Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski
Lectures Overview

- Lecture 1: Introduction and Motivation
- Lecture 2: Classic Approaches to Concurrent Programming
  - Lecture 3: Objects and Concurrency
  - Lecture 4: From Concurrent to Distributed Programming
Outline

- Mutex
- Semaphore
- Monitor
- Rendez-vous
- But there is an **operating system**, typically for I/O
- The operating system (**kernel**) occupies part of the memory
- Tasks can execute OS code (through **system calls** or interruptions)
- The OS code may modify the OS memory
- Tasks can thus **interfere** with each other
Example 1: Counter

Consider concurrent programming with preemptive scheduling, **reentrant procedure**

Operations of concurrent invocations will be interleaved

Result if t1 and t2 execute `increase_counter()` concurrently?
Example 2: Printer Allocation

module console;
defines write;
    ... 
end console;

process p1;
begin 
    ... 
    console.write("No more space");
    ... 
end p1;

process p2;
begin
    ... 
    console.write("Disk full");
    ... 
end p2;
module critical_section;
defines enter, exit;
    ...
end critical_section;

process p1;
begin
    ...
    critical_section.enter();
    console.write("No more space");
    critical_section.exit();
    ...
end p1;

process p2;
begin
    ...
end p2;
- **Safety** (properties)
  - “Nothing bad ever happens”
  - Here: At any moment, no more than one task is in the CS, i.e., executing in between `enter()` and `exit()`

- **Liveness** (properties)
  - “Eventually something good happens”
  - Here:
    - If no task is in the CS, then a task may enter
    - A task willing to enter the CS eventually will
module critical_section;
defines enter, exit;
    var busy : boolean := false;

procedure enter();
begin
    while busy do end;
    busy := true
end enter;

procedure exit();
begin
    busy := false;
end exit;
end critical_section;
Mutex

- **(Mutual exclusion)**
  - busy and waiting are critical resources
  - lock() and release() are critical sections

- Implementation
  - Kernel support for lock() and release()
  - By masking interruptions
Using mutexes

process p1;
begin
  ...
  mutex.lock();
  console.write(“No more space”);
  mutex.release();
...
end p1;
module mutex;
defines lock, release;
    var busy : boolean := false;
    waiting;  // task queue

procedure lock();
begin
    if not busy do busy := true end;
    else add task to waiting;
    endif;
end lock;

procedure release();
begin
    if waiting not empty unleash task in front of waiting;
    else busy := false;
    endif;
end release;

end mutex;
**OO mutexes? mOOtexas?**

- **Mutexes in Java**
  - A lock can be acquired on (any) object
  - Tasks (threads) are synchronized on objects
  - Built-in mechanism `synchronized(...) {...}`

```java
Object mutex = new Object();
...
Synchronized(mutex) {
    System.out.println("Disk full");
}
```
Evaluation

- No explicit entering/exiting of critical section
  - Conditional exit?
- `wait()` and `notify()` / `notifyAll()`

```java
public class Mutex {
    private boolean busy := false;

    public synchronized void lock() {
        if (busy) wait();
        busy := true;
    }

    public synchronized void release() {
        busy := false;
        notify();
    }
}
```
Fairness

- Safety: “Nothing bad happens”
  - E.g., never two tasks in CS
- Liveness: “Eventually something good happens”
  - E.g., eventually a waiting task gets into CS

- How about several tasks waiting on lock()?
  - Which one is awaken after release()?

- Fairness
  - Assumptions about scheduling
  - Often required to prove liveness properties
  - Often interpreted as FIFO
Variants of Fairness

- Unconditional
  - Every unconditional atomic action that is eligible is executed eventually

- Weak
  - Unconditional + every conditional atomic action for which the condition is continuously true will eventually execute

- Strong
  - Unconditional + every conditional atomic action for which the condition is true infinitely often will eventually execute
**Starvation**
- A task keeps getting neglected by scheduler

**Further liveness issues**
- **Liveloop**: each task is waiting on (resources) another task (might take), none proceeds (“generous”)
  - Absence: system never enters a state with no *true* progress
- **Deadlock**: each task is waiting on (resources kept by) another task, none can proceed (“greedy”)
  - Absence: system never enters a state with no follow-up state
Semaphores

- Mutex is a very low-level mechanism
  - Only implements mutual exclusion
  - No inherent support for coordination, i.e., passing control, conditions

- Semaphore [Dijkstra’68] is a generalization of mutex
  - Semaphore’s state is represented by integer $x$
  - If $x \leq 0$, then $|x|$ represents the number of waiting tasks
Thus

