mutual exclusion

Johan Lukkien
Questions

• Mutual exclusion is apparently an important concern or problem
  – where does it occur, in which contexts?
  – what are solutions and approaches?
  – what are performance concerns?
  – what is the effect of blocking on timing?
Mutual exclusion

• One way of looking at the bridge problem is to regard the bridge as a *shared resource*  
  – acquired by cars from both directions

• We say
  – ‘the bridge is accessed under *mutual exclusion*’
  – or: ‘the entry/exit actions form a *critical section*’

• Programming mutual exclusion means, in fact, turning *sequences of atomic actions* into a single transaction that can be regarded as atomic  
  – using locking/unlocking as, for example, through spinlocks
    • ‘low-level synchronization’
  – or using the concept of a (binary) *semaphore* or *mutex*
Mutual exclusion

• What to protect?
  – resources, to achieve non-interrupted interaction
    • memory locations, computing resources, network

• How to protect?
  – protect the instruction sequences that access the resource
    • admitting just one at any time
    • protecting a (set of) instruction sequence(s) means that everything accessed from that sequence is lumped together as a shared resource
  – choose a reasonable granularity, e.g. in sharing data
    • associate a locking mechanism with data items
    • decide which data to protect together

• Which method of exclusion?
  – ‘high-level’ primitives delivered by the OS
  – ‘low-level’ primitives like spin-locks
Where is low-level synchronization relevant?

- When the overhead for queueing the blocking process and switching to another one is too much
- When such queueing is being implemented

Both cases happen frequently inside the implementation of the kernel
- typically, multiprocessor implementations of shared data structures (scheduler, process management, kernel activities)
- device drivers
- interrupt service routines
  - particularly, nested interrupts

Requirement: locking periods must be very brief and bounded!

- Spinlocks are also available to multithreaded programs
  - particularly, with modern hyper threaded processors or multiple cores
pthread spinlock API

- Declare and initialize a spinlock
- Shared by threads in same process, or via shared memory across processes (pshared)
- Responsibility of programmer to avoid deadlock
  - though `pthread_spin_lock()` returns EDEADLK

```
#include <pthread.h>
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
/* pshared: possibly allocate in shared process space */
int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
/* return EBUSY if busy */
```
Synchronization with spinlocks

• What is needed is to perform two actions together as a single atomic step, as provided with spinlocks

• Example: Test&Set (hardware instructions by the processor):
  – `func Test&Set (lock): <old := lock; lock := true; return old>
  – `<lock := false>

• Use shared variable `lock` (init `false`)

\[ P_X = \textbf{while true do} \]
\[
\text{StateX := 1; } \\
\text{StateX := 2; } \\
\textbf{while Test&Set (lock) do skip od; } \\
\text{StateX := 3; StateX := 4; } \\
\text{lock := false; } \\
\text{StateX := 5 } \\
\textbf{od} \]

\[ P_Y = \textbf{while true do} \]
\[
\text{StateY := 1; } \\
\text{StateY := 2; } \\
\textbf{while Test&Set (lock) do skip od; } \\
\text{StateY := 3; StateY := 4; } \\
\text{lock := false; } \\
\text{StateY := 5 } \\
\textbf{od} \]
Implementation of spinlocks

; Intel syntax

locked: ; The lock variable. 1 = locked, 0 = unlocked.
    dd     0

spin_lock:
    mov    eax, 1 ; Set the EAX register to 1.
    xchg   eax, [locked] ; Atomically swap the EAX register with
                        ; the lock variable.
                        ; This will always store 1 to the lock, leaving
                        ; the previous value in the EAX register.
    test   eax, eax ; Test EAX with itself. Among other things, this will
                    ; set the processor's Zero Flag if EAX is 0.
                    ; If EAX is 0, then the lock was unlocked and
                    ; we just locked it.
                    ; Otherwise, EAX is 1 and we didn't acquire the lock.
    jnz    spin_lock ; Jump back to the MOV instruction if the Zero Flag is
                    ; not set; the lock was previously locked, and so
                    ; we need to spin until it becomes unlocked.
    ret     ; The lock has been acquired, return to the calling
            ; function.

spin_unlock:
    mov    eax, 0 ; Set the EAX register to 0.
    xchg   eax, [locked] ; Atomically swap the EAX register with
                        ; the lock variable.
    ret     ; The lock has been released.

