# G22.2250-001
# Operating Systems

Lecture 3

Threads, Process Synchronization

September 18, 2007

## Outline

Announcements
- Lab 1 due today!
  - Please send e-mail to TA if you cannot submit by end-of-day today

- (Review) Process management system calls
- Threads
  - Multithreading models
- Process cooperation
  - Shared memory vs. message passing
- Process synchronization
  - Introduction to the critical section problem
    - Petersen's 2-process solution
  - Locks, Semaphores, Condition variables

*[ Silberschatz/Galvin/Gagne: Chapters 3.4 – 3.5, 4, 6.1-6.4]*

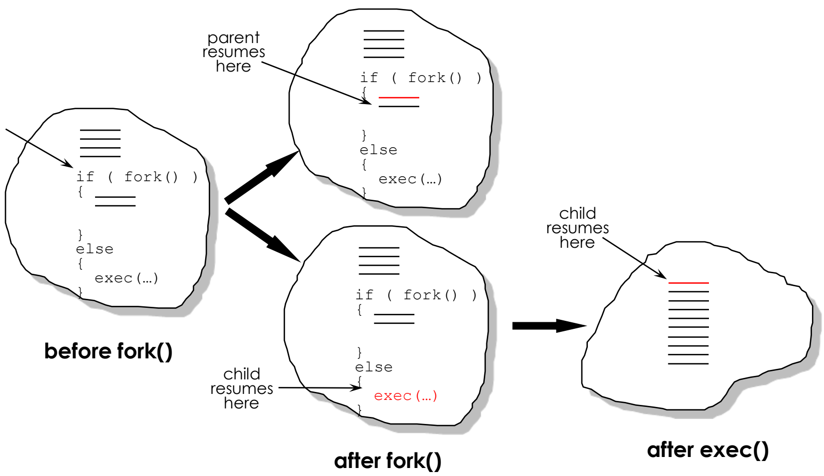
## (Review)
## System Calls for Process Management

- Creation
  - a "parent" process spawns a "child" process; a *fork* in UNIX
    - child may or may not inherit parent's memory
    - child is added to the ready queue
  - the parent-child association is maintained via process IDs (PIDs)
- Termination
  - normal: a process asks the OS to delete it; an *exit* in UNIX
    - all resources of a terminated process are deallocated and reclaimed
    - on termination, the child's PID and output may be passed back to the parent
  - abnormal: another process (typically the parent) can cause termination
    - if the child exceeds its usage, becomes obsolete, or the parent is exiting the system due to some other problem
    - a process (almost always) terminates when its parent does
- Communication
- Coordination

