

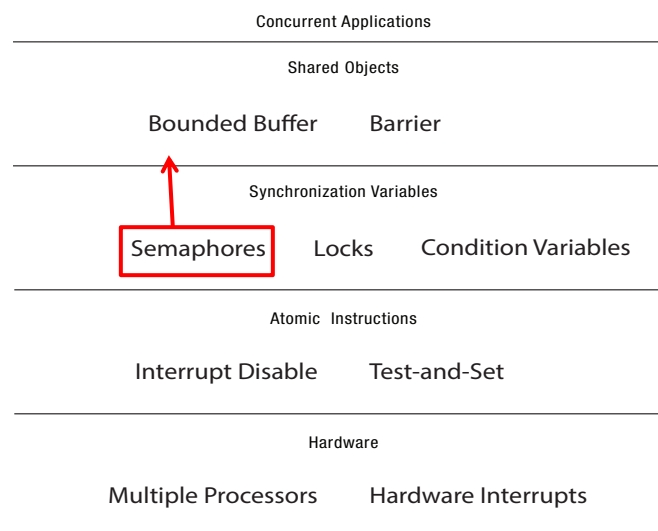
CS 422/522 Design & Implementation
of Operating Systems

Lectures 8-9: Implementing Synchronization

Zhong Shao
Dept. of Computer Science
Yale University

1

The big picture



2

Semaphores (Dijkstra 1965)

- ◆ Semaphores are a kind of generalized lock.
 - * They are the main synchronization primitives used in the earlier Unix.
- ◆ Semaphores have a non-negative integer value, and support two operations:
 - **semaphore->P()**: an atomic operation that waits for semaphore to become positive, then decrements it by 1
 - **semaphore->V()**: an atomic operation that increments semaphore by 1, waking up a waiting P, if any.
- ◆ Semaphores are like integers except:
 - (1) none-negative values; (2) only allow P&V --- can't read/write value except to set it initially; (3) operations must be atomic: two P's that occur together can't decrement the value below zero. Similarly, thread going to sleep in P won't miss wakeup from V, even if they both happen at about the same time.

3

Implementing semaphores

P means "test" (proberen in Dutch)

V means "increment" (verhogen in Dutch)

```
class Semaphore { int value = initialValue; }
```

```
Semaphore::P() {
  Disable interrupts;
  while (value == 0) {
    Put on queue of threads waiting
    for this semaphore;
    Go to sleep;
  }
  value = value - 1;
  Enable interrupts
}
```

```
Semaphore::V() {
  Disable interrupts;
  if anyone on wait queue {
    Take a waiting thread off wait
    queue and put it on the ready
    queue;
  }
  value = value + 1;
  Enable interrupts
}
```

4

Binary semaphores

Like a lock; also known as "mutex"; can only have value 0 or 1 (unlike the previous "counting semaphore" which can be any non-negative integers)

```
class Semaphore { int value = 0 or 1; }
```

```
Semaphore::P() {  
    Disable interrupts;  
    while (value == 0) {  
        Put on queue of threads waiting  
        for this semaphore;  
        Go to sleep;  
    }  
    value = 0;  
    Enable interrupts  
}
```

```
Semaphore::V() {  
    Disable interrupts;  
    if anyone on wait queue {  
        Take a waiting thread off wait  
        queue and put it on the ready  
        queue;  
    }  
    value = 1;  
    Enable interrupts  
}
```

5

How to use semaphores

◆ Binary semaphores can be used for mutual exclusion:

initial value of 1; P() is called before the critical section; and V() is called after the critical section.

```
semaphore->P();  
// critical section goes here  
semaphore->V();
```

◆ Scheduling constraints

- having one thread to wait for something to happen

* Example: Thread::Join, which must wait for a thread to terminate. By setting the initial value to 0 instead of 1, we can implement waiting on a semaphore

◆ Controlling access to a finite resource

6

Scheduling constraints

- ◆ Something must happen after one another

```

Initial value of semaphore = 0;
Fork a child thread
Thread::Join calls P    // will wait until something
                        // makes the semaphore positive
-----
Thread finish calls V   // makes the semaphore positive
                        // and wakes up the thread
                        // waiting in Join
  
