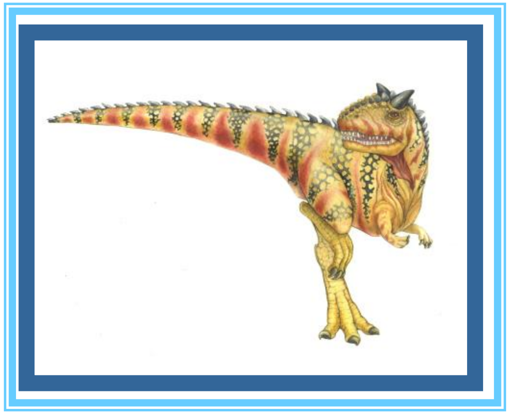
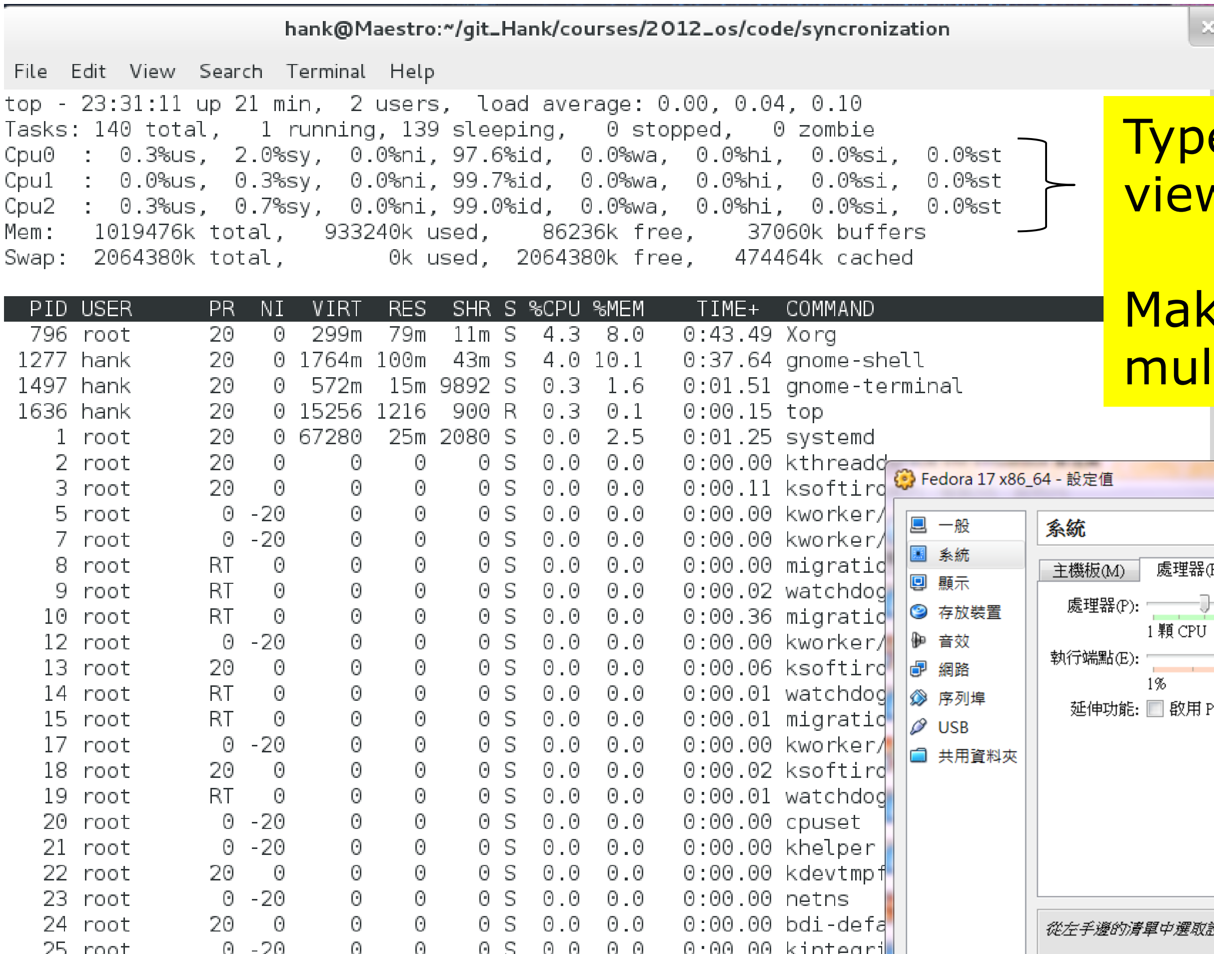
Chapter 6: Synchronization

