T–79.4301 Parallel and Distributed Systems (4 ECTS)

Lecture 2

18th of September 2008

Keijo Heljanko

Keijo.Heljanko@tkk.fi

Common Flaws

Some common flaws in concurrent systems include:

- Deadly Embrace
- Circular Blocking
- Deadlock
- Starvation (livelock)
- Underspecification
- Overspecification

Deadly Embrace

A common problem in resource allocation

- Consider two callers A and B making a telephone call to each other simultaneously
- To connect a call two shared resources must be exclusively allocated: the caller's telephone line and the receiver's line
- It would be natural to use a protocol where we first allocate the caller's line and only then the receiver's line

Deadly Embrace (cnt.)

- However, if A and B call simultaneously each other, it can be the case that both A and B allocated their own lines but fail to allocate the receiver's line
- If there is no recovery mechanism in place, the system might deadlock

Deadly Embrace - Process A

In pseudocode process A might look like:

```
process A:
// Code removed
lock(line_A);
// Code removed
lock(line_B);
// Code removed
release(line_B);
// Code removed
release(line_A);
// Code removed
endprocess;
```


Deadly Embrace - Process B

In pseudocode process B might look like:

```
process B:
// Code removed
lock(line_B);
// Code removed
lock(line_A);
// Code removed
release(line_A);
// Code removed
release(line_B);
// Code removed
endprocess;
```


Deadly Embrace - Deadlock

An execution leading to a potential deadlock can be:

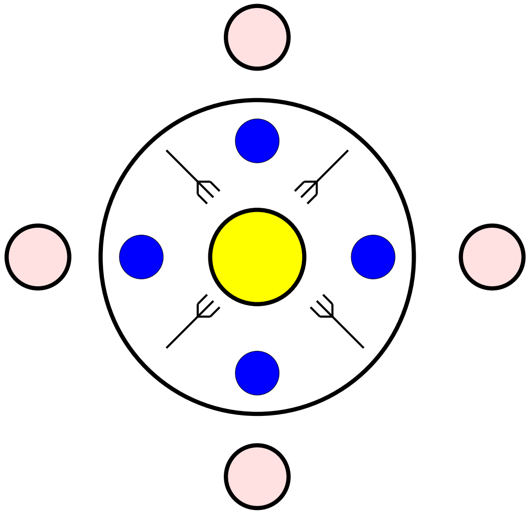
```
    Process A:          Process B:

    lock(line_A)
                        lock(line_B)

// Deadlock:
// Process A is waiting for line B
// Process B is waiting for line A
```


Circular Blocking

The deadly embrace extended over any number of processes. Classic academic example: Dining philosophers

- There are $n \geq 2$ philosophers sitting around a round table thinking
- Because all thinking is a tough job also the philosophers need to eat
- The dish prepared for them is particularly slippery spaghetti which requires two forks to be eaten
- Unfortunately there are only n forks available, distributed one between each pair of philosophers

Philosophers



Dining Philosophers

- The philosophers have agreed on a protocol to allocate the forks:
  - Think until hungry
  - Grab left fork
  - Grab right fork
  - Eat
  - Return right fork
  - Return left fork
  - Repeat from the beginning

Dining Philosophers (cnt.)

- Assume we have four philosophers: p(0),p(1),p(2),p(3)
- The forks are: f[0],f[1],f[2],f[3]
- The fork f[0] is to left of p(0) and to the right of p(3)
- It is now easy to see that the philosophers can all starve: Can you see how?

