UNIVERSITY of STIRLING

Concurrent and Distributed Systems

Inter-Process Communication & Synchronisation

Process (thread) Interactions

- Processes may need to *cooperate* to carry out a task
  - A process may make a request and *wait* for the service to be done
  - A process may need to send the requested data, or *signalling* that a task has been done.
- Cooperating processes are in charge of implementing interaction policies, which should fulfil desirable properties, such as absence of deadlock, fairness (non-starvation), …

  We need to keep this in mind when programming concurrent – either processes or threads - applications

- Competing (cooperating) processes need to *wait* to acquire a shared resource and need to be able to *signal* to indicate that they have finished with that resource

Synchronisation for shared data

- A number of processes are simultaneously accessing memory
  - Reading a memory location is atomic
  - Writing a memory location is atomic
- There may be arbitrary interleaving of machine instruction execution → arbitrary interleaving of memory accesses
- Between any two instructions any number of instructions from any other computation may be executed
- Consider again the usual standard example x:=x+1

  read + op + write

Critical Section Specification

Recall:

1. At most one process inside a critical section (mutual exclusion)
2. No assumptions on speeds or numbers of processors and processes
    1. Processes may be running on a multiprocessor
    2. Not necessarily a Hardware operation available
3. No process running outside its critical region may block other processes
4. No process should have to wait forever to enter its critical region
    1. no starvation,
    2. progress – no indefinite busy wait
5. No deadlock!

Possible Solutions

- Solutions
  - Hardware Level
    - Interrupts
    - TSL
  - Software Level
    - Low Level
      - Lock Variable
      - Strict Alteration
      - Peterson
      - Semaphore
    - Abstract Level
      - Monitors
      - Message Systems

Approaches with Busy Waiting

1. Disabling Interrupts

Interrupts are the base for timing and preemption and external device handling. Could be disabled:

- No clock interrupts can occur
- No CPU context switch
- No other processes can access the shared resource
- Possible abuse of the system!
- Not applicable to computers with more than one CPU (disabling interrupts only affects one CPU! processes on other CPU may continue running)
- Not applicable as a general mutual exclusion mechanism!

2. Test and Set Lock (TSL)

- Hardware operation often supported by multiprocessor computers
- Reads the contents of a memory address into a register and writes a nonzero value to that location
- Is register zero? If not some process is in the critical section
- Guaranteed to be indivisible
- No context switch can occur
- Memory bus is blocked to prevent other CPUs from accessing memory during the operation

TSL- implementation example

Uses **tsl** to implement a critical region using flag (memory cell)

```
enter_region:
        tsl register, flag        ;copy flag to register (r) & set flag to 1 (w)
        cmp register, #0          ;was flag set? If register = 0 return 0
        jnz enter_region          ;if not zero from cmp (flag set) loop
        ret                       ;critical region entered, return to caller
```

Any other enter_region *call here loops on* cmp register, #0

```
leave_region:
        mov flag, #0              ;unset flag
        ret                       ;left critical section, return to caller
```

(for xi86 CPUs was xchg m, r)

3. Lock Variables (I)

- Single, shared variable; initially set 0
- Enter a critical section: test variable
  - if 0, set it 1, enter critical section
  - If 1, wait until it becomes 0
- Race condition still occurs, if context switch happens after the check (two processes might read 0, then …)
- No solution (as is)!

Let’s see something more elaborated …

Lock Variables (II)

```
public class ME extends BaseME{

   public ME() {
          flag[0] = false;
          flag[1] = false;
   }

public void enteringCS(int t) {
    int other = 1 - t;
    flag[t] = true;

    while (flag[other] == true)
          Thread.yield();
   }

   public void leavingCS(int t) {
          flag[t] = false;
   }

   private volatile boolean[]
          flag = new boolean[2];
}
```

