines of code
    int io_odd;  //probablity of a io occurence
    for(int i=0;i<num_of_lines;i++){
        io_odd=rand()%(procType*30+5);     //averagely,  an  io-bounded
proc io occurs 1 out of 5, normal: 1/35, cpu-bounded: 1/65;
        if(io_odd!=0)
            loc.push_front(PROC_RUN);
        else loc.push_front(PROC_BLOCK);
    }
}

int Proc::proc_execute(){
    if(loc.size()==0) return PROC_EXIT;
    else{
        int proc_state=loc.front();
        loc.pop_front();
        return proc_state;
    }
}

int Proc::restCommands() {
    return loc.size();
}
```

Comments:
This code includes the declaration and definition of class Proc, whose object stands for a individual process. It has several attributes such as priority, PID, proc type, block time, proc state and line of code. And it also has some public methods in order to let outsider access the attributes. And each of the process has a state mutex in order to protect critical sections in terms of "state" variable.

A process is simulated using a list<int>, containing series of "1" and "0", where 0 represents an I/O block and 1 a normal instruction.

There is another class in this code called ProcWrapper, which is used to wrap time quanta and Proc together. And short-term scheduler will pass the object of this class to slave processor via pipe.

readyqueue.h/readyqueue.cpp:

```
#ifndef READYQUEUE_H
#define READYQUEUE_H
#include "proc.h"

class ReadyMLFQ{
private:
    pthread_mutex_t readyMLFQMutex;
    pthread_cond_t condc;
    std::queue<Proc *> *readyMLFQ[LEVEL];
    void priorityBoost();
    int boostCounter;
    int totalSize();
public:
    ReadyMLFQ();

    void putProc(Proc *process);

    Proc * getProc();
    ~ReadyMLFQ();
};
#endif

---------------------------------------------------------------------------------------------------
#include "readyqueue.h"

ReadyMLFQ::ReadyMLFQ(){
    pthread_mutex_init(&readyMLFQMutex,NULL);
    pthread_cond_init(&condc, NULL);
    boostCounter=0;
    for(int i=0;i<LEVEL;i++){
        readyMLFQ[i]=new std::queue<Proc *>;
    }
}

void ReadyMLFQ::putProc(Proc *process){
    synchronized(readyMLFQMutex){
        boostCounter++;
        if(boostCounter>=BOOST_TRIGGER){
            priorityBoost();
            boostCounter=0;
        }
        readyMLFQ[process->getPriority()-1]->push(process);
        //If size of the ready queue is just becoming positive, then send
a signal to wake up the waited thread.
        if(totalSize()==1) pthread_cond_signal(&condc);
    }
}

Proc * ReadyMLFQ::getProc(){
    Proc *procPtr=NULL;
    synchronized(readyMLFQMutex){
        for(int i=LEVEL-1;i>=0;i--) {
            //Wait when the ready queue is empty. It will block and wait
for signal.
            while(totalSize()==0)             pthread_cond_wait(&condc,
&readyMLFQMutex);
            if(!readyMLFQ[i]->empty()){
                procPtr=readyMLFQ[i]->front();
                readyMLFQ[i]->pop();
                break;
            }
        }
    }
    return procPtr;
}


int ReadyMLFQ::totalSize(){
    int sum=0;
    for(int i=0;i<LEVEL;i++){
        sum+=readyMLFQ[i]->size();
    }
    return sum;
}

ReadyMLFQ::~ReadyMLFQ(){
    pthread_mutex_destroy(&readyMLFQMutex);
    pthread_cond_destroy(&condc);
    for(int i=0;i<LEVEL;i++){
        delete(readyMLFQ[i]);
    }
}

void ReadyMLFQ::priorityBoost(){
    for (int i=LEVEL-2; i>=0; i--) {
        while(!readyMLFQ[i]->empty()) {
            Proc *ptrProc=readyMLFQ[i]->front();
            ptrProc->changePriority(LEVEL-1-i);
            readyMLFQ[LEVEL-1]->push(ptrProc);
            readyMLFQ[i]->pop();
        }
    }
}
```

Comments:
This is the declaration and definition of class "ReadyMLFQ". Standard template library that C++ provides is used to accomplish the date structure and operation. This class stands for the ready queue that contains all processes whose status is "ready". Class "ReadyMLFQ" provides two public functions for short-term scheduler and slave processor to call.

One is "getProc()", which is called by short-term scheduler to take one process out of the "ReadyMLFQ". The function checks the queue of each level from the highest to the lowest, finds the first nonempty queue and takes the process in the front out of queue. If there is nothing in the "ReadyMLFQ", the function returns NULL, or it returns a pointer points to a process.

Another function is "putProc()", which is called when a process runs out of time and swaps out. The function puts the pointer in the back of queue.

There are also two private functions called by two public functions. One is called "totalSize()", which returns the total number of processes by adding the number of processes in every queue together. Another one is called "priorityBoost()", which sets all processes' priority the highest to **prevent starvation** when "putProc()" is executed too many times.

Class "ReadyMLFQ" has a mutex to protect the critical section so that "putProc()" and "totalSize()" cannot both be accessed at the same time. What's more, it also provides a conditional variable. If there is nothing in the queue, the function "getProc()" will wait there until the "putProc()" wakes up it.

blockqueue.h/blockqueue.cpp:

