Preemptive scheduling
More about mutexes

Knut Omang
Ifi/Oracle
9 Sep, 2010

 ORACLE

Today:

- Interrupts
  - the 2nd source of parallelism
- Preemptive scheduling
  - how to turn a single processor into a virtual multiprocessor
- Implementation of mutual exclusion
  - with/without special hardware support

Interrupts and Exceptions

- Interrupts and exceptions (Intel terminology)
  - suspend the execution of the running thread of control
  - activates some kernel routine
- Three categories of interrupts
  - Software interrupts
  - Hardware interrupts
  - Exceptions

Software Interrupts

- INT instruction
- Explicitly issued by program
- Synchronous to program execution
- Example: INT 10h

Hardware Interrupts

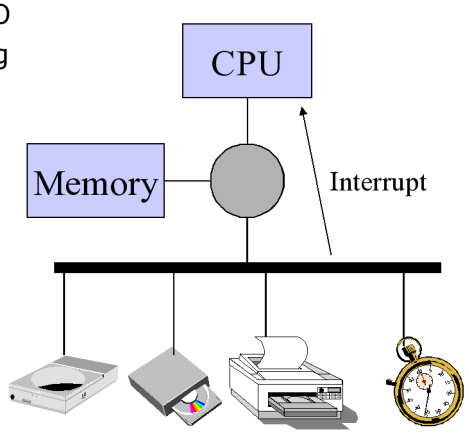
- Set by hardware
  - Clock: timer-interrupt at specified frequency
    - Separate unit or integral part of interrupt controller
  - Peripherals: Asserted by hardware when attention needed
  - Asynchronous to program execution
- Non-maskable (NMI)
  - In x86* used to report catastrophic hw failures
  - processed immediately once current instruction is finished.
- Maskable interrupts:
  - May be permanently or temporarily masked

Typical Maskable Interrupt Requests

- Peripherals generate an interrupt request to signal need for attention:
  - An action is required on the part of the program in order to continue operation
  - A previously-initiated operation has been completed with no errors encountered
  - A previously-initiated operation has encountered an error condition and cannot continue

I/O and Timer Interrupts

- Overlapping computation and I/O:
  - Within single thread: Non-blocking I/O
  - Among multiple threads: Also blocking I/O with scheduling
- Sharing CPU among multiple threads
  - Set timer interrupt to enforce maximum time-slice
  - Ensures even and fair progression of concurrent threads
- Maintaining consistent kernel structures
  - Disable/enable interrupts cautiously in kernel



Exceptions

- Initiated by processor
- Three types:
  - Fault:
    - Faulting instruction causes exception without completing. When thread resumes (after IRET), the faulting instruction is re-issued. For example page-fault
  - Trap:
    - Exception is issued after instruction completes. When thread resumes (after IRET), the immediately following instruction is issued.

      May be used for debugging
  - Abort:
    - Serious failure. May not indicate address of offending instruction

Preemptive Scheduling

- Scheduler process
  - select a READY process
  - set it up to run for a maximum of some fixed time (time-slice)
- Scheduled process computes happily (running)
  - unaware of time-slice set by the scheduler
- The process exhausts its time-slice
  - scheduler suspends the process
  - scheduler select another process to run, if possible
- Means scheduler needs to be running!
  - May need "wake-up" call!

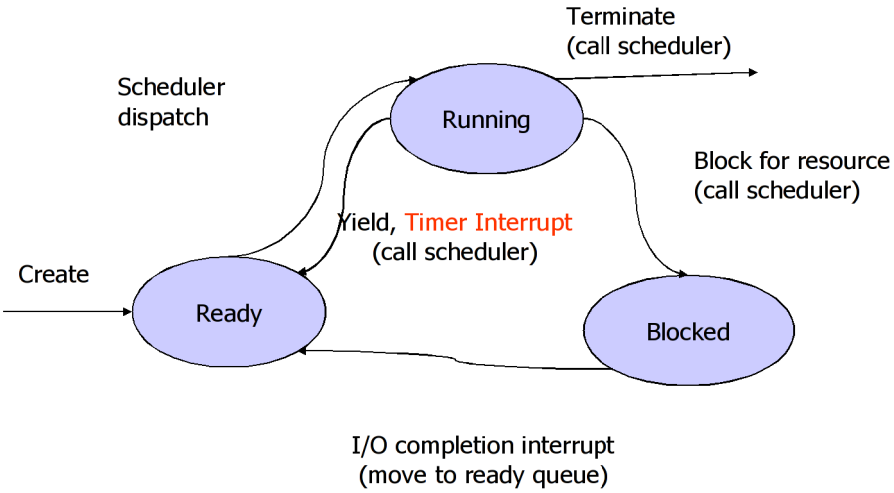
Transparent vs. non-transparent interleaving and overlapping