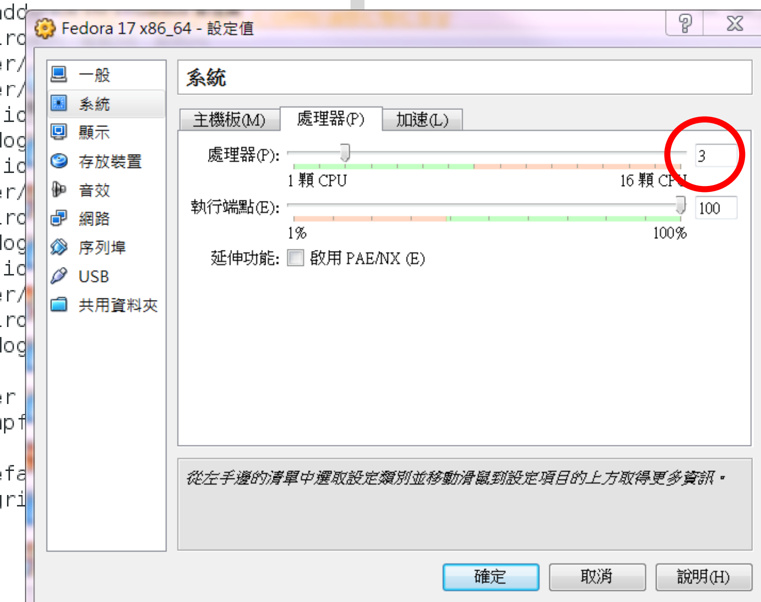


Synchronization


```
hank@Maestro:~/git_Hank/courses/2012_os/code/syncronization
File  Edit  View  Search  Terminal  Help
top - 23:31:11 up 21 min,  2 users,  load average: 0.00, 0.04, 0.10
Tasks: 140 total,   1 running, 139 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.3%us,  2.0%sy,  0.0%ni, 97.6%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.3%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.3%us,  0.7%sy,  0.0%ni, 99.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1019476k total,   933240k used,    86236k free,    37060k buffers
Swap:  2064380k total,        0k used,  2064380k free,   474464k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  796 root      20   0  299m  79m  11m S  4.3  8.0   0:43.49 Xorg
 1277 hank      20   0 1764m 100m  43m S  4.0 10.1   0:37.64 gnome-shell
 1497 hank      20   0  572m  15m 9892 S  0.3  1.6   0:01.51 gnome-terminal
 1636 hank      20   0 15256 1216  900 R  0.3  0.1   0:00.15 top
    1 root      20   0 67280  25m 2080 S  0.0  2.5   0:01.25 systemd
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.11 ksoftirq
    5 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/
    7 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migratio
    9 root      RT   0     0    0    0 S  0.0  0.0   0:00.02 watchdog
   10 root      RT   0     0    0    0 S  0.0  0.0   0:00.36 migratio
   12 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/
   13 root      20   0     0    0    0 S  0.0  0.0   0:00.06 ksoftirq
   14 root      RT   0     0    0    0 S  0.0  0.0   0:00.01 watchdog
   15 root      RT   0     0    0    0 S  0.0  0.0   0:00.01 migratio
   17 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/
   18 root      20   0     0    0    0 S  0.0  0.0   0:00.02 ksoftirq
   19 root      RT   0     0    0    0 S  0.0  0.0   0:00.01 watchdog
   20 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 cpuset
   21 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 khelper
   22 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kdevtmpf
   23 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 netns
   24 root      20   0     0    0    0 S  0.0  0.0   0:00.00 bdi-defa
   25 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kintegri
```

Type ‘1’ to toggle SMP view

Make sure you have multiple cores



Single thread addition

Single_thread_add.cpp

```
unsigned long long int cnt;

void* AdditionX(void* pParam);
void* AdditionY(void* pParam);


int main() {

    timespec start_realtime, end_realtime;
    const unsigned long long int limit = 100000000;
    cnt = 0;


    clock_gettime(CLOCK_REALTIME, &start_realtime );
    printf("start running...\n");

    AdditionX((void*)&limit);
    AdditionY((void*)&limit);

    printf("cnt = %llu\n", cnt);

    clock_gettime(CLOCK_REALTIME, &end_realtime );

    printf("duration = %d nanoseconds\n", (end_realtime.tv_sec - start_realtime.tv_sec)*1000000000 +
                                           (end_realtime.tv_nsec - start_realtime.tv_nsec));

    return 0;
}
```

Single thread addition

Single_thread_add.cpp

```cpp
void* AdditionX(void* pParam)
{
    unsigned long long int limit = *((const unsigned long long int*)pParam);

    printf("thread X  limit = %d\n", limit);

    while(limit-- > 0) {
        cnt++;
    }

    return 0;
}


void* AdditionY(void* pParam)
{
    unsigned long long int limit = *((const unsigned long long int*)pParam);

    printf("thread Y limit = %d\n", limit);

    while(limit-- > 0 ) {
        cnt++;
    }

    return 0;
}
```

Single thread addition

```
hank@Maestro:~/git_Hank/courses/2012_os/code/syncronization
File  Edit  View  Search  Terminal  Help
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 0 (delta 0)
Jnpacking objects: 100% (6/6), done.
From ssh://code.cs.nctu.edu.tw/var/git/person/Hank
   dbddac1..464d0c1  master     -> origin/master
Jpdating dbddac1..464d0c1
Fast-forward
 courses/2012_os/code/syncronization/complex.cpp | 80 --------------------------------
 1 file changed, 80 deletions(-)
 delete mode 100644 courses/2012_os/code/syncronization/complex.cpp
[hank@Maestro syncronization]$ ls -al
total 20
drwxrwxr-x. 2 hank hank 4096 Oct 31 23:45 .
drwxrwxr-x. 6 hank hank 4096 Oct 31 21:21 ..
-rw-rw-r--. 1 hank hank 1399 Oct 31 22:27 multiple_thread_add.cpp
-rw-rw-r--. 1 hank hank 1502 Oct 31 22:30 multiple_thread_inc.cpp
-rw-rw-r--. 1 hank hank 1100 Oct 31 21:47 single_thread_add.cpp
[hank@Maestro syncronization]$
[hank@Maestro syncronization]$ g++ -g ./single_thread_add.cpp -lrt -lpthread
[hank@Maestro syncronization]$ ./a.out
start running...
thread X  limit = 100000000
thread Y limit = 100000000
cnt = 200000000
duration = 456750331 nanoseconds
[hank@Maestro syncronization]$
```

Multithread addition

multiple_thread_add.cpp

```
unsigned long long int cnt;

void* AdditionX(void* pParam);
void* AdditionY(void* pParam);

int main() {

    pthread_t tidX, tidY;
    pthread_attr_t thread_attr;
    timespec start_realtime, end_realtime;

    const unsigned long long int limit = 100000000;

    cnt = 0;

    pthread_attr_init(&thread_attr);
    pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_JOINABLE);


    clock_gettime(CLOCK_REALTIME, &start_realtime );
    printf("start running...\n");


    pthread_create(&tidX, &thread_attr, AdditionX, (void*)&limit );
    pthread_create(&tidY, &thread_attr, AdditionY, (void*)&limit );


    pthread_join(tidX,0);
    pthread_join(tidY,0);

    printf("cnt = %llu\n", cnt);

    clock_gettime(CLOCK_REALTIME, &end_realtime );

    printf("duration = %d nanoseconds\n", (end_realtime.tv_sec - start_realtime.tv_sec)*1000000000 +
                                          (end_realtime.tv_nsec - start_realtime.tv_nsec));


    return 0;
}
```

