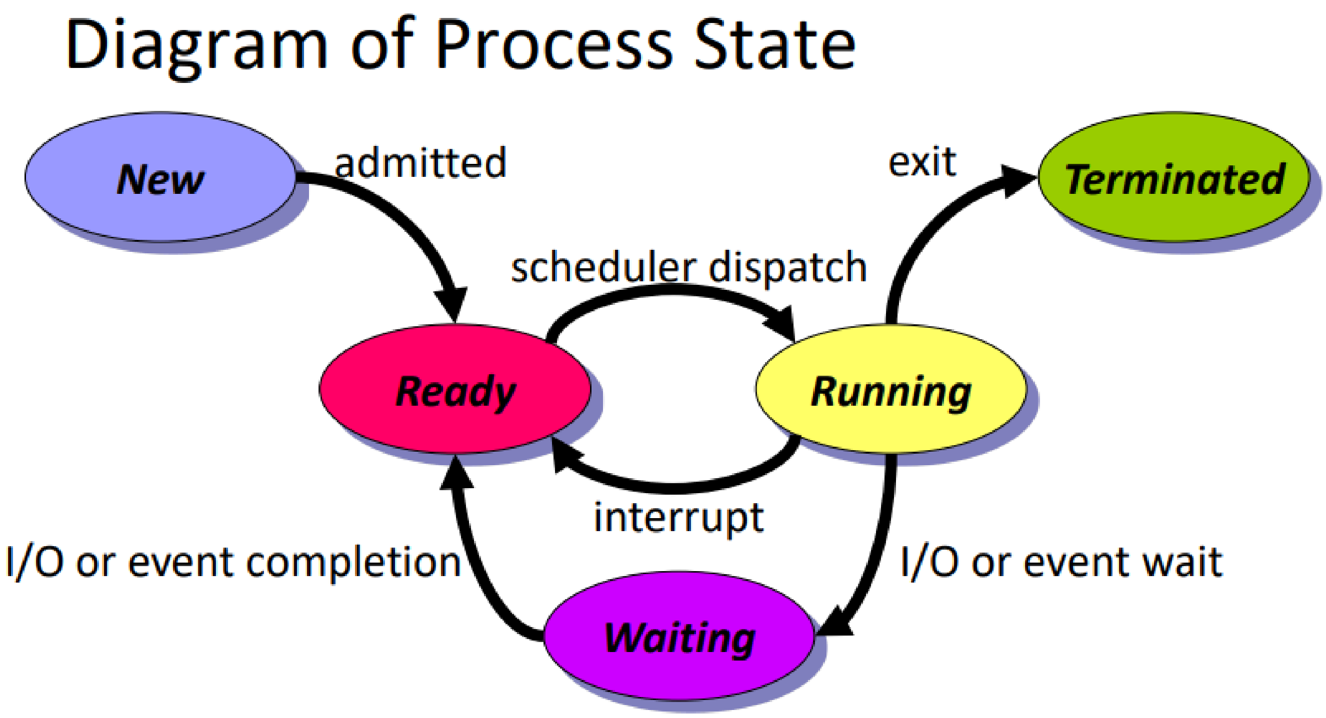
113-1 Operating System MP3

Group 45: 111062332 朱誼學, 111062333 高英耀

Contributions	**111062332**	**111062333**
Trace code	50%	50%
Implementation	50%	50%
Report	50%	50%
Explanation	We've done all our work in discord vc	

Trace code

The image from page8 of chapter3 slide



NOTE: Explanation in Trace code section are consisted of

1. Function explanation (Explain what the functions are mainly doing)
2. Explanation of Purpose and Detail (Explain the state changing routine)
    a. If already know how functions work, just look at this part.

1-1. New → Ready:

StartUp

- In `threads/main.cc`, main function will launch the NachOS system. Create and initialize kernel, then call `kernel->ExecAll()`

```
kernel = new Kernel(argc, argv);
kernel->Initialize();
...
kernel->ExecAll();
```

Kernel::ExecAll()

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

1. In `Kernel->Initialize`, all executable file names will be load into execfile.
2. And `ExecAll()` will sequentially execute all files by calling `Kernel->Exec()`
3. When all the process are finished, the currentThread in this scope, which is the main thread can call `Finish()` to halt NachOS.

