CS4231
Parallel and Distributed Algorithms

Lecture 2

Instructor: YU Haifeng

Review of Last Lecture

- Mutual exclusion problem in shared-memory systems
- Software solutions
  - Unsuccessful attempts
  - Peterson's algorithm
  - Bakery algorithm
- Hardware solutions
  - Disabling interrupts to prevent context switch
  - Special machine-level instructions

Today's Roadmap

- "Synchronization Primitives"
- Why do we need synchronization primitives?
- Synchronization primitive: Semaphore
- Synchronization primitive: Monitor

The Busy Wait Problem

- Solutions developed in last lecture has a common busy wait problem
  - Wastes CPU cycles
  - We want to release the CPU to other processes
  - Need OS support

- Synchronization primitives
  - OS-level APIs that the program may call
  - Don’t worry about how they are implemented
- Semaphores
- Monitors

Semaphore Semantics

- Internally, each semaphore has
  - A boolean *value* – initially true
  - A *queue* of blocked processes – initially empty

```
P():
    if (value == false) {
            add myself to queue
            and block;
    }
    value = false;

V():
    value = true;
    if (queue is not empty) {
            wake up one arbitrary
            process on the queue
    }
```

Executed atomically (e.g., interrupt disabled)

(if blocks, will context switch to some other process)

Executed atomically (e.g., interrupt disabled)

Example: P0 invokes P() when value is false, and then P1 invokes V()

Process 0	*Process 1*

P():
if (*value* == false) {
add myself to *queue*
and block;
}
P0 blocks and P1 takes over	

	V():
	value = true;
	if (queue is not empty) {
	wake up one arbitrary
	process on the queue
	}

value = false;	

Semaphore Semantics

- Exactly one process is waken up in V()
  - The process waken up is chosen arbitrarily
  - Some implementations choose the first process on the queue – Should always check the API semantics for the system you are using

Using Semaphore for Mutual Exclusion

- RequestCS() { P(); }
- ReleaseCS() { V(); }
- The nice thing about this mutual exclusion design is no busy waiting

Dining Philosopher Problem (Dijkstra'65)

- 5 philosophers, 5 chopsticks
- Use 5 semaphores, one for each chopstick:
  - Chopstick[1..5]

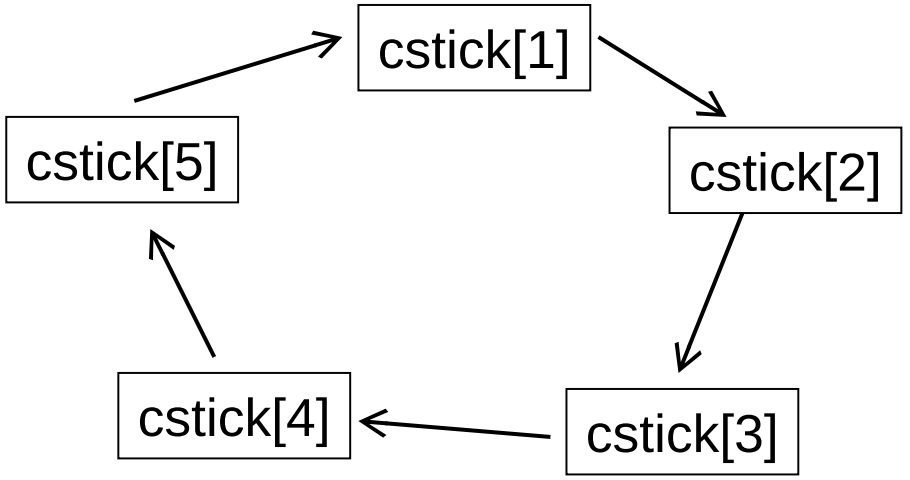
Philosopher i

```
while (true) {
    //think for a while, getting hungry
    chopstick[i].P();
    chopstick[(i+1) % 5].P();
            //eat now; (critical section)
    chopstick[i].V();
    chopstick[(i+1) % 5].V();
}
```

The Danger of Deadlock

philosopher 1	*philosopher 2*	*philosopher 3*	*philosopher 4*	*philosopher 5*
cstick[1].P();	cstick[2].P();	cstick[3].P();	cstick[4].P();	cstick[5].P();
cstick[2].P();	cstick[3].P();	cstick[4].P();	cstick[5].P();	cstick[1].P();