- Thread 0 and thread 1
- Possible endless loop:

  if context switch occurs after setting the flag, both threads wait for the other

4. Strict Alternation

```
public class StrictA extends baseME{

    public StrictA() {
        turn = TURN_0;
    }

    public void enteringCS(int t) {
        while (turn != t)
            Thread.yield();
    }

    public void leavingCS(int t) {
        turn = 1 - t;
    }

    private volatile int turn;
}
```

- `turn` keeps track which thread may enter its critical section
- After the first thread finishes, the second may enter, afterwards the first again
- However, if process one wants to enter and it is process two's turn?
- Solution, if all the processes are equally fast
- Violates the above condition: A thread outside its critical section blocks another thread

5. Peterson's Solution (I)

- Algorithm involves
  - an array, one element per thread, and
  - a flag
- First thread sets its array element and thus indicates interest to enter its CS
- `flag` is set to the other thread
- If the second thread also wants to enter its CS (array element is set), the first thread blocks
- If both threads call `enterCS` simultaneously, one thread overwrites `flag`, → the first thread proceeds and the second enters the CS afterwards
- No endless blocking!

Peterson's Solution (II)

```
public class PSol extends baseME {

  public PSol() {
    flag[0] = false;
    flag[1] = false;
    turn = TURN_0;
  }

 public void enteringCS(int t) {
  int other = 1 - t;
  flag[t] = true;
  turn = other;
  while (  (flag[other] == true) &&
           (turn == other) )
     Thread.yield();
 }

  public void leavingCS(int t) {
        flag[t] = false;
  }

  private volatile int turn;
  private volatile boolean[]
        flag = new boolean[2];
}
```

- This solution works!
- based on busy (active) wait: the while runs continuously (if not interrupted), or yield()

Busy Waiting (BW)

- All solutions presented so far used busy waiting to block processes
  - Does not really block the process but it just enters a loop
  - Wastes CPU time
  - Priority inversion problem (in (BW)
    - Two processes L (low priority) and H (high priority)
    - H is run whenever it is in ready state
    - While L is in its critical section, H becomes ready → context switch
    - H wants to enter critical section → busy waiting for L to leave
    - L is never scheduled to run → L never leaves critical section
    - H never progresses!
- BW can be advantageous in multiprocessor systems
  - For short delays
  - No context switch necessary

6. Semaphore

- Integer variable, managed by correct implementations of critical sections (in P() and V())

- Construct that does not need busy waiting

- Accessed only by two operations
  - **P()** – decrement semaphore (Dutch)
    - Process or thread blocks if semaphore will be negative
  - **V()** – increment semaphore

- P() and V() are executed indivisibly (only one thread or process at the time can modify semaphore by using P() or V())

Using Semaphore

- Counting semaphore (unrestricted values)
  - Can be used to protect a number of resources
  - Semaphore is initialised with the available number
- Binary semaphore (only values of 0 and 1)

```
Semaphore S; // initialized to 1
    P(S);
    CriticalSection();
    V(S);
```

- Associate a process queue with a semaphore
- Processes change into waiting state with P() (if negative)
- Processes are woken up from within V() (change state to ready)
- Control is with the CPU scheduler

Java Semaphores

```
public class Semaphore {

   public Semaphore() {
            value = 0;
    }
   public Semaphore(int v) {
            value = v;
    }

   public synchronized void P() { /* see next slide */ }
   public synchronized void V() { /* see next slide */ }

   private int value;
}
```

P() and V() operations

```
public synchronized void P() {
    while (value <= 0) {
      try {
        wait();
      }
      catch (InterruptedException e) { }
    }
    value --;
  }
```

```
public synchronized void V() {
    ++value;

    notify();
  }
```

Example using Semaphore

```
public class SemaphoreExample {

  public static void main(String args[]) {
    Semaphore sem = new Semaphore(1);  // get a semaphore & initialise 1
    Worker[] bees = new Worker[5];      // get 5 threads

    for (int i = 0; i < 5; i++)
      bees[i] = new Worker(sem);         // provide semaphore to threads

    for (int i = 0; i < 5; i++)
      bees[i].start();                   // start threads

  }  // end main
}  // end class
```

