Operating Systems (234123)

Synchronization

Dan Tsafrir (11/5/2015)
Partially based on slides by Hagit Attiya
Context

• A set of threads (or processes) utilize shared resource(s) simultaneously
  – For example, threads share the memory
  – Processes can also share memory (using the right system calls)
  – In this talk we will use the terms thread/process interchangeably

• Need to synchronize also in uni-core setup, not just multicore
  – A user-land thread can always be preempted in favor of another thread
  – Interrupts might occur, triggering a different context of execution

• => Need to learn about synchronization
  – This is true for any parallel code that shares resources
  – An OS kernel happens to be such a code (many believe students should learn about parallelism from the very first course...)
Example: withdraw money from bank

• Assume
  – This is the code that runs whenever we make a withdrawal
  ```c
  int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount
    put_balance( account, balance )
    return balance
  }
  ```
  – Account holds $50K
  – Account has two owners
  – Both owners make a withdrawal simultaneously from two ATMs
    1. One takes out $30K
    2. The other takes out $20K
  – Every op is done by a different thread on a different core
Example: withdraw money from bank

- **Best case scenario from bank’s perspective**
  - $0 in account

```c
int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount // $50K - $30K = $20K
    put_balance( account, balance )
    return balance // $20K
}
```

```c
int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount // $20K - $20K = $0K
    put_balance( account, balance )
    return balance // $0K
}
```
Example: withdraw money from bank

- A better scenario from owners’ perspective
  - $20K in account...

- We say that the program suffers from a “race condition”
  - Outcome is nondeterministic and depends on the timing of uncontrollable, unsynchronized events

```c
int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount  // $50K - $30K = $20K
}
```

```c
int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount  // $50K - $20K = $30K
    put_balance( account, balance )
    return balance     // $30K
}
```

```c
int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount  // $50K - $30K = $20K
    put_balance( account, balance /*$20K*/ )
    return balance     // $20K !!!
}
```
Example: too much milk

<table>
<thead>
<tr>
<th>time</th>
<th>Ninet</th>
<th>Yuda</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:00</td>
<td>checks fridge</td>
<td></td>
</tr>
<tr>
<td>15:05</td>
<td>goes to supermarket</td>
<td>checks fridge</td>
</tr>
<tr>
<td>15:10</td>
<td>buys milk</td>
<td>goes to supermarket</td>
</tr>
<tr>
<td>15:15</td>
<td>gets back home</td>
<td>buys milk</td>
</tr>
<tr>
<td>15:20</td>
<td>puts milk in fridge</td>
<td>gets back home</td>
</tr>
<tr>
<td>15:25</td>
<td></td>
<td>puts milk in fridge</td>
</tr>
</tbody>
</table>
Milk problem: solution #1?

• Leave note on fridge before going to supermarket
  – Probably works for humans
  – But for threads...

```c
if( no milk )
  if( no note )
    leave note
    buy milk
    remove note
```
Milk problem: solution #1 😞

- Too much milk, again...

```
if( no milk )
  if( no note )
  leave note
  buy milk
  remove note
```

```
if( no milk )
  if( no note )
  leave note
  buy milk
  remove note
```
Milk problem: solution #2?

• Leave & check note before checking fridge!

thread A:
leave note A
if( ! note B )
  if( no milk )
    buy milk
remove note A

thread B:
leave note B
if( ! note A )
  if( no milk )
    buy milk
remove note B
Milk problem: solution #2 😞

• Leave & check note before checking fridge!

thread A:
leave note A

if( ! note B ) // there is!
  if( no milk )

buy milk
remove note A

thread B:
leave note B

if( ! note A ) // there is!
  if( no milk )

if( ! note A ) // there is!
  if( no milk )

buy milk
remove note B

• => No milk...
Milk problem: solution #3?

**thread A** (change ‘if’ to ‘while’)

- leave note A
- **while** (note B) do NOP
- if( no milk )
  - buy milk
- remove note A

**thread B** (same as before)

- leave note B
- if( ! note A )
  - if( no milk )
    - buy milk
- remove note B

- **In the past, they would have told you**
  - That this solution works! 😊
  - But that it’s
    - Asymmetric (complicates things)
    - Unfair (“A” works harder: if arrive together, “A” will buy the milk)
    - Works for only two threads (what if there are more?)
Milk problem: solution #3?

