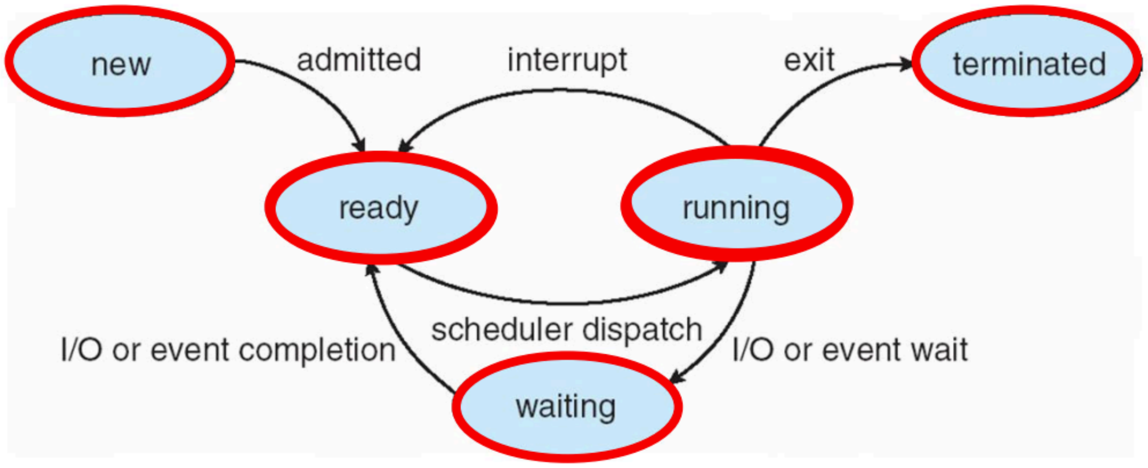
**Course :** Operating Systems **Instructor:** 黃能富

**Student name:** Tuguldur Tserenbaljir **Assignment:** Final Project Report

**Student ID:** 109006271

## Part 1) Trace Code

The following diagram can be used to describe the how nachos manages the lifecycle of a process (or thread) (Figure 1.1.1 Diagram of Process State).



**(Figure 1.1.1 Diagram of Process State)**

### 1-1. New→Ready

This transition prepares a thread for execution by allocating necessary resources and positioning it in the ready queue. It signifies the system's readiness to manage new tasks efficiently. This path involves initializing and preparing a new thread to be ready for execution:

**1.1.1) UserProgKernel::InitializeAllThreads():** This function is responsible for initializing all the threads that are specified to run within the Nachos system. It orchestrates the creation and preparation of multiple threads, ensuring they are properly set up to be executed by the system.

The transition from "New" to "Ready" is crucial as it marks the threads' readiness for execution after all necessary initializations are completed.

```
void
UserProgKernel::InitializeAllThreads()
{
    for (int i = 1; i <= execfileNum; i++){
        // cout << "execfile[" << i << "]: " << execfile[i] <<
" start " << endl;
        int a = InitializeOneThread(execfile[i],
threadPriority[i], threadRemainingBurstTime[i]);
        // cout << "execfile[" << i << "]: " << execfile[i] <<
" end "<< endl;
    }
    // After InitializeAllThreads(), let main thread be
terminated that we can start to run our thread.
    currentThread->Finish();
    // kernel->machine->Run();
}
```

**1.1.2) *UserProgKernel::InitializeOneThread(char, int, int)**:** This function initializes a single thread. Parameters might include configuration or initial state information.

```
int
UserProgKernel::InitializeOneThread(char* name, int priority,
int burst_time)
{
    //<TODO>
    // When each execfile comes to Exec function, Kernel helps
to create a thread for it.
```

```
    // While creating a new thread, thread should be
initialized, and then forked.
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void
*)t[threadNum]);
    //<TODO>

    threadNum++;
    return threadNum - 1;
}
```

The process of creating a new thread involves several intricate steps. Initially, an address space is allocated for the thread, providing the necessary memory where the thread's program will execute. This allocation is essential for setting up the thread to operate independently, transitioning its status from **'New'** to **'Ready**.'

Subsequently, the **Fork** function is invoked to start the thread's execution. This function determines the initial point in the program where the thread will begin and sets up a separate execution stack, effectively duplicating necessary execution details such as the program counter.