Dining Philosophers Pseudocode

```
#define left(i)  (f[(i)])
#define right(i)  (f[((i)+1)%n])
process p(i):
     while true do
          think();
          lock(left(i));
          lock(right(i));
          eat();
          release(right(i));
          release(left(i));
     enddo;
endprocess;
```

Dining Philosophers - Deadlock

A deadlock execution is:

```
p(0):        p(1):        p(2):        p(3):
lock(f[0])
             lock(f[1])
                          lock(f[2])
                                       lock(f[3])
```

After this:
p(0) is waiting for p(1) to release fork f[1]
p(1) is waiting for p(2) to release fork f[2]
p(2) is waiting for p(3) to release fork f[3]
p(3) is waiting for p(0) to release fork f[0]

Dining Philosophers vs. Real Life

The aim of the dining philosophers example is to:

- Show that circular blocking chains can be arbitrarily long
- Show that the possibility of stumbling on the deadlock by a randomly picked test run is extremely small: There is exactly one deadlocking state, and an exponential (in n) number of non-deadlocking states
- Dining philosophers was too easy: Often locking problems are much harder to spot just because the programs are larger and the locking is less structured

Deadlock

- Deadlocks are a common problem in distributed systems.
- As seen before deadlocks can occur from the use of blocking lock primitives.
- In a message passing system deadlocks might occur due to processes waiting for messages from one another in similar manner as processes are waiting for other processes to release locks
- Mixing priority based scheduling with locking is also known to easily lead to deadlocks

Starvation (livelock)

- Starvation (livelock) is a different problem. In it a part of the system is live and executing but other parts of the system are blocked indefinitely.
- Example: High priority process using a busy wait (spinlock) to wait for a low priority process to release a lock in an OS kernel. However, the low priority process is never given CPU time because the scheduler always picks the highest priority runnable task to be executed.

Deadlock and starvation will be treated more formally later.

Under- and Overspecification

Examples in a message passing (data-communications protocol) setting:

- Underspecification: A message arrives in a protocol implementation and there is no code to handle it (unexpected reception)
- Overspecification: There is code in a protocol implementation to cope with the reception of messages which are not possible in the protocol (dead code)

Non-concurrency Bugs

Of course your standard set of normal bugs not related to concurrency applies

- Incorrect control flow
- Incorrect data manipulation
- Wrong assumptions about the environment
- Null pointer exceptions
- Uninitialized data
- Array out of bounds errors
- Memory management problems (e.g., leaks, accessing freed memory)

The Spin Model Checker

- The model checker Spin was designed at Bell Labs by Gerard J. Holzmann (currently at NASA)
- It received the ACM Software System award in 2002. (Other winners: Unix, TeX, Smalltalk, Postscript, TCP/IP)
- Originally designed for data-communications protocol analysis
- The modelling language of Spin is called Promela
- The Spin Website has more material: `http://www.spinroot.com/`

Spin

Some of the reasons why Spin is successful

- Very efficient implementation (using generated C code)
- Contains advanced model checking algorithms, several of which are enabled by default
- A graphical user interface available (Xspin)
- Has been around for a while (15 years) and has been solidly supported by Holzmann

The Acronyms

- Spin = (Simple Promela Interpreter)
- Promela = (Protocol/Process Meta Language)

The Books

- The version 1.0 of Spin was published in Jan 1991:
  - Gerard J. Holzmann: Design and Validation of Computer Protocols, Prentice Hall, Nov 1990.
  - Book still available as PDF from:
    `http://spinroot.com/spin/Doc/Book91.html`
- A new book on Spin is much more up to date (v. 4.x):
  - Gerard J. Holzmann: The Spin Model Checker - Primer and Reference Manual, Addison-Wesley, Sep 2003, ISBN 0-321-22862-6.
  - For Book extras see:
    `http://spinroot.com/spin/Doc/Book_extras/index.html`

Promela

- The input language of the Spin model checker
- Control flow syntax inherited from Dijkstra's guarded command language
- Message passing primitives from Hoare's CSP language
- Syntax for data manipulation from Kernighan and Ritchie's C language

Modelling in Promela

This part is based on a nice Spin Beginners' Tutorial by Theo C. Ruys:
`http://spinroot.com/spin/Doc/SpinTutorial.pdf`
and The Spin Model Checker - Primer and Reference Manual

A Promela model consists of a set of processes communicating with each other through:

- Global variables
- Message queues of fixed capacity (called channels in Promela)
- Synchronization (rendezvous) on common actions

Promela Model

A Promela model consists of:

- Type declarations
- Channel declarations
- Variable declarations
- Process declarations
- Optionally: the `init` process (the “main()” process)

State of a Promela Model

The state of a Promela model consists of states of:

- Running processes (program counter)
- Data objects (global and local variables)
- Message channels

Finite State Models

Promela models are always finite state because

- All data objects have a bounded domain
- All message channels have a bounded capacity
- The number of running processes is limited (max 255 processes)
- The number of Promela statements in each process is finite - Also no procedure mechanism exists

Thus analysis of Promela models is in theory decidable.