Multithread Addition

multiple_thread_add.cpp

hank@Maestro:~/git_Hank/courses/2012_os/code/syncronization

File Edit View Search Terminal Help

```
[hank@Maestro syncronization]$ ls -al
total 32
drwxrwxr-x. 2 hank hank  4096 Nov  1 00:02 .
drwxrwxr-x. 6 hank hank  4096 Oct 31 21:21 ..
-rwxrwxr-x. 1 hank hank 11287 Nov  1 00:02 a.out
-rw-rw-r--. 1 hank hank  1379 Nov  1 00:02 multiple_thread_add.cpp
-rw-rw-r--. 1 hank hank  1502 Oct 31 22:30 multiple_thread_inc.cpp
-rw-rw-r--. 1 hank hank  1100 Oct 31 21:47 single_thread_add.cpp
[hank@Maestro syncronization]$ git reset --hard
HEAD is now at 464d0c1 j
[hank@Maestro syncronization]$ ls -al
total 32
drwxrwxr-x. 2 hank hank  4096 Nov  1 00:03 .
drwxrwxr-x. 6 hank hank  4096 Oct 31 21:21 ..
-rwxrwxr-x. 1 hank hank 11287 Nov  1 00:02 a.out
-rw-rw-r--. 1 hank hank  1399 Nov  1 00:03 multiple_thread_add.cpp
-rw-rw-r--. 1 hank hank  1502 Oct 31 22:30 multiple_thread_inc.cpp
-rw-rw-r--. 1 hank hank  1100 Oct 31 21:47 single_thread_add.cpp
[hank@Maestro syncronization]$
[hank@Maestro syncronization]$ g++ -g ./multiple_thread_add.cpp -lrt -lpthread
[hank@Maestro syncronization]$ ./a.out
start running...
thread X  limit = 100000000
thread Y limit = 100000000
cnt = 112770930
duration = 733176733 nanoseconds
[hank@Maestro syncronization]$
```

Multithread Addition

```
void* AdditionX(void* pParam)
{
    unsigned long long int limit =
        *((const unsigned long long int*)pParam);

    printf("thread X  limit = %d\n", limit);

    while(limit-- > 0) {
        cnt++;
    }

    return 0;
}
```

```
void* AdditionY(void* pParam)
{
    unsigned long long int limit =
        *((const unsigned long long int*)pParam);

    printf("thread Y limit = %d\n", limit);

    while(limit-- > 0 ) {
        cnt++;
    }

    return 0;
}
```

The code of AdditionX and the code of AdditionY may interleave

Any Problem?

The code of AdditionX and the code of AdditionY may overlap

Any Problem?

Multithread Addition

```
e for function AdditionX(void*):
<+0>:     push   %rbp
<+1>:     mov    %rsp,%rbp
<+4>:     sub    $0x20,%rsp
<+8>:     mov    %rdi,-0x18(%rbp)
<+12>:    mov    -0x18(%rbp),%rax
<+16>:    mov    (%rax),%rax
<+19>:    mov    %rax,-0x8(%rbp)
<+23>:    mov    -0x8(%rbp),%rax
<+27>:    mov    %rax,%rsi
<+30>:    mov    $0x400bb8,%edi
<+35>:    mov    $0x0,%eax
<+40>:    callq  0x400740 <printf@plt>
<+45>:    jmp    0x400a59 <AdditionX(void*)+65>
<+47>:    mov    0x2005aa(%rip),%rax        # 0x600ff8 <cnt>
<+54>:    add    $0x1,%rax
<+58>:    mov    %rax,0x20059f(%rip)        # 0x600ff8 <cnt>
<+65>:    cmpq   $0x0,-0x8(%rbp)
<+70>:    setne  %al
<+73>:    subq   $0x1,-0x8(%rbp)
<+78>:    test   %al,%al
<+80>:    jne    0x400a47 <AdditionX(void*)+47>
<+82>:    mov    $0x0,%eax
<+87>:    leaveq
<+88>:    retq
```

```
e for function AdditionY(void*):
<+0>:     push   %rbp
<+1>:     mov    %rsp,%rbp
<+4>:     sub    $0x20,%rsp
<+8>:     mov    %rdi,-0x18(%rbp)
<+12>:    mov    -0x18(%rbp),%rax
<+16>:    mov    (%rax),%rax
<+19>:    mov    %rax,-0x8(%rbp)
<+23>:    mov    -0x8(%rbp),%rax
<+27>:    mov    %rax,%rsi
<+30>:    mov    $0x400bce,%edi
<+35>:    mov    $0x0,%eax
<+40>:    callq  0x400740 <printf@plt>
<+45>:    jmp    0x400ab2 <AdditionY(void*)+65>
<+47>:    mov    0x200551(%rip),%rax        # 0x600ff8 <cnt>
<+54>:    add    $0x1,%rax
<+58>:    mov    %rax,0x200546(%rip)        # 0x600ff8 <cnt>
<+65>:    cmpq   $0x0,-0x8(%rbp)
<+70>:    setne  %al
<+73>:    subq   $0x1,-0x8(%rbp)
<+78>:    test   %al,%al
<+80>:    jne    0x400aa0 <AdditionY(void*)+47>
<+82>:    mov    $0x0,%eax
<+87>:    leaveq
<+88>:    retq
```

cnt++ is compiled into multiple instructions

Multithread Inc

multiple_thread_inc.cpp

```
void* AdditionX(void* pParam)
{
    unsigned long long int limit = *((const unsigned long long int*)pParam);

    printf("thread X  limit = %d\n", limit);

    while(limit-- > 0) {
        asm  ("incq %0": "=m" (cnt));
    }

    return 0;
}
```

A single line of assembly

Will this version run correctly on a uniprocessor?

```
void* AdditionY(void* pParam)
{
    unsigned long long int limit = *((const unsigned long long int*)pParam);

    printf("thread Y limit = %d\n", limit);

    while(limit-- > 0 ) {
        asm  ("incq %0": "=m" (cnt));
    }

    return 0;
}
```

Will this version run correctly on a multiprocessor?

Module 6: Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
         while (count == BUFFER_SIZE)
                      ; // do nothing
                buffer [in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                 count--;

                       /*  consume the item in nextConsumed
 }
```

Race Condition

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with “count = 5” initially:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

Critical Section

Producer thread

```
while (true) {

     /*  produce an item and put in nextProduced
     */
     while (count == BUFFER_SIZE ); // do
     nothing
     buffer [in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Consumer thread

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

Code in this "critical section" should not be interleaved or overlapped

Critical Section - How

CriticalSection cs;

Producer thread

```
while (true) {

     /*  produce an item and put in nextProduced
     */
     while (count == BUFFER_SIZE ); // do
     nothing
     buffer [in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     cs.lock();
     count++;
     cs.unlock();
}
```

Consumer thread

```
while (true)  {
     while (count == 0)       ; // do nothing
     nextConsumed =  buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     cs.lock();
     count--;   /*  consume the item in nextConsumed
     cs.unlock();
}
```

Requirements on Lock / Unlock

1. Mutual Exclusion

   If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. Freedom from Deadlock (Progress)

   If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. Freedom from Starvation (Bounded Waiting)

   A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

"Freedom from starvation" implies "freedom from deadlock"

1st attempt on the design of a Critical Section Lock

```
class CriticalSectionLockOne
{
        bool flag[2];

public:
        void lock()
        {
                int i = ThreadID.get();
                int j = 1-i;
                flag[i] = true;
                while(flag[j]) ;  // wait
        }

        void unlock()
        {
                int i = ThreadID.get();
                flag[i] = false;
        }
};
```

Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

1st attempt on the design of a Critical Section Lock

- Does CriticalSectionLockOne satisfy
  - mutual exclusion?
  - freedom from deadlock?
  - freedom from starvation?