Kernel::Exec(char*)

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

    return threadNum - 1;
}
```

This function mainly does the following three tasks.

1. Create a new thread (class `Thread`)
    - Thread control block
    - Contain StackTop and MachineState(registers)
    - Thread instructions
        - Begin, Fork, Sleep, Yield, Finish
2. Create an instance of AddrSpace
    - Memory Management(ex. pageTable)
    - Run a program
    - Do context switch
3. Call `Fork()`
    - Detail in the next section

Thread::Fork(VoidFunctionPtr, void*)

```
void Thread::Fork(VoidFunctionPtr func, void *arg) {
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int)func << " "
<< arg);
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);  // ReadyToRun assumes that interrupts
                                  // are disabled!
    (void)interrupt->SetLevel(oldLevel);
}
```

- Allocate a stack, calling `StackAllocate()` to setup stack and `MachineState` for the created thread.
- Initialize the stack so that a call to SWITCH can cause it to run the procedure.
- Put the thread on the ready queue.

Thread::StackAllocate(VoidFunctionPtr, void*)

```
//  We just focus on the important part of the code
void Thread::StackAllocate(VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    stackTop = stack + StackSize - 4;
    *(--stackTop) = (int) ThreadRoot;

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

- Allocate space for stack
- Make stackTop point to the top of the stack (high address)
- Place `ThreadRoot` onto the stack
- Setup `machineState`, which are kernel registers

Scheduler::ReadyToRun(Thread*)

```
void Scheduler::ReadyToRun(Thread *thread) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

- After `Fork()` called `StackAllocate()`, disable the interrupt first, then call this function to push created thread into ready queue.

Explanaion of Purpose and Detail

- A new created thread must be put in the ready queue before it starts running.
- Files will be stored in `execfile` for the CLI argument.
- `ExecAll()` will parse over the array and pass all entries to `Exec()`
- `Exec()` will create a new thread and address space, then Fork the new thread.
- In `Fork()`, the thread's stack will be allcated be calling `StackAllocate()`
- After allcating the stack, it will back to `Fork()`, then the thread will be append to the ready queue. (complete the path of New → Ready)

1-2. Running → Ready:

Machine::Run()

```
// Not included the Debug codes
void Machine::Run() {
    Instruction *instr = new Instruction;  // storage for decoded instruction

    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
    }
}
```

- This function simulates the process running user programs.
- An infinite loop will keep fetching instructions and simulate time action by `OneTick()`

Interrupt::OneTick()

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

    // advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }

    // check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff);
    CheckIfDue(FALSE);
    ChangeLevel(IntOff, IntOn);
    if (yieldOnReturn) {
        yieldOnReturn = FALSE;
        status = SystemMode;
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

- Increase `totalTicks` depends on current mode.
- Check whether there is any pending interrupt by calling `CheckIfDue()`
  - If has pending interrupt, fire and trigger `CallBack()`
  - `CallBack()` function in `CheckIfDue` will eventually trigger `Alarm::CallBack()`, which set `yieldOnReturn` to `TRUE`.
- After `CheckIfDue()`, whether `Yield()` is called depends on the value of `yieldOnReturn`.
- Here's the detail code for `CheckIfDue()`.

```
// CheckIfDue(bool)
// Removed debug codes
...
inHandler = TRUE;
do {
    next = pending->RemoveFront();  // pull interrupt off list
    next->callOnInterrupt->CallBack();  // call the interrupt handler
    delete next;
} while (!pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks));
inHandler = FALSE;
return TRUE;
```

Thread::Yield()

```
void Thread::Yield() {
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Yielding thread: " << name);

    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void)kernel->interrupt->SetLevel(oldLevel);
}
```

- Yield means that this thread will be interrupted.
- If there's any thread found in ready queue, put this thread on the end of ready queue intermediatly by calling `ReadyToRun()`.
- Run the next thread by `Run()`. (Context switch is done in this function)

Scheduler::FindNextToRun()

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

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

- Check whether ready queue is empty, pop out the first thread if is not empty.

Scheduler::ReadyToRun(Thread*)

```
void Scheduler::ReadyToRun(Thread *thread) {
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

- Set the input thread's state to `READY`
- Put the thread on ready queue. (Running → Ready)