Additionally, the process involves a counter called **threadNum** to keep track of how many threads have been initialized, incrementing this counter with each new thread.

**1.1.3) *Thread::Fork(VoidFunctionPtr, void)**:** Creates a new thread. The function passed to Fork will be the entry point of the new thread. This is where the thread begins execution.

```
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);
}
```

The process of preparing a thread for execution involves several key steps. Initially, a stack is allocated which the thread will use to store essential information, such as return addresses from functions. To ensure this setup is uninterrupted, the system's interrupts are temporarily disabled, creating a **"Do Not Disturb"** environment necessary for configuration reasons.

Once the thread is fully set up, it is added to the ready queue, indicating that it is waiting its turn to run. Finally, interrupts are re-enabled, allowing normal system activities to resume, ensuring the thread can eventually proceed with execution when selected by the scheduler.

**1.1.4) *Thread::StackAllocate(VoidFunctionPtr, void)**:** Allocates a stack for the new thread and sets up its initial CPU context. This is essential for the thread to run independently.

```
void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
//rest of code
}
```

The process of setting up a thread's stack involves several detailed steps tailored to the specific needs of the thread and the underlying computer architecture. Initially, memory for the stack is allocated based on a predefined size (**StackSize**), ensuring sufficient space for the thread's operations and structured as an array of integers to represent the stack's contents. The top of the stack (**stackTop**) is then adjusted according to the specific architecture (**like x86, SPARC, etc.**), leaving room for essential items such as return addresses and including a **STACK_FENCEPOST** at the beginning as a safety marker for error checking and overflow prevention.

The stack is further initialized with the execution context: it sets up pointers to critical functions like **ThreadRoot**, which is the entry function when the thread starts, **ThreadBegin** for additional setup, func as the main function the thread executes, arg as the argument to this function, and

ThreadFinish for cleanup after func completes. This structured setup ensures that when the thread is activated, it has a clear execution path and all necessary resources arranged to facilitate smooth operation.

1.1.5) *Scheduler::ReadyToRun(Thread)**: Sets a thread up to be next in line for running. It ensures that when the system is ready to give the thread some CPU time, the thread is ready to go. This function is crucial for managing multiple threads, allowing the system to handle many tasks efficiently.

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

    Statistics* stats = kernel->stats;
    //<TODO>
    // After inserting Thread into ReadyQueue, don't forget to
reset some values.
    // Hint: L1 ReadyQueue is preemptive SRTN(Shortest
Remaining Time Next).
    // When putting a new thread into L1 ReadyQueue, you need
to check whether preemption or not.
    //<TODO>
    // readyList->Append(thread);
}
```

**1-2. Running→Ready**

Implementing preemptive multitasking, this transition allows the system to manage CPU time among multiple threads fairly, enhancing responsiveness and resource utilization. This path is for preempting a running thread, making it ready again for the CPU.

**1.2.1) Machine::Run():** continuously executing instructions of the currently active thread, effectively mimicking the operation of a real CPU. It manages the system's time properly and periodically checking for interrupts to ensure efficient allocation and use of system resources while maintaining proper attention to other threads and operations.