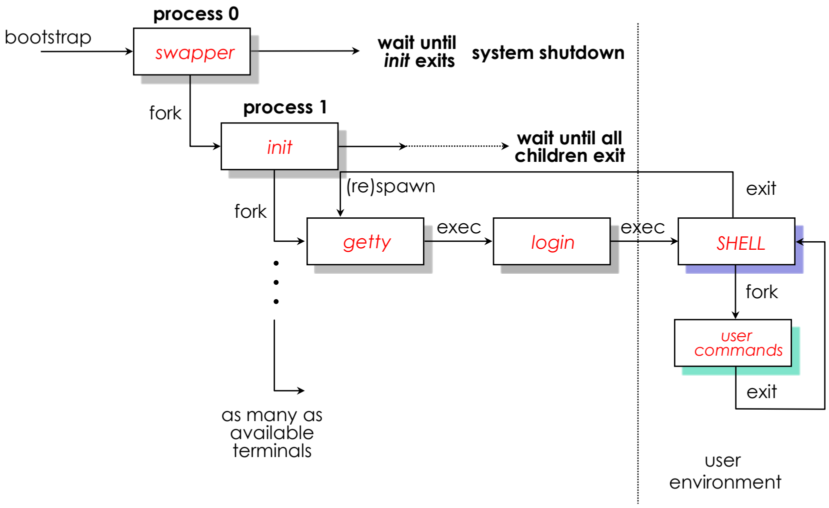


## (Review)
## Example: Process Creation in UNIX

Two system calls: fork, exec



# UNIX System Initialization

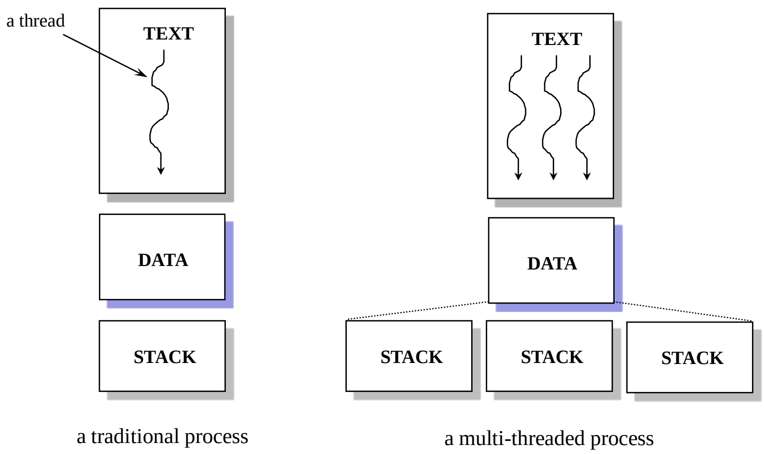





# Threads

- A thread is similar to a process
  - sometimes called a *lightweight process*
  - several threads (of control) can execute within the same address space
- Like a process, a thread
  - is a basic unit of CPU utilization
  - represents the state of a program
  - can be in one of several states: *ready*, *blocked*, *running*, or *terminated*
  - has its own program counter, registers, and stack
  - executes sequentially, can create other threads, block for a system call
- Unlike a process, a thread
  - shares with peer threads, its code section, data section, and operating-system resources such as open files and signals
  - is *simpler and faster*

## Threads versus Processes (contd.)






## Threads: Why Simpler?

Threads share the process address space

- Benefits for the user:
  - communication is easier
  - communication is more efficient
  - security may not be necessary
    assumed to operate within the same protection domain
  - one blocking thread need not block other threads in the process
- Benefits for the OS
  - context switching is more efficient
    - memory mappings can remain unchanged
    - cache need not be flushed
  - can run a process across multiple nodes of a multiprocessor
    - performance advantages if threads can execute in parallel (e.g., web servers)

## Types of Threads

- User-level threads (e.g., pthreads: Section 4.3.1, Java threads: Section 4.3.3)
  - – OS does not know about them
  - – implemented/scheduled by library routines
  - ⬆ operations are faster (context switch, communication, control)
  - ⬇ blocking operations block the entire process (even with ready threads)
  - ⬇ operations based on local criteria may be less effective (e.g., scheduling)

- Kernel-level threads (e.g., Solaris 2, WinXP: Section 4.5.1, Linux: 4.5.2)
  - – known to the OS
  - – scheduled by the OS
  - ⬆ process need not block if one of its threads blocks on a system call
  - ⬇ thread operations are expensive
    - switching threads involves kernel interaction (via an interrupt)
  - ⬆ the kernel can do a better job of allocating resources

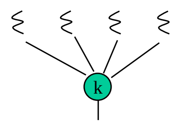


## Multithreading Models

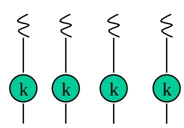
- Most systems provide support for both user and kernel threads

Three dominant models for mapping threads to kernel resources

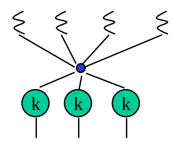
- Many-to-one
  - – Thread management done in user space
  - – Entire process blocks if a thread does a blocking operation
  - – E.g., systems without kernel threads




- One-to-one
  - – Each user-thread mapped to a kernel thread
  - – Allows more concurrency
  - – E.g., Windows 2000, XP (fibers: many-to-one)




