CSC 256/456: Operating Systems

Synchronization Principles II

John Criswell
University of Rochester

Synchronization Issues

- Race conditions and the need for synchronization
- Critical Section Problem
  - Mutual Exclusion
  - Progress
  - Bounded waiting
- Busy waiting vs. Blocking

The Good

counter++

counter—

Counter
4

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

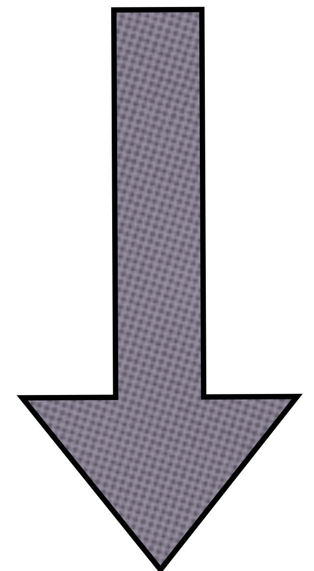
The Good

counter++

counter—

Counter
4

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;



- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

Counter
4

The Other Good

counter++

counter—

Counter
4

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;

The Other Good

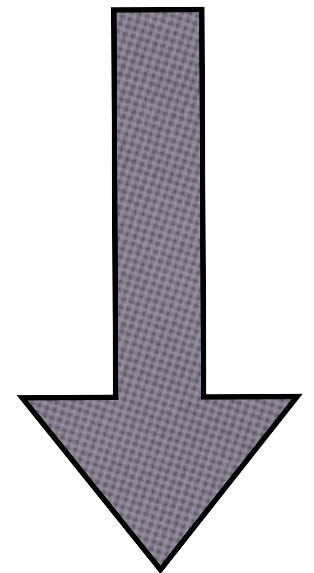
counter++

counter—

Counter
4

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;



Counter
4

The Ugly

counter++

counter—

Counter
4

- register1 = counter;
- register2 = counter;
- register1 = register1 + 1;
- register2 = register2 - 1;
- counter = register2;
- counter = register1;

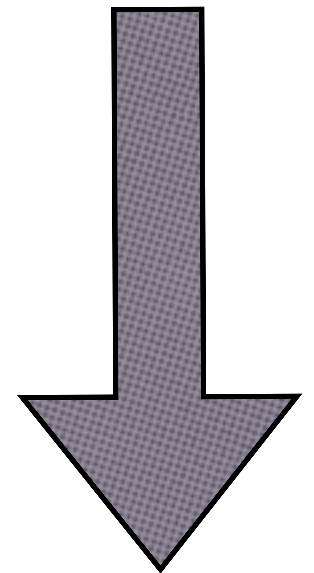
The Ugly

counter++

counter—

Counter
4

- register1 = counter;
- register2 = counter;
- register1 = register1 + 1;
- register2 = register2 - 1;
- counter = register2;
- counter = register1;



Counter
5!

Building Synchronization Mechanisms

- Atomic hardware instructions
- Mutex (binary semaphore)
- Semaphore

Outline

- Classical synchronization problems
  - Bounded buffer
  - Multiple reader/single writers
  - Dining philosophers
- Monitors
- Condition variables

Classical Synchronization Problems

Three Classical Problems

- Bounded Buffer Problem
- Multiple Readers/Single Writer Problem
- Dining Philosophers Problem

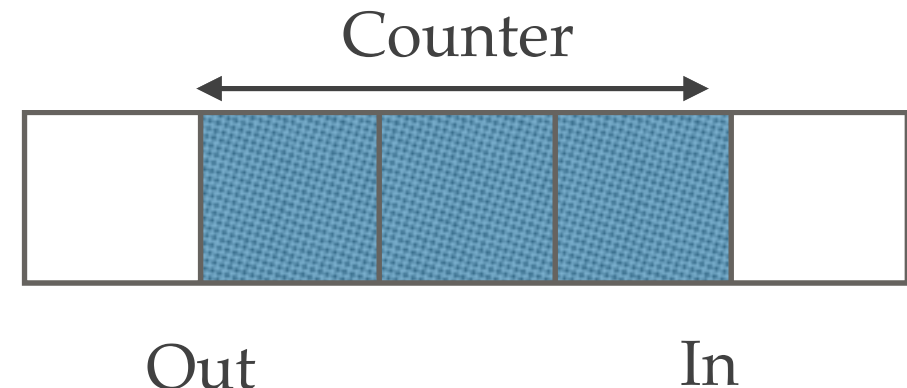
Semaphores on My Mind

- Think of a semaphore as a count of available resources
- When the count hits zero, a process must wait
- wait() (or down()) decreases the count of resources
- signal() (or up()) increases the count of resources

Bounded Buffer Problem

Shared Data

typedef struct {…} item;
item buffer[BUFFER_SIZE];
int in=0, out=0;
int counter = 0;



