Preemptive Scheduling and Mutual Exclusion with Hardware Support

Thomas Plagemann

With slides from
Otto J. Anshus & Tore Larsen (University of Tromsø) and Kai Li (Princeton University)

Preemptive Scheduling

- Scheduler select a READY process and sets it up to run for a maximum of some fixed time *(time-slice)*
- Scheduled process computes happily, oblivious to the fact that a maximum time-slice was set by the scheduler
- Whenever a running process exhausts its time-slice, the scheduler needs to suspend the process and select another process to run (assuming one exists)
- To do this, the scheduler needs to be running! To make sure that no process computes beyond its time-slice, the scheduler needs a mechanism that guarantees that the scheduler itself is not suspended beyond the duration of one time-slice. *A "wake-up" call is needed*

Interrupts and Exceptions

- Interrupts and exceptions suspend the execution of the running thread of control, and 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 components (for example *timer*), and peripheral devices (for example disk)
  - *Timer component, set to generate timer-interrupt at any specified frequency! Separate unit or integral part of interrupt controller*
- Asynchronous to program execution
- Non-maskable (NMI), and maskable interrupts.
  - NMI are processed immediately once current instruction is finished.
  - Maskable interrupts may be permanently or temporarily masked

Maskable Interrupt Request

- Some IO devices generate an interrupt request to signal that:
  - 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

Non-maskable Inerrupt Requests

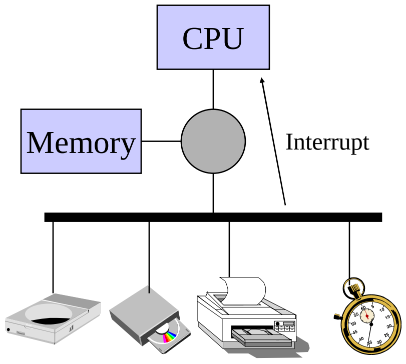
- In the PC-compatible world, the processor's non-maskable interrupt request input (NMI) is used to report catastrophic HW failures to the OS

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
- *Have used Intel terminology in this presentation. Classification, terminology, and functionality varies among manufacturers and authors*

I/O and Timer Interrupts

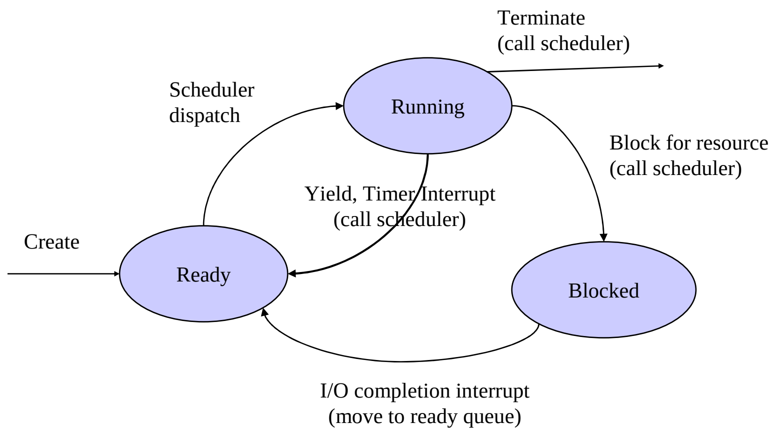
- Overlapping computation and I/O:
  - Within single thread: Non-blocking I/O
  - Among multiple threads: Also blocking I/O with scheduling
- Sharing CPU among multiple threads
  - Set timer interrupt to enforce maximum time-slice
  - Ensures even and fair progression of concurrent threads
- Maintaining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



