Real-Time Software

Synchronization, Atomicity, Deadlocks

René Rydhof Hansen

8 October 2010

Today's Goals

- Shared variable communication and synchronisation
- Busy waiting
- Semaphores
- Conditional Critical Regions
- Monitors

Communication and Synchronisation with Shared Memory

Communication

- Message based
- Shared memory (shared variable)
  - Useful when tasks share physical memory
  - Multi-core systems
  - (Some) Multi-processor systems
  - Distributed systems... not so much.

Synchronisation

- "Full" communication not always needed
- Tasks may need to synchronise
  - Temporal ordering
  - Waiting for something to happen
  - Called condition synchronisation

Implementing Communication with Shared Memory

Example (Well-known PSS territory)

```
task T1;
   ...
   x := x + 1;   -- x is shared with T2
   ...

task T2;
   ...
   x := x + 1;   -- x is shared with T1
   ...
```

Problem?

Assumption

Atomicity is assumed at memory (word) level

Implementing Communication with Shared Memory

Example (Well-known PSS territory)

```
task T1;
   ...
   x := x + 1;   -- x is shared with T2
   ...

task T2;
   ...
   x := x + 1;   -- x is shared with T1
   ...
```

Problem? Non-atomic operations, race conditions!
Solution?

Assumption

Atomicity is assumed at memory (word) level

Implementing Communication with Shared Memory

Example (Well-known PSS territory)

```
task T1;
   ...
   x := x + 1;   -- x is shared with T2
   ...

task T2;
   ...
   x := x + 1;   -- x is shared with T1
   ...
```

Problem? Non-atomic operations, race conditions!
Solution? Mutual exclusion!

Assumption

Atomicity is assumed at memory (word) level

Mutual Exclusion

Definition (Critical Section)

Sequence of statements that must be executed atomically

Implementing mutual exclusion for critical sections

```
task P;
  loop
    ...
    entry protocol
      critical section
    exit protocol
    ...
  end;
```

What protocol?

Implementing a Mutual Exclusion Protocol

Peterson's Algorithm for Mutual Exclusion

```
task P1;
  loop
    flag1 := up;
    turn := 2;
    while flag2 = up and
          turn = 2 do
      null;
    end;
    -- CRITICAL SECTION
    flag1 := down;
    ...
```

```
task P2;
  loop
    flag2 := up;
    turn := 1;
    while flag1 = up and
          turn = 1 do
      null;
    end;
    -- CRITICAL SECTION
    flag2 := down;
    ...
```

- Hard to generalise to *n* processes
- Alternative: Decker's

Shared Memory Impl. of Condition Synchronisation

Example (Condition sync. using busy wait)

Ensuring that P1 waits for (signal from) P2.

```
task P1;
  ...
  while flag = down do
    null;
  end;
  ...
```

```
task P2;
  ...
  flag := up;
  ...
```

Problem?

Problems

- Inefficient
- Too error prone
- May lead to livelock

Shared Memory Impl. of Condition Synchronisation

Example (Condition sync. using busy wait)

Ensuring that P1 waits for (signal from) P2.

```
task P1;
  ...
  while flag = down do
    null;
  end;
  ...
```

```
task P2;
  ...
  flag := up;
  ...
```

Problem?

Problems

- Inefficient
- Too error prone
- May lead to livelock

Suspend and Resume

Example (Suspend and Resume)

Ensuring that P1 waits for (signal from) P2.

```
task P1;
  ...
  while flag = down do
    suspend;
  end;
  flag := down;
  ...
```

```
task P2;
  ...
  flag := up;
  resume P1;
  ...
```

Problem?

Suspend and Resume

Example (Suspend and Resume the WRONG way)

Ensuring that P1 waits for (signal from) P2.

```
task P1;
  ...
  while flag = down do
    suspend;
  end;
  flag := down;
  ...
```

```
task P2;
  ...
  flag := up;
  resume P1;
  ...
```

Problem? Race condition! Use special suspend protocol, e.g., two stage suspend, suspend objects (Ada)

Semaphores

Why?

- Simplify (and structure) synchronisation
- Avoid busy waits
- Simple mutual exclusion

Definition (Semaphore)

A semaphore is a non-negative integer with atomic increment and decrement operators associated:

1. `wait(S)` decrement `S` and suspend when it reaches zero
2. `signal(S)` increment `S`

Implementation of semaphores

- Often using hardware support: test and set, swap

Semaphores: The Downside

- Low level
- Error prone
- Brittle (one “small” error can take down the entire system)
- May lead to deadlock

Deadlock

- Prevention
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Avoidance
- Detection and recovery

Conditional Critical Regions

- A section of code that is executed in mutual exclusion
- Shared variables are grouped into named regions and tagged as resources
- Tasks cannot enter a region in which another task is executing
- Condition synchronisation is provided by guards
- Before a task can enter a critical region, the guard is evaluated (under mutual exclusion)

Conditional Critical Regions

Example (Bounded buffer using CCR)

```
resource buf : buffer;

task producer;
  loop
    region buf when buffer.size < N do
        ...
    end region
  end loop;
end producer

task consumer;
  loop
    region buf when buffer.size > 0 do
      -- take char from buffer etc
    end region
  end loop;
end consumer
```

Monitors

Monitors

- Monitors provide encapsulation and efficient condition synchronisation
- Critical regions are written as procedures encasulated in a single module
- Variables that must be accessed under mutual exclusion are hidden
- All method calls into module are executed under mutual exclusion
- Only operations are visible outside monitor
- What about condition synchronisation? Condition variables
  - Different semantics
  - Wait and signal operators
  - Wait blocks and releases hold on the monitor

Monitors

Example (Bounded buffer with monitors)

```
monitor buffer;
  export append, take;
    var (*declare necessary vars*)

    procedure append (I : integer);
      ...
    end;

    procedure take (var I : integer);
      ...
    end;
  begin
    (* initialisation *)
   end;
```

Monitors

Example (Bounded buffer with monitors)

```
procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait(spaceavailable);
    end if;
    BUF[top] := I;
    NumberInBuffer := NumberInBuffer+1;
    top := (top+1) mod size;
    signal(itemavailable)
  end append;
```

Monitors

Example (Bounded buffer with monitors)

```
procedure take (var I : integer);
  begin
    if NumberInBuffer = 0 then
      wait(itemavailable);
    end if;
    I := BUF[base];
    base := (base+1) mod size;
    NumberInBuffer := NumberInBuffer-1;
    signal(spaceavailable);
  end take;
```

Monitors: The Bad and The Ugly

- Bad: handling of condition synchronisation
- Bad: still too low level
- Ugly: brittle
- Ugly: internal structure hard to understand

Summary

Summary:

- Shared variable communication and synchronisation
- Busy waiting
- Semaphores
- Conditional Critical Regions
- Monitors
- Mutual exclusion (Peterson's algorithm)
- Suspend resume