- Consider a semaphore as an integer variable that is a resource counter

- The value of the semaphore is a number to indicate whether the resources are available or not
  - Decreased with $P()$ (acquiring)
  - Increased with $V()$ (releasing)

- Since the use of semaphores is to provide shared resources synchronization, the semaphore value must be “stored in” the kernel
module semaphore;
defines P, V;
  var val : int;
  waiting;  // task queue

procedure P(); // proberen
begin
  val := val - 1;
  if val < 0 then add task to waiting;
  endif;
end P;

procedure V(); // verhogen
begin
  val := val + 1;
  if not val > 0 then unleash task in front of waiting;
  endif;
end V;

end semaphore;
Mutex with Semaphore

mutex: semaphore init 1;

process p1;
begin
  ...
  mutex.P();
  console.write("No more space");
  mutex.V();
  ...
end p1;
Resource Allocation

- Suppose a critical resource is available in \( x \) copies
- Introduce a mutex for each of the \( x \) resources?
  - resource[1].lock() \( \lor \) resource[2].lock() \( \lor \) ...
- Two issues
  - At most \( x \) tasks executing in critical section
  - Attribute resource to task in critical section

```plaintext
process p1;
begin
  ...
  var nb: int;
  printers.req(nb);
  console[nb].write("No more space");
  printers.ret(nb);
  ...
end p1;
```
module printers;
defines req, ret;
var
  occupied: array of boolean // init false
  mutex: semaphore init 1;
  available: semaphore init x;

procedure req(var nb: int);
  begin
    available.P();
    mutex.P();
    nb such that occupied[nb] is false
    occupied[nb] := true;
    mutex.V();
  end req;

procedure ret(nb: int);
  begin
    mutex.P();
    occupied[nb] := false;
    mutex.V();
    available.V();
  end ret;

end printers;
Barrier Synchronization

- Explicit synchronization

- Semaphores used as signals
  - E.g., for events
  - Signalling semaphore(s) initialized to 0
  - Task signals event by $\text{V}(\ )$
  - Task waits for event by $\text{P}(\ )$

- For barriers (synchronize two or more tasks)
  - Use two semaphores
  - One to signal arrival at barrier
  - One for departure
Example: Workers

work1, work2: semaphore init 0;

procedure work1();
begin
    do something
    V(work1);
    P(work2);
end work1;

procedure work2();
begin
    do something
    V(work2);
    P(work1);
end work2;
done: semaphore init 0;
coordinate[1..n]: semaphore init 0;

procedure work(int worker);
begin
  do something
  V(done);
  P(coordinate[worker]);
end work;

procedure coordinate();
begin
  for [int i = 1..n] P(done); // all tasks completed
  for [int i = 1..n] V(coordinate[i]);
end coordinate;

- Do we need n semaphores for coordinating?
Split Binary Semaphore

- Use semaphores to signal data state rather than process state

- Two or more binary semaphores
  - At most one is on at same time
  - Initially only one is on

- In every execution path, a $P()$ is (eventually) followed by some $V()$ (possibly on different semaphore)
  - In between mutual exclusion

- Example: producer/consumer
Producers and Consumers

- Tasks asynchronously producing and consuming
  - Items must be buffered

```plaintext
full: semaphore init 0;
empty: semaphore init 1;

procedure produce(...);
  P(empty);
  deposit in buffer
  V(full);
end produce;

procedure consume(...);
  P(full);
  withdraw from buffer
  V(empty);
end consume;
```

- Do we need a mutex in between $P()$ s and $V()$ s?
Readers and Writers

- Generalization of mutual exclusion problem
  - Resources are either read or written
    - A single write at most, or
    - Several reads

- Variants according to priorities
  - Writers are prioritary
  - Readers are prioritary
  - FIFO
Consider at time \( t \)
- Critical section occupied by \( r1 \)
- Arrival order \( r2, w1, r3, w2, r4 \)

Reader priority
- \( (r1, r2, r3, r4), w1, w2 \)

Writer priority
- \( (r1, r2), w1, w2, r3, r4 \)

