CS 3204
Operating Systems

Lecture 6

Godmar Back

Announcements

- Project 1 due on Sep 29, 11:59pm
- Help session slides online
- Out of town next week
  - No office hours
- Lectures will be held:
  - Tuesday: Dr. Butt
  - Thursday: Dr. Tilevich
- These are not optional "guest lectures", they continue with the class material relevant to project and midterm

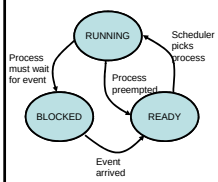
Project 1 Suggested Timeline

- End of this week:
  - Have read relevant project documentation, set up CVS, built and run your first kernel, designed your data structures for alarm clock
- Alarm clock by Sep 16
- Pass all basic priority tests by Sep 18
- Priority Inheritance & Advanced Scheduler will take the most time to implement & debug, start them in parallel
  - Should have design for priority inheritance figured out by Sep 23
  - Develop & test fixed-point layer independently by Sep 23
- Due date Sep 29

Processes & Threads (Summary)

- Had looked at APIs with which to create processes/threads
- Spawning vs. cloning
- "fork/join" paradigm (will be implemented in Project 2)
- Various embeddings of threading APIs in languages (C/POSIX threads, Java, C#)

Using thread_yield() to implement preemption



- Current thread ("RUNNING") is moved to READY state, added to READY list.
- Then scheduler is invoked. Picks a new READY thread from READY list.
- Case a): there's only 1 READY thread. Thread is rescheduled right away
- Case b): there are other READY thread(s)
  - b.1) another thread has higher priority – it is scheduled
  - b.2) another thread has same priority – it is scheduled provided the previously running thread was inserted in tail of ready list.
- "thread_yield()" is a call you can use whenever you identify a need to preempt current thread.
- Exception: inside an interrupt handler, use "intr_yield_on_return()" instead

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

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 implements its locks on top of semaphores

pthread_mutex example

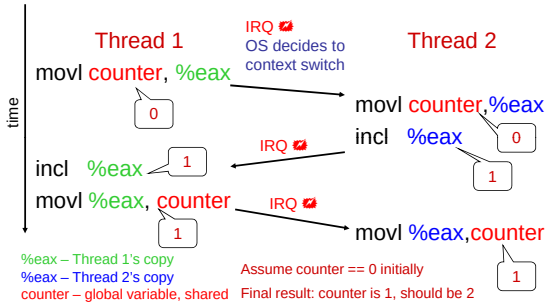
```
/* Define a mutex and initialize it. */
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

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* – threads either see old or new value, but none in between

Atomicity & Critical Sections

- Atomic: indivisible
  - Certain machine instructions are atomic
  - But need to create larger atomic sections
- 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: a critical section does not necessarily imply that thread executes section without interruption (i.e., preemption), or even that thread completes section – just that other threads can't enter this critical section while one thread is inside it/hasn't left it
- Solutions can be entirely software, or entirely hardware
  - Usually combined
  - Different solutions for uniprocessor vs multiprocessor scenarios

Implementing Critical Sections

- Will look at:
  - Disabling interrupts approach
  - Semaphores
  - Locks

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();
}
```

Implementing CS by 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();
}
```


Implementing CS by 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

Critical Section Problem

- A solution for the CS Problem must
  1) Provide mutual exclusion: at most one thread can be inside CS
  2) Guarantee Progress: (no deadlock)
     - if more than one threads attempt to enter, one will succeed
     - ability to enter should not depend on activity of other threads not currently in CS
  3) Bounded Waiting: (no starvation)
     - A thread attempting to enter critical section eventually will (assuming no thread spends unbounded amount of time inside CS)
- A solution for CS problem should be
  - Fair (make sure waiting times are balanced)
  - Efficient (not waste resources)
  - Simple