• In the past, they would have told you – That this solution works! 
  – But that it’s asymmetric (complicates things)
  – Unfair (“A” works harder: if arrive together, “A” will buy the milk)
  – Works for only two threads (what if there are more?)

thread A (change ‘if’ to ‘while’)
leave note A
while( note B ) do NOP
if( no milk )
  buy milk
remove note A

But
it doesn’t work

thread B
leave note B
if( ! note A )
  if( no milk )
    buy milk
  remove note B

OS (234123) - synchronization
Memory coherency & consistency

MAMAS IN A NUTSHELL PART #2
Memory Trade-Offs

- Large (dense) memories are slow
- Fast memories are small, expensive and consume high power
- Goal: give the processor a feeling that it has a memory which is large (dense), fast, consumes low power, and cheap
- Solution: a Hierarchy of memories

![Memory Hierarchy Diagram]

<table>
<thead>
<tr>
<th>Speed</th>
<th>Fastest</th>
<th>Slowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Smallest</td>
<td>Biggest</td>
</tr>
<tr>
<td>Cost</td>
<td>Highest</td>
<td>Lowest</td>
</tr>
<tr>
<td>Power</td>
<td>Highest</td>
<td>Lowest</td>
</tr>
</tbody>
</table>
## Typical latency in memory hierarchy

<table>
<thead>
<tr>
<th>Memory level</th>
<th>Size</th>
<th>Latency (roughly, not accurate numbers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU registers</td>
<td>≈ 100s of bytes</td>
<td>≈ 0.5ns</td>
</tr>
<tr>
<td>L1 cache</td>
<td>≈ 64KB</td>
<td>≈ 1ns–3ns</td>
</tr>
<tr>
<td>L2 cache</td>
<td>≈ 1MB–8MB</td>
<td>≈ 10ns–20ns</td>
</tr>
<tr>
<td>Main memory (DRAM)</td>
<td>≈ 1GB–32GB</td>
<td>≈ 100ns–200ns</td>
</tr>
<tr>
<td>SSD</td>
<td>256GB</td>
<td>read ≈ 25us, write ≈ 250us ... 2ms</td>
</tr>
<tr>
<td>Hard disk (SATA)</td>
<td>≈ 1TB–4TB</td>
<td>≈ 5 ms</td>
</tr>
</tbody>
</table>
Cores typically have private caches (Nehalem, i7)
The cache “coherency” problem – relates to a single memory location

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache contents for CPU-1</th>
<th>Cache contents for CPU-2</th>
<th>Memory contents for location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU-1 reads X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU-2 reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU-1 stores 0 into X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Stale value, different than corresponding memory location and CPU-1 cache. (The next read by CPU-2 might yield “1”.) A “coherent memory” doesn’t allow such stale reads.
Consistency – relates to different locations

• Assume
  – Locations A & B are cached by C1 & C2

• If store operations are
  1. Immediately seen by other cores
  2. Cannot be reordered with load ops

• Then it’s impossible for both “if” conditions to be true
  – Reaching the “if” statements means either A or B must hold

• On modern HW, however, both items can be violated due to performance considerations
  1. Stored values can be temporarily stored in a local “store buffer”, temporarily remaining invisible to other cores
  2. Cores can reorder load & store ops if there are no (apparent) dependencies between them

• Should this be allowed?
  – Determined by the memory consistency model

\[
\begin{array}{|c|c|}
\hline
\text{Core C1} & \text{Core C2} \\
\hline
A = 0; // init & B = 0; // init \\
\hline
... & ... \\
\hline
A = 1; & B = 1; \\
\hline
\text{if ( B == 0 ) ...} & \text{if ( A == 0 ) ...} \\
\hline
\end{array}
\]
## Consistency models