Example using Semaphore

```
public class Worker extends Thread {

    public Worker(Semaphore s) { sem = s;}        // constructor
    public void run() {
    while (true) {
                    sem.P();          // enter critical section, may block here
                    // in critical section
                    sem.V();          // leave cs and wakeup other threads
                    // out of critical section
          }  // end loop
    } // end run method

    private Semaphore sem;
}
```

Homework

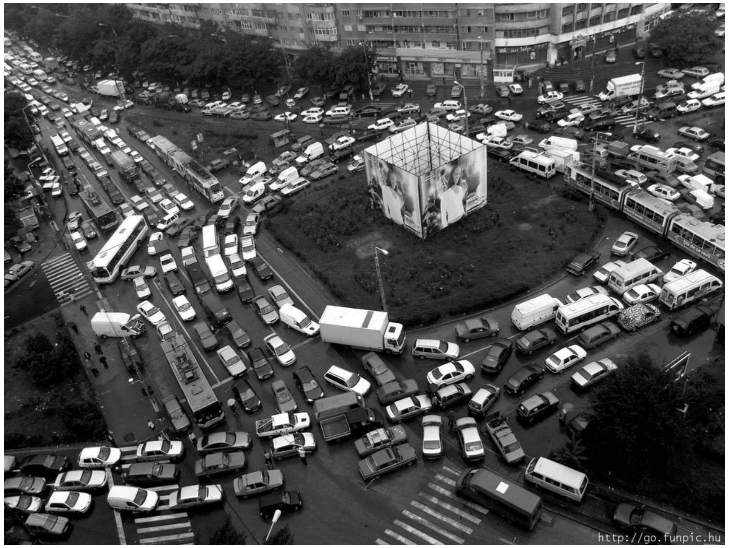
Use semaphore(s) to implement a rendezvous.

Deadlock with P() and V()

- **Deadlock** (of a set of processes)
  – two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes.

- Let *S* and *Q* be two semaphores initialised to 1

P_0	P_1
P(*S*);	P(*Q*);
P(*Q*);	P(*S*);
⋮	⋮
V(*S*);	V(*Q*);
V(*Q*)	V(*S*);



- **Starvation** (of one or more proceses)
  – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended. (LIFO queue)

7. Monitors

- Semaphores are error prone!
  - Correct order is essential
  - Errors are hard to detect, depend on particular execution sequence
    - Swap the order of P() and V()
    - Replace V() with P()
    - Omit either P() or V()
  - Consider major, large-scale software development projects
- Monitors are a high-level construct to prevent such errors

Monitors

- Monitor presents a set of programmer defined operations that provide automatic mutual exclusion
- Monitor type also contains variables to define the state of an instance of the monitor
- A monitor method can only access monitor internal data and formal parameters
- Local variables may only be accessed from within the monitor
- Monitor construct prohibits concurrent access to all methods defined within that monitor

Monitors

```
monitor Monitor-name
{
    integer i;        // variables
    condition c;      // condition variables
    public producer(…) {
            …
    }
    public consumer(…) {
            …
    }
}
```