- Many-to-many
  - – Combination of the above two
  - – E.g., Solaris 2

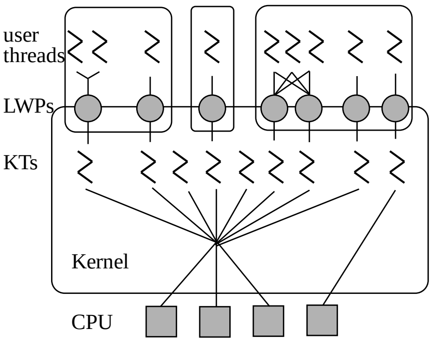


# POSIX Threads (pthreads)

- A portable API for multithreaded programs
  - Some pthreads implementations do map threads to kernel threads
  - Most rely on user-level threading support
    - Assembly instructions to save/restore registers
- Calls for creating, exiting, joining pthreads
  - pthread_create: start execution of this thread
    - Takes function pointer as an argument
  - pthread_exit:terminate execution of this thread
  - pthread_join: wait for a particular thread to exit
- Other calls
  - Help set thread attributes (stack size, scheduling behavior, etc.)
  - Specify signal handling
    - Signals are a way of allowing processes to respond to events
      - Interrupts (Ctrl-C), others
    - Multithreaded systems need to define a way for signals to be communicated to individual threads (see Section 4.4.3)
      - All threads , a specific thread, only those threads that do not block the signal, …



# Processes and Threads in Solaris 2




- OS schedules execution of kernel threads (KTs)
  - runs them on the CPUs
  - a KT can be pinned to a CPU
- A task consists of one or more lightweight processes (LWPs)
  - LWPs in a task may
    - contain several *user-level threads*
    - issue a system call
    - block
- A LWP is associated with a KT
- There are KTs with no LWP

- Linux has a simpler model (see Section 4.5.2): only kernel threads (tasks)
  - System calls for process creation (fork) and thread creation (clone)

# Outline

Announcements

- Lab 1 due today!
  - Please send e-mail to TA if you cannot submit by end-of-day today
- (Review) Process management system calls
- Threads
  - Multithreading models
- Process cooperation
  - Shared memory vs. message passing
- Process synchronization
  - Introduction to the critical section problem
    - Petersen's 2-process solution
  - Locks, Semaphores, Condition variables

*[ Silberschatz/Galvin/Gagne: Chapters 3.4 – 3.5, 4, 6.1-6.4]*



# Process Cooperation

- Why do processes cooperate?
  - *modularity*: breaking up a system into several sub-systems
    - e.g.: an interrupt handler and device driver that need to communicate
  - *convenience*: users might want to have several processes share data
  - *speedup*: a single program is run as several sub-programs
- How do processes cooperate?
  - communication abstraction: *producers* and *consumers*
    - *producers* produce a piece of information
    - *consumers* use this information
  - abstraction helps deal with general "phenomena" and simplifies correctness arguments
- Two general classes of process cooperation techniques
  - shared memory
  - message passing

## Shared Memory (Procedure-oriented System)

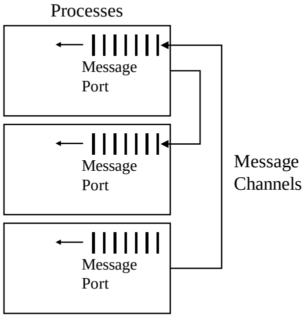
- Processes can directly access data written by other processes
  - examples: POSIX threads, Java, Mesa, small multiprocessors
- A finite-capacity shared buffer

```
N: integer                              -- buffer size
nextin = nextout = 1 initially;         -- start of buffer
buffer: array of size N
Producer:
   Repeat
     -- produce an item in tempin
     while (nextin+1) mod n = nextout do wait-a-bit;
     buffer[nextin] := tempin;
     nextin := (nextin+1) mod n;
Consumer:
   Repeat
     while nextin = nextout do wait-a-bit;
     tempout := buffer[nextout];
     nextout := (nextout+1) mod n;
     -- consume the item in tempout
```



## Message Passing (Message-oriented System)

- Execution is in separate address spaces
  - communication using message channels
  - examples: UNIX processes, large multiprocessors, etc.
- Components
  - messages and message identifiers
  - message channels and ports
    - channels (pipes) must be bound to ports
    - queues associated with ports
  - message transmission operations
    - SendMessage[channel, body] returns id
    - AwaitReply[id]
    - RecvMessage[port] returns id
    - SendReply[id, body]