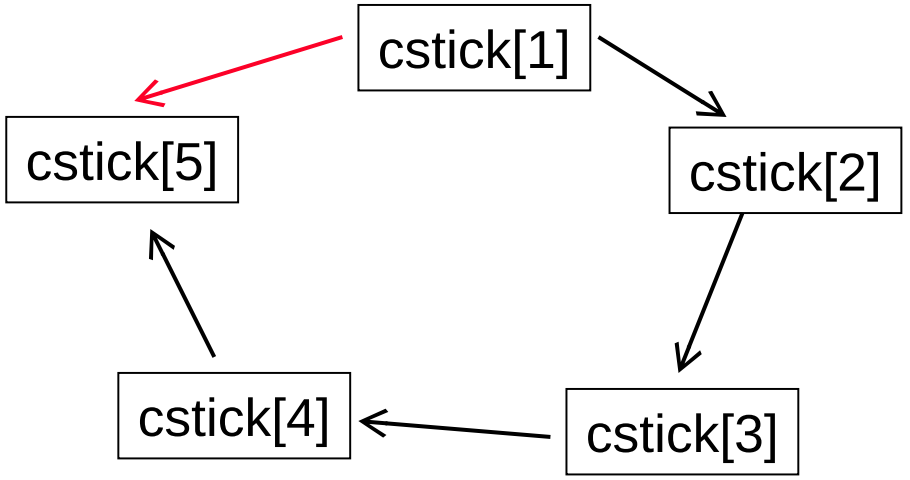
- It is possible for all processes to block
  - Deadlock !



Avoiding Deadlock

philosopher 1	*philosopher 2*	*philosopher 3*	*philosopher 4*	*philosopher 5*
cstick[1].P();	cstick[2].P();	cstick[3].P();	cstick[4].P();	cstick[1].P();
cstick[2].P();	cstick[3].P();	cstick[4].P();	cstick[5].P();	cstick[5].P();

- Avoid cycles (or have a total ordering of the chopsticks)



Today's Roadmap

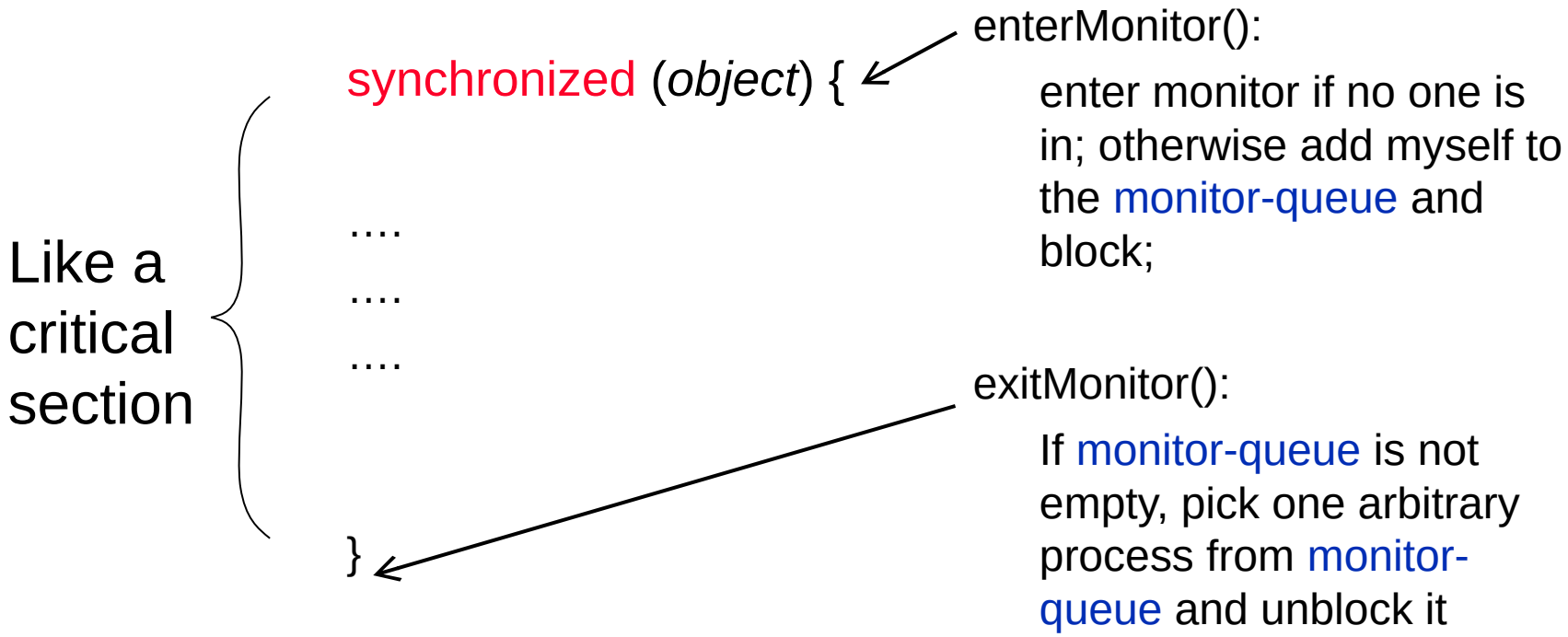
- "Synchronization Primitives"
- Why do we need synchronization primitives?
- Synchronization primitive: Semaphore
- Synchronization primitive: Monitor
  - Monitor semantics
  - Using monitors to solve synchronization problems