```
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();
    }
}
```

Additionally**, Machine::Run** incorporates debugging support. As it allows for detailed inspection and analysis of the execution process at various stages, for the purpose of understanding and troubleshooting of the simulated system.

**1.2.2) Interrupt::OneTick():** advancing time and managing system responsiveness through interrupt handling and supporting preemptive multitasking. This function is essential for simulating a realistic operating system environment where multiple processes or threads share CPU time.

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

// advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
    stats->systemTicks += SystemTick;
    } else {                    // USER_PROGRAM
    stats->totalTicks += UserTick;
    stats->userTicks += UserTick;
    }
    DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");

// check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff); // first, turn off interrupts
                // (interrupt handlers run with
                // interrupts disabled)
    CheckIfDue(FALSE);       // check for pending interrupts
    ChangeLevel(IntOff, IntOn); // re-enable interrupts
    if (yieldOnReturn) {     // if the timer device handler
asked
                    // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode;         // yield is a kernel
routine
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

If a timer interrupt or another interrupt handler has set **yieldOnReturn** to true, indicating a need for a context switch (usually because the current thread's time slice is complete or a higher priority thread needs to run), the function facilitates this.

It temporarily sets the machine status to **SystemMode**, triggers the current thread to yield its CPU time **(kernel->currentThread->Yield())**, and restores the original machine status after the yield.

**1.2.3) Thread::Yield():** function facilitates voluntary CPU release, allowing threads to relinquish their control over CPU resources willingly. Which is particularly useful in scenarios when threads are waiting for resources or when simple cooperative multitasking is being implemented. By disabling interrupts during this process, the function ensures that the transition from the current thread to the next occurs smoothly and without interruption.

Additionally, Thread::Yield prepares both the current and the next threads for the context switch, thereby preserving the execution order and efficiency of thread operations within the system.

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

    ASSERT(this == kernel->currentThread);

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

    //<TODO>
    // 1. Put current_thread in running state to ready state
    // 2. Then, find next thread from ready state to push on
running state
    // 3. After resetting some value of current_thread, then
context switch
    kernel->scheduler->Run(nextThread, finishing);
    //<TODO>

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

**1.2.4) Scheduler::FindNextToRun():** Picks the next thread to run from the ready queue.

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

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

    //<TODO>
    // a.k.a. Find Next (Thread in ReadyQueue) to Run
    //<TODO>
}
```

Function begins by ensuring that interrupts are disabled (ASSERT(kernel->interrupt->getLevel() == IntOff)), a crucial step to guarantee selecting the next thread to run without any interruptions.

The function then checks the state of the **ready** queue (**readyList**). If the queue is **empty**, it returns **NULL** which means no threads available to run amd the system might need to idle.

In contrast, if the queue is **not empty**, the function proceeds to remove and return the front thread from the queue (**readyList->RemoveFront()**). This thread is typically the next to run, following a First-Come, First-Served (FCFS) scheduling policy or a similar method, ensuring a straightforward and fair approach to thread scheduling.

**1.2.5) *Scheduler::ReadyToRun(Thread)**:** Same as above in 1-1 New->Ready.

**1.2.6) *Scheduler::Run(Thread, bool)**:** managing thread execution and transitions within Nachos. It ensures that threads can be switched safely and efficiently, maintaining the continuity and order of processes within the environment.

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

```
//  cout << "Current Thread" <<oldThread->getName() << "
Next Thread"<<nextThread->getName()<<endl;

    ASSERT(kernel->interrupt->getLevel() == IntOff);

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

## 1-3. Running→Waiting

This shift is for I/O-bound processes, allowing the system to utilize CPU resources efficiently while a thread waits for I/O operations to complete. It demonstrates the system's ability to handle asynchronous events effectively. Occurs when a thread performs I/O and must wait for the operation to complete.

**1.3.1) ExceptionHandler(ExceptionType) case SC_PrintInt:** The ExceptionHandler function in the Nachos operating system simulation handles different types of exceptions that can occur during the execution of user programs (including system calls). This function helps in transitioning threads from the "**Running**" to "**Waiting**" state. Especially during operations like console output.

```
        //rest of code
        case SC_PrintInt:
            // DEBUG(dbgMLFQ, "Print Int");
            val=kernel->machine->ReadRegister(4);
            kernel->synchConsoleOut->PutInt(val);
            DEBUG(dbgMLFQ, "\n");
            return;
```

```
           //rest of code
```

**1.3.2) SynchConsoleOutput::PutInt() & ConsoleOutput::PutChar(char):** Outputs a character or integer to the console, which is an I/O operation.

```
SynchConsoleOutput::SynchConsoleOutput(char *outputFile)
{
    consoleOutput = new ConsoleOutput(outputFile, this);
    lock = new Lock("console out");
    waitFor = new Semaphore("console out", 0);
}
```

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

**SynchConsoleOutput(char *outputFile)** and **SynchConsoleOutput::PutChar(char ch)** methods work together to manage synchronized console output in a multi-threaded environment efficiently. The constructor sets up a ConsoleOutput object to handle the low-level operations of writing to the console or a specified output file, incorporates a Lock to ensure exclusive access during output operations, and initializes a **Semaphore** named **waitFor** to **zero** for synchronizing the completion of these operations. When a thread executes the **PutChar** method, it first acquires the lock to prevent other threads from writing simultaneously. It then sends a character to the console, waits on the semaphore to ensure the operation completes without interference, and releases the lock once done, maintaining order and data integrity across multiple threads.

**1.3.3) Semaphore::P():** used to manage access to shared resources by multiple threads, ensuring that operations involving these resources are performed without conflicts.