Scheduler::Run(Thread*, bool)

```
// Captured the important part of this function.
void Scheduler::Run(Thread *nextThread, bool finishing) {
    Thread *oldThread = kernel->currentThread;

    // mark that we need to delete current thread
    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

    // Preprocess of Context Switch
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }

    oldThread->CheckOverflow();

    // switch to the next thread
    kernel->currentThread = nextThread;
    // nextThread is now running
    nextThread->setStatus(RUNNING);

    // Defined in switch.S
    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    CheckToBeDestroyed();  // check if thread we were running
                           // before this one has finished
                           // and needs to be cleaned up
    // NULL means cleaned up, otherwise, hasn't finished
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

- In this function, will run a new process.
- Doing context switch before running a new process.
    - Check whether the old thread is terminated, if true, marked as needed to be cleaned up.
        - Depends on `finishing`.
    - Save register values and user state into thread space.
    - Change current thread pointer in kernel, and set the thread status to `RUNNING`.
    - call `SWITCH()`
- `SWITCH()` is an important function defined as assembly, which will be explained later.
- When `SWITCH()` is called, jump to the next thread's code section and execute.

- When the next thread is terminated, return to where `SWITCH()` is called and keep running the old thread.
- If old thread is finished, destroy the thread.
- If not, restore the registers and user state (ex. pageTable).

Explanation of Purpose and Detail

- When an interrupt occurs, the current thread will change from running state to ready state.
- Start from `Machine::Run()`, running infinite loop for `OneInstruction()` and `OneTick()`
    - `OneInstruction()` is for executing, and `OneTick()` handles the interrupt.
- In `OneTick()`, call `CheckIfDue()` to check whether there a pending interrupt, if true, the callback function of `PendingInterrupt` will eventually trigger `Alarm::CallBack()`, which makes `yieldOnReturn` to be `True`.
- When return to the next row of `CheckIfDue`, if there's any pending interrupt, `yieldOnReturn` will be toggled and call `Thread::Yield()` for current thread.
- In `Yield`, if next thread is found, current thread will change to ready state and be put in the ready queue.
    - complete the path of `Running → Ready`
- After that, next thread starts running.

1-3. Running → Waiting

SynchConsoleOutput::PutChar(char)

```
void SynchConsoleOutput::PutChar(char ch) {
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P();
    lock->Release();
}
```

- call `lock->Acquire()` so that only 1 thread can access I/O each time.
- `Putchar(char)` will output a character to simulated I/O
- call `waitfor->p()`: wait until `semaphore value` > 0, keep running when trigger callback
- release the lock()
- `lock` is used to ensure only 1 thread using `SynchConsoleOutput` each time, `waitfor` is used to ensure `PutChar()` is done.
- Eventhough both of them call `Semaphore::P()`, the purpose are a little bit different.

Semephore::P()

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

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

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

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

- Disable interrupts and cache the previous state of interrupt
- When `Semaphore::value==0`, insert currentThread into `Semaphore::queue`(waiting queue)
- Then call `currentThread->Sleep(FALSE)`, this will put currentThread in `BLOCKED` state.
- When semapore is available, leave the while loop and deduct `value`.
- Re-enable interrupt.

List::Append(T)

```
template <class T>
void List<T>::Append(T item) {
    ListElement<T> *element = new ListElement<T>(item);

    ASSERT(!IsInList(item));
    if (IsEmpty()) {  // list is empty
        first = element;
        last = element;
    } else {  // else put it after last
        last->next = element;
        last = element;
    }
    numInList++;
    ASSERT(IsInList(item));
}
```

- `List<T>`is a linked-list structure defined in nachos.
- `Append()` is one of the member function in class `List`.
- This function will insert an item to the end of list.

Thread::Sleep(bool)

```
void Thread::Sleep(bool finishing) {
    Thread *nextThread;
    setStatus(BLOCKED);

    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();
    }
    kernel->scheduler->Run(nextThread, finishing);
}
```

- In `Sleep()` , the thread instance which call the function will be `BLOCKED`
- Find the next thread to run.
  - If found: call `Scheduler::Run()`, then switch to nextThread
  - Else, call `Interrupt::Idle()`, handle pending interrupt or terminate the system if all processes are terminated.

Scheduler::FindNextToRun()

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

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

- If ready queue is not empty, return the first thread in the queue.

Schduler::Run(Thread*, bool)

```
void Scheduler::Run(Thread *nextThread, bool finishing) {
    Thread *oldThread = kernel->currentThread;
    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }

    oldThread->CheckOverflow();

    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);

