MP3: CPU Scheduling

Cover page

- 徐竣霆： Trace code, Implementation, Report.

- 江承紘：Trace code, Implementation, Report.

Trace Code

1-1.New->Ready

Kernel::ExecAll()

Iterate through all the execfile, Exec() them. And set the current thread to Finished(Sleep())(Parent finished after all the children finished).

Kernel::Exec(char *)

Create a new thread and allocate Address Space for the thread. Call Fork().

Thread::Fork(VoidFunctionPtr, void*)

Fork() allowes caller(parent) and callee(child) to execute concurrently. Call StackAllocate() to allocate stack for child. Then call ReadyToRun()(Should disable interrupt before, and enable interrupt afterward).

Thread::StackAllocate(VoidFunctionPtr, void*)

Use AllocBoundedArray()(in lib/sysdep.cc) to create a continuous space as stack. This is a x86 system

```
#ifdef x86
    // the x86 passes the return address on the stack.  In order for SWITCH()
    // to go to ThreadRoot when we switch to this thread, the return addres
    // used in SWITCH() must be the starting address of ThreadRoot.
    stackTop = stack + StackSize - 4;    // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif
```

Since the stack address is from high to low, set the stackTop to the highest(the start space, also where ThreadRoot(implemented in switch.S) locates). Then set stack as 0xdedbeef, to detect stack overflows.

```
#else
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
```

Then do this, setting all the machineState.(for Context Switch).(defined in switch.h)

Scheduler::ReadyToRun(Thread*)

Set the status of this Thread to Ready.
And Append this thread to readyList(FIFO, so it's actually a queue).

1-2.Running->Ready

Machine::Run()

Notice that this routine is re-entrant, so that it can run concurrently. (Detail was explained in MP1).

Interrupt::OneTick()

Two things can call OneTick():

1. Interrupts are re-enabled
2. A user instruction is executed

First advance the simulation time(based on current status(system/user)). Then call CheckIfDue() to check for pending interrupts. (turn off/ turn on interrupts before/after)
Check yieldOnReturn(for context switch). If true, call Yield() to yield the CPU from current Thread to others.

Thread::Yield()

Relinquish the CPU, call FindNextToRun() to find the next Thread to run.
Call ReadyToRun() to put the current Thread back to the readyList.
Call Run() to run the next Thread.

Schedulre::FindNextToRun()

If readyList is empty, return NULL directly.
Otherwise return the front of readyList, and pop_front().

Scheduler::ReadyToRun(Thread*)

As 1-1, set current Thread as ready, and push_back() to put it back to the readyList.

Scheduler::Run(Thread*, bool)

Dispatch the CPU to nextThread, check if currentThread(oldThread) is finished. If so, mark currentThread toBeDestroyed. Otherwise, save the state of the oldThread.
Switch the currentThread to nextThread and set the state as RUNNING.
Call SWITCH()(the context switch routine, details explained in 1-6 Switch(Thread*, Thread*)).

Call CheckToBeDestroyed() to delete the marked FINISHED thread.
Restore the state of oldThread(if needed).

1-3.Running->Waiting

SynchConsoleOutput::PutChar(char)

Make sure there is only one thread using I/O device at a time.(lock->Acquire())
Write a character to the console display.
Do synchronization(call P()).
Release the lock(lock->Release()).

Semaphore::P()(in synch.cc (http://synch.cc))

To handle synchronization.
While value == 0, means semaphore is not availavble, after append the currentThread in the waiting queue, put the currentThread to sleep. Otherwise, value–(semaphore is available).

List::Append(T)(in list.cc (http://list.cc))

Check whether the item is in the list already, if not, append it to the end of the list.

Thread::Sleep(bool)

Relinquish the CPU, since currentThread is blocked waiting on semaphore()(in this path).

Set the status of currentThread as BLOCKED.
If there are no threads in ready queue to run, call kernel->interrupt->Idle() to idle the CPU until next I/O interrupt to turn threads from waiting to ready.
Then call Run() to run the nextThread, it will do context switch. When it's time to switch back, it will restore the state and awake the sleeping thread.

