Lecture 2

The Shared Update Problem

Exercise 1

- What is the minimum?

```
private int counter = 0;
private final int rounds = 100000;

public process update
 ((int id = 0; id<2; id++)) {
   for(int i = 0; i<rounds; i++)
      counter++;
}
```

The Shared Update Problem

- Summary: Last time
  - Introduction to concurrency
  - Processes/threads in JR/Java
  - The shared update problem: mutex
- Today
  - Specifying atomic actions
  - Solving the shared update problem
    - Achieving mutex with shared variables
  - Introduction to a first programming language construct for synchronisation: *semaphores*

Mutual Exclusion

- Mutual exclusion
  - The property that only one process can execute in a given piece of code
- How can we achieve it?
  - Theory: possible with just shared variables
    - very inefficient at programming language level
    - but sometimes necessary in very low-level (HW)
    - good example to study concurrent behaviours
  - Practice: programming language features (semaphores, monitors, …)

Critical Section

- The airline reservation problem
  - Travel agents might run the following code:

```
void reserveSeat(Position p) {
    if (seat.free(p))
        seat.reserve(p);
}
```

  - and then issue a valid ticket for the seat at position p

Possible Run

Travel agent A

```
void reserveSeat(25J) {
   if (seat.free(25J))
      seat.reserve(25J);
}
```

Travel agent B

```
void reserveSeat(25J) {
   if (seat.free(25J))
      seat.reserve(25J);
}
```

Possible Run

Travel agent A

```
void reserveSeat(25J) {
   if (seat.free(25J))
      seat.reserve(25J);
}
```

Travel agent B

```
void reserveSeat(25J) {
   if (seat.free(25J))
      seat.reserve(25J);
}
```

Possible Run

Travel agent A

```
void reserveSeat(25J) {
    if (seat.free(25J))
        seat.reserve(25J);
}
```

Travel agent B

```
void reserveSeat(25J) {
    if (seat.free(25J))
        seat.reserve(25J);
}
```

Possible Run

Travel agent A

```
void reserveSeat(25J) {
   if (seat.free(25J))
      seat.reserve(25J);
}
```

Travel agent B

```
void reserveSeat(25J) {
   if (seat.free(25J))
      seat.reserve(25J);
}
```

Possible Run

Travel agent A

```
void reserveSeat(25J) {
   if (seat.free(25J))
      seat.reserve(25J);
}
```

Travel agent B

```
void reserveSeat(25J) {
   if (seat.free(25J))
      seat.reserve(25J);
}
```

Possible Run

Travel agent A

```
void reserveSeat(25J) {
    if (seat.free(25J))
        seat.reserve(25J);
}
```

Travel agent B

```
void reserveSeat(25J) {
    if (seat.free(25J))
        seat.reserve(25J);
}
```

Possible Run

Travel agent A

Travel agent B

Specifying Synchronisation

- We use a notation to specify atomic actions
  - Not part of JR
  - Purely for describing the desired behaviour of a program

<S> – statement S is executed atomically

<await (B) S> – execute <S>, starting only when B is true

Implementing `await`

- await statement is very expressive
  - Mutual exclusion
  - Conditional synchronisation
- Difficult to implement in general
- Though, some special cases are easy
  - await statement without body
    - `<await (B) ;>`
    - Sufficient for solving the shared update problem in low-level programming
  - Other interesting cases will come later

Airline Reservations

- The pieces of code that check the availability and reserve the seat access a shared resource
  - They are critical sections
  - we can specify the desired behaviour as:

```
void reserveSeat(Position p) {
    <if (seat.free(p))
        seat.reserve(p);>
}
```

Achieving Mutex

- Clever programming
- Hardware support (multiprocessor systems)
  - special atomic instructions
- Programming language support
  - Semaphores, locks
  - Monitors, …
- Avoid shared variables/critical sections
  - Use message passing

The Mutual Exclusion Problem

- General overview

```
process CS ((int i=0;i<N;i++)) {
   while (true) {
      Non-critical section
      Entry Protocol
      Critical Section
      Exit Protocol
      Non-critical section
   }
}
```

The Mutual Exclusion Problem

- Assumptions
  - No variables are shared between critical and non-critical sections and the protocol
  - The critical section always terminates
  - Read/Write operations are atomic (`x=1`)
  - Scheduler is weakly fair
    - A process waiting to execute `<await(B) S>` where `B` is constantly true, will eventually get the processor.