    SWITCH(oldThread, nextThread);
    // we're back, running oldThread
    CheckToBeDestroyed();
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState(); \
        oldThread->space->RestoreState();
    }
}
```

- Check whether oldThread is finished.
- Save user state if oldThread is user porgram.
- Change nextThread to running state.
- Do Context Switch in `SWITCH()`
- After context switch, if oldThread is finished, delete it.
- If is not finished, next time switch back to oldThread, continue running process.

Explanation of Purpose and Detail

- When handling synchronization issue, the thread which accessed I/O will be `BLOCKED`. Can also sayed that is put into waiting queue.
- Both `lock` and `waiting` will call `Semaphore::P()`, this means that a thread is occupying the resource (I/O).
- If `value` is greater than 0, the usage of I/O is available, then decrease `value` by 1 if a thread going to use it.
- When `value == 0`, put currentThread into waiting queue, and call `Sleep()` to blocked currentThread.
    - complete the path of `Running → Waiting`
- While oldThread is `BLOCKED`, find a new thread to run, if not found, kernel will be set to `Idle()`.

1-4. Waiting → Ready

Semaphore::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);
}
```

- Usually called in `callback()` function.
- Disable interrupt.
- Insert the thread, which `Semaphore::P()` put into waiting queue, into ready queue. Then set state to `READY`.
- Increase `value` by 1, indicates that I/O is available now.
- Re-enable interrupt.

Scheduler::ReadyToRUn(Thread*)

```
void Scheduler::ReadyToRun(Thread *thread) {
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

- Set to `READY` state.
- Append into ready queue.

Explanation of Purpose and Detail

- When either return the `lock` or in many of the `callback()` functions, `Semaphore::V()` is called to represent that the resource (Usually I/O) is available now.
- `Semaphore::V()` will increase `value` by 1, so that other threads can use this resource (Usually I/O).
- At the same time, get a thread needed to use the resource from waiting queue (`Semaphore::queue`).
- Change state from `BLOCKED` to `READY`, and append to ready queue (Scheduler::readyList).
- Complete the procedure of `Waiting → Ready`.

1-5. Running → Terminated

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;
```

- In `ExceptionHandler()`, `which` will indicate the kind of exception.
    - for SC_Exit, is `SyscallException`
- After reading the data from register, it will get the type of syscall (`SC_Exit`).
- Then call the thread's finish function.

Thread::Finish()

```
void Thread::Finish() {
    (void)kernel->interrupt->SetLevel(IntOff);

    if (kernel->execExit && this->getIsExec()) {
        kernel->execRunningNum--;
        if (kernel->execRunningNum == 0)
            kernel->interrupt->Halt();
    }
    Sleep(TRUE);  // invokes SWITCH
}
```

- Call `Sleep(TRUE)` to terminate the thread.
- By passing `finishing=TRUE`, the thread will be deleted by scheduler.

Thread::Sleep(bool)

```
void Thread::Sleep(bool finishing) {
    Thread *nextThread;
    setStatus(BLOCKED);
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
        kernel->interrupt->Idle();
    kernel->scheduler->Run(nextThread, finishing);
}
```

- Put old thread BLOCKED (waiting state). And set kernel idle until there a thread in ready queue.
- If next thread is found, call scheduler to run the thread.
- `finishing` is passed into `Run()`, and the old thread will be deleted after context switch.

Scheduler::FindNextToRun()

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

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

- Check whether there is any thread in ready queue.

Scheduler::Run(Thread*, bool)

```
void Scheduler::Run(Thread *nextThread, bool finishing) {
    Thread *oldThread = kernel->currentThread;

    if (finishing) {  // mark that we need to delete current thread
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

    oldThread->CheckOverflow();  // check if the old thread
                                 // had an undetected stack overflow

    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    CheckToBeDestroyed();  // check if thread we were running
                           // before this one has finished
                           // and needs to be cleaned up

    if (oldThread->space != NULL) {     // if there is an address space
        oldThread->RestoreUserState();  // to restore, do it.
        oldThread->space->RestoreState();
    }
}
```

- Didn't capture the codes for context switch and debugging.
- First, check the value of `finishing`, if `TRUE`, point `toBeDestroyed` to old thread.
- The old thread will be deleted in `CheckToBeDestroyed()`.
    - In either `Scheduler::Run()` or `Thread::Begin()`

Explanation of Purpose and Detail

- Start from triggering the exception handler with SC_Exit system call.
- Then will call the thread's `Finish()` function to begin the terminating procedure.
- Using `Sleep()` as a part of the procedure by sending `finishing = TRUE`.
- Be idle until a thread is found in ready queue.
- Pop out the next thread from ready queue, and run the thread.
- If `finishing` is `TRUE` in `Run()`, it mean the old thread is able to terminate.
- Let `toBeDestroyed` point to the old thread.
- The old thread will be deleted after context switch. (Since the old thread is never used)
  - If next thread it's not a new thread (has done context switch), then the thread will call `CheckToBeDestroyed()` in `Scheduler::Run()` when next thread back to running state.
  - Else if next thread is a new created thread, the `CheckToBeDestroyed()` will be called in `Thread::Begin()` in Thread's constructor.
- That is, the finishing thread is terminated.
  - complete the path of `Running → Terminated`

1-6. Ready → Running

Scheduler::FindNextToRun()

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

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

- First, check whether the ready queue is empty.
- If is not empty, return the first thread in the queue, else return `null`.

