Operating systems
MP3

Team 67

108062128 林育丞

108062135 呂佳恩

分工表

林育丞	Trace Code, Question 回答整理, Report 整理
呂佳恩	Trace Code, Implementation, Report 整理

‘

(1) Trace code, explanation

1-6. New→Ready

Kernel::ExecAll( )

Call Exec() for each file that needs execution.
Call finish() when the loop ends

↓

Thread::Fork(VoidFunctionPtr, void*)

First parameter passes the procedure address (which is ForkExecute()).
Second parameter passes the pointer of the thread object.
Call **StackAllocate( )** to create space for this thread.
Set interrupt to off.
Put current thread into scheduler for run **(ReadyToRun()).**
Set interrupt back to the original value.

↓

Thread::StackAllocate(VoidFunctionPtr, void*)

First parameter passes the procedure address (which is ForkExecute( )).
Second parameter passes the pointer of the thread object.
Call AllocBoundedArray( ) to allocate space needed for this thread.
#ifdef directive allows for conditional compilation
Setup machine state for the thread.

↓

Scheduler::ReadyToRun(Thread*)

Set this thread to Ready state.
Append it on the ready list.

1-2. Running→Ready

Machine::Run( )

In UserMode, call OneInstruction() and OneTick() in an endless loop.

↓

Interrupt::OneTick( )

Advance simulated time for UserMode or for KernelMode.
Check if there is any incoming Pending Interrupt.
Check if timer asked for a context switch, if true, then call **Yield( )**.

↓

Thread::Yield( )

Set interrupt to off.
If there are other threads to run, put current thread to the back of the ready list, then SWITCH to next thread and run it.
If there are no other threads to run, this function will return immediately.

↓

Scheduler::FindNextToRun()

Remove the front-most thread in the ready list according to the priority level, if any.

↓

Scheduler::ReadyToRun(Thread*)

Set this thread to ready status.
Append this thread to the ready list.

↓

Scheduler::Run(Thread*, bool)

Check if finishing, if true, it will destroy the current thread.
Check if thread is user program, if so, save CPU regs.
CheckOverflow( ) will check if stack overflow happened.
Change to next thread and set its state to Running.
Call SWITCH to stop current thread and start new thread.
Call **CheckToBeDestroyed( )** if there is any toBeDestroyed thread, if so, delete it.
Try to restore available address spaces.

1-3. Running→Waiting (only consider console output as an example)

SynchConsoleOutput::PutChar(char)

Use **lock->Acquire( )** to obtain a lock, to make sure only one thread can access this I/O service at a time.
Call **consoleOutput->PutChar(ch)** to do the output.
Use waitFor->P( ) to wait for the callback function to call waitFor->V( ).
Finally release this lock.

↓

Semaphore::P( )

Set interrupt to off
Check if semaphore value == 0, if so, append this thread to the back of the waiting queue of this semaphore class, and make this thread sleep.
Until the callback function of this ConsoleOutput class is called (after ConsoleTime passed), the value of semaphore will increase to 1, by calling **semaphore->V( )**.
Then the slept thread can finally return and set interrupt to its original level.

↓

Thread::Sleep(bool)

With parameter finishing == FALSE, this function will block this thread but not delete it.
Try to find next thread to run. If none, then call **interrupt->idle( )**.
If there exists a thread to run, then run it.

↓

Scheduler::ReadyToRun(Thread*)

Set this thread to ready status.
Append this thread to the ready list.

↓

Scheduler::Run(Thread*, bool)

Check if finishing, if true, it will destroy the current thread.
Check if thread is user program, if so, save CPU regs.
CheckOverflow( ) will check if stack overflow happened.
Change to next thread and set its state to Running.
Call SWITCH to stop current thread and start new thread.
Call **CheckToBeDestroyed( )** if there is any toBeDestroyed thread, if so, delete it.
Try to restore available address spaces.

1-4. Waiting→Ready (only consider console output as an example)

Semaphore::V( )

Set interrupt to off.
Retrieve the thread from the waiting queue, if any, and put it in the ready list.
Increase the semaphore value to break the while loop in Semaphore::P( ).
Restore interrupt to its original value.

↓

Scheduler::ReadyToRun(Thread*)

Set this thread to ready status.
Append this thread to the ready list.

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

ExceptionHandler(ExceptionType) case SC_Exit

Read the return value of this thread from register 4 and show it in the console.
Call Finish( ) to finish this thread.

↓

Thread::Finish()

Set interrupt to off.
Call Sleep(TRUE)

↓

Thread::Sleep(bool)

With the parameter finishing == TRUE, the thread will be destroyed in the end.
Try to find next thread to run. If none, then call **interrupt->idle( )**.
If there exists a thread to run, then run it.

↓

Scheduler::FindNextToRun( )

Remove the front-most thread in the ready list according to their priority, if any.

