# Machine Problem 3 – CPU scheduling Rev. 191109

Deadline: 2019/12/15 23:59

I. Goal

The default CPU scheduling algorithm of Nachos is a simple round-robin scheduler with 100 ticks time quantum. The goal of this MP is to replace it with a **multilevel feedback queue, and understand the implementation of process lifecycle management and context switch mechanism.**

II. Assignment

1. **Trace code: Explain the purposes and details of the following 6 code paths** to understand how nachos manages the lifecycle of a process (or thread) as described in the Diagram of Process State in our lecture slides (ch3 p8).

1-1. New→Ready

```
Kernel::ExecAll()
        ↓
Kernel::Exec(char*)
        ↓
Thread::Fork(VoidFunctionPtr, void*)
        ↓
Thread::StackAllocate(VoidFunctionPtr, void*)
        ↓
Scheduler::ReadyToRun(Thread*)
```

1-2. Running→Ready

```
Machine::Run()
        ↓
Interrupt::OneTick()
        ↓
Thread::Yield()
        ↓
Scheduler::FindNextToRun()
        ↓
Scheduler::ReadyToRun(Thread*)
        ↓
Scheduler::Run(Thread*, bool)
```

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

```
SynchConsoleOutput::PutChar(char)
        ↓
Semaphore::P()
        ↓
SynchList<T>::Append(T)
        ↓
Thread::Sleep(bool)
        ↓
Scheduler::FindNextToRun()
        ↓
Scheduler::Run(Thread*, bool)
```

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

```
Semaphore::V()
        ↓
Scheduler::ReadyToRun(Thread*)
```

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

```
ExceptionHandler(ExceptionType) case SC_Exit
        ↓
Thread::Finish()
        ↓
Thread::Sleep(bool)
        ↓
Scheduler::FindNextToRun()
        ↓
Scheduler::Run(Thread*, bool)
```

1-6. Ready→Running

```
Scheduler::FindNextToRun()
        ↓
Scheduler::Run(Thread*, bool)
        ↓
SWITCH(Thread*, Thread*)
        ↓
```

(depends on the previous process state, ex. [New,Running,Waiting]→Ready)

↓

for loop in `Machine::Run()`

**Note: switch.S contains the instructions to perform context switch. You must understand and describe the purpose of these instructions in your report. (You can try to understand the x86 instructions first. Appendix C and the MIPS version equivalent to x86 can get a lot of help.)**

2. Implementation

2-1. Implement a **multilevel feedback queue** scheduler with aging mechanism as described below:

(a) There are **3 levels of queues**: L1, L2 and L3. L1 is the highest level queue, and L3 is the lowest level queue.

(b) All processes must have a valid **scheduling priority between 0 to 149**. Higher value means higher priority. So 149 is the highest priority, and 0 is the lowest priority.

(c) A process with priority between **0~49 is in L3** queue. A process with priority between **50~99 is in L2** queue. A process with priority between **100~149 is in L1** queue.

(d) **L1 queue uses preemptive SJF**(shortest job first) scheduling algorithm. If current thread has the lowest approximate burst time, it should not be preempted by the threads in ready queue. The job execution time is approximated using the equation:

$$t_i = 0.5 * T + 0.5 * t_{i-1}$$ (type double)

(e) **L2 queue uses non-preemptive priority** scheduling algorithm. If current thread has the highest priority, it should not be preempted by the threads in ready queue.

(f) **L3 queue uses round-robin** scheduling algorithm with time quantum **100 ticks**.

(g) An **aging mechanism** must be implemented, so that the priority of a process is **increased by 10 after waiting for more than 1500 ticks (The operations of preemption and priority updating can be delayed until the next timer alarm interval)**.

2-2. **Add a command line argument "-ep"** for nachos to initialize priority of process.
E.g.,: the command below will launch 2 processes: test1 with initial priority 40, and test2 with initial priority 80.

```
$ ../build.linux/nachos -ep test1 40 -ep test2 80
```

2-3. **Add a debugging flag 'z'** and use the DEBUG('z', expr) macro (defined in debug.h) to print following messages. Replace { . . . } to the corresponding value.

(a) Whenever a process is inserted into a queue:

```
[A] Tick [{current total tick}]: Thread [{thread ID}] is inserted into queue L[{queue level}]
```

(b) Whenever a process is removed from a queue:

```
[B] Tick [{current total tick}]: Thread [{thread ID}] is removed from queue L[{queue level}]
```

(c) Whenever a process changes its scheduling priority:

```
[C] Tick [{current total tick}]: Thread [{thread ID}] changes its priority from [{old value}]
to [{new value}]
```

(d) Whenever a process updates its approximate burst time:

```
[D] Tick [{current total tick}]: Thread [{thread ID}] update approximate burst time, from:
[{t_{i-1}}], add [{T}], to [{t_i}]
```

(e) Whenever a context switch occurs:

```
[E] Tick [{current total tick}]: Thread [{new thread ID}] is now selected for execution,
thread [{prev thread ID}] is replaced, and it has executed [{accumulated ticks}] ticks
```

- Rules: You MUST follow the following rules in your implementation:

(a) **Do not modify any code under machine folder (except Instructions 2. below).**

(b) **Do NOT call the Interrupt::Schedule() function from your implemented code. (It simulates the hardware interrupt produced by hardware only.)**

(c) Only update approximate burst time $t_i$ (include both user and kernel mode) when process change its state from running to waiting. In case of running to ready (interrupted), its **CPU burst time T must keep accumulating after it resumes running**.

