Introduction to Synchronization

Professor Hugh C. Lauer
CS-3013 — Operating Systems

(Slides include copyright materials from *Operating Systems: Three Easy Step*, by Remzi and Andrea Arpaci-Dusseau, from *Modern Operating Systems*, by Andrew S. Tanenbaum, 3rd edition, and from other sources)

Challenge

- Now that concurrent execution in the same address space has been established, …
- … how do threads or concurrent actions synchronize themselves with each other?

Challenge (continued)

- How does one thread "know" that another has completed an action?

- How do separate threads "keep out of each others' way" with respect to some shared resource?

- How do threads divide up and share the load of particularly long computations?

Context

- Separate threads in same process
- Separate threads or processes making system calls to same kernel
- Separate processes sharing some common resource
- …

Digression (thought experiment)

```
int y = 0;
```

```
int main(int argc, char **argv)
{
  extern int y;

  y = y + 1;

  return y;
}
```

Upon completion of `main`, `y == 1`

Thought experiment (continued)

```
int y = 0;
```

Thread 1

```
int main(int argc, char
     **argv)
{
  extern int y;

  y = y + 1;

  return y;
}
```

Thread 2

```
int main2(int argc, char
     **argv)
{
  extern int y;

  y = y - 1;

  return y;
}
```

Assuming threads run "at the same time," what are *possible values* of y after both threads terminate?

Definition – *Atomic Operation*

- **An operation that either happens entirely or not at all**
  - No partial result is visible or apparent
  - Appears to be non-interruptible
- **If two atomic operations happen “at the same time”**
  - Effect is as if one is first and the other is second
  - (Usually) don’t know which is which

Hardware Atomic Operations

- **On** (essentially) **all computers, <u>reading</u> and <u>writing</u> of machine words can be considered as *atomic***
    - Non-interruptible
    - It either happens or it doesn't
    - Not in conflict with any other operation
- **When two attempts to read or write the *same* data, one is first and the other is second**
    - Don't know which is which!
- **No other guarantees**
    - (unless we take extraordinary measures)

Atomic Operations (continued)

- **Most modern processors provide some additional atomic operations**

- **Read-modify-write of memory values — e.g.,**
  - Test-and-Set
  - Fetch-and-add
  - Swap Register with Memory
  - Memory-memory Exchange
  - …

Generally confined to one memory location

Difficult to implement with multiple processors and multiple caches

Definitions

- **Definition: *race condition***
  - When two or more concurrent activities try to do something with the same set of variables resulting in different values
  - Random outcome

- ***Critical Region* (aka *critical section*)**
  - One or more fragments of code that operate on the same data, such that at most one activity at a time may be permitted to execute anywhere in that set of fragments.

