CS342301: Operating System
MP3: CPU Scheduling

Team no. 46

Member Contribution:

陳敬中: Trace Code, Report, Implementation (50 %)

白宸安: Trace Code, Report, Implementation (50 %)

MP3

Part 1. Trace Code

1-1. New → Ready

- In this code path, a process or a thread is created and added to the ready queue.

`void Kernel::ExecAll()`

This function calls `Exec(char *)` for each executable files to be executed.

`int Kernel::Exec(char *name)`

Here, each executable file is allocated a thread for it to run on. The address space is also allocated here for the thread. Then, we fork on the thread by calling `Fork()` . We pass in the function pointer to `ForkExecute()` and a pointer to the thread as its argument so that the thread can be executed.

`void Thread::Fork(voidFunctionPtr func, void *arg)`

After `ForkeExecute()` and the thread are passed into `Fork()`, an execution stack is allocated and the thread is put into the ready queue of the scheduler. Note that we have to explicitly disable the interrupt before putting the thread to the ready queue.

`void Thread::StackAllocate(voidFunctionPtr func, void *arg)`

Here, the execution stack is allocated for the function `ForkExecute()`. It initializes the stack with an initial stack frame for ThreadRoot. This stack frame will enable interrupts as the thread begins, execute the thread by calling `ForkExecute()`, and then calls `Finish()` to end execution.

`void Scheduler::ReadyToRun(Thread *thread)`

This function puts the thread into the ready queue by:

1. set the status of the thread to “Ready”
2. Append the thread to the `readyList` of the scheduler

1-2. Running → Ready

- This code path illustrates that a running thread might yield its control of CPU and goes back to the ready queue,

`void Machine::Run()`

As a user thread is in running in the machine, after each instruction is executed, the Kernel calls `OneTick()`.

`void Interrupt::OneTick()`

After this is called, the kernel checks the timer device(`yieldOnReturn`). If it asks for a context switch, we are going to do it now. We then switch to kernel mode, so the kernel can make the current thread yield its control of the CPU by calling

`Yield()` .

`void Thread::Yield()`

Before yielding, we have to disable the interrupts first. Then, the kernel scheduler will find the next thread to run from the ready queue by calling `FindNextToRun()`. After that, the yielding thread is placed back to the ready queue by `ReadyToRun()`, and the scheduler calls `Run()` to prepare the next thread to run. The interrupts are re-enabled at last.

`Thread *Scheduler::FindNextToRun()`

This function returns the pointer to the thread at the front of the `readyList`. If there is none in it, it returns `NULL`.

`void Scheduler::ReadyToRun(Thread *thread)`

This function puts the thread into the ready queue by:

1. set the status of the thread to “Ready”
2. Append the thread to the `readyList` of the scheduler

`void Scheduler::Run(Thread *nextThread, bool finishing)`

After the scheduler finds the next thread to run, it calls this `Run()` to dispatch the CPU to the next thread. It first stores the state of the old thread, set the new thread to running state, and lastly performs context switch. After returning from the switch, it deals with the old thread according to the `finishing` value.

1-3. Running → Waiting

`void SynchConsoleOutput::PutChar(char ch)`

This function performs a synchronized console output that simple display a character `ch`. It uses a lock to ensure that there is always one writer at a time. After calling `consoleOutput->PutChar(ch)`, it uses a semaphore `waitFor->P()` to wait for the call back function. The call back function is called by the machine when the output is finished.

`void Semaphore::P()`

Semaphore is an important system data structure, so we need to ensure interrupt is disabled before modifying its value. If the value of the semaphore is greater than 0, it means the semaphore is still available, and `P()` simply decreases its value by one. Otherwise, if the value is zero, the semaphore is no longer available. In this case, the current thread has to wait for this semaphore to be available again, so `P()` calls `queue->Append(currentThread)` to add the current thread to the waiting queue of this semaphore. Other than that, `P()` also calls `currentThread->Sleep(False)`, so the current thread sleeps until the semaphore is available.

`void List<T>::Append(T item)`

List is a simple data structure implemented as a singly-linked list. The `waitFor` semaphore uses it as an FIFO queue. Append is a member function of List that adds the item to the end of the List. Therefore, `P()` simply just appends the current thread to the waiting queue of the semaphore.

