Semaphores & Monitors

Today
- Semaphores
- Monitors
- … and some other primitives

Next time
- Deadlocks
Last time - locks

- Memory objects with two operations
  - acquire() - Prevents progress of the thread until the lock can be acquired
  - release()

- Come in two varieties – spinning and blocking
  - Spin if you expect a short wait and there are multiple cores
    ```
    acquire (lock) {
      while(lock->held); // caller busy waits
      lock->held = 1;
    }
    ```
  - Block if there’s only one core or you expect a long wait

- Of course, both operations must be atomic
  - Need hardware help – TSL or xchg
    ```
    while(xchg(&lk->locked, 1) != 0)
    ```
Semaphores

- A synchronization primitive
- Higher level of abstraction than locks
- Invented by Dijkstra in ‘68 as part of THE operating system
- Atomically manipulated by two operations
  - Down(sem) / wait / P
    - The calling process/thread locks until semaphore is available
  - Up(sem) / signal / V
    - If there are processes/threads blocked there, wake one up; keep count of signals otherwise
Blocking in semaphores

- Each semaphore has an associated queue of processes/threads
  - Down(sem) / wait / P
    - If sem was “available” (>0), decrement sem & let thread continue
    - If sem was “unavailable” (<=0), decrement it by 1, place thread on associated queue
  - P – not really for proberen or passeer but for a made-up word prolaag – “try to reduce”

```c
typedef struct {
    int value;
    struct thread *L;
} semaphore;

down(S):
    --Sem.value;
    if (Sem.value < 0){
        add this thread to Sem.L;
        block;
    }
```

Atomic action
Semaphores

...  
- \( V/\text{signal}/\text{up}(\text{sem}) \)
  - If thread(s) are waiting on the queue, unblock one
  - If no threads are waiting, increment \( \text{sem} \)
    - The signal is “remembered” for next time \( \text{up}(\text{sem}) \) is called
  - Might as well let the “up-ing” thread continue execution

- \( V – \text{verhogen} – \) increase in Dutch

\[
\text{up}(S):
\]

```c
 Sem.value++;  
if (Sem.value <= 0) {  
 remove a process P from Sem.L;  
 wakeup(P);  
}
```

Atomic action

- With multiple CPUs – lock semaphore with TSL
- *But then how’s this different from previous busy-waiting?*
# Semaphores

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
<th>Sem.L</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{}</td>
<td></td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>P1 down</td>
<td>0</td>
<td>{}</td>
<td>P1</td>
</tr>
<tr>
<td>P2 down</td>
<td>-1</td>
<td>{P2}</td>
<td>P1</td>
</tr>
<tr>
<td>P3 down</td>
<td>-2</td>
<td>{P2,P3}</td>
<td>P1</td>
</tr>
<tr>
<td>P1 up</td>
<td>-1</td>
<td>{P3}</td>
<td>P2</td>
</tr>
</tbody>
</table>

```java

down(Sem):  
--Sem.value;
if (Sem.value < 0) {
    add this thread to Sem.L;
    block;
}

up(Sem):  
Sem.value++;
if (S.value <= 0) {
    remove a thread P from Sem.L;
    wakeup(P);
}
```

---

[Image of a page with text and tables related to semaphores and their operations.]
Semaphores – Producer-Consumer

```
semaphore empty, // # of empty buffers, set to all
    full,  // count of full buffers, set to 0
mutex; // initially 1

Producer
while (TRUE) {
    item = produce_item();
    down(empty);
    down(mutex);
    insert_item(item);
    up(mutex);
    up(full);
}

Consumer
while (TRUE) {
    down(full);
    down(mutex);
    item = remove_item();
    up(mutex);
    up(empty);
    consume_item(item);
}
```

- Using semaphores in two different ways
  - To ensure mutual exclusion (mutex) and
  - For synchronization (empty/full) – to ensure certain event sequences do/don’t occur
Types of semaphores

- Binary semaphores – mutex
  - Sem is initialized to 1
  - Used to guarantee mutual exclusion
  - Useful with thread packages

    ```
    mutex_lock:
    TSL REGISTER, MUTEX
    CMP REGISTER, #0
    JXE ok
    CALL thread_yield
    JMP mutex_lock

    ok: RET
    ```

- Counting semaphores
  - Let \(N\) threads into critical section, not just one
  - Sem is initialized to \(N\), number of (identical) units available
  - Allow threads to enter as long as there are units available
Readers-writers problem

- Model access to database
- One shared database
  - Multiple readers allowed at once
  - Only one writer allowed at a time
    - If writers is in, nobody else is

```c
semaphore db,      // mutex for writers (only one) and
mutex;       // first/last reader
mutex;   // mutual exclusion for rc update
int rc;            // read count or number of readers in

void writer(void)
{
    while(TRUE) {
        think_up_data();
        down(&db);
        write_db();
        up(&db);
    }
}
```
Readers-writers problem

```c
void reader(void)
{
    while(TRUE) {
        down(&mutex);  // mutex for update rc (reader count)
        ++rc;
        if (rc == 1) down(&db); // mutex for writers (only one) and
                                // first/last reader
        up(&mutex);
        read_db();
        down(&mutex);  // on the way out
        --rc;
        if (rc == 0) up(&db); // if I am the last reader, let the
                                // writer in
        up(&mutex);
        use_data();
    }
}
```

