Concurrently executing threads often share data structures.

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).

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(semaphore)”, or sometimes “wait(semaphore)”.
- After leaving the critical section, a thread calls “V(semaphore)”, or sometimes “signal(semaphore)”.

Using semaphores for MX

```c
/* semaphore s is used to protect critical section CS */
/* before entering CS a thread executes wait(s) */
/* by definition of wait: */
/* 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 */

/* semaphore s is used to protect critical section CS */
/* after executing CS the thread does */
/* by definition of signal: */
/* increments s */
/* checks if s > 0; if it is then the other thread is woken up */
```

Semaphore values

- Semaphores (simplified slightly):
  - 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

milk–>P();
if (noMilk)
buy milk;
milk–>V();

Thread B

milk–>P();
if (noMilk)
buy milk;
milk–>V();

“noMilk” is a semaphore initialized to 1

Execution:

After:

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>queue A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: milk–&gt;P();</td>
<td>0</td>
<td>in CS</td>
<td></td>
</tr>
<tr>
<td>B: milk–&gt;P();</td>
<td>-1</td>
<td>finish</td>
<td>waiting</td>
</tr>
<tr>
<td>A: milk–&gt;V();</td>
<td>0</td>
<td>finish ready, in CS</td>
<td></td>
</tr>
<tr>
<td>B: milk–&gt;V();</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Producer/consumer problem

```c
int p, c, *buffer[BOUND];
semaphore empty, full, mutex;
producer() {
    while (true) {
        /* produce p */
        wait(empty);
        wait(mutex);
        /* add p to buffer */
        signal(mutex);
        signal(full);
    }
}
consumer() {
    while (true) {
        wait(full);
        wait(mutex);
        /* remove c from buffer*/
        signal(mutex);
        signal(empty);
        /* consume c */
    }
}
```

Readers/writers problem

```c
int readcount;
semaphore wrt, mutex;
writer() {
    wait(wrt);
    /* perform write */
    signal(wrt);
}
reader() {
    wait(mutex);
    readcount++;
    if (readcount==1)
        wait(wrt);
    signal(mutex);
    /* perform read */
    wait(mutex);
    readcount--;
    if (readcount==0)
        signal(wrt);
    signal(mutex);
}
```

Two versions of Semaphores

Semaphores from last time (simplified):

```c
wait(s) {
    while (s <= 0)
    signal(s) {
    s = s + 1
    }
    s = s - 1
    continue into CS
}
```

“Classical” version of semaphores:

```c
wait(s) {
    if (s > 0)
        block the thread
    that called wait(s)
    otherwise
        continue into CS
}
```

Dining philosophers problem

The problem was first defined and solved by Dijkstra in 1972: five philosophers sit at the table and alternate between thinking and eating from a bowl of spaghetti in the middle of the table. They have five forks. A philosopher needs 2 forks to eat. Picking up and laying down a fork is an atomic operation.

Objective: design an algorithm to ensure that any “hungry” philosopher eventually eats one solution - protect each fork by a semaphore.

What’s wrong with this solution?
+ there is a possibility of deadlock
+ fix: make odd philosophers pick odd fork first - guarantees absence of deadlock but not fairness
+ can we use bakery algorithm?

Implementing semaphores: busy waiting (spinlocks)

```c
wait(semaphore s) {
    while (s <= 0)
        do nothing
    s--;
}
```

Evaluation:
- Doesn’t support queue of blocked threads waiting on the semaphore
- Waiting threads wastes time busy-waiting (doing nothing useful, wasting CPU time)

The code inside wait(s) and signal(s) is a critical section also, and it’s not protected
Implementing semaphores: busy waiting (spinlocks)

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

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

Evaluation:
- Doesn't support queue of blocked threads waiting on the semaphore
- Waiting threads wastes time busy-waiting (doing nothing useful, wasting CPU time)
- Doesn't work on multiprocessors
- Can interfere with timer, which might be needed by other applications
- OK for OS to do this, but users aren't allowed to disable interrupts! (Why not?)

semaphores: using hardware support

- 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 (lock==TRUE), test&set atomically:
  + reads TRUE and returns TRUE
  + loop test is true, so loop continues until someone releases the lock
- Evaluation:
  - Can be made to work, even on multiprocessors (although there may be some cache consistency problems)
  - Doesn't support queue of blocked threads waiting on the semaphore
  - Waiting threads waste time busy-waiting (doing nothing useful, wasting CPU time)

Read-modify-write (RMW) instructions

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

Semaphores without busy waiting

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);
    }
}

Convoys

- Frequent contention for (or coarse grained locking of) semaphores leads to convoys!
- A convoy is a situation when a always has to wait on a semaphore
- Example: file system access is protected by one "big" semaphore
- Breaking down a semaphore causing convoys into two sequential semaphores seldom helps - pipeline effect

Avoiding convoys

- Eliminating convoys is done by reprogramming the code so that the access to data protected by convoy causing semaphore can be done in parallel - stack up the pipes!
- Example: processes doing I/O on different pipes proceed in parallel
Semaphores - evaluation

- Semaphores provide the first high-level synchronization abstraction that is possible to implement efficiently in OS.
- This allows avoiding ad hoc Kernel synchronization techniques like non-preemptive kernel.
- Allows to implement in multiprocessors.
- Programming with semaphores is error prone - the code is often cryptic.
- For signal and wait to be atomic on multiprocessor architecture - low level locking primitives (like test&set instruction) need to be available.
- Blocking and unblocking require context switch - performance degradation.
- No means of finding out whether the thread will block on semaphore.
- Convoys.