Scheduler::Run(Thread*, bool)

```
void Scheduler::Run(Thread *nextThread, bool finishing) {
    Thread *oldThread = kernel->currentThread;

    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

    SWITCH(oldThread, nextThread);


    CheckToBeDestroyed();  // check if thread we were running
                           // before this one has finished
                           // and needs to be cleaned up

    if (oldThread->space != NULL) {     // if there is an address space
        oldThread->RestoreUserState();  // to restore, do it.
        oldThread->space->RestoreState();
    }
}
```

- Justify that should old thread be deleted.
- `Scheduler::Run()` will save the current thread's state in thread control block (if it's user program) and call `SWITCH` to do context switch. (Will explain in next section)
- After context switch, if the thread has done context before, keep running. Otherwise, execute from threadRoot. (Please refer to the SWITCH part)
- Do `CheckToBeDestroyed()`
- If is user program, load back the states.

SWITCH(Thread*, Thread*)

- Noted that, movl a b, means movethe data from b to a, a and b can either be a memory address or a register.
- Recall that we call `StackAllocate()` when create a new thread. So we can tell that,
    - machineState[7] stores (void*) ThreadRoot
    - machineState[2] stores (void*) ThreadBegin
    - machineState[5] stores (void*) func
    - machineState[3] stores (void*) arg
    - machineState[6] stores (void*) ThreadFinish
- When we call `SWITCH(oldThread, nextThread)`, the corresponding code is,

```
push Thread* nextThread
push Thread* oldThread
call SWITCH
```

- Hence, the stack from up to down will be: `nextThread`, `oldThread`, `return address` (esp points to `ra`)
- Moreover, by observing the structure of class `Thread`, we know that:
    - Thread* +0 will get stackTop
    - Thread* +4 will get machineState[0]
    - Thread* +8 will get machineState[1] ...
- The expainantion for the code, write with code comment.

```
SWITCH:
        # temporary save the data in %eax to _eax_save (an arbitary address)
        movl    %eax,_eax_save

        # %esp is the stack pointer, so %esp+4 points to oldThread
        # save the address of oldThread into %eax
        movl    4(%esp),%eax

        # _EBX is define as constant 8 in 'switch.h'
        # As mentioned above, _EBX(%eax) is equal to 8(Thread*)
        # So this row means save the register value into machine[1]
        movl    %ebx,_EBX(%eax)

        # Vice versa...
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)

        # stored into stackTop
        movl    %esp,_ESP(%eax)
```

```
        # get the saved value of %eax
        # and save into  machineState[0]
        movl    _eax_save,%ebx
        movl    %ebx,_EAX(%eax)

        # get return address from stack into %ebx
        movl    0(%esp),%ebx

        # save it into pc storage machineState[7]
        movl    %ebx,_PC(%eax)

        # get the address of nextThread into %eax
        movl    8(%esp),%eax

        # get the value should be put into %eax from machineState[0]
        movl    _EAX(%eax),%ebx

        # save it temporary
        movl    %ebx,_eax_save

        # get the other values from machineState and store into registers
        movl    _EBX(%eax),%ebx
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp

        # restore stack pointer from stackTop
        movl    _ESP(%eax),%esp

        # restore return address into %eax
        movl    _PC(%eax),%eax

        # copy the return address onto the stack
        movl    %eax,4(%esp)

        # get the temp value and store into %eax
        movl    _eax_save,%eax

        # return
        ret
```

- When executing `SWITCH`, it save values in registers into oldThread, and bring out the values in newThread.
- Here may have 2 situations:
    - nextThread hasn't done context switch yet. (Just created)
        - Since it's just created, it will execute `ThreadRoot` saved in `stackTop` to start running a brand-new thread. Besides, `ret` stores above `ThreadRoot`, it'll be executed after `ThreadRoot` finish.