- Many variants: See Section 3.5

➡ Focus on shared memory for next few lectures

## Outline

Announcements

- Lab 1 due today!
  - Please send e-mail to TA if you cannot submit by end-of-day today

- (Review) Process management system calls
- Threads
  - Multithreading models
- Process cooperation
  - Shared memory vs. message passing
- Process synchronization
  - Introduction to the critical section problem
    - Petersen's 2-process solution
  - Locks, Semaphores, Condition variables

*[ Silberschatz/Galvin/Gagne: Chapters 3.4 – 3.5, 4, 6.1-6.4]*



## Bounded Buffers Using Counters

```
N: integer                              -- buffer size
counter: integer = 0 initially;
nextin = nextout = 1 initially;        -- start of buffer
buffer: array of size N

Producer:
  Repeat
    -- produce an item in tempin
    while counter = N do wait-a-bit;
    buffer[nextin] := tempin;
    nextin := (nextin+1) mod n;
    counter := counter+1;
Consumer:
  Repeat
    while counter = 0 do wait-a-bit;
    tempout := buffer[nextout];
    nextout := (nextout+1) mod n;
    counter := counter-1;
    -- consume the item in tempout
```

Producer and Consumer processes are asynchronous! execution of these two statements can be interleaved (e.g., because of interrupts)

## Interleaving of Increment/Decrement

- Each of increment and decrement are actually implemented as a series of machine instructions on the underlying processor

| Producer | Consumer |
|---|---|
| register1 := counter | register2 := counter |
| register1 := register1 + 1 | register2 := register2 - 1 |
| counter := register1 | counter := register2 |

- An interleaving
  - counter = 5; a producer followed by a consumer

| Producer | Consumer | |
|---|---|---|
| register1 := counter | | {register1 = 5} |
| register1 := register1 + 1 | | {register1 = 6} |
| | register2 := counter | {register2 = 5} |
| | register2 := register2 - 1 | {register2 = 4} |
| counter := register1 | | {counter = 6} |
| | counter := register2 | {counter = 4} |



## The Problem

- Increment and decrement are not *atomic* or *uninterruptable*
  - two or more operations are executed atomically if the result of their execution is equivalent to that of some serial order of execution
  - operations which are always executed atomically are called atomic
    - byte read; byte write;
    - word read; word write
- The code containing these operations creates a race condition
  - produces inconsistencies in shared data
- Reasons for non-atomic execution

  (uniprocessors)
  - interrupts
  - context-switches

  (multiprocessors)
  - Execution on different processors

## The Solution

- The producer and consumer processes need to synchronize
  - so that they do *not* access shared variables at the same time
  - this is called mutual exclusion
    - the *shared* and *critical* variables can be accessed by only one process at a time
  - access must be serialized even if the processes attempt concurrent access
    - in the previous example: counter increment and decrement operations
- General framework for achieving this: Critical Sections
  - work independent of the particular context or need for synchronization



## Critical Sections

- Critical sections: General framework for process synchronization

```
ENTRY-SECTION
CRITICAL-SECTION-CODE
EXIT-SECTION
```

  - the **ENTRY-SECTION** controls access to make sure that no more than one process $P_i$ gets to access the critical section at any given time
    - acts as a *guard*
  - the **EXIT-SECTION** does bookkeeping to make sure that other processes that are waiting know that $P_i$ has exited
- How can we implement critical sections?
  - turn off interrupts around critical operations
  - ✓ build on top of atomic memory load/store operations
  - ✓ provide higher-level primitives

## Two-Process Solutions: Turn Counters

- Shared integer variable: **`turn`** (initialized to 0)
  - for $i \in \{0, 1\}$: $P_i$ executes:

```
while (turn != i) wait-a-bit;
CRITICAL SECTION;
turn := j;
```

  - the while loop is the *entry* section
    - process $P_i$ waits till its turn occurs
  - the single instruction `turn := j` constitutes the *exit* section
    - informs the other process of its turn