FIFO
- \( (r1, r2), w1, r3, w2, r4 \)
public class Sema {
    private int value;

    public Sema(int value) {
        this.value = value;
    }

    public synchronized void P() {
        while (value <= 0) wait();
        --value;
    }

    public synchronized void V() {
        ++value;
        notify();
    }
}

Semaphores in Java
- **java.util.concurrent.Semaphore**
  
  **As of Java 1.5**

```java
public class Semaphore {
    ...
    public Semaphore(int nb) {...}
    public Semaphore(int nb, boolean fair) {...}
    ...
    public void acquire() {...}
    public void release() {...}
    ...
    public void acquire(int nb) {...}
    public void release(int nb) {...}
    ...
}
```
Barrier Synch in Java

- `java.util.concurrent.CountDownLatch`
  - One-shot barrier

- `java.util.concurrent.CyclicBarrier`
  - Resettable barrier

- `join()` in `java.lang.Thread`
Semaphores are sometimes still rather unwieldy
  Cf. reader/writer

Monitors [Brinch-Hansen, Hoare ’74]
  (Yet) more high-level abstraction(?)

Incarnations
  Modula 2
  ...
  Ada ’95 (protected objects)
  Java (every java.lang.Object)
Principle

- Procedures of monitor are critical sections
  - All executed in mutual exclusion

- A queue for waiting tasks
  - Tasks are scheduled for waiting

- Signals/conditions are used to pass control between tasks within procedure(s) of the monitor
  - Cf. baton passing
  - Implement (boolean) guards
  - A queue per signal
  - Two primitives `wait()` and `signal()`
Overview

Chair of Software Engineering

Concurrent Object-Oriented Programming
producer/consumer

monitor produce_consume;
defines produce, consume;
var
    empty, full: signal;
    buffer: int := nil;

procedure produce(value: int);
begin
    if not buffer == nil
        empty.wait();
    endif;
    buffer := value;
    full.signal();
end produce;

procedure consume(var value: int);
begin
    if buffer == nil
        full.wait();
    endif;
    value := buffer;
    buffer := nil;
    empty.signal();
end consume;

end produce_consume;
Java Monitors

- Every object is potentially a monitor
- Methods to execute in mutual exclusion are tagged synchronized
- `wait()` and `notify() / notifyAll()` act on "inherent" signal
  - Caller must possess lock on corresponding object
Can an arbitrary object be used as signal?

Caution: must be synchronized, yet wait() only relinquishes “inmost lock”...
  - No nested synchronization!
  - Threads would need to “recollect” locks

Global design pattern
  - “Implement” signals as conditions
  - Loop on wait(), à-la
    - while (!condition) wait();
public class ReaderWriter {
        private int readers := 0;
        private int writers := 0;

        synchronized public start_read() {
            while (writers != 0)
                wait();
            readers := readers + 1;
            notify();
        }

        synchronized public end_read() {
            readers := readers - 1;
            notify();
        }

        synchronized public start_write() {
            while (readers + writers != 0)
                wait();
            writers := writers + 1;
        }

        synchronized public end_write() {
            writers := writers - 1;
            notify();
        }
    }
Flavors

- Java- vs Modula-style examples
  - One signal in Java vs unbound number in Modula

- Effects?
  - Priorities? Easier to implement?
  - How to implement FIFO?
Thus

- As of Java 1.5
  - `java.util.concurrent.locks`.

- Interfaces
  - **Lock**: finer control than `wait()` etc.
  - **Condition**: implement multiple wait queues/signals per monitor
  - **ReaderWriterLock**: inherent support for Reader/Writer
Two subtypes of monitors

- **Hoare** monitors ("signal and wait")
  - A signalling task is suspended if task waiting for signal
  - Awoken task executes
  - Signalling task resumes after awoken task finished

- **Mesa** monitors (Xerox - "signal and continue")
  - Signalling task continues until leaves monitor
  - "Awoken" task executes thereafter
In Fact...

- A semaphore can be implemented by a monitor
  - Two procedures \( \text{P}() \) and \( \text{V}() \) resp.
  - A single signal implements wait queue

- And inversely, cf. baton passing
  - One semaphore for mutual exclusion init 1
  - One semaphore for each signal init 0
    - Keep track of nb of waiting tasks per signal

- Semaphores vs mutex?
  - Cf. baton passing (mutex for sema init 1, mutex.lock() for sema init 0)
Monitors (cf. Java) get closer to “natural” interaction between objects
- Based on calls of “custom” procedures
  - With specific synchronization semantics
- Yet, signals/conditions within procedures
  - Procedures as logical “units” of computation are not indivisible