```
void
Semaphore::P()
{
    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);
        // cout << "Run to Waiting over" << endl;
    }
    value--;            // semaphore available, consume its
value

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

When a thread, such as Thread A, needs access to a shared resource like a printer, it first disables interrupts to prevent other threads from disrupting its operation during the resource availability check. If the printer is not available (indicated by a semaphore value of 0),

Thread A adds itself to a waiting queue and goes into sleep mode, effectively pausing its operations without consuming CPU resources. Once the printer becomes available (semaphore value changes to 1), Thread A is awakened, decrements the semaphore to claim the printer, and proceeds with its printing task.

After successfully securing the printer, Thread A re-enables interrupts to allow other threads to continue their normal operations, ensuring smooth and orderly resource management within the system.

**1.3.4) SynchList<T>::Append(T):** Used for queueing tasks or data in synchronized lists.

```
template <class T>
void
SynchList<T>::Append(T item)
{
    lock->Acquire();        // enforce mutual exclusive access
to the list
    list->Append(item);
    listEmpty->Signal(lock);    // wake up a waiter, if any
    lock->Release();
}
```

The function in the Nachos simulation implements a synchronized method to append items to a shared list, ensuring that operations are thread-safe. Initially, the function acquires a lock **(lock->Acquire())**, securing exclusive access to the list to prevent multiple threads from appending simultaneously, which could cause data corruption or inconsistent states.

Once exclusive access is confirmed, the specified item is appended to the end of the list **(list->Append(item))**, under the protection of the lock. If any threads are waiting for items to be added, the function then signals one waiting thread **(listEmpty->Signal(lock))** to inform it that an item is now available.

Finally, the lock is released **(lock->Release())**, allowing other operations on the list to proceed, thus ensuring that the list maintains consistency and correctness even as other threads interact with it.

**1.3.5) Thread::Sleep(bool):**

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

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Sleeping thread: " << name << ", ID: " <<
ID);

    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) ==
NULL)
        kernel->interrupt->Idle();  // no one to run, wait for
an interruptd
    //rest of code
```

This function manages the transition of a thread from "**Running**" to "**Waiting**" or "**Terminated**", depending on whether the thread is temporarily blocked or finishing permanently. It begins by ensuring atomic operations through disabled interrupts, a necessary condition for safe context switching. The thread sets its own status to "**BLOCKED**", and then attempts to find another thread to run using **kernel->scheduler->FindNextToRun()**.

If no other threads are ready, the CPU enters an idle state via **kernel->interrupt->Idle()**, waiting for an interrupt that signals a thread has become ready. Once a ready thread is found, the scheduler is invoked to switch the current thread out with kernel->scheduler->Run(nextThread, finishing), facilitating a seamless transition of CPU control. This method ensures that system resources continue to be utilized efficiently, supporting robust multitasking by properly managing threads that are no longer able to proceed due to completion or awaiting resources.

**1.3.6) Scheduler::FindNextToRun() & *Scheduler::Run(Thread, bool)**:** Same as above in 1-2 Running->Ready.

**1-4. Waiting→Ready**

This transition marks the completion of awaited events (like I/O operations), moving threads back to the ready queue to ensure that they continue their execution at the next opportunity. It reflects the system's capability to manage dependencies and synchronize operations. This path reactivates a thread after an I/O operation completes:

**1.4.1) Semaphore::V():** Similar as above in 1-3 Running->Waiting. This function manages the release of a resource that might have been occupied by another thread. The function ensures atomic operations by first disabling interrupts. If any threads are waiting for the resource (i.e., in the semaphore's queue), it transitions one of those threads from "Waiting" to "Ready" using the Scheduler::ReadyToRun() method. After potentially activating a waiting thread, the semaphore's value is incremented to reflect the increased availability of the resource. Lastly, interrupts are turned on again so that threads can run their normal course without any disruption and without causing unfairness in the system.

**1.4.2) *Scheduler::ReadyToRun(Thread)**:** Same as above in 1-1 New->Ready.

**1-5. Running→Terminated**

This final transition in a thread's lifecycle is vital for releasing resources and cleaning up after a thread's completion, ensuring system stability and preventing resource leaks. This path covers the end of a thread's lifecycle:

**1.5.1) ExceptionHandler(ExceptionType) case SC_Exit:** Similar as above in 1-3 Running->Waiting. It is a case where it manages the termination of a program This includes handling debug information, reading the program's exit value, logging this value for clarity, and ultimately terminating the current thread. This ensures that the program exits in an orderly fashion, releasing its resources and providing feedback on its termination status.

**1.5.2)Thread::Finish():** Cleans up the thread-specific resources.

