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)

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

Many solutions...

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)

Peterson’s solution

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)
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 producer(void)
{
  int item;
  while (TRUE)
  {
    item = produce_item();
    down(&empty);
    down(&mutex);
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}

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

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**)

Part of an example...

```c
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 berserker
  – 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

Example of message passing

```c
#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(producer, &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