```
#ifndef BLOCKQUEUE_H
#define BLOCKQUEUE_H
#include "proc.h"

class mycomparison
{
public:
  bool operator() (Proc*& lhs, Proc*& rhs)
  {
    return (lhs->getBlocTime() < rhs->getBlocTime());
  }
};

class BlockQueue{
private:
    pthread_mutex_t blockQueueMutex;
    std::priority_queue<Proc  *,  std::vector<Proc  *>,  mycomparison>
blockQueue;
public:
    BlockQueue();
    Proc* getProc();

    void putProc(Proc *process);
};
#endif
```

```
#include "blockqueue.h"

BlockQueue::BlockQueue(){
    pthread_mutex_init(&blockQueueMutex,NULL);
}

Proc* BlockQueue::getProc(){
    Proc* tmp=NULL;
    synchronized(blockQueueMutex){
    if(!blockQueue.empty()){
    tmp=blockQueue.top();
       blockQueue.pop();
    }
    }
    return tmp;
}

void BlockQueue::putProc(Proc *process){

     synchronized(blockQueueMutex){
    blockQueue.push(process);
     }
}
```

Comments:
This is the declaration and definition of class "BlockQueue". Standard template library that C++ provides is used to accomplish the date structure and operation. This class stands for the block queue which contains all processes that is I/O blocked. Class "BlockQueue" provides two public functions "getProc()" and "putProc()". The two functions are protected by a mutex so that they can't be accessed at the same. The data structure of block queue is **min-priority queue**. If the mid-term scheduler wants to pick up a process out of blockqueue, it will choose the process that has the smallest block time, that is, the top of the **heap** and pop it. Another function "putProc" is called when slave processors detect I/O block occurs and put the process into **heap**.

masterprocessor.h:

```
#ifndef MASTERPROCESSOR_H
#define MASTERPROCESSOR_H

#include "readyqueue.h"
#include "blockqueue.h"

class MasterProcessor {
```

```
public:
  MasterProcessor(ReadyMLFQ *rq0, BlockQueue *bq0, int proc_pip0[][2],
int idle_pip0[][2]);
   ~MasterProcessor() {
       for (std::list<Proc *>::iterator it=all_processes.begin(); it !=
all_processes.end(); it++) {
           delete *it;
       }
   }
   static void *runShortTermScheduler(void *self){
       ((MasterProcessor*)self)->shortTermScheduler();
       return NULL;
   }
   static void *runMidTermScheduler(void *self){
       ((MasterProcessor*)self)->midTermScheduler();
       return NULL;
   }
   static void *runLongTermScheduler(void *self){
       ((MasterProcessor*)self)->longTermScheduler();
       return NULL;
   }
protected:
   MasterProcessor(){}


private:
   void shortTermScheduler();
   void midTermScheduler();
   void longTermScheduler();
   pthread_t pt[3];
   ReadyMLFQ *rq;
   BlockQueue *bq;
   int (*proc_pip)[2];
   int (*idle_pip)[2];
   std::queue<int> IDSpace;
   std::list<Proc *> all_processes; //Store all the processes created
from long-term scheduler
};

#endif
```

Comments:

This code is the header of master processor, including the declaration of the class MasterProcessor. It also includes the declaration of the short-term, mid-term and long-term schedulers as one of the method in class MasterProcessor.

masterporcessor.cpp:

```
#include "masterprocessor.h"

/*This file consists of the master processor's short-, mid- and long-term
 *schedulers' definition.
 */