TSO = total store order model

<table>
<thead>
<tr>
<th>Type</th>
<th>Alpha</th>
<th>ARMv7</th>
<th>PA-RISC</th>
<th>POWER</th>
<th>SPARC RMO</th>
<th>SPARC PSO</th>
<th>SPARC TSO</th>
<th>x86</th>
<th>x86 oostore</th>
<th>AMD64</th>
<th>IA-64</th>
<th>zSeries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Loads reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Stores reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Stores reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Atomic reordered with loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Atomic reordered with stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Dependent loads reordered</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Consistency – how to enforce ordering

- On x86, a memory fence would eliminate the problem
  - Makes all store ops before the fence visible to all the load ops after the fence

<table>
<thead>
<tr>
<th>Processor P1</th>
<th>Processor P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = 0; // \text{init} )</td>
<td>( B = 0; // \text{init} )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( A = 1; )</td>
<td>( B = 1; )</td>
</tr>
<tr>
<td>( \text{mfence;} )</td>
<td>( \text{mfence;} )</td>
</tr>
<tr>
<td>if (( B == 0 )) \ldots</td>
<td>if (( A == 0 )) \ldots</td>
</tr>
</tbody>
</table>
BACK TO SYNCHRONIZATION
Milk problem: solution #3 😞

**thread A** (change ‘if’ to ‘while’)
leave note A
while( note B ) do NOP
if( no milk )
  buy milk
remove note A

**thread B** (same as before)
leave note B
if( ! note A )
  if( no milk )
    buy milk
remove note B

• For some consistency models, thread A might read stale value of note B
  – And vice versa
  – Meaning, once again, both would purchase milk

• The solution could be fixed on machines we know & love:
  – With the help of explicit “memory barriers” (AKA “fences’), placed after the “leave note” line (responsible for memory consistency)
Towards a solution: critical sections

• The heart of the problem
  – Unsynchronized access to data structure
    • In our examples, this was simple global vars ("note A", "account")
    • But could be linked lists, or hash table, or a composition of various data structures
      • (Regardless of whether it’s the OS, or a user-level parallel job)
  – Doing only part of the work before another thread interferes
    • If only we could do it atomically…

• The operation we need to do atomically: “critical section”
  – Atomicity of the critical section would make sure
    • Other threads don’t see partial results
  – The critical section could be the same code across all threads
    • But could also be different
Towards a solution: critical sections

• Example for the same code across threads
  –
  ```c
  int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount
    put_balance( account, balance )
    return balance
  }
  ```

• Example for different code
  – If one thread increments and the other decrements a shared variable
  – Insertion/deletion of an element to/from a linked list
Towards a solution: requirements

- **Mutual exclusion ("mutex")**
  - To achieve atomicity
    - Threads execute an entire critical section one at a time
    - Never simultaneously
  - Thus, a critical section is a "**serialization point**"

- **Progress**
  - At least one thread gets to do the critical section at any time
  - No "**deadlock**"
    - Deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus no one ever finishes
  - No "**livelock**"
    - Same as deadlock, except state changes
    - E.g., 2 people in narrow corridor, trying to be polite by moving aside to let the other pass, ending up swaying from side to side
Towards a solution: optional requirements

• Fairness, which means...
  – No starvation, namely, a thread that wants to do the critical section, would eventually succeed
  – Nice to have: bounded waiting
  – Nice to have: FIFO
Solution: locks

• Abstraction that supports two operations
  – acquire(lock)
  – release(lock)

• Semantics
  – Only one thread can acquire at any give time
  – Other simultaneous attempts to acquire are forced to wait
    • Until the lock is released, at which point another thread will get it
  – Thus, at most one thread holds a lock at any given time

• Therefore, if a lock “protects” a critical section, meaning
  – Lock is acquired at the beginning of the critical section
  – Lock is released at the end of the critical section

• Then atomicity is guaranteed
Solution: locks

```c
int withdraw( account, amount ) {  
    acquire( account->lock )  
    balance = get_balance( account )  
    balance -= amount  
    put_balance( account, balance )  
    release( account->lock )  
    return balance  
}
```
Solution: locks

