Precept 4: Multicore + Preemption

COS 318: Fall 2017

Project 4 Schedule

- Precept: Monday 11/6, 7:30pm
  - (You are here)
- Design Review: Monday 11/13
- Due: Sunday, 11/19, 11:55pm

Precept Overview

- Adding multicore support
- Preemption
- Producer-Consumer problem
- Project specific topics

Multicore Support

Supporting “SMP”

- SMP: Symmetric Multiprocessing
  - All CPUs have equivalent access to resources
- Bootup: BSP initializes system + activate APs
- Each CPU has a core and a LAPIC
  - LAPIC: Performs interrupt routing and delivery

Stack organization

- Each processor needs its own kernel stack
  - Different from stacks used by process's kernel threads
  - Location specified by processor's TSS
- We use "kernel bootstrap stacks"
  - Switch to process kernel stack after performing setup

Multiprocessor OS: CPU State

- Must distinguish between global state and per-CPU state
- What state is private to a CPU?

Multiprocessor OS: Locking

- We can now have multiple CPUs in the kernel at the same time
  - What if they write to the same kernel memory?
- Strawman approach: Big Kernel Lock
- Our approach: Fine grained locking

Preemptive Scheduling

Preemption: Clock interrupts

- Current OS: One process can hog CPU
  - Want to preempt processes after a timeout
- On timer interrupt: forcefully switch to another thread
  - Allows interleaving without explicit yields

Preemption: Scheduling

- LAPIC can give us timer interrupts
  - Count number of milliseconds thread has run for
  - Yield once runtime > threshold
- Choose another thread to run
  - We use round-robin

Preempting Kernel Execution

Which part will be affected?

- Temporarily enable interrupts during the executions of `sys_produce` and `sys_consume`
- Leave other parts of the kernel unchanged;
- So only enable interrupts during these two functions.

Disable Interrupts in Produce and Consume

- When `sys_produce` or `sys_consume` call functions in the kernel, they should first disable interrupts.
- `intr_local_enable` and `intr_local_disable`: kern/dev/intr.h;

What you should do?

- Only adding statements to enable or disable interrupts;
- Don’t worry about how preempting kernel execution is achieved (Read Spec if you have interest);

Improvement on Trap function

- Calling trap function => Switch kernel stack and page structure;
- Unnecessary when the interrupt is triggered in the kernel.
- Method: Remember the last active thread ID for each CPU;

Improvement on Sys_Spawn function

- Detect possible errors and set appropriate error codes;
- Possible Error Codes: `E_EXCEEDS_QUOTA`, `E_MAX_NUM_CHILDREN_REACHED`, `E_INVAL_CHILD_ID` (Can be found in kern/lib/syscall.h)

The Producer and Consumer

What you should do

- Implement condition variables and a bounded buffer as shared object.
- Utilized the spinlock.c => CV
- Once Bounded-buffer is full, The producer process should be put in the waiting list;
- Similarly, Empty => The consumer process;

What you should do

- Open-Ended Part
- Please add appropriate debug output so that you and graders know your codes are working (eg. when buffer is full => prompt “buffer is full”, “add Consumer process 1 to waiting list”).

Project Specific Topics

General Tips

- Read Section 2.3 (Interprocess Communication)
  - CV / Monitor version of Producer / Consumer should give you a general idea
- Debugging concurrent programs is hard
  - gdb can show what each thread is doing
- Please clean up before you submit!

Design Review

- Explain how to use condition variables and locks to implement a bounded buffer.
- Provide pseudocode for the implementation of `sys_produce` and `sys_consume`, using above bounded buffer.

Questions?