Overview of Presentation
Spin-locks, Interrupts and Exclusion
- reasonable use of interrupt routines and disabling
- spin locks revisited, and sleep/wakeup exclusion
Semaphores
- introduction, operations
- implementing semaphore operations
- using semaphores as locks and events
Topics deferred to next week
- deadlocks

key points from previous lecture
synchronization
- the mutual exclusion problem
- the asynchronous notification problem
problems and solutions
- limitations of interrupt disabling and spin locks
- atomic instructions for safe updates of shared data
the sleep/wakeup race condition

Disables and Spin-locks revisited
examples in the text used them inappropriately
- atomic instructions and higher level locking operations
  are the answer for most OS, and all user mode needs
- locks are software serialization mechanisms, and there
  are hardware-related situations where they won't work
interrupt disabling is often necessary
- in exclusion between drivers and interrupt handlers
spin-locks occasionally do make sense
- in very specific Symmetric Multi-Processor situations

Interrupts - when to disable them
To avoid potentially disastrous preemption
- only for use within the operating system
- neither necessary nor possible in user mode
critical sections in hardware manipulation
- giving commands or getting status to/from device
- operations in interrupt handler would conflict
critical sections in communications queues
- operation that could cause interrupt routine to block
  protect as few instructions as possible
- disabled time is expensive
**being careful with interrupt routines**

Interrupt service time is very costly
- some or all interrupts have been disabled
- scheduled processes have been preempted
- the devices are idle, awaiting new instructions
- the system is less responsive

Interrupt routines cannot block or yield the CPU
- they are not a scheduled thread that can block/run
- they run on a "hijacked process stack"
- cannot do resource allocations that might block
- cannot do synchronization operations that might block

---

**locks and interrupts**

**a dangerous combination**

Synchronous Code

```
while(TS(lockp));
/* critical section */
```

Interrupt Handler

```
while(TS(lockp));
/* critical section */

lockp = 0;
...
```

Interrupt handler will loop forever
- interrupts will remain disabled forever
- synchronous code will never complete
- so lock will never be released

---

**When should we use spin-locks?**

only necessary for SMP serialization of OS code
- never necessary in user mode, rare even in OS
only if critical section is extremely brief
- e.g. one digit number of instructions
- preemption must be virtually impossible
only if conflict w/interrupt routines is impossible
- interrupts are disabled
- interrupt routines don't use this resource
only if sleeping is not a real option
- e.g. code called by a trap or interrupt handler

---

**Semaphores - signaling devices**

Concept introduced in 1968 by Edsger Dijkstra
- cooperating sequential processes
THE classic synchronization mechanism
- behavior is well specified and universally accepted
- a foundation for most synchronization studies
- a standard reference for all other mechanisms
more powerful than simple locks
- they incorporate a FIFO waiting queue, and so they address both exclusion and asynchronous completion
- they have a counter rather than a binary flag, which makes them more versatile
Semaphores - Operations

Semaphore has a count, and a waiting queue

P (proberen) ... "test"
- if count is positive, decrement it and return
- if count is non-positive, add process to waiting queue
- yield CPU, and try again after we are reawakened

V (verhogen) ... "raise"
- increment count
- if count is now positive and queue non-empty, awaken the first process on the waiting queue

remember the sleep/wakeup race

```c
void sleep( eventp *e ) {
    while(e->posted == FALSE) {
        add_to_queue( &e->queue, myproc );
        myproc->runsate |= BLOCKED;
        yield();
    }
}

void wakeup( eventp *e ) {
    struct proc *p;
    e->posted = TRUE;
    p = get_from_queue(&e->queue);
    if (p) {
        p->runstate &= ~BLOCKED;
        resched();
    }
}
```

locking to solve the sleep/wakeup race

requires a spin-lock to work on SMPS
- sleep/wakeup may be called on two processors
- the critical section is short and cannot block
- we must spin, because we cannot sleep ... the lock we need is the one that protects the sleep operation
also requires interrupt disabling in sleep
- wakeup is often called from interrupt handlers
- interrupt possible during sleep/wakeup critical section
- If spin-lock already is held, wakeup will block for ever
very few operations require both of these

Semaphores - Implementing structure

```c
struct semaphore {
    int sem_count; /* semaphore count */
    lock_t sem_lock; /* lock to protect semaphore operations */
    struct queue sem_queue; /* queue of waiting processes */
};
```

count contains current value of semaphore
- count is always greater than or equal to zero
lock to protect semaphore during P/V critical sections
- e.g. to prevent the sleep/wakeup race
NOTE: the lock is not for the resource, but for the semaphore queue of processes currently waiting for the semaphore
- if count is positive, they should wake up and try again
Semaphores - Implementing P

```c
void p (struct semaphore *s) {
    for(;;) {
        save = intr_enable(DISABLE);
        while( TS( &s->sem_lock ) );
        if (sem_count > 0) {/* count is not allowed to go negative */
            s->sem_count--;
            s->sem_lock = 0; intr_enable(save);
            return;
        }
        add_to_queue( &s->sem_queue, myproc );
        myproc->runstate |= BLOCKED;
        s->sem_lock = 0; intr_enable(save);
        yield();
    }
}
```

Semaphores - Implementing V

```c
void v(struct semaphore *s) {
    struct proc_desc *p = 0;
    save = intr_enable(DISABLE);
    while( TS( &s->sem_lock ) );
    /* ASSERT: at all times s->sem_count >= 0 */
    s->sem_count++;
    /* the only way to see if someone is blocked is to look at the queue */
    if (p = get_from_queue( &s->sem_queue )) {
        if (p)
            reschedule( p );
        s->sem_lock = 0; intr_enable(save);
    }
}
```

Example – P and V

<table>
<thead>
<tr>
<th>disable</th>
<th>lock</th>
<th>count</th>
<th>queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>proc A</th>
<th>proc B</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>V</td>
</tr>
<tr>
<td>P</td>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>proc A</th>
<th>proc B</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>wake</td>
</tr>
</tbody>
</table>

Is this Implementation Correct?