1st attempt on the design of a Critical Section Lock

- Does CriticalSectionLockOne satisfy
  - mutual exclusion?
  - freedom from deadlock?
  - freedom from starvation?

1^{st} attempt on the design of a Critical Section Lock

- Proof of that CriticalSectionLockOne satisfies mutual exclusion

Assume thread *A* and *B*.

Let CS_A^j be the interval during which *A* executes the critical section for the *j*-th time.

Let CS_B^k be the interval during which *B* executes the critical section for the *k*-th time.

$\text{write}_A(x=v)$ denotes the event in which *A* assigns value *v* to field *x*.
$\text{read}_A(v==x)$ denotes the event in which *A* reads *v* from field *x*.

Suppose CriticalSectionLockOne does not satisfy mutual exclusion. Then there exist integers *j* and *k* such that $CS_A^j \not\to CS_B^k$ and $CS_B^k \not\to CS_A^j$. Consider each thread's last execution of the lock() method before entering its *k*-th (*j*-th) critical section.

Inspecting the code, we see that

$\text{write}_A(\text{flag}[A]=\text{true}) \to \text{read}_A(\text{flag}[B]==\text{false}) \to CS_A$ (1)
$\text{write}_B(\text{flag}[B]=\text{true}) \to \text{read}_B(\text{flag}[A]==\text{false}) \to CS_B$ (2)
$\text{read}_A(\text{flag}[B]==\text{false}) \to \text{write}_B(\text{flag}[B]=\text{true})$ (3)

By transitivity of the precedence order, we have

$$\text{write}_A(\text{flag}[A]=\text{true}) \to \text{read}_A(\text{flag}[B]==\text{false}) \to \text{write}_B(\text{flag}[B]=\text{true}) \to \text{read}_B(\text{flag}[A]==\text{false})$$

It follows that $\text{write}_A(\text{flag}[A]=\text{true}) \to \text{read}_B(\text{flag}[A]==\text{false})$ without an intervening write to the flag[] array, a contradiction.

1st attempt on the design of a Critical Section Lock

- CriticalSectionLockOne deadlocks if $write_A(flag[A]=true)$ and $write_B(flag[B]=true)$ events occur before $read_A(flag[B])$ and $read_B(flag[A])$ events, then both threads wait forever.

2nd attempt on the design of a Critical Section Lock

```
class CriticalSectionLockTwo
{
        int victim;

public:
        void lock()
        {
                int i = ThreadID.get();
                victim = i;  // let the other go first
                while(victim ==i) ;      // wait
        }

        void unlock() {}
};
```

Exercise: prove that CriticalSectionLockTwo satisfies
mutual exclusion.

But CriticalSectionLockTwo deadlocks if one thread runs completely before the other

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int victim;
  - Boolean flag[2]
- The variable victim indicates whose turn it is to wait.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Peterson Lock

```
class CriticalSection_PetersonLock
{
        bool flag[2];
        int victim;
public:
        void lock()
        {
                int i = ThreadID.get();
                int j = 1-i;
                flag[i] = true;  // I’m ready
                victim = i;  // you go first
                while(flag[j] && victim==i) ; // wait
        }

        void unlock()
        {
                int i = ThreadID.get();
                flag[i] = false;  // I’m not ready
        }
};
```

PetersonLock satisfies mutual exclusion

Suppose not. Consider the last execution of the lock method() by threads A and B. Inspecting the code, we see that

$$\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow \text{write}_A(\text{victim}=A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{write}_B(\text{victim}=B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

WLOG, assume that A was the last thread to write to the victim field

$$\text{write}_B(\text{victim}=B) \rightarrow \text{write}_A(\text{victim}=A) \quad (3)$$

Eq.(3) implies that A observed victim to be A in Eq. (1). Since A nevertheless entered its critical section, it must have observed flag[B] to be *false*, so we have

$$\text{write}_A(\text{victim}=\text{A}) \rightarrow \text{read}_A(\text{flag}[B]==\text{false}) \quad (4)$$

Eq.(2) ~ Eq.(4), together with the transitivity of precedence order, imply

$$\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{write}_B(\text{victim}=B) \rightarrow \text{write}_A(\text{victim}=A) \rightarrow \text{read}_A(\text{flag}[B]==\textit{false})$$

It follows that $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{read}_A(\text{flag}[B]==\textit{false})$.
This is a contradiction because no other write to flag[B] was performed before the critical section executions. $\square$

PetersonLock is starvation-free

Suppose not. WLOG, suppose that *A* runs forever in the lock() method. It must be executing the **while** statement, waiting until either flag[*B*] becomes *false* or victim is set to *B.*

What is *B* doing while *A* fails to make progress?

> Perhaps B is repeatedly entering and leaving its critical section. If so, however, then *B* sets victim to *B* as soon as it reenters the critical section. Once victim is set to *B,* it does not change, and A must eventually return from the lock() method(), a contradiction.
>
> So it must be that *B* is also stuck in the lock() method call, waiting until either flag[*A*] becomes false or victim is set to *A*. But victim cannot be both A and B, a contradiction. □

x86 Guaranteed Atomic Operations

- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1

8.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

Accesses to cacheable memory that are split across cache lines and page boundaries are not guaranteed to be atomic by the Intel Core 2 Duo, Intel® Atom™, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, and P6 family processors provide bus control signals that permit external memory subsystems to make split accesses atomic; however, nonaligned data accesses will seriously impact the performance of the processor and should be avoided.

x86 memory ordering guarantees

- [Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1]

CHAPTER 8
MULTIPLE-PROCESSOR MANAGEMENT

Peterson Lock in action

- multithread_add_peterson.html

```
class Peterson
{
    volatile bool X,Y;
    volatile pthread_t victim;

public:

    Peterson()
    {
        X = Y = false;
        victim = -1;
    }

    void lock()
    {
        if ( pthread_self()==tidX) { // thread X wanna to acquire lock
            X = true;
            victim = tidX;
            asm volatile("mfence" :::  "memory");
            while(Y && victim==tidX) ;  // spin
        }
        else {  //thread Y wanna acquire lock
            Y = true;
            victim = tidY;
            asm volatile("mfence" :::  "memory");
            while(X && victim==tidY); // spin
        }
    }
```

