CS342301 2023 MP3 – CPU scheduling

Deadline: 2023/05/02 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 Preemptive Shortest Job First 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()
↓
```

```
List<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)
2.5↓
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 **Preemptive** Shortest Job First scheduling algorithm

(a) 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:

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

Where T is the total running ticks within a CPU burst and the NachOS kernel statistic can be used to calculate the ticks.

(b) 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.

(c) When the running thread becomes ready state, the remaining burst time $\max(0, t_i - T)$ of the current running thread is used to compare with other threads in the ready queue.

(d) The operations of preemption **MUST** be delayed until the **next timer alarm** interval in alarm.cc Alarm::Callback.

2-2. 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`

(b) Whenever a process is removed from a queue:
`[B] Tick [{current total tick}]: Thread [{thread ID}] is removed from queue`

(c) Whenever a process updates its approximate burst time:
`[C] Tick [{current total tick}]: Thread [{thread ID}] update approximate burst time, from: [{`t_{i-1}`}], add [{T}], to [{`t_i`}]`

(d) Whenever a context switch occurs without preemption:
`[D] Tick [{current total tick}]: Thread [{new thread ID}] is now selected for execution, thread [{prev thread ID}] starts IO, and it has executed [{accumulated ticks}] ticks`

(e) Whenever a context switch occurs with preemption:
`[E] Tick [{current total tick}]: Thread [{new thread ID}] is now selected for execution, thread [{prev thread ID}] is preempted, and it has executed [{accumulated ticks}] ticks`

2-3. Hint: The following files "may" be modified…

- **threads/thread.***
- **threads/scheduler.***
- **lib/debug.h**

■ 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.

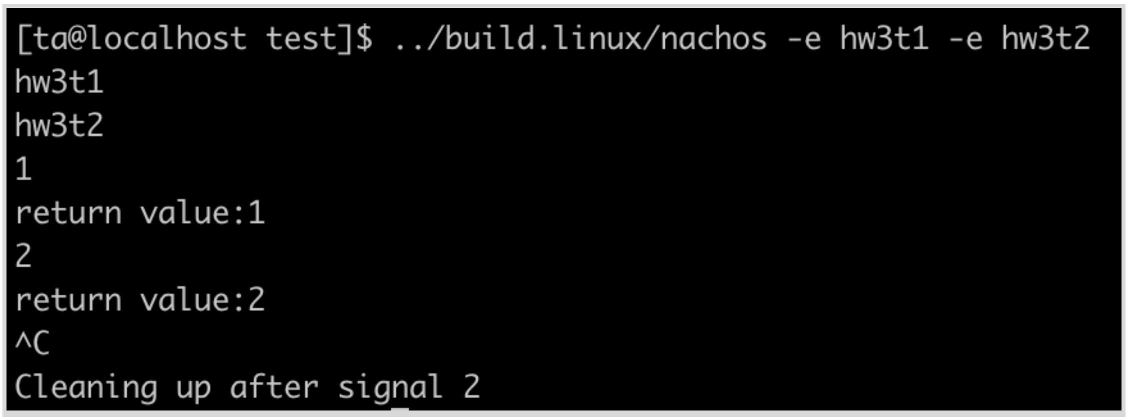
- Report
  - Cover page, including studentID, teamID.
  - Explain the code trace required in Part II-1.
  - Explain your implementation for Part II-2 in detail.

★ Instructions

1. Copy the code for MP3 to a new folder.
```
$ cp -r /home/os2023/share/NachOS-4.0_MP3 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.
```
$ cd NachOS-4.0_MP3/code/test
$ cp /home/os2023/share/hw3t{1,2,3}.c ./
$ make hw3t1 hw3t2 hw3t3
$ ../build.linux/nachos -e hw3t1 -e hw3t2
```

***************************** IMPORTANT *****************************

hw3t1 should terminate first, no matter what the printing order is.


```
[ta@localhost test]$ ../build.linux/nachos -e hw3t1 -e hw3t2
hw3t1
hw3t2
1
return value:1
2
return value:2
^C
Cleaning up after signal 2
```

***************************** IMPORTANT *****************************

★ Grading

1. Implementation correctness – 60%

   (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) Upload to eeclass with the filename **MP3_report_[StudentID].pdf** (E.g. MP3_report_111062584.pdf)
3. Demo – 25%
    (a) We will ask several questions about your codes.
    (b) Demos will take place on our server, so you are responsible to make sure your code works on our server.
4. TAs won't help you recover from file loss or misedited. Please make sure you backup your code. You can use git or simply make a copy in your local.
5. **Late submissions will not be accepted.** Refer to the course syllabus for detailed homework rules and policies.
6. **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 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).