Scheduler::FindNextToRun()

explained in 1-2.

Scheduler::Run(Thread*, bool)

explained in 1-2.

1-4.Waiting->Ready(Note: only need to consider console output as an example)

Semaphore::V()(in synch.cc (http://synch.cc))

First, call OneTick()(in Run()). Then, call OneTick(). In OneTick(), call checkIfDue() to check for pending interrupts.
If so, pull the front off the interrupt list and call the interrupt handler(next->callOnInterrupt->CallBack()). The CallBack() defined in callback.h, while the CallBackObj in this path is ConsoleOutput. So call ConsoleOutput::CallBack(). This will further call SynchConsoleInput::CallBack()(callWhenDone). And this will do waitFor()->V().

```
void
Semaphore::V()
{
    DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->stats->totalTicks);
    Interrupt *interrupt = kernel->interrupt;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    if (!queue->IsEmpty()) {  // make thread ready.
        kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }
    value++;

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

The Semaphore::V() cooperate with Semaphore P() to deal with the synchronization problem.
It will first set the front of Semaphore as Ready, put it back to readyList(ReadyToRun()).
Unlike P(), V() will do value++ here(kind of like wait() and signal() in Chapter 6).

Scheduler::ReadyToRun(Thread*)

As explained above.

1-5.Running->Terminated(Note: start from the Exit system call is called)

ExceptionHandler(ExceptionType) case SC_Exit

```
case SC_Exit:
    DEBUG(dbgAddr, "Program exit\n");
    val=kernel->machine->ReadRegister(4);
    cout << "return value:" << val << endl;
    kernel->currentThread->Finish();
    break;
```

will do kernel->currentThread->Finish() to finish currentThread().

Thread::Finish()

Call Sleep()(After IntOff). Notice that we cannot deallocate the current thread immediately while we are still running on this thread. Instead, we have to tell the scheduler to call the destructor, once we are not running this thread(in other thread).

Thread::Sleep(bool)

Explained in 1-3. Put the currentThread to sleep and find nextThread to run.

Scheduler::FindNextToRun()

Explained in 1-2.

Scheduler::Run(Thread*, bool)

Explained in 1-2.
The marked FINISHED thread will be delete in this case.

1-6.Ready->Running

Scheduler::FindNextToRun()

Explained in 1-2.

Scheduler::Run(Thread*, bool)

Explained in 1-2.
Set the state of nextThread from ready to running, then do SWITCH().

SWITCH(Thread*, Thread*)

(implemented in switch.S)

```
#ifdef x86

        .text
        .align  2

        .globl  ThreadRoot
        .globl  _ThreadRoot

/* void ThreadRoot( void )
**
** expects the following registers to be initialized:
**      eax     points to startup function (interrupt enable)
**      edx     contains inital argument to thread function
**      esi     points to thread function
**      edi     point to Thread::Finish()
*/
```

```
_ThreadRoot:
ThreadRoot:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   InitialArg
        call    *StartupPC
        call    *InitialPC
        call    *WhenDonePC

        # NOT REACHED
        movl    %ebp,%esp
        popl    %ebp
        ret



/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
*/
        .comm   _eax_save,4

        .globl  SWITCH
        .globl  _SWITCH
_SWITCH:
SWITCH:
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into eax
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack
        movl    %ebx,_PC(%eax)          # save it into the pc storage

        movl    8(%esp),%eax            # move pointer to t2 into eax
```

```
        movl    _EAX(%eax),%ebx         # get new value for eax into eb
        movl    %ebx,_eax_save          # save it
        movl    _EBX(%eax),%ebx         # retore old registers
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp         # restore stack pointer
        movl    _PC(%eax),%eax          # restore return address into e
        movl    %eax,4(%esp)            # copy over the ret address on
        movl    _eax_save,%eax

        ret

#endif // x86
```

That's where all the definition located.

