Process coordination

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Admin

- New TA: Hafiz Kamran Khalil
- Clarification of office hours:
  - Tuesday
  - Or by appointment
- Code navigation
- Course feedback
- Lecture speed

Previous lectures

- Context switch
- Null process
- Process management functions:
  - Suspension
  - Resumption
  - Creation
  - Termination

Previous lecture

- code for **newpid**()

Note about CPU registers

- CPU registers are special memory used by the processor
  - Usually function as temporary placeholders
- PC (“program counter”) is the same as IP (“instruction pointer”)
  - Special register that tells the CPU the memory location where to find the next instruction

Goals for today

- How can processes coordinate in a concurrent system?
- What are the use-cases for coordination?
- What are the primitives available to processes and the kernel?
- How are those primitives implemented?

Coordination of processes

- Necessary in a concurrent system
- Avoids conflicts when accessing shared resources
- Allows multiple processes to cooperate
- Can also be used when:
  - Process waits for IO
  - Process waits for another process

Two approaches to process coordination

- Use hardware mechanisms
  - Most useful for multiprocessor systems
  - May rely on busy waiting
- Use operating system mechanisms
  - Works well with a single processor
  - No unnecessary execution

General process coordination

- Producer/consumer interaction
- Mutual execution

Producer-consumer synchronization

- Typical scenario:
  - Shared circular buffer
  - A producing process add an item to the buffer
  - A consuming process removes an item from the buffer
- Must guarantee
  - A producer blocks when buffer is full
  - A consumer blocks when buffer is empty