- Non-preemptive scheduling ("co-routines")
  - Current process or thread has control, no other process or thread will execute on the same processor before current says Yield
    - Access to shared resources simplified for single CPU systems
- Preemptive scheduling (timer and I/O interrupts)
  - Current process or thread will loose control at any time without even discovering this, and another will start executing
    - Access to shared resources must be synchronized
    - Makes single processor (almost) look like multiprocessor

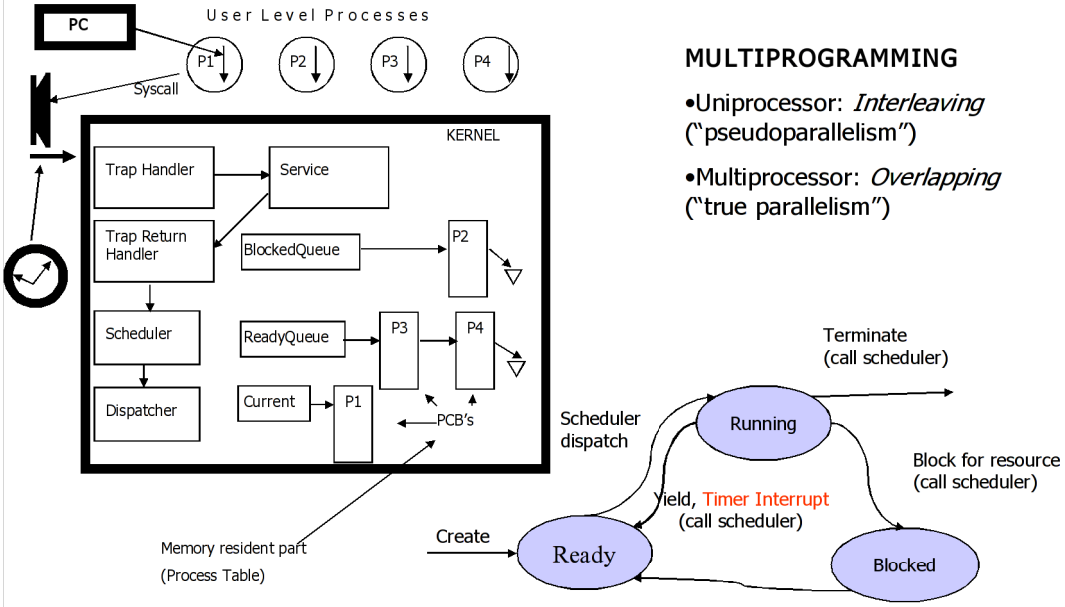
When to Schedule?

- Process/thread created
- Process/thread exits
- Process/thread blocks
- I/O interrupt
- Timer (when quantum is used)
- Explicit (yield)

