Semaphores & Monitors

Today
- Semaphores
- Monitors
- ... and some other primitives

Next time
- Deadlocks
Semaphores

- A variable atomically manipulated by two operations – down (P) & up (V)
- Each semaphore has an associated queue of processes/threads
  - P/wait/down(sem)
    - If sem was “available” (>0), decrement sem & let thread continue
    - If sem was “unavailable” (<=0), place thread on associated queue; run some other thread

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

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

- Semaphores thus have history
Semaphores

...  

– V/signal/up(sem)
  • If thread(s) are waiting on the associated queue, unblock one (place it on the ready queue)
  • If no threads are waiting, increment sem
    – The signal is “remembered” for next time up(sem) is called
  • Might as well let the “up-ing” thread continue execution

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

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>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);
    }
```
Semaphores

empty = # available slots, full = 0, mutex = 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);
}

● Semaphores and I/O devices
Mutexes

- Two different uses of semaphores
  - Synchronization – full & empty
  - Mutex – used for mutual exclusion

- Useful w/ thread packages
- Other possible operation

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

```c
mutex_unlock:  
  MOVE MUTEX, #0  
  RET
```
Mutexes in Pthreads

```c
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`
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 between 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

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

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

What happens if the buffer is 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
- Monitors and mutual exclusion
  - Only one process active at a time - how?
  - Synchronization code is added by the compiler

At most one thread in the monitor at any given time

Queue of threads waiting to get into the monitor
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
  - Can only be accessed from within the monitor
  - Two operations – *wait & signal*
  - A thread that waits “steps outside” the monitor (to a wait queue associated with that condition variable)
  - What happen after the signal?
    - Hoare – process awakened run, the other one is suspended
    - Brinch Hansen – process doing the signal must exit the monitor
    - *Third option? Process doing the signal continues to run (Mesa)*
  - *Wait* is not a counter – signal may get lost
public class ProducerConsumer {
    static final int N = 100; // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor
    public static void main(String args[]) {
        p.start(); // start the producer thread
        c.start(); // start the consumer thread
    }
    static class producer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // actually produce
    }
    static class consumer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // consumer loop
                item = mon.remove();
                consume_item(item);
            }
        }
        private void consume_item(int item) { ... } // actually consume
    }
Monitors in Java

```java
static class our_monitor {
    // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer[hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N; // slot to place next item in
        count = count + 1; // one more item in the buffer now
        if (count == 1) notify(); // if consumer was sleeping, wake it up
    }
    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
        val = buffer[lo]; // fetch an item from the buffer
        lo = (lo + 1) % N; // slot to fetch next item from
        count = count - 1; // one few items in the buffer
        if (count == N - 1) notify(); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {} }
}
```
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
  - ...
#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
Coming up

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