```
static void ThreadFinish()    {
kernel->currentThread->Finish(); }
```

The static method manages a thread's transition from "Running" to "Terminated" state, typically triggered by task completion or an Exit system call.

By invoking kernel->currentThread->Finish(), it directly terminates the active thread.

Thread::Finish() confirms it is executed by the thread in control. It releases all associated resources for a clean exit, removes the thread from the scheduler's active list to free up CPU time.

Thus, if no other threads are ready, it places the system in an idle state until activity resumes.

**1.5.3)Thread::Sleep(bool):** Same as above in 1-3 Running->Waiting.

**1.5.4)Scheduler::FindNextToRun() & *Scheduler::Run(Thread, bool)**:** Same as above in 1-2 Running->Ready.

**<u>1-6. Ready→Running</u>**

Essential for the scheduler, this transition picks out which thread should be executed next, using the scheduler's approach to manage the workload and identify important processes.

**1.6.1)Scheduler::FindNextToRun():** Same as above in 1-2 Running->Ready.

**1.6.2)*Scheduler::Run(Thread, bool)**:** Same as above in 1-2 Running->Ready.

**1.6.3)SWITCH(Thread, Thread)**:** Assembler routine in switch.s that saves the state of the current thread and loads the state of the new thread.  The SWITCH routine is designed to save the state of the current thread (thread t1) and restore the state of the thread to be executed next (thread t2). This involves manipulating processor registers and the stack to ensure that the transition between threads is seamless and that each thread resumes execution correctly from where it left off.

```
/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->             thread *t2
**      4(esp)  ->             thread *t1
**       (esp)  ->             return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
*/
```

**1.6.4) Machine::Run():** Same as in 1-2 Running->Ready

**Part 2) Implementation**

**2.1) /userprog/kernel.cc ⇒ UserProgKernel::UserProgKernel(int argc, char **argv)**
(Add a command line argument -epb for nachos to initialize priority of process from project guide 2.3)

```
       //<TODO>
        // Get execfile & its priority & burst time from argv, then save
them.
        else if (strcmp(argv[i], "-epb") == 0) {
            execfile[++execfileNum] = argv[++i];
            int priority = atoi(argv[++i]);
            int burstTime = atoi(argv[++i]);
            cout << "Creating thread for executable: " <<
execfile[execfileNum] << " with priority " << priority << " and burst
time " << burstTime << endl;
        }
        //<TODO>
```

The function scans command-line arguments for the -epb flag, indicating subsequent arguments specify an executable file, its priority, and burst time following the -epb flag, the next three arguments are:

- **Executable** Path/Name: The location or name of the executable file.
- **Priority**: Numerical value specifying the priority level for scheduling.
- **Burst Time:** Duration the thread should run in a single burst.

These are sequentially read and the index i is incremented to move through the arguments.

- **Executable Name**: Stored in an array **execfile** at a position indexed by **execfileNum**, which counts the number of executables processed.

**Example: ./nachos -epb testprog 5 100**

Recognize -epb and read testprog as the executable, 5 as its priority, and 100 as the burst time.

**2.2) /userprog/kernel.cc ⇒ void ForkExecute(Thread *t)**

```
            // cout << "Thread: " << (void *) t << endl;
    //<TODO>
    // When Thread t goes to Running state in the first time, its file
should be loaded & executed.
    // Hint: This function would not be called until Thread t is on
running state.
    DEBUG(dbgThread, "Entering ForkExecute for Thread ID " << t->getID()
<< " at system time " << kernel->stats->totalTicks);
    t->space->Load(t->getName());
    t->space->Execute(t->getName());
    //<TODO>
```

- Loads and executes the program associated with a thread when it first enters the running state.
- Calls **t->space->Load(t->getName())** to prepare the thread's address space with the program's executable content.

- Invokes **t->space->Execute(t->getName())** to start executing the loaded program.

**2.3) /userprog/kernel.cc ⇒ int UserProgKernel::InitializeOneThread(char* name, int priority, int burst_time)**