- what if an interrupt comes in ...
  - while process A is executing sleep
  - while process A is executing wakeup
- what if A calls sleep on an SMP ...
  - while process B is executing sleep
  - while process B is executing wakeup
- what if A calls wakeup on an SMP ...
  - while process B is executing sleep
  - while process B is executing wakeup

SMP parallelism is usually worse than preemption
using semaphores for exclusion

initialize semaphore count to one
  - count reflects number of threads allowed to hold lock
use P operation to take the lock
  - the first P will succeed
  - subsequent attempts will block
use V operation to release the lock
  - restore semaphore count to one again
  - if any threads are waiting, unblock the first in line

Semaphores - for exclusion

```c
int can_write_check(int amount) {
    static struct semaphore checkbook = {1, 0, 0}; /* count = 1, available */
    static int current_balance;
    int ret;
    p(&checkbook);
    if (current_balance < amount) {
        ret = -1;
    } else {
        current_balance -= amount;
        ret = amount;
    }
    v(&checkbook);
    return(ret);
}
```

The semaphore vs. the lock

the semaphore is a synchronization object
  - it controls or protects access to checkbook
there are critical sections of code in P and V
  - the sleep/wakeup race must be prevented
  - we protect this with the sem_lock (and disables)
there are two protected objects here
  - checkbook updates, protected by the semaphore
  - semaphore (count/queue) updates, protected by lock
getting lock does not convey access to checkbook
  - it only enables you to do a V, or attempt to do a P

using semaphores for completion

initialize semaphore count to zero
  - count reflects number of available completions
use P operation to await an event
  - block until count becomes positive
  - decrement count to consume one event
use V operation to signal completion of event
  - increment count to reflect available completion
  - awaken next thread if any are currently waiting
Semaphores - completion events

```c
struct semaphore pipe_semaphore = { 0, 0, 0 }; /* count = 0; pipe empty */
char buffer[BUFSIZE]; int read_ptr = 0, write_ptr = 0;

char pipe_read_char() {
    p(&pipe_semaphore);         /* wait for input available */
    c = buffer[read_ptr++];     /* get next input character */
    if (read_ptr >= BUFSIZE) read_ptr -= BUFSIZE; /* circular buffer wrapping */
    return(c);
}

void pipe_write_string(char *buf, int count) {
    while (count-- > 0) {
        buffer[write_ptr++] = *buf++;   /* store next character */
        if (write_ptr >= BUFSIZE) write_ptr -= BUFSIZE; /* circular buffer wrap */
        v(&pipe_semaphore);            /* signal char available */
    }
}
```

Example - circular buffer management

```
Buffer
|   |   |
write_ptr
read_ptr
sem count
Read     Write "abc"  Wake  Read
```

Active/Passive - the preemption thing

Standard semaphore semantics are not complete
- who runs after a V unblocks a P?
- the running V'er or the blocked P'er

There are arguments for each behavior
- gratuitous context switches increase overhead
- producers and consumers should take turns
- if we delay P'er, someone else may get semaphore

Preemptive priority-based scheduler can do this
- reassess scheduling whenever someone wakes up
- P'ers priority controls who will run after wake-up

who to run next - it can be tricky

problem – priority inversion
- given a lock that may be needed by multiple threads
- a low priority thread is preempted while holding lock
- a high priority thread blocks for the lock
- blocked thread is gated by holders priority

solution – priority inheritance
- when a high priority process blocks for a lock
- temporarily transfer its priority to current lock holder
- help high priority thread by helping low priority thread
**Limitations of Semaphores**

- Semaphores are a very spartan mechanism
- They are simple, and have few features
- More designed for proofs than synchronization
- They lack many practical synchronization features
- It is easy to deadlock with semaphores
- One cannot check the lock without blocking
- They do not support reader/writer shared access
- No way to recover from a wedged V'er
- No way to deal with priority inheritance

Nonetheless, most OSs support them.

**OS Synchronization Services**

- Why synchronization is implemented in OS
  - Exclusion sometimes involves interrupt disabling
  - Exclusion often involves control of scheduling
  - Control of resource sharing is a trusted operation
  - OS needs synchronization for its own data structures

- Why OS supported services are simple
  - Simpler mechanisms are often more general
  - Intended as building blocks for higher level services
  - Come in many flavors (read/write, SMP, interrupt,...)

**Application Synchronization Services**

- Associated with sharable resources
  - Usually OS resources like files, mail boxes, processes
  - Critical sections in multi-threaded applications

- Usually implemented with system calls
  - Most serialized objects are implemented in the OS
  - Sleeping and waking up are implemented in the OS
  - User-mode processes are interruptable/preemptable

- Thread synchronization may be entirely user-mode
  - Based on locks, implemented with atomic instructions
  - If thread scheduling is implemented in user-mode

**For next lecture**

- Read chapter 9
- There will be a quiz on chapter 9
- Next lecture – Advanced synchronization
  - Monitors: Extended semaphores
  - The dining philosophers problem and solution
  - Unix and Windows synchronization mechanisms
  - Inter-process Communication mechanisms
key points

semaphores and their use
- how semaphores work
- using for locking, completion, producer/consumer

correct semaphore implementation
- solving the sleep/wakeup race
- correct use of interrupt disables and spin locks
- limitations of spin locks and interrupt disabling
- priority inversion