Mutual Exclusion

Chapter 5 of [OS4e],
Chapter 7 of [OSC]:

- Cooperation versus Interference
- Critical Sections
- Software Solution
- Hardware Solution
- Operating System Solution

Cooperation versus Interference

- Threads share memory (same address space)
  - read/write same memory locations
- Scheduling of threads results in their traces of instructions being interleaved in unpredictable fashion

How can one ensure cooperation, rather than interference, between threads?

- Processes face similar problem over shared resources
  - e.g., UNIX System V.4 provides system calls to create shared memory (shmget) and attach it (shmat) to address space of process
Examples: Thread A       Thread B
1. \(x=1\)               \(y=2\)
   Result independent of order of execution

2. \(x=1; y=1\)           \(x=2; y=2\)
   Result nondeterministic:
   – both 1
   – both 2
   – one 1, the other 2
N.B. loading/storing word in memory is atomic operation

Critical Sections

Organize each thread so as to include

- **critical section**
  – where access to shared resource is allowed

When not in critical section, thread is known not to interfere with other threads

Guarantee somehow that

- at most one thread is in its critical section at any time (*mutual exclusion*)
Denying entry to critical section altogether guarantees mutual exclusion!

Also require that
- one of threads waiting to enter its critical section must eventually be permitted to do so (*progress*)

This assumes that
- thread in critical section must eventually leave

Unfortunately, threads might be allowed to enter and leave critical section, whilst one thread has to wait forever (*starvation*)

This requires
- scheduling to be *fair* in some sense

---

### Software Solution

```java
public class Peterson {
    // Peterson's algorithm guarantees mutual
    // exclusion between two threads
    // in that at most one thread can be in its
    // critical section at a time.

    static volatile int turn;
    static volatile boolean flag0 = false;
    static volatile boolean flag1 = false;

    public static void main(String argv[]) {
```
new Thread(new Runnable(){
    public void run(){
        for(;;){
            flag0 = true;
            turn = 1;
            while (flag1 && (turn == 1)){
                Thread.yield();
                // can be omitted
            }
            // Thread 0 inside
            // critical section
            flag0 = false;
        }
    } // can be omitted
}).start();

new Thread(new Runnable(){
    public void run(){
        for(;;){
            flag1 = true;
            turn = 0;
            while (!flag1 && (turn == 0)){
                Thread.yield();
                // can be omitted
            }
            // Thread 1 inside
            // critical section
            flag1 = false;
        }
    } // can be omitted
}).start();

Hardware Solutions

There is a much simpler solution to critical-section problem:

- **disable interrupts** before entering critical section
- **enable interrupts** upon leaving

Unfortunately

- doesn’t work when multiprocessing
- interrupts can only be disabled in supervisor mode
- interrupts should only be disabled for short periods, but critical sections can be long
Some computers include *Test-and-Set* instruction

- test contents of memory location
- if 0, set it to 1 and return *true*
- otherwise, leave unchanged and return *false*

N.B. above implemented as atomic operation

Critical-section problem is solved by using shared variable *lock*, initially 0, and having threads

- repeatedly test-and-set *lock* until *true* is returned, before entering critical section
- reset *lock* to 0 upon leaving

Other computers include *Exchange* instruction

- contents of main memory location and contents of processor register are swapped

N.B. above implemented as atomic operation

This solves critical-section problem in similar way to Test-and-Set

- repeatedly exchange *lock* with register containing 1, until register contains 0, before entering critical section
Comparing solutions using atomic test-and-set or exchange with software solutions (just using atomic load/store)

- simpler, and so easier to verify
- starvation is possible

All are suitable for multiprocessing, but
- entry to critical section requires busy-waiting (spinlock)
  - wastes processor time
  - high-priority thread might never give up processor!

Operating System Solution

Yet another mechanism, the semaphore, was used by Edsger Dijkstra in the 1960’s to implement THE Operating System

Semaphore \( x \) is supplied with initial non-negative value on creation

- \( \text{wait}(x) \)
  - waits until current value of \( x \) is positive, then
  - decrements current value of \( x \)

- \( \text{signal}(x) \) increments current value of \( x \)
Semaphore lock with initial value 1 enforces mutual exclusion on critical section:
- \texttt{wait(lock)} before entering critical section
- \texttt{signal(lock)} upon leaving

N.B. This is a \textit{binary semaphore}

Another application of binary semaphores is to \textit{synchronize} threads
- one thread waits (on a semaphore) for other to signal (on that semaphore) that it has reached a certain point

N.B. semaphore should be initialized to 0

An application requiring semaphore to take values between 0 and \textit{n} is management of \textit{n} instances of a resource
- semaphore is initialized to \textit{n}
- thread waits on semaphore to request instance
- thread signals on semaphore to release instance
Busy-waiting of uniprocessor can be avoided in implementation of semaphore x

- within kernel, maintain x.value (integer) and x.waiting (queue of threads)
  - x.value takes value supplied to x on creation
  - x.waiting is initially empty

- on wait(x):
  - disable interrupts
  - if x.value == 0, add current thread to x.waiting and to blocked queue of short-term scheduler, enable interrupts and jump to scheduler
  - otherwise, decrement x.value, enable interrupts and return

- on signal(x):
  - disable interrupts
  - if x.waiting is empty, increment x.value
  - otherwise, remove thread from x.waiting and transfer it from blocked to ready queue of short-term scheduler
  - enable interrupts and return

N.B. To avoid starvation, x.waiting should be a FIFO queue

Note also that implementations of wait(x) and signal(x) are (short) critical sections

- replace interrupts by spinlocks for multiprocessing