Lecture 9

Concurrency and Process Synchronization
Critical Sections

- Last time we saw the idea of a critical section.

- Goal of schemes to protect critical sections:
  - Enforce mutual exclusion
  - Bound waiting
  - Progress
Peterson’s solution

- Simple software-based scheme for critical section problem.

- Two processes share two data elements:

```java
int turn;
boolean flag[2];
```
Peterson’s solution

do {
  flag[i] = TRUE;
  turn = j
  while (flag[j] && turn == j);

  // critical section
  flag[i] = FALSE;

  // remainder
}

\text{\small $i = \text{self}$ ; $j = \text{other}$}
Peterson’s solution

```c
do {
    flag[i] = TRUE;
    turn = j
    while (flag[j] && turn == j);

    // critical section
    flag[i] = FALSE;

    // remainder
}
```

1. Does it enforce mutual exclusion?
Peterson’s solution

```c
do {
    flag[i] = TRUE;
    turn = j
    while (flag[j] && turn == j);
    // critical section
    flag[i] = FALSE;
    // remainder
}
```

2. Progress satisfied? Which processes influence while()?
Peterson’s solution

do {
    flag[i] = TRUE;
    turn = j
    while (flag[j] && turn == j);
    // critical section
    flag[i] = FALSE;
    // remainder
}

3. Bounded waiting?
Synchronization hardware

- Things become easier when hardware support exists to assist with synchronization.

- Uniprocessor solution: disable interrupt handling during critical sections.
  - Guaranteed to allow sequence to run without interruption.

- Does not scale to multiprocessor systems.
  - Requires overhead to negotiate interrupts being turned off on all processors.
Atomicity

- What we need is hardware support for specialize atomic operations.

- Atomicity: an **atomic** operation is one that cannot be divided into suboperations.
  - Or more accurately, the system will guarantee that they execute *as though* they are indivisible.

- Hardware support for atomic operations makes synchronization easier.
Test-and-set

boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
Test-and-set

- Captures state of boolean pointed to by argument.
- Sets value of pointer to TRUE.
- Returns value that pointer pointed at upon entry.

- All of this is performed atomically.
Mutex w/ Test-And-Set

do {
    while (TestAndSet(&lock)) ;
    // critical section
    lock = FALSE;
    // remainder
} while (TRUE);

 Doesn’t satisfy all of requirements though.
 No bounded wait.
Hardware support

- Modern hardware provides instructions to perform test-and-set operations.
  - Even for multiprocessors. Doing so on a parallel computer is an architectural challenge though.

- Often atomic swap operations are also provided.

```c
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
**BTS** - Bit Test and Set (386+)

**Usage:** BTS dest, src

**Modifies Flags:** CF

The destination bit indexed by the source value is copied into the Carry Flag and then set in the destination.

<table>
<thead>
<tr>
<th>operands</th>
<th>286</th>
<th>386</th>
<th>486</th>
<th>Size Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg16,immed8</td>
<td>6</td>
<td>6</td>
<td></td>
<td>4-8</td>
</tr>
<tr>
<td>mem16,immed8</td>
<td>8</td>
<td>8</td>
<td></td>
<td>4-8</td>
</tr>
<tr>
<td>reg16,reg16</td>
<td>6</td>
<td>6</td>
<td></td>
<td>3-7</td>
</tr>
<tr>
<td>mem16,reg16</td>
<td>13</td>
<td>13</td>
<td></td>
<td>3-7</td>
</tr>
</tbody>
</table>
Atomic instructions

- Test and set is somewhat primitive. Other atomic operations exist in hardware that are more flexible.

- Read-modify-write
- Fetch-and-add
- Compare-and-swap
  - Check if a location in memory holds the same value as the argument. If so, modify memory.
  - Returns either flag (memory modified or not) or value read from memory before new value written in.
  - CAS is more powerful than T&S or F&A
    - I can provide a reference to a paper on this to curious people
TestAndSet w/ bounded wait

- Shared data:

```java
boolean waiting[n];
boolean lock;
```
Mutex w/ TestAndSet & bounded wait

do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = false;
    // critical section
    j = (i+1)%n;
    while ((j != 1) && !waiting[j])
        j = (j+1) % n;
    if (j==i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // remainder
} while (TRUE);
A more flexible method of synchronization invented by Dijkstra.

A semaphore is an integer with two operations defined on it:
- P(s) : Test and decrement
- V(s) : Increment

You can use semaphores to build locks and more interesting synchronization structures.
P()
V()

signal(S) {
    S++;
}


Semaphores

- Binary semaphore
  - 0 or 1.
  - Just a plain old lock.

- Counting semaphore
  - A range of integers
  - More flexible than a lock.
  - *Example*: Allow up to 5 threads into a region of code at any point in time.
Semaphores: for more than just locking

- Semaphores are for synchronization.
- This means more than locking.
- Synchronization is the act of communicating some amount of information and acting on it to make a coordinated decision.
Example

Assume sem starts initialized to zero.

```c
some_code();
signal(sem);
wait(sem);
some_other_code();
```

Thread 1

Thread 2
Example

Guarantees `some_code()` always runs before `some_other_code()`.

Thread 1

```c
some_code();
signal(sem);
```

Thread 2

```c
wait(sem);
some_other_code();
```
What communication occurred?

- The semaphore allows a thread to set a flag for the other to see.

- Semaphores are somewhat like leaving a note on a shared board for another process to look at and act on.
Busy waiting and spin locks

- Consider wait() again.

