CS 3204
Operating Systems

Lecture 9

Godmar Back

Announcements

- Project 1 is due Feb 27, 11:59pm
  - Not a whole lot of time, find a team now.
- *nix Crash Course offered: Feb 9, 8:30pm
- Project 1 Help Session
  - 1) 7pm MCB 129
  - 2) TBA later this week
- Reading: Section 5.1 through 5.4

Concurrency & Synchronization

pthread_mutex example

```
/* Define a mutex and initialize it. */
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static int counter = 0; /* A global variable to protect. */

/* Function executed by each thread. */
static void *
increment(void *_)
{
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
}
```

```
movl    counter, %eax
incl    %eax
movl    %eax, counter
```


A Race Condition

time

Thread 1		Thread 2
movl counter, %eax (0)	IRQ OS decides to context switch	
		movl counter,%eax (0)
		incl %eax (1)
incl %eax (1)	IRQ	
movl %eax, counter (1)	IRQ	
		movl %eax,counter (1)

%eax – Thread 1's copy
%eax – Thread 2's copy
counter – global variable, shared

Assume counter == 0 initially
Final result: counter is 1, should be 2

Race Conditions

- Definition: *two or more threads read and write a shared variable, and final result depends on the order of the execution of those threads*
- Usually timing-dependent and intermittent
  - Hard to debug
- Not a race condition if all execution orderings lead to same result
  - Chances are high that you misjudge this
- How to deal with race conditions:
  - Ignore (!?)
    - Can be ok if final result does not need to be accurate
    - Never an option in CS 3204
  - Don't share: duplicate or partition state
  - Avoid "bad interleavings" that can lead to wrong result

Not Sharing: Duplication or Partitioning

- Undisputedly best way to avoid race conditions
  - Always consider it first
  - Usually faster than alternative of sharing + protecting
  - But duplicating has space cost; partitioning can have management cost
  - Sometimes must share (B depends on A's result)
- Examples:
  - Each thread has its own counter (then sum counters up after join())
  - Every CPU has its own ready queue
  - Each thread has its own memory region from which to allocate objects
- Truly ingenious solutions to concurrency involve a way to partition things people originally thought you couldn't

Aside: Thread-Local Storage

- A concept that helps to avoid race conditions by giving each thread a copy of a certain piece of state
- Recall:
  - All local variables are already thread-local
    - But their extent is only one function invocation
  - All function arguments are also thread-local
    - But must pass them along call-chain
- TLS creates variables of which there's a separate value for each thread.
- In PThreads/C (compiler or library-supported)
  - Dynamic: pthread_create_key(), pthread_get_key(), pthread_set_key()
    - E.g. myvalue = keytable(key_a)→get(pthread_self());
  - Static: using __thread storage class
    - E.g.: __thread int x;
- Java: java.lang.ThreadLocal

In Pintos:
Add member to struct thread

Race Condition & Execution Order

- Prevent race conditions by imposing constraints on execution order so the final result is the same regardless of actual execution order
  - That is, exclude "bad" interleavings
  - *Specifically*: disallow other threads to start updating shared variables while one thread is in the middle of doing so; make those updates *atomic*.

Atomicity & Critical Sections

- Atomic: indivisible
- Certain machine instructions are atomic
- Critical Section
  - A synchronization technique to ensure atomic execution of a segment of code
- Requires *entry()* and *exit()* operations

```
pthread_mutex_lock(&lock);     /* entry() */
counter++;
pthread_mutex_unlock(&lock);  /* exit() */
```


Critical Sections (cont'd)

- Critical Section Problem also known as mutual exclusion problem
- Only one thread can be inside critical section; others attempting to enter CS must wait until thread that's inside CS leaves it.
- Note: different from "all-or-nothing" meaning atomic has in database theory & practice
  - Does not necessarily imply that thread executes section without interruption, or even that thread completes section – just that other threads can't enter it while one thread is inside it
- Solutions can be entirely software, or entirely hardware
  - Usually combined
  - Different solutions for uniprocessor vs multiprocessor scenarios

Disabling Interrupts

- All asynchronous context switches start with interrupts
  - So disable interrupts to avoid them!

```
intr_level old = intr_disable();
/* modify shared data */
intr_set_level(old);
```

```
void intr_set_level(intr_level to)
{
    if (to == INTR_ON)
      intr_enable();
    else
      intr_disable();
}
```

Avoiding context switches: Variation (1)

- Variation of "disabling-interrupts" technique
  - That doesn't actually disable interrupts
  - If IRQ happens, ignore it
- Assumes writes to "taking_interrupts" are atomic and sequential wrt reads

```
taking_interrupts = false;
/* modify shared data */
taking_interrupts = true;
```

```
intr_entry()
{
  if (!taking_interrupts)
    iret
  intr_handle();
}
```


Avoiding context switches: Variation (2)

- Code on previous slide could lose interrupts
  - Remember pending interrupts and check when leaving critical section
- This technique can be used with Unix signal handlers (which are like "interrupts" sent to a Unix process)
  - but tricky to get right

```
taking_interrupts = false;
/* modify shared data */
if (irq_pending)
   intr_handle();
taking_interrupts = true;
```

```
intr_entry()
{
  if (!taking_interrupts) {
    irq_pending = true;
    iret
  }
  intr_handle();
}
```


Avoiding context switches: Variation (3)

- Instead of setting flag, have irq handler examine PC where thread was interrupted
- See Bershad '92: Fast Mutual Exclusion on Uniprocessors

```
critical_section_start:
  /* modify shared data */
critical_section_end:
```

```
intr_entry()
{
  if (PC in (critical_section_start,
             critical_end_end)) {
    iret
  }
  intr_handle();
}
```


Disabling Interrupts: Summary

- (this applies to all variations)
- Sledgehammer solution
- Infinite loop means machine locks up
- Use this to protect data structures from concurrent access by interrupt handlers
  - Keep sections of code where irqs are disabled minimal (nothing else can happen until irqs are reenabled – latency penalty!)
  - If you block (give up CPU) mutual exclusion with other threads is not guaranteed
    - Any function that transitively calls thread_block() may block
- Want something more fine-grained
  - Key insight: don't exclude *everybody* else, only those contending for the same critical section