- nextThread has done context switch before
    - Because of doing `movl %esp, _ESP(%eax)`, `stackTop` stored return address in last context switch.
    - That is, this thread will keep running the code after `SWITCH`

(Depends on the previous process state)

- Combining the explanation mentioned above,
  - If the previous state is `New`, run `ThreadRoot` (`ThreadRoot` will be explained later)
  - Else, has done context switch before, keep running the code after `SWITCH`.
- Explanation for `ThreadRoot`

```
ThreadRoot:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   InitialArg  # arg
        call    *StartupPC  # Run (void*) ThreadBegin
        call    *InitialPC  # (void*) func
        call    *WhenDonePC # run (void*) ThreadFinish

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

- `ThreadRoot` is a procedure to start running a new thread.
  - eax points to startup function.
  - edx contains initial argument to thread function.
  - esi points to thread function.
  - edi point to `Thread::Finish()`.
- After running `ThreadRoot` will run:
  - `ThreadBegin`: Simply call `kernel->currentThread->Begin()`.
    - `CheckToBeDestroyed()` is also called in `Thread:Begin()`.
  - `(void*) func`: Equivalent with `ForkExecute(Thread *t).
    - Call `AddrsSpace:load` to load execute file data into memory.
    - Call `AddrSpace::Execute()` to initialize resgisters and goto `kernel->machine->Run()`.
  - `Thread::Finish`: Call `kernel->currentThread->Finish()`.
    - Call `Sleep(TRUE)`, set `finishing = TRUE` to terminate the thread.

for loop in Machine::Run()

```
void Machine::Run() {
    Instruction *instr = new Instruction;  // storage for decoded instruction
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
    }
}
```

- Main part of simulating, consistantly call `OneInstruction()` to read instructions from registers[PCReg], decode, execute, and call `OneTick()` handling interrupt.

Explanation of Purpose and Detail

- Find thread in ready queue by `Scheduler::FindNextToRun()`, and goto `Scheduler::Run()`
- In `Run()`, change the state of nextThread from `Ready` to `Running`, then do context switch.
- After `SWITCH`, can be seperate into two situation
    - Has done context switch before: Return address stored in `stackTop`, so keep running the code after `SWITCH`.
    - Newly created thread: Run `ThreadRoot`, and `ThreadBegin()`, `(void*) func`, `ThreadFinish()` sequentially.
        - `ThreadBegin()`: call `Thread::Begin()`
        - `(void*) func, ForkExecute(Thread* t)`: load file, initialize, call `Machine::Run()`
            - complete the path of `Ready → Running`
        - `ThreadFinish()`: call `Thread:Finish()`

Implementation Part

Multi-Level Feedback Queue definition

```
class MLFQ { // Multi-Level Feedback Queue
public:
   SortedList<Thread*>* L1;
   SortedList<Thread*>* L2;
   List<Thread*>* L3;
   MLFQ();
   ~MLFQ();
   void Append(Thread* thread);
   Thread* RemoveFront();
   void Apply(void (*f)(Thread*));
   bool IsEmpty();
};
```

- We create a new ready class: MLFQ (Multi-Level Feedback Queue)
  - Note that, `MLFQ` has same member function name with `List`, so that we can keep using the origin code related to `readyList`.
  - And change the type of readList in `Scheduler`

```
MLFQ* readyList;
```

Construtor

```
MLFQ::MLFQ() {
    L1 = new SortedList<Thread*>(cmpRemainTime);
    L2 = new SortedList<Thread*>(cmpPriority);
    L3 = new List<Thread*>;
}
```

- Instantiate all of the three queue. L1, L2, use SortedList becuase they have to return the thread with least remain time or least priority.
- The compare functions are design like this:
  - `getRemainTime` is used because there are many place require to get the remain burst time.

```
int cmpRemainTime(Thread* t1, Thread* t2) {
    double time1, time2;
    time1 = getRemainTime(t1);
    time2 = getRemainTime(t2);
    if(time1 == time2) return t1->getID() - t2->getID();
    else return static_cast<int>(time1 - time2);
}
```

```
int cmpPriority(Thread* t1, Thread* t2) {
   if(t1->priority == t2->priority) return t1->getID() - t2->getID();
   else return t2->priority - t1->priority;
}
```

Explanation for getRemainTime()