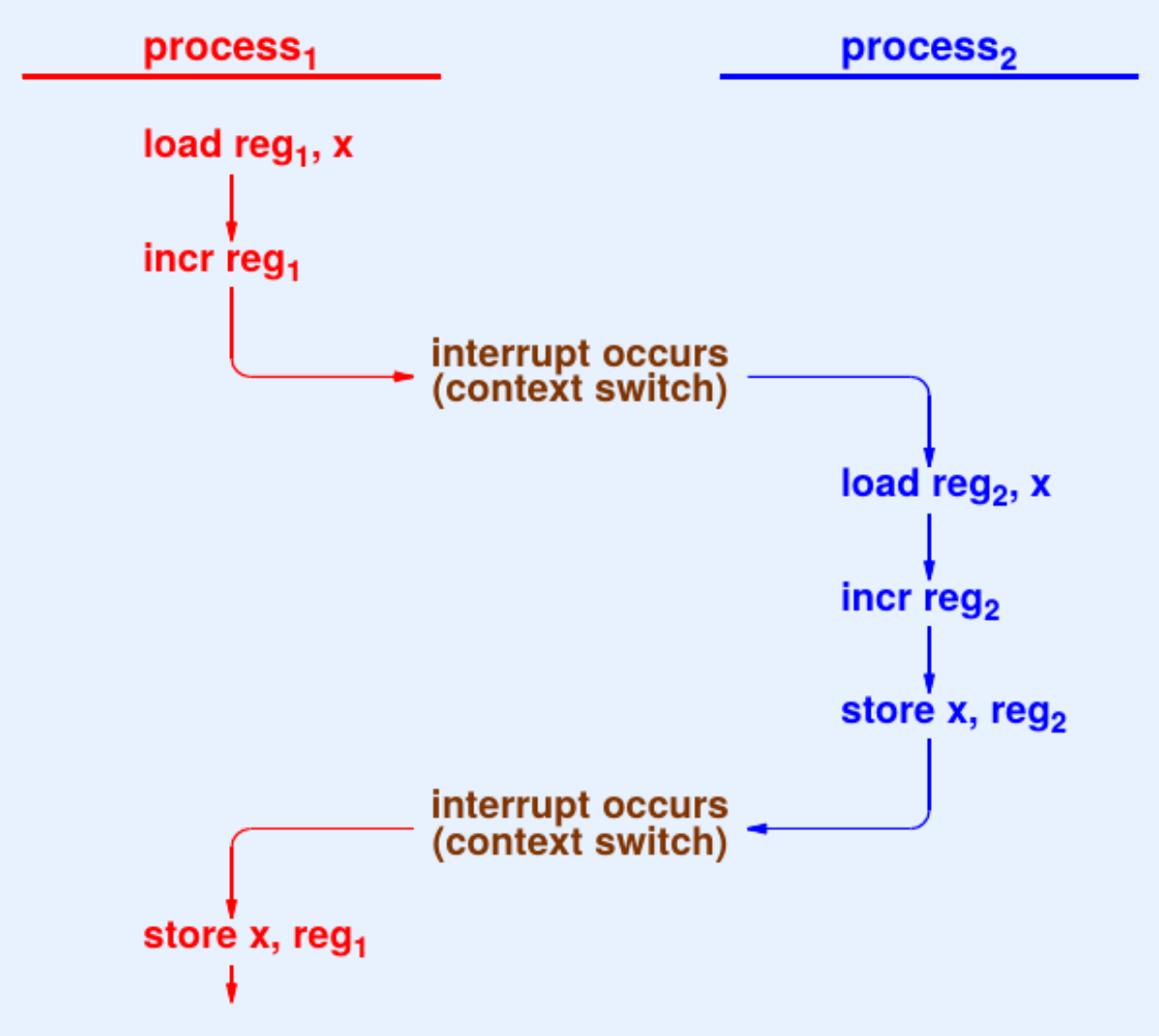
Mutual exclusion

- Allowing only one process to access a critical section at one time
- Concurrent processes access shared data / resources
- Non-atomic operations may produce unexpected results
  - Examples?