```
#ifdef x86

/* the offsets of the registers from the beginning of the thread object */
#define _ESP     0
#define _EAX     4
#define _EBX     8
#define _ECX     12
#define _EDX     16
#define _EBP     20
#define _ESI     24
#define _EDI     28
#define _PC      32

/* These definitions are used in Thread::AllocateStack(). */
#define PCState         (_PC/4-1)
#define FPState         (_EBP/4-1)
#define InitialPCState  (_ESI/4-1)
#define InitialArgState (_EDX/4-1)
#define WhenDonePCState (_EDI/4-1)
#define StartupPCState  (_ECX/4-1)

#define InitialPC       %esi
#define InitialArg      %edx
#define WhenDonePC      %edi
#define StartupPC       %ecx

#endif // x86
```

In ThreadRoot:
Will call the initial functions defined in switch.h(by setting registers).

In SWITCH:

```
movl    %eax,_eax_save          # save the value of eax
movl    4(%esp),%eax            # move pointer to t1 into eax
movl    %ebx,_EBX(%eax)         # save registers
movl    %ecx,_ECX(%eax)
movl    %edx,_EDX(%eax)
movl    %esi,_ESI(%eax)
movl    %edi,_EDI(%eax)
movl    %ebp,_EBP(%eax)
movl    %esp,_ESP(%eax)         # save stack pointer
movl    _eax_save,%ebx          # get the saved value of eax
movl    %ebx,_EAX(%eax)         # store it
```

Save the current eax on the stack so that we can use it as a pointer to t1. Then saves the values of the other registers(from t1) to the stack, as well as the stack pointer and the saved value of eax.

```
movl    0(%esp),%ebx            # get return address from stack into ebx
movl    %ebx,_PC(%eax)          # save it into the pc storage

movl    8(%esp),%eax            # move pointer to t2 into eax

movl    _EAX(%eax),%ebx         # get new value for eax into ebx
movl    %ebx,_eax_save          # save it
movl    _EBX(%eax),%ebx         # retore old registers
```

Get the return address from stack and save it in to the pc storage(t1), then move the pointer to t2 into eax, restore the saved value of eax into eax.

```
movl    _EBX(%eax),%ebx         # retore old registers
movl    _ECX(%eax),%ecx
movl    _EDX(%eax),%edx
movl    _ESI(%eax),%esi
movl    _EDI(%eax),%edi
movl    _EBP(%eax),%ebp
movl    _ESP(%eax),%esp         # restore stack pointer
```

Restore the value of old registers and the stack pointer(t2).

```
movl    _PC(%eax),%eax          # restore return address into eax
movl    %eax,4(%esp)            # copy over the ret address on the stack
movl    _eax_save,%eax

ret
```
Restore the return address and copy it over the return address on stack. Finally, return from function, finish the thread switch, start executing t2.

This complete the context switch.

(depends on the previous process state, e.g.,[New,Running,Waiting]→Ready)

New -> Ready: 1-1
Running -> Ready: 1-2
Waiting -> Ready: 1-4

For 1-2(Running -> Ready) and 1-4(Waiting -> Ready), they turn to Ready because of Context Switch, so while they Ready -> Running, we have to restore their value, register and PC... and continue from previous place. For 1-1(New -> Ready), when Ready -> Running, just simply start from scratch.

for loop in Machine::Run()

Explained in MP1.
Executing the program.

Implementation

`code/lib/debug.h`

By the requirement 2-3 in spec, we add a debugging flag **z**.