```c
wait (S) {
    while (S <= 0) ;
    S--;}
```

- This inner loop spins until the condition succeeds. This can be wasteful.
Avoiding this via system calls

- We can avoid busy waiting by using system calls to allow a process to voluntarily place itself in the waiting queue.

- It will be placed in the ready queue using a system call when a thread signals on the semaphore.
Semaphore as structure

- This requires semaphores to be implemented as:
  - Integer representing semaphore value.
  - Queue representing set of processes blocked and waiting.

```c
typedef struct {
    int value;
    struct process *list;
} semaphore;
```
P()

wait (semaphore *S) {
    S->value--;  
    if (S->value < 0) {
        enqueue_self(S);  
        block();  
    }
}
V()

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        P = pop_process(S);
        wakeup(P);
    }
}
Negative semaphore

- Semaphore can go negative here.
- Count is number of processes blocked on semaphore.

- Queue of processes (e.g. FIFO) can ensure bounded waiting.
- Head + tail pointers can also allow efficient enqueue/dequeue operations.
Wait/Signal atomicity

- These must be implemented atomically.

- This requires special care on SMPs.
  - Either disabling interrupts.
  - Can use spinlocks instead.
  - Not too bad of a solution since wait() and signal() are fast, so spin won’t last long to ensure wait()/signal() atomicity.
Deadlock and starvation

- Deadlock occurs when a dependency exists between a set of processes where each is blocked waiting on another, and the code to unblock one resides behind the blocking call.
Starvation

- Starvation occurs when a process blocks and ceases to make progress.

Example:

- Semaphore w/ queue that is maintained in a LIFO fashion.
- If a constant stream of other processes exist to contend for the semaphore, the first process in may never continue past the semaphore.
Livelock

- Deadlock occurs when progress ceases.

- Livelock manifests similarly, but can be harder to detect.
  - Program reaches a point that it never makes it past.
  - BUT, the program counter keeps going and the process never enters the waiting state.
  - The lack of “freezing” can make it harder to pin down.
Priority inversion

- Situation where high priority processes need access to kernel resources being accessed by lower priorities processes

- Lock is held by lower priority processes, so higher priority process is blocked by lower one.

- Results in unintended scheduling situations.
Example

- $L < M < H$: Three processes

- L access a resource, then H attempts to.
  - H blocks due to lock held by L

- M comes along and preempts process L
  - Lower priority process M has now impacted how long H must wait for L to give up lock
Priority inversion

- Only occurs in systems with more than two priority levels
  - In other words, common systems

- Priority inheritance protocols can fix it
  - Lower priority processes inherit priority level of processes attempting to access resources they have locked
  - When low priority process is done with lock, it reverts to its original priority level
Issue

- Priority inversion can have an adverse impact on timings

- High priority processes sometimes must execute in a timely fashion
  - Priority inversion causes timing to be unpredictably longer than desired

- Mars Pathfinder example in book is interesting example
Examples: Dining philosophers

- Philosophers around a table eating dinner
- A single chopstick is placed between each plate
- Philosophers require two chopsticks to eat
- When they want to eat, they pick up the chopsticks one at a time

- Problem: Deadlock
Solutions to deadlock

- Seat one less philosopher than we have chopsticks
- Pick up chopsticks in a critical section when both are available
- Break symmetry
  - Odd philosophers start with left; even with right
  - Or just have one philosopher act differently

- Breaking the deadlock problem is one goal
  - We also want to prevent starvation
Examples: Bounded buffer

- Locks used to allow producer and consumer to know when the buffer is empty or full and act accordingly.
  - Producer blocks if buffer is full
    - Signals blocked consumers when value produced
  - Consumer blocks if buffer is empty
    - Signals blocked producers when value consumed
Monitors

- Invented by Per Brinch Hansen ~40 years ago

- Combine mutual exclusion concepts (e.g. semaphores) with encapsulation (objects)
  - Original monitor concept was inspired by the grandparent of all OO languages – Simula.

- Brinch Hansen used them for OS structuring and in the language “Concurrent Pascal”
  - Present today in languages like C# and Java
Monitors

Basic concept:
- Design an object that encapsulates:
  - Data to be protected via mutual exclusion accesses
  - Methods that provide mutual exclusion
  - Locks necessary to implement mutual exclusion
Monitors

```java
public class mymonitor {
    private int sensitiveData;
    private Lock myLock;

    public int doSomething() {
        acquire(myLock);
        // critical section
        release(myLock);
    }

    public int doSomethingElse() {
        acquire(myLock);
        // critical section
        release(myLock);
    }
}
```

- **Data encapsulation**
- **Locking structure associated with data**
- **Critical sections implemented as methods with access to protected data and locks**
Java synchronization

- Java does monitors!

- All objects in Java have a lock corresponding to them
  - You don’t see this as the programmer usually

- The “synchronized” keyword is used to define blocks representing critical sections.
Java synchronization

public class mymonitor {
    private int sensitiveData;

    public synchronized int doSomething() {
        // critical section
    }

    public synchronized int doSomethingElse() {
        // critical section
    }
}

Data encapsulation

Critical sections implemented as synchronized methods.

Start of method acquires lock associated with object instance (“this”) at start, releases at the end of method.
Java synchronization

```java
public class mymonitor {
    private int sensitiveData;

    public int doSomething() {
        synchronized (this) {
            // critical section
        }
    }

    public int doSomethingElse() {
        synchronized (this) {
            // critical section
        }
    }
}
```

Equivalent code with explicit synchronization on (this)