- Mutual exclusion?
  - assume atomic loads and stores
- Drawbacks?
  - if $P_1$ never wants to execute the critical section, $P_0$ cannot reenter;
    - access *must* alternate

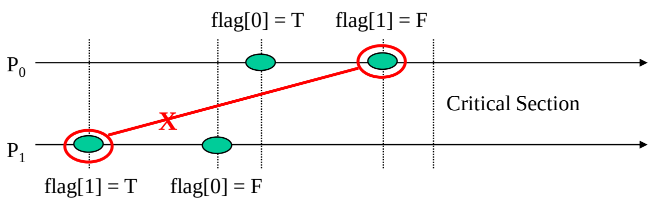


## Two-Process Solutions: Array of Flags

- Boolean array **`flag`** (initialized to false), $P_i$ executes:

```
1:   flag[i] := true;
2:   while flag[j] wait-a-bit;
     CRITICAL SECTION
3:   flag[i] := false;
```

- Mutual exclusion?




- Is this good enough?
- No: $P_0$ *and* $P_1$ can be looping on instruction 2 forever

## Criteria for Correctness

Three conditions

- Mutual exclusion
- Progress
  - at least one process requesting entry to a critical section will be able to enter it if there is no other process in it
- Bounded waiting
  - no process waits indefinitely to enter the critical section once it has requested entry



## Two-Process Solutions: Petersen's Algorithm

- Combines the previous two ideas

```
1:  flag[i] := true
2:  turn := j
3:  while (flag[j] and (turn == j)) wait-a-bit
    CRITICAL SECTION
4:  flag[i] := false
```

- Does the algorithm satisfy the three criteria?

## Petersen's Algorithm: Mutual Exclusion

```
1:  flag[i] := true
2:  turn := j
3:  while (flag[j] and (turn == j)) wait-a-bit
    CRITICAL SECTION
4:  flag[i] := false
```

- Suppose: $P_0$ is in its critical section, and $P_1$ is wanting to enter
- This can happen only if either
  - (*case* 1) $P_0$ found `flag[1]` false, or
  - (*case* 2) $P_0$ found `turn == 0`
  - in the first case: $P_1$ will set `turn` after $P_0$ did, and find `turn == 0`
  - in the second case: $P_1$ has already set `turn = 0`
  - in both cases: $P_1$ will wait till `flag[0] == false`



## Petersen's Algorithm: Progress and Bounded Waiting

```
1:  flag[i] := true
2:  turn := j
3:  while (flag[j] and (turn == j)) wait-a-bit
    CRITICAL SECTION
4:  flag[i] := false
```

- To prove *progress:*
  - if $P_1$ is not ready to enter the critical section
    - `flag[1]` will be false ➡ $P_0$ can enter
- To prove *bounded waiting*:
  - let $P_0$ be in the critical section and $P_1$ be waiting on instruction 3 above
  - if $P_0$ exits and goes elsewhere,
    - either $P_1$ will find `flag[0]` to be false
    - if not, $P_0$ will attempt to reenter the critical section, setting `turn := 1`
    - in either case, $P_1$ will find the condition for waiting in (3) to be false and will enter the critical section

## Can These Solutions be Extended to >2 Processes?

- N-process solutions
  - do exist: Bakery Algorithm (see handout from OSC, 6th Edition)
  - but reasoning gets even more complicated!
- So, we can implement critical sections using only support for atomic memory loads and stores …
- … But, there must be an easier way!
- Higher-level synchronization primitives
  - locks (mutexes), semaphores, condition variables
  - rely on more support from hardware
    - disabling of interrupts: only around the primitives
    - atomic read-modify-write operations



## Synchronization Primitives (1): Locks (Mutexes)

- Locks
  - a single boolean variable **L**
    - in one of two states: **AVAILABLE, BUSY**
  - accessed via two *atomic* operations
    - **LOCK** (also known as **Acquire**)

```
while ( L != AVAILABLE ) wait-a-bit
L = BUSY;
```

    - **UNLOCK** (also known as **Release**)

```
L = AVAILABLE;
wake up a waiting process (if any)
```

  - process(es) waiting on a LOCK cannot "lock-out" process doing UNLOCK
- Critical sections using locks

```
LOCK( L )
CRITICAL SECTION
UNLOCK( L )
```

  - Mutual exclusion? Progress? Bounded waiting?