`void Thread::Sleep(bool finishing)`

Here, the originally running thread is set to `blocked`, so it stops execution. Then, the scheduler calls `FindNextToRun()` and `Run()` to find the next thread for execution and perform context switch to it. Note that when `P()` calls `Sleep()`, the argument `Finishing = False`. Thus, `Run()` will not destroy it, so when the original thread is waken, it can continue its execution.

`Thread *Scheduler::FindNextToRun()`

This function returns the pointer to the thread at the front of the `readyList`. If there is none in it, it returns `NULL`.

`void Scheduler::Run(Thread *nextThread, bool finishing)`

After the scheduler finds the next thread to run, it calls this `Run()` to dispatch the CPU to the next thread. It first stores the state of the old thread, set the new thread to running state, and lastly performs context switch. After returning from the switch, it deals with the old thread according to the `finishing` value.

1-4. Waiting → Ready

`void Semaphore::V()`

After the console output is done successfully, the interrupt handler `void SynchConsoleOutput::CallBack()` is called. This call back function then calls `waitFor->V()`. Then, `waitFor->V()` wakes up the thread at the front of its queue by calling `ReadyToRun()`. In other words, the thread is put back into the ready queue. Finally, the value of the semaphore is increased by 1.

`void Scheduler::ReadyToRun(Thread *thread)`

This function puts the thread into the ready queue by:

1. set the status of the thread to “Ready”
2. Append the thread to the `readyList` of the scheduler

1-5. Running → Terminated

`void ExceptionHandler(ExceptionType which)`

After the Exit system call is made, the `ExceptionHandler()` identifies it and handles it with the case `SC_Exit`. Here, `kernel->currentThread->Finish()` is called to end the current thread.

`void Thread::Finish()`

In `Finish()`, we call `Sleep(TRUE)` to stop the thread’s execution and eliminate it. Notice that we have to disable the interrupts first and ensure the thread calling is the current thread running.

`void Thread::Sleep(bool finishing)`

Here, the originally running thread is set to `blocked`, so it is stops execution. Then, the scheduler calls `FindNextToRun()` and `Run()` to find the next thread for execution and perform context switch to it. Note that when `Finish()` calls `Sleep()`, the

argument `Finishing = TRUE`. Thus, `Run()` will destroy it.

`Thread *Scheduler::FindNextToRun()`

This function returns the pointer to the thread at the front of the `readyList`. If there is none in it, it returns `NULL`.

`void Scheduler::Run(Thread *nextThread, bool finishing)`

After the scheduler finds the next thread to run, it calls this `Run()` to dispatch the CPU to the next thread. It first stores the state of the old thread, set the new thread to running state, and lastly performs context switch. After returning from the switch, it deals with the old thread according to the `finishing` value. Here, since `finishing == TRUE`, we set the old thread as `toBeDestroyed`. Then, `CheckToBeDestroyed()` is called to delete the thread and the address space of it.

1-6. Ready → Running

`Thread *Scheduler::FindNextToRun()`

This function returns the pointer to the thread at the front of the `readyList`. If there is none in it, it returns `NULL`.

`void Scheduler::Run(Thread *nextThread, bool finishing)`

After the scheduler finds the next thread to run, it calls this `Run()` to dispatch the CPU to the next thread. It first stores the state of the old thread, set the new thread to running state, and lastly performs context switch. After returning from the switch, it deals with the old thread according to the `finishing` value.

`SWITCH(oldThread, newThread)`

The context switch is performed here. First, the original value of `%eax` is stored, because we need it to store the pointer to a thread. We then store the pointer to the old thread to `%eax` with the instruction `movl 4(%esp), %eax`. Since the thread pointers are stored in the execution stack of SWITCH, we can access it through `%esp`. Then, we store the general-purpose registers, stack pointer and the return address to the storage of the old thread. Then, we move the pointer to the new thread to `%eax`. Subsequently, we reloads the values in the storage of the new thread to the general-purpose registers, stack pointer and the program counter. The return address of the new thread is now on the Execution stack, and we can return to it with `ret`.