Monitors

- Semaphore are quite low-level
  - Monitors are higher-level and easier to use
  - You can always use one to implement the other
- Java only has monitors

Monitor Semantics

- Monitor object M
- Every object in Java is a monitor



sometime we also say that a monitor has a monitor lock

Monitor Semantics

- Each monitor has two queues of blocked processes
  - monitor-queue and wait-queue
- Three special methods for using when inside a monitor

```
synchronized (object) {
    ....
    object.wait();
    ....
    object.notify();
    ....
    object.notifyAll();
    ....
}
```

Add myself to the wait-queue, exit monitor, and then block --- all done atomically

If wait-queue is not empty, pick one arbitrary process from wait-queue and unblock it

If wait-queue is not empty, unblock all processes in wait-queue

Two Kinds of Monitors

Process 0	*Process 1*

synchronized (object) {
….
object.wait();	

	synchronized (object) {
	….
	object.notify();

which process should continue execution at this point?	
}	}

Only one process can be inside the monitor: Two possibilities….

First kind: Hoare-style Monitor

Process 0	*Process 1*

synchronized (object) {
….
object.wait();	

	synchronized (object) {
	….
	object.notify();

…. }	
	…. }

process 0 takes over the execution

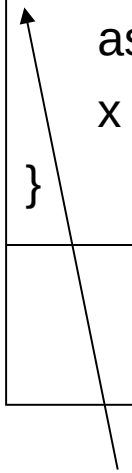
First kind: Hoare-style Monitor

Process 0	*Process 1*
synchronized (object) { if (x != 1) object.wait();	

	synchronized (object) {
	  x=1;
	  object.notify();

assert(x == 1); // x must be 1
x = 2;
}	

	// x may no longer be 1 here }



process 0 takes over the execution

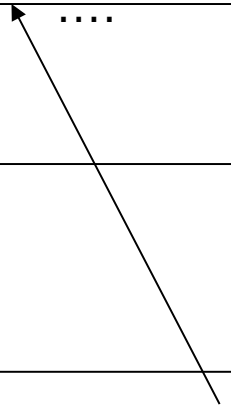
Second kind: Java-style Monitor

Process 0	*Process 1*

synchronized (object) {
….
object.wait();	

	synchronized (object) {
	….
	object.notify();
	….

…..	}
}	



process 1 continues execution

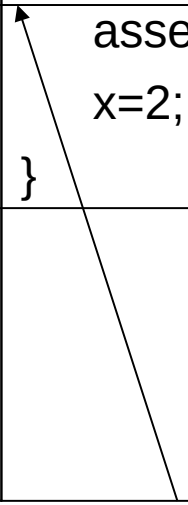
Second kind: Java-style Monitor

Process 0	*Process 1*
synchronized (object) { if (x != 1) object.wait();	

	synchronized (object) {
	  x=1;
	  object.notify();

	assert(x == 1); // x must be 1
	  x=2;
	}

// needs to acquire monitor lock
// x may not be 1 here
}	



process 1 continues execution

Second kind: Java-style Monitor

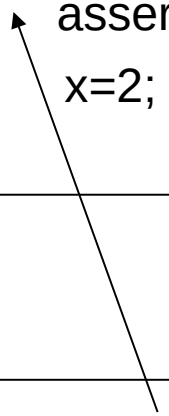
Process 0	*Process 1*

synchronized (object) {
while (x != 1)
object.wait();	

	synchronized (object) {
	x=1;
	object.notify();

	assert(x == 1); // x must be 1
	x=2;
	}

assert(x == 1); // if P0 gets here
}	



process 1 continues execution

Two kinds of monitors: More

- Java-style monitor is more popular
  - But you should always check the semantics of the monitor if you are not using Java
- Synchronization code is extremely difficult (if not impossible) to debug
  - There are bugs that people fail to debug after many years
  - Need to get it right the first time

