CS3211 – Parallel & Concurrent Programming
Concurrency Concepts

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

Chapter 4 of "Principles of the Spin Model Checker" by Ben-Ari, **and**
http://spinroot.com/spin/Man/ (lot of online material, including a Manual)

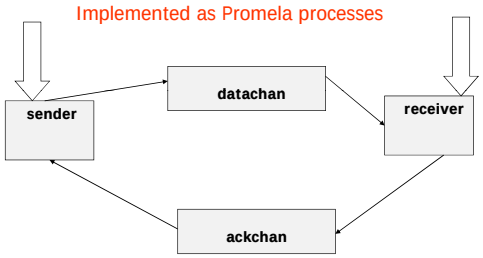
Today's and next lecture

- To discuss concurrency concepts
  - We use Promela, a low-level modeling language.
- We then move to concurrent programming
  - We then use Java, a multi-threaded programming language.
- So far …
  - Promela as a language
  - Basic constructs: if, do, …

Example: ABP

- Alternating Bit Protocol
  - Reliable channel communication between sender and receiver.
  - Exchanging msg and ack.
  - Channels are lossy
  - Attach a bit with each msg/ack.
  - Proceed with next message if the received bit matches your expectation.
- chan datachan = [2] of { bit };
- chan ackchan = [2] of { bit };

ABP architecture



Sender

```
active proctype Sender()
{   bit out, in;
    do
    :: datachan!out ->
            ackchan?in;
            if
            :: in == out
                 -> out = 1- out;
            :: else fi
    od
}
```

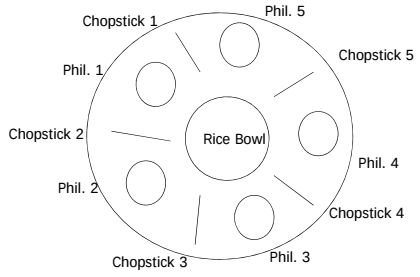
Receiver

```
active proctype Receiver()
{   bit in ;
    do
    :: datachan?in -> ackchan!in
    :: timeout -> ackchan!in
    od
}
```


Timeouts

- Special feature of the language
  - Time independent feature.
    - Do not specify a time as if you are programming.
  - True if and only if there are no executable statements in any of the currently active processes.
  - True modeling of deadlocks in concurrent systems (and the resultant recovery).

Something to think about



The task – Exercise for you

- Design the philosopher and the fork processes in Promela such that
  - A philosopher eats only when he has both chopsticks – left and right
  - No two philosophers hold the same chopstick simultaneously
  - No deadlock (circular wait among processes) and
  - No starvation (literally so!)
- How will the processes communicate?
  - Any of the mechanisms learnt today

An approach that does not work

```
Modeling philosopher[i]
While (true){
    wait for fork[i];
    wait for fork[i+1];
    eat
    release fork[i];
    release fork[i+1];
}
```

Deadlock – each philosopher may pick up their left fork first, and keep on waiting for the right fork.

Asymmetric solution

- The first four philosophers execute the same code, but the fifth philosopher executes the following.

```
Loop forever
        think
        wait(fork[0])
        wait(fork[4])
        eat
        release(fork[0])
        release(fork[4])
End loop
```


Today's lecture

- Dining Philosophers' problem
- To discuss concurrency concepts
  - We use Promela, a low-level modeling language.
- We then move to concurrent programming
  - We then use Java, a multi-threaded programming language.

Concurrent processes

- Promela supports multiple communicating processes in a description.
  - Default concurrency semantics: Asynchronous composition
  - At any point, only one process is active.
  - Also known as interleaving semantics.

Interleavings

```
byte n = 0;

active proctype P(){
   n = 1;
   printf("Process P, n =%d\n", n)
}

active proctype Q(){
   n = 2;
   printf("Process Q, n =%d\n", n)
}
```

Proc.	n	Stmt.	n	Output
P	0	n = 1	1	
P	1	printf	1	n = 1 printed
Q	1	n = 2	2	
Q	2	printf	2	n = 2 printed

Proc.	n	Stmt.	n	Output
P	0	n= 1	1	
Q	1	n = 2	2	
P	2	printf	2	n = 2 printed
Q	2	printf	2	n=2 printed

Sequential Consistency

- What are all the allowed execs. of a concurrent program?
  - Each process must proceed in program order.
  - Statements from across different processes may be arbitrarily interleaved.
- All executions satisfying the above two properties make the exec. model called sequential consistency.
  - Intuitive understanding of concurrent program execution by the programmer.
  - How many executions are there for the concurrent program given in the previous slide?