```
            // cout << "Thread: " << (void *) t << endl;
    //<TODO>
    // When Thread t goes to Running state in the first time, its file
should be loaded & executed.
    // Hint: This function would not be called until Thread t is on
running state.
    DEBUG(dbgThread, "Entering ForkExecute for Thread ID " << t->getID()
<< " at system time " << kernel->stats->totalTicks);
    t->space->Load(t->getName());
    t->space->Execute(t->getName());
    //<TODO>
```

- Sets the thread's priority **(t[threadNum]->setPriority(priority))** to determine its scheduling priority
- Configures the burst time **(t[threadNum]->setRemainingBurstTime(burst_time))**
- Loads the executable into memory by calling **t[threadNum]->space->Load(name)**
- Forks the thread setting **ForkExecute** as the entry point to begin execution.

**2.4)threads/thread.h⇒ Finish Get/Set values in class**

```
    void setPriority(int priority) { Priority = priority; }
    int getPriority() const { return Priority; }

    void setWaitTime(int waitTime) { WaitTime = waitTime; }
    int getWaitTime() const { return WaitTime; }

    void setRemainingBurstTime(int remainingBurstTime) {
RemainingBurstTime = remainingBurstTime; }
    int getRemainingBurstTime() const { return RemainingBurstTime; }
    void setRunTime(int runTime) { RunTime = runTime; }
    int getRunTime() const { return RunTime; }

    void setRRTime(int rrTime) { RRTime = rrTime; }
    int getRRTime() const { return RRTime; }
    //<TODO>
```

Declare thread operations in class Thread **(thread.h)**. So that later on they can used inside the function that we implement.

**2.5) threads/thread.cc⇒ void Thread::Yield ()**

```
       //<TODO>
    // 1. Put current_thread in running state to ready state
    if (status == RUNNING) {
        setStatus(READY); // Update the thread's status to READY
    }

    // 2. Then, find next thread from ready state to push on running
state
    nextThread = kernel->scheduler->FindNextToRun();

    //nextThread = scheduler->FindNextToRun();
    // 3. After resetting some value of current_thread, then context
switch
    if (nextThread != NULL) {
        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }

    //<TODO>
```

- Temporarily frees the CPU from the current running thread to allow other threads the opportunity to run.
- Saves the current interrupt level and disables interrupts **(IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);)** to ensure atomicity

- Checks if the current thread status is **RUNNING**.
- Sets the thread's status to **READY** using **setStatus(READY);,** indicating it is ready to execute but not currently using the CPU.
  Finds the next thread to run using **nextThread = kernel->scheduler->FindNextToRun();.**
- Places the current thread back on the ready list to be scheduled later: **kernel->scheduler->ReadyToRun(this);.**
- Switches to the next thread if one is available, Restores the previous interrupt level after yielding

## 2.6) threads/thread.cc⇒ void Thread::Sleep ()

```
       // 1. Update RemainingBurstTime
    if (finishing) {
        RemainingBurstTime = 0;  // Example, could be based on actual
time spent
        DEBUG(dbgThread, "Finishing thread: " << name << "
RemainingBurstTime set to 0");
    }
    // 2. Reset some value of current_thread, then context switch
    kernel->currentThread->setWaitTime(0);
    kernel->scheduler->Run(nextThread, finishing);
    //<TODO>
```

- Manages the transition of the current thread to a **BLOCKED** state, with additional handling for threads that are finishing.
- If **finished** sets the **remaining burst time** to 0 because the job is done.
- Also, set the **wait time** to **0**, preparing the thread for a clean state when it next becomes active.
- Invokes a context switch to move to the next available thread

**2.7) threads/scheduler.cc ⇒int getQueueLevel() ,int CompareThreadsByRemainingTime(Thread* a, Thread* b)**

**2.8) threads/scheduler.cc ⇒Construct and Deconstruct L1,L2,L3**

