Operating System MP3

Report

Team 47

2020/12/21

Team Members:

A discussion between two members was held on the implementation and the code tracing.

廖允寬 (Liao) 106034071 - Worked on Part I and Part II of the report.

洪紫珈 (Evelyn) 107062174 - Worked on Part I and Part II of the report.

PART I - Code Tracing

Explain the purposes and details of the following 6 code paths to understand how NachOS manages the lifecycle of a process (or thread) as described in the Diagram of Process State in our lecture slides (chp.3 p.8)

1. New → Ready

In ExecAll(), we use a for loop to execute all the files that await execution.

```
for (int i=1;i<=execfileNum;i++)
       { int a = Exec(execfile[i]); }
```

During each file execution, we use a prepared threadNum to create a new thread and place it in the thread list. Then it asks for a new address space by 'new AddrSpace()'which sets up the virtual address for this thread and the translation from program memory to physical memory.

```
t[threadNum]->space = new AddrSpace();
```

This thread is then executed with ForkExecute which distributes an execution stack with stackAllocate(func, arg).

```
t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
                                          &
StackAllocate(func, arg);
```

Disable the interrupts and set this thread(process) to ready so it will be scheduled to execute, then return to the original level which enables the interrupts.

```
oldLevel = interrupt->SetLevel(IntOff);
scheduler->ReadyToRun(this);
(void) interrupt->SetLevel(oldLevel);
```

Then 'threadNum++' prepares the next available thread number for the next thread to end a thread execution in 'Exec()' and returns the original threadNum. Finally, the thread created is set to finished and status becomes blocked in Sleep(true).

2. Running → Ready

In Machine::Run(), once we execute an instruction, the OneTick function is called.

```
kernel->interrupt->OneTick();
```

In Interrupt::OneTick(), the program increases the simulated time according to machine status and checks if there are any pending interrupts to be called.

```
if (status == SystemMode) {
      stats->totalTicks += SystemTick;
      stats->systemTicks += SystemTick;
} else {
      stats->totalTicks += UserTick;
```

```
        stats->userTicks += UserTick;
}
```

if the timer device handler asks for a context switch, it will call Yield().

```
if (yieldOnReturn) {
        yieldOnReturn = FALSE;
        status = SystemMode;
        kernel->currentThread->Yield();
        status = oldStatus;
}
```

In Thread::Yield(), it does a context switch on a thread that changes from running state to ready state, so it searches the next thread from the ready queue and puts the current thread back to the ready queue, and then calls Run() to run the next thread.

```
nextThread = kernel->scheduler->FindNextToRun();
                                &
kernel->scheduler->ReadyToRun(this);
kernel->scheduler->Run(nextThread, FALSE);
```

In Scheduler::FindNextToRun(), if the ready queue is empty, then return NULL, else it returns the thread and removes the thread from the ready queue.

```
if (readyList->IsEmpty()) return NULL;
else return readyList->RemoveFront();
```

In Scheduler::ReadyToRun(Thread*), the program sets the thread status to READY and add the thread into the ready list.

```
thread->setStatus(READY)
readyList->Append(thread);
```

In Scheduler::Run(Thread*, bool). the program first stores the status and states of the old thread, then runs the next Thread.

```
Thread *oldThread = kernel->currentThread;
ASSERT(kernel->interrupt->getLevel() == IntOff);
        if (finishing) {    // mark that we need to delete current thread
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
}
if (oldThread->space != NULL) { // if this thread is a user program,
        oldThread->SaveUserState();     // save the user's CPU registers
        oldThread->space->SaveState();
}
oldThread->CheckOverflow();         // check if the old thread
                            // had an undetected stack overflow
kernel->currentThread = nextThread;  // switch to the next thread
```

‘finishing’ will be true if the oldThread completes the execution and Sleep(TRUE). So This thread will be destroyed and all states are saved.
Then, the currentThread becomes the newly found thread and a context switch will be done.

```
SWITCH(oldThread, nextThread);
```

Lastly, if toBeDestroyed != NULL, then it will be deleted.

```
if (toBeDestroyed != NULL) {
    delete toBeDestroyed;
    toBeDestroyed = NULL;
}
```

3. Running → Waiting

In SynchConsoleOutput::PutChar(char), it uses

```
lock->Acquire();
                                        &
lockHolder = kernel->currentThread;
```

to get the available IO resources by making the currentThread the lockholder. This means, the resource will not be used by other threads, and this prevents multiple writers at the same time.