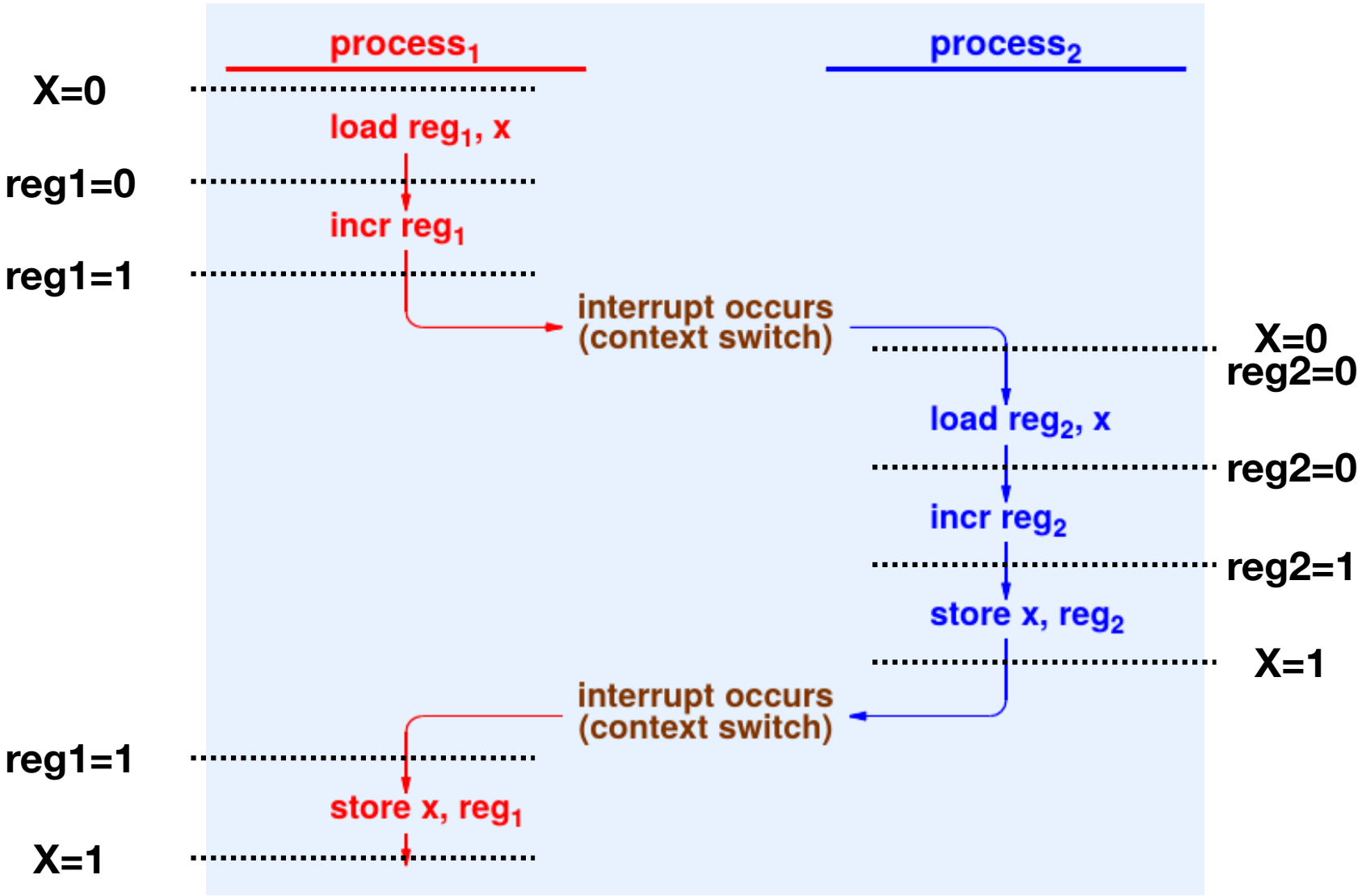
Example

- The C statement "**myvariable++;**" is not atomic
- Compiler may produce code such as:
  - 1. Load contents from **myvariable** into register **eax**
  - 2. Increment the register **eax**
  - 3. Store contents that are in **eax** into **myvariable**

Two processes attempting to increment a variable concurrently



Two processes attempting to increment a variable concurrently



2 increment operations but X was only incremented by 1

Other examples?

Critical section

- Ensure exclusive access
  - Ensure that only one process accesses a shared item at any time
- Critical section: part of the code
  - Each piece of shared data must be protected from concurrent access
  - Programmer inserts mutex operations
    - Before access to shared item
    - After access to shared item
  - Protected code known as critical section

Solution?

- Basic synchronization:
  - Disabling all interrupts
  - Spin locks
  - Semaphores
- More advanced synchronization:
  - HW transactional memory
  - Read copy update
  - Lock-free data structures

Interrupt mask: Low-level mutual exclusion

- Mutual exclusion can be simply guaranteed if there is no context switches
- Context switches occur by:
  - Calls to resched()
  - Interrupts (which can call resched())
- Low-level mutual exclusion: mask interrupts and avoid rescheduling

Interrupt mask: Low-level mutual exclusion

- Hardware mechanism that control interrupts
  - Through updating internal machine register
  - `IF` flag of `EFLAGS register in x86
- OS can:
  - Examine current interrupt mask (find out whether interrupts are enabled)
  - Set interrupt mask to prevent interrupts
  - Clear interrupt mask to allow interrupts

Interrupt mask: Low-level mutual exclusion

- Technique used:
  - Saves the current interrupt status
  - Disables interrupts
  - Proceeds through critical section
  - Restores interrupt status
- Code: **disable()** and **restore()** in Xinu
- **Q:** Why not simply disable() and enable()?
- Disable/restore allows nested interrupt masking

Why interrupt masking is insufficient

- Stopping interrupts penalizes all processes when one process executes a critical section
  - Stops all I/O activity
  - Restricts execution to one process for the entire system
- It also does not prevent real concurrency (in multiprocessors)

Spin locks

- Atomic hardware operation tests a memory location and changes it
- Common atomic instructions:
  - Test-and-set (e.g., **bts** in x86):
    - Write 1 (set) to a memory location and return its old value
  - Compare-and-swap (e.g., **cmpxchg** in x86)
    - Compares the contents of a memory location with a given value. If they are the same, modifies the contents of that memory location to a new value.

Example: Spin lock in GCC

- A lock variable holds:
  - `1` if locked
  - `0` if unlocked
- Check more: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html

```
void spin_lock(int *p) {
    while(!__sync_bool_compare_and_swap(p, 0, 1));
}

void spin_unlock(int volatile *p) {
    asm volatile (""); // acts as a memory barrier.
    *p = 0;
}
```

Example: Spin lock in x86 assembly

- A lock variable holds:
  - `1` if locked
  - `0` if unlocked
- Notice what **cmpxchg** does

```
spin_lock:
    xorl %ecx, %ecx
    incl %ecx
spin_lock_retry:
    xorl %eax, %eax
    ; cmpxchg: Compare EAX with r/m32.
    ; If equal, ZF is set and r32 is loaded into r/m32.
    ; Else, clear ZF and load r/m32 into EAX
    cmpxchgl %ecx, (lock_addr)
    jnz spin_lock_retry
    ret

spin_unlock:
    movl $0 (lock_addr)
    ret
```

How to implement spin locks with test-and-set?

