Introduction to Synchronization

CS-3013 Operating Systems
Hugh C. Lauer

(Slides include materials from Modern Operating Systems, 3rd ed., by Andrew Tanenbaum and from Operating System Concepts, 7th ed., by Silbershutz, Galvin, & Gagne)
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 share 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
• ...

CS-3013, A-Term 2011
Digression (thought experiment)

\[ \text{static int } y = 0; \]

\[
\text{int main(int argc, char **argv)} \\
\{ \\
    \text{extern int } y; \\
    y = y + 1; \\
    \text{return } y; \\
\}
\]

Upon completion of \text{main}, \( y = 1 \)
Thought experiment (continued)

```c
static int y = 0;
```

Thread 1
```c
int main(int argc, char **argv)
{
    extern int y;
    y = y + 1;
    return y;
}
```

Thread 2
```c
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 \texttt{y} after both 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 (nearly) all computers, reading and writing operations 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)
Definitions

• **Definition: race condition**
  • When two or more concurrent activities are trying to do something with the same variable 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

Process A

Process B

A enters critical region

A leaves critical region

B attempts to enter critical region

B enters critical region

B leaves critical region

B blocked

Time

T₁

T₂

T₃

T₄
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
Possible ways to protect Critical Sections

- Without OS assistance — Locking variables & busy waiting
  - Peterson’s solution (§2.3, p. 123)
  - Hardware atomic read-modify-write – e.g.
    - Test & Set
    - Compare and swap

- With OS assistance — abstract synchronization operations
  - Single processor
  - Multiple processors
Requirements – Controlling Access to a Critical Section

- 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 others
- No starvation — i.e. no combination of timings that could cause a computation to wait forever to enter its critical section
Non-solution

```c
static int turn = 0;
```

Thread 1

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

Thread 2

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

What is wrong with this approach?
Peterson’s solution

```c
static int turn = 0;
static 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
Another approach: Test & Set Instruction
(atomic read-modify-write instruction built into CPU hardware)

```c
static 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;
}
```

Busy waiting!

What about this solution?
Variations of Atomic Operations

- Exchange \((a, b)\)
  - \(\text{temp} = b\)
  - \(b = a\)
  - \(a = \text{temp}\)

- Compare and Swap \((\text{var}, \text{old}, \text{new})\)
  - \(\text{previous} = \text{var};\)
  - \(\text{If (previous} = \text{=} \text{old)}\)
    - \(\text{var} = \text{new};\)
  - \(\text{return previous};\)
Wait-Free Synchronization

• A whole mathematical theory about efficacy of these 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
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!
Protecting a 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
  - if s > 0, atomically decrement s and proceed.
  - if s = 0, block the process until some other computation executes post_s(s).
- An operation post_s(semaphore *s):
  - If one or more processes are blocked on s, allow precisely one of them to unblock and proceed.
  - Otherwise, atomically increment s and continue
Critical Section control with Semaphore

static 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.
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)

Ready queue

Semaphore A
count = 0

Semaphore B
count = 2

PCB

PCB

PCB

PCB
Implementation

- **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
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
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
Complications for Multiple Processors

- Disabling interrupts is not sufficient for atomic operations
  - Semaphore operations must themselves be implemented in critical sections
  - Queuing and dequeuing PCB’s must also be implemented in critical sections
  - Other control operations need protection

- These problems all have solutions but need deeper thought!
Synchronization in Multiple Processors (continued)

- Typical solution – *spinlocks*
  - Kernel process (or interrupt handler) trying to enter a kernel critical section does *busy waiting*
  - *Hardware atomic operations*

- Constraint
  - Critical sections are very short – a few nanoseconds!
  - Process holding a spinlock may not be pre-empted or rescheduled by any other process or kernel routine
  - Process may not sleep, take page fault, or wait for *any reason* while holding a spinlock
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
  - See §2.3 of Tanenbaum
Questions?
Process States

- new
- admitted
- interrupt
- exit
- terminated
- ready
- running
- waiting

I/O or event completion
Scheduler dispatch
I/O or event wait