Then, Lock::Acquire() calls semaphore->P() which checks if there are free resources available.

```
while (value == 0) {        // semaphore not available
      queue->Append(currentThread);   // so go to sleep
      currentThread->Sleep(FALSE);
}
value--;            // semaphore available, consume its value
```

When value equals 0, it means no semaphores are available, so the program appends the current thread into the waiting queue in P(). Otherwise, it will decrement the value by 1 as an available resource is assigned to the thread. When no semaphores are unavailable, then Sleep(FALSE) is used to make sure the current thread relinquishes CPU. And it makes sure other threads use the CPU by FindingNextToRun(). When there are threads in the ready list, the program uses

```
kernel->scheduler->Run(nextThread, finishing);
```

to run the newly found thread.

As an available resource is assigned to the thread, it returns to PutChar() and uses consoleOutput->PutChar() to call WriteFile(). When the thread completes console display, the program runs to

```
waitFor->P();
```

This function simply awaits the call back function from consoleOutput::PutChar() as it completes the console output.

Then the SynchConsoleOutput::Putchar() calls lock->Release() to free up the resource. LockHolder is also set to NULL and becomes available for the next Thread.

```
void Lock::Release(){
    ASSERT(IsHeldByCurrentThread());
    lockHolder = NULL;
    semaphore->V();
}
```

4. Waiting → Ready

In Semaphore::V(), the program appends this current thread to the readylist again and removes it from the waiting queue. Not to mention, value is incremented by 1 to make sure the number of free sources are correct when the thread completes IO.

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

In Scheduler::ReadyToRun(Thread*)
Set the thread status to READY and add the thread into the ready list.

```
thread->setStatus(READY)
readyList->Append(thread);
```

5. Running → Terminated

When the thread has completed the execution, ExceptionHandler will be called and the exception type can be found in the register 2.
In ExceptionHandler(ExceptionType), the case 'SC_Exit' calls

```
kernel->currentThread->Finish();
```

In Thread::Finish(), we first disable the interrupt by

```
(void) kernel->interrupt->SetLevel(IntOff);
```

because Sleep() assumes interrupts are disabled.

```
Sleep(TRUE);                // invokes SWITCH
```

Then Sleep(TRUE) sets the status to BLOCK and finds a thread in the ready queue to execute by FindNextToRun(), otherwise, do nothing.

In FindNextToRun(), if the ready queue is empty, then return NULL, else it returns the thread and removes the thread from the ready queue.

```
if (readyList->IsEmpty()) return NULL;
```

```
else return readyList->RemoveFront();
```

Finally, in Run(), the program first stores the status and states of the old thread, then runs the next Thread.

```
Thread *oldThread = kernel->currentThread;
ASSERT(kernel->interrupt->getLevel() == IntOff);
      if (finishing) {    // mark that we need to delete current thread
      ASSERT(toBeDestroyed == NULL);
      toBeDestroyed = oldThread;
}
if (oldThread->space != NULL) { // if this thread is a user program,
      oldThread->SaveUserState();     // save the user's CPU registers
      oldThread->space->SaveState();
}
oldThread->CheckOverflow();         // check if the old thread
                        // had an undetected stack overflow
kernel->currentThread = nextThread;  // switch to the next thread
nextThread->setStatus(RUNNING);      // nextThread is now running
```

'finishing' will be true if oldThread has completed the execution and Sleep(TRUE). So This thread will be destroyed and all states are saved.

Then, the currentThread becomes the newly found thread, and NachOS does a context Switch.

```
SWITCH(oldThread, nextThread);
```

Lastly, if toBeDestroyed != NULL, then it will be deleted.

```
if (toBeDestroyed != NULL) {
    delete toBeDestroyed;
    toBeDestroyed = NULL;
}
```

6. Ready → Running

In Scheduler::FindNextToRun(), if the ready queue is empty, then return NULL, else it returns the thread and removes the thread from the ready queue.

```
if (readyList->IsEmpty()) return NULL;
else return readyList->RemoveFront();
```

In Scheduler::Run(Thread*, bool), the program first stores the status and states of the old thread, then runs the next Thread.

```
Thread *oldThread = kernel->currentThread;
ASSERT(kernel->interrupt->getLevel() == IntOff);
```

```
      if (finishing) {    // mark that we need to delete current thread
      ASSERT(toBeDestroyed == NULL);
      toBeDestroyed = oldThread;
}
if (oldThread->space != NULL) { // if this thread is a user program,
      oldThread->SaveUserState();     // save the user's CPU registers
      oldThread->space->SaveState();
}
oldThread->CheckOverflow();         // check if the old thread
                        // had an undetected stack overflow
kernel->currentThread = nextThread;  // switch to the next thread
```

'finishing' will be false if the oldThread has yet completed the execution and Sleep(FALSE). So all states of this thread are saved.