Returning from Switch

1. New → Ready

   When a new thread is created, it forks on itself, allocates the execution stack and is added to the ready list. After context switch, the new thread starts its execution on the stack starting from the threadRoot.

2. Running → Ready

   After context switch, the thread simply resumes its execution.

3. Waiting → Ready

   A waiting thread could be waiting for an I/O operation. After the I/O operation is done, the thread is waken up and put into the ready queue. After the context switch, it continues in `Run()` , and restores its state. Then, it goes back to the while loop in `P()`. Since now the value is greater than 0, it can exit the loop and keep running.

Part 2. Implementation

kernel.cc / kernel.h

We added an array called priority in kernel.h to store the initial values of threads' priority.

In the constructor of the Kernel, we added a code segment for command line argument of "-ep". It reads the executable file name and its priority.

```
class Kernel {
    private:
        int priority[10];
}
```

```
Kernel::Kernel(int argc, char **argv)
{
    // ...
        } else if (strcmp(argv[i], "-ep") == 0) {
            execfile[++execfileNum] = argv[++i];
            priority[execfileNum] = atoi(argv[++i]);
        }
```

```
    }
}
```

scheduler.h

In scheduler.h, we add two SortedList and one List to represent L1, L2, L3.

```
// *** Multi-level Queue
SortedList<Thread *> *L1, *L2;
List<Thread *> *L3;
```

and some helper function.

```
void priorityUpdate();

// *** Multi-level Queue
SortedList<Thread *> *getL1() {return L1;}
SortedList<Thread *> *getL2() {return L2;}
List<Thread *> *getL3() {return L3;}
// ***
```

When `Kernel::Exec()` is called to create a thread for an executable file, it also passes the priority to the thread.

```
int Kernel::Exec(char* name, int p)
{
    // ...
  // *** thread priority
    t[threadNum]->priority = p;
    // ***
    // ...
    return threadNum-1;
}
```

scheduler.cc

In scheduler.cc, we first implement two compare function for SortedList L1, L2.

```
int greaterCmp(Thread *a, Thread *b) {
      if (a->priority > b->priority) return -1;
      else if (a->priority < b->priority) return 1;
      else return 0;
};
int lessCmp(Thread *a, Thread *b) {
      if ((a->expected - a->burst) < (b->expected - b->burst)) return -1;
      else if ((a->expected - a->burst) > (b->expected - b->burst)) return 1;
      else return 0;
};
```

and then add them in the constructor and destructor of Scheduler class.

We modify the Scheduler::ReadyToRun. It puts the thread into corresponding list(L1, L2, L3) base on the priority of that thread.

```
// *** Choosing next thread from multi-level queue
int queueLevel = 0;
if (thread->priority < 50) {
    L3->Append(thread);
    queueLevel = 3;
} else if (thread->priority < 100) {
    L2->Insert(thread);
    queueLevel = 2;
} else {
    L1->Insert(thread);
    queueLevel = 1;
}
DEBUG(dbgFlag, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->get
// ***
```

Then, we implement a new function Scheduler::priorityUpdate to do the aging work. It first iterate through the list, add timerTicks to each thread’s timeWaited, which is the time that this thread has waited in ready queue. If the thread’s timeWaited ≥ 1500, then update its priority by 10, and reset the timeWaited.

