9 Lecture: Even More Concurrency

Outline:
- Announcements
- Synchronization without busy waiting
- Sleep and Wakeup
- Semaphores
  - Monitors: (Hoare 1974, Brinch Hansen 1975)
- More Interprocess Communication
  - Further generalization: message passing
- Robust Programming Clinic
- Classic IPC Problems

9.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>Jun 5 23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>Jun 7 23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>Jun 8 10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- Don’t just click the little ‘X’ to stop minix
- tryAsgn2
- Call stacks, Algol 60
- how to do exit()
- Wind up a stack.

9.2 Synchronization without busy waiting

9.3 Sleep and Wakeup

A nice idea, but broken.
9.4 Semaphores

Generalized `sleep()` and `wakeup()` using a counter.

- `p(counter)` down(counter) decrement counter; wait if zero
- `v(counter)` up(counter) increment counter; wakeup others if was zero

Must be **atomic**:

- usually done as a system call with disabled interrupts—note this is not the same as a long-term busywait.
- LWP can turn off signals to achieve the same effect.

Semaphores can be used for both (And these are different)

- mutual exclusion (binary semaphore)
- synchronization (initially N for producer-consumer)

Figure 15 shows a solution to the producer-consumer problem using semaphores. Still must be careful: **if you lock in the wrong order, you can deadlock**. Consider the effect of reversing the downs in figure 15.

```c
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void) {
    while(TRUE) {
        produce();
        down(&empty);
        down(&mutex);
        enter_item();
        up(&mutex);
        up(&full);
    }
}

void consumer(void) {
    while(TRUE) {
        down(&full);
        down(&mutex);
        remove_item();
        up(&mutex);
        up(&empty);
        consume();
    }
}
```

Figure 15: A semaphore-based producer-consumer implementation

9.4.1 Monitors: (Hoare 1974, Brinch Hansen 1975)

Recall the sensitivity of semaphores to ordering we saw in the example of the other time: If the producer and consumer lock `mutex` and `full/empty` in the wrong order, they will deadlock.

**Monitors** are a higher-level synchronization mechanism requiring programming-language support.

- A monitor is region of code where only one process can be active at a time. (enforced by the language).
• Communication between processes is done via **condition variables** with the primitives:

  **wait(condition)** wait until a signal. A process that blocks in the monitor releases other
  processes to enter.

  Note that this will be ok, because the release is *voluntary*.

  **signal(condition)** send a signal. To enforce the exclusion principle, a process that signals is
  required to leave the monitor immediately.

  A signal wakes one process waiting on that condition variable.

  These are very like sleep and wakeup, but automagically synchronized so counters are unnec-
  essary.

A monitor-based solution to the producer-consumer problem is shown in Figure 16.

### 9.5 More Interprocess Communication

Ok, so, looking at our mechanisms that work

- **Spin Locks**: Waste Time
- **Semaphores**: Complicated (easy to get wrong)
- **Monitors**: Require language support

So what else can we do?

#### 9.5.1 Further generalization: message passing

*If you need me, just whistle.*

Monitors require language support, a tricky thing in the “real world”. Message passing gets much
of the same safety using operating system support.

Processes communicate via two primitive functions:

• send(dst,message)

• receive(src,message)

Where do they get sent? (How do we address them?)

Options:

• individual processes

• mailboxes (possibly multiple consumers)

In either case, there may or may not be buffers. If there is a buffer, the OS has to manage this
buffer (a meta-producer-consumer problem :). If not, a sender is stopped until there is a receiver:

**rencyvous**

Issues with message passing,

• messages can be lost (acknowledgement)

• messages can be duplicated (sequence numbers)

• messages can be forged (authentication)

• performance:
monitor ProducerConsumer

condition full, empty;

integer count;

procedure enter()
begin
    if count = N then
        wait(full);
        enter_item();
        count := count + 1;
    if count = 1 then
        signal(empty);
end;

procedure remove()
begin
    if count = 0 then
        wait(empty);
        remove_item();
        count := count - 1;
    if count = N - 1 then
        signal(full);
end;

count := 0;
end monitor

procedure consumer()
begin
    while TRUE do
begin
    ProducerConsumer.remove();
    consume_item();
end
end;

procedure producer()
begin
    while TRUE do
begin
    produce_item();
    ProducerConsumer.enter();
end
end;

Figure 16: A monitor-based solution to the producer-consumer problem.
– Copying overhead (copy, copy, copy)
– Transmission latency (You may as well do it yourself)

But it can elegantly solve some problems, as with the producer-consumer solution given in Figure 17. Message passing producer-consumer: send empty messages and send back full ones.

```c
#define N 100

void producer(void) {
    message m;
    thing item;
    while(TRUE) {
        produce(&item);
        receive(consumer,&m);
        build_message(&m,item);
        send(consumer,&m);
    }
}

void consumer(void) {
    message m;
    thing item;
    int i;
    /* send N empties */
    for(i=0;i<N;i++)
        send(producer,&m);
    /* go to work */
    while(TRUE) {
        receive(producer,&m);
        extract_item(&m,item);
        send(producer,&m);
        consume(&item);
    }
}
```

Figure 17: A message-passing solution to the producer-consumer problem.

9.6 Classic IPC Problems

• Producer/Consumer (been there, done that?)

• Dining Philosophers (1965): synchronization rite of passage.
  Competing access for limited resources.
  Issues:
  – deadlock
  – starvation

• Readers/Writers: e.g. a database.
  Issues:
  – Many readers can work simultaneously, but only one writer.
  – Starvation: If we let all the readers by (no problem), the writer may starve.
  – perhaps an aging scheme?