Producer process

```
item nextProduced;
while (1) {
    while (counter==BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

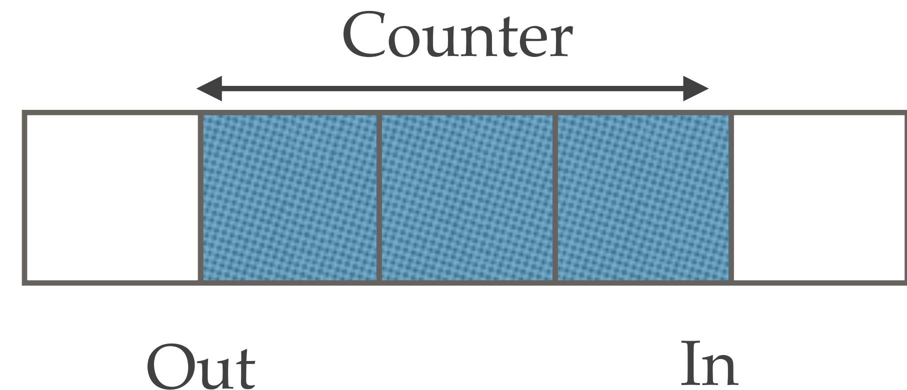
Consumer process

```
item nextConsumed;
while (1) {
    while (counter==0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    counter--;
}
```

Bounded Buffer Solution

<u>Shared Data</u>

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```



Semaphore	Initialized To…	Purpose
mutex	1	Controls access to the buffer
empty	n	Number of empty slots
full	0	Number of full slots

Bounded Buffer Solution

Producer process

```
while (1) {
        produce an item;
        wait (empty);
        wait (mutex);
        add nextp to buffer;
        signal (mutex);
        signal (full);
}
```

Consumer process

```
while (1) {
        wait (full);
        wait (mutex);
        remove an item from buffer;
        signal (mutex);
        signal (empty);
        consume item;
}
```

Can you have a race condition with only reads?

Multiple Readers/Single Writer

- Shared resource
  - Network routing tables
  - Database
- Want to permit concurrent readers
- Writers need exclusive access

Multiple Reader/Single Writer

Shared Data

```
int read_count = 0;
semaphore mutex = 1;
semaphore rw_mutex = 1;
```

Semaphore	Initialized To…	Purpose
mutex	1	Controls access to read_count
rw_mutex	1	Controls write access

Multiple Reader/Single Writer Solution

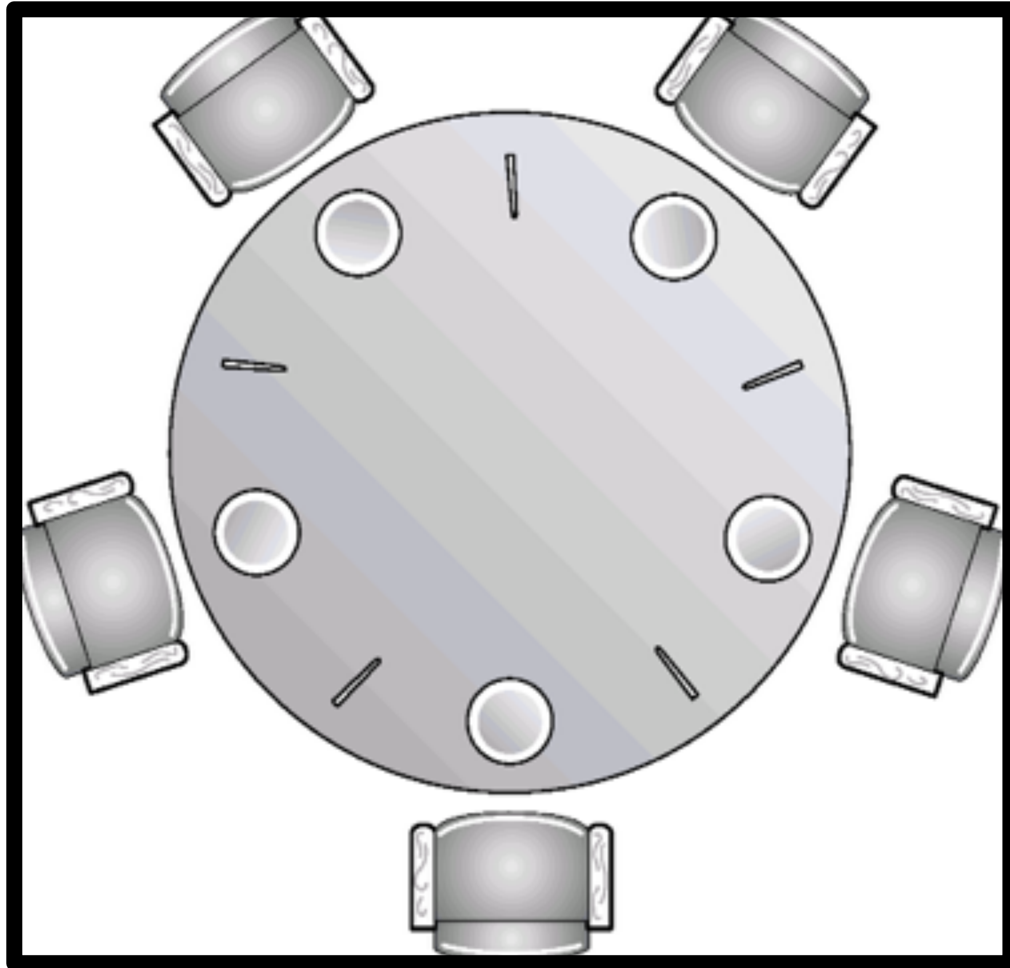
Reader Process