**What problem do you see for the writer?**

Idea for an alternative solution: When a reader arrives, if there’s a
writer waiting, the reader could be suspended behind the writer
instead of being immediately admitted.
Mutexes in Pthreads

- Basic mechanism – mutex
  
  ```
  pthread_mutex_init - create it
  pthread_mutex_destroy - destroy it
  pthread_mutex_lock - acquire it or block
  pthread_mutex_trylock - acquire or fail (you can spin then)
  pthread_mutex_unlock - release it
  ```

- Also supports conditions variables
  
  - Typically used to block threads until a condition is met
  - Must always be associated with a mutex to avoid a race condition between a thread preparing to wait and another one signaling it (signal arriving before the thread is actually waiting)

  ```
  pthread_cond_init - create it
  pthread_cond_destroy - destroy it
  pthread_cond_wait - yield until the condition is satisfied
  pthread_cond_signal - restart one of the threads waiting on it
  pthread_broadcast - restart all threads waiting on it
  ```
Mutexes in Pthreads

```c
#include <pthread.h>

pthread_mutex_t mutex;
pthread_cond_t condc, condp;

void *producer(void *ptr)
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&mutex);
        while (buffer != 0) pthread_cond_wait(&condp, &mutex);
        buffer = i;
        pthread_cond_signal(&condc); /* wakeup consumer */
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}

void *consumer(void *ptr)
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&mutex);
        while (buffer == 0) pthread_cond_wait(&condc, &mutex);
        buffer = 0;
        pthread_cond_signal(&condp); /* wakeup producer */
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}
```

Clearly missing a few definitions, including main

It atomically unlocks `mutex`, passed on as a parameter, and waits.
Problems with semaphores & mutex

- Solves most synchronization problems, but:
  - Semaphores are essentially shared global variables
    - Can be accessed from anywhere (bad software engineering)
  - No connection bet/ the semaphore & the data controlled by it
  - Used for both critical sections & for coordination (scheduling)
  - No control over their use, no guarantee of proper usage

“What happens if the buffer is full?”

```
// producer
while (TRUE) {
  item = produce_item();
  down(empty);
  down(mutex);
  insert_item(item);
  up(mutex);
  up(full);
}
```

```
// producer
while (TRUE) {
  item = produce_item();
  down(mutex);
  down(empty);
  insert_item(item);
  up(mutex);
  up(full);
}
```
Monitors

- Monitors - higher level synchronization primitive
  - A programming language construct
    - Collection of procedures, variables and data structures
  - Monitor’s internal data structures are private
  - One can use the ideas working with shared objects

- Monitors and mutual exclusion
  - Only one process active at a time - how?
  - Synchronization code is added by the compiler (or the programmer using locks)
Monitors

- Once inside a monitor, a thread may discover it can’t continue, and
  - Wants to wait, or
  - Inform another one that some condition has been satisfied

- To enforce sequences of events – Condition variables
  - Only accessed from within the monitor
  - Three operations – wait, signal & broadcast
Monitors

- **Wait**
  - Atomically releases the lock
  -Suspends execution of the calling thread, place it in the waiting queue
  -When the calling thread is re-enable, it requires the lock before returning from the wait

- A thread that waits “steps outside” the monitor (to the associated wait queue)

- A condition variable is memoryless, it has no internal state (the shared object defines its own); so, *wait* is not a counter – signal may get lost
Monitors

- Signal – Takes one waiting thread off the condition variable’s waiting queue and marks it as eligible to run
- Broadcast – Like signal but for all waiting

What happen after the signal?
- Hoare – process awakened run, the other one is suspended
- Brinch Hansen – process signaling must exit the monitor
- Third option? Process signaling continues to run (Mesa)
- All systems today use Hansen’s semantics

As a programmer – this means wait must always be called from within a loop

```c
while (predicateOnStateVar(...)) {
    wait(&lock);
}
```
Message passing

- IPC in distributed systems
- Message passing
  \[
  \text{send}(\text{dest}, \ & \text{msg})
  \]
  \[
  \text{recv}(\text{src}, \ & \text{msg})
  \]
- Design issues
  - Lost messages: acks
  - Duplicates: sequence #s
  - Naming processes
  - Performance
  - ...
### Producer-consumer with message passing

```c
#define N 100 /* num. of slots in buffer */

void producer(void)
{
    int item; message m;

    while(TRUE) {
        item = produce_item();
        receive(consumer, &m);
        build_message(&m, item);
        send(consumer, &m);
    }
}

void consumer(void)
{
    int item, i; message m;

    for(i = 0; i < N; i++)
        send(producer, &m);

    while(TRUE) {
        receive(producer, &m);
        item = extract_item(&m);
        send(producer, &m);
        consume_item(item);
    }
}
```
Barriers