(from wikipedia)

- Is this complete?
- Does it work for
  - 1 processor?
  - multiple processors
- Are there problem scenarios?
Single processor case

• On a single processor, the two processes are executed interleaved; the ‘busy waiting’ makes no sense then (why?).

• The technique for a single processor is to inhibit interrupts for the duration of the critical section.
  – must all interrupts be disabled?

\[
\begin{align*}
P_X &= \textbf{while} \ \text{true} \ \textbf{do} \\
& \quad \text{StateX} := 1; \\
& \quad \text{StateX} := 2; \\
& \quad \text{disable\_interrupts();} \\
& \quad \text{StateX} := 3; \ \text{StateX} := 4; \\
& \quad \text{enable\_interrupts();} \\
& \quad \text{StateX} := 5 \\
& \textbf{od} \\
\end{align*}
\]

\[
\begin{align*}
P_Y &= \textbf{while} \ \text{true} \ \textbf{do} \\
& \quad \text{StateY} := 1; \\
& \quad \text{StateY} := 2; \\
& \quad \text{disable\_interrupts();} \\
& \quad \text{StateY} := 3; \ \text{StateY} := 4; \\
& \quad \text{enable\_interrupts();} \\
& \quad \text{StateY} := 5 \\
& \textbf{od}
\end{align*}
\]
Exclusion with multiple processors

• A processor $P$ requiring exclusion needs to block interference from $P$ itself and from other processors
  – respectively, using interrupt disabling and spinlocks
    • why does interrupt disabling work?
    • in which order does this have to occur?
• The idea with spinlocks is that the common case is that of no contention. Resources are wasted in case of contention. If there is contention,
  – an implementation that continuously assigns a new value to a variable also triggers the cache coherency infrastructure continuously.
  – $P$ is blocked and wasting cycles.
    • be selective in what is being blocked: only those actions that interfere with what the spinlock is protecting
    • could yield the processor temporarily while waiting
• We need predictability in implementations
  – make the implementation *fair and/or prioritized*
• We need to support *nesting*
First approximation: multi-processor

• spinlock (lock):
  \[\text{disable\_interrupts}(); /* note: affects processor that executes this */\]
  \[\text{while Test\&Set(lock) do skip od}\]

• unlock (lock)
  \[lock := false; enable\_interrupts()\]
Second approximation: cache friendly

- spinlock (lock)

  disable_interrupts();
  while Test&Set(lock) do
    while lock do skip od
  od

- unlock (lock)

  lock := false; enable_interrupts()
Third approximation: nesting

- Just enabling interrupts is wrong in case of nesting
  - why?
- Local (recursive) context management is a better structuring
  - requires strict nesting of spinlock() / unlock() for different locks

- spinlock (lock):
  
  \[
  \text{OldContext := disable_interrupts();} \\
  \text{while Test\&Set(lock) do} \\
  \quad \text{while lock do skip od} \\
  \od
  \]

- unlock (lock):
  
  \[
  \text{lock := false; restore_context (OldContext)}
  \]

- (\text{OldContext} is a fresh local variable of the \text{caller})
Fourth approximation: reducing local blocking

- In case of contention, the local processor is blocked. This can be reduced.

- spinlock (lock):
  
  OldContext := disable_interrupts();
  
  while Test&Set(lock) do
    while lock do
      restore_context (OldContext);
      "possibly: pause sometime or suspend";
      OldContext := disable_interrupts();
    od
  od

- unlock (lock):
  
  lock := false; restore_context (OldContext)
Semaphores (Dijkstra)

- Semaphore s is an integer s with initial value $s_0 \geq 0$ and atomic operations $P(s)$ and $V(s)$. The effect of these operations is defined as follows:

  $P(s): < \text{await}(s \geq 0); s := s - 1 >$
  
  $V(s): < s := s + 1 >$

- “< >” denotes again atomicity: the implementation of $P$ and $V$ must guarantee this.
- ‘$\text{await}(s \geq 0)$’ represents blocking until ‘$s > 0$’ holds. This is indivisibly combined with a decrement of s.
- A semaphore is therefore always non-negative.
- Other names for $P$ and $V$: $\text{wait/signal}$, $\text{wait/post}$, $\text{lock/unlock}$.
- Semaphores can be used to implement mutual exclusion.
Example