```
Scheduler::Scheduler()
{
    readyListL1 = new
SortedList<Thread*>(CompareThreadsByRemainingTime); // SRTN
    readyListL2 = new List<Thread *>(); // FCFS
    readyListL3 = new List<Thread *>(); // Round Robin
}
```

```
Scheduler::~Scheduler()
{
    //<TODO>
    // Remove L1, L2, L3 ReadyQueue
    delete readyListL1;
    delete readyListL2;
    delete readyListL3;
    //<TODO>
    //delete readyList;
}
```

```
int Thread::getQueueLevel() const {
    if (Priority < 50) {
        return 3; // L3 queue
    } else if (Priority < 100) {
        return 2; // L2 queue
    } else {
        return 1; // L1 queue
    }
}

// Define this in an appropriate location that's accessible when you
instantiate the SortedList
int CompareThreadsByRemainingTime(Thread* a, Thread* b) {
    // Ensure that you safely handle possible null pointers if necessary
    if (!a || !b) return 0; // or handle differently based on your logic

    return a->getRemainingBurstTime() - b->getRemainingBurstTime();
}
```

Build the constructor and deconstructor for **L1, L2** and **L3. L3** is the only one implemented differently from the other 2 because it is a preemptive **Shortest Remaining Time Next** algorithm.

- L3 Queue: Assigned to threads with a priority less than 50. This queue typically handles lower-priority tasks.
- L2 Queue: Assigned to threads with a priority between 50 and 99. It's designated for medium-priority tasks.
- L1 Queue: Assigned to threads with a priority 100 or greater. This queue is reserved for high-priority tasks.

CompareThreadsByRemainingTime⇒ Provides a comparison mechanism for sorting threads based on their remaining burst time. This is critical for scheduling algorithms that prioritize shorter jobs, like **Shortest Remaining Time First** (SRTF).

**2.9) threads/scheduler.cc ⇒void Scheduler::ReadyToRun (Thread *thread)**

```
        //<TODO>
    // According to priority of Thread, put them into corresponding
ReadyQueue.
    // After inserting Thread into ReadyQueue, don't forget to reset
some values.
    // Hint: L1 ReadyQueue is preemptive SRTN(Shortest Remaining Time
Next).
    // When putting a new thread into L1 ReadyQueue, you need to check
whether preemption or not.

    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    char buffer[256];
    switch (thread->getQueueLevel()) {
        case 1:
         static_cast<SortedList<Thread*>*>(readyListL1)->Insert(thread);
// Cast to SortedList and use Insert
            break;
        case 2:
            readyListL2->Append(thread);
            break;
        case 3:
            readyListL3->Append(thread);
            break;
    }

    thread->setStatus(READY);
    (void) kernel->interrupt->SetLevel(oldLevel);
    //<TODO>
```

- The thread's queue level is determined using **thread->getQueueLevel()**, which classifies the thread into one of three queues **(L1, L2, L3)** based on its priority.
- L1 Queue: This queue operates on a **Shortest Remaining Time Next** (**SRTN**) policy, which may involve preemptive scheduling where the running process is replaced by a more critical task.
- L2 Queue: Threads are managed on a **first-come, first-served basis.**
- L3 Queue: Also managed on a **first-come, first-served basis.**
- The function temporarily disables interrupts (by setting the interrupt level to **IntOff**) to prevent race conditions
- The status of the thread is set to **READY**, indicating that it is prepared to be scheduled for execution when its turn comes.

**2.10) threads/scheduler.cc ⇒ Thread *Scheduler::FindNextToRun ()**

```
    // a.k.a. Find Next (Thread in ReadyQueue) to Run
    Thread* nextThread = NULL;

    if (!readyListL1->IsEmpty()) {
        nextThread = readyListL1->RemoveFront();
    } else if (!readyListL2->IsEmpty()) {
        nextThread = readyListL2->RemoveFront();
    } else if (!readyListL3->IsEmpty()) {
        nextThread =  readyListL3->RemoveFront();
    }

    return nextThread;
```

Sequentially checks each of the three priority queues—starting with the highest priority (L1), then medium (L2), and finally the lowest (L3). It selects the front thread from the first non-empty queue:

- L1 Check: If the highest priority queue (L1) is not empty, it removes and returns the thread at the front.
- L2 Check: If L1 is empty, it proceeds to check the second priority queue (L2), removing and returning the front thread if available.
- L3 Check: If both L1 and L2 are empty, it checks the lowest priority queue (L3) and returns the front thread.
- returns the thread selected to run next. If all queues are empty, it returns **NULL**, indicating no thread is ready to run.

**2.11) threads/scheduler.cc ⇒ void Scheduler::Aging()**