Rendez-vous: concurrent tasks “meet”
- By performing (completing) entries
  - Accepted at particular state of the callee
  - Upon prior fulfillment of conditions (guards)
task Int_Buffer is
  entry Put(I: in Integer);
  entry Get(I: out Integer);
end;

task body Int_Buffer is
  Val: Integer;
begin
  loop
    accept Put(I: in Integer) do
      Val := I;
      end Put;
    accept Get(I: out Integer) do
      I := Val;
      end Get;
  end loop;
end Int_Buffer;
Syntax and Semantics

- Header
  - Lists externally callable entries ("procedures")
    - Atomic units synchronizing caller and callee

- Body
  - Describes entries
  - Synchronization of calls to these, e.g.,
    - Order (accept ... end; ... accept ... end;)
    - Alternatives (select ... or ... ... end;)
    - Guards (when ... => accept ... end;)
    - Possibly loops
task Bounded_Int_Buffer is
  entry Put(I: in Integer);
  entry Get(I: out Integer);
end;

task body Bounded_Int_Buffer is
  A: Int_Array; X, Y: Index := 0;
  Count: Integer range 0..N := 0;
begin
  loop
    select
      when Count < N =>
        accept Put(I: in Integer) do
          A(X) := I;
        end Put;
        X := X + 1;
        Count := Count + 1;
      when Count > 0 =>
        accept Get(I: out Integer) do
          I := A(Y);
        end Get;
        Y := Y + 1;
        Count := Count - 1;
    end select;
  end loop;
end Bounded_Int_Buffer;
Note

- Synchronization of caller and callee on entry *only*
  - I.e., between `accept` and `end`
  - *After end, caller is released*

- Pattern
  - Release caller asap.
  - Use entries mainly for synchronization and passing arguments
  - Instructions following `accept` are performed consecutively, i.e., mutual exclusion
Evaluation

- Coming close to an object/procedure-based programming style
  - Guards define *what entries* are eligible for execution at *what point*
  - Defined outside of entries
    - State machine-like approach
Further

- Rich integration approach
  - Termination
    - From within: terminate
    - Outside: abort
  - Sleeping and (alternatives for) time outs (delay)
    - Towards real-time programming

- Client side can express alternatives
  - select
  - else
  - delay
  - ...

However

- Very explicit notion and handling of tasks
  - Tasks call entries on each other
    - Entries “limited” procedures
  - Objects vs tasks

- Vs Java
  - Procedures contain synchronization: -
    - while(...) wait() pattern
  - Threads are (active) objects: +
    - With behavior (run() / start())
Back to the Future: Monitors

- (Ada’95 has module-based type system, no classes as in Java)

- Specific, protected, types and objects
  - Combination of monitors and rendez-vous
  - Functions can only read data, yet execute in parallel
  - Procedures (also vs functions) are mutually exclusive

- Inherent implementation of readers/writers
Entries and Guards

- An entry encompasses guard
  - Cf. preconditions in SCOOP
  - “Replace” explicit signals/conditions within procedures

- Guards are reevaluated
  - Every time a procedure or entry terminates
  - Not after functions
  - Reevaluation of guards takes precedence over “new” calls
    - Scheduling can be redefined
protected type Semaphore
  (Start: Integer:= 1) is

  entry Secure;
  procedure Release;

private
  Count: Integer:= Start;
end Semaphore;

protected body Semaphore is

  entry Secure
    when Count > 0 is
    begin
      Count:= Count - 1;
    end Secure;

  procedure Release is
  begin
    Count:= Count + 1;
  end Release;

end Semaphore;
Differences with (Basic) Java

- Java uses condition variables
  - Ada uses conditional wait (no notify()!)
- Java allows non-synchronized methods
  - Ada enforces synchronization among all entries
- Java has one waiting queue per object
  - Ada has one waiting queue per entry
- Java’s queues are unordered
  - Ada queues are FIFO
- In Java, which thread is notify()’ed is unknown
  - In Ada, it is the head of the queue.
1. Mutex:
   Simple abstraction for mutual exclusion
2. Semaphore:
   More selective waiting, copies of critical resources
   “Baton passing” enables conditional waiting
3. Monitor:
   Custom procedures, inherent mutual exclusion
   Support for conditional waiting with signals
4. Monitors with guards, rendez-vous:
   Procedures as indivisible code units
   Inherent support for conditional waiting
Conclusions

- Several “classic” problems, e.g.,
  - Producer/consumer
  - Readers/writers
- Several “classic” tools, e.g.,
  - Semaphores
  - Monitors
- More or less integrated with objects
  - Simple object model so far...
  - Threads/tasks are still handled explicitly
  - Distribution?
  - Reuse?