- 2 threads make a withdrawal
- What happens when the pink tries to acquire?
- Is it okay to return outside the critical section?
  - Depends
  - Yes, if you want the balance at time of withdraw, and you don’t care if it changed since
  - Otherwise, need to acquire lock outside of withdraw(), rather than inside

```c
int withdraw( account, amount ) {
    acquire( account->lock )
    balance = get_balance( account )
    balance -= amount
    return balance
}
```

```c
int withdraw( account, amount ) {
    acquire( account->lock )
    put_balance( account, balance )
    release( account->lock )
    balance = get_balance( account )
    balance -= amount
    put_balance( account, balance )
    release( account->lock )
    return balance
}
```

```c
put_balance( account, balance )
release( account->lock )
```
Implementing locks

• **When you try to implement a lock**
  – You quickly find out that it involves a critical section...
  – Recursive problem

• **There are 2 ways to overcome the problem**
  1. Using SW only, no HW support
     • Possible, but complex, error prone, wasteful, and nowadays completely irrelevant, because...
  2. Using HW support (*all* contemporary HW provides such support)
     • There are special ops that ensure mutual exclusions (see below)
     • Fairness aspects aren’t ensured
       – That’s typically not a problem within the kernel, because
         » Critical sections are extremely short
         » Or else we ensure fairness
spinlocks

FINE-GRAINED SYNCHRONIZATION
Possible kernel spinlock with x86’s xchg

```c
struct spinlock {
    uint locked; // is the lock held? (0|1)
};

inline uint
xchg(volatile uint *addr, uint newval) {
    uint result;
    asm volatile("lock; xchg %0, %1") :
        "+m" (*addr), "+a" (result) :
        "1" (newval) :
        "cc");
    return result;
}

(The ‘while’ is called “spinning”)

void
acquire(struct spinlock *lk) {
    disableInterrupts(); //kernel/this core
    while( xchg(&lk->locked, 1) != 0 )
        // xchg() is atomic, and the “lock”
        // adds a “fence” (so read/write ops
        // after acquire aren’t reordered
        // before it)
    ;
}

void
release(struct spinlock *lk) {
    xchg(&lk->locked, 0);
    enableInterrupts(); //kernel; this core
}
```

OS (234123) - synchronization
Kernel spinlock issues

• In our implementation, interrupts are disabled
  – While the lock is held
  – Including while the kernel is spinning
  – If we want to allow interrupts while spinning...

• Why?
  – Responsiveness
  – If some other thread of execution holds the lock
  – Although kernel threads don’t go to sleep with locks, and they hold locks for very short periods of time

```c
void acquire(struct spinlock *lk) {
    disable_interrupts();
    while( xchg(&lk->locked, 1) != 0 ) {
        enable_interrupts();
        disable_interrupts();
    }
}
```
Kernel spinlock issues

• **In the kernel, on a uni-core, do we need to lock?**
  – Yes, if interrupts access the same data structures (“locking” then also means disabling interrupts as in the previous example)

• **On a multicore, do we care if *other* cores take interrupts?**
  – No, as if they want the lock, they will spin regardless
  – (In particular, within the interrupt handler)

• **User space can’t disable interrupts, but...**
  – Is there something equivalent to interrupts that we need to address when considering synchronization?
    • Signals (we may want to block them for a while if the access the same data structures as the regular program)
  – Can we make sure we spin until we get the lock?
    • No, the OS might preempt us
    • Though that’s oftentimes fine and acceptable
Spinlock issues

- Other atomic ops to implement spinlocks
  - Compare-and-swap (“CAS”)
  - Test-and-set
Semaphores, conditional variables, monitors

COARSE-GRAINED SYNCHRONIZATION: BLOCK (SLEEP) WHILE WAITING
Spin or block? (applies to kernel & user)

• **Two basic alternatives**
  – Spin (busy waiting)
    • Might waste too many cycles
  – Block (go to sleep) until the event you wait for has occurred
    • Free up the CPU for other useful work
  – OS offers blocking as a service; it doesn’t offer spinning. If users want to spin, they can do it on their own, they don’t need the kernel for it

• **When to spin?**
  – Rule of thumb:
    • Spin if we’re going to get the lock very soon, much sooner than a context switch takes