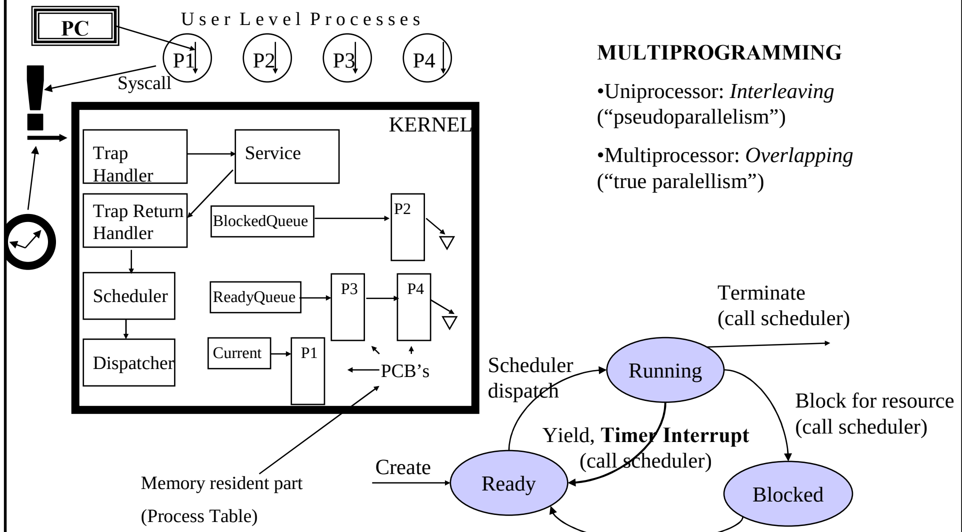
When to Schedule?

- Process created
- Process exits
- Process blocks
- I/O interrupt
- Timer

Process State Transitions



Process State Transitions (cont.)



MULTIPROGRAMMING

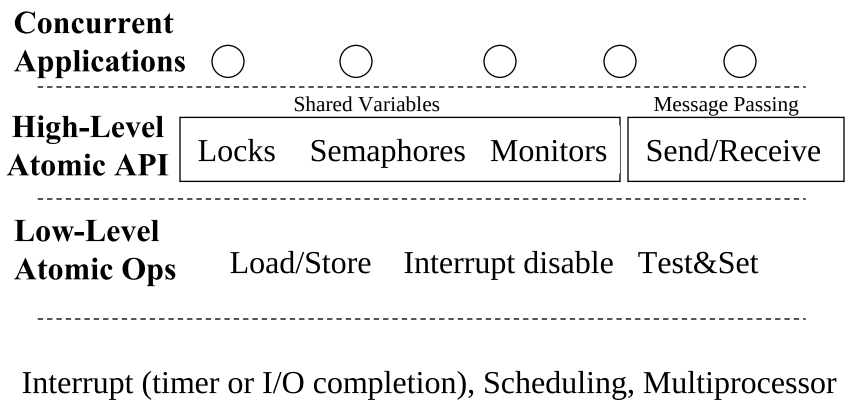
•Uniprocessor: *Interleaving* ("pseudoparallelism")

•Multiprocessor: *Overlapping* ("true paralellism")

Transparent vs. Non-transparent Interleaving and Overlapping

- Non-preemptive scheduling ("Yield")
  - Current process or thread has control, no other process or thread will execute before current says Yield
    - Access to shared resources simplified
- 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

Implementation of Synchronization Mechanisms



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)

A Fast Mutual Exclusion Algorithm (Fischer)

Executed by process no. i.

X is shared memory.

<op> is an Atomic Operation.

”While x ≠ 0 do skip;”

Or could block? How?

Entry:

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

Critical Region

```
use shared resource
```

Exit

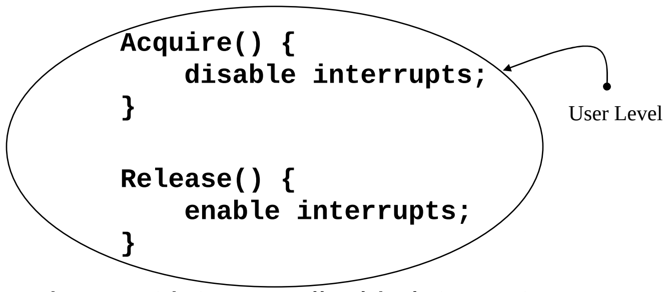
```
<x := 0>;
```

We are assuming that COMMON CASE will be fast and that all processes will get through eventually

Disable 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
    - and make sure we have a safe ENTRY or EXIT