```
while (1) {
        wait (mutex);
        read_count++;
        if (read_count == 1)
                wait (rw_mutex);
        signal (mutex);
        read();
        wait (mutex);
        read_count—;
        if (read_count == 0)
                signal (rw_mutex);
        signal (mutex);
}
```

Writer Process

```
while (1) {
        wait (rw_mutex);
        write();
        signal (rw_mutex);
}
```

Dining-Philosophers Problem



<u>Philosopher i ($1 \le i \le 5$):</u>

```
while (1) {
    eat;
    think;
}
```

- Eating requires both chopsticks (both left and right)
- Need to avoid starvation

Dining Philosophers Solution

Philosopher i

```
while(1) {
    ...
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    eat;
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think;
    ...
};
```

Shared Data

semaphore chopstick[5];
Initially all values are 1;

Dining Philosophers Solution

Philosopher i

```
while(1) {
    ...
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    eat;
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think;
    ...
};
```

Shared Data

semaphore chopstick[5];
Initially all values are 1;

Any potential problems?

Deadlock!

- Each philosopher picks up his/her left chopstick!

Potential Solutions for Deadlock Problem

- Limit eating to 4 philosophers
- Require chopsticks to be picked up in pairs
  - Requires a critical section
- Make the pickup algorithm asymmetric
  - Even philosophers pick up left stick then right stick
  - Odd philosophers pick up right stick then left stick

Summary

- Mechanisms for solving synchronization
  - Atomic hardware instructions
  - Mutex
  - Semaphore
- Using these mechanisms is ***fraught with peril!***

Monitors

Monitors

- High-level programming synchronization construct
- Private variables
- Procedures
  - Only 1 runs at a time
  - Can only access
    - Monitor variables
    - Formal parameters

```
monitor name {
        shared variable declarations
        procedure body P1 (...) {
                . . .
        }
        procedure body Pn (...) {
                . . .
        }
        {
                initialization code
        }
}
```

Condition Variables

- Makes monitors more flexible
- Operations
  - wait(): Suspends process
  - signal(): Wakes up process

```
monitor name {
    condition x;
    condition y;
    procedure body P1 (...) {
        x.wait ();
        . . .
    }
    procedure body Pn (...) {
        . . .
        x.signal();
    }
}
```

Condition Variable Details

- *var*.wait()
  - Puts process to sleep
  - Waits until someone signals on the same variable
- *var*.signal()
  - Wakes up a process waiting on variable *var*
  - Does nothing if no process is waiting on *var*

Condition Variable Semantics

- On signal, can't have two processes in monitor
- Solutions:
  - Waiting process runs, signaling process blocks
  - Signal must be followed by monitor exit
  - Waiting process continues to wait until signaling process exits monitors

Dining Philosophers Solution

```
monitor dp {
	enum {THINKING, HUNGRY, EATING} state[5];
	condition self[5];

	void pickup(int i) {
		state[i] = HUNGRY;
		test(i);
		if (state[i] != EATING)
			self[i].wait();
	}

	void putdown (int i) {
		state[i] = THINKING;
		test((i+4)%5);
		test((i+1)%5);
	}

	void test (int i) {
		if (state[(i+4)%5]!=EATING && state[(i+1)%5]!=EATING && state[i] == HUNGRY) {
			state[i] = EATING;
			self[i].signal();
		}
	}

	void init() {
		for (int i=0; i<5; i++)
			state[i] = THINKING;
	}
}
```

Other Synchronization Methods

Functional Languages

- Pure functional languages have no shared memory
  - Once a value is created, it is never modified
  - No race conditions
- Compiler uses dependencies to parallelize code

Actor Languages

- Use message passing instead of shared memory
- Actors send messages to each other
- Concurrency issues still exist
  - Messages can be asynchronous
  - Messages can be reordered
- Implementation may use shared memory synchronization

Wait-free Data Structures

- Removes the need to wait
  - Operation performs in finite number of steps regardless of the speed of other processes
- Consensus number
  - Ranks ability to implement wait-free systems with hardware atomic operations
  - data structure consensus <= atomic op consensus

Consensus Numbers

Atomic Op	Consensus Number
swap	2
fetch and add	2
test and set	2
compare and swap	infinite

A Note About Systems vs. Theory

- Synchronization is a *combination* of *Systems* and *Theory*
- Theory
  - Explain requirements
  - Prove that approaches work properly
  - Illuminate limitations of approaches (e.g., consensus numbers)
- Systems
  - Hardware support
  - Understand which requirements are important in practice

Theory and Systems work together

Credits and Disclaimer

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, and Kai Shen. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).
- Consensus information from *Wait-Free Synchronization* by Maurice Herlihy