```
double getRemainTime(Thread* t) {
    if(t->getStatus() == RUNNING)
        return t->apxBurstTime - (t->CPUBurstTime + kernel->stats->totalTicks - t-
>cacheBurstTime);
    else
        return t->apxBurstTime - t->CPUBurstTime;
}
```

- "Current" CPUBurstTime is consist of last updated CPUBurstTime and CPUBurstTime in current running cycle.
    - We only update CPUBurstTime when the thread leaves running state.
- So, the return value of `getRemainTime()` depends on the thread's state.
    - If in running state, current CPUBurstTime has to consider `totalTicks`- `cacheBurstTime`
        - `cacheBurstTime` stores the tick when the thread enters running state.

Explanation for other MLFQ member function

- Explain the functions with comments

```
// Destructor (Prevent memory leaking)
MLFQ::~MLFQ() {
    delete L1;
    delete L2;
    delete L3;
}

// Determine which level of queue should insert by the value of priority
void MLFQ::Append(Thread* t) {
    DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< t->getID() << "] is inserted into queue L[" << t->getLevel() << "]");

    int p = t->priority;

    // Three of the levels are defined properly by nachos
    // Just use the origin member function
    if(p >= 100) L1->Insert(t);
    else if(p >= 50) L2->Insert(t);
    else L3->Append(t);
}
```

```
// RemoveFront() is used to return a thread to be executed.
Thread* MLFQ::RemoveFront() {
    Thread* t;

    // Consider the priority, check whether is empty
    // if not empty, return the first thread in the queue.
    // Also done with the pre-defined member function
    if(!L1->IsEmpty()) t = L1->RemoveFront();
    else if(!L2->IsEmpty()) t = L2->RemoveFront();
    else if(!L3->IsEmpty()) t = L3->RemoveFront();
    else t = NULL;

    DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< t->getID() << "] is removed from queue L[" << t->getLevel() << "]");

    return t;
}

// Extend apply function to all the queues.
void MLFQ::Apply(void (*f)(Thread*)) {
    L1->Apply(f);
    L2->Apply(f);
    L3->Apply(f);
}

// Just check if three queues are all empty()
bool MLFQ::IsEmpty() {
    return L1->IsEmpty() && L2->IsEmpty() && L3->IsEmpty();
}
```

Implementation in class Thread

```
    // Define several attributes to solve the `shortest job first` task.
    int priority;
    double apxBurstTime; // approximate
    int cacheBurstTime;
    int CPUBurstTime;
    int beReadyTime;
    int getLevel();
```

- `priority` is the priority for scheduling, which may be increased by aging.
- `apxBurstTime` = Approximate Remaining Burst Time
- `cacheBurstTime` = The initial value of `kernel->stats->totalTicks` every time the thread go into running state.
- `CPUBurstTime` = Total running time
    - Reset to 0, when entering waiting state (`BLOCKED`)
    - Stop accumulating, when entering ready state (`READY`)
- `beReadyTime` = The time when entering ready state
    - This attribute is for aging.

- `getLevel()`: Return the which queue level is the thread in.

```
// 0-49 for L3, 50-99 for L2, 100-149 for L1
int Thread::getLevel() {
    int level[3] = {3, 2, 1};
    return level[priority/50];
}
```

Thread::setStatus

```
void Thread::setStatus(ThreadStatus st) {
    if(st == RUNNING){ // ready to running -> initialize cacheBurstTime
        cacheBurstTime = kernel->stats->totalTicks;
    }
    else if(st == READY){
        if(status == RUNNING){ // running to ready -> update CPUBurstTime
            CPUBurstTime += kernel->stats->totalTicks - cacheBurstTime;
        else CPUBurstTime = 0;
        beReadyTime = kernel->stats->totalTicks;
    }
    else if(st == BLOCKED){ // running to waiting -> calculate apx, and reset
CPUBurstTime
        CPUBurstTime += kernel->stats->totalTicks - cacheBurstTime;
        // Here should print debug message [D]
        apxBurstTime = 0.5 * CPUBurstTime + 0.5 * apxBurstTime;
    }
    status = st;
}
```

- When change the state ro `RUNNING`
    - Set cacheBurstTime to `kernel->stats->totalTicks()`
- When change the state to `READY`
    - If it's from running to ready
        - Update the `CPUBurstTime` by accumulated `CPUBurstTime` with the running time in current running cycle.
    - Otherwise, reset CPUBurstTime.
        - We clear the `CPUBurstTime` here because 'else' is the waiting state, it's the same to reset time when entering waiting state or leaving.
    - Remember to save the tick when go into ready state.
- When change the state to `BLOCKED`
    - Update the `CPUBurstTime` before compute the `apxBurstTime`
- At the end, set status to `st`.

Implementation for Preempt