If the thread was originally in L2 or L3, when priority update, we also remove it from origin list, and call ReadyToRun(i2->Item()) or ReadyToRun(i3->Item()), to automatically update its corresponding list by using ReadyToRun.

```
// 更新priority
void Scheduler::priorityUpdate(){
    Statistics *stats = kernel->stats;

    //L1

    ListIterator<Thread *> *i1 = new ListIterator<Thread *>(L1);
    for(; !i1->IsDone(); i1->Next()){
        i1->Item()->timeWaited = i1->Item()->timeWaited + TimerTicks;

        if(i1->Item()->getID() > 0 && i1->Item()->timeWaited >= 1500){ // 確保thread ID是v
            int lastPriority = i1->Item()->priority;
            i1->Item()->priority = i1->Item()->priority + 10;
            if(i1->Item()->priority > 149) i1->Item()->priority = 149;
            i1->Item()->timeWaited = 0;

            DEBUG(dbgFlag, "[C] Tick [" << kernel->stats->totalTicks
                << "]: Thread [" << i1->Item()->getID()
                << "] changes its priority from [" << lastPriority
                << "] to [" << i1->Item()->priority  << "]");
            // if(i1->Item()->priority > 149) i1->Item()->priority = 149;
            // L1->Remove(i1->Item());
            // ReadyToRun(i1->Item());
```

```
        }
    }


    //L2

    ListIterator<Thread *> *i2 = new ListIterator<Thread *>(L2);
    for(; !i2->IsDone(); i2->Next()){
        i2->Item()->timeWaited = i2->Item()->timeWaited + TimerTicks;

        if(i2->Item()->getID() > 0 && i2->Item()->timeWaited >= 1500){ // 確保thread ID是v
            i2->Item()->timeWaited = 0;
            i2->Item()->priority = i2->Item()->priority + 10;
            DEBUG(dbgFlag, "[C] Tick [" << kernel->stats->totalTicks
                << "]: Thread [" << i2->Item()->getID()
                << "] changes its priority from [" << i2->Item()->priority - 10
                << "] to [" << i2->Item()->priority  << "]");
            // if(i2->Item()->priority > 149) i2->Item()->priority = 149;
            DEBUG(dbgFlag, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
                << i2->Item()->getID() << "] is removed from queue L[2]");
            L2->Remove(i2->Item());
            ReadyToRun(i2->Item());
        }
    }
```

```
    //L3
    ListIterator<Thread *> *i3 = new ListIterator<Thread *>(L3);
    for(; !i3->IsDone(); i3->Next()){
        i3->Item()->timeWaited = i3->Item()->timeWaited + TimerTicks;
        if(i3->Item()->getID() > 0 && i3->Item()->timeWaited >= 1500){ // 確保thread ID是v
            i3->Item()->timeWaited = 0;
            i3->Item()->priority = i3->Item()->priority + 10;
            DEBUG(dbgFlag, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
                << "] changes its priority from [" << i3->Item()->priority - 10
                << "] to [" << i3->Item()->priority << "]");
            // if(i3->Item()->priority > 149) i3->Item()->priority = 149;

            // 直接透過ReadyToRun來更新thread的queue，反正一樣是從L3的頭到尾去for loop
            // 所以重新放回L3的順序還是一樣的，差別只在如果有要去L2的會被放過去
            DEBUG(dbgFlag, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
                << "] is removed from queue L[3]");
            L3->Remove(i3->Item());
            ReadyToRun(i3->Item());
        }
    }
}
```

We modify Scheduler::FindNextToRun to make it first find L1, second L2, and last L3 to be the next thread to run.

```
Thread *next;
if (!L1->IsEmpty()) {
    next = L1->RemoveFront();
    DEBUG(dbgFlag, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << next->ge
        << "] is removed from queue L[1]");
    return next;
} else if (!L2->IsEmpty()) {
    next = L2->RemoveFront();
    DEBUG(dbgFlag, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << next->ge
        << "] is removed from queue L[2]");
    return next;
} else if (!L3->IsEmpty()) {
    next = L3->RemoveFront();
    DEBUG(dbgFlag, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << next->ge
        << "] is removed from queue L[3]");
    return next;
} else {
    return NULL;
}
```

Last, in Scheduler::Run, when the status of the thread is RUNNING, we reset its timeWaited.

```
nextThread->setStatus(RUNNING);      // nextThread is now running
// *** update waited time
```

```
nextThread->timeWaited = 0;
// ***
```

Thread.cc / Thread.h

We added 6 attributes for the Thread object

1. `priority`: contains the priority of a thread
2. `timeWaited`: records the how long a thread has been waiting in the ready queue
3. `L3ExecTick`: records the running time of a thread, used for Round-Robin scheduling for threads in level-3-queue
4. `expected`: expected CPU burst, updated after a thread goes from running to waiting
5. `burst`: the actual CPU burst time
6. `accumulatedTicks`: records the CPU burst time of the last execution of a thread

```
int priority;       // priority of threads used for scheduling
int timeWaited;     // 等待多少tick，用來update priority
int L3ExecTick;     // 算L3中的thread執行了多少tick，超過100 tick就換人
double expected;    // expected CPU burst
int burst;       // CPU burst time
int accumulatedTicks;
```

In the constructor of Thread, all these six attributes are initialized to 0.