Then, the currentThread becomes the newly found thread and a context switch will be done.

```
SWITCH(oldThread, nextThread);
```

In SWITCH(), the program saves the content of eax into _eax_save, and then thread t1's(old thread) stack pointer can be moved to eax as the base register.

```
movl    %eax,_eax_save
movl    4(%esp),%eax
```

The content in each register is store to different address according to the offset

```
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
```

Restore the original %eax by _eax_save to %ebx, likewise, the content is stored to the address according to the offset.

```
movl    _eax_save,%ebx          # get the saved value of eax
movl    %ebx,_EAX(%eax)         # store it
```

Finally, store the return address to program counter storage.

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

Now, move the pointer to thread t2(next thread) into eax, so that we can access all the addresses by the base address and offset.

```
movl    8(%esp),%eax
```

After getting the base address, read the _EAX to %ebx and then move to _eax_save

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

Restore each register in the same way as above.

```
movl    _EBX(%eax),%ebx         # restore 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 return address to stack and restore %eax by _eax_save, and complete the context switch.

```
movl    _PC(%eax),%eax          # restore return address into eax
movl    %eax,0(%esp)            # copy over the ret address on the stack
movl    _eax_save,%eax
```

In Machine::Run(), once we execute an instruction, the OneTick function is called.

```
kernel->interrupt->OneTick();
```

In Interrupt::OneTick(), the program increases the simulated time according to machine status and checks if there are any pending interrupts to be called.

```
if (status == SystemMode) {
      stats->totalTicks += SystemTick;
      stats->systemTicks += SystemTick;
} else {
      stats->totalTicks += UserTick;
      stats->userTicks += UserTick;
}
```

if the timer device handler asks for a context switch, it will call Yield().

```
if (yieldOnReturn) {
      yieldOnReturn = FALSE;
      status = SystemMode;
      kernel->currentThread->Yield();
      status = oldStatus;
}
```

PART II - Implementation

2-1. Implementing a Muti-level Feedback Queue:

In kernel.h, we added an array to record each thread's priority.

```
int priority[10];
```

In kernel.cc, the "ep" flag was added into Kernel::Kernel(), initializing the priority array.

```
if (strcmp(argv[i], "-ep") == 0){
    execfile[++execfileNum] = argv[++i];
    priority[execfileNum] = atoi(argv[++i]);
    cout << execfile[execfileNum] << "\n";
}
```

And each thread's priority is given by this line in Kernel::Exec().

```
t[threadNum]->priority = priority[threadNum];
```

In thread.h, we added these variables and their descriptions are as commanded, and we set their values to zero in thread.cc Thread::Thread().

```
int priority; // thread's priority
int StartTime; //the time that thread start in running state
int BurstTime; //the time that thread during running state
int TimeInReadyQueue; // time waiting in ready queue
int ReadyStartTime; // time when the thread enters the ready list
double predictTime; // 0.5*BurstTime + 0.5*predictTime
```

Also, in thread.cc Thread::Yield(), we calculate the burst time here because this function takes out the thread from running state and puts it into the ready queue.

```
this->BurstTime += (kernel->stats->totalTicks - this->StartTime);
```

For Thread::Sleep(), thread status: Running->Waiting, so we calculate the burst time first, and then calculate the approximate burst time by the formula in the specification. Before running the next thread, we set the current thread's burst time to zero so that it can count the next burst time in CPU. Note that the burst time we used is only for calculating the next predict burst time, not summing up all the CPU bursts of the thread.

```
this->BurstTime += (kernel->stats->totalTicks - this->StartTime);
                                    &
this->predictTime = 0.5*this->predictTime + 0.5*this->BurstTime;
```

In scheduler.h, we added two SortedList<Thread *> and one List<Thread *> which are L1, L2, L3 respectively.

