

Synchronization II

Jo, Heeseung

Today's Topics

Spinlock is not enough

- What if a lock is held by others?
- What if a condition is not met inside the critical section?

Higher-level synchronization mechanisms

- Semaphores
- Monitors

Higher-level Synchronization

Motivation

- Spinlocks and disabling interrupts are useful only for very short and simple critical sections
 - Wasteful otherwise
 - These primitives are "primitive" - don't do anything besides mutual exclusion
- Need higher-level synchronization primitives that
 - Block waiters
 - Leave interrupts enabled within the critical section
- Two common high-level primitives:
 - Semaphores: binary (mutex) and counting
 - Monitors: Language construct with condition variables
- We'll use our "atomic" locks as primitives to implement them

Semaphores (1)

Semaphores

- A synchronization primitive higher level than locks
- Invented by Dijkstra in 1968, as part of the "THE" OS
- Does not require busy waiting
- Manipulated atomically through two operations:
 - Wait (S): decrement, block until semaphore is open
= P(), after Dutch word for test, also called down()
 - Signal (S): increment, allow another to enter
= V(), after Dutch word for increment, also called up()

Semaphores (2)

Blocking in semaphores

- Each semaphore **has an associated queue** of processes/threads
- When **wait()** is called by a thread,
 - If semaphore is "open", thread continues
 - If semaphore is "closed", thread blocks, **waits on queue**
- **signal()**
 - Opens the semaphore
 - If thread(s) are waiting on a queue, one thread is unblocked
- In other words, semaphore has history
 - The history is a counter
 - If counter falls below 0, then the semaphore is closed
 - wait() decreases the counter while signal() increases it

Implementing Semaphores

```
typedef struct {
    int value; // 1 or N
    struct process *L;
} semaphore;

void wait (semaphore S) {
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        block ();
    }
}

void signal (semaphore S) {
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup (P);
    }
}
```

wait() / signal()
are critical
sections!

Hence, they must be
executed atomically

HOW??

Algorithm

H/W instruction

Interrupt
disable/enable

Types of Semaphores

Binary semaphore (a.k.a mutex)

- Guarantees mutually exclusive access to resource
- Only one thread/process allowed entry at a time
- Counter is initialized to 1

Counting semaphore

- Represents a resource with many units available
 - e.g., 5 printers
- Allows threads/processes to enter as long as more units are available
- Counter is initialized to N (=units available)

Deadlock and Starvation

Deadlock

- Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
...	...
...	...
signal (S);	signal (Q);
signal (Q);	signal (S);

Starvation

- Indefinite blocking
- A process may never be removed from the semaphore queue in which it is suspended

Priority Inversion

- Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via priority-inheritance protocol

Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Dining-Philosophers Problem
- Readers and Writers Problem
- ...
- ...

Bounded Buffer Problem (1)

Producer/consumer problem

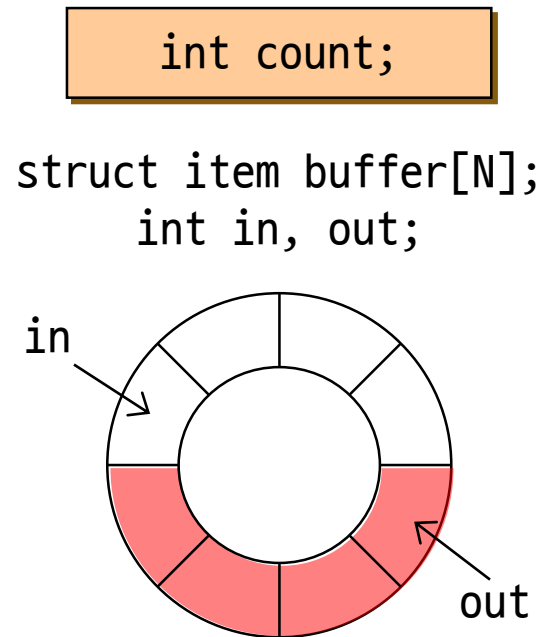
- There is a set of resource buffers shared by producer and consumer
- Producer inserts resources into the buffer
 - Output, disk blocks, memory pages, etc.
- Consumer removes resources from the buffer
- Producer and consumer execute in different rates
 - No serialization of one behind the other
 - Tasks are independent

