Cooperation models
Global vs. local model

- Global environment
  - Processes (threads) can share data
  - Shared memory

- Local environment
  - Processes/ do not share data
  - No shared memory
Global environment

Proc. P1

Stack1

Data 1

Text 1

Proc. P2

Stack2

Data 2

Text 2

Shared data

Shared text
Global environment

O1, O4 private objects
O2, O3 shared objects
• competition, cooperation
Local environment

Proc. P1

Stack1

Data 1

Text 1

Proc. P2

Stack2

Data 2

Text 2

Shared text
Local environment

O1-O5 are private objects
Competition through server processes
Cooperation through communication
Local environment

Cooperation (communication, synchronization) by means of message passing
Synchronization

Global environment
Synchronization Motivation

Thread 1

p = someFn();

isInitialized = true;

Thread 2

while (! isInitialized ) ;

q = aFn(p);

if q != aFn(someFn())
    panic
# Too Much Milk Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:35</td>
<td>Leave for store.</td>
<td></td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, put milk away.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td>Buy milk.</td>
</tr>
<tr>
<td>1:00</td>
<td>Arrive home, put milk away.</td>
<td>Oh no!</td>
</tr>
</tbody>
</table>
Definitions

• **Race condition (corsa critica):** output of a concurrent program depends on the order of operations between threads

• **Mutual exclusion (mutual esclusione):** only one thread does a particular thing at a time

• **Critical section (sezione critica):** piece of code that only one thread can execute at once

• **Lock:** prevent someone from doing something
  – Lock before entering critical section, before accessing shared data
  – unlock when leaving, after done accessing shared data
  – wait if locked (all synch involves waiting!)
Too Much Milk, Try #1

• Correctness property
  – Someone buys if needed (liveness)
  – At most one person buys (safety)

• Try #1: leave a note
  
  if !note
    if !milk {
      leave note
      buy milk
      remove note
    }
Too Much Milk, Try #1

Thread A

if (!note) {
    if (!milk)
        leave note
    buy milk
    remove note
}

Thread B

if (!note){
    if (!milk)
        leave note
    buy milk
    remove note
}
Too Much Milk, Try #2

Thread A

leave note A
if (!note B) {
  if (!milk)
    buy milk
}
remove note A

Thread B

leave note B
if (!noteA){
  if (!milk)
    buy milk
}
remove note B
Too Much Milk, Try #3

Thread A
leave note A
while (note B) // X
do nothing;
if (!milk)
  buy milk;
remove note A

Thread B
leave note B
if (!noteA){   // Y
  if (!milk)
    buy milk
}remove note B

Can guarantee at X and Y that either:
(i) Safe for me to buy
(ii) Other will buy, ok to quit
Lessons

• Solution is complicated
  – “obvious” code often has bugs

• Modern compilers/architectures reorder instructions
  – Making reasoning even more difficult

• Generalizing to many threads/processors
  – Peterson’s algorithm: even more complex
Another example: a stack

- Two threads interact by means of a stack, with push and pop operations
- Expected behaviour:

```
T_1
............
Top--
Stack[top]=y
............

T_2
............
Z=Stack[top]
Top++
............
```

Critical sections
Another example: a stack

• Here the operations of $T_1$ are interrupted by a pop executed by $T_2$
• Possible behaviour:

\[ \begin{array}{c}
T_{1} \\
\text{.................} \\
\text{Top--} \\
\text{Stack[top]=y} \\
\text{.................} \\
\end{array}\]

\[ \begin{array}{c}
T_{2} \\
\text{.................} \\
\text{Z=Stack[top]} \\
\text{Top++} \\
\text{.................} \\
\end{array}\]

Interference!
Exercise:

• Show a concrete case (with numbers in the stack) in which the stack becomes inconsistent and/or the processes read wrong data from the stack
Locks

• **lock_acquire**
  – wait until lock is free, then take it
• **lock_release**
  – release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)
Locks allow concurrent code to be much simpler:

lock_acquire()
if (!milk) buy milk
lock_release()

• How do we implement locks? (Later)
  – Hardware support for read/modify/write instructions
Lock Example: Malloc/Free

```c
char *malloc(n) {
    lock_acquire(Mlock);
    p = allocate memory
    lock_release(Mlock);
    return p;
}

void free(char *p) {
    lock_acquire(Mlock);
    put p back on free list
    lock_release(Mlock);
}
```
Rules for Using Locks

• Lock is initially free
• Always acquire before accessing shared data structure
  – Beginning of procedure!
• Always release after finishing with shared data
  – End of procedure!
  – DO NOT throw lock for someone else to release
• Never access shared data without lock
  – Danger!
Will this code work?

```c
newP() {
    p = malloc(sizeof(p));
    p->field1 = ...
    p->field2 = ...
    return p;
}

if (p == NULL) {
    lock_acquire(lock);
    if (p == NULL) {
        p = newP();
    }
    release_lock(lock);
}
use p->field1
```
Lock example: Bounded Buffer

```
tryget() {
    item = NULL;
    lock.acquire();
    if (nelem>0) {
        item = buf[front];
        front = (front++)%size;
        nelem --;
    }
    lock.release();
    return item;
}

tryput(item) {
    lock.acquire();
    if (nelem < size) {
        buf[last] = item;
        last = (last ++)%size;
        nelem ++;
    }
    lock.release();
}
```