• **When to block?**
  – Rule of thumb:
    • Don’t spin if we’re going to get the lock much later than the duration of a context switch
Spin or block? (applies to kernel & user)

• Consider the following parallel program canonical structure
  – for(i=0; i<N; i++) {
    compute(); // duration is C cycles
    sync(); // takes S cycles; spin? block?
  }

• Runtime is
  – \( N \times (C + S) = N \times C + N \times S \)
  – \( N \times C \) is a given; as an OS, we can’t really do anything about it
  – But with S, we may have a choice...
  – What happens if \( C \ll S \)?
    • Sync time dominates
  – If we have a fairly good idea that spinning would end much sooner than a context switch => should spin, or else runtime would explode
  – This is how it’s typically done within kernels (spinlocks are used to protect very short critical sections)
Coarse-grained synchronization

• Waiting for a relatively long time
  – Better relinquish the CPU:
    • Leave runnable queue
    • Move to sleep queue, waiting for an event

• The OS provides several services that involve sleeping
  – All the system calls that are blocking: read, write, select, setitimer, sigsuspend, pipe, sem*, ...

• Unlike spinning, user-land can’t sleep on its own
  – It must request the OS to sleep, since sleeping involves changing process states
SEMAPHORES
Semaphore – concept

• Proposed by Dijkstra (1968)
• Allows tasks to
  – Coordinate the use of (several instances of) a resource
• Use “flag” to announce
  – “I’m waiting for a resource”, or
  – “Resource has just become available”
• A party who announces it is waiting for a resource
  – Will get the resource if its available, or
  – Will go to sleep if the resource isn’t available
  – In which case it’ll be awakened when the resource becomes available
Semaphore – fields

• Value (integer)
  – Nonnegative
    => counting the number of “resources” currently available
  – Negative
    => counting the number of tasks waiting for a resource
  – (Though in some implementations value is always nonnegative)

• A queue of waiting tasks
  – Waiting for the resource to become available
  – When ‘value’ is allowed to be negative:
    • \(|value| = \text{queue.length}\)
Semaphore – interface

- **Wait**(semaphore)
  - value -= 1
  - If( value >= 0 ) // it was >= 1 before we decreased it
    - Task can continue to run (it has been assigned the resource)
  - Else
    - Place task in waiting queue
  - A.k.a. P() or proben
Semaphore – interface

- **Signal(semaphore)**
  - value += 1
  - If( value <= 0 ) // it was <= -1 before we increased it
    - Remove one of the tasks from the wait-queue
    - Wake it (make it runnable)

- A.k.a. V() or verhogen
Semaphore – POSIX system calls

• A semaphore is a kernel object manipulated through
  – semctl(), semget(), semop(), sem_close(), sem_destroy(),
    sem_getvalue(), sem_init(), sem_open(), sem_post(), sem_unlink(),
    sem_wait(),

• More info
  – http://pubs.opengroup.org/onlinepubs/009695399/functions/semop.html
Semaphore vs. spinlock

• **Typical granularity**
  – Coarse (semaphore) vs. fine (spinlock)

• **Spinlock is “more primitive”**
  – Spinlock usually used to implement a semaphore

• **If the maximal ‘value’ of a semaphore is 1**
  – Then the semaphore is conceptually similar to a spinlock
  – (Sometime called a “binary semaphore”)

• **But if the maximal ‘value’ is bigger than 1**
  – Counting a resource
  – Don’t have to “wait” in order to “signal”
Producer-consumer ring

USING SEMAPHORES
Producer/consumer – problem

• Problem
  – Two threads share an address space
  – The “producer” produces elements
    • (E.g., decodes video frames to be displayed (element = frame))
  – The “consumer” consumes elements
    • (E.g., displays the decoded frames on screen)

• Typically implemented using
  – “Cyclic buffer” (indexed modulo n) a.k.a. “ring buffer”
Producer/consumer – faulty solution

int c = 0;  // shared variable

**producer:**
while( 1 )
    wait until (c < n);
    buf[pp] = new item;
    pp = (pp+1) mod n;
    c += 1;

**consumer:**
while( 1 )
    wait until (c >= 1);
    consume buf[cp];
    cp = (cp+1) mod n;
    c -= 1;

- **Might access ‘c’ simultaneously**
  - The “+=“ and “-=“ operations aren’t necessarily atomic

