OS HW03 GROUP 21

Part 1:Trace Code

1. Explain following path

i. New – Ready

Creating thread for execution and allocating resources like stack and address space, initializing the thread's machine state, and disabling interrupts for safe setup. Once ready, the thread is placed in the Ready Queue, waiting for the CPU to execute it.

1) Kernel::ExecAll()

This function executes the programs in execfile sequentially using loop. It calls Exec() to execute them. After finishing all program, call Finish() to clean up and free resources.

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

2) Kernel::Exec()

First, The Exec() function create a new thread object which is a control block for thread including ID, name, status... Second, allocating an address space object for the thread that just created. Last, call the Fork() to start the thread using ForkExecute as the entry and use threadNum to track the number of thread.

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;
    return threadNum-1;
}
```

3) Thread::Fork()

First, declare the pointers to interrupter and scheduler for later use. Call StackAllocate() to allocate stack space and initialize the machine state for the thread. Since moving the thread to the ready state involves modifying the ready queue, it is crucial to disable interrupts first by calling SetLevel(IntOff). Next, move the thread to the ready state by calling scheduler->ReadyToRun(this). Finally, restore the interrupt level to its original state to resume normal interrupt handling after completing the setup.

```
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);
    (void) interrupt->SetLevel(oldLevel);
}
```

4) **Thread::StackAllocate()**

The purpose of this function is to allocate and initialize the thread's stack. First, use AllocBoundedArray to allocate the stack memory. Second. Set stackTop to the top of the stack and put ThreadRoot as the first element. This aims to ensure that when the thread begins, it will start execution at ThreadRoot, which then transfers control to the specific function the thread is meant to run. Last, Initializes the thread's machine state to set the appropriate function as the thread's entry point.

```
void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
//Here, I use x86 for example
#ifdef x86
    // the x86 passes the return address on the stack.  In order for SWITCH()
    // to go to ThreadRoot when we switch to this thread, the return addres
    // used in SWITCH() must be the starting address of ThreadRoot.
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif

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

5) **Scheduler::ReadyToRun()**

After disabling the interrupt, this function will be called. Setting the thread status to ready and appending it to the readyList.

```
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);
}
```

ii. Running – Ready

This occurs when a running thread yields the CPU , allowing other threads to execute. This happens through the Yield() function, which moves the current thread to the readyList and schedules another thread to run.

1) Machine::Run()

Only if program has syscall, it has to switch to kernel mode. Therefore, set the status to UserMode. Using for loop to decode and execute the instruction using OneInstruction(instr). Simulating time progression by calling kernel->interrupt->OneTick().

```
void
Machine::Run()
{
    Instruction *instr = new Instruction;  // storage for decoded instruction

    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread->
             getName();
        cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}
```

2) Interrupt::OneTick()

First, update the simulated time, incrementing totalTicks by UserTick or SystemTick depending on the current mode. Second, disable interrupts using ChangeLevel(IntOn, IntOff) and check for any pending interrupts using CheckIfDue(FALSE). If a context switch is required (yieldOnReturn is TRUE), switching to kernel mode and the current thread yields the CPU by calling Yield(). Last, switch back to oldStatus after finishing.

```
void
Interrupt::OneTick()
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;
    if (status == SystemMode) {
         stats->totalTicks += SystemTick;
         stats->systemTicks += SystemTick;
    } else {
         stats->totalTicks += UserTick;
         stats->userTicks += UserTick;
    }
    DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");

    ChangeLevel(IntOn, IntOff);
    CheckIfDue(FALSE);
    ChangeLevel(IntOff, IntOn);
    if (yieldOnReturn) {
         yieldOnReturn = FALSE;
         status = SystemMode;
         kernel->currentThread->Yield();
         status = oldStatus;
    }
}
```

3) Thread::Yield()

The aim of Yield() is to switch thread to execute. First, disable interrupts to ensure thread-safe operations. Calls FindNextToRun() to get the next thread in the readyList. If a new thread is available, the current thread is moved back to the readyList and performing a context switch by invoking run(nextThread, FALSE).

```
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);
}
```

4) Scheduler::FindNextToRun()

Selects the next thread to execute from the readyList. Returns NULL if the readyList is empty. Otherwise, it removes and returns the thread at the front of the list.

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

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

5) Scheduler::ReadyToRun()

This aim to place a thread into the readyList. Setting the thread status to ready and appending it into readyList.

```
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);
}
```

6) Scheduler::Run()

Here is to perform a context switch between the current thread and the next thread. If the current thread is finishing, it marks it for destruction using toBeDestroyed. Saving the user state of the current thread by calling SaveUserState() and space->SaveState(). Switching the context to nextThread using the assembly routine SWITCH(oldThread, nextThread). After returning from SWITCH, checks and cleans up any completed thread, and restores the user state of the resumed thread.