Prevent instruction reordering by processor

Prevent instruction reordering by optimizing compiler

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

```
void lock()
{
      asm("cli"); // disable interrupt
}
```

```
void unlock()
{
      asm("sti"); // enable interrupt
}
```

- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words
  - http://en.wikipedia.org/wiki/Read-modify-write

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
            critical section
    release lock
            remainder section
} while (TRUE);
```

TestAndndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv:
 }
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
        while ( TestAndSet (&lock ))
                ;   // do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

} while (TRUE);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                    //    critical section

         lock = FALSE;

                   //      remainder section

 } while (TRUE);
```

xchg (swap) on x86

*C:\Users\Hank\Downloads\postgresql-9.2.1.tar\postgresql-9.2.1\postgresql-9.2.1\src\include\storage\s_lock.h - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

s_lock.h

```c

#ifdef __x86_64__        /* AMD Opteron, Intel EM64T */
#define HAS_TEST_AND_SET

typedef unsigned char slock_t;

#define TAS(lock) tas(lock)

static __inline__ int
tas(volatile slock_t *lock)
{
    register slock_t _res = 1;

    /*
     * On Opteron, using a non-locking test before the locking instruction
     * is a huge loss.  On EM64T, it appears to be a wash or small loss,
     * so we needn't bother to try to distinguish the sub-architectures.
     */
    __asm__ __volatile__(
        "   lock            \n"
        "   xchgb   %0,%1   \n"
:       "+q"(_res), "+m"(*lock)
:
:       "memory", "cc");
    return (int) _res;
}

#define SPIN_DELAY() spin_delay()
```

C++ source file | length : 26285 | lines : 1027 | Ln : 214 | Col : 18 | Sel : 0 | UNIX | ANSI | INS

Critical sections with test-and-set, swap

- Support more than two processes
  - Baseline Peterson lock supports only 2 threads
- Are the proposed solutions correct?
  - Mutual exclusion
  - Deadlock freedom
  - Starvation freedom

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;                     Wait until released or no one busy.
        // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;                Look for a waiting process.
        if (j == i)
                lock = FALSE;                   No process waiting.
        else
                waiting[j] = FALSE;             Process j is waiting. Release it.
        // remainder section
} while (TRUE);
```

Semaphore

- Invented by Edsger Dijkstra in 1965
- Synchronization tool that does not require busy waiting (spinning)
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks

Semaphore

- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
        // Critical Section
    signal (mutex);
            // remainder section
} while (TRUE);
```

Semaphore Implementation

- Implementation of wait and signal
  - ```
    wait (S) {
          while S <= 0
              yield(); // give up CPU time slices
            S--;
     }
    ```
  - ```
    signal (S) {
       S++;
    }
    ```
- Both methods are atomic
  - The execution of wait(S) cannot be interrupted by other executions of wait(S) or executions of signal(S)
  - The execution of signal(S) cannot be interrupted by other executions of wait(S) or executions of signal(S)
  - Interrupts can occur during executions of wait(S) and signal(S) (preemptive scheduling still works !)

Semaphore Implementation

- Think about the above implementation with two threads on one CPU
  - Does it satisfy mutual exclusion, deadlock freedom, and starvation freedom?
  - Any advantage over spinning ?
    - Peterson lock (software based mutual exclusion)
    - Atomic test and set / swap
  - Any potential issue?
- What if there are many threads (say 100) contending for a lock ?
  - If one thread acquires the lock and is preempted before releasing it, what will we expect?
- There is also the risk of starvation !
  - A thread may get caught in an endless yield loop while other threads repeated enter and exit the critical section.

Semaphore Implementation

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the scheduler ready queue.

Semaphore Implementation

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

The implementation is correct only if the two methods are atomic !

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

How to ensure atomicity?

```
typedef struct __lock_t {
        int flag;
        int guard;
        queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
        m->flag = 0;
        m->guard = 0;
        queue_init(m->q);
}

void lock(lock_t *m) {
        while (TestAndSet(&m->guard, 1) == 1)      ; //acquire guard lock by spinning
        if (m->flag == 0) {
              m->flag = 1; // lock is acquired
              m->guard = 0;
        } else {
              queue_add(m->q, gettid());
              m->guard = 0;
              block();
        }
}

void unlock(lock_t *m) {
        while (TestAndSet(&m->guard, 1) == 1)      ; //acquire guard lock by spinning
        if (queue_empty(m->q))
              m->flag = 0; // let go of lock; no one wants it
        else
              wakeup(queue_remove(m->q)); // hold lock (for next thread!)
        m->guard = 0;
}
```

wakeup/waiting race

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

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do  {

     //   produce an item
     wait (empty);
     //  add the item to the  buffer
      signal (full);
} while (TRUE);
```

- The structure of the consumer process

```
do {
        wait (full);
        //  remove an item from  buffer
        signal (empty);
          //  consume the item
  } while (TRUE);
```

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do  {

    //   produce an item

    wait (empty);
    wait (mutex);

    //  add the item to the  buffer

     signal (mutex);
     signal (full);
} while (TRUE);
```

- The structure of the consumer process

```
do {
       wait (full);
        wait (mutex);

         //  remove an item from  buffer

         signal (mutex);
         signal (empty);
         //  consume the item
  } while (TRUE);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

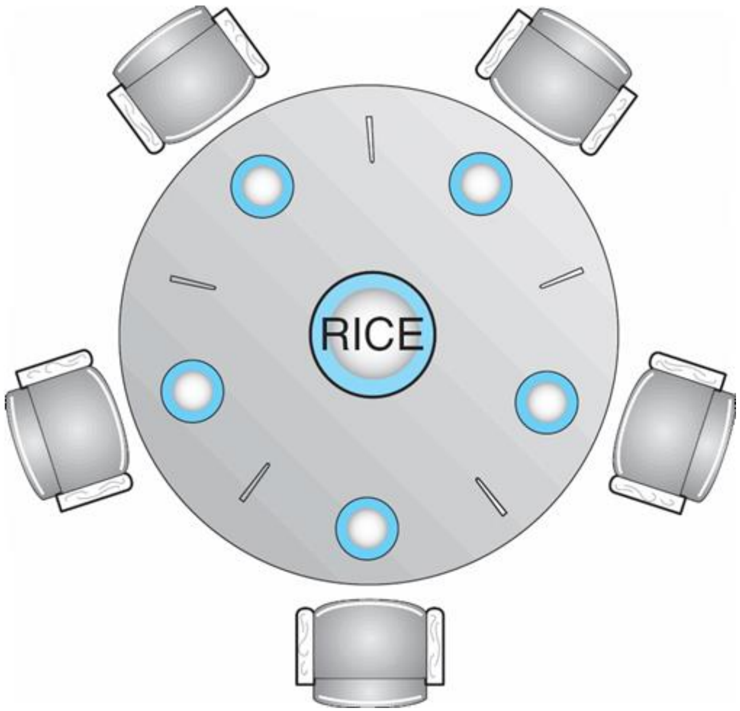
```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
    } while (TRUE);
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
do  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

