1-1. 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()

↓

↓ Before our implementation : remove the front-most thread in the ready list, 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.

↓

SynchList<T>::Append(T)

↓

↓ Append the thread to the back of waiting queue.

↓ The list used in semaphore is List<Thread \*>, and it has nothing to do with this SynchList<T> though. 這一段不要放 可以問一下助教用意

↓

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()

↓

↓ Before our implementation : remove the front-most thread in the ready list, 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()

↓

↓ Before our implementation : remove the front-most thread in the ready list, 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.

■