Preemptive Scheduling



Process state transitions



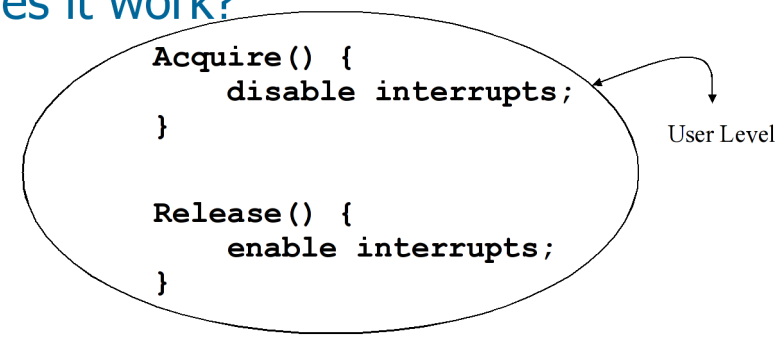
MULTIPROGRAMMING

- Uniprocessor: *Interleaving* ("pseudoparallelism")
- Multiprocessor: *Overlapping* ("true parallelism")

Disabling interrupts

- CPU scheduling
  - Internal events
    - Threads do something to relinquish the CPU
  - External events
    - Interrupts cause rescheduling of the CPU
- Disabling interrupts
  - Delay handling of external events
  - Requires care: the state of the system is in your hands (no keyboard, no mouse, no net, no scheduler, no timeout...)

Mutual exclusion by disabling interrupts: Does it work?



- Kernel cannot let **users** disable interrupts
- Kernel can provide two system calls, Acquire and Release (lock/unlock), but need ID of critical region
- Remember: no preemption when interrupts are off!

Disabling Interrupts

```
Acquire(lock) {
  disable interrupts;
  while (lock != FREE){
    enable interrupts;
    disable interrupts;
    }
  lock = BUSY;
  enable interrupts;
}
```

Spins

```
Release(lock) {
  disable interrupts;
  lock = FREE;
  enable interrupts;
}
```

- We are at Kernel Level!: So why do we need to *disable* interrupts at all?
- Why do we need to enable interrupts inside the loop in `Acquire`?
- Would this work for multiprocessors?

Disabling interrupts with reschedule

```
Acquire(lock) {
  disable interrupts;
  while (lock != FREE) {
    insert(caller, lock_queue);
    BLOCK; /* call scheduler */
  }
  lock = BUSY;
  enable interrupts;
}
```

```
Release(lock) {
  disable interrupts;
  if (nonempty(lock_queue)) {
    out(tid, lock_queue);
    READY(tid);
  }
  lock = FREE;
  enable interrupts;
}
```

- When must Acquire *re-enable* interrupts in going to sleep?
  - Before insert()?
  - After insert(), but before block?
- Would this work on multiprocessors?

Kernel level solution without busy waiting?

```
Acquire(lock) {
  while (TAS(lock)) {
    enqueue the thread;
    block;
  }
}
```

```
Release(lock) {
  if (anyone in queue) {
    dequeue a thread;
    make it ready;
  } else
  lock:=OPEN;
}
```

- **Mutual exclusion on the thread queue for each lock? (queue typically a shared resource)**
  - **example of need for nested lock applications**
- **User level: yield?**
- **What about mutexes at interrupt level?**

Handling interrupts in the kernel

- Traditional: Linux, Windows NT,..,
  - Interrupt execute in place of the current thread
    - which processor? Any? A fixed processor? All?
    - introduces dangerous dependency! See example later today!
- Solaris
  - threads execute on whatever processor is available
    - processor affinity? lots of state must be moved across caches?
  - Interrupts execute on designated threads
    - level specific

Implementation of mutual exclusion: How depends on intended usage

- Atomic memory load and store only
  - doable, but complicated, slow, and not scalable in space and time
- Disable Interrupts
  - solves problem within processor!
- Atomic read-modify-write
  - necessary to get performance!
- Message passing
  - no special hw needed
  - slow!

