Inter-Process Communication
Issues

- How a process can pass information to another
- Make sure processes don’t get into each others’ way
- Sequencing and dependencies
The issues...

- They apply to threads as well
- Communication: easy for threads (common address space)
- Remaining two issues apply to thread as to processes
Race condition

- Example: printer spooler (a daemon)

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 abc</td>
<td>out = 4</td>
</tr>
<tr>
<td>5 prog.c</td>
<td></td>
</tr>
<tr>
<td>6 prog.n</td>
<td>in = 7</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Spooler directory
Race condition

- Two processes reading/writing on the same data and the result depends on who runs precisely when is called a race condition
- Since obviously we’d like computation to be deterministic
Critical regions

- **Mutual exclusion**
- The part of the program where the shared memory (or something else) is accessed is called a *critical section*
- This is not enough (more rules):
  - Not two processes simultaneously in their critical regions
  - No assumptions may be made about speed and number of CPUs
  - No process running outside its critical region may block another process
  - No process should have to wait forever to enter its critical region
Ideally

A enters critical region
A leaves critical region

B enters critical region
B leaves critical region

B attempts to enter critical region
B blocked

Many solutions...

- Disabling interrupts
- Locks
- TSL instruction (hardware)
- Semaphores
- Mutexes
- Monitors
- Message passing
- ...

OS 2009-10
Disabling interrupts

- Simplest solution
- CPU switches from process to process only when an interrupt occurs (e.g. the clock interrupt)
- This approach can be taken by the kernel
- Should the OS trust the user in disabling/enabling interrupts? Too dangerous!
Locks

- A lock variable (alone it doesn’t work)
- Strict alternation (no two in the critical region, not convenient)

```c
while (TRUE)
{
    while (turn!=0);
    critical_region();
    turn = 1;
    noncritical_region();
}

while (TRUE)
{
    while (turn!=1);
    critical_region();
    turn = 0;
    noncritical_region();
}
```
Peterson’s solution

```c
#define FALSE 0
#define TRUE 1
#define N 2

int turn;
int interested[N]; // initialized = 0

void enter_region(int process)
{
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE);
}

void leave_region(int process)
{
    interested[process] = FALSE;
}
```

OS 2009-10
TSL instruction

- TSL RX, LOCK (test and set lock)
- Reads the content of LOCK into RX and stores a non-zero value into LOCK atomically (can’t be interrupted)
Example

**enter_region:**
- TSL REGISTER, LOCK
- CMP REGISTER, #0
- JNE enter_region
- RET

**leave_region:**
- MOVE LOCK, #0
- RET
Semaphores

- An **atomically** accessible counter. Similar to a lock but with multiple values and possibly blocking a process without busy-waiting
- There are two operations possible:
  - **Up**, **Down**
- Down, if 0 the process will go to sleep otherwise it decrements the semaphore and continues execution
- Up, increments the semaphore, if a process is sleeping on the semaphore, it is awakened, the caller never blocks
Example consumer-producer

```c
#define N 100
typedef int semaphore; /// with a bit of imagination

semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void consumer(void)
{
    int item;
    while (TRUE)
    {
        down(&full);
        d(t)
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

void producer(void)
{
    int item;
    while (TRUE)
    {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

OS 2009-10
Mutexes

- Semaphores with binary values
- What’s nice? Simpler implementation than semaphores
- Of course, a semaphore can be made to behave as a mutex and vice-versa a mutex is enough to implement a semaphore
Monitors

• Abstract construct (a package):
  ○ It’s a sort of class (in fact there’s something similar in Java)
  ○ Monitor’s data is *private*
  ○ Only one process can be **active** in a monitor at a given time
  ○ Condition variables: **wait** and **signal** primitives (equivalent to **down** and **up**)

OS 2009-10
monitor ProducerConsumer
    condition full, empty;
    integer count;

    /// PROCEDURES_HERE()
    /// it’s guaranteed that no process can change
    /// count at the same time, just need to check
    /// the full and empty conditions

    count = 0;
end monitor;
Message passing

• Why? Distributed systems for example
  - `send(destination, &message)`
  - `receive(source, &message)`
Issues with message passing

- **Acknowledgement (message)**
  - We need to be sure a message is not lost otherwise synchronization will go berserk
  - Message numbering
  - A good part of the study on computer networks

- **Authentication**
  - Make sure only who’s supposed to receive the message actually receives it and vice-versa
#define N 100

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

    while (TRUE)
    {
        item = produce_item();
        receive(consumer, &m);
        // waits for an EMPTY
        build_message(&m, item);
        send(consumer, &m);
    }
}

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

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

    while (TRUE)
    {
        receive(consumer, &m);
        item = extract_item(&m);
        send(producer, &m);
        send an EMPTY
        consume_item(item);
    }
}
Access to database

- Many readers
- Only one writer

- Issues: no write until all readers are out, but try not to accept other readers if a write is pending!
Dining philosophers