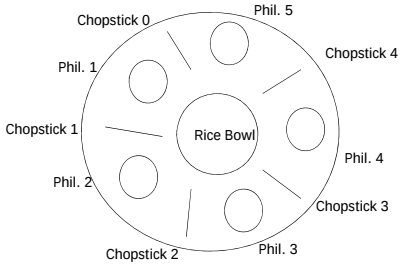
CS3211 – Parallel & Concurrent Programming
Concurrency Concepts

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

Additional optional Session on Sat, January 19.

Something to think about



The task – Exercise for you

- Design the philosopher processes 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 so far

An approach that does not work

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

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

Asymmetric solution

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

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


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

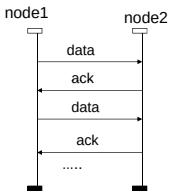
Concurrent Execution

```
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();
        }
}
```



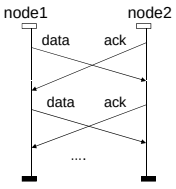
Concurrent Execution

```
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		
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 ?

Semantically equivalent?

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
}
```

Is starvation possible?

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.