Synchronization – Critical Regions

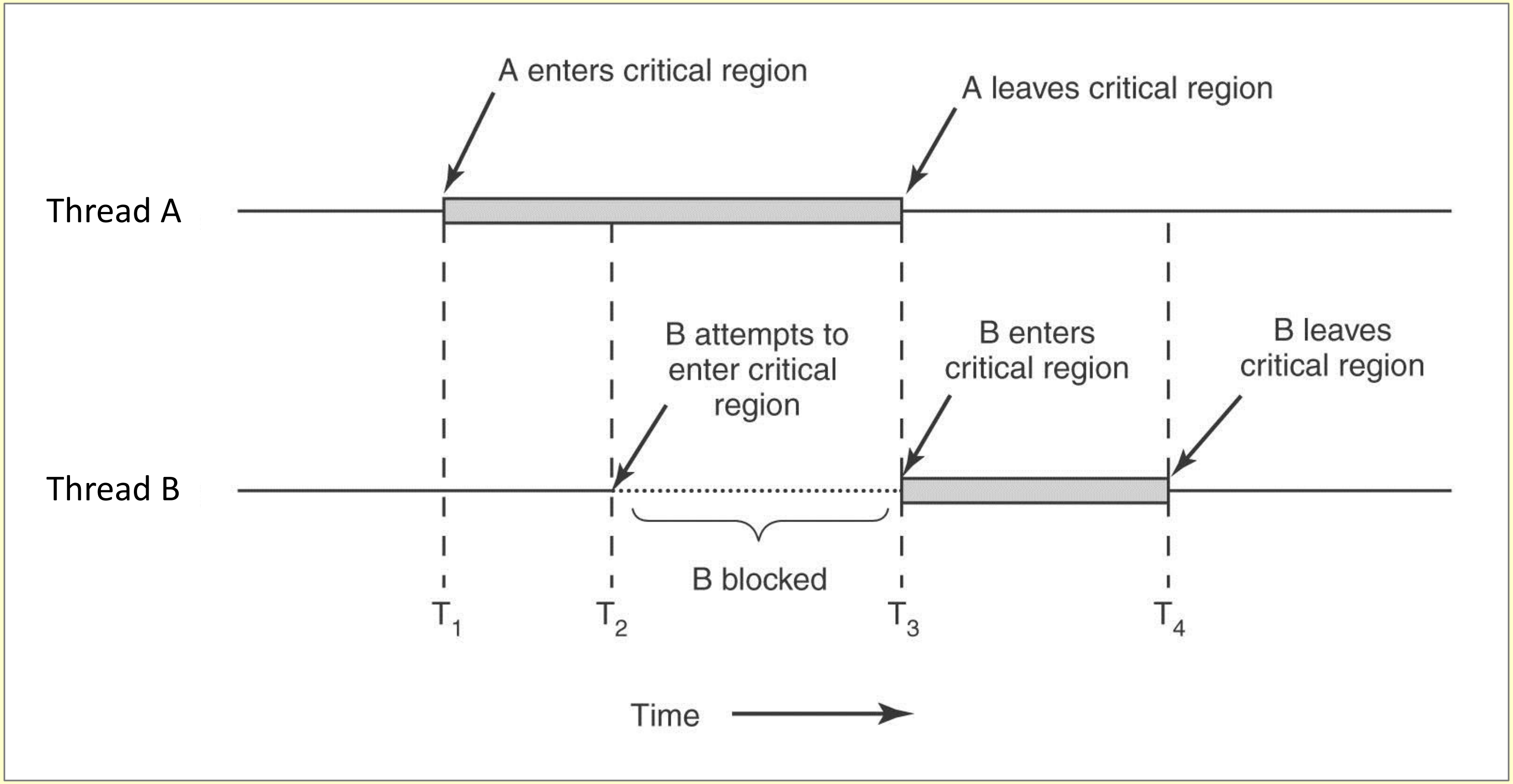


Fig 2-22, Tanenbaum, See also OSTEP Fig. 26.7

Class Discussion

- **How do we keep multiple computations from being in a critical region at the same time?**
  - Especially when number of computations is > 2
  - Remembering that read and write operations are atomic

example

Possible ways to protect critical regions

- **Without OS assistance**
  - Locking variables & busy waiting
  - Wait-free solutions

- **With OS assistance — abstract data synchronization operations**
  - Single processor
  - Multiple processors

Controlling access to a critical region

- **Classical requirements (Dijkstra):–**
  - Only one computation in critical section at a time
  - Symmetrical among *n* computations
  - No assumption about relative speeds
  - A stoppage outside critical section does not lead to potential blocking of access to region by others
  - No *starvation* — i.e. no combination of timings that could cause a computation to wait forever to enter its critical section
- **Practical requirement:–**
  - Completion in bounded time, regardless of behavior of other processes — i.e., *wait-free*

Non-solution

```
int turn = 0;
```

Thread 1

```
while (TRUE) {
    while (turn !=0)
        /*loop*/;
    critical_region();
    turn = 1;
    noncritical_region1();
};
```

Thread 2

```
while (TRUE) {
    while (turn !=1)
        /*loop*/;
    critical_region();
    turn = 0;
    noncritical_region2();
};
```

What is wrong with this approach?

Peterson's solution (2 processes or threads)

```
int turn = 0;
int interested[2];
```

```
void enter_region(int process) {
   int other = 1 - process;

   interested[process] = TRUE;
   turn = process;
   while (turn == process &&
      interested[other] == TRUE)
      /*loop*/;
};

void leave_region(int process) {
   interested[process] = FALSE;
};
```

This is a simplification of Dijkstra's 1965 solution for *n* processes. See .pdf

Busy waiting!

Another approach: Test & Set

(atomic read-modify-write instruction built into CPU hardware)

```
int lock = 0;
extern int TestAndSet(int *i);
   /* atomically sets the value of
   i to 1 and returns the previous
   value of i.  */

void enter_region(int *lock) {
   while (TestAndSet(lock) == 1)
      /* loop */ ;
};

void leave_region(int *lock) {
   *lock = 0;
};
```

What about this solution?