Atomicity

- Statements in Promela are atomic.

```
if
:: a!= 0 ->  c = b/a
:: a ==0 -> c = b
fi
```

a is global (shared across processes, including this one)

Is division by zero impossible?

No, because another process may set
a= 0
between the evaluation of a !=0 and the execution of c = b/a

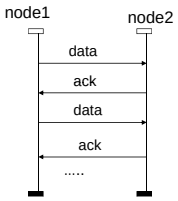
Concurrent Execution

```
chan data, ack = [1] of bit;

proctype node1() {
        do
        :: data!1;
        :: ack?1;
        od
        }

proctype node2() {
        do
        :: ack!1;
        :: data?1;
        od
        }

init{ atomic{
        run node1(); run node2();
        }
}
```



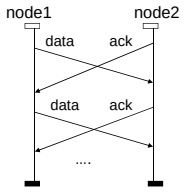
Concurrent Execution

```
chan data, ack = [1] of bit;

proctype node1() {
        do
        :: data!1;
        :: ack?1;
        od
        }

proctype node2() {
        do
        :: ack!1;
        :: data?1;
        od
        }

init{ atomic{
        run node1(); run node2();
        }
}
```



Interference across processes

- Main challenge in concurrent programming
  - Interleaving semantics across processes, and
  - Sharing of variables across processes.
- Different interleavings modify shared variables differently
  - Causing various unpredictable interference across processes.

A simple example to show interference

```
byte n = 0;

active proctype P() {
    byte temp;
    temp = n + 1;
    n = temp;
    printf("P, %d", n)
}

active proctype Q() {
    byte temp;
    temp = n + 1;
    n = temp;
    printf("Q, %d", n)
}
```

Proc.	Stmt.	n	P:temp	Q:temp	Output
P	temp=n+1	0	0	0	
Q	temp=n+1	0	1	0	
P	n=temp	0	1	1	
Q	n=temp	1	1	1	
P	printf("P..")	1	1	1	P,1
Q	printf("Q..")	1	1	1	Q,1

Incrementing n twice we expect 2,
Yet the terminal values are 1.

More on interference

```
byte n = 0;

active proctype P() {
    byte temp;
    atomic{
        temp = n + 1; n = temp;
    }
    printf("P, %d", n)
}

active proctype Q() {
    byte temp;
    atomic{
        temp = n + 1; n = temp;
    }
    printf("Q, %d", n)
}
```

What are the possible pairs of printed values in the two processes?

Even more on interference

```
byte n;

proctype P(byte id;) {
    byte temp;
    atomic{ temp = n + 1; n = temp;}
    printf("Process P%d, n = %d\n", id, n)
}

init{
    n = 0;
    atomic{ run P(1); run P(2) }
    (_nr_pr == 1) -> printf("final value of n=%d", n)
}
```

What are the possible terminal values of n?

Synchronization

- Processes implicitly communicate via shared variables.
- However, for other reasons
  - Processes may need to explicitly synchronize.
- What reasons?
  - e.g. Mutually exclusive access to shared variables.
- How to synchronize?
  - Busy waiting
  - Acquiring and releasing locks.

Busy waiting

```
bool  wantP = false, wantQ = false;

active proctype P() {
    do
    :: printf("noncritical section\n");
       wantP = true;
       do
       :: !wantQ -> break;
       :: else -> skip
       od;
       printf("Crit. Section P\n");
       wantP = false
    od
}

active proctype Q() {
    do
    :: printf("noncritical section\n");
       wantQ = true;
       do
       :: !wantP -> break;
       :: else -> skip
       od;
       printf("Crit. Section Q\n");
       wantQ = false
    od
}
```


Busy waiting

```
bool  wantP = false, wantQ = false;

active proctype P() {
    do
    :: printf("noncritical section\n");
       wantP = true;
       do
       :: !wantQ -> break;
       od;
       printf("Crit. Section P\n");
       wantP = false
    od
}

active proctype Q() {
    do
    :: printf("noncritical section\n");
       wantQ = true;
       do
       :: !wantP -> break;
       od;
       printf("Crit. Section Q\n");
       wantP = false
    od
}
```

What is the effect of removing the else choice ?