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    if (finishing) {
          toBeDestroyed = oldThread;
    }
    if (oldThread->space != NULL) {
          oldThread->SaveUserState();
          oldThread->space->SaveState();
    }
    oldThread->CheckOverflow();
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);
    CheckToBeDestroyed();
    if (oldThread->space != NULL) {
          oldThread->RestoreUserState();
          oldThread->space->RestoreState();
    }
}
```

iii. Running – Waiting

Occurs when a thread is unable to proceed because it is waiting for a specific condition, such as I/O completion or a resource becoming available. In this state, the thread voluntarily gives up the CPU by calling a synchronization primitive like Semaphore::P(), which places it in a Waiting

Queue. The thread's status changes to BLOCKED, and it remains in the Waiting state until the condition is met, allowing the scheduler to switch to another thread.

1) **SynchConsoleOutput::PutChar()**

   This function is to put a character on the console. Because there can't be two threads doing PutChar simultaneously, it has to lock first. Then, calling consoleOutput->PutChar(ch) to output the character. Waiting for the operation to complete by calling waitFor->P(), and releasing the lock after the operation is done.

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

2) **Semaphore::P()**

   Perform a wait operation on a semaphore. First, disable interrupts to ensure atomicity. If value equal to 0, the thread is added to the semaphore's queue and then sleeps by calling currentThread->Sleep(FALSE). Decrement the semaphore value once it is greater than 0. Then, restore the previous interrupt state.

```
void
Semaphore::P()
{
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {
          queue->Append(currentThread);
          currentThread->Sleep(FALSE);
    }
    value--;
    (void) interrupt->SetLevel(oldLevel);
}
```

3) **List<T>::Append()**

   Define the append for the list. If the list is empty, the new element becomes the first and last element. Otherwise, the new element is added after the current last element. Last, increments the number of items in the list.

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

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

4) **Thread::Sleep()**
First, set the thread's status to BLOCKED. Continuously calls kernel->scheduler->FindNextToRun() to find a thread from the ready list. If no thread is found, the system idles. Otherwise, if it found a thread, it calls Run() to perform a context switch.

```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;
    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Sleeping thread: " << name);
    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();  // no one to run, wait for an interrupt
    }
    kernel->scheduler->Run(nextThread, finishing);
}
```

5) **Scheduler::FindNextToRun()**
This is to fetches the next thread to run from the ready list. If the ready list is empty, it returns NULL. Otherwise, it removes and returns the front thread of the ready list.

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

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

6) **Scheduler::Run()**
This performs a context switch between the current thread and the next thread. If the current thread is finishing, it marks it for destruction using toBeDestroyed. Saving the user state of the current thread by calling SaveUserState() and space->SaveState(). Switching the context to

nextThread using the assembly routine SWITCH(oldThread, nextThread). After returning from SWITCH, checks and cleans up any completed thread, and restores the user state of the resumed thread.

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    if (finishing) {
         toBeDestroyed = oldThread;
    }
    if (oldThread->space != NULL) {
         oldThread->SaveUserState();
         oldThread->space->SaveState();
    }
    oldThread->CheckOverflow();
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);
    CheckToBeDestroyed();
    if (oldThread->space != NULL) {
         oldThread->RestoreUserState();
         oldThread->space->RestoreState();
    }
}
```

iv. Waiting – Ready

When the semaphore becomes available, the blocked thread waiting for it is woken up, marked as READY, and moved to the ready list.

1) **Semaphore::V()**

Since the operation must be atomic, interrupts are first disabled to prevent race conditions. Next, it checks if there are any threads blocked, waiting for the semaphore. If so, it wakes up the first thread in the waiting queue and invokes ReadyToRun(). Afterward, the semaphore value is incremented to signal the availability. Finally, interrupts are re-enabled.

```
void Semaphore::V()
{
    Interrupt *interrupt = kernel->interrupt;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    if (!queue->IsEmpty()) {
      kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

2) **Scheduler::ReadyToRun()**

Since the readyList is shared between threads, it is important to disable interrupts to prevent other threads or interrupts from modifying the list or thread status. After disabling interrupts, the thread's status is set to READY to indicate that it is now eligible to run. Finally, the thread is appended to the readyList, where it will remain until the scheduler selects it to run.

