CS 3204
Operating Systems

Lecture 7

Godmar Back

Virginia Tech

Announcements

- Project 1 due Monday Sep 25, 11:59pm
- Announcements:
  - Unix Crash Course by VTLUUG:
    - Tuesday Sep 12, 6-8:30pm Squires 116
- Reading assignments:
  - Chapters 1, 2 – skim. Read carefully 1.5.
  - Read carefully Chapter 3.1-3.3
  - Read carefully Chapter 6.1-6.4

Project 1 Suggested Timeline

- Today Sep 12:
  - Should have finished alarm clock, implemented basic priority scheduling & should pass almost all alarm & basic priority tests
- Complete basic priority by Sep 13
- Priority Inheritance & Advanced Scheduler take the most time, start them in parallel – will take the most time to implement & debug
- Due date Sep 25

Type-safe arithmetic types in C

```
typedef struct
 {
   double      re;
   double      im;
 } complex_t;

static inline complex_t
complex_add(complex_t x, complex_t y)
{
  return (complex_t){ x.re + y.re, x.im + y.im };
}
```

Pitfall: typedef int fixed_point_t;
fixed_point_t x;
int y;
x = y; // no compile error

```
static inline double
complex_real(complex_t x)
{
  return x.re;
}

static inline double
complex_imaginary(complex_t x)
{
  return x.im;
}

static inline double
complex_abs(complex_t x)
{
  return sqrt(x.re * x.re + x.im * x.im);
}
```


Concurrency & Synchronization

Virginia Tech

Overview

- Will talk about locks, semaphores, and monitors/condition variables
- For each, will talk about:
  - What abstraction they represent
  - How to implement them
  - How and when to use them
- Two major issues:
  - Mutual exclusion
  - Scheduling constraints
- Project note: Pintos is somewhat unusual in that it implements its locks on top of semaphores

pthread_mutex example

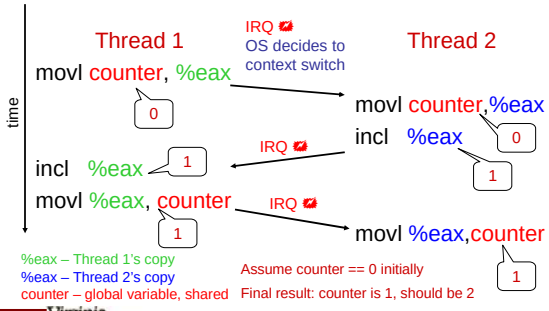
```
/* Define a mutex and initialize it. */
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

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



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();
}
```


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