In practice the available memory and time is the limit.

Processes

A process type (`proctype`) consists of

- Name - name of the proctype
- List of formal parameters - inputs given at start
- Local variable declarations
- Body - a sequence of statements: code of the procedure

Processes (Example)

In the following code the init process runs two instances of the you_run proctype

```
proctype you_run(byte x)
{
        printf("x = %d, pid = %d\n", x, _pid)
}

        /* leaving pids implicit */
init {
        run you_run(0);
        run you_run(1)
}
```


Processes (Example cnt.)

We can use spin for random simulation as follows:

```
$ spin ex1.pml
          x = 0, pid = 1
              x = 1, pid = 2
3 processes created
$ spin ex1.pml
              x = 1, pid = 2
          x = 0, pid = 1
3 processes created
$ spin ex1.pml
          x = 0, pid = 1
          x = 1, pid = 1
3 processes created
```


Processes (Example cnt.)

Note that Spin used indentation to show which process printed what. (You can use `spin -T` to disable this.) You can provide a seed to the Spin pseudorandom number generator as follows:

```
$ spin -n5 ex1.pml
          x = 0, pid = 1
              x = 1, pid = 2
3 processes created
```

In Promela the `init` process gets alway the pid `0` but the other processes dynamically allocate their pids

Process

- Is defined by `proctype` definition
- Executes concurrently with all other processes, the scheduling used is completely non-deterministic
- There may be several processes of the same type
- Local state:
  - Program counter
  - Contents of local variables

Creating Processes

- Processes are created using the `run` statement.
  To be precise: `run` expression (with a side-effect).
- Processes can also be created at the startup by adding `active[numprocs]` in front of a `proctype Foo()` to create `numprocs` instances of `proctype Foo`

Example:

```
active [2] proctype you_run()
{
        printf("my pid is: %d\n", _pid)
}
```


Creating Processes (cnt.)

Running the example:

```
$ spin ex2.pml
          my pid is: 1
      my pid is: 0
2 processes created

$ spin ex2.pml
      my pid is: 0
          my pid is: 1
2 processes created
```


Variables and Types

The Promela basic types are (sizes match those of C).

Type	Typical Range
`bit`	`0,1`
`bool`	*false*, *true*
`byte`	`0..255`
`chan`	`1..255`
`mtype`	`1..255`
`pid`	`0..255`
`short`	$-2^{15}..2^{15}-1$
`int`	$-2^{31}..2^{31}-1$
`unsigned`	$0..2^{32}-1$

Example Declarations

```
bit x, y;              /* two single bits, initially 0 */
bool turn = true;      /* boolean value, initially true */
byte a[12];       /* array of 12 bytes initialised to 0 */
short b[4] = 89;  /* array of 4, all initialised to 89 */
int cnt = 67;              /* integer initialised to 67 */
unsigned v : 5;            /* unsigned stored in 5 bits */
unsigned w : 3 = 5;    /* value range 0..7, initially 5 */
mtype n; /* uninitialised mtype (enumeration) variable */

chan in = [3] of {short, byte, bool}; /* message channel
with 3 messages capacity, messages have three fields   */
```

Mtype

The `mtype` (message type) keyword is a way of introducing enumerations in Spin.
Example:

```
mtype = { apple, pear, orange, banana };
mtype = { fruit, vegetables, cardboard };

init {
        mtype n = pear; /* initialise n to pear */

        printf("the value of n is %e\n", n)
}
```


Mtype (cnt.)

Running the example in Spin:

```
$ spin ex3.pml
      the value of n is pear
1 process created
```