Preemptive scheduling
More about mutexes

Knut Omang
Ifi/Fast Search & Transfer ASA
6 Sept, 2001

Today:

• Application level thread packages
  – a few words about some of the implementations
• Implementation of mutual exclusion
  – with/without special hardware support
• Interrupts
  – the 2nd source of parallelism
• Preemptive scheduling
  – how to turn a single processor into a virtual multiprocessor
• Kernel level threading and interrupts
  – a real-life example from Linux 2.2.x: why “smart shortcuts” may not be so smart..
Thread Packages

- Java Threads
- POSIX threads
- Solaris threads
- WIN32 threads

Java Threads

- Integrated into the language

```java
class dummyThread extends Thread {
    int id;
    public dummyThread(int id){this.id=id;}
    public void run(){System.out.println("Hello World from thread "+id);}
}
dummyThread dt = new dummyThread(42);
dt.start();
dt.join();
```
POSIX Threads

• Language independent library

```c
pthread_create(&thread, NULL, worker, (void *)job);
pthread_join(thread);
```

Solaris Threads

• Similar to POSIX however a thread is called a Lightweight process (LWP)
• Introduces a new level of threading on top of LWPs called threads
• LWP are kernel level
• Threads are user level
WIN32 Threads

• API is designed to match the rest of the WIN32 API
• Introduces a second level of threading called fibers
• Threads are kernel level
• Fibers are user-level - and non-preemptive

Mutual exclusion lock usage:
watch out for the implicit partial order between locks!

lock(A)  A, B, C part of partial monotonic ordering of all locks
lock(B)
unlock(B)
unlock(A)
...
lock(A)  A > B
lock(C)  A > C
unlock(C)
unlock(A)  means
A must always be grabbed outside of C (parenthetically) if they are to be held simultaneously!
no relation between C and B yet.
Mutual exclusion lock implementation

Shared memory implementation without special hardware support

- Underlying assumption:
  - Load and stores are atomic
  - memory is sequentially consistent:
    - Any sequence of memory accesses from a particular processor will be seen in the same order by any other processor

> not true by default on most modern architectures...

A Mutual Exclusion Algorithm (Fisher)

Executed by process no. i.
X is shared memory.
<op> is an Atomic Operation.

Entry: * 

Critical Region

Exit: *

Repeat
  await <x = 0>
  <x := i>
  <delay>
  until <x = i>

use shared resource

<x := 0>

"While x /= 0 do skip;"

Or could block? How?

Any additional assumptions necessary?
Shared memory implementation without special hardware support

• Dekker
• Dijkstra 1965
  - general algorithm
• Peterson 1981
  - fast algorithm for 2 processes
• Lamport 1986: “A Theory of Inter process communication”
  - proof: simplest general algorithm wrt. number of memory accesses
  - store,store,load,store,load,store,store...

Problems: complicated, time consuming not constant space..
Details in Ifi Research report 208..

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 */

Spin until lock = open

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

Release(lock) {
    lock = FALSE;
}

TAS (lock): (atomic!)

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

- 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

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 is shared resource)
- Example of need for nested lock applications
- User level: yield?
- Need to decide intended usage of mutexes vs. monitors etc.
- What about mutexes at interrupt level?
Interrupts

• Why
  - Timer interrupt to do CPU management
  - Asynchronous I/O to overlap with computation

• Interrupt
  - Between instructions
  - Within an instruction
  - Enable and disable

Process state transitions

MULTI PROGRAMMING
• Uniprocessor: Interleaving ("pseudoparallelism")
• Multiprocessor: Overlapping ("true parallelism")
Preemptive Scheduling

- Create
  - Scheduler dispatch
  - Running
    - Yield, Timer interrupt (call scheduler)
    - Block for resource (call scheduler)
    - I/O completion interrupt (move to ready queue)
  - Ready
- Terminated (call scheduler)

Transparent vs. non-transparent interleaving and overlapping

- Non-preemptive scheduling (“Yield”)
  - 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
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!
• Uni-processor only, won’t work on multiprocessors
Disabling Interrupts

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

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?
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

Synchronization mechanisms
(increasing complexity)

- Mutual exclusion locks
  - short critical regions (simple data structure updates)
- Semaphores
  - generalized mutexes (more than one inside)
- Monitors (condition queues)
  - longer regions protecting resources, sleep with wakeup when resource is available
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!

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

Real life example:
Process queueing/dequeuing in linux 2.2.x

(From the linux 2.2.16 kernel source:)

/* Note that we only need a read lock for the wait queue (and thus do
* not have to protect against interrupts), as the actual removal from
* the queue is handled by the process itself.
*/

Goal: an implementation of monitors (condition queues) in Linux (kernel level)

Why?
- Linux offers low level primitives only (abstracted and simplified)

spin_lock/spin_unlock -- mutual exclusion locks based on busy waiting
spin_lock_irqsave/spin_unlock_irqrestore -- mutual exclusion: interrupt disabling+spin
enqueue/dequeue(task,queue) -- add/remove myself to/from a process queue
schedule() -- invoke scheduler (yield)
wake_up_next(queue) -- next process in "queue" put back on the run queue
Condition variables implementation

/* assuming lock is held and * interrupts turned off */

void cond_wait(cond c, mutex lock)
{
    enqueue(current, c.queue);
    spin_unlock_irqrestore(lock);
    schedule();
    dequeue(current, c.queue);
    spin_lock_irqsave(lock);
}

/* assuming lock is held and * interrupts turned off */

void cond_signal(cond c)
{
    wake_up_next(c.queue);
}

Example case (implementation)

Usage: resource management
...
spin_lock_irqsave(lock);
if (<my resource not available>)
    cond_wait(c, lock);
<grab resource>
spin_unlock_irqrestore(lock);
...
spin_lock_irqsave(lock);
<release resource>
cond_signal(c);
spin_unlock_irqrestore(lock);

Linux impl. of enqueue/wake_up

global mutex queue_lock;
void enqueue(task t, queue q)
{
    spin_lock_irqsave(queue_lock);
    < do the queuing of t in q>
    spin_unlock_irqrestore(queue_lock);
}

task dequeue(queue t)
{
    task t;
    spin_lock(queue_lock);
    t := pop(queue);
    spin_unlock(queue_lock);
    return t;
}
Example case: proc.1/proc.2/interrupt,p.1
scenario on dual processor system

process A
(processor 1)
(inside tcp/ip stack)
....
dequeue(A,tcp)
spin_lock(queue_lock)
<holds queue_lock..>
...Interrupted!...

interrupt context
(executing within A)
(processor 1)

process B
(processor 2)
<holds lock L, int.off>
cond_wait
enqueue(B,res)
...spinning on
queue_lock....!

spin_lock_irqsave(L)
...spinning on L...