The Mutual Exclusion Problem

- Requirement 1: Mutex
  - At most one process at a time is in its critical section

```
process CS ((int i=0;i<N;i++)) {
   while (true) {
      Non-critical section
      Entry Protocol
       Critical Section
      Exit Protocol
      Non-critical section
   }
}
```

The Mutual Exclusion Problem

- Requirement 2: No deadlock/livelock
  - If both processes attempt to enter their critical section, one will succeed

```
process CS ((int i=0;i<N;i++)) {
   while (true) {
      Non-critical section
       Entry Protocol
      Critical Section
      Exit Protocol
      Non-critical section
   }
}
```

The Mutual Exclusion Problem

- Requirement 3: Eventual entry
  - A process attempting to enter its critical section will eventually succeed

```
process CS ((int i=0;i<N;i++)) {
   while (true) {
      Non-critical section
       Entry Protocol
      Critical Section
      Exit Protocol
      Non-critical section
   }
}
```

Attempt 1

- Use a variable `turn` to indicate who may enter next

```
int turn = 0;
process CS ((int i=0;i<2;i++)) {
   while (true) {
      //Non-critical section
      <await(turn==i) ;>
      //Critical Section
      turn = (i+1)%2;
   }
}
```

Attempt 1

- Implemented using *busy-wait (spin loop, spinning)*

```
int turn = 0;
process CS ((int i=0;i<2;i++)) {
   while (true) {
      //Non-critical section
      while (turn!=i) ;
      //Critical Section
      turn = (i+1)%2;
   }
}
```

Attempt 1 – Analysis

- Mutex
  - ok
- Deadlock
  - ok
- Starvation
  - What if non-critical section does not terminate?

Attempt 2

- Use a `flag` to indicate who has entered

```
private boolean flag[] = {false, false};
process CS ((int i=0;i<2;i++)) {
   other = (i+1)%2;
   while (true) {
      //Non-critical section
      <await (!flag[other]) ;>
      flag[i] = true;
      //Critical Section
      flag[i] = false;
   }
}
```

Attempt 2 – Analysis

- Mutex
  - no
- Deadlock
  - ok
- Starvation
  - ok

Attempt 3

- Use a `flag` to indicate who wants to enter

```
private boolean flag[] = {false, false};
process CS ((int i=0;i<2;i++)) {
   other = (i+1)%2;
   while (true) {
      //Non-critical section
      flag[i] = true;
      <await (!flag[other]) ;>
      //Critical Section
      flag[i] = false;
   }
}
```

Attempt 3 – Analysis

- Mutex
  - ok
- Deadlock
  - It can happen!
- Starvation
  - ok

1+3 = Peterson's algorithm

- flag+turn: I want to enter, after you

```
private int turn = 0;
private boolean flag[] = {false, false};
process CS ((int i=0;i<2;i++)) {
   other = (i+1)%2;
   while (true) {
      flag[i] = true;
      turn = other
      <await (!flag[other] || turn==i) ;>
      //Critical Section
      flag[i] = false;
}}
```

How do we know it works?

- It is not easy to show properly.
  - The general version (arbitrary n) is even worse
- Testing
  - Exponentially many traces
  - A given scheduler (implementation) may only explore a small number of traces
- Mathematical proof
  - See course “Software engineering using Formal methods”.
- Alternative algorithms explored in the book.

Complex Instructions

- We only assumed an atomic:
  - Read, and
  - Write
- Most modern hardware has larger atomic operations
  - Used to implement multiprocessor synchronisation at a lower level
    - operating systems
    - embedded systems

Right for the job?

- As a pure software solution to the problem
  - These algorithms are not practical
  - They all contain a busy-wait loop

```
while (!B) ;
```