- **Busy waiting**
  - Which might be suboptimal
Producer/consumer – semaphore solution

Could be perceived as less “structured” than locks
   – “Locking” thread isn’t the one that does the “releasing”

Works with only one consumer/producer pair
   – Q: what if there are multiple?
   – A: serialize access to pp and cp
      • E.g., spinlock right after ‘wait’ and unlock right before ‘signal’

```
semaphore_t free_space( n );
semaphore_t avail_items( 0 );

producer:
    while( 1 )
        wait( free_space );
        buf[pp] = new item;
        pp = (pp+1) mod n;
        signal( avail_items );

consumer:
    while( 1 )
        wait( avail_items );
        consume( buf[cp] );
        cp = (cp+1) mod n;
        signal( free_space );
```
USING SEMAPHORES
Concurrent readers, exclusive writer (CREW)

• Multiple tasks want to read/write the same data element

• For consistency, need to enforce the following rules
  – No problem for several tasks to read simultaneously
  – But when a task is writing, no other task is allowed to read or write

• Table denoting if multiple access is allowed

<table>
<thead>
<tr>
<th></th>
<th>reader</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>reader</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>writer</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
• **Doesn’t work**
  – No mutual exclusions between readers and writers
  – Only one reader at a time

```c
int r = 0;
semaphore_t sRead ( 1 ); // number of concurrent readers
semaphore_t sWrite ( 1 ); // defends (serializes) ‘r’
// writers’ mutual exclusion

writer:
    wait( sWrite )
    [write]
    signal( sWrite )

reader:
    wait( sRead )
    [read]
    signal( sRead )
```
int r = 0; // number of concurrent readers
semaphore_t sRead ( 1 ); // defends (serializes) ‘r’
semaphore_t sWrite ( 1 ); // writers’ mutual exclusion

writer:
wait( sWrite )
[write]
signal( sWrite )

reader:
wait( sWrite )
wait( sRead )
[read]
signal( sRead )
signal( sWrite )

• Doesn’t work
  – Only one reader at a time
**CREW 😊**

- **Idea**
  - Only 1st reader waits for write semaphore
- **Works**
  - But might starve writers...
  - Think about how to fix

```c
int r = 0; // number of concurrent readers
semaphore_t sRead ( 1 ); // defends (serializes) ‘r’
semaphore_t sWrite ( 1 ); // writers’ mutual exclusion

**writer:**
wait( sWrite )
[write]
signal( sWrite )

**reader:**

```
```
How to implement Semaphores 😞

```c
struct semaphore_t {  int value;   wait_queue_t wq; };  

void wait(semaphore_t *s) {
    s->value -= 1;
    if (s->value < 0) {
        enqueue( self, &s->wq );
        block( self );
    }
}

void signal(semaphore_t *s) {
    s->value += 1;
    if (s->value <= 0) {
        p = dequeue( &s->wq );
        wakeup( p );
    }
}
```

- Doesn’t work
  - The semaphore_t fields (value and wq) are accessed concurrently
  - (For starters...)
How to implement Semaphores 😞

```c
struct semaphore_t { int value; wait_queue_t wq; lock_t l; };

void wait(semaphore_t *s) {
    lock( &s->l );
    s->value -= 1;
    if( s->value < 0 ) {
        enqueue( self, &s->wq );
        unlock( &s->l );
        block( self );
    }
    else
        unlock( &s->l );
}