- Let $s_0 = 1$
- At most one process can ‘pass’ the semaphore
  - two processes passed would mean the semaphore was decreased twice
  - hence, the value would be negative
- This semaphore $s$ can only be 1 or 0 because of the behavior of the program
  - therefore, other terminology is sometimes used, e.g. lock() / unlock()

```plaintext
PX = while true do
    StateX := 1;
    StateX := 2;
    P(s);
    StateX := 3; StateX := 4;
    V(s);
    StateX := 5
od

PY = while true do
    StateY := 1;
    StateY := 2;
    P(s);
    StateY := 3; StateY := 4;
    V(s);
    StateY := 5
od
```

| critical section |  | } | critical section |  | } |
POSIX: mutex (1003.1c)

- Special, two-state (i.e., 1 / 0) semaphore: *mutex*
  - between threads
  - specifically for mutual exclusion
- Restrictions
  - don’t use copies of a mutex in the calls below
  - *lock()* and *unlock()* always by same thread (“ownership”)

```c
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
    /* static initialization, not always possible */
status = pthread_mutex_init (&m, attr); /* attr: NULL: default; should return 0 */
status = pthread_mutex_destroy (&m);  /* should return 0 */
status = pthread_mutex_lock (&m);       /* should return 0 */
status = pthread_mutex_trylock (&m);  /* returns EBUSY if m is locked */
status = pthread_mutex_unlock (&m);   /* should return 0 */
```

$P(m)$

$V(m)$
Exercises

M.1 The implementation given for Spinlock() was not fair. Can you find under which circumstances this unfairness may lead to starvation? Can you make it more fair by adding some more state information?

M.2 Instead of Test&Set(), other atomic operations are sometimes provided. Examples are:

- **Fetch&Add (x):** \(<\text{prev} := x; x := x+1; \text{return}(\text{prev})>\>
- **Swap(x,y):** \(<t := x; x := y; y := t>\>

Make implementations of Spinlock() and Unlock() using these primitives.

M.3 The order in the implementation of Spinlock() was to first disable interrupts and then acquire the lock. Could it be the other way round? Explain why, or why not. Is the order in Unlock() relevant?
Exercises

**M.4** Semaphore or Mutex implementations need a data structure to record the semaphore state, including a list of waiters. Give a pseudo code for such an implementation and explain where exclusion is required. Which exclusion method should be used there?

**M.5** Suppose we do not disable interrupts in the `Spinlock()` implementation. Explain potential problems.

**M.6** Suppose a process (or thread, interrupt routine) acquires two spinlocks, and other processes do the same. Show how this can lead to a deadlock. Propose a solution to this problem.
Reducing blocking, deadlock; increasing fairness

• A complete disabling of interrupts is quite heavy and often not necessary.
  – Instead, an interrupt level is used which is part of the current processor state. Only tasks/interrupts higher than this level may interrupt the current execution.
  – For this to work, these higher interrupts must not access the spinlock or the protected resources.

• In order to avoid deadlock, locks must be required in fixed order. This can be achieved, for example, by a serial ID.

• Fairness can be achieved using Lamports’ ticket algorithm. This is straightforward using Fetch&Add as a primitive but can be implemented using Test&Set as well. Fair spinlocks are also called queued spinlocks.
Priority

• Scheduling of threads is preemptive, and priority-based

• *Priority*: an intermediate between application and OS
  – the selected thread is the one with the highest priority
  – does not always reflect importance, but is used to control *schedulability*

• Separation of concerns: policy and mechanism
  – the programmer, or the OS, assigns priority
    • they do this according to a certain *policy*, they know what is best
  – the OS knows how to implement priority based scheduling
    • the *mechanism*
Mars Rover Pathfinder

- July 4, 1997, landing on Mars
- After a few days into the mission random resets occurred
  - after start of gathering meteorological data
- Attributed to ‘software glitches’ or ‘system overload’
- OS: VxWorks

- Cause: shared resource: information bus
  - high priority task: moves important information
  - low priority meteorological data gathering task: gets interrupted by a middle priority, unrelated task
  - a watchdog reset is triggered upon a long delay of the high priority task
  - .... priority inversion
  - .... forgot to declare the correspondent semaphore as ‘priority ceiling’ (or inheritance)
Priority inversion

- A low priority job obtains a resource; a high priority job waits on it.