```
const char dbgAll = '+';        // turn on all debug messages
const char dbgThread = 't';     // threads
const char dbgSynch = 's';      // locks, semaphores, condition vars
const char dbgInt = 'i';        // interrupt emulation
const char dbgMach = 'm';       // machine emulation
const char dbgDisk = 'd';       // disk emulation
const char dbgFile = 'f';       // file system
const char dbgAddr = 'a';       // address spaces
const char dbgNet = 'n';        // network emulation
const char dbgSys = 'u';                // systemcall
const char dbgTraCode = 'c';
const char dbgMP3 = 'z';        // MP3 add
```

code/threads/thread.h

1. Adding `aging(int)` function to be implemented in `thread.cc` .

2. Adding five variables and its `get` and `set` functions to record:

    - burst time
    - accumulated execution time
    - priority of the thread
    - time when the thread enters the ready state
    - time when the thread enters the running state

```
// MP3 Add
private:
    double burstTime;
    double accumulatedExecutionTime;
    int execPriority;
    int enterReadyTime;
    int enterRunningTime;
public:
    void    aging(int currentTime);
    void    setBurstTime(double t)              { burstTime = t; }
    double  getBurstTime()                      { return burstTime; }
    void    setAccumulatedExecutionTime(int t)  { accumulatedExecutionTime = t; }
    double  getAccumulatedExecutionTime()       { return accumulatedExecutionTime; }
    void    setPriority(int priority)           { execPriority = priority; }
    int     getPriority()                       { return execPriority; }
    void    setReadyTick(int t)                 { enterReadyTime = t;}
    int     getReadyTick()                      { return enterReadyTime; }
    void    setRunTick(int t)                   { enterRunningTime = t; }
    int     getRunTick()                        { return enterRunningTime; }
```

code/threads/thread.cc

1. `Thread::Thread()`
   Initialize the five variables we have created.

```
    burstTime = accumulatedExecutionTime = 0;                   // MP3 add
    execPriority = enterReadyTime = enterRunningTime = 0;       // MP3 add
```

2. `Thread::Yield()`
   Calculate the elapsed time between the current time and when the thread enters the running state, then add the elapsed time to the accumulated

execution time.

```
if (nextThread != NULL) {
    int elapsedTicks = kernel -> stats -> totalTicks - kernel -> currentThread -> getRunTick();            // MP3 add
    int totalAccumulatedTime = kernel -> currentThread -> getAccumulatedExecutionTime() + elapsedTicks; // MP3 add
    kernel -> currentThread -> setAccumulatedExecutionTime(totalAccumulatedTime);                        // MP3 add
    DEBUG(dbgMP3 , "[E] Tick " << kernel->stats->totalTicks <<                                           // MP3 add
        "Thread " << nextThread->getID() << "is now selected for execution, thread " <<                  // MP3 add
        kernel->currentThread->ID << " is replaced, and it has executed" <<                              // MP3 add
        kernel->currentThread->getAccumulatedExecutionTime() << "ticks \n" );                            // MP3 add
kernel->scheduler->ReadyToRun(this);
kernel->scheduler->Run(nextThread, FALSE);
}
```

3. `Thread::Sleep()`

   Update the accumulated execution time like that in `Thread::Yield()` , then calculate the approximated burst time given by spec and set the burst time to the current thread.

```
int elapsedTicks = kernel -> stats -> totalTicks - kernel -> currentThread -> getRunTick();            // MP3 add
int totalAccumulatedTime = kernel -> currentThread -> getAccumulatedExecutionTime() + elapsedTicks; // MP3 add
kernel -> currentThread -> setAccumulatedExecutionTime(totalAccumulatedTime);                        // MP3 add
DEBUG(dbgMP3 , "[D] Tick [" << kernel -> stats -> totalTicks <<                                      // MP3 add
    "]: Thread [" << kernel -> currentThread -> getID() <<                                            // MP3 add
    "] update approximate burst time, from: ["<< kernel -> currentThread -> getBurstTime() <<         // MP3 add
    "], add [" << kernel -> currentThread -> getAccumulatedExecutionTime() <<  "], to [" <<           // MP3 add
    double(kernel -> currentThread -> getBurstTime() * 0.5) +                                         // MP3 add
    double(kernel -> currentThread -> getAccumulatedExecutionTime() * 0.5) << "] \n");                // MP3 add
double newBurstTime = 0.5 * kernel -> currentThread -> getBurstTime() +                               // MP3 add
                      0.5 * kernel -> currentThread -> getAccumulatedExecutionTime();                 // MP3 add
kernel -> currentThread -> setBurstTime(newBurstTime);                                                // MP3 add
kernel -> currentThread -> setAccumulatedExecutionTime(0);                                            // MP3 add
```

4. `Thread::aging()`

   Calculate the waiting time and update the priority while waiting for more than 1500 ticks.