```
void Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

v. **Running – Terminated**

When a thread finishes execution, deferred cancellation is used for safe clean up. The thread is first blocked, then marked for deletion. After a context switch to another thread, the finished thread is cleaned up.

1) **ExceptionHandler(ExceptionType: case SC_Exit)**

Print the return value and invoke the Finish() routine to finish the current thread.

```
case SC_Exit:
     val=kernel->machine->ReadRegister(4);
     cout << "return value:" << val << endl;
     kernel->currentThread->Finish();
     break;
```

2) **Thread::Finish ()**

Interrupts are disabled to ensure atomicity. Then, the Sleep routine is invoked with the argument TRUE, indicating that the thread has finished execution.

```
void Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
    Sleep(TRUE);
}
```

3) **Thread::Sleep ()**

Since the current thread has finished, it relinquishes the CPU. First, its status is set to BLOCKED to indicate that it is not ready to run. Then, the function attempts to find the next thread to execute. If the ready list is empty, the CPU is idled until the next I/O interrupt occurs. Once another thread becomes available, do the context switche to the next thread, with the finishing argument indicating that the current thread has completed execution.

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

4) **Scheduler::FindNextToRun ()**

If the ready list is empty, it returns NULL, indicating that no threads are available to run. Otherwise, it removes the front thread from the list and returns it as the next thread to execute.

```
Thread * Scheduler::FindNextToRun ()
{
    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

5) Scheduler::Run ()

In multithreaded programming, deferred cancellation helps control the safe and predictable termination of threads. When a thread finishes, it is first marked for deletion by assigning it to toBeDestroyed, which indicates that it will be cleaned up later. Then, a context switch is performed to the next thread using the SWITCH routine. After returning to the old thread, it is no longer running, so it is safely cleaned up through the CheckToBeDestroyed() function.

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

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

6) Scheduler::CheckToBeDestroyed()

This function deletes the thread that has finished execution.

```
void
Scheduler::CheckToBeDestroyed()
{
    if (toBeDestroyed != NULL) {
        delete toBeDestroyed;
    toBeDestroyed = NULL;
    }
}
```

vi. Ready – Running

When a thread in the ready queue is selected to execute, a context switch occurs, saving the state of the current thread and loading the state of the selected one. The process then varies depending on the previous state of the new thread, but it eventually reaches the loop in Machine::Run() to execute instructions.

1) Scheduler::FindNextToRun ()

When there are threads in the ready queue, this function removes the front thread from the list and returns it as the next thread to execute.

```
Thread * Scheduler::FindNextToRun ()
{
    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

2) **Scheduler::Run ()**

Before switching to the next thread, the CPU registers and address space of the current thread are saved. Then, the next thread's status is set to running, and a context switch is performed. After the switch, the CPU registers and address space of the old thread are restored.

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

    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);

    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

3) **SWITCH(Thread*, Thread*)**

Declared in thread.h and implemented in switch.s.
The following code demonstrates context switching in an x86 environment.

Step 1: save the state of the old thread
The process saves the current state of the old thread into its corresponding data structure, machineState. First, the original value in the eax register is temporarily saved. Next, the pointer to the old thread is moved into the eax register. Then, the values of all relevant registers are sequentially saved into the machineState of the old thread. Afterward, the original value of eax is restored and stored in machineState[EAX]. Finally, the return address from the stack is retrieved and saved into machineState[PC] to preserve the thread's execution context.

```
SWITCH:
        movl    %eax,_eax_save    # save the original value of eax register
        movl    4(%esp),%eax      # move pointer to oldthread in stack into eax
        movl    %ebx,_EBX(%eax)   # machineState[EBX] = value of register ebx
        movl    %ecx,_ECX(%eax)   # same as above
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)   # save stack pointer to machineState[ESP]
        movl    _eax_save,%ebx    # get the original value of eax
        movl    %ebx,_EAX(%eax)   # store it to machineState[EAX]
        movl    0(%esp),%ebx      # get return address from stack into ebx
        movl    %ebx,_PC(%eax)    # save it into the machineState[PC]
```

Step2: reload the state of the new thread
The process reloads the state of the new thread from its machineState into the corresponding registers. First, the pointer to the new thread is moved into the eax register. Next, the value in machineState[EAX] is temporarily saved. Then, the values of all relevant registers in the machineState of the new thread are sequentially reloaded into the corresponding registers. After that, the return address stored in machineState[PC] is copied onto the stack and the original value of eax is restored. Finally, return to the program counter by jumping to the address stored in the return address.

```
SWITCH:
        movl    8(%esp),%eax      # move pointer to newthread in stack into eax
        movl    _EAX(%eax),%ebx   # get new value for eax into ebx
        movl    %ebx,_eax_save    # save it
        movl    _EBX(%eax),%ebx   # retore registers value from machineState
        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
        movl    _PC(%eax),%eax    # restore return address into eax
        movl    %eax,4(%esp)      # copy over the ret address on the stack
        movl    _eax_save,%eax    # restore the original value of eax
        ret                       # return to program counter
```