             //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            //  think

} while (TRUE);
```

Problems with Semaphores

- Used for 2 independent purposes
  - Mutual exclusion
  - Condition synchronization
- Hard to get right
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
  - Small mistake easily leads to deadlock / livelock
- Would it be nice to have?
  - Separation of mutual exclusion and condition synchronization
  - Automatic *wait* and *signal*

Monitors

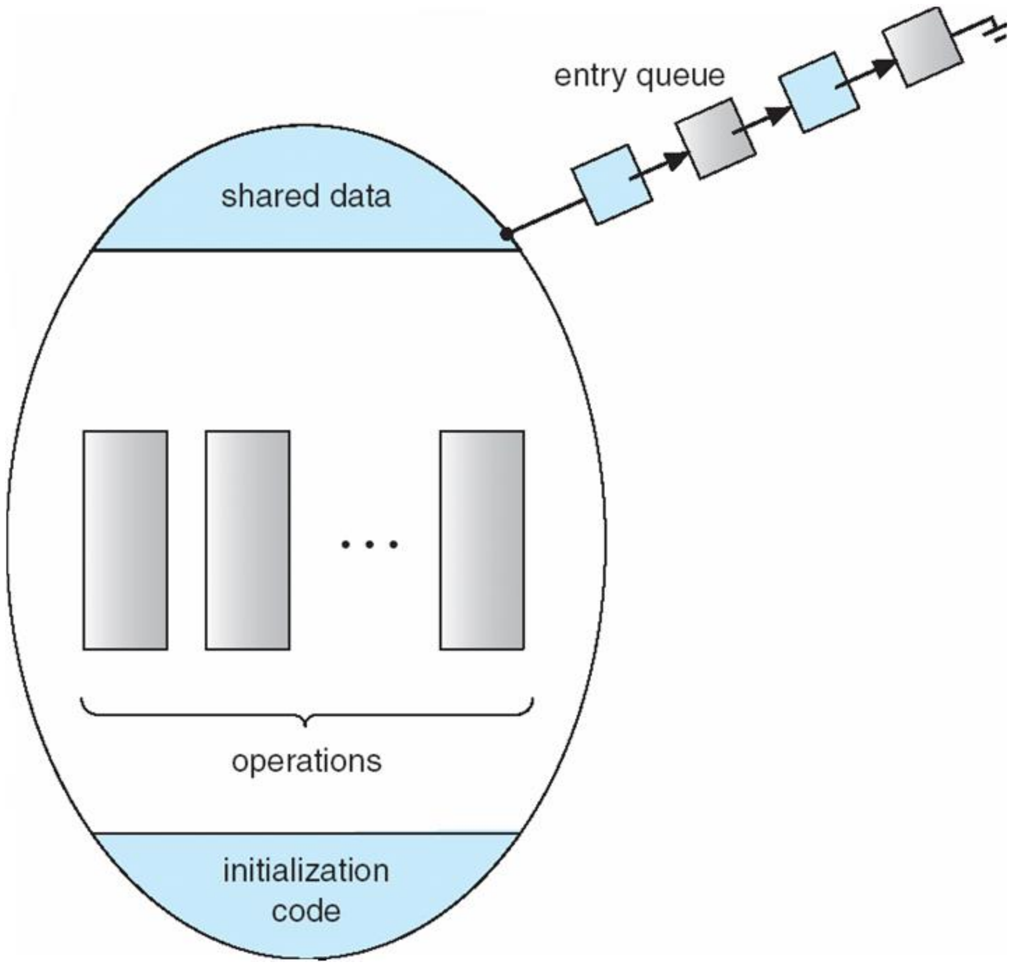
- Invented by Tony Hoare in 1974
- Like a C++ class
  - Consists of *vars* and *procedures*
  - 3 key differences form a regular class:
    - Only one thread in a monitor at a time (automatic mutual exclusion)
    - Special type of variable, called “condition variable”
      - 3 special ops on a condition variable: *wait*, *signal*, and *broadcast*
    - No public variables allowed (must call procedures to access variables)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
              …

    procedure Pn (…) {……}

    Initialization code ( ….) { … }
              …
}
```

Schematic view of a Monitor



Bounded Buffer by Monitor

```
BoundedBuffer  {

    int BUFFER[MAX_SIZE];
    int head, tail, size;

     Enque (int v)  {
         BUFFER[tail] = v;
         tail = (tail+1) % MAX_SIZE;
          size++;
     }

     Deque (int v)  {
           int i = head;
         head = (head+1) % MAX_SIZE;
         size--;
         return BUFFER[i];
     }

      Init ()   {
         head = tail = size = 0;
      }
};
```

Any problem?