Bounded Buffer Problem (2)

No synchronization

Producer

```
void produce(data)
{
    while (count==N);
    buffer[in] = data;
    in = (in+1) % N;
    count++;
}
```



Consumer

```
void consume(data)
{
    while (count==0);
    data = buffer[out];
    out = (out+1) % N;
    count--;
}
```

Bounded Buffer Problem (3)

counter++ could be implemented as

- `register1 = count`
- `register1 = register1 + 1`
- `count = register1`

counter-- could be implemented as

- `register2 = count`
- `register2 = register2 - 1`
- `count = register2`

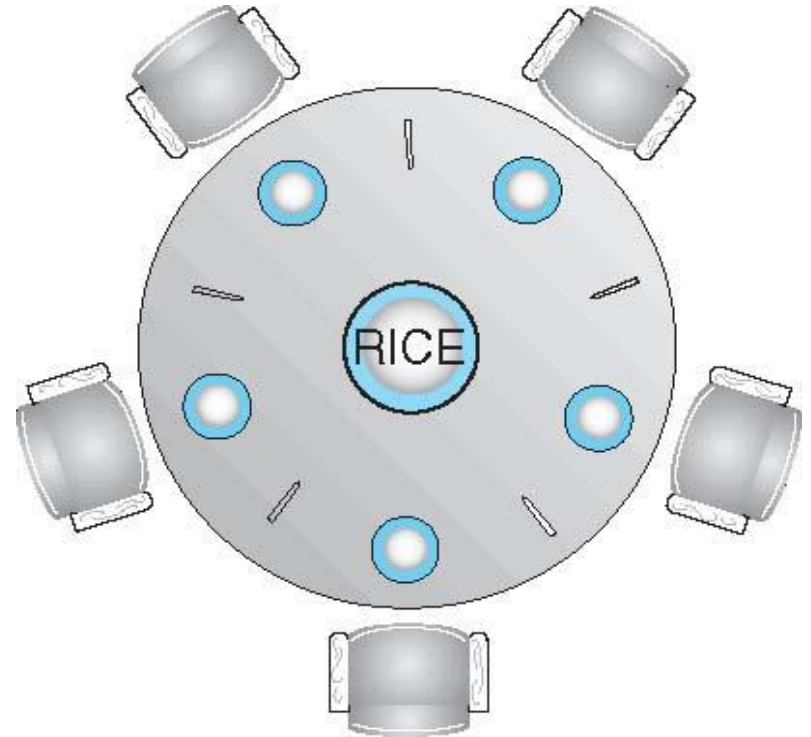
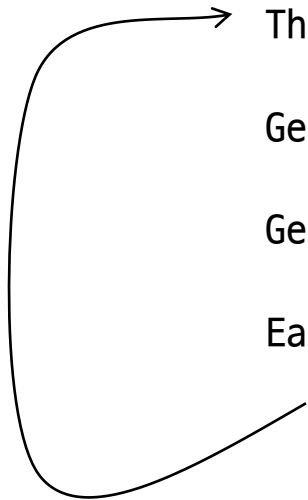
Consider this execution interleaving with "count = 5" initially:

- | | |
|--|------------------------------|
| 1. P: <code>register1 = count</code> | <code>{register1 = 5}</code> |
| 2. P: <code>register1 = register1 + 1</code> | <code>{register1 = 6}</code> |
| 3. C: <code>register2 = count</code> | <code>{register2 = 5}</code> |
| 4. C: <code>register2 = register2 - 1</code> | <code>{register2 = 4}</code> |
| 5. P: <code>count = register1</code> | <code>{count = 6}</code> |
| 6. C: <code>count = register2</code> | <code>{count = 4}</code> |