```

7

Scheduling with semaphores

- ◆ In general, scheduling dependencies between threads $T_1, T_2, \dots, T_n$ can be enforced with $n-1$ semaphores, $S_1, S_2, \dots, S_{n-1}$ used as follows:
 - T_1 runs and signals $V(S_1)$ when done.
 - T_m waits on S_{m-1} (using P) and signals $V(S_m)$ when done.
- ◆ (contrived) example: schedule $\text{print}(f(x,y))$

```

float x, y, z;
sem  Sx = 0, Sy = 0, Sz = 0;

T1:      T2:      T3:
x = ...;  P(Sx);    P(Sz);
V(Sx);    P(Sy);    print(z);
y = ...;  z = f(x,y); ...
V(Sy);    V(Sz);
...       ...
  
```

8

Producer-consumer with semaphores (1)

◆ Correctness constraints

- * consumer must wait for producer to fill buffers, if all empty (scheduling constraints)
- * producer must wait for consumer to empty buffers, if all full (scheduling constraints)
- * Only one thread can manipulate buffer queue at a time (mutual exclusion)

◆ General rule of thumb: use a separate semaphore for each constraint

Semaphore fullBuffers;	// consumer's constraint
	// if 0, no coke in machine
Semaphore emptyBuffers;	// producer's constraint
	// if 0, nowhere to put more coke
Semaphore mutex;	// mutual exclusion

9

Producer-consumer with semaphores (2)

```

Semaphore fullBuffers = 0; // initially no coke
Semaphore emptyBuffers = numBuffers;
// initially, # of empty slots semaphore used to
// count how many resources there are
Semaphore mutex = 1; // no one using the machine

Producer() {
    emptyBuffers.P(); // check if there is space
                    // for more coke
    mutex.P();       // make sure no one else
                    // is using machine

    put 1 Coke in machine;

    mutex.V();       // ok for others to use machine
    fullBuffers.V(); // tell consumers there is now
                    // a Coke in the machine
}

```

```

Consumer() {
    fullBuffers.P(); // check if there is
                    // a coke in the machine
    mutex.P();       // make sure no one
                    // else is using machine

    take 1 Coke out;

    mutex.V();       // next person's turn
    emptyBuffers.V(); // tell producer
                    // we need more
}

```

What if we have 2 producers and 2 consumers?

10

Order of P&Vs --- what can go wrong

```
Semaphore fullBuffers = 0; // initially no coke
Semaphore emptyBuffers = numBuffers;
// initially, # of empty slots semaphore used to
// count how many resources there are
Semaphore mutex = 1; // no one using the machine
```

```
Producer() {
    mutex.P(); // make sure no one else
              // is using machine
    emptyBuffers.P(); // check if there is space
                    // for more coke

    put 1 Coke in machine;

    fullBuffers.V(); // tell consumers there is now
                   // a Coke in the machine
    mutex.V(); // ok for others to use machine
}
```

```
Consumer() {
    mutex.P(); // make sure no one
              // else is using machine
    fullBuffers.P(); // check if there is
                   // a coke in the machine

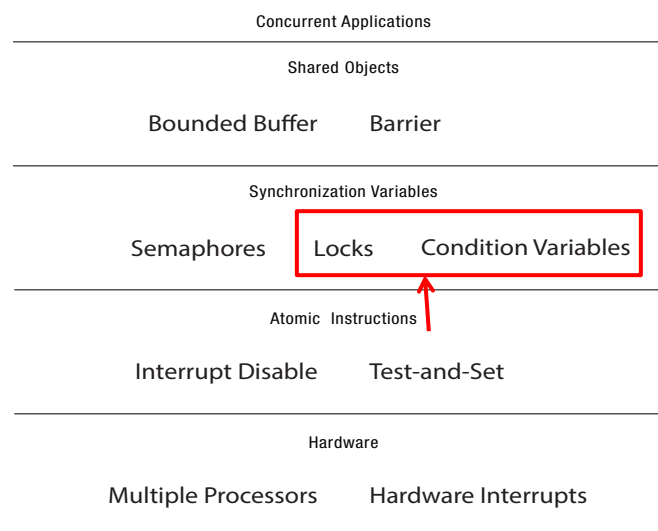
    take 1 Coke out;

    emptyBuffers.V(); // tell producer
                    // we need more
    mutex.V(); // next person's turn
}
```