Busy waiting!

Variations of Atomic Operations

- **Exchange (a, b)**
  - temp = b
  - b = a
  - a = temp

- **Compare and Swap (var, old, new)**
  - previous = var;
  - If (previous == old)
    var = new;
  - return previous;

Net effect

- We can simulate the *atomicity* of critical sections using instructions available in computer processor

- Is this the best approach?

- Sometimes yes, sometimes no!

There are entire courses and textbooks devoted to this subject!
The Art of Multiprocessor Programming, by Maurice Herlihy & Nir Shavit, Morgan Kaufman, 2012

Possible Ways to Protect Critical Sections

- Without OS assistance
  - Locking variables & busy waiting
  - Wait-free solutions

- **With OS assistance — abstract data synchronization operations**
  - Single processor
  - Multiple processors

Protecting Critical Section with OS Assistance

Implement an *abstraction*:–

- **A data type called *semaphore***
  - Non-negative integer values plus a queue
- **An operation *wait_s*(semaphore **s*) such that**
  - Atomically test *s* > 0 and, if so, decrement *s* and proceed.
  - if *s* = 0, block the process or thread until some other process or thread executes *post_s*(*s*)
- **An operation *post_s*(semaphore **s*):–**
  - If one or more processes or threads are blocked on *s*, allow precisely one of them to unblock and proceed
  - Otherwise, atomically increment *s* and continue

Critical Section control with Semaphore

```
semaphore mutex = 1;
```

Thread 1

```
while (TRUE) {
    wait_s(mutex);

    critical_region();

    post_s(mutex);

    noncritical_region1();
};
```

Thread 2

```
while (TRUE) {
    wait_s(mutex);

    critical_region();

    post_s(mutex);

    noncritical_region2();
};
```

Does this meet the requirements for controlling access to critical sections?

Semaphores – History

- **Introduced by E. Dijkstra in 1965**
- ***wait_s*() was called *P*()**
  - Initial letter of a Dutch word meaning “test”
- ***post_s*() was called *V*()**
  - Initial letter of a Dutch word meaning “increase”

- **In Linux kernel (and some other modern systems)**
  - *wait_s*() is called *down()*
  - *post_s*() is called *up()*

Abstractions

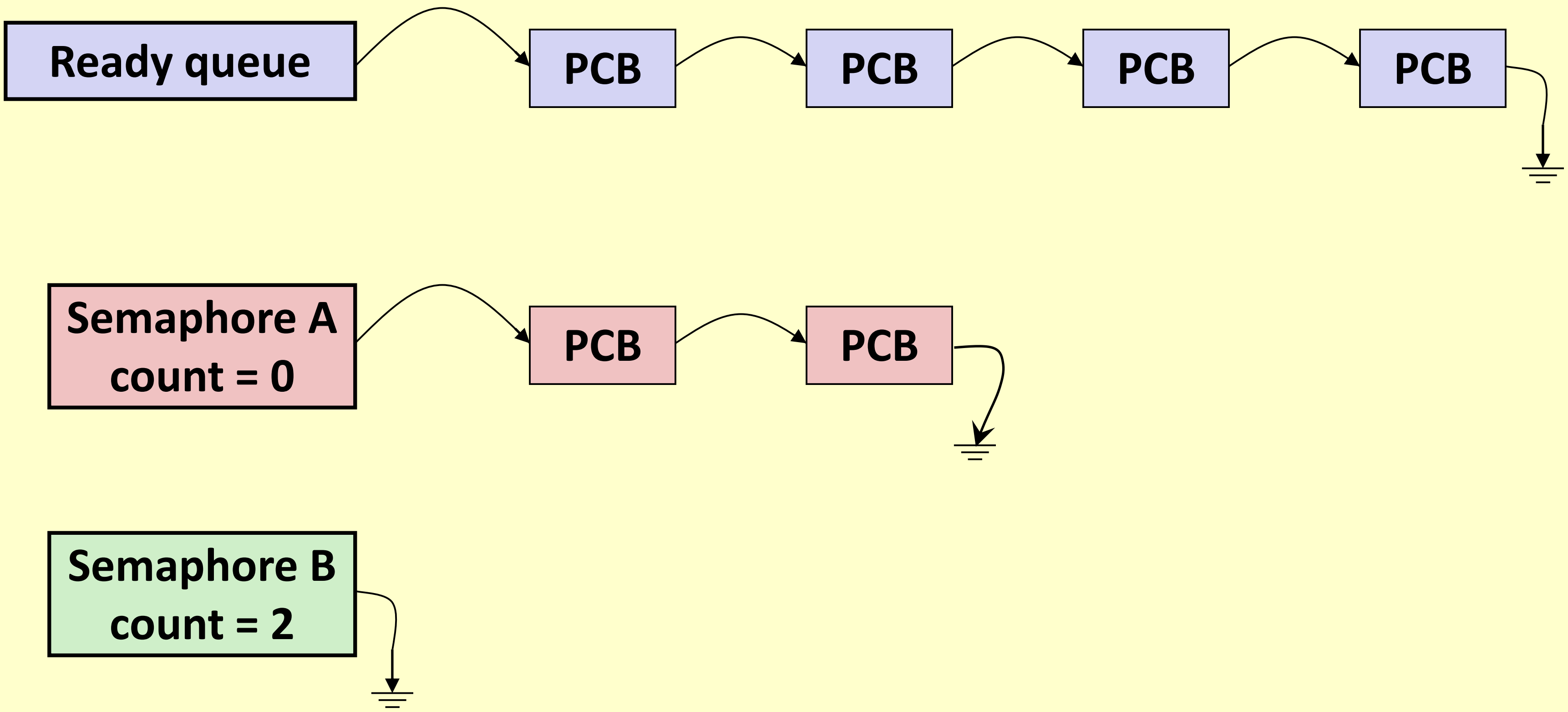
- **The *semaphore* is an example of a powerful *abstraction* defined by OS**
  - I.e., a data type and some operations that add a capability that was not in the underlying hardware or system.
- **Any program can use this abstraction to control critical sections and to create more powerful forms of synchronization among computations.**
- **OSTEP, Ch 28**