- To synchronize groups of processes
- Type of applications
  - Execution divided in phases
  - Process cannot go into new phase until all can
- e.g. Temperature propagation in a material
Join

- Sometimes you want to wait until a thread has terminated – *join()*

- A common use:
  - Start *N* threads
  - Sit in a loop waiting for all threads …
    - It really doesn't matter much which one finishes first, you just wait in an arbitrary order

- Similar but not the same as a barrier
  - *join()* waits until threads have terminated, and so given up all their resources
  - A barrier is achieved before threads have terminated
Dining philosophers problem

- Philosophers eat/think
- To eat, a philosopher needs 2 chopsticks
- Picks one at a time
- **How to prevent deadlock**

```c
#define N 5

void philosopher(int i) {
    while (TRUE) {
        think();
        take_chopstick(i);
        take_chopstick((i+1)%N);
        eat();
        put_chopstick(i);
        put_chopstick((i+1)%N);
    }
}
```

**Nonsolution**

Why not just protect all this with a mutex?

Now: Everybody takes the left chopstick!
Dining philosophers example

```c
void philosopher(int i) {
    while(TRUE) {
        think();
        take_chopstick(i);
        eat();
        put_chopstick(i);
    }
}

void take_chopstick(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_chopstick(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(int i) {
    if ((state[i] == hungry &&
        state[LEFT] != eating &&
        state[RIGHT] != eating) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

state[] – too keep track of philosopher’s state (eating, thinking, hungry)
s[] – array of semaphores, one per philosopher
void philosopher(int i)
{
    while(TRUE) {
        dp.take_chopstick(i);
        eat();
        dp.put_chopstick(i);
    }
}

Monitor dp
{
    enum {EATING, HUNGRY, EATING}
    state[5];
    condition s[5];

    void take_chopstick(int i)
    {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            s[i].wait();
    }

    void put_chopstick(int i)
    {
        state[i] = THINKING;
        test(LEFT); test(RIGHT);
    }

    void test(int i)
    {
        if ((state[i] == HUNGRY &&
            state[LEFT] != EATING &&
            state[RIGHT] != EATING) {
            state[i] = EATING;
            s[i].signal();
        }
    }

    void setup()
    {
        for (i = 0; i < 5; i++)
            s[i].wait();
        state[i] = THINKING;
    }
} /* end Monitor dp */
Coming up

- Deadlocks
  How deadlock arise and what you can do about them
Monitors in Java

- Not truly a monitor
  - Every object contains a lock
  - The synchronized keyword locks that lock
  - Can be applied to methods or blocks of statements

- Synchronized method
  - e.g. atomic integer

- Synchronized statements
  - You can lock any object, and have the lock released when you leave the block of statements

```java
class atomicInt {
    int value;
    ... 
    public synchronized postIncrement() {
        return value++;
    }
    ...
}

class Foo {
    void foo (ArrayList list) {
        ... 
        synchronized(list) {
            // manipulate list now
        }
        ...
    }
}
```
The sleeping barber problem

One barber, one barber chair and $n$ chairs for waiting customers …

Arriving customer wakes up the barber.

Additional customers arriving while barber’s busy – either wait or leave.

No customers, take a nap.
The sleeping barber problem

#define CHAIRS 5

void barber (void)
{
    while (TRUE) {
        ...
        ...
        /* sleep if no customers */
        --waiting;
        ...
        ...
        cut_hair();
    }
}

void customer (void)
{
    ...
    if (waiting < CHAIRS) {
        ++waiting; /* sit down */
        ...
        ...
        ...
        get_haircut();
    } else { /* go elsewhere */
        ...
    }
}

Semaphores:
- Customer - count waiting customers (excluding the one in the barber chair)
- Barbers - number of barbers who are idle
- mutex - for mutual exclusion
#define CHAIRS 5

void barber (void)
{
  while (TRUE) {
    down(&customers);
    /* sleep if no customers */
    down(&mutex);
    --waiting;
    up(&barbers);
    up(&mutex);
    cut_hair();
  }
}

void customer (void)
{
  ...
  if (waiting < CHAIRS) {
    ++waiting; /* sit down */
    ...          
    ...          
    ...          
    get_haircut();
  } else { /* go elsewhere */
    ...          
  }
}

Semaphores:
- Customer - count waiting customers (excluding the one in the barber chair)
- Barbers - number of barbers who are idle
- mutex - for mutual exclusion
The sleeping barber problem

```c
#define CHAIRS 5

void barber (void)
{
    while (TRUE) {
        down(&customers);
        /* sleep if no customers */
        down(&mutex);
        --waiting;
        up(&barbers);
        up(&mutex);
        cut_hair();
    }
}

void customer (void)
{
    down(&mutex);
    if (waiting < CHAIRS) {
        ++waiting; /* sit down */
        up(&customers);
        up(&mutex);
        down(&barbers);
        get_haircut();
    } else { /* go elsewhere */
        up(&mutex);
    }
}
```

Semaphores:
- Customer - count waiting customers (excluding the one in the barber chair)
- Barbers - number of barbers who are idle
- mutex - for mutual exclusion