```
void Thread::aging(int currentTime) {   // MP3 add
    int prevPriority = execPriority;
    int newPriority = execPriority;
    int waitingTime = currentTime - enterReadyTime;

    while(waitingTime > 1500) {
        newPriority += 10;
        waitingTime -= 1500;
        enterReadyTime += 1500;
    }
    newPriority = min(newPriority, 149);
    if (prevPriority != newPriority)
        DEBUG(dbgMP3 , "[C] Tick [" << currentTime << "]: Thread [" << ID <<
            "] changes its priority from [" << prevPriority << "] to [" << newPriority << "]\n");
    execPriority = newPriority;
}
```

`code/threads/kernel.h`

1. Modify the Exec function so that we can also send the thread priority.

```
// int Exec(char* name);                      // MP3 Delete
int Exec(char *name, int priorityNumber); // MP3 Add
```

2. Add `threadPriorityNumber` since each thread has its priority.

```
private:

Thread* t[10];
char*   execfile[10];
int threadPriorityNumber[10]; // MP3 Add
int execfileNum;
```

code/threads/kernel.cc

1. Add the flag `-ep` like the flag `-e`, and initialize each `threadPriorityNumber`.

```
        } else if (strcmp(argv[i], "-ep") == 0) { // MP3 add
            execfile[++execfileNum] = argv[++i];
            threadPriorityNumber[execfileNum] = atoi(argv[++i]);
            cout << execfile[execfileNum] << " has the priority of " << threadPriorityNumber[execfileNum] << "\n";
            // check whether the priority is valid
            ASSERT(0 <= threadPriorityNumber[execfileNum] && threadPriorityNumber[execfileNum] <= 149);
        }
```

2. Add `t[threadNum] -> setPriority(priorityNumber)` to each thread, and delete the original `Exec` function since it won't be used anymore.

```
int Kernel::Exec(char *name, int priorityNumber) { // MP3 Add
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum] -> setPriority(priorityNumber);
    t[threadNum] -> space = new AddrSpace();
    t[threadNum] -> Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);

    ++threadNum;

    return threadNum - 1;
}

// int Kernel::Exec(char* name) // MP3 Delete
// {
//  t[threadNum] = new Thread(name, threadNum);
//  t[threadNum]->space = new AddrSpace();
//  t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
//  threadNum++;

//  return threadNum-1;
```

3. Modify the calling of `Exec()` in `ExecAll()`.

```
void Kernel::ExecAll()
{
    for (int i=1;i<=execfileNum;i++) {
        int a = Exec(execfile[i], threadPriorityNumber[i]); // MP3 Modify
    }
    currentThread->Finish();
    //Kernel::Exec();
}
```

code/threads/scheduler.h

1. Use `SortedList` to create `L1` and `L2`, and just use a simple list for `L3` since `L1` and `L2` have their own sorting rule. Remove the original `readyList` at the same time.

```
private:
/* MP3 delete
  List<Thread *> *readyList;  // queue of threads that are ready to run, but not running
*/
  SortedList<Thread *> *L1, *L2; // MP3 add
  List<Thread *> *L3;            // MP3 add
  Thread *toBeDestroyed;  // finishing thread to be destroyed
                // by the next thread that runs
```

2. Add three functions: `UpdatePriority()`, `Scheduling()`, `SortUpdatedList()` to be implemented in `scheduler.cc`, and add query functions of the three lists for `alarm.cc` to call. The detail will be explained in `alarm.cc`.