Data Structures for Implementing Semaphores

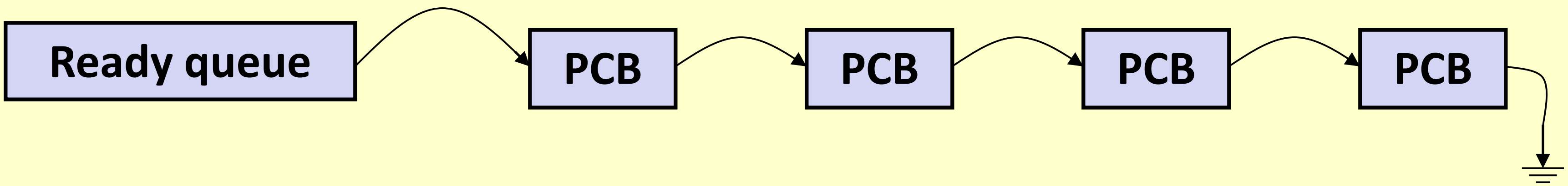
```
class State {
    long int PSW;
    long int regs[R];
    /*other stuff*/
}
class PCB {
    PCB *next, *prev, *queue;
    State s;

    PCB (…); /*constructor*/
    ~PCB(); /*destructor*/
}
```

```
class Semaphore {
  int count;
  PCB *queue;

  friend wait_s(Semaphore *s);
  friend post_s(Semaphore *s);

  Semaphore(int initial);
    /*constructor*/
  ~Semaphore();
    /*destructor*/
}
```

Semaphore Data Structures (continued)

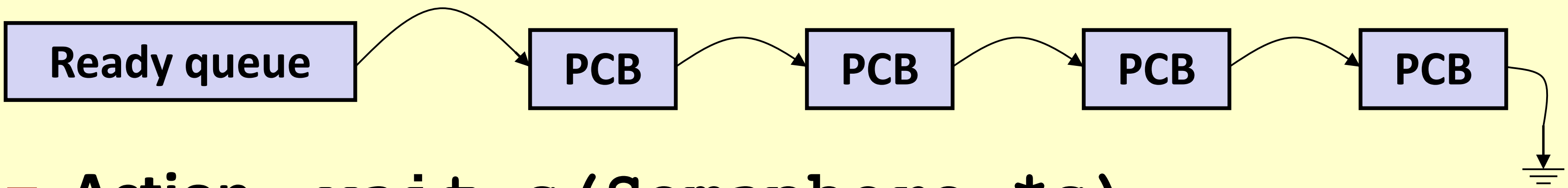


From earlier lesson



- **Action – dispatch a process or thread to CPU**
  - Remove first PCB from *ReadyQueue*
  - Load registers and PSW
  - Return from interrupt or trap
- **Action – interrupt a process or thread**
  - Save PSW and registers in PCB
  - If not blocked, insert PCB into *ReadyQueue* (in some order)
  - Take appropriate action
  - Dispatch same or another process from *ReadyQueue*