```
bool Scheduler::shouldPreempt() {
    Thread* cur = kernel->currentThread;
    Thread* t;
    MLFQ* readyQueue = kernel->scheduler->readyList;
    const int timeQuantum = 100;
    float remainTime, curRemainTime, runningBurstTime;
    Scheduler* a;
    MLFQ* b;
    SortedList<Thread*>* c;
    switch (cur->getLevel()) {
        case 1:
            if(kernel->scheduler->readyList->L1->IsEmpty()) return false;
            t = kernel->scheduler->readyList->L1->Front();
            if(!t) return false;
            remainTime = getRemainTime(t);
            curRemainTime = getRemainTime(cur);
            if(remainTime < curRemainTime)
                return true;
            break;
        case 2:
            if(!kernel->scheduler->readyList->L1->IsEmpty())
                return true;
            break;
        case 3:
            runningBurstTime = kernel->stats->totalTicks - cur->cacheBurstTime;
            if(!readyQueue->L1->IsEmpty() || !readyQueue->L2->IsEmpty() ||
runningBurstTime > timeQuantum)
                return true;
            break;
        default:
            break;
    }
    return false;
}
```

- `shouldPreempt` is called in `Alarm::Callback()` as the spec requires.
- Should preempt or not depends on the level.
    - For level 1, compare the remain time of current thread and the first thread in L1 queue.
        - Since L1 queue is sorted by remain time.
    - For level 2, return true if L1 is not empty
        - Because only threads in L1 can preempt the level 2 thread.
    - For level 3, if either L1 or L2 is not empty, return true. Or if the running time greater than 100 ticks, return true.
    - Else, return false, don't have to be preempt.

Implementation for aging

```
void aging(Thread* t){
    if(kernel->stats->totalTicks - t->beReadyTime > 1500){
        DEBUG(dbgScheduler, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << t->getID() << "] changes its priority from [" << t->priority << "] to [" << min(t->priority + 10, 149) << "]");
        t->priority = min(t->priority + 10, 149);
        t->beReadyTime = kernel->stats->totalTicks;
    }
}

void Scheduler::updatePriority() {
    kernel->scheduler->readyList->Apply(aging);
}
```

- `updatePriority()` is called in `Alarm::Callback()` as the spec requires.
- Apply aging function to all of the queues.
  - `aging()` will update the priority with `min(priority+10, 149)`.
  - Reset `beReadyTime`.

Implementation of "-ep" flag

```
    } else if (strcmp(argv[i], "-e") == 0) { // MP3
        execfile[++execfileNum] = new ExecFile(argv[++i], 0);
        cout << execfile[execfileNum]->name << "\n";
    } else if (strcmp(argv[i], "-ep") == 0) { // MP3
        execfile[++execfileNum] = new ExecFile(argv[++i], atoi(argv[++i]));
```

- Add these code inside `Kernel::Kernel`.
- Build a new class call `ExecFile` to save both filename and priority.
  - The structure of `ExecFile`

```
class ExecFile { // MP3
    public:
    ExecFile(char* n, int p) { name = n; priority = p; }
    ~ExecFile();
    char* name;
    int priority;
};
```

- Next, set the priority inside `Kernel::Exec()`:

```
int Kernel::Exec(ExecFile* info) {
    t[threadNum] = new Thread(info->name, threadNum);
    t[threadNum]->priority = info->priority;
    t[threadNum]->setIsExec();
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum - 1;
}
```

Implementation of DEBUG message

- [A], Print whenever a process is inserted into a queue.
  - call in `MLFQ::Append()`, cause we insert threads into ready queue in this function.
- [B], Print whenever a process is removed from a queue.
  - call in `MLFQ::RemoveFront()`, cause this the function returns the thread which is going to run.
- [C], Print whenever a process changes its scheduling priority.
  - call in `aging()`, cause this is the function update the priority.
  - call in `Thread::setStatus()`, when changing to ready state, cause this is where we reset out priority to `initPriority`.
- [D], Print whenever a process updates its approximate burst time.
  - call in `Thread::setStatus()`, when changing to waiting state, cause this is where we updates `apxBurstTime`.
- [E], Print whenever a context switch occurs.
  - call in `Scheduler::Run()` before `SWITCH`, cause this is the function doing conext switch.
    - This output can be correct because we reset the `CPUBurstTime` when leaving waiting state.

That's all.