Deadlock---two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

11

Implementing synchronization



12

Implementing synchronization

Take 1: using memory load/store

- See too much milk solution/Peterson's algorithm

Take 2:

```
Lock::acquire()
{ disable interrupts }
Lock::release()
{ enable interrupts }
```

Take 3: queueing locks

No point on running the threads waiting for locks

13

Lock implementation, uniprocessor

```
Lock::acquire() {
    disableInterrupts();

    if (value == BUSY) {
        waiting.add(myTCB);
        myTCB->state = WAITING;
        next = readyList.remove();
        switch(myTCB, next);
        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }

    enableInterrupts();
}
```

```
class Lock {
    private int value = FREE;
    private Queue waiting;
    public void acquire();
    public void release();
}
```

```
Lock::release() {
    disableInterrupts();
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

14

Multiprocessor

- ◆ Read-modify-write instructions
 - Atomically read a value from memory, operate on it, and then write it back to memory
 - Intervening instructions prevented in hardware
- ◆ Examples
 - Test and set
 - Intel: xchgb, lock prefix
 - Compare and swap
- ◆ Any of these can be used for implementing locks and condition variables!

15

Spinlocks

A spinlock is a lock where the processor waits in a loop for the lock to become free

- Assumes lock will be held for a short time
- Used to protect the CPU scheduler and to implement locks

```
Spinlock::acquire() {
    while (testAndSet(&lockValue) == BUSY)
        ;
}
```

```
Spinlock::release() {
    lockValue = FREE;
    memorybarrier();
}
```

16

How many spinlocks?

- ◆ Various data structures
 - Queue of waiting threads on lock X
 - Queue of waiting threads on lock Y
 - List of threads ready to run
- ◆ One spinlock per kernel?
 - Bottleneck!
- ◆ Instead:
 - One spinlock per lock
 - One spinlock for the scheduler ready list
 - * Per-core ready list: one spinlock per core

17

What thread is currently running?

- ◆ Thread scheduler needs to find the TCB of the currently running thread
 - To suspend and switch to a new thread
 - To check if the current thread holds a lock before acquiring or releasing it
- ◆ On a uniprocessor, easy: just use a global
- ◆ On a multiprocessor, various methods:
 - Compiler dedicates a register (e.g., r31 points to TCB running on this CPU; each CPU has its own r31)
 - If hardware has a special per-processor register, use it
 - Fixed-size stacks: put a pointer to the TCB at the bottom of its stack
 - * Find it by masking the current stack pointer

18

Lock implementation, multiprocessor

```
class Lock {
    private int value = FREE;
    private SpinLock spinLock;
    private Queue waiting; ...}

    Lock::acquire() {
        disableInterrupts();
        spinLock.acquire();

        if (value == BUSY) {
            waiting.add(myTCB);
            scheduler->suspend(&spinlock);
        } else {
            value = BUSY;
            spinLock.release();
        }
        enableInterrupts();
    }
}
```

```
Lock::release() {

    disableInterrupts();
    spinLock.acquire();

    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value = FREE;
    }

    spinLock.release();
    enableInterrupts();
}
```

19

Lock implementation, multiprocessor (cont'd)

```
class Scheduler {
    private:
        Queue readyList;
        SpinLock schedulerSpinLock;
    public:
        void suspend(SpinLock *lock);
        void makeReady(Thread *thread);
}

void
Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();

    readyList.add(thread);
    thread->state = READY;

    schedulerSpinLock.release();
    enableInterrupts();
}
```

```
void
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;

    disableInterrupts();
    schedulerSpinLock.acquire();

    lock->release();

    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread, chosenTCB);
    runningThread->state = RUNNING;

    schedulerSpinLock.release();
    enableInterrupts();
}
```

20

Condition variable implementation, multiprocessor

```

class CV {
    private Queue waiting;
    public void wait(Lock *lock);
    public void signal();
    public void broadcast();
}

// Monitor lock held by current thread.
void CV::wait(Lock *lock) {
    assert(lock.isHeld());
    waiting.add(myTCB);

    // Switch to new thread & release lock.
    scheduler.suspend(&lock);
    lock->acquire();
}

// Monitor lock held by current thread.
void CV::signal() {
    if (waiting.notEmpty()) {
        thread = waiting.remove();
        scheduler.makeReady(thread);
    }
}

void CV::broadcast() {
    while (waiting.notEmpty()) {
        thread = waiting.remove();
        scheduler.makeReady(thread);
    }
}

```

21

Semaphore implementation, a comparison