Implementation – Semaphore actions



- **Action – `wait_s(Semaphore *s)`**
  - Implement as a Trap (with interrupts *disabled*)

  `if (s->count == 0)`

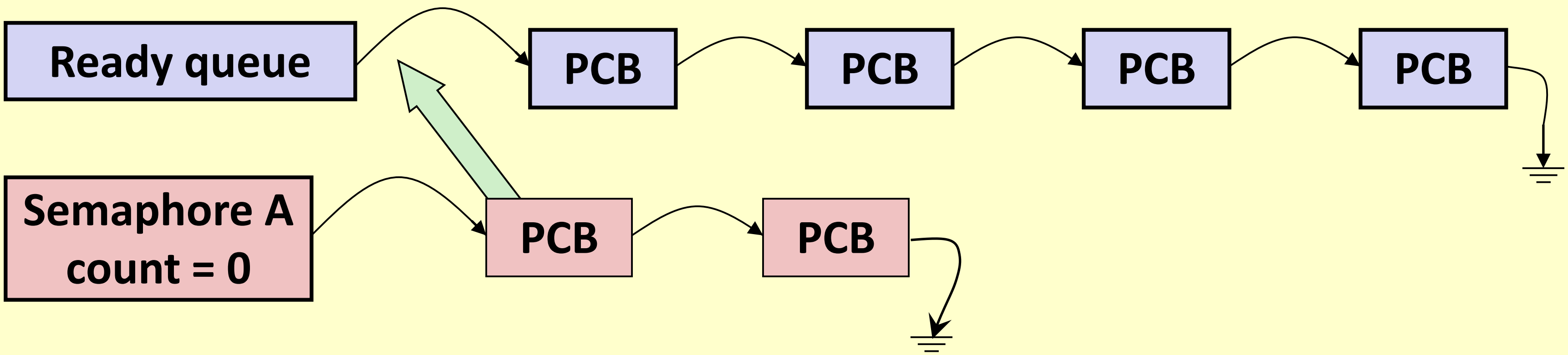
  - Save registers and PSW in PCB
  - Queue PCB on **`s->queue`**
  - Dispatch next process on ReadyQueue

  `else`

  - **`s->count = s->count - 1;`**
  - Re-dispatch current process

Event wait

Implementation – Semaphore actions



- **Action – `post_s(Semaphore *s)`**
  - Implement as a Trap (with interrupts disabled)

`if (s->queue != null)`

- Save current process in ReadyQueue
- Move first process on **`s->queue`** to ReadyQueue
- Dispatch some process on ReadyQueue

`else`

- **`s->count = s->count + 1;`**
- Re-dispatch current process

Event completion

Interrupt Handling

- (Quickly) analyze reason for interrupt
- Execute equivalent of *post_s* to appropriate semaphore as necessary
  - Implemented in device-specific routines
  - Real work of interrupt handler is done in a separate task-like entity in the kernel
- More about interrupt handling later in the course

Semaphores — Summary

- **Interrupts transparent to processes**
- **Can be used to simulate *atomic* actions**
  - On single processor systems
- **`wait_s()` and `post_s()` behave as if they are atomic**
- **Useful for synchronizing among processes and threads**

Semaphores – Epilogue

- *A* way for generic processes to synchronize with each other

- Not the *only* way

- Not even the *best* way in most cases

- More in next topic

Questions?

Synchronization and Atomic Operations within Linux Kernel

- **Reading assignment**
  - Robert Love, *Linux Kernel Development*, 3rd edition, Chapter 10
- **Lots of tools**
  - Atomic operations on integer (32- and 64-bit) and bits
  - Spin Locks
  - Kernel Semaphores
  - Kernel Mutexes
  - …

Complications for Multiple Processors

- **Disabling interrupts is not enough for implementing “atomic” updates to kernel data**
  - Semaphore operations must themselves be implemented in critical sections!
  - Queuing and dequeuing PCB’s must also be implemented atomically
  - Interrupt handlers need to update status atomically
  - Other control operations need protection