- Test-and-set (e.g., **bts** in x86):
  - Write 1 (set) to a memory location and return its old value

```
void spin_lock(int *p) {
    while(test_and_set(*p) == 1);
}

void spin_unlock(int volatile *p) {
    asm volatile (""); // acts as a memory barrier.
    *p = 0;
}
```

Memory consistency

- Many CPU architectures do not guarantee sequential consistency (for normal memory operations) in multiprocessors
  - I.e., a CPU may observe the effects of other CPU operations in a different order than the program order
- **Memory barriers** are sometimes necessary to prevent memory reorderings
  - But they are slow, so they're used sparingly

Semaphores

High-Level Mutual Exclusion

- Idea is to create a facility with the following properties
  - Permit designer to specify multiple critical sections
  - Allow independent control of each critical section
  - Provide an access policy (e.g., FIFO)
- A single mechanism is sufficient: the **counting semaphore**

Semaphores

- Semaphore: a variable allocated for each item to be protected
- Applications must be programmed to use the semaphore before accessing a shared item
- Semaphore mechanisms guarantee only one process can access the shared item at any given time
- Implementation avoids **busy-waiting**

Semaphore

- Instance can be created dynamically
- Each instance is given a unique name
  - Typically an integer
  - Known as a semaphore ID
- Instance consists of a tuple (**count** and **set**):
  - **Count**: is an integer
  - **Set**: is a set of processes waiting on the semaphore

Operations on semaphores in Xinu

- Create a new semaphore: semcreate()
- Delete an existing semaphore: semdelete()
- Wait on an existing semaphore: wait()
  - Decrements **count**
  - Adds calling process to set of waiting processes if resulting count is negative
- Signal an existing semaphore: signal()
  - Increments **count**
  - Makes a process ready if any is waiting

Key uses of counting semaphores

- Two basic paradigms:
  - Cooperative mutual exclusion
  - Direct synchronization (e.g., producer-consumer)

Mutual exclusion with semaphores

- Initialize: create a mutex semaphore
- Use: bracket critical section with calls to **wait** and **signal**
- Guarantee: only one process can access the critical section at any time (other will be blocked)

```
sid = semcreate (1);

wait(sid);
// ...
// critical section (use shared resource)
// ...
signal(sid);
```

Producer-Consumer Synchronization With Semaphores

- Two semaphores suffices
- Initialize: create producer and consumer semaphores
  - For the producer: count == number of “available storage locations”
  - For the consumer: count == number of “product ready”

Initialization

```
psem = semcreate(buffer-size);
csem = semcreate(0);
```

Producer-Consumer Synchronization With Semaphores

Initialization

```
psem = semcreate(buffer-size);
csem = semcreate(0);
```

Producer

```
while (1) {
    wait(psem);
    // fill the next buffer slot
    signal(csem);
}
```

Consumer

```
while (1) {
    wait(csem);
    // extract_from_buffer_slot;
    signal(psem);
}
```

Semaphore invariant

- Establishes relationship between semaphores concept and implementation
- Makes code easy to create and understand
- Surprisingly elegant:

- a semaphore count of negative N means that the queue contains N waiting processes.
- a nonnegative semaphore count means that the queue is empty

Counting semaphores in Xinu

- Stored in an array of semaphore entries
- Each entry:
  - Corresponds to one instance (one semaphore)
  - Contains an integer count and pointer to list of processes
- Semaphore ID is index into array
- Policy for management of waiting process is FIFO

Process state used with semaphores

- When process is waiting on a semaphore, process is not:
  - Executing
  - Ready
  - Suspended
- **SUSPENDED** state is only used for **suspend**() and **resume**()
- Therefore a new state is needed
- Xinu uses the **WAITING** state for a process blocked on a semaphore

Semaphore code in Xinu

- **wait**() and **signal**()

Summary

- Coordination is critical in concurrent systems
  - Prevents incorrect semantics
  - Enables efficient implementation of producer-consumer
- Mutual exclusion mechanisms to protect critical sections
- Spin locks
- Counting semaphores:
  - Two key fields (count and blocked process list)
  - Prevent busy wait
  - More general than mutual exclusion