```
  void Scheduler::Aging() {
    ListIterator<Thread *> *it1 = new ListIterator<Thread
*>(readyListL1);
    ListIterator<Thread *> *it2 = new ListIterator<Thread
*>(readyListL2);
    ListIterator<Thread *> *it3 = new ListIterator<Thread
*>(readyListL3);

    while (!it1->IsDone()) {
        Thread *thread = it1->Item();
        if (kernel->stats->totalTicks - thread->getRRTime()  > 400) {
            int oldPriority = thread->getPriority();
            thread->setPriority(oldPriority + 10);
           //debug statement
        }
        it1->Next();
    }

    while (!it2->IsDone()) {
        Thread *thread = it2->Item();
        //rest of code
```

- Iterate Over L1 Queue: It uses a ListIterator to traverse the highest priority queue (L1). For each thread, if it has been waiting for more than 400 ticks, its priority is increased by 10.
- Iterate Over L2 Queue: Similarly, the function iterates through the medium priority queue (L2), applying the same aging mechanism to threads waiting for more than 400 ticks.
- Iterate Over L3 Queue: Lastly, the same process is applied to the lowest priority queue (L3), ensuring that threads in all queues are considered for priority boosting based on their waiting time.

- This aging process helps maintain system responsiveness by boosting the priority of older waiting threads, allowing them a better chance to execute, especially in a multi-level priority queue system where newer or higher initial priority threads might otherwise dominate CPU time.

**PART 3 DEBUG**

To node DEBUG statements from functions from above removed for the purpose to save space in the report, and redundancy. Where the DEBUG statements are explained in this part.

**a) void Scheduler::ReadyToRun (Thread *thread) , Inserted into Queue**

```
    char buffer[256];
    sprintf(buffer, "[InsertToQueue] Tick [%d]: Thread [%d] is inserted 
into queue L[%d]",
            stats->totalTicks, thread->getID(), 
thread->getQueueLevel());
    DEBUG('z', buffer);
```

**b) void Scheduler::FindNexttoRun () , Deleted from Queue**

```
    char buffer[256];
    sprintf(buffer, "[RemoveFromQueue] Tick [%d]: Thread [%d] is removed 
from queue L[%d]",
            kernel->stats->totalTicks, nextThread->getID(), 
nextThread->getQueueLevel());
    DEBUG('z', buffer);
```

**c) void Scheduler::Aging(), Update Priority**

```
            char buffer[256];
            sprintf(buffer, "[UpdatePriority] Tick [%d]: Thread [%d]
        changes its priority from [%d] to [%d]",
                            kernel->stats->totalTicks, thread->getID(),
        oldPriority, thread->getPriority());
                    DEBUG('z', buffer);
```

**e)void Scheduler::Run (Thread *nextThread, bool finishing), Context switching**

```
    char buffer[256];
    sprintf(buffer, "[ContextSwitch] Tick [%d]: Thread [%d] is now
selected for execution, thread [%d] is replaced, and it has executed
[%d] ticks",
            kernel->stats->totalTicks, nextThread->getID(),
oldThread->getID(), oldThread->getRunTime());
    DEBUG('z', buffer);
```

**Since, the DEBUG holds only 2 arguments, here is how I implemented it.**

**1)**A character array **buffer** is used to hold the formatted string. The size 256 is chosen to ensure that the string, which includes multiple variable values and static text, fits without overflowing, which could lead to errors or security issues.

**2)** The **sprintf** function is utilized here to format a string that includes both static text and variable values into the buffer. This function is similar to printf but instead of sending output to console, it stores the output in the string (buffer) provided.

3) This line outputs the formatted string stored in **buffer** to a debugging log (or console, depending on the implementation of the **DEBUG** macro), associated with a specific debug level (**'z'** in this case).

**Part 3) Contributions**

No group member just did it myself. Since no one wanted to partner with me I couldn’t finish debugging the code where the workload was heavy. Although the code runs, it is still bugged. I have tried my best to do the workload alone but was unsuccessful at the end.