↓

Scheduler::Run(Thread*, bool)

Same explanation as before, but with finishing == true, it will destroy the old thread.

1-6. Ready→Running

Scheduler::FindNextToRun( )

Remove the front-most thread in the ready list according to their priority, if any.

↓

Scheduler::Run(Thread*, bool)

Same explanation as before

↓

SWITCH(Thread*, Thread*)

Save registers, stack pointer, and return address from old thread.
Load registers, stack pointer, and return address from new thread.
Return to where the return address pointed at.
For main thread, it will return to the ThreadRoot function defined in switch.S.
For other threads, it will return to the SWITCH function in Scheduler::Run( ).

↓

for loop in Machine::Run()

The execution path is ForkExecute → AddrSpace::Execute → Machine::Run
call OneInstruction( ) and OneTick( ) in an endless loop.

(2) Implementation

First, we add some properties to thread and scheduler

```
private:
  // NOTE: DO NOT CHANGE the order of the
  // THEY MUST be in this position for SW
  int *stackTop;                          /
  void *machineState[MachineStateSize]; /
  int priority;
  int CPUburst;
  int age;
```

We add functions to modify the private properties so that they are protected

```
  void resetage();
  void Aging();
  void setpriority(int p);
  int getpriority();
  void setburst(int b){ CPUburst = b; }
  int getburst(){ return CPUburst; }
  int s_tick;
```

Then, we add the 3 queues into the Scheduler

```
SortedList<Thread *> *L1readyList;
SortedList<Thread *> *L2readyList;
List<Thread *> *L3readyList;  // queue of threads that are ready to run,
```

For the first two List, we have to write the comparator since it requires sjf and priority to be the comparator

```
int Scheduler::priority(Thread *a, Thread *b){
    int a_priority = a->getpriority();
    int b_priority = b->getpriority();
    return ((a_priority == b_priority) ? 0 : ((a_priority > b_priority) ? -1 : 1));
}
int Scheduler::sjf(Thread *a, Thread *b){
    double timeA = a->CPUguesssedburst - a->getburst();
    double timeB = b->CPUguesssedburst - b->getburst();
    if(timeA < 0)
        timeA = 0;
    if(timeB < 0)
        timeB = 0;
    // cout << "burstA = " << a->getburst() << endl;
    // cout << "burstB = " << b->getburst() << endl;
    return ((timeA == timeB)? priority(a, b):((timeA > timeB) ? 1 : -1 ));
}
```

And for the round robin, aging and preemptive implementation, we add that in the alarm.cc file.

```
if (status != IdleMode) {
    Thread *now = kernel->currentThread;
    // If now thread is L1, then try to find if other thread in L1 can replace it
    if (now->getpriority() > 99) {
        if (kernel->scheduler->IsPreemptive()){
            interrupt->YieldOnReturn();
        }
    }
    // Else now is a L2 L3 thread, then if L1 is not empty, L1 can preempt now
    else {
        if (!kernel->scheduler->L1_empty()) {
            interrupt->YieldOnReturn();
        } else {
            if (now->getpriority() >= 0 && now->getpriority() <= 49) {
                interrupt->YieldOnReturn();
            }
        }
    }
}
kernel->scheduler->Aging();
```

For threads that have a L1 priority, we check if it is preemptive. Since the spec requires us to use the remaining burst time, we check if the remaining time is less than the first of the L1 list

```
if (!L1_empty()) {
    Thread *first = L1readyList->Front();
    Thread *now = kernel->currentThread;
    int newBurst = kernel->stats->totalTicks - now->s_tick;
    // int newBurst = now->getburst();
    // int tmp = now->CPUguesssedburst - newBurst;
    // int cmpburst = first->CPUguesssedburst;
    int tmp = now->CPUguesssedburst - newBurst - now->getburst();
    int cmpburst = first->CPUguesssedburst- first->getburst();
    if(cmpburst < 0)
        cmpburst = 0;
    if(tmp<0)
        tmp = 0;
    // cout << "Now burst is" << now->getburst() << " The guesstime
    // cout << "First burst is" << first->getburst() << " The guess
    // cout << '\n' << first->CPUguesssedburst << "\n" << now->CPUg
    return cmpburst < tmp;
}
```

And we age the thread in alarm.cc, if the thread is ran, we set age to 0;

```
void Thread:: Aging()
{
    age++;
    if (age >= 15)
    {
        age = 0;
        int pre = priority;
        priority += 10;
        if (priority >= 149)
            priority = 149;
```

We have to set burst to 0 when sleep. Also updating the guess time

```
int burst = this->getburst();
//
int newBurst = burst + kernel->stats->totalTicks - this->s_tick;
this->setburst(0);
double guess = 0.5*newBurst + 0.5*this->CPUguesssedburst;
this->CPUguesssedburst = guess;
```