```
    void UpdatePriority();  // MP3 add
    void Scheduling();      // MP3 add
    void SortUpdatedList(); // MP3 add

    bool isL1Empty()  { return L1 -> IsEmpty(); } // MP3 add
    bool isL2Empty()  { return L2 -> IsEmpty(); } // MP3 add
    bool isL3Empty()  { return L3 -> IsEmpty(); } // MP3 add
    Thread* L1Front() { return L1 -> Front(); }   // MP3 add
    Thread* L2Front() { return L2 -> Front(); }   // MP3 add
    Thread* L3Front() { return L3 -> Front(); }   // MP3 add
```

code/threads/scheduler.cc

1. Comparison function for `L1` . Calculate the remaining burst time and return the priority.

```
// MP3 add
int PreemptiveSJF(Thread* First, Thread* Second) {
    double remain_burst_time1 = (First -> getBurstTime()) - (First -> getAccumulatedExecutionTime());
    double remain_burst_time2 = (Second -> getBurstTime()) - (Second -> getAccumulatedExecutionTime());
    if (remain_burst_time1 < remain_burst_time2)
        return -1;
    else if (remain_burst_time1 == remain_burst_time2)
        return 0;
    else
        return 1;
}
```

2. Comparison function for `L2` . Just compare its priority.

```
// MP3 add
int NonPreemptive(Thread *First, Thread *Second) {
    if (First -> getPriority() > Second -> getPriority())
        return -1;
    else if (First -> getPriority() == Second -> getPriority())
        return 0;
    else
        return 1;
}
```

3. Modify the constructor

```
Scheduler::Scheduler()
{
    // readyList = new List<Thread *>;                  // MP3 delete
    L1 = new SortedList<Thread *>(PreemptiveSJF);    // MP3 add
    L2 = new SortedList<Thread *>(NonPreemptive);    // MP3 add
    L3 = new List<Thread *>;                         // MP3 add
    toBeDestroyed = NULL;
}
```

4. Modify the destructor

```
Scheduler::~Scheduler()
{
    // delete readyList;  // MP3 delete
    delete L1, L2, L3;    // MP3 add
}
```

5. `UpdatePriority` : If the list is not empty, we then iterate all its threads and call aging.

```
// MP3 add
void Scheduler::UpdatePriority() {
    int ticks = kernel -> stats -> totalTicks;

    if (!isL1Empty()) {
        for (ListIterator<Thread *> *iter = new ListIterator<Thread* >(L1); !iter -> IsDone(); iter -> Next())
            iter -> Item() -> aging(ticks);
    }

    if (!isL2Empty()) {
        for (ListIterator<Thread *> *iter = new ListIterator<Thread* >(L2); !iter -> IsDone(); iter -> Next())
            iter -> Item() -> aging(ticks);
    }

    if (!isL3Empty()) {
        for (ListIterator<Thread *> *iter = new ListIterator<Thread* >(L3); !iter -> IsDone(); iter -> Next())
            iter -> Item() -> aging(ticks);
    }

    return;
}
```

6. `Scheduling` : For each thread in `L2` , if its priority is greater than 99, then move it to `L1` . For each thread in `L3` , if its priority is greater than 99 or 49, then move it to `L1` or `L2` , respectively. Once a thread is removed from or insert to a list, print a debug message.