Initially: nelem = front = last = 0; lock = FREE;
size is buffer capacity
Condition Variables

• Called only when holding a lock

• Wait: atomically release lock and relinquish processor until signaled

• Signal: wake up a waiter, if any

• Broadcast: wake up all waiters, if any
Example: Producer/Consumer

Definition:
• One (or more) producer(s) deposits messages in a shared buffer
• One (or more) consumer(s) extracts messages from the buffer

Requirements:
• Each message that is produced must be consumed exactly once.
Example: Bounded Buffer

get() {
    lock.acquire();
    while (nelem == 0)
        empty.wait(lock);
    item = buf[front];
    front = (front++) % size;
    nelem --;
    full.signal(lock);
    lock.release();
    return item;
}

put(item) {
    lock.acquire();
    while (nelem == size)
        full.wait(lock);
    buf[last] = item;
    last = (last++) % size;
    nelem ++;
    empty.signal(lock);
    lock.release();
}

Initially: nelem = front = last = 0; size is buffer capacity
empty/full are condition variables
Pre/Post Conditions

• What is state of the bounded buffer at lock acquire?
  – nelem>=0 (and front <= last, without considering the modular operation)
  – nelem<=size (and front + size >= last, without considering the modular operations)
  – (also true on return from wait)

• Also true at lock release!
• Allows for proof of correctness
Condition Variables

• ALWAYS hold lock when calling wait, signal, broadcast
  – Condition variables provide synchronization FOR shared state
  – ALWAYS hold lock when accessing shared state
• Condition variable is memoryless
  – If signal when no one is waiting, no op
  – If wait before signal, waiter wakes up
• Wait atomically releases lock
  – What if wait, then release?
  – What if release, then wait?
Condition Variables, cont’d

• When a thread is woken up from wait, it may not run immediately
  – Signal/broadcast put thread on ready list
  – When lock is released, anyone might acquire it

• Wait MUST be in a loop
  while (needToWait())
    condition.Wait(lock);

• Simplifies implementation
  – Of condition variables and locks
  – Of code that uses condition variables and locks
When waiting upon a Condition, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.
Structured Synchronization

• Identify objects or data structures that can be accessed by multiple threads concurrently
• Add locks to object/module
  – Grab lock on start to every method/procedure
  – Release lock on finish
• If need to wait
  – while(needToWait()) condition.Wait(lock);
  – Do not assume, when you wake up, that signaller just ran
• If do something that might wake someone up
  – Signal or Broadcast
• Always leave shared state variables in a consistent state
  – When lock is released, or when waiting
Mesa vs. Hoare semantics

- **Mesa (in textbook, Hansen)**
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor

- **Hoare**
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!
Mesa & Hoare semantics

• The producer/consumer solution works well with both Mesa & Hoare semantics
• However it does not make any assumption on the order in which:
  – The producers that are waiting are waked up and deposit their messages
  – The consumers that are waiting are waked up
FIFO Bounded Buffer
(Hoare semantics)

get() {
    lock.acquire();
    while (nelem == 0)
        empty.wait(lock);
    item = buf[front];
    front = (front++) % size;
    nelem --;
    full.signal(lock);
    lock.release();
    return item;
}

put(item) {
    lock.acquire();
    while (nelem == size)
        full.wait(lock);
    buf[last] = item;
    last = (last++) % size;
    nelem ++;
    empty.signal(lock);
    // CAREFUL: someone else ran
    lock.release();
}

Initially: nelem = front = last = 0; size is buffer capacity
empty/full are condition variables
FIFO Bounded Buffer (Mesa semantics)

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!

- Easily extends to case where queue is LIFO, priority, priority donation, ...
  - With Hoare semantics, not as easy
FIFO Bounded Buffer
(Mesa semantics, put() is similar)

get() {
    lock.acquire();
    if ( nelem == 0 || !nextGet.empty() ) {
        self = new Condition;
        nextGet.Append(self);
        while (nelem == 0)
            self.wait(lock);
        nextGet.Remove(self);
        delete self;
    }
    item = buf[front];
    front= (front++) % size;
    nelem --;
    if (!nextPut.empty())
        nextPut.first()->signal(lock);
    lock.release();
    return item;
}
Implementing Synchronization

<table>
<thead>
<tr>
<th>Concurrent Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphores</td>
</tr>
<tr>
<td>Locks</td>
</tr>
<tr>
<td>Condition Variables</td>
</tr>
<tr>
<td>Interrupt Disable</td>
</tr>
<tr>
<td>Atomic Read/Modify/Write Instructions</td>
</tr>
<tr>
<td>Multiple Processors</td>
</tr>
<tr>
<td>Hardware Interrupts</td>
</tr>
</tbody>
</table>
Implementing Synchronization