## Synchronization Primitives (2): Semaphores

- Semaphores
  - a single integer variable S
  - accessed via two *atomic* operations
    - **WAIT** (sometimes denoted by **P**)
      ```
      while S <= 0 do wait-a-bit;
      S := S-1;
      ```
    - **SIGNAL** (sometimes denoted by **V**)
      ```
      S := S+1;
      wake up a waiting process (if any)
      ```
  - WAITing process(es) cannot "lock out" a SIGNALing process

- Binary semaphores
  - S is restricted to take on only the values 0 and 1
  - WAIT and SIGNAL become similar to LOCK and UNLOCK
  - are *universal* in that counting semaphores can be built out of them



## Uses of Semaphores

- Mutual exclusion (initially S = 1)
  ```
  P( S )
  CRITICAL SECTION
  V( S )
  ```

- Sequencing (initially S = 0)

| $P_1$ | $P_2$ |
|---|---|
| `Statement 1` | |
| `V( S )` | `P( S )` |
| | `Statement 2` |

- Detailed examples of its use in Lecture 4

## Universality of Binary Semaphores

- Implement operations on a (counting) semaphore `CountSem`
  - use binary semaphores S1 = 1, S2 = 0
  - integer C = initial value of counting semaphore

```
P(CountSem)                          V(CountSem)

P(S1);                               P(S1);
C := C-1;                            C := C+1;
if ( C < 0 ) then                     if ( C <= 0 ) then V(S2);
   begin V(S1); P(S2); end                         else V(S1);
V(S1);
```

  - S1 ensures mutual exclusion for accessing C
  - S2 is used to block processes when C < 0
  - is a race condition possible after V(S1) but before P(S2)?



## Synchronization Primitives (3): Condition Variables

- Condition variables
  - an *implicit* process queue
  - three operations that *must be performed within a critical section*
    - **WAIT**
      `associate self with the implicit queue`
      `suspend self`
    - **SIGNAL**
      `wake up exactly one suspended process on queue`
      - has no effect if there are no suspended processes
    - **BROADCAST**
      `wake up all suspended processes on queue`

- Two types based on what happens to the process doing the SIGNAL
  - Mesa style (Nachos uses Mesa-style condition variables)
    - **SIGNAL**-ing process continues in the critical section
    - resumed process must re-enter (so, is not guaranteed to be the next one)
  - Hoare style
    - **SIGNAL**-ing process immediately exits the critical section
    - resumed process now occupies the critical section

## Uses of Condition Variables

- Can be used for constructing
  - critical sections, sequencing, …
- Primary use is for waiting on an event to happen
  - after checking that it has not already happened
    - WHY IS THIS IMPORTANT?
- Example: Three processes that need to cycle among themselves

  <print 0>; <print 1>; <print 2>; <print 0>; <print 1>; …
  - One variable: **turn**; three condition variables: $\mathbf{cv_0}$, $\mathbf{cv_1}$, $\mathbf{cv_2}$
  - Process $P_i$ executes (in a critical section)

```
while ( turn != i) WAIT(cv_i)
<do the operation>
turn := (turn + 1) mod 3; SIGNAL(cv_turn)
```



## Higher-level Synchronization Primitives

- Several additional primitives are possible
  - Built using locks, semaphores, and condition variables
- An example: Event Barriers (see Nachos Lab 3)

## Implementing the Synchronization Primitives

- Need support for atomic operations from the underlying hardware
  - applicable only to a small number of instructions
    - else, can implement critical sections this way

Three choices

- Use n-process mutual-exclusion solutions
  - complicated
- ✓ Selectively disable interrupts on uniprocessors
  - so, no unanticipated context switches ➡ atomic execution
  - solution adopted in Nachos (see Lab 2 for details)
- ✓ Rely on special hardware synchronization instructions

