CS342301 2021 MP3 – CPU scheduling

Deadline: 2021/12/19 23:59

★ 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.

★ Assignment

- 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 (chp.3 p.8).

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, e.g.,
[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.)

- 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, priority between **50 - 99** is in **L2** queue, and priority between **100 - 149** is in **L1** queue.

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

$$t_i = 0.5 * T + 0.5 * t_{i-1} \text{ (type double)}, i > 0, t_0 = 0$$

Update burst time when state becomes waiting state, stop updating when state becomes ready state, and resume accumulating when state moves back to running state.

(e) **L2** queue uses a **non-preemptive priority** scheduling algorithm. A thread in L2 queue won't preempt other threads in L2 queue; however, it will preempt thread in L3 queue.

(f) **L3** queue uses a **round-robin** scheduling algorithm with time quantum **100 ticks** (you should select a thread to run once 100 ticks elapsed). If two threads enter the L3 queue with the same priority, either one of them can execute first.

(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** (Note: The operations of preemption and priority updating must be delayed until the **next timer alarm** interval in alarm.cc Alarm::Callback).

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 the **machine folder** (except for 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 $\mathbf{t}_i$ (include both user and kernel mode) when the process changes its state **from running state to waiting state**. In case of running to ready (interrupted), its CPU burst time T must keep accumulating after it resumes running.

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

(e) Due to rule (d), the below example is an acceptable solution: 2 threads *x*, *y* are waiting in the L3 queue, and thread *x* started executing at ticks 20. At ticks 100, the timer alarm was triggered and hence caused the rescheduling. So *x* left the running state and *y* started running.

★ 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/os2021/share/hw3t{1,2,3}.c ./
   $ cat /home/os2021/share/MP3_Makefile >> Makefile
   $ make hw3t1 hw3t2 hw3t3
   $ ../build.linux/nachos -ep hw3t1 <p1> -ep hw3t2 <p2>
   ```

★ 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 the 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 detail.
   (d) Upload to iLMS with the filename **MP3_report_<Group Number>.pdf**
3. Demo – 30%
   (a) Demonstrate your implementation, and answer questions from TAs in 20 minutes.
   (b) Some random test cases will be used for correctness verification.
4. **Plagiarism**
   (a) **NEVER SHOW YOUR CODE** to others.
   (b) If the codes are similar to other people (including your upperclassman) and you can't answer questions properly during the demo, you will be identified as plagiarism.

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
$ ../build.linux/nachos -ep hw3t1 60 -ep hw3t3 50   #L2
```

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	subtract register esp by 4 (%esp = %esp - 4), then move 32bit value of register eax to the memory address pointed by register esp
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).