Busy waiting

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("noncritical section\n");
     wantP = true;
     !wantQ;
     printf("Crit. Section P\n");
     wantP = false
  od
}

active proctype Q() {
  do
  :: printf("noncritical section\n");
     wantQ = true;
     !wantP;
     printf("Crit. Section Q\n");
     wantQ = false
  od
}
```

No need to loop, the process blocks if condition is false

Busy waiting

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("noncritical section\n");
     wantP = true;
     !wantQ;
     printf("critical section\n");
     wantP = false
  od
}

active proctype Q() {
  do
  :: printf("noncritical section\n");
     wantQ = true;
     !wantP;
     printf("critical section\n");
     wantP = false
  od
}
```

Mutual exclusion is preserved, what about deadlock and non-starvation?

Common "mistakes"

- Non mutually exclusive access to shared variables.
  - "Unexpected" states due to certain sequences of statements involving multiple processes.
- Deadlock
  - Reach a state where no process can progress.
- Starvation
  - A process wanting to access a shared variable (say entering a critical section) should be able to do so "eventually"
    - In finite time
    - In bounded time.

Deadlock scenario

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: wantP = true;
     !wantQ;
     wantP = false
  od
}

active proctype Q() {
  do
  :: wantQ = true;
     !wantP;
     wantQ = false
  od
}
```

wantP = true; wantQ = true;
Both processes are now blocked.

Busy waiting

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("noncritical section\n");
     atomic{
     !wantQ; wantP = true; }
     printf("critical section\n");
     wantP = false
  od
}

active proctype Q() {
  do
  :: printf("noncritical section\n");
     atomic{
     !wantP; wantQ = true; }
     printf("critical section\n");
     wantQ = false
  od
}
```


Synchronization

- Processes implicitly communicate via shared variables.
- However, for other reasons
  - Processes may need to explicitly synchronize.
- What reasons?
  - e.g. Mutually exclusive access to shared variables.
- How to synchronize?
  - Busy waiting
  - Acquiring and releasing locks.

Locking

```
byte sem = 1;

active proctype P() {
    do
    :: printf("Noncritical section P\n");
       atomic{ sem > 0; sem--; }
       printf("Critical section P\n");
       sem++;
    od
}

active proctype Q() {
    do
    :: printf("Noncritical section Q\n");
       atomic{ sem > 0; sem--; }
       printf("Critical section Q\n");
       sem++;
    od
}
```

When we program in Java, we do not program in the protocols for ensuring mutual exclusion. Instead, we assume a locking mechanism and program the non-critical / critical sections.

Locking

```
byte sem = 1;

active proctype P() {
    do
    :: printf("Noncritical section P\n");
       atomic{ sem == 1; sem--; }
       printf("Critical section P\n");
       sem++;
    od
}

init{
    atomic{ run P(); run P(); }
}
```

`atomic{ sem == 1; sem--; }` Implementation of lock acquire
`sem++` Implementation of lock release

Communication among processes

- Shared variables
  - (same as concurrent prog. in Java)
- Message Passing
  - (we will later use MPI for parallel programming)
  - At the application level, the issue of locking does not arise.
    - Seemingly, no shared variables !
    - So, we do not need to worry about this now!
  - However, in reality, the message buffers or channels are shared global variables and the programmer will need some mechanism to mutually ensure exclusive access
    - Two processes cannot read/write to the channel at the same time.

Client Server Example

```
chan request = [0] of { byte} ;

active proctype Server() {
   byte client;
      do
      :: request? client->
            printf("Client %d\n", client)
      od
}

active proctype Client(){
      request! _pid;
}
```

Send and receive of message is a handshake.
Both sender and receiver block until the other process is ready.

Client Server Example

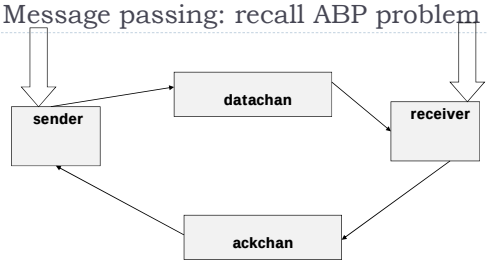
```
chan request = [1] of { byte} ;

active proctype Server() {
   byte client;
      do
      :: request? client->
            printf("Client %d\n", client)
      od
}

active proctype Client(){
      request! _pid;
}
```

The receiver blocks until the message is sent.

Message passing: recall ABP problem



Alternating Bit Protocol
Reliable channel communication between sender and receiver.
Exchanging msg and ack. Channels are lossy .
Attach a bit with each msg/ack. Proceed with next message if the received bit matches your expectation.

