Previous lecture overview

- Concurrently executing threads often share data structures.
- If multiple threads are allowed to access shared data structures unhindered race condition may occur
- To protect shared data structure from race conditions the thread’s access to it should be mutually exclusive
- MX may be implemented in software:
  - for two threads - Peterson’s algorithm
  - for multiple threads - bakery algorithm
- MX may be implemented using hardware support
- Writing efficient MX algorithms is not trivial and OS usually provides MX primitives for a programmer (as well as uses them internally)

Lecture 10: Semaphores

- Definition of a semaphore
- using semaphores for MX
- semaphore solutions for common concurrency problems:
  - producer/consumer
  - readers/writers
  - dining philosophers
- implementation of semaphores
  - using spinlocks
  - using test-and-set instructions
  - semaphores without busy waiting
- evaluation of semaphores

Semaphores — OS support for mutual exclusion

- Semaphores were invented by Dijkstra in 1965, and can be thought of as a generalized locking mechanism.
- semaphore supports two atomic operations
  - P / wait and V / signal. The atomicity means that no two P or V operations on the same semaphore can overlap
  - The semaphore initialized to 1
  - Before entering the critical section, a thread calls “P(s)”, or sometimes “wait(s)”
  - After leaving the critical section, a thread calls “V(s)”, or sometimes “signal(s)”

Semaphores — details

- P and V manipulate an integer variable the value of which is originally “1”
- Before entering the critical section, a thread calls “P(s)” or “wait(s)”,
  - wait (s):
    - s = s – 1
    - if (s < 0)
      - block the thread that called wait(s) on a queue associated with semaphore s
    - otherwise
      - let the thread that called wait(s) continue into the critical section
- After leaving the critical section, a thread calls “V(s)” or “signal(s)”,
  - signal (s):
    - s = s + 1
    - if (s ≤ 0), then
      - wake up one of the threads that called wait(s), and run it so that it can continue into the critical section

Bounded wait condition (not specified originally): if signal is continuously executed each individual blocked process is eventually woken up

Using semaphores for MX

```c
int s = 1; // semaphore

void t1 ()
{
    while (true) {
        wait(s); /* CS */
        signal(s); /* non-CS */
    }
}

void t2 ()
{
    while (true) {
        wait(s); /* CS */
        signal(s); /* non-CS */
    }
}
```

- The semaphore s is used to protect critical section CS
- before entering CS a thread executes wait(s)
- by definition of wait it:
  - decrements s
  - checks if s is less than 0; if it is then the thread is blocked. If not then the thread proceeds to CS excluding the other from reaching it
- after executing CS the thread does signal(s)
  - increments s
  - checks if s≤0; if it is then the other thread is woken up

Semaphore values

- Semaphores (again):
  - wait (s):
    - s = s – 1
    - if (s < 0)
      - block the thread that called wait(s) on a queue associated with semaphore s
    - otherwise
      - let the thread that called wait(s) continue into the critical section
  - signal (s):
    - s = s + 1
    - if (s ≤ 0), then
      - wake up one of the threads that called wait(s), and run it so that it can continue into the critical section

- Semaphore values:
  - Positive semaphore = number of (additional) threads that can be allowed into the critical section
  - Negative semaphore = number of threads blocked (note — there’s also one in CS)
  - Binary semaphore has an initial value of 1
  - Counting semaphore has an initial value greater than 1
"Too much milk" with semaphores

Too much milk

### Thread A
- `P(fridge);`
- if (noMilk)
  - buy milk;
  - noMilk = false;
- `V(fridge);`

### Thread B
- `P(fridge);`
- if (noMilk)
  - buy milk;
  - noMilk = false;
- `V(fridge);`

*"fridge" is a semaphore initialized to 1, noMilk is a shared variable*

 Execution:

**After:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(fridge);</td>
<td>0 in CS</td>
</tr>
<tr>
<td>P(fridge);</td>
<td>- 1 in CS waiting</td>
</tr>
<tr>
<td>V(fridge);</td>
<td>0 finish ready, in CS</td>
</tr>
<tr>
<td>V(fridge);</td>
<td>1 finish</td>
</tr>
</tbody>
</table>