void signal(semaphore_t *s) {
    lock( &s->l );
    s->value += 1;
    if( s->value <= 0 ) {
        p = dequeue( &s->wq );
        wakeup( p );
    }
    unlock( &s->l );
}
```

- Doesn’t work
  - The well-known “problem of lost wakeup” (make sure you can find it)
Semaphore – implementation 😊

• In the kernel
  – There’s an interaction between sleep and spinlock and context switch
  – A task is able to go to sleep (block) holding a lock...
  – ...And wake up holding that lock
  – The kernel does the magic of making it happen without deadlocking
    the system (freeing the lock in between)
  – To see how it’s really done:

236376 – Operating Systems Engineering (OSE)
http://webcourse.cs.technion.ac.il/236376
Build an operating system from “scratch”
(minimalistic, yet fully functional)

• Note that the semaphore does busy-waiting
  – But it has a very short critical section
Amdahl’s law

- **What’s the maximal expected speed when parallelizing?**
  - Let $n$ be the number of threads
  - Let $s$ be the fraction of the algorithm that is strictly serial ($0 \leq s \leq 1$)
  - Let $T_n$ be the time it takes to run the algorithm with $n$ threads
  - Then, optimally,
    \[
    T_n = T_1 \times \left( s + \frac{1-s}{n} \right) \geq T_1 \times s
    \]
  - And
    \[
    \text{speedup} = \frac{T_1}{T_n} = \frac{1}{s + \frac{1-s}{n}} \leq \frac{1}{s}
    \]
Towards a solution: optional requirements

• Side note: terminology (overloaded, not directly related)
  – “Lock-free” algorithm
    • No critical sections in the algorithm
    • There’s progress (as defined above); starvation is possible
  – “Wait free” algorithm
    • Per-thread progress is ensured; no starvation
  – Thus, every wait-free algorithm is a lock-free algorithm
  – For more details, see
    http://en.wikipedia.org/wiki/Non-blocking_algorithm
CONDITION VARIABLES

OS (234123) - synchronization
כשלון צפיפות Мари

**wait(cond, &lock)**
- שחרר את המנעול (希יב להחזיק בו).
- המתן לפעולה `signal`.
- המתן למנעול (כשחזר המחזיק במנעול).

**signal(cond)**
- הער את אחד הממתינים ל(cond), אשר עבר להמתין למנועל.
- וחזר להמתין אם אין ממתינים.

**broadcast(cond)**
- הער את כל התהליכים הממתינים.
- עוברים להמתין למנعزل.
- וחזר להמתין אם אין ממתינים.
משתני תקני כרזה:

- כאשר תהליך מקבל `signal` הוא לא ייאלץ את המנעול באוטומטי, עודין צורי לחכות להשהיגה.
  - `mesa-style` מפריעה

- בניגוד לספורים, `signal` לא זוכר היסטורי.
  - `signal(cond)` מחליק לא יידבץ אם או יידבץ על `cond`.
lock QLock;
condition notEmpty;

Enqueue (item):
lock_acquire( QLock)
put item on queue
signal(notEmpty)
lock_release( QLock)

Dequeue (item):
lock_acquire( QLock)
while queue empty
  wait(notEmpty, &QLock)
remove item from queue
lock_release( QLock)

 لماذا צירך?
producer:
lock_acquire(bLock);
while (buffer is full)
    wait(not_full,&bLock);
add item to buffer;
signal(not_empty);
lock_release(bLock);

consumer:
lock_acquire(bLock);
while (buffer is empty)
    wait(not_empty,&bLock);
get item from buffer
signal(not_full);
lock_release(bLock);
משתני תıyor

اعدة מיוכרי של שיקול תコーות
[C.A.R. Hoare, 1974]

▲ אובייקט (בمعنى של שיפוט-תكنולוגיה, object-oriented)
הגישה לאובייקט מקנה שליטה במנוע עם פרוצדורת אתחול וגיישה. 
הגישה לאובייקט(Mock Shedule bằngעל (באופק לא מפורש)
〆 שליחת signal מנוגרת את המנוע המעבר את השליטה ב
 señal למקבל ה señal

נתוך בכמה שפוא-תكنולוגיה מודרנית (뽀ארש, Java).

OS (234123) - synchronization
67
POSIX synchronization

Objects for synchronization
- Located in shared memory.
- Available to threads of different processes.

(mutex locks) (pthread_mutex_XXX) mutex locks
- Creation, destruction: init, destroy
- Lock, unlock, trylock

(condition variables) (pthread_cond_XXX) condition variables
- Creation, destruction: init, destroy
- Wait, signal, broadcast