- **These problems need deeper thought!**
  - *The* central issue in transition from single processor kernel to multi-threaded kernel (Linux 2.4.x to 2.6.x)

Hierarchy of Solutions

- **Wait-free operations**
  - Supported by hardware
- **Disciplined use of spin locks**
  - Avoid unbounded locking
- **Higher-level synchronization primitives within kernel**

Wait-Free Synchronization

- **A whole mathematical theory about efficacy of atomic hardware operations**
    - Atomic Read-Modify-Write is the weakest
    - Exchange is stronger
    - Compare-and-Swap is the strongest
- **All require extraordinary circuitry in processor memory, and bus to implement atomically**
- **Herlihy, Maurice, "Wait-Free Synchronization,"** ***ACM Transactions on Programming Languages and Systems,*** **vol 11, #1, January 1992, pp. 124-149 (.pdf)**

Compare-and-Swap

```
int compare_and_swap (int* reg,
     int oldval, int newval)
{
   int old_reg_val = *reg;
   if (old_reg_val == oldval)
      *reg = newval;
   return old_reg_val;
}
```

- **As implemented in IBM 370 and successors**
  - **Test `old_reg_val` again `oldval` to determine if success**
- **Similar to CMPXCH on Intel architectures**

What can we do with Compare-and-Swap?

- Atomic arithmetic or bitwise operations
- Add an object to or remove an object from head of a linked list
- …

- All with no lock variable!

Used to implement Linux kernel atomic operations

Multi-word Compare-and-Swap Operations

- Harris, Timothy L., Fraser, Keir, and Pratt, Ian A., "A Practical Multi-Word Compare-and-Swap Operation," University of Cambridge Computer Laboratory, Cambridge, UK (.pdf)
- For updating queues, linked lists, etc., without lock variables

Load Linked / Store Conditional

- **Alternative to Compare-and-Swap**
  - DEC Alpha, PowerPC, MIPS, ARM, etc.
- ***Load Linked*** **(LL)**
  - Like a regular load instruction, but keep tabs on that memory address
  - Keep a hardware flag on memory location (in cache)
- ***Store Conditional*** **(SC)**
  - Like a regular store instruction, but *fails* if
    - Flagged location has been updated since LL instruction
    - Context switch occurs since LL instruction
    - Anything else that might have changed target location

Load Linked / Store Conditional (continued)

- **Works well with RISC processors**
- **Works well with multi-level caches**
    - Provided cached implement cache-consistency
- **Easy to simulate Compare-and-Swap**

- **May *not* be nested**

Load Linked / Store Conditional (example)

Atomic increment

```
try:    LL      R2, 0(R1)
        ADD     R3, R2, #1
        SC      R3, 0(R1)
        BEQZ    R3, try
```

Questions?

Linux Kernel — Hierarchy of Solutions

- **Wait-free operations**
  - Supported by hardware
- **Disciplined use of spin locks**
  - Avoid unbounded locking
- **Higher-level synchronization primitives within kernel**

Disciplined Use of Spin Locks

- **Spin Lock:– A lock variable that is waited-for by busy waiting**
  - E.g., *Test-and-Set*

- **Widely used in Linux kernel**
  - Protect shared data
  - Critical sections must be ve*ry short*
  - Robust in multi-processor environment
  - May be statically or dynamically allocated

Disciplined Use of Spin Locks

- **Rules**
  - Critical sections must *very* short – a few nanoseconds!
  - Process holding a spinlock may not be pre-empted or rescheduled by any other process or kernel routine
    - I.e., disable interrupts!
  - Process may not sleep, take page fault, or wait for *any reason* while holding a spin lock
- **Interrupt handler must**
  - Disable interrupts on current processor before locking

Reader-Writer Spin Locks

- **Multiple tasks may hold a lock as "reader"**
- **Only one task may hold lock as "writer"**
  - Must wait till all readers clear
- **Reader may upgrade status to "writer"**

Questions?

Hierarchy of Solutions

- **Wait-free operations**
  - Supported by hardware
- **Disciplined use of spin locks**
  - Avoid unbounded locking
- **Higher-level synchronization primitives within kernel**

Additional Synchronization Primitives in Linux Kernel

- **Semaphores**
  - Counting and binary
- **Mutexes**
  - Specialized variation of binary semaphore
- **Completion Variable**
- **Sequential Lock**

These all enable processes to sleep, be pre-empted, wait for events, etc.

Questions?