```
SortedList<Thread *> *L1;
```

```
SortedList<Thread *> *L2;
List<Thread *> *L3;
```

L1, L2's corresponding compare function are as below:

```
int Timecmp(Thread* t1,Thread* t2){//L1 predict burst time comparison
    return (t1->predictTime >= t2->predictTime) ? 1 : -1;
    // shorter predict time do first
}
int Pricmp(Thread* t1,Thread* t2){// L2 priority comparison
    return (t1->priority <= t2->priority) ? 1 : -1;
    // higher priority do first
}
```

In scheduler.cc Scheduler::ReadyToRun(),
According to each thread's priority, we insert the thread to a different ready queue, and we record the ReadyStartTime here.

```
thread->ReadyStartTime = kernel->stats->totalTicks;
if (thread->priority < 150 && thread->priority >= 100){
    L1->Insert(thread);
}
else if (thread->priority < 100 && thread->priority >= 50) {
    L2->Insert(thread);
}
else if (thread->priority < 50) {
    L3->Append(thread);
}
```

Scheduler::FindNextToRun ()
First, we find the next thread from the ready queues by the order L1->L2->L3, once we remove a thread from the ready queue, we calculate the TimeInReadyQueue because the thread is going to switch to running state.

```
if(!L1->IsEmpty()){
    thread = L1->RemoveFront();
    thread->TimeInReadyQueue += (kernel->stats->totalTicks -
thread->ReadyStartTime);
}else if(!L2->IsEmpty()){
    thread = L2->RemoveFront();
    thread->TimeInReadyQueue += (kernel->stats->totalTicks -
thread->ReadyStartTime);
}else if(!L3->IsEmpty()){
    thread = L3->RemoveFront();
```

```
    thread->TimeInReadyQueue += (kernel->stats->totalTicks -
thread->ReadyStartTime);
}else{
    thread = NULL;
}
return thread;
```

Scheduler::Run()

We set the next thread's StartTime when context switch occur because nextThread's state changes to running.

```
nextThread->StartTime = kernel->stats->totalTicks;
```

Scheduler::agingCheck()

The following four functions are called to do aging and check if it needs to do preemption.

```
aging(L1);
aging(L2);
aging(L3);
preemptL1(L1);
```

Scheduler::aging()

In this function, we loop the ready queue, and for each thread, we check whether its TimeInReadyQueue is larger than 1500, if so, we add 10 to its priority and subtract 1500 from TimeInReadyQueue.

```
int total =  kernel->stats->totalTicks - iterThread->ReadyStartTime +
iterThread->TimeInReadyQueue;
int oldPriority = iterThread->priority;
if((oldPriority >= 0 && oldPriority < 150) && total >= 1500){
            iterThread->TimeInReadyQueue = total - 1500;
            iterThread->ReadyStartTime = kernel->stats->totalTicks;
            iterThread->priority = (oldPriority+10>149) ? 149 : oldPriority+10;
```

If the iterThread's priority is larger than 99 after adding 10, it will be inserted into the L1 queue, and if the current thread's priority is less than 100 or its approximate burst time is longer than iterThread, the current thread will be preempted.

```
if(iterThread->priority >= 100){
      L1->Insert(iterThread);
      if(kernel->currentThread->priority < 100){
            kernel->alarm->preemptive = true;
      } else if(kernel->currentThread->priority >=100 &&
      kernel->currentThread->predictTime > iterThread->predictTime){
            kernel->alarm->preemptive = true;
      }
}
```

If the iterThread's priority is larger than 49 and less than 100, it will be inserted into the L2 queue, and if the current thread's priority is less than 50, the current thread will be preempted.

```
else if(iterThread->priority >= 50){
	L2->Insert(iterThread);
	if(kernel->currentThread->priority < 50){
		kernel->alarm->preemptive = true;
	}
}
```

If none of the above cases, the iterThread will be inserted into L3 queue.

```
else{
	L3->Append(iterThread);
}
```

Scheduler::preemptL1()

Check if any thread's approximate burst time is shorter than current thread in L1 queue, if so, the current thread will be preempted.

```
if(kernel->currentThread->priority >=100 && kernel->currentThread->predictTime > iterThread->predictTime){
	kernel->alarm->preemptive = true;
}
```

In alarm.h, we added a bool variable.

```
bool preemptive;
```

In alarm.cc, we call agingCheck() first and then call YiledOnReturn if we need to do preemption or the current thread is in L3 queue.