/* ---------------------- MASTER PROCESSOR------------------------- */

MasterProcessor::MasterProcessor(ReadyMLFQ *rq0, BlockQueue *bq0, int
proc_pip0[][2], int idle_pip0[][2]){
    rq = rq0; //Initialized the pointer of ready queue
    bq = bq0; //Initialized the pointer of block queue
    proc_pip = proc_pip0;   //initialized  the  pointer  of  pipe  for
passing process
    idle_pip = idle_pip0;   //initialized  the  pointer  of  pipe  for
telling idle situation
    while(!IDSpace.empty()){
        IDSpace.pop();
    }
    for (int i=0; i<MAX_PROCESS_NUMBER; i++) IDSpace.push(i+1);

    if(pthread_create(&pt[0],      NULL,      &runShortTermScheduler,
(void*)this)) {  //Create short-term scheduler as a thread
        printf("Could not create shortTerm on MasterProcessor\n");
    }
   if(pthread_create(&pt[1], NULL, &runMidTermScheduler, (void*)this))
{  //Create mid-term scheduler as a thread
        printf("Could not create midTerm on MasterProcessor\n");
    }
    if(pthread_create(&pt[2],      NULL,      &runLongTermScheduler,
(void*)this)) {  //Create long-term scheduler as a thread
        printf("Could not create longTerm on MasterProcessor\n");
    }
}

void MasterProcessor::shortTermScheduler() {
    //This array is to store the wrap object from class ProcWrapper for
each slave processor
    ProcWrapper * pw[SLAVES_NUMBER];
    for (int i=0; i<SLAVES_NUMBER; i++) {
        //Setting non block for reading the idle_pipe
        if  (  fcntl(idle_pip[i][0],  F_SETFL,  O_NONBLOCK)  ==  -1)
printf("non block fail on slave %d\n",i);
        pw[i] = NULL;
    }

    while (1) {
        for (int i=0; i<SLAVES_NUMBER; i++) {
            bool isIdle = false;
            read(idle_pip[i][0],  &isIdle,  sizeof(bool));  //non-block
reading the idle_pipe
            if ( isIdle ) {  //If the slave is idle
                Proc *pro;
                pro = rq->getProc(); //Get a process from ready queue, will
block if there is no process in ready queue
                if (pw[i] != NULL) {
                    delete pw[i]; //In case of memory leak
                    pw[i] = NULL;
                }
                pw[i] = new ProcWrapper(pro, TIME_UNIT * (1 << (LEVEL -
pro->getPriority() + 1)) );
                printf("##          Short-Term-Scheduler          ##:\n
Process(PID=%d,Priority=%d)\n   Moved  out  from  ReadyQueue\n   with
timeQuanta=%d\n  Assigned  to  Processor(%d)\n",  pw[i]->pro->getID(),
pw[i]->pro->getPriority(), pw[i]->timeQuanta, i+1);
                write(proc_pip[i][1], &pw[i], sizeof(ProcWrapper *));
            }

        }
    }
}

void MasterProcessor::midTermScheduler() {
    while (1) {
            Proc* pro=bq->getProc();
            if(pro != NULL){
                pro->setState(PROC_RUN);
                printf("##  Mid-Term-Scheduler  ##:\n   Process(PID=%d)
IO-Block  ends\n  Moved  out  from  BlockQueue\n  Put  in  ReadyQueue\n",
pro->getID());
                rq->putProc(pro);
            }
        }
}
```

```
void MasterProcessor::longTermScheduler() {
    while (1) {
        for    (list<Proc    *>::iterator    it=all_processes.begin();
it!=all_processes.end();) {
            if ( !(*it)->isRunning() ) {
                printf("## Long-Term-Scheduler ##:\n  Process(PID=%d) is
deleted\n", (*it)->getID());
                IDSpace.push((*it)->getID());
                delete *it;
                it = all_processes.erase(it);
            }
            else {
                it++;
            }
        }

        if (IDSpace.empty()) {
            //continue;
            sleep(FOR_A_WHILE);
        }
        else {
            Proc * pro = new Proc(IDSpace.front());
            IDSpace.pop();
            all_processes.push_back(pro);
            printf("##  Long-Term-Scheduler  ##:\n   Process(PID=%d)  is
created\n  Put in ReadyQueue\n", pro->getID());
            rq->putProc(pro);
        }
    }
}
```

Comments:
This code is the body of class MasterProcessor, including the definition of the respective class. And this code actually includes the whole content of the short-term, mid-term and long-term schedulers.

Job of short-term scheduler:
Getting process from ready queue, and give the process to an idle space processor with certain time quanta.

Job of mid-term scheduler:
Checking the Block queue to see if there is any end of IO-Block of a process. If so, get the process out of block queue and put it into ready queue.

Job of long-term scheduler:
Creating process and deleting process in certain speed.

slaveprocessor.h:

