Review

- The Critical Section problem
- Peterson's Algorithm
- Implementing Atomicity
- Busy waiting
- Blocking

Outline

- Semaphores
  - what they are
  - how they are used
- Monitors
  - what they are
  - how they are used

What's wrong with this picture?

Our solutions to CS have two unappealing features:

- They are mistifying and unclear
- They use busy waiting

Semaphores

A semaphore consists of:

- A variable *s*
- A thread set *T*
- A function *P* (*Passeren*; wait)
- A function *V* (*Vrijgeven*; signal)

syntax example: Semaphore s(5);

Semaphore Operations

Initialization(*val*)
{
  s = *val* ;
  $T = \perp$;
}

P(s)

```
s--;
if(s < 0)
{
   add caller thread to T;
   block;
}
```

This code executes atomically.

V(s)

```
s++;
if(s ≤ 0)
{
   Remove a thread t from T;
   Unblock(t);
}
```

This code executes atomically.

Mutual exclusion with semaphores

Thread T_0	Thread T_1
while (!terminate) {	**while** (!terminate) {
P(s);	P(s);
CS	CS
V(s);	V(s);
NCS_0	NCS_1
}	}

init: *s* = 1

Questions, Questions...

- What if we have *n* threads instead of 2?
- What if we want to allow at most *k* threads in the CS?

Binary Semaphore

Two types:

- Counting semaphores
  - *s* takes any values
- Binary semaphores
  - *s* takes only 0 or 1

Binary Semaphores

P(s)

```
if (s == 0)
{
    Add caller thread to T;
    Block;
}
else
    s--;
```

V(s)

```
if ( T != ⊥)
{
    Remove a thread t from T;
    Unblock(t);
}
else if (s == 0)
        s = 1;
```

Important properties of Semaphores -I

- Semaphores are *non-negative* integers
- The *only* operations you can use to change the value of a semaphore are P() and V() (except for the initial setup)
- Semaphores have two uses
  - *Mutual exclusion*: you know all about it
  - *Synchronization*: how do we make sure that T_1 completes execution before T_2 starts?

Important Properties of Semaphores- II

- We assume that a semaphore is *fair:*

  No thread t that is blocked because of a P(s) operation remains blocked if s is V'd infinitely often

  Does this guarantee bounded waiting?
- In practice, FIFO is mostly used, transforming the set into a queue.
- V(*s*) never blocks.
- Binary semaphores are as expressive as general semaphores (given one can implement the other)
  - But they are different:

    {*s* = 0} V(s) V(s) P(s) P(s) VS

    {*s* = 0} V_b(s) V_b(s) P_b(s) P_b(s)

Classic Problems: Producer-Consumer I

- Producer adds items to a shared buffer
- Consumer takes them out
- Buffer avoids producers and consumers to proceed in lock step
- Buffer is finite

Examples:

- cpp|cc|as
- coke machine
- ready queue in a multithreaded multiprocessor
- ….

Producer Consumer II

- Two types of constraint:
  - Mutual Exclusion
    - *Only one thread can manipulate the buffer at any one time*
  - Condition Synchronization
    - *Consumer must wait if buffer is empty*
    - *Producer must wait if buffer is full*
- Are these safety or liveness properties?

Producer Consumer III

- Use a separate semaphore for each constraint

```
Semaphore mutex;       // protects the shared buffer
Semaphore fullSlots;   // counts full slots: if 0, no
                       // coke
Semaphore emptySlots;  // counts empty slots: if 0,
                       // nowhere to put more coke
```

Producer Consumer IV

```
Semaphore new mutex (1);
Semaphore new
   emptySlots(numSlots);
Semaphore new fullSlots(0);

Producer() {
   P(emptySlots);
   P(mutex);
   put 1 coke in the machine
   V(mutex);
   V(fullSlots);
}
```

Is the order of *P*s important?

```
Consumer() {
   P(fullSlots);
   P(mutex);
   take a coke out
   V(mutex);
   V(emptySlots);
}
```

Is the order of *V*s important?

Beyond Semaphores

- Semaphores huge step forward (immagine having to do producer consumer with Peterson's algorithm…)
- BUT…Semaphores are used both for mutex and condition synchronization
  - hard to read code
  - hard to get code right

Monitors (late 60's, early 70's)

- A programming language construct, much like what we call today objects
- Consists of:
  - General purpose variables
  - Methods
  - Initialization code (constructor)
  - Condition variables
- Can implement a monitor-like style of programming without without the language construct, using a combination of **locks** and **condition variables**

Locks

A lock provides mutual exclusion to shared data:

`Lock::Acquire()` - wait until lock is free, then grab it

`Lock::Release()` - unlock; wake up anyone waiting in Acquire

Rules:

- Always acquire before accessing a shared data structure
- Always release after finishing with shared data structure
- Lock is initially free

Example: Producer Consumer

```
addToBuff() {
   lock.Acquire();       // lock before using shared data
   put item on buffer, if there is space
   lock.Release();
}
removeFromBuff() {
   lock.Acquire();       // lock before using shared data
   if something on buffer, remove it
   lock.Release();
   return item;
}
```

Houston, we have a problem…

- How do we change `removeFromBuff` so that it check if there is something in the buffer before trying to remove it?
- Logically, want to suspend thread inside the critical section
- What is the problem?
- Solution: suspend and **atomically** release lock