```
if ((status != IdleMode && ((kernel->currentThread->priority < 50) || preemptive))) {
	kernel->alarm->preemptive = false;
	interrupt->YieldOnReturn();
}
```

2-3. Debugging Message:

In debug.h, we added the following line to print the display message.

```
const char dbgFlag = 'z';
```

A. When a process is inserted into a queue:
In <scheduler.cc>, we add the following lines to Scheduler::ReadyToRun(Thread *thread) and to Scheduler;:aging(List<thread*> *list).

```
DEBUG(dbgFlag, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread ["
		<< thread->getID() << "] is inserted into queue L[1]");
```

```
                                        &
DEBUG(dbgFlag, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< thread->getID() << "] is inserted into queue L[2]");
                                        &
DEBUG(dbgFlag, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< thread->getID() << "] is inserted into queue L[3]");
```

When running ReadyToRun, the threads are inserted into the list according to their priority. Also, if the priority is increased in 'aging' and the thread wasn't in L1, then it will be moved to L1 which also prints the debug message for L1. Not to mention, the samefor L2, if priority increased and the thread was in L3, then it will be moved to L2 with corresponding debug message.

B. When a process is removed from a queue:
In <scheduler.cc>, we add the following lines to Scheduler;:aging(List<thread*> *list). When the priority changes and the thread is moved to another queue, then message is printed.

```
DEBUG(dbgFlag,"[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< iterThread->getID()  << "] is removed from queue L[2]");
                                        &
DEBUG(dbgFlag,"[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< iterThread->getID()  << "] is removed from queue L[3]");
```

It is important to note that thread will only be moved from L2 and L3 in the operation of aging, because threads in L1 with increased priority can only remain its place in L1.

Also, when we go through the queues in Scheduler::FindNextToRun(), we remove the threads from the queue when it is being selected for the next running thread. So the debug message is printed when any thread is removed from any of the three queues.

C. When a process changes its scheduling priority:
In <scheduler.cc>, we add the following lines to Scheduler;:aging(List<thread*> *list), when 1500 ticks passed and priority of thread is increased by 10 or reached 149, the line is printed.

```
if((oldPriority >= 0 && oldPriority < 150) && total >= 1500) {
      DEBUG(dbgFlag ,"[C] Tick [" << kernel->stats->totalTicks << "]:
      Thread [" <<  iterThread->getID()  << "] changes its priority from["
       << oldPriority << "] to [" << iterThread-> priority <<"]");
}
```

D. When a process updates its approximate burst time:

In <thread.cc>, we add the following lines to Thread::Sleep(bool finishing).
Sleep is called in two conditions, either when a thread goes from run to wait or run to terminate.

```
If (!finishing){
      DEBUG(dbgFlag,"[D] Tick [" << kernel->stats->totalTicks << "]:
      Thread [" << this->getID()  << "] update approximate burst time,
      from: [" << this->predictTime << "], add [" << this->BurstTime <<
      "], to [" << 0.5*this->predictTime + 0.5*this->BurstTime << "]");
      this->predictTime = 0.5*this->predictTime + 0.5*this->BurstTime;
}
```

'!finishing' means the thread goes from run to wait, and its approximate burst time is changed every time a thread has done using CPU. Accordingly, the debug message for D is printed when !finishing is detected.

E. When a context switch occurs:
In <thread.cc>, we add the following lines to Thread::Sleep(bool finishing).
Again, sleep is called when a thread goes from run to terminate. Whether the thread has done execution or not, a context will happen, so a debug message for E is written in Sleep(bool finishing).

```
void
Thread::Sleep (bool finishing)
{
      DEBUG(dbgFlag,"[E] Tick [" << kernel->stats->totalTicks << "]:
      Thread [" << nextThread->getID()  << "] is now selected for
      execution, thread [" << this->getID() << "] is replaced, and it has
      executed [" << this->BurstTime << "] ticks");
}
```

Also, in Thread::Yield(), since a context switch is required when a thread goes from run to ready. Hence, the debug message is printed.

```
if (nextThread != NULL) {
      DEBUG(dbgFlag,"[E] Tick [" << kernel->stats->totalTicks << "]:
      Thread [" << nextThread->getID()  << "] is now selected for
      execution, thread [" << this->getID() << "] is replaced, and it has
       executed [" << this->BurstTime << "] ticks");
}
```