```
#ifndef SLAVEPROCESSOR_H
#define SLAVEPROCESSOR_H

#include "readyqueue.h"
#include "blockqueue.h"

class SlaveProcessor {
public:
    SlaveProcessor(ReadyMLFQ *rq0, BlockQueue *bq0, int *s_proc_pip0, 
int *s_idle_pip0, int slaveID0);
    static void *run(void *self){
        ((SlaveProcessor*)self)->running();
        return NULL;
    }
protected:
    SlaveProcessor(){}

private:
    void running();
    pthread_t pt;
    ReadyMLFQ *rq;
    BlockQueue *bq;
    char indent[SLAVE_INDENT_WIDTH*SLAVES_NUMBER];
    int *s_proc_pip;
    int *s_idle_pip;
    int slaveID;
};

#endif
```

Comments:
This code is the header of class SlaveProcessor, including the declaration of the respective class.

slaveprocessor.cpp:

```
#include "slaveprocessor.h"

SlaveProcessor::SlaveProcessor(ReadyMLFQ *rq0, BlockQueue *bq0, int 
*s_proc_pip0, int *s_idle_pip0, int slaveID0) {
    rq = rq0;
    bq = bq0;
    s_proc_pip = s_proc_pip0;
    s_idle_pip = s_idle_pip0;
    if(pthread_create(&pt, NULL, &run, (void*)this)) {
        printf("Could not create running thread on SlaveProcessor\n");
    }
    slaveID = slaveID0;
    int i;
    for (i=0; i<slaveID*SLAVE_INDENT_WIDTH; i++) indent[i] = ' ';
    indent[i] = '\0';
}

void SlaveProcessor::running() {
    while (1) {
        ProcWrapper *pw;
        bool const isIdle = true;
        write(s_idle_pip[1], &isIdle, sizeof(bool)); //Write back the idle
signal to idle_pipe
        read(s_proc_pip[0], &pw, sizeof(ProcWrapper *));    //read the
process_pipe, this will block if the pipe is not yet been filled things by
short-term-scheduler

        int proc_state = PROC_RUN;

        printf("%s(%d) Process(PID=%d) executing\n%s      Commands to be
run:%d\n",indent,       slaveID,       pw->pro->getID(),       indent,
pw->pro->restCommands());
        for (int i=0; i<pw->timeQuanta; i++) {
            proc_state = pw->pro->proc_execute();
            if (proc_state == PROC_BLOCK || proc_state == PROC_EXIT) break;
        }

        switch(proc_state) {
        case PROC_BLOCK: //Process IO Block
            printf("%s(%d)  Process(PID=%d)  IO-Block\n%s          Put  in
BlockQueue\n%s    Commands to be run:%d\n",indent, slaveID, pw->pro->getID(),
indent, indent, pw->pro->restCommands());
            pw->pro->setState(PROC_BLOCK);
            pw->pro->setBlocTime(1+rand()%100);
            bq->putProc(pw->pro);
            break;
        case PROC_EXIT://Process finish executing and exit
            printf("%s(%d)   Process(PID=%d)   exits\n",indent,   slaveID,
pw->pro->getID());
```

```
            pw->pro->setState(PROC_EXIT);
            break;
        case PROC_RUN://use up the time quanta but not finishes
        default:
            if ( pw->pro->getPriority() > 1 ) pw->pro->changePriority(-1);
            printf("%s(%d) Process(PID=%d) swapped out\n%s    Changed priority
to %d\n%s    Back to ReadyQueue\n%s    Commands to be run:%d\n",indent, slaveID,
pw->pro->getID(),    indent,    pw->pro->getPriority(),    indent,    indent,
pw->pro->restCommands());
            rq->putProc(pw->pro);
            break;
        }
    }
}
```

Comments:
This code includes the detail information of the definition of the class SlaveProcessor.

The main job of slave processor is to execute the process gained from short-term scheduler and execute the process in certain time quanta.

If the IO-Block happened, slave processor will put the process into block queue.

If process run out of time quanta, slave processor will degrade the priority of the process and put it into ready queue.

If process finished executing, slave process will set the status of process as PROC_EXIT.

8. Conclusion

This simulation provides very clear a structure of how Operating System schedules processes with a Multi-level Feedback Queue implementation and multiple processors; Inter-threads communications and data integrity are well achieved by a “faked” monitor mechanism, conditional variable and pipe facilities. The outcome of this simulation is of great value, and as an open source project, it can be shared and used for other purposes, such as teaching materials for college.