```

Semaphore::P() {
    disableInterrupts();
    spinLock.acquire();

    if (value == 0) {
        waiting.add(myTCB);
        suspend(&spinlock);
    } else {
        value--;
    }

    spinLock.release();
    enableInterrupts();
}

Semaphore::V() {
    disableInterrupts();
    spinLock.acquire();

    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value++;
    }

    spinLock.release();
    enableInterrupts();
}

```

22

"Semaphores considered harmful!"

- ◆ Using separate lock and condition variable classes makes code more self-documenting and easier to read
 - The code is clearer when the role of each synchronization variable is made clear through explicit typing
- ◆ A stateless condition variable bound to a lock is a better abstraction for generalized waiting than a semaphore
 - Semaphores rely on the programmer to carefully map the object's state to the semaphore's value ...
- ◆ Nevertheless, semaphores are used for synchronizing communication between an I/O device and threads waiting for I/O completion.

23

Implementing Condition Variables using Semaphores (Take 1)

```
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    semaphore.V();
}
```

24

Implementing Condition Variables using Semaphores (Take 2)

```
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    if (semaphore queue is not empty)
        semaphore.V();
}
```

25

Implementing Condition Variables using Semaphores (Take 3)

```
wait(lock) {
    semaphore = new Semaphore;
    queue.Append(semaphore); // queue of waiting threads
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    if (!queue.Empty()) {
        semaphore = queue.Remove();
        semaphore.V(); // wake up waiter
    }
}
```

26

Lock implementation, Linux

- ◆ Most locks are free most of the time
 - Why?
 - Linux implementation takes advantage of this fact
- ◆ Fast path
 - If lock is FREE, and no one is waiting, two instructions to acquire the lock
 - If no one is waiting, two instructions to release the lock
- ◆ Slow path
 - If lock is BUSY or someone is waiting, use multiproc impl.
- ◆ User-level locks
 - Fast path: acquire lock using test&set
 - Slow path: system call to kernel, use kernel lock

27

Lock implementation, Linux

```
struct mutex {
  /* 1: unlocked ;
     0: locked;
     negative : locked,
                possible waiters */

  atomic_t count;
  spinlock_t wait_lock;
  struct list_head wait_list;
};
```

```
// atomic decrement
// %eax is pointer to count

lock decl (%eax)
jns 1f // jump if not signed
      // (if value is now 0)
call slowpath_acquire
1:
```

28

Communicating Sequential Processes (CSP/Google Go)

- ◆ A thread per shared object
 - Only thread allowed to touch object's data
 - To call a method on the object, send thread a message with method name, arguments
 - Thread waits in a loop, get msg, do operation
- ◆ No memory races!

29

Example: Bounded Buffer

```
get() {
  lock.acquire();
  while (front == tail) {
    empty.wait(lock);
  }
  item = buf[front % MAX];
  front++;
  full.signal(lock);
  lock.release();
  return item;
}
```

```
put(item) {
  lock.acquire();
  while ((tail - front) == MAX) {
    full.wait(lock);
  }
  buf[tail % MAX] = item;
  tail++;
  empty.signal(lock);
  lock.release();
}
```

Initially: front = tail = 0; MAX is buffer capacity
empty/full are condition variables

30

Bounded Buffer (CSP)

<pre> while (cmd = getNext()) { if (cmd == GET) { if (front < tail) { // do get // send reply // if pending put, do it // and send reply } else // queue get operation } </pre>	<pre> else { // cmd == PUT if ((tail - front) < MAX) { // do put // send reply // if pending get, do it // and send reply } else // queue put operation } </pre>
--	---

31

Locks/CVs vs. CSP

- ◆ Create a lock on shared data
 - = create a single thread to operate on data
- ◆ Call a method on a shared object
 - = send a message/wait for reply
- ◆ Wait for a condition
 - = queue an operation that can't be completed just yet
- ◆ Signal a condition
 - = perform a queued operation, now enabled

32

Remember the rules

- ◆ Use consistent structure
- ◆ Always use locks and condition variables
- ◆ Always acquire lock at beginning of procedure, release at end
- ◆ Always hold lock when using a condition variable
- ◆ Always wait in while loop
- ◆ Never spin in sleep()