```
// MP3 add
void Scheduler::Scheduling() {
    ListIterator<Thread *> *iter;
    Thread* currentThread;
    if (!isL2Empty()) {
        for (ListIterator<Thread *> *iter = new ListIterator<Thread *>(L2); !iter -> IsDone(); iter -> Next()) {
            currentThread = iter -> Item();
            int threadPriority = iter -> Item() -> getPriority();
            if (threadPriority > 99) { // put currentThread from L2 to L1
                L2 -> Remove(currentThread);
                DEBUG(dbgMP3, "[B] Tick[" << kernel -> stats -> totalTicks << "]: Thread [" << currentThread -> getID() <<
                    "] is removed from queue L[2]\n");
                L1 -> Insert(currentThread);
                DEBUG(dbgMP3, "[A] Tick[" << kernel -> stats -> totalTicks << "]: Thread [" << currentThread -> getID() <<
                    "] is inserted into queue L[1]\n");
            }
        }
    }
    if (!isL3Empty()) {
        for (ListIterator<Thread *> *iter = new ListIterator<Thread *>(L2); !iter -> IsDone(); iter -> Next()) {
            currentThread = iter -> Item();
            int threadPriority = iter -> Item() -> getPriority();
            if (threadPriority > 99) { // put currentThread from L3 to L1
                L3 -> Remove(currentThread);
                DEBUG(dbgMP3, "[B] Tick[" << kernel -> stats -> totalTicks << "]: Thread [" << currentThread -> getID() <<
                    "] is removed from queue L[3]\n");
                L1 -> Insert(currentThread);
                DEBUG(dbgMP3, "[A] Tick[" << kernel -> stats -> totalTicks << "]: Thread [" << currentThread -> getID() <<
                    "] is inserted into queue L[1]\n");
            } else if (threadPriority > 49) { // put currentThread from L3 to L2
                L3 -> Remove(currentThread);
                DEBUG(dbgMP3, "[B] Tick[" << kernel -> stats -> totalTicks << "]: Thread [" << currentThread -> getID() <<
                    "] is removed from queue L[3]\n");
                L2 -> Insert(currentThread);
                DEBUG(dbgMP3, "[A] Tick[" << kernel -> stats -> totalTicks << "]: Thread [" << currentThread -> getID() <<
                    "] is inserted into queue L[2]\n");
            }
        }
    }
}
```

7. Before scheduling the next thread onto the CPU, we call `SortUpdatedList` to ensure the priority in `L1` and `L2` are well sorted.

```
// MP3 add
void Scheduler::SortUpdatedList() {
    SortedList<Thread *> *tmpL1 = new SortedList<Thread *>(PreemptiveSJF);
    SortedList<Thread *> *tmpL2 = new SortedList<Thread *>(NonPreemptive);
    if (!isL1Empty()) {
        for (ListIterator<Thread *> *iter = new ListIterator<Thread *>(L1); !iter -> IsDone(); iter -> Next())
            tmpL1 -> Insert(iter -> Item());
        delete L1;
        L1 = tmpL1;
    }

    if (!isL2Empty()) {
        for (ListIterator<Thread *> *iter = new ListIterator<Thread *>(L2); !iter -> IsDone(); iter -> Next())
            tmpL2 -> Insert(iter -> Item());
        delete L2;
        L2 = tmpL2;
    }
    return;
}
```

8. Modify `ReadyToRun`. Put the thread into the corresponding Ready queue.

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    // thread->setStatus(READY);    // MP3 delete
    // readyList->Append(thread);   // MP3 delete
    // MP3 add
    thread -> setReadyTick(kernel -> stats -> totalTicks);
    int threadPriority = thread -> getPriority();
    if (threadPriority < 50) {
        L3 -> Append(thread);
        DEBUG(dbgMP3, "[A] Tick [" << kernel -> stats -> totalTicks << "]: Thread [" << thread -> getID() <<
            "] is inserted into queue L[3]\n");
    } else if (threadPriority < 100) {
        L2 -> Insert(thread);
        DEBUG(dbgMP3, "[A] Tick [" << kernel -> stats -> totalTicks << "]: Thread [" << thread -> getID() <<
            "] is inserted into queue L[2]\n");
    } else {
        L1 -> Insert(thread);
        DEBUG(dbgMP3, "[A] Tick [" << kernel -> stats -> totalTicks << "]: Thread [" << thread -> getID() <<
            "] is inserted into queue L[1]\n");
    }
    return;
}
```