Bounded Buffer by Monitor

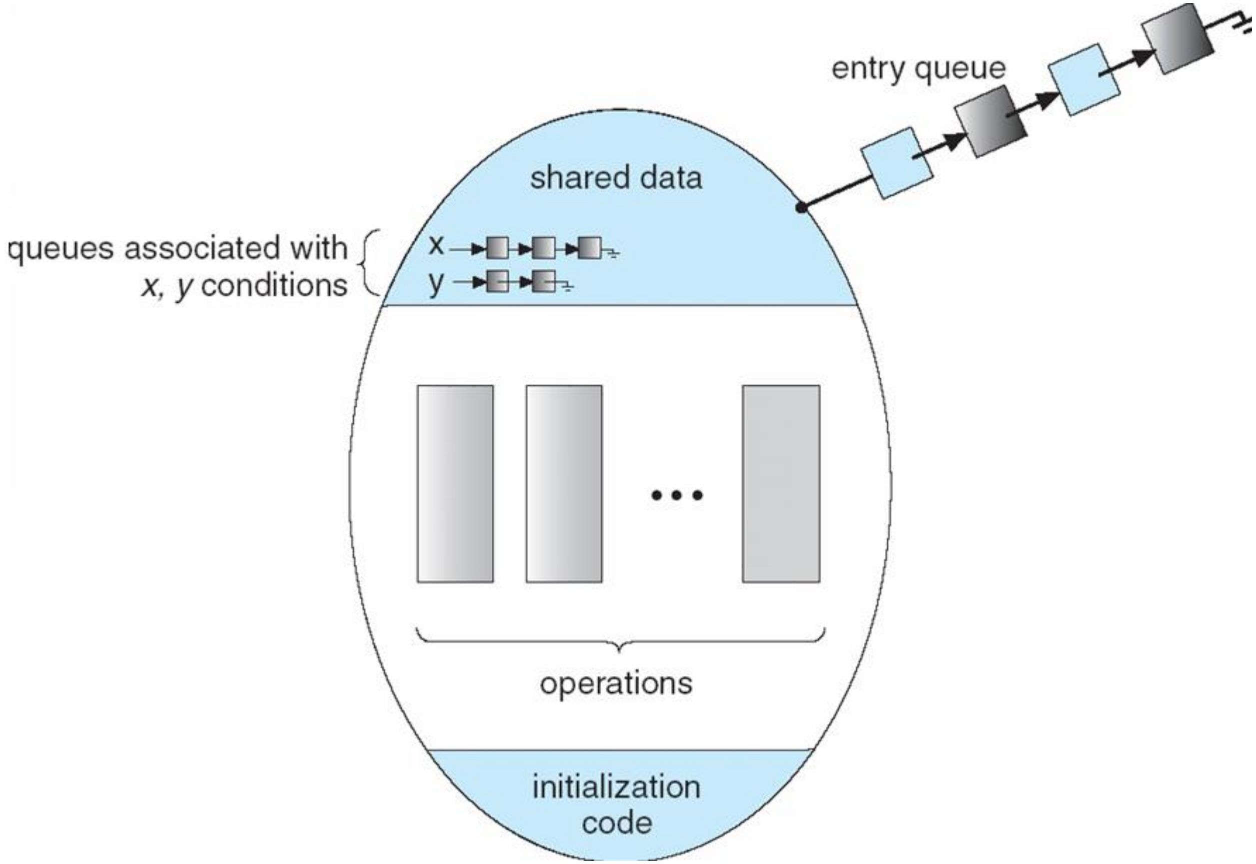
```
BoundedBuffer  {

    int BUFFER[MAX_SIZE];
    int head, tail, size;

     Enque (int v)  {
         while( size == MAX_SIZE);
         BUFFER[tail] = v;
         tail = (tail+1) % MAX_SIZE;
         size++;
     }

     Deque (int v)  {
          while(size==0);
          int i = head;
         head = (head+1) % MAX_SIZE;
         size--;
         return BUFFER[i];
     }
```

Any problem?

Condition Variables

- Need a mechanism for condition synchronization
  - condition x, y;
  - Automatic *unlock* and *lock* for mutual exclusion
- Two operations on a condition variable:
  - cond.wait ()
    - Thread is put on queue for “cond”, goes to sleep.
  - cond.signal ()
    - If queue for “cond” not empty, wake up on thread
  - cond.broadcast()
    - Wake up all threads waiting on queue for “cond”