4) **ret**

After context switch, it will return to the return address. For different scienario, the return address is different.

a) **New – Ready**

For the first execution, the machineState[PC] is set as ThreadRoot() in Thread::StackAllocate(). The routine ThreadRoot() is declared in thread.h and implemented in switch.s.

```
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    machineState[PCState] = (void*)ThreadRoot;
}
```

Step 1:
The function save the current value of the ebp register to the stack. Afterward, the ebp

register is updated with the current value of the esp register.

```
ThreadRoot:
  pushl  %ebp
  movl   %esp,%ebp
```

Step 2: Push the initial argument
The initial argument is assigned in Thread::StackAllocate().

```
ThreadRoot:
  pushl  InitialArg
```

```
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    machineState[InitialArgState] = (void*)arg
}
```

Step 3: Call StartupPC
When ThreadRoot() calls the StartupPC, it jumps to the ThreadBegin function, which was assigned in Thread::StackAllocate().

```
ThreadRoot:
  call   *StartupPC
```

```
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
machineState[StartupPCState] = (void*)ThreadBegin;
}
```

Inside ThreadBegin, the Thread::Begin() method is invoked. The purpose of Thread::Begin() is to perform two main tasks:: first, deallocate the previously running thread if it finished; second, enable interrupts to get time-sliced

```
static void ThreadBegin() { kernel->currentThread->Begin(); }
```

```
void Thread::Begin ()
{
    kernel->scheduler->CheckToBeDestroyed();
    kernel->interrupt->Enable();
}
```

Step 4: Call InitialPC
When ThreadRoot() calls the InitialPC, it jumps to the function (func) that was assigned in Thread::StackAllocate().

```
ThreadRoot:
  call   *InitialPC
```

```
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
machineState[InitialPCState] = (void*)func;
}
```

In this case, the func is passed from Kernel::Exec, where it is set to ForkExecute.

```
int Kernel::Exec(char* name)
{
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute,
                       (void *)t[threadNum]);
}
```

```
void Thread::Fork(VoidFunctionPtr func, void *arg)
{
    StackAllocate(func, arg);
}
```

The ForkExecute function is responsible for loading the user program and then executing it by calling AddrSpace::Execute.

```
void ForkExecute(Thread *t)
{
    if ( !t->space->Load(t->getName()) ) {
        return;
    }
    t->space->Execute(t->getName());
}
```

Within AddrSpace::Execute, the machine starts running the user-level program by invoking kernel->machine->Run().

```
void AddrSpace::Execute(char* fileName)
{
    kernel->currentThread->space = this;
    this->InitRegisters();
    this->RestoreState();
    kernel->machine->Run();
    ASSERTNOTREACHED();
}
```

b) **Running – Ready**

If the thread's previous process state was Running, it must have previously triggered a SWITCH, which saved its machineState[PC]. When the thread transitions back to Running, the ret instruction in SWITCH will use the saved machineState[PC] to return to the loop in Machine::Run(). This return address typically points to the instruction immediately following the point where the context switch occurred.

c) **Waiting – Ready**

If the thread's previous process state was Waiting, it must have previously triggered a SWITCH to wait for an I/O operation or a synchronization event, which saved its machineState[PC]. Once the thread is switched back to Running, the ret instruction in SWITCH will use the saved machineState[PC] to return to the loop in Machine::Run(). This return address typically points to the instruction immediately following the point where the context switch occurred.

5) **Machine::Run()**

Regardless of the thread's previous state (New, Running, or Waiting), the ret instruction in the SWITCH function will eventually return control to Machine::Run(). For the New state, execution will start from the very beginning, while for Running or Waiting states, execution will resume from the instruction immediately following the point where the context switch occurred.

```
void Machine::Run()
{
    Instruction *instr = new Instruction;
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}
```

Part 2:Contribution

1. **Describe details and percentage of each member's contribution.**

   313551047 陳以瑄, contribution (50%):
   Trace code: (4) Waiting – Ready (5) Running – Terminated (6) Ready - Running

   111550150 俞祺譯, contribution (50%):
   Trace code: (1) New – Ready (2) Running – Ready (3) Running - Ready