Condition Variables

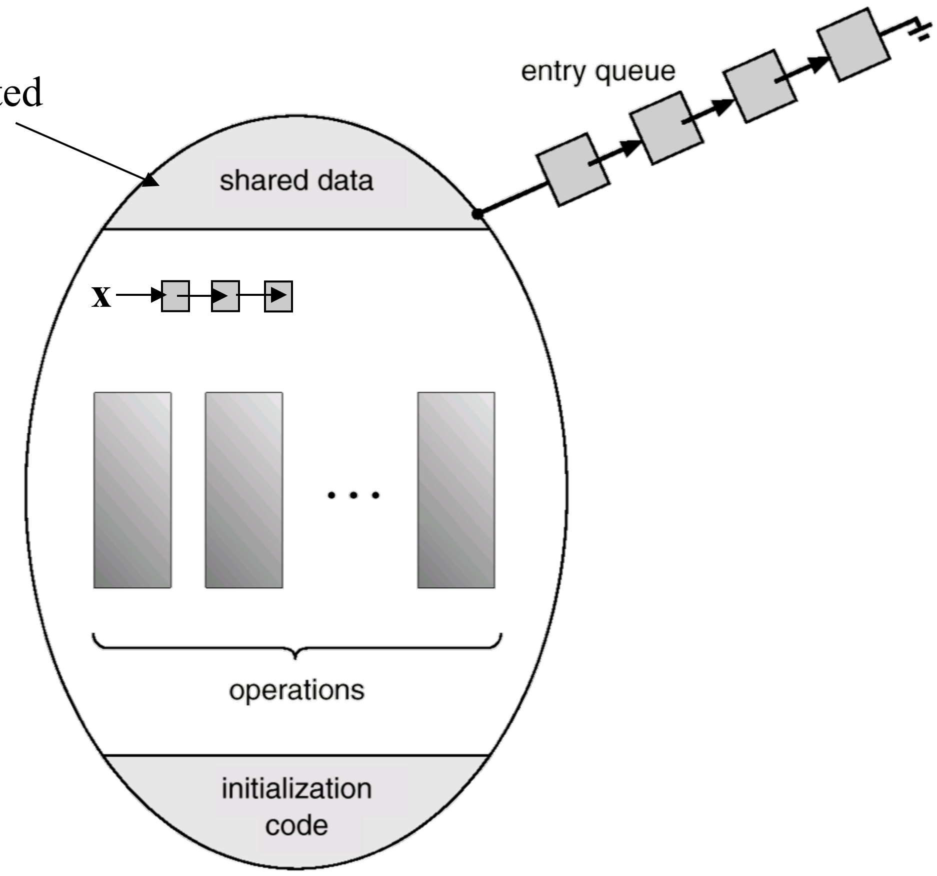
- Condition variables are used for user specific synchronisation

  (buffer full/empty) condition x, y;
- Operations wait() and signal() are defined

  x.wait() suspends the invoking thread until
  x.signal() is called by another thread
- Thread frees the monitor after blocking
- After signalling, a thread:
  - Signal-and-wait: signalling thread waits for other thread to finish in the monitor or to block on another condition
  - Signal-and-continue: signalling thread continues processing. The woken up thread continues afterwards.

Monitors

Queues associated with conditions



Message Passing (recap)

- Consider distributed systems without shared memory
  - Semaphores are too low level
  - Monitors are inapplicable
  - No information exchange possible between machines
- Message Passing!

Message Passing

- Implements two-way messages
  - Send(destination, message)
  - Receive(source, message)
- Implemented as system calls rather than language constructs (no shared memory)
  - Messages are buffered by the operating system
  - Usually provided by a library
- Receiver may block when there are no messages or return with an error code
- Issues
  - Messages may be lost by the network
  - Naming of processes
  - Authentication of processes
  - On the same machine: performance

Producer – Consumer Example

- ! No shared memory – possible implementation
- All messages are the same size
- Messages are buffered by the OS
- N messages are used (N elements in a buffer)
- Consumer starts by sending N empty messages to the producer
- Whenever the producer has an item to send, it takes an empty message, fills it, and sends it to the consumer
- Producer will be blocked if there are no empty messages waiting
- Consumer blocks if there are no filled messages waiting
- Zero buffer option possible

Producer-Consumer Problem

```
#define N 100                               /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                              /* message buffer */

    while (TRUE) {
        item = produce_item( );             /* generate something to put in buffer */
        receive(consumer, &m);              /* wait for an empty to arrive */
        build_message(&m, item);            /* construct a message to send */
        send(consumer, &m);                 /* send item to consumer */
    }
}

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

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);              /* get message containing item */
        item = extract_item(&m);            /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```