Take 1: using memory load/store
  – See “too much milk” solution / Peterson’s algorithm

Take 2:
  lock.acquire() { disable interrupts }
  lock.release() { enable interrupts }
Lock Implementation, Uniprocessor

LockAcquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(current TCB);
        suspend(); *
    } else {
        value = BUSY;
    }
    enableInterrupts();
}

* Invokes the scheduler, context switch & enable interrupts

LockRelease() {
    disableInterrupts();
    if (!waiting.Empty()){
        thread = waiting.Remove();
        readyList.Append(thread);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
Multiprocessor

• Read-modify-write instructions
  – Atomically read a value from memory, operate on it, and then write it back to memory
  – Intervening instructions prevented in hardware

• Examples
  – Test and set
  – Intel: xchgb, lock prefix
  – Compare and swap

• Does it matter which type of RMW instruction we use?
  – Not for implementing locks and condition variables!
Spinlocks

Lock where the processor waits in a loop for the lock to become free
  – Assumes lock will be held for a short time
  – Used to protect ready list to implement locks

SpinlockAcquire() {
    while (testAndSet(&spinLockValue) == BUSY) {
    }
}
SpinlockRelease() {
    spinLockValue = FREE;
}
Spinlocks: a low-level implementation

// &spinLockValue is a memory cell containing a binary value: Free (0) or BUSY (1)

// TSL R, &spinLockValue :
// writes the content of &spinLockValue in R and writes BUSY (1) in &LockValue

SpinlockAcquire(Lockvalue) {
    Loop: TSL R, & spinLockValue
    CMP R, BUSY
    JEQ Loop:  
    RET // at this point &lockValue == BUSY!!!!
}

SpinlockRelease() {
    MOV #FREE, &spinLockValue  // this unlocks a thread in the loop, if any
}

Lock Implementation, Multiprocessor

LockAcquire()
{
    spinLock.Acquire();
    if (value == BUSY){
        waiting.add(current TCB);
        sched.suspend(&spinLock); /*
    } else {
        value = BUSY;
        spinLock.Release();
    }
    * scheduler: marks thread as waiting; release spinlock; schedules next thread;
}

LockRelease()
{
    spinLock.Acquire();
    if (!waiting.Empty()){
        thread = waiting.Remove();
        sched.makeReady (thread, &spinLock); /*
    } else {
        value = FREE;
    }
    spinLock.Release();
    * scheduler: marks thread as ready, put it in the ready list.
Lock Implementation, Linux

• Fast path
  – If lock is FREE, and no one is waiting, test&set

• Slow path
  – If lock is BUSY or someone is waiting, see previous slide

• User-level locks
  – Fast path: acquire lock using test&set
  – Slow path: system call to kernel, to use kernel lock
Semaphores

• Semaphore has a non-negative integer value
  – P() atomically waits for value to become > 0, then decrements
  – V() atomically increments value (if no waiter is present); else it wakes up one waiter

• Semaphores are like integers except:
  – Only operations are P and V
  – Operations are atomic
    • If value is 1, two P’s will result in value 0 and one waiter

• Semaphores are useful for
  – Unlocked wait: interrupt handler, fork/join
P&V Implementation, Multiprocessor

P(sem) {
    spinLock.Acquire();
    disableInterrupts();
    if (sem.value == 0) {
        waiting.add(current TCB);
        suspend(&spinLock); *
    } else {
        sem.value --;
        spinLock.Release();
        enableInterrupts();
    }
} * suspends, invokes scheduler, context switch & enable interrupts

V(sem) {
    spinLock.Acquire();
    disableInterrupts();
    if (!waiting.Empty()) {
        thread = waiting.Remove();
        readyList.Append(thread);
    } else {
        sem.value ++;
    }
    spinLock.Release();
    enableInterrupts();
}
Semaphore Bounded Buffer

get() {
    empty.P();
    mutex.P();
    item = buf[front]
    front= (front+1) % size;
    mutex.V();
    full.V();
    return item;
}

put(item) {
    full.P();
    mutex.P();
    buf[last] = item;
    last = (last +1) % size;
    mutex.V();
    empty.V();
}

Initially: front = last = 0; size is buffer capacity
empty/full are semaphores (initialized to 0 and size)
Mutex is a semaphore initialized to 1
Implementing Condition Variables using Semaphores (Take 1)

```java
wait(lock) {
    lock.release();
    sem.P();
    lock.acquire();
}

signal() {
    sem.V();
}
```
Implementing Condition Variables using Semaphores (Take 2)

```java
wait(lock) {
    lock.release();
    sem.P();
    lock.acquire();
}

signal() {
    if semaphore is not empty
        sem.V();
}
```
Implementing Condition Variables using Semaphores (Take 3)

```java
wait(lock) {
    sem = new Semaphore;
    queue.Append(sem); // queue of waiting threads
    lock.release();
    sem.P();
    lock.acquire();
}

signal() {
    if !queue.Empty() {
        sem = queue.Remove();
        sem.V(); // wake up waiter
    }
    sem.V(); // wake up waiter
}
```
Synchronization Summary

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()