Does This Work?



```
Acquire() {
    disable interrupts;
}

Release() {
    enable interrupts;
}
```

- Kernel cannot let **users** disable interrupts
- Kernel can provide two system calls, Acquire and Release, but need ID of critical region
- Remember: Critical sections can be arbitrary long (no preemption!)
- Used on uni-processors, but won’t work on multiprocessors

Disabling Interrupts with Busy Wait

```
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?
- Why not have a “disabled” Kernel?

Using Disabling Interrupts with Blocking

```
Acquire(lock) {
  disable interrupts;
  while (lock == BUSY) {
    insert(caller, lock_queue);
    BLOCK;
  } else
    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?

Atomic Read-Modify-Write Instructions

- What we want: Test&Set(lock):
  - Returns TRUE if lock is TRUE (closed), else returns FALSE and closes lock.
- Exchange (**xchg**, x86 architecture)
  - Swap register and memory
- Compare and Exchange (**cmpxchg,** 486 or Pentium)
  - `cmpxchg d,s: If Dest = (al,ax,eax), Dest = SRC;`
    `               else  (al,ax,eax) = Dest`
- LOCK prefix in x86
- Load link and conditional store (MIPS, Alpha)
  - Read value in one instruction, do some operations
  - When store, check if value has been modified. If not, ok; otherwise, jump back to start
- The Butterfly multiprocessor
  - atomicadd: one processor can read and increment a memory location while preventing other processors from accessing the location simultaneously

A Simple Solution with Test&Set

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

TAS (lock):

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

Spin until lock = open

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

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

- Waste CPU time (busy waiting by all threads)
- Low priority threads may never get a chance to run (starvation possible because other threads always grabs the lock, but can be lucky…): No Bounded Waiting (a MUTEX criteria)
- No fairness, no order, random who gets access

Test&Set with Minimal Busy Waiting

CLOSED = TRUE
OPEN = FALSE

```
Acquire(lock) {
  while (TAS(lock.guard))
    ;
  if (lock.value) {
   enqueue the thread;
   block and lock.guard:=OPEN;
   %Starts here after a Release()
  }
  lock.value:=CLOSED;
  lock.guard:=OPEN;
}
```

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

- Two levels: Get inside a mutex, then check resource availability (and block (remember to open mutex!) or not).
- Still busy wait, but only for a short time
- Works with multiprocessors

A 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;
}
```

- BUT: No mutual exclusion on the thread queue for each lock: queue is shared resource
  - Need to solve another mutual exclusion problem
- Is there anything wrong with using this at the user level?
  - Performance
  - “Block”??

Different Ways of Spinning

```
while (TAS(lock.guard))
  ;
```

- Always execute **TAS**

```
while (TAS(lock.guard)) {
  while (lock.guard)
    ;
}
```

- Perform **TAS** only when **lock.guard** is likely to be cleared
  - TAS is expensive

Using System Call **Block/Unblock**

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

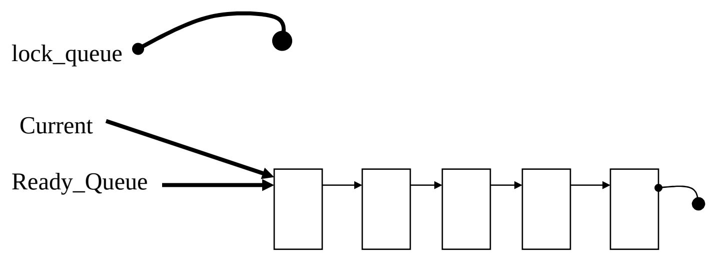
```
Release(lock) {
  lock = 0;
  Unblock( lock );
}
```

- Block/Unblock are implemented as system calls
- How would you implement them?
  - Minimal waiting solution

Block and Unblock

```
Block (lock) {
    insert (current, lock_queue, last);
    goto scheduler (;
}
```

```
Unblock (lock) {
    insert (out (lock_queue, first), Ready_Queue, last);
    goto scheduler;
}
```


lock_queue
Current
Ready_Queue