

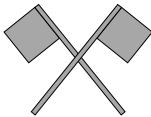
Synchronization Hardware

- **Interrupt lockout**
 - Prevent context switch
 - Uniprocessor only!
- **TestAndSet instruction**
 - Test and modify (atomic operation)
 - Does not bound waiting
 - See text, pages 164-167

How to implement?

Semaphores

- **General synchronization tool**
- **A protected integer variable**
 - Initialized to zero or ?
- **Access by atomic operations**
 - Wait (P): test/decrement
 - Signal (V): increment



Simple Semaphore Class

```

class Sem
{
public:
    Sem(); // Constructor
    void P(); // Test/wait
    void V(); // Signal
    ...
private:
    int semval;
};

```

Simple Semaphore Operations

```
// Constructor
void Sem::Sem()
: semval(0)
{
  // No further init.
}
```

Operations must be atomic (hardware or OS?)

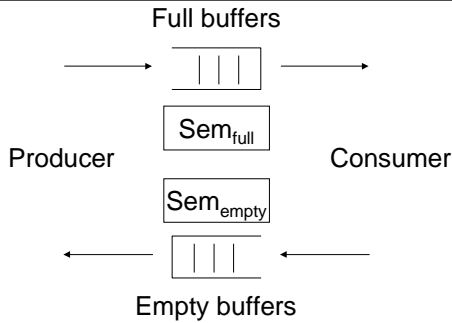
```
// Test/wait
void Sem::P()
{
  while (semval <= 0)
  { /* wait */ }
  --semval;
}
```

```
// Signal
void Sem::V()
{
  ++semval;
}
```

Semaphore Application

- **Producer/consumer buffers**
 - Producer fills, consumer empties
- **Dual semaphores (counting)**
 - Full buffers (0), empty buffers (N)
 - Producer waits on empty, signals full
 - Consumer waits on full, signals empty

Semaphore Application



Mutex

- **Binary semaphore** (initially one)
 - Only one process in critical section
- **Entry section = P(mutex)**
 - Wait for semaphore == 1, set to 0
- **Exit section = V(mutex)**
 - Increment semaphore to 1

Semaphore Blocking



- **Simple implementation**
 - Busy wait: "spinlock"
 - Multi-processor or OS preemption?
- **Blocking implementation**
 - If wait not satisfied
 - Block process
 - Add PID to semaphore queue

What about bounded waiting?

Blocking Trade-Offs

- **Non-blocking (spinlock)**
 - Wastes CPU cycles in busy waiting
 - OK for short locks (critical sections)
- **Blocking (OS semaphores)**
 - Context switch may be costly
 - OK for long sections or uniprocessor

OS Process Queuing

- **If wait cannot be satisfied**
 - OS knows process cannot run
- **Add to semaphore queue**
 - List of processes waiting
 - Linked through PCB's?
- **When signal occurs**
 - Check for waiting processes

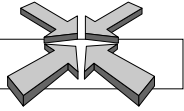
Adaptive Locks

- **Process A requests lock**
- **OS checks lock holder (B)**
- **Adaptive spin/block decision**
 - If B running, A spins (multi-CPU only?)
 - B may release lock soon?
 - If B blocked, A blocks also

Lock Granularity 

- **Especially inside OS kernel**
- **One lock for all OS data**
 - Low lock overhead
 - Greater collision potential
- **Multiple locks** (by data segment)
- **Adaptive locking** (notify holder?)

Monitors



- **High-level synchronization**
 - Build on low-level primitives
 - Consolidate multiple semaphores?
 - Localize and coordinate operations
- **Implementation**
 - Built into some languages
 - Java “synchronized”
 - Available as run-time library

Monitor Components

- **Condition variables**
 - Predicates defined on variables
- **Operations** (critical section)
 - Modify or test condition variables
- **Synchronization**
 - Wait, signal, mutual exclusion

C. A. R. Hoare, Lamson/Redell, etc.

Condition Variables

- **Semaphore** E.g., POSIX pthread_cond*
 - Controls single resource availability
- **More complex condition?**
 - Waiting for combination of factors
 - E.g., need 2 A's, a B, and a C
 - Predicate: (A>1) && (B>0) && (C>0)

Condition Example



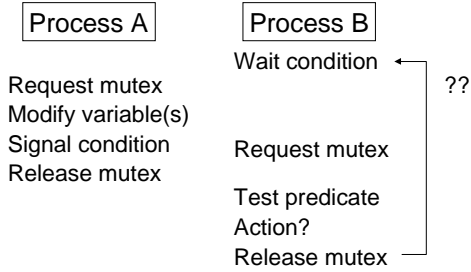
- Taxi passenger assignment
 - Arriving airport passengers
- Group by destination
- Fill cab (e.g., 3 passengers)
- Predicate for dispatch?
 - Passenger, cab availability

Monitor Operations

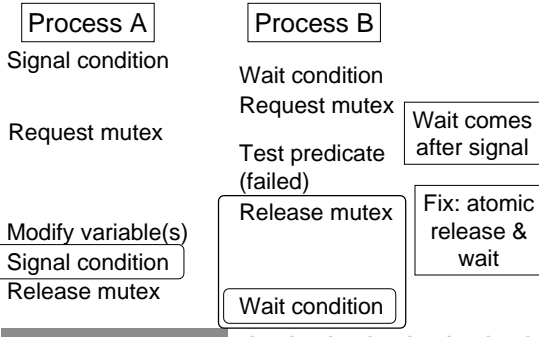
"Mesa semantics" variant; others exist

- Mutex for any data set/test
- Condition signal
 - Raised when any data modified
- Condition wait
 - Blocks until condition signaled
 - Process then evaluates predicate

Monitor Example



Monitor Problem



Monitor Signal Types

- **Notify**
 - Single waiting process notified
 - Presumes that it will succeed?
 - Or signal again to next waiting proc?
- **Broadcast**
 - All waiting processes notified

Dining Philosophers