- Can implement one primitive in terms of another
  - Nachos Lab 2



## Implementation Choices (1): Interrupt Disabling

- Semaphores

P(S)

```
DISABLE-INTERRUPTS
while S <= 0 do wait-a-bit
            [ENABLE-INTERRUPTS; YIELD CPU; DISABLE-INTERRUPTS]
S := S-1;
ENABLE-INTERRUPTS
```

V(S)

```
DISABLE-INTERRUPTS
S := S+1;
ENABLE-INTERRUPTS
```

- Drawback
  - a process spins on this loop (busy waiting) till it can enter critical section
  - can waste *substantial* amount of CPU cycles idling
    - Even if `wait-a-bit` is implemented as
      - give up CPU (i.e. put at the end of ready queue)
    - since there are still context switches
  - not a very useful utilization of valuable cycles

## Efficient Semaphores

- Implement P and V differently
  - maintain an explicit *wait queue* organized as a scheduler structure

```
type semaphore = record
        value: integer;
        L: list of processes;
        end;

P(S):   S.value := S.value - 1;        V(S):   S.value := S.value + 1;
        if ( S.value < 0 )                     if ( S.value <= 0 )
           then begin                             then begin
                add process to S.L                     remove P from S.L
                block;                                 wakeup(P);
           end;                                   end;
```

  - still need atomicity: can use previously discussed solutions
    - can have spinning but only for a small period of time (~10 instructions)
  - queue enqueue/dequeue must be fair
    - not required by semantics of semaphores



## Implementation Choices (2): Hardware Support

- Rationale: Hardware instructions enable simpler/efficient solutions to common synchronization problems
  - disabling interrupts is a brute-force approach
  - does not work on multiprocessors
    - simultaneous disabling of all interrupts is not feasible

- Two common primitives
  - test-and-set
  - swap

## Semantics of Hardware Primitives

- Test-and-set
  - given boolean variables X, Y, atomically set X := Y;  Y := true

```
boolean Test-and-set( boolean &target ) {
    boolean rv = target;
    target = true;
    return rv;
}
```

- Swap
  - atomically exchange the values of given variables X and Y

```
temp = X; X = Y; Y = temp;
```

  - can emulate test-and-set

```
boolean Test-and-set( boolean &target ) {
  boolean t := true;
  swap (target, t);
  return t;
}
```



## Implementing Locks Using Test-and-Set

```
LOCK:      L : boolean := false
           while Test-and-set(lock)  wait-a-bit
UNLOCK     lock := false
```

- Properties of this implementation
  - Mutual exclusion?
    - first process $P_i$ entering critical section sets lock := true
      - test-and-set (from other processes) evaluates to true after this
    - when $P_i$ exits, lock is set to false, so the next process $P_j$ to execute the instruction will find test-and-set = false and will enter the critical section
  - Progress?
    - trivially true
  - Unbounded waiting
    - possible since depending on the timing of evaluating the test-and-set primitive, other processes can enter the critical section first
    - See Section 6.4 for a solution to this problem

## Synchronization Primitives in Real OSes

- Unix: Single CPU OS
  - implement critical sections using interrupt elevation
    - disallow interrupts that can modify the same data
    - (Linux 2.4 and earlier, Section 6.8.3) disable kernel preemption
  - another possibility: interrupts never "force" a context switch
    - they just set flags, or wake up processes
  - primitives
    - **`sleep`** (address);
    - **`wake_up`** (address); -- wakes up all processes sleeping on address
  - typical code

```
ENTRY: while (locked) sleep(bufaddr);
       locked = true;
EXIT:  locked = false; wake_up (bufaddr);
```



## Synchronization Primitives in Real OSes (contd.)

- Solaris 2: multi-CPU OS
  - for brief accesses only
    - adaptive mutexes
    - starts off as a standard spinlock semaphore
      - if lock is held by running thread, continues to spin
        - » valid only on a multi-CPU system
      - otherwise blocks
  - for long-held locks
    - (process queue) semaphores
    - condition variables
      - wait and signal
    - reader-writer locks
      - for frequent mostly read-only accesses
  - turnstiles
    - the queue structure on which threads block when waiting for a lock
    - associated with threads rather than lock objects
      - Each thread can block on at most one object, so more efficient