Dining Philosopher (1)

Dining philosopher problem

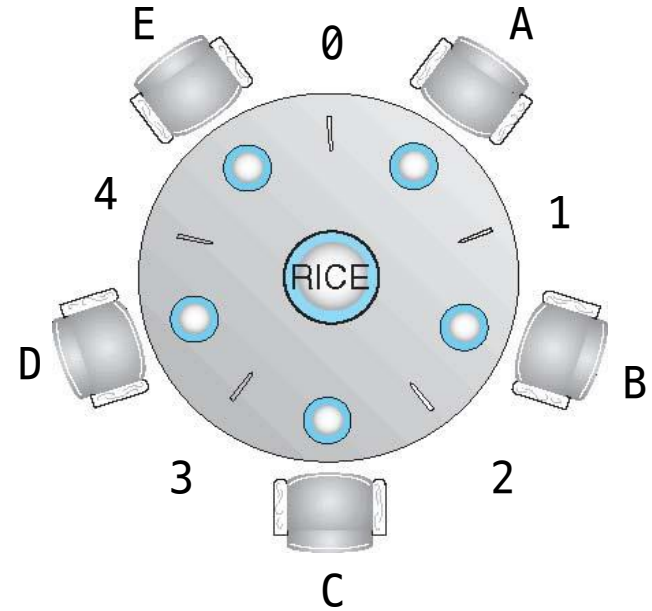
- Dijkstra, 1965
- Life of a philosopher
 - Repeat forever:
 - Thinking
 - Getting hungry
 - Getting two chopsticks
 - Eating



Dining Philosopher (2)

A simple solution

```
Semaphore chopstick[N]; // initialized to 1
void philosopher (int i)
{
    while (1) {
        think ();
        wait (chopstick[i]);
        wait (chopstick[(i+1) % N]);
        eat ();
        signal (chopstick[i]);
        signal (chopstick[(i+1) % N]);
    }
}
```



Problems with Semaphores

Drawbacks

- They are essentially **shared global variables**
 - Can be accessed from anywhere (bad software engineering)
- Used for both critical sections (mutual exclusion) and for coordination (scheduling)
- No control over their use, **no guarantee of proper usage**
- Incorrect use of semaphore operations:
 - `signal (mutex) ... wait (mutex)`
 - `wait (mutex) ... wait (mutex)`
 - Omitting of `wait (mutex)` or `signal (mutex)` (or both)
- Deadlock and starvation

Thus, **hard to use and prone to bugs**

- Another approach: use **programming language support**

Monitors (1)

Monitor

- A programming language construct that supports controlled access to shared data
 - Synchronization code added by compiler, enforced at runtime
 - Allows the safe sharing of an abstract data type among concurrent processes
- A monitor is a software module that encapsulates
 - Shared data structures
 - Procedures that operate on the shared data
 - Synchronization between concurrent processes that invoke those procedures
- Monitor protects the data from unstructured access
 - Guarantees only access data through procedures, hence in legitimate ways

Monitors (2)

Monitor example

- In Java, "synchronized" keyword

```
class A extends Thread {  
    static int x;  
    public void run() {  
        add1();  
        sub1();  
    }  
    void add1() {  
        x=x+1;  
    }  
    void sub1() {  
        x=x-1;  
    }  
}
```

```
class A extends Thread {  
    static int x;  
    public void run() {  
        add1();  
        sub1();  
    }  
    synchronized void add1() {  
        x=x+1;  
    }  
    synchronized void sub1() {  
        x=x-1;  
    }  
}
```

Synchronization Mechanisms

Spinlocks

- Busy waiting

H/W support

- TestAndSet
- SWAP

Disabling interrupts

Semaphores

- Binary semaphore = mutex (\cong lock)
- Counting semaphore

Monitors

- Language construct with condition variables