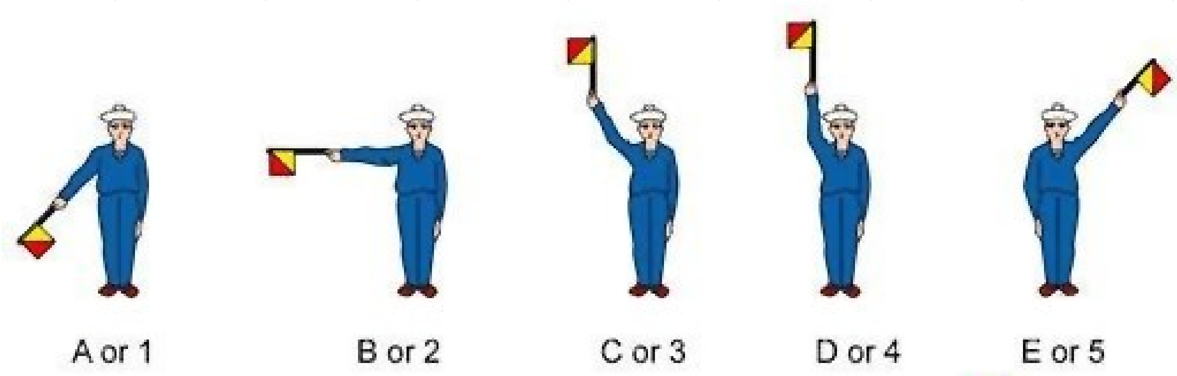
  - Consumes a great deal of processor resources and is very inefficient
- But often useful in low-level programming
  - OS
  - Embedded devices

Beyond busy waiting

- A more suitable solution would be as follows:
  - *Entry Protocol*: if Critical Section is busy then sleep, otherwise enter
  - *Exit Protocol*: if there are sleeping processes, wake one, otherwise mark the critical section as not busy
- Semaphores support this solution
  - and more

Semaphores – an overview

- First special construct for solving synchronisation problems



- Invented in the mid 60’s
  - Edsger Wybe Dijkstra [1930–2002]

Semaphore Specification

- An abstract datatype containing a nonnegative integer accessed by two atomic operations P and V

```
class Semaphore {

   private int sv;

   Semaphore(int init): <sv = init>
   P(s): <await (sv>0) sv = sv -1>
   V(s): <sv = sv + 1>
}
```

Semaphore Operation Names

- A short note on the names P and V
- P stands for passeren which means ”to pass”
- V stands for vrygeven which means ”to release”
- Dijkstra was Dutch

Critical Section – Semaphores

- JR has built in semaphores

```
sem mutex = 1;
process CS ((int i=0;i<2;i++)) {
   while (true) {
      //Non-critical section
      P(mutex);
      //Critical Section
      V(mutex);
   }
}
```

Critical Section – Semaphores

- Java has a library support
  - java.util.concurrent

```
Semaphore mutex = new Semaphore(1, true);

public void run() {
   while (true) {
      //Non-critical section
      mutex.acquireUninterruptibly();
      //Critical Section
      mutex.release();
   }
}
```

Critical Section – Semaphores

- Java: the more usual way

```
Semaphore mutex = new Semaphore(1);

public void run() {
   try {
      while (true) {
         //Non-critical section
         mutex.acquire();
         //Critical Section
         mutex.release();
   }} catch(InterruptedException e) {
}}
```

Binary Semaphores and Locks

- A semaphore which only ever takes on the values 0 and 1 is called a *binary semaphore*
- When a binary semaphore s is used for simple mutex:

```
P(mutex);
//Critical Section
V(mutex);
```

  - it is also referred to as a lock.
    - P(s) – “acquiring the lock”
    - V(s) – “releasing the lock”

Java Built-In Locks

- A lock is created for every object in Java
- To use this lock we employ the keyword `synchronized`

```
class MutexCounter {
   private int counter = 0;

   public synchronized void increment() {
           counter++;
   }
}
```

Java Built-In Locks

- Alternative to a synchronized method is a synchronized block
  - Less structured, but occasionally useful

```
class MutexCounter  {
   private int counter = 0;

   public void increment() {
      // lock this object
      synchronized (this) {
         counter++;
      }
}}
```

Liseberg Counter – Revisited

```
public void run() {
    try {
        for(int j = 0; j<100; j++) {
            Thread.sleep(…));
            System.out.println(
                Thread.currentThread().
                    getName()+" enters "+j);
            counter.increment();
        }
    }
    catch (InterruptedException e) {
    }
}
```

Liseberg Counter – Revisited

```
public Main() {
   Thread t1 = new Thread(this,"Process 1");
   Thread t2 = new Thread(this,"Process 2");

   t1.start();
   t2.start();

   try {
      t1.join();
      t2.join();
      System.out.println("Counter: "+counter);
   }
   catch (InterruptedException e) { }
}
```

Java Locks: Summary

- Each object has a lock
- Each lock has a queue of waiting threads
- The order of the queue is not specified
  - Could be implemented
    - FIFO
    - LIFO
    - etc.

Summary

- Today's lecture
  - Shared update using variables
  - Introduction to Semaphores
  - Locks in Java

- Next time
  - programming with semaphores: beyond locks

Real Life Deadlock