---

### Producer/consumer problem

- `int p, c, buff[B], front=0, rear=0; semaphore empty(B), full(0), mutex(1);`
- `producer() { while (true) { /* produce p */ wait(empty); wait(mutex); buff[rear]=p; signal(mutex); signal(full); /* consume c */ } }`
- `consumer() { while (true) { wait(full); wait(mutex); c=buff[front]; front=(front+1) % B; signal(mutex); signal(empty); /* consume c */ } }`

---

### Readers/writers problem

- `int readcount; semaphore wrt(1), mutex(1);`
- **writer()**
  - `wait(wrt); /* perform write */ signal(wrt);`  
- **reader()**
  - `wait(mutex); readcount ++; if(readcount=1) wait(wrt); signal(mutex); /* perform read */ signal(mutex); signal(mutex);`  

---

### Dining philosophers problem

- `semaphore fork[5](1); philosopher(int i) { while(true) { wait(fork[i%5]); if(i%2) /* think */ signal(fork[i%5]); else if(i%2) /* eat */ signal(fork[i%5]); /* think */ } }`

---

### Two versions of Semaphores

- Semaphores from last time (simplified):
  - `wait(s); s = s - 1`
  - `if (s < 0) block the thread that called wait(s) otherwise continue into CS`
  - "Classical" version of semaphores:
    - `wait(s); if (s <= 0) signal(s); if (s < 0) wake up one of the waiting threads`
    - `continue into CS`
  - Do both work? What is the difference?

---

### Implementing semaphores: busy waiting (spinlocks)

- `wait(semaphore s) { while (s <= 0) do nothing */ signal(s); s++; }`
  
- **Problem:** wait and signal operations are not atomic
  
  - `idea: inside wait continuously check the semaphore variable (spins) until unblocked`
Implementing semaphores: busy waiting (spinlocks)

```
wait(semaphore s) {
    /* disable interrupts */
    while (s <= 0)
        ; /* do nothing */
    /* enable interrupts */
}

signal(semaphore s) {
    /* disable interrupts */
    s++;
    /* enable interrupts */
}
```

adv: may be efficient on multiprocessors — no need for context switch

disadvantages
- does not support bounded wait condition
- waiting thread wastes time busy-waiting (doing nothing useful, wasting CPU time)
- how long can a thread wait?
- can interfere with timer (interrupts)

Read-modify-write (RMW) instructions

```
int testnset(boolean *i){
    if (*i==FALSE)
        *i=TRUE;
    return(FALSE);
    else
        return(TRUE);
}
```

RMW instructions atomically read a value from memory, modify it, and write the new value to memory
- Test&set — on most CPUs
- Exchange — Intel x86 — swaps values between register and memory
- Compare&swap — Motorola 68xxx — read value, if value matches value in register r1, exchange register r1 and value
- Compare,compare&swap - SPARC
- RMW is not provided by "pure" RISC processors!

Semaphores using hardware support

This is a partial implementation

If lock is free (lock==false), test&set atomically:
- reads false, sets value to true, and returns false
- loop test fails, meaning lock is now busy

If lock is busy, test&set atomically:
- reads true and returns true
- loop test is true, so loop continues until someone releases the lock

Why is this implementation incomplete?
Adv: ensures atomicity of operation
Dis: does not support bounded wait

Semaphores (almost) without busy waiting

```
bool lock=false;

struct semaphore {
    public:
        int v;
        struct queue q;
} *s;

thread *ct;

wait(s) {
    s->v --;
    if(s->v < 0){
        enqueue(ct,s->q);
        block(ct);
    }
    signal(s) {
        thread *t;
        s->v++;
        if(s->v <= 0) {
            t= dequeue(s->q);
            wakeup(t);
        }
    }
}
```

*ct: pointer to current thread
*t: pointer to semaphore
*v: semaphore value
&q: queue of blocked threads waiting for semaphore

block blocks thread
wakeup wakes up a thread

This is an incomplete implementation. Why?
adv:
- no busy waiting,
- supports bounded wait
- dis: requires context switch

Semaphores - evaluation

- Semaphores provide the first high-level synchronization abstraction that is possible to implement efficiently in OS.
- this allows avoid using ad hoc Kernel synchronization techniques like non-preemptive kernel
- allows to implement in multiprocessors
- problems
  - programming with semaphores is error prone - the code is often cryptic
  - for signal and wait to be atomic on multiprocessor architecture - a low level locking primitives (like test&set instruction) need to be available
  - efficient blocking and unblocking require context switch - performance degradation
  - no means of finding out whether the thread will block on semaphore