Mutual exclusion lock implementation with hardware support

- Special atomic instructions:
  - test&set/compare&swap(loc,v,n):

if(*loc == v) *loc = n; <return old val.of *loc>

  - fetch&add(loc,inc): *loc += inc
  - load locked/store conditional
  - holding bus – (not very scalable)

A simple solution with Test&Set

INITIALLY: Lock := FALSE; /* OPEN */

TAS (lock): (atomic!)

```
{TAS := lock;
 lock := TRUE;}
```

Spin until lock = open

```
Acquire(lock) {
  while (TAS(lock) != 0)
    ;
}
```

```
Release(lock) {
  lock = FALSE;
}
```

- Fast in case of no contention (2 instructions if inlined)
- May waste CPU time (busy waiting by all threads)
- Starvation possible: Low priority threads may never get a chance to run
- No fairness, no order, random who gets access

Hardware Support for Mutex

- Atomic memory load and store
  - Assumed by Dijkstra (CACM 1965): Shared memory w/atomic R and W operations
  - L. Lamport, "A Fast Mutual Exclusion Algorithm," ACM Trans. on Computer Systems, 5(1):1-11, Feb 1987.
- Disable Interrupts
- Atomic read-modify-write
  - IBM/360: Test And Set proposed by Dirac (1963)
  - IBM/370: Generalized Compare And Swap (1970)

Atomic Read-Modify-Write Instructions on x86

- Exchange (**`xchg`**, general x86 architecture)
  - Swap register and memory
- Compare and Exchange (**`cmpxchg`**, 486 or Pentium)
  - cmpxchg d,s: if (*d == (**`al,ax,eax`**)) *d= s;
    else (**`al,ax,eax`**) = *d;
- LOCK prefix (general x86)

Spin lock with Test&Set

INITIALLY: Lock := FALSE; /* OPEN */

Spin until lock = open

```
Acquire(lock) {
  while (TAS(lock))
    ;
}
```

```
Release(lock) {
  lock = FALSE;
}
```

TAS (lock):

```
{TAS := lock;
 lock := TRUE;}
```

- Fast in the important case (no contention)
- May waste CPU time (busy waiting by all threads)
- Starvation possible: Low priority threads may never get a chance to run
- No fairness, no order, random who gets access

Futexes (Linux)

- A kernel-space wait queue
  - attached to user-space aligned integer.
- Multiple processes or threads operate on the integer entirely in user space (using atomic operations to avoid interfering with one another)
- Use system calls to request operations on the wait queue, but only when contention

Safe interprocess communication revisited: - know what you are doing or you have lost!

- Murphy's law:
  - *Anything that can go wrong will eventually go wrong!*
- No assumptions about thread speed (time independence)
  - "Ole-Johan's semicolons" – the semicolon where it all may go wrong...
- Forward progress (but not necessarily for all threads)
- With preemptive scheduling:
  - a thread might lose control at any point!

Mutual exclusion lock usage:

watch out for the implicit partial order between locks!

```
lock(A)
  lock(B)
  unlock(B)
unlock(A)
…
lock(A)
   lock(C)
   unlock(C)
unlock(A)
```

A, B, C part of partial monotonic ordering of all locks

A > B

A > C

means

A must always be grabbed outside of C (parenthetically) if they are to be held simultaneously!

no relation between C and B yet.

Monotonic ordering of locks – why?

Process 1:

```
lock A
lock B
unlock B
unlock A
```

Process 2:

```
lock B
lock C
unlock C
unlock B
```

Process 3:

```
lock C
if <some rare case>
  lock A
  unlock A
fi
unlock C
```

Time independence:

"suppose we have this fast process and this other slow process…"

process 1: (inc, dec: atomic ops)

```
if (!o)
   lock(olock)
   if (!o) o = new object;
   unlock(olock);
inc(o.users);
<using o>
…

dec(o.users);
if (o.users == 0)
   lock(olock);
   if (o.users == 0)
      delete o; o = NULL;
   unlock(olock);
```

Can you see any problems with this algorithm??

Time independence:

"suppose we have this fast process and this other slow process…"

process 1: (inc, dec: atomic ops)

```
if (!o)
   lock(olock)
   if (!o) o = new object;
   unlock(olock);
inc(o.users);
<using o>
…

dec(o.users);
if (o.users == 0)
   lock(olock);
   if (o.users == 0)
      delete o; o = NULL;
   unlock(olock);
```

process 2:

```
if (!o) …  ← o already created, ok..
inc(o.users);
<using o>
dec(o.users);

… (proc.2 got scheduled out)

if (o.users == 0) ← o not valid!!
```