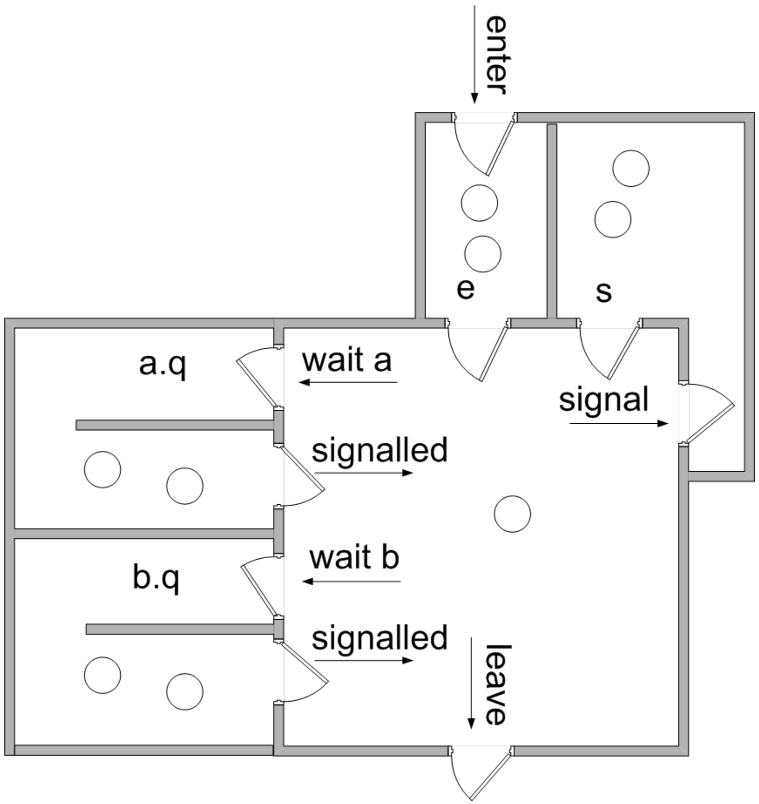
Monitor with Condition Variables



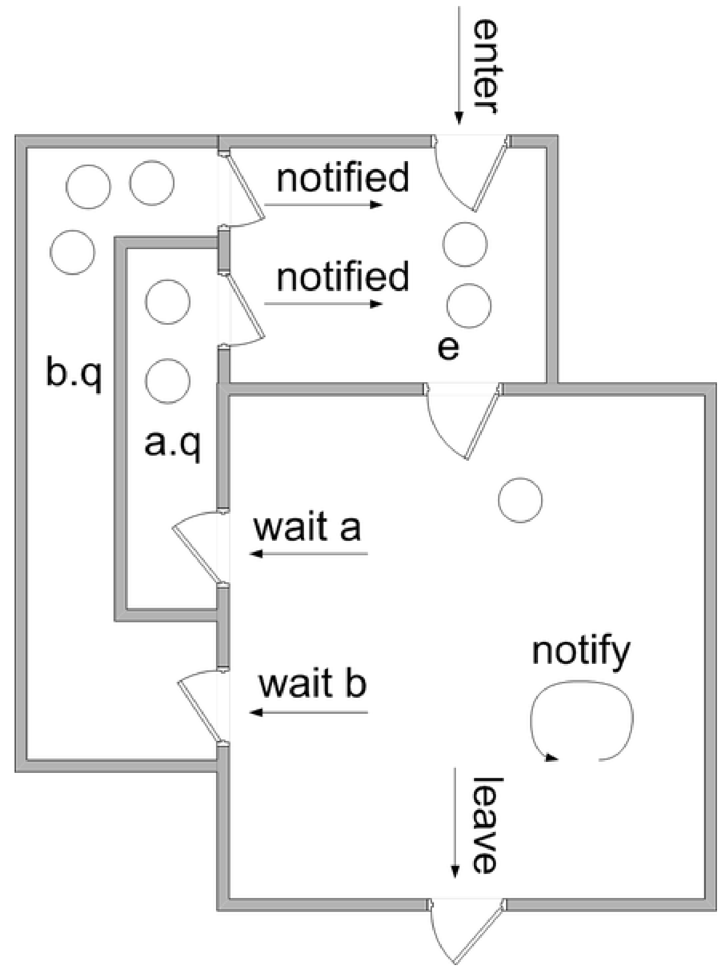
Semantics of Signal

- Signal and Wait (Hoare-style)
  - Signaler passes lock, CPU to waiter; waiter runs immediately
  - Waiter gives lock, CPU back to signaler when
    - It exits critical section
    - Or, it waits again
- Signal and Continue (Mesa-style)
  - signaler continues executing
  - waiter put on ready queue
  - when waiter actually gets to run
    - May have to wait for lock again
    - State may have changed! Use “while”, not “if”
  - Used in Java, Pthread, …)
- http://www.cs.mtu.edu/~shene/NSF-3/e-Book/MONITOR/monitor-types.html

Hoare Style vs Mesa Style



Hoare Style



Mesa Style

Bounded Buffer by Monitor

```
BoundedBuffer {

    int BUFFER[MAX_SIZE];
    int head, tail, size;
    cond full, empty;

     Enqueue (int v) {
         while( size == MAX_SIZE)
                 full.wait();
         BUFFER[tail] = v;
         tail = (tail+1) % MAX_SIZE;
         size++;
          if (size ==1) empty.signal();
     }

     Deque (int v) {
          while(size==0)
                empty.wait();
          int i = head;
         head = (head+1) % MAX_SIZE;
         size--;
         if ( size == MAX_SIZE-1)
                full.signal();
         return BUFFER[i];
     }

…………
}
```

Solution to Dining Philosophers

```
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 left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
      }
```

Solution to Dining Philosophers (cont)

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

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

Solution to Dining Philosophers (cont)

- Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

  DiningPhilosophters.pickup (i);

  EAT

  DiningPhilosophers.putdown (i);

Hoare Style Monitor Implementation (using semaphores)

- Need mutual exclusion semaphore mutex (init to 1) so that only one process is active within monitor
- Need a semaphore next (next to exit) for the signaling process to suspend itself
  - initialized to zero
- next_count is number of processes blocked on next
- Before exiting a procedure, process must either:
  - Signal other waiting processes in monitor next before exiting, or
  - Signal mutex and exit

Monitor Implementation (Hoare Style)

The monitor “compiler” has to automatically insert this code into compiled procedures:

```
Procedure F:
    wait(mutex);
    ...
    body of F
    ...
    if (next_count>0)
            signal(next);
    else
            signal(mutex);
end;
```

Condition Variable Implementation (Hoare)

Each condition x has a count, and a standard semaphore (with associated queue) initialized to 0

```
x.wait() {
    x.count++;
    if (next_count>0)
        signal(next);
    else
        signal(mutex);
    wait(x.sem);
    x.count--;
}
```

```
x.signal() {
    if (x.count >0){
        next_count++;
        signal(x.sem);
        wait(next);
        next_count--;
    }
}
```

Monitor Implementation (Mesa Style)

- Need mutual exclusion semaphore mutex (init to 1) so that only one process is active within monitor
- The monitor "compiler" has to automatically insert this code into compiled procedures:

```
Procedure F:
    wait(mutex);
    ...
    body of F
    ...
    signal(mutex);
end;
```

Chapter 7

Condition Variable Implementation (Mesa)

Each condition x has a count, and a standard semaphore (with associated queue) initialized to 0

```
x.wait() {
    x.count++;
    signal(mutex);
    wait(x.sem);
    wait(mutex);
}
```

```
x.signal() {
    if (x.count >0){
        x.count--;
        signal(x.sem);
    }
}
```

Chapter 7

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
            if (busy)
                x.wait(time);
            busy = TRUE;
    }

    void release() {
            busy = FALSE;
            x.signal();
    }

    initialization code() {
          busy = FALSE;
    }
}
```

Difference between Monitors and Semaphores

- Monitors enforce mutual exclusion
- *Semaphore wait* vs *Monitor wait*
  - Semaphore wait blocks if value is 0
  - Monitor wait always blocks
- *Semaphore signal* vs *Monitor signal*
  - Semaphore signal either wakes up a thread or increments value
  - Monitor signal only has effect if a thread waiting
- Semaphores have “memory”

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Java-style monitors

- Integrated into the class mechanism
  - Annotation “synchronized” can be applied to a member function
    - This function executes with implicit mutual exclusion
  - Wait, Signal, and Broadcast are called monitor wait, notify, and notifyAll, respectively
- http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - <T_i starts> written to log when transaction T_i starts
  - <T_i commits> written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - Undo(T_i) restores value of all data updated by T_i
  - Redo(T_i) sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be idempotent
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <T_i starts> without <T_i commits>, undo(T_i)
  - If log contains <T_i starts> and <T_i commits>, redo(T_i)

Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0, T_1 atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

Schedule 1: T_0 then T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

- Nonserial schedule allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
  - Conflict if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don't conflict
  - Then S' with swapped order O_j O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
  - S is conflict serializable

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
  - Exclusive – Ti has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
  - $TS(T_i) < TS(T_j)$ if Ti entered system before T_j
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
- Suppose Ti executes read(Q)
  - If $TS(T_i) < W\text{-timestamp}(Q)$, Ti needs to read value of Q that was already overwritten
    - read operation rejected and T_i rolled back
  - If $TS(T_i) \geq W\text{-timestamp}(Q)$
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

Timestamp-ordering Protocol

- Suppose Ti executes write(Q)
  - If $TS(T_i) < R\text{-timestamp}(Q)$, value Q produced by T_i was needed previously and T_i assumed it would never be produced
    - Write operation rejected, T_i rolled back
  - If $TS(T_i) < W\text{-tiimestamp}(Q)$, T_i attempting to write obsolete value of Q
    - Write operation rejected and T_i rolled back
  - Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

End of Chapter 6