- A middle priority job pre-empts the low priority job:
  - the high priority job now waits on the middle priority job
  - ... and executes effectively at the low priority

- Unbounded inversion
Concurrent control protocols

- Concurrency control protocols manage concurrent and competitive access to resources
  - they manipulate priority for this purpose
  - but may also put restrictions

- A (concurrency control) policy should
  - at least bound the inversion time
  - adhere to the job priorities as good as possible
Priority inheritance protocol

- The priority of a job $P$ is *dynamically adjusted* to be the maximum of
  - the priority of any job that is blocked on the allocated resources of $P$
  - (... and its own priority)
  - this adjustment is done *transitively*, i.e., if the priority of a waiter becomes adjusted then this adjustment is forwarded

- ...middle priority jobs will wait now.
Sources and types of blocking

• Direct blocking
  – another job holds the resource

• Push-through blocking
  – job A suffers push-through blocking if a lower priority job changes priority temporarily to a higher one than A
    • due to a specific concurrency control protocol

• Chained blocking (transitive blocking)
  – sequence of blockings
    • job A blocks on (resource R0 held by) job B
    • job B blocks on (resource R1 held by) job C
Chained blocking

FIGURE 8-9 A worst-case blocking scenario for priority-inheritance protocol.
Sources and types of blocking

• Direct blocking
  – another job holds the resource

• Push-through blocking
  – job A suffers push-through blocking if a lower priority job changes priority temporarily to a higher one than A
    • due to a specific concurrency control protocol

• Chained blocking (transitive blocking)
  – sequence of blockings
    • job A blocks on (resource R0 held by) job B
    • job B blocks on (resource R1 held by) job C

• Deadlock:
  – circular waiting (no chain but circle), greedy consumers
  – blocking critical sections

• Avoidance blocking
  – blocking according to a strategy to avoid some of the above
(Immediate) Priority Ceiling

- Each resource (mutex, spinlock) has a *ceiling* which is the maximum priority of any job ever using it

- **Rule:**
  - the dynamic priority of a job is the maximum ceiling of any resource it has acquired
    - hence, adjusted upon `Lock()` and `Unlock()`

- **Properties, on a single processor (!):**
  - this simply means that a job using a resource cannot be pre-empted by any other job that uses the same resource
    - no need to represent a queue of waiters
    - while executing, the resources of a job are free (mimics an ‘interrupt level’)
  - avoids cyclic deadlock
  - avoids chained blocking
  - wait on at most one lower priority job
POSIX: mutex (1003.1c-2008)

Defining the attribute
• the **type** (deadlocking, deadlock-detecting, recursive, etc).
• the **robustness** (what happens when the owner dies)
• the **process-shared attribute** (for sharing a mutex across process boundaries).
• the **protocol** (concurrency control).
• the **priority ceiling** (max priority of a user of \( m \) – new dynamic priority of caller).

```c
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
/* static initialization, not always possible */
status = pthread_mutex_init (&m, attr); /* attr: NULL: default; should return 0 */
status = pthread_mutex_destroy (&m);  /* should return 0 */
status = pthread_mutex_lock (&m);       /* should return 0 */
status = pthread_mutex_trylock (&m);  /* returns EBUSY if m is locked */
status = pthread_mutex_unlock (&m);   /* should return 0 */
```
M.7 Consider 4 repetitive tasks, A, B, C and D sharing two resources, r1 and r2 protected by mutexes m1 and m2. Jobs of the 4 tasks are as follows.

- A: d; Lock(m1); d; Lock(m2); d; Unlock(m1); d; Unlock(m2)
- B: d; Lock(m2); d; Lock(m1); d; Unlock(m1); d; Unlock(m2)
- C: d; Lock(m1); d; Unlock(m1)
- D: d; Lock(m2); d; Unlock(m2)

Here, d refers to a variable delay representing some idleness or computing (not really relevant). Priority of A=4, B=3, C=2, D=1 (4 is highest).

Draw Gantt charts in case of no concurrency control, priority inheritance and priority ceiling. Show occurrence of inversion, chained blocking and deadlock and show the dynamic priority.