Nested Monitor in Java

```
synchronized (ObjA) {
    synchronized (ObjB) {
            ObjB.wait();
    }
}
```

```
synchronized (ObjA) {
    synchronized (ObjB) {
            ObjB.notify();
    }
}
```

- Here Java only releases the monitor lock on ObjB and not the monitor lock on ObjA.
- Hence this piece of code will block and will not reach ObjB.notify() – deadlock!
- Different monitor implementations may differ in how nested monitors are implemented – check the spec to tell…

Today's Roadmap

- "Synchronization Primitives"
- Why do we need synchronization primitives?
- Synchronization primitive: Semaphore
- Synchronization primitive: Monitor
  - Monitor semantics
  - Using monitors to solve synchronization problems

The Producer-Consumer Problem

- A circular buffer of size n
- A single producer and a single consumer
- Producer places item to the end of the buffer if
  - Buffer is not full
- Consumer removes item from the head of the buffer if
  - Buffer is not empty

- Example: hard drive as producer and printer as consumer

Using Monitor to Solve the Producer/Consumer problem

object sharedBuffer;

```
void produce() {
    synchronized (sharedBuffer) {
        if (sharedBuffer is full)
            sharedBuffer.wait();
        add an item to sharedBuffer;
        if (sharedBuffer *was* empty)
            sharedBuffer.notify();
    }
}
```

```
void consume() {
    synchronized (sharedBuffer) {
        if (sharedBuffer is empty)
            sharedBuffer.wait();
        remove item from sharedBuffer;
        if (sharedBuffer *was* full)
            sharedBuffer.notify();
    }
}
```

Using Monitor to Solve the Producer/Consumer problem

object sharedBuffer;

```
void produce() {
    synchronized (sharedBuffer) {
        if (sharedBuffer is full)
            sharedBuffer.wait();
        add an item to sharedBuffer;
        if (sharedBuffer *was* empty)
            sharedBuffer.notify();
    }
}
```

notification is lost if no process is waiting – very different semantics from V()

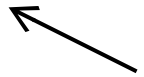
The Reader-Writer Problem

- Multiple readers and writers are accessing a file
  - A writer must have exclusive access
  - But readers may simultaneously access the file

```
int numReader, numWriter; Object object;
```

```
void writeFile() {
    synchronized (object) {
        while (numReader > 0 ||
                numWriter > 0)
            object.wait();
        numWriter = 1;
    }
    // write to file;
    synchronized (object) {
        numWriter = 0;
        object.notifyAll();
    }
}
```

```
void readFile() {
 synchronized (object) {
        while (numWriter > 0)
            object.wait();
        numReader++;
    }
    // read from file;
    synchronized (object) {
        numReader--;
        object.notify();
    }
}
```



you can prove that it must be a writer who is notified

The Starvation Problem

- Writers may get starved if there is a continuous stream of readers
  - There's a way to avoid that….will be your homework…

Summary

- Why do we need synchronization primitives
  - Busy waiting waste CPU
  - But synchronization primitives need OS support
- Semaphore:
  - Using semaphore to solve dining philosophers problem
  - Avoiding deadlocks
- Monitor:
  - Easier to use than semaphores / more popular
  - Two kinds of monitors
  - Using monitor to solve producer-consumer and reader-writer problem

Homework Assignment (on this and next few slides)

- Give a starvation-free solution to the reader-writer problem, using Java-style monitors.

- Give a solution to the following synchronization problem , using Java-style monitors:
  - There are 3 processes (P, Q, and R). Process P repeatedly prints “P”, process Q repeatedly prints “Q”, and process “R” repeatedly prints “R”.
  - The number of “R” printed should always be less than or equal to the sum of the numbers of “P” and “Q” printed.

- Given a solution to the following sleeping barber problem, using Java-style monitors:
  - There is a process called barber. The barber does haircut for any waiting customer. If there is no customer, the barber goes to sleep.
  - There are multiple processes each called a customer. A customer waits for the barber if there is any chair left in the barber shop. Otherwise the customer leaves immediately. If there is an available chair, the customer occupies it. If the barber is sleeping, the customer wakes up the barber. There are total n chairs in the barber shop.
  - You should write out the pseudo-code for the barber process and the customer processes.

- Bring your completed homework to class next week