`void Thread::Yield()`

When a thread calls `Yield()`, it goes from running to ready. Therefore, we update `L3ExecTick` and `accumulatedTicks` here.

```
void
Thread::Yield()
{
    // ...
    nextThread = kernel->scheduler->FindNextToRun();
        // update here
    accumulatedTicks = burst; // get the accumulated ticks for debug message
    L3ExecTick = 0;           // reset L3ExecTick

    if (nextThread != NULL) {
                // ...
        }
        // ...
}
```

`void Thread::Sleep(bool finishing)`

When a thread calls `Sleep()`, it goes from running to waiting. That is, its has ended its CPU burst. Therefore, we have to update the next approximated burst time here using the formula, and reset the burst time and `L3ExecTime`. We also have to store the burst time in `accumulatedTicks` for the debug message.

```
void
Thread::Sleep(bool finishing) // 進waiting
{
    Thread *nextThread;
    // ...
    // *** update approximate burst time
    if (!finishing) {
        DEBUG(dbgFlag, "[D] Tick [" << kernel->stats->totalTicks
            << "]: Thread [" << this->getID()
            << "] update approximate burst time, from: [" << expected
            << "], add [" << burst << "], to ["
            << (0.5 * expected + 0.5 * burst) << "]");
        expected = 0.5 * expected + 0.5 * burst;
        accumulatedTicks = burst;
        burst = 0;
        L3ExecTick = 0;
    }
    // ***

    status = BLOCKED;
```

```
        // ...
}
```

interrupt.cc

`void Interrupt::OneTick()`

After a user instruction is executed, we update `busrt` and `L3ExecTime`. If in system mode, we update them with system ticks instead of user ticks.

```
void
Interrupt::OneTick()
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;

        // advance simulated time
    if (status == SystemMode) {
                // ...
        // *** update CPU burst time and L3Exectime
        kernel->currentThread->burst += SystemTick;
        kernel->currentThread->L3ExecTick += SystemTick;
        // ***
    } else {
```

```
        // ...
        // *** update CPU burst time and L3Exectime
        kernel->currentThread->burst += UserTick;
        kernel->currentThread->L3ExecTick += UserTick;
        // ***
    }
        // ...
}
```

Alarm.cc

`void Alarm::CallBack()`

Every 100 ticks, the alarm call back function is called.

Here, we start with the `kernel->scheduler->priorityUpdate()`, it implements the aging mechanism.

Then, we perform the preemption if needed. There are 3 possible scenarios.

1. The current thread is in L3: Then, we may preempt it if it has executed more than 100 ticks, or if there are any threads in L1 or L2.
2. The current thread is in L2: Then, we may preempt it only if there are any threads in L1.
3. The current thread is in L1: Then, we may preempt it only if there are any threads in L1 and its remaining time is less than that of current thread. Since L1 is implemented as a SortedList, we have to do a linear search to find such a thread.

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

        // *** aging
    kernel->scheduler->priorityUpdate();
        // ***

        // *** preemption
    if(currentThread->priority < 50 && (currentThread->L3ExecTick >= 100 || !(scheduler->
        interrupt->YieldOnReturn();
        DEBUG(dbgFlag, "Tick [" << kernel->stats->totalTicks << "]: thread [" << kernel->
    } else if (currentThread->priority < 100 && !scheduler->getL1()->IsEmpty()) {
        interrupt->YieldOnReturn();
        DEBUG(dbgFlag, "Tick [" << kernel->stats->totalTicks << "]: thread [" << kernel->
    } else {
        ListIterator<Thread *> *i1 = new ListIterator<Thread *>(scheduler->getL1());
        bool stop = false;
        for(; !i1->IsDone() && !stop; i1->Next()){          // linear search
            if ((i1->Item()->expected - i1->Item()->burst) < (currentThread->expected - c
                interrupt->YieldOnReturn();
```

```
                DEBUG(dbgFlag, "Tick [" << kernel->stats->totalTicks << "]: thread [" <<
                stop = true;
            }
        }
    }
        // ***
}
```