(d) **The operations and rescheduling events of aging can be delayed until the timer alarm is triggered (the next 100 ticks timer interval)** .

(e) [Recommended] In order to increase compile and debug efficiency, write function body code in *.cc, only leave prototypes in *.h.

Note: Ask TA if you are not clear on any of these rules.

## III. Instructions

1. Copy your code for MP2 to a new folder

```
$ cp -r NachOS-4.0_MP2 NachOS-4.0_MP3
```

2. To observe scheduling easily by PrintInt(), change ConsoleTime to 1 in machine/stats.h.

```
const int ConsoleTime = 1;
```

3. Comment out postOffice at Kernel::Initialize() and Kernel::~Kernel() in kernel.cc.

```
// postOfficeIn = new PostOfficeInput(10);
// postOfficeOut = new PostOfficeOutput(reliability);
// delete postOfficeIn;
// delete postOfficeOut;
```

4. Test your implementation, try different p1 and p2. (Appendix A provides some examples)

```
$ cd NachOS-4.0_MP3/code/test
$ cp /home/os2019/share/hw3t{1,2}.c ./
$ cat /home/os2019/share/hw3Makefile >> Makefile
$ make hw3t1 hw3t2
$ ../build.linux/nachos -ep hw3t1 <p1> -ep hw3t2 <p2>
```

IV. Grading

1. Implementation correctness – 55%
   (a) Pass all test cases.
   (b) Correctness of working items.
   (c) Your working directory will be copied for validation after deadline.
2. Report – 15%
   (a) **Cover page** including team members, team member contribution.
   (b) Explain the code trace required in Part II-1.
   (c) Explain your implementation for Part II-2 in details.
   (d) Must uploaded to ILMS in pdf format.
3. Demo – 30%
   (a) You must explain **test cases in Appendix B ONLY based on the output logging information**.
   (b) You must demonstrate your implementation, and answer questions from TAs in **20 minutes**. (**Exceeding this time limit could result in point deduction!!!**)
   (c) **Some random test case** will be used for correctness verification during the demo.

Appendix A

../build.linux/nachos -ep hw3t1 0 -ep hw3t2 0 #L3
../build.linux/nachos -ep hw3t1 50 -ep hw3t2 50 #L2
../build.linux/nachos -ep hw3t1 50 -ep hw3t2 90 #L2
../build.linux/nachos -ep hw3t1 100 -ep hw3t2 100 #L1
../build.linux/nachos -ep hw3t1 40 -ep hw3t2 55 #L3→L2
../build.linux/nachos -ep hw3t1 40 -ep hw3t2 90 #L3→L2
../build.linux/nachos -ep hw3t1 90 -ep hw3t2 100 #L2→L1

Appendix B

../build.linux/nachos -ep hw3t1 100 -ep hw3t2 100 -d z #L1
../build.linux/nachos -ep hw3t1 90 -ep hw3t2 100 -d z #L2→L1

Appendix C

x86 registers (32bit)

| Register | Description |
|---|---|
| eax, ebx, ecx, edx, esi, edi | general purpose registers |
| esp | stack pointer |
| ebp | base pointer |

x86 instructions (32bit, AT&T)

| Instruction | Description |
|---|---|
| movl %eax, %ebx | move 32bit value from register eax to register ebx |
| movl 4(%eax), %ebx | move 32bit value at memory address pointed by (the value of register eax plus 4), to register ebx |
| ret | set CPU program counter to the memory address pointed by the value of |

| | |
|---|---|
| | register esp |
| pushl %eax | move 32bit value of register eax to the memory address pointed by register esp, then subtract register esp by 4 (%esp = %esp - 4) |
| popl %eax | move 32bit value at the memory address pointed by register esp to register eax, then add register esp by 4 (%esp = %esp + 4) |
| call *%eax | push CPU program counter + 4 (return address) to stack, then set CPU program counter to memory address pointed by the value of register eax |

x86 calling convention (cdecl)

| Item | Description |
|---|---|
| Caller passes function parameters | Caller pushes the arguments to stack in the descending order. |
| Callee catches function parameters | Callee reads the arguments from the memory address pointed by (register esp + 4) in the ascending order. By the way, the value of memory address pointed by register esp is the return address described above (call instruction). |
| Function parameters cleaning up | Caller is responsible for cleaning up the arguments (pop). |