ABP modeling: Once again

```
active proctype Sender()
{   bit out, in;
    do
    :: datachan!out ->
            ackchan?in;
            if
            :: in == out
                 -> out = 1- out;
            :: else fi
    od
}
```

```
active proctype Receiver()
{   bit in ;
    do
    :: datachan?in -> ackchan!in
    :: timeout -> ackchan!in
    od
}
```

```
chan datachan = [2] of { bit };
chan ackchan = [2] of { bit };
```


How Message Passing occurs in real-life

- Interrupt-driven communication
  - An interrupt happens to the CPU, whenever data is ready to be read.
    - To ensure mutually exclusive access of message buffers, disable interrupts while servicing the current interrupt.
    - Not captured at the application level send receive we are studying!
- Or, the CPU polls (via certain sensors) at regular intervals to check whether data is available
  - Check whether data is available on the channel and then perform receive action, popularly known as polling.
  - Instead of being blocked at request?client as if the server checks periodically if the client has sent its data.

Let us finish with a real-life situation

- July 4, 1997
  - NASA's Pathfinder landed on Mars.
  - Tremendous engineering feat.
  - Hard to design the control software with concurrency and priority driven scheduling of threads.
  - The SpaceRover would lose contact with earth in unpredictable moments.

Mars PathFinder Problem

"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once"." …

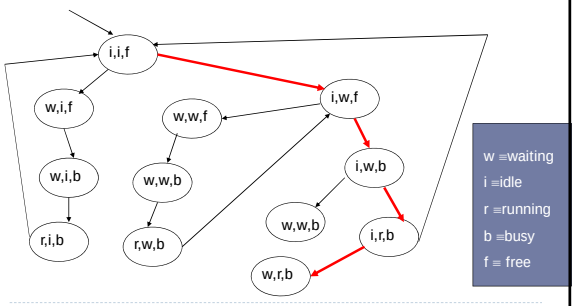
Essence of the problem in our modeling language

```
mtype = { free, busy, idle, waiting, running };
mtype H = idle;   mtype L = idle; mtype mutex = free;
```

```
active  proctype high();
{end:  do
       :: H = waiting;
          atomic { mutex == free ->
                   mutex = busy };
          H = running;
          atomic{ H=idle; mutex=free }
       od
}
```

```
active proctype low() provided (H == idle)
{ end: do
       :: L = waiting;
          atomic{ mutex== free->
                  mutex = busy};
          L = running;
          atomic{ L=idle; mutex = free }
   od
}
```


State Space Graph



Deadlock

- Counterexample
  - Low priority thread acquires lock
  - High priority thread starts
  - Low priority process cannot be scheduled
  - High priority thread blocked on lock
- Actual error was a bit more complex with three threads of three different priorities
  - Timer went off with such a deadlock resulting in a system reset and loss of transmitted data.

The actual problem

- "Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft."
  - A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes)."
  - The meteorological data gathering task ran as an infrequent, low priority thread, … When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
  - The spacecraft also contained a communications task that ran with medium priority."

☞High priority: retrieval of data from shared memory
Medium priority: communications task
Low priority: thread collecting meteorological data

The actual problem

- "Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset."

☞High priority: retrieval of data from shared memory
Medium priority: communications task
Low priority: thread collecting meteorological data

Priority Inversion

- In a real-life concurrent system
  - Concurrently executing processes are assigned priorities.
  - If 2 processes are ready, the higher priority process is allowed to execute (non-preemptive execution).
    - A higher priority process can delay a lower priority process.
    - The reverse situation should not be allowed
  - [Such decisions are enforced by the scheduler in OS or JVM]
- How priority inversion may occur
  - A lower priority process acquires the lock on a shared var.
  - The higher priority process appears later, tries to acquire the same lock, but has to wait until the lock is released by the lower priority process.

Summary

- Concurrent processes (threads in Java)
- Interleaving semantics
  - Asynchronous composition
  - Scheduled by a OS/JVM scheduler in practice.
- Communication among processes
  - Shared variables (same as concurrent prog. in Java)
  - Message Passing (we will later use MPI for parallel programming)
- Explicit Synchronization
  - Busy Waiting
  - Locks – acquire and release.

Now …

- Demo of Promela usage (by Ju Lei)
- Use the SPIN tool, Promela is its front-end.
- http://spinroot.com/spin/whatispin.html
- SPIN is actually a checker (which checks all interleavings of a program), but we primarily use it as a programming environment to understand concurrency concepts, in our CS3211 module.