9. Calling `SortUpdateList()` in the begining. Check `L1`, `L2`, and `L3` to see if there are threads. Since we have sort the lists, the front thread of the lists has the higest priority. Once a thread is found, schedule it onto the CPU and remove

it from the list. If no threads are found, return `NULL` .

```
Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    // MP3 delete
    /*if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }*/
    // MP3 add
    SortUpdatedList();
    Thread* nextThread = NULL;
    if (!isL1Empty()) {
        nextThread = L1 -> RemoveFront();
        nextThread -> setRunTick(kernel -> stats -> totalTicks);
        DEBUG(dbgMP3, "[B] Tick [" << kernel -> stats -> totalTicks << "]: Thread [" << nextThread -> getID() <<
            "] is removed from queue L[1]\n");
    } else if (!isL2Empty()) {
        nextThread = L2 -> RemoveFront();
        nextThread -> setRunTick(kernel -> stats -> totalTicks);
        DEBUG(dbgMP3, "[B] Tick [" << kernel -> stats -> totalTicks << "]: Thread [" << nextThread -> getID() <<
            "] is removed from queue L[2]\n");
    } else if (!isL3Empty()) {
        nextThread = L3 -> RemoveFront();
        nextThread -> setRunTick(kernel -> stats -> totalTicks);
        DEBUG(dbgMP3, "[B] Tick [" << kernel -> stats -> totalTicks << "]: Thread [" << nextThread -> getID() <<
            "] is removed from queue L[3]\n");
    }
    return nextThread;
}
```

10. In `Run()` , once the old thread is switched back, we print the debug message and update its `enterRunningTime` .

```
    SWITCH(oldThread, nextThread);

    DEBUG(dbgMP3 , "[E] Tick [" << kernel -> stats -> totalTicks <<
        "]: Thread ["<< nextThread -> getID() << "] is now selected for execution, thread [" <<
        oldThread -> getID() << "] is replaced, and it has executed [" <<
        kernel -> stats -> totalTicks - oldThread -> getRunTick() << "] ticks \n");

    // we're back, running oldThread
    oldThread -> setRunTick(kernel -> stats -> totalTicks);
```

`code/threads/alarm.cc`

In the begining, we update the lists by calling `UpdatePriority()` , `Scheduling()` , and `SortUpdatedList()` . Then we add the following rules to the `Callback()` function to decide whether the current thread should be preempted.

- The priority of the current thread is greater than 99, and the remain burst time of the front thread of `L1` is smaller than that of the current thread.
- The priority of the current thread is greater than 49 but less than 100, and `L1` has thread.
- The priority of the current thread is less than 50 and there exists a thread in `L1` , `L2` , or `L3` .

Since we can not access the private data `L1` , `L2` , and `L3` of the `scheduler` class, we implement the public functions in `scheduler.h` .

```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();

    if (status != IdleMode) {
    // interrupt->YieldOnReturn(); // MP3 delete
        kernel -> scheduler -> UpdatePriority();
        kernel -> scheduler -> Scheduling();
        kernel -> scheduler -> SortUpdatedList();
        if (kernel -> currentThread -> getPriority() > 99) {
            if (!kernel -> scheduler -> isL1Empty()) {
                Thread* L1Front = kernel -> scheduler -> L1Front();
                Thread* currentThread = kernel -> currentThread;
                double remainBurstTime1 = currentThread -> getBurstTime() - currentThread -> getAccumulatedExecutionTime();
                double remainBurstTime2 = L1Front -> getBurstTime() - L1Front -> getAccumulatedExecutionTime();
                if (remainBurstTime1 > remainBurstTime2)
                    interrupt -> YieldOnReturn();
            }
        } else if (kernel -> currentThread -> getPriority() > 49) {
            if (!kernel -> scheduler -> isL1Empty())
                interrupt -> YieldOnReturn();
        } else {
            if ((!kernel -> scheduler -> isL1Empty())
             || (!kernel -> scheduler -> isL2Empty())
             || (!kernel -> scheduler -> isL3Empty()))
                interrupt -> YieldOnReturn();
        }
    }
}
```