Concurrency - 2

- **Buffered communication**
- **Monitors - a higher level concept than semaphores**
- **Concurrency mechanisms in PLs**
- **Message passing**

More Ada Syntax

- **Task syntax**

procedure go_to_movie **is**
  task find_seats;
  task buy_popcorn**;**
  task body find_seats **is** **… end;**
  task body buy_popcorn **is … end;**
  begin
    watch_movie;-- ***both tasks start executing here***
  end; -- ***Ada runtime system waits for all tasks to complete here***

- **Static model of concurrency**

Buffer Synchronization

```
task buffer
    entry enter (x: in char);
    entry remove( x: out char);
body{ declare and initialize private queue Q;
    loop{ select
        when (Q not full) =>
            accept enter(v);
                add v to back of Q;
            end enter;
        or when (Q not empty) =>
            accept remove(v)
                remove front of Q into v;
            end remove;
        end select}
    end loop}
    }
```

```
task producer
body{
    c: char;
    loop{
        get(c);
        c := process(c);
        buffer.enter(c);
        }
}
```

```
task consumer
body{
    d: char;
    loop{
        buffer.remove(d);
        put (chew (d));
        }
}
```


To use buffer solution

```
procedure main{
    task buffer{…};
begin   c: consumer; //both processes ;launched at procedure elaboration time
        p: producer; //both refer to global task buffer{}
…
end main;
```

What happens if…
- •producer and consumer alternate?
- •producer is faster?
- •consumer is faster?
- •have more than one consumer: c1, c2: consumer;
- •have more than one producer: p1, p2: producer:

Buffer with Semaphores

- **Use binary semaphore to implement critical section on a global queue. Assume that *not full* and *not empty* are manipulated by the queue procedure**
- **This next attempt is buggy; can you see the problem?**
  - **Lesson: semaphores are low-level and difficult to program correctly**


```
task type binary-semaphore
      entry p;
      entry v;
body{
      loop{ accept p;
               accept v;
            }
}
critical : new binary-semaphore;
declare and initialize global queue Q;
startup the consumer and producer.
```

```
task consumer
body{d: char;
      loop{
            critical.p;
            if not empty remove d from Q;
            critical.v;
            put(chew(d));
            }
}
```

```
task producer
body{ c: char;
      loop{
            get(c);
            c := process(c);
            critical.p;
            if not full, add c to Q;
            critical.v;
            }
}
```

What's possibly wrong here?

Another Example

```
ok,fin : new binary-semaphore;
ok := 0; fin := 1;
procedure producer
{ while (there is more input) do
            {fin.p;
        →   {write rec to buffer}
            ok.v;}
}
```

```
procedure consumer
{ while (true) do
            {ok.p;
        →   {read rec from buffer}
            fin.v;}
}
```

Here ok is 1 when there is something to read in the buffer. so consumer has input. fin is 1 when the buffer should be overwritten with new input. so producer needs to write.

Example of ensuring an alternating access to a shared resource with 2 binary semaphores.


```
task PRODCON is
    entry GIVE (C: in CHARACTER)
    entry TAKE (D: out CHARACTER)
end;
task body PRODCON is
    LIMIT: constant INTEGER := 100;
    POOL: array (1 .. LIMIT) of CHARACTER;
    INP,OUTP: INTEGER range 1 .. LIMIT := 1;
    COUNT: INTEGER range 0 .. LIMIT := 0;
    begin
      loop  select
            when COUNT < LIMIT =>
             accept GIVE (C: in CHARACTER) do
                POOL(INP) := C;
             end;
             INP := INP mod LIMIT + 1;
            COUNT := COUNT + 1;
           or  when COUNT > 0
           accept TAKE (D: out CHARACTER) do
              D := POOL (OUTP);
           end;
            OUTP := OUTP mod LIMIT + 1;
             COUNT := COUNT -1;
           end select;
      end loop;
  end PRODCON
```

specification of task

implementation of task

add a character

remove a character

Input: "a b c"	inp	outp	count
initially	1	1	0
give("a")	2		1
give("b")	3		2
take(d) "a"		2	1
give("c")	4		2
take(d) "b"		3	1
take(d) "c"		4	0

Real Ada example from Horwitz, Fundamentals of PLs, 1984 CS Press

Monitors

- **Module with operations, internal state and condition variable(s)**
- **Only one operation can be active at a time**
  - **If a thread calls a busy monitor, then the thread waits**
  - **Monitor operation can suspend itself by *wait*ing on a condition variable**
  - **Monitor operation may *signal* a condition variable**
- **Equal in power to semaphores but less error prone**

Monitors

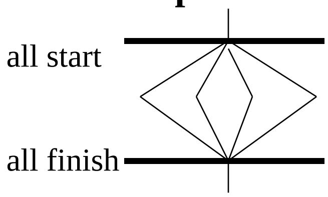
```
monitor  buffer {
 private
        Queue  Q;
        f(illable): condition
        e(mptyable) : condition
 public
    entry add(v){
        if Q is full { wait e}
        enter v in Q;
        signal   f
        } //executed atomically!

    entry remove (v){
        if Q is empty {wait f}
        remove v from Q;
        signal   e
        } //executed atomically!
```

```
task producer
body{
    c: char;
    loop{
        get(c);
        c := process(c);
        buffer.add(c);
        }
}
```

```
task consumer
body{
    d: char;
    loop{
        buffer.remove(d);
        put (chew (d));
        }
}
```

Concurrency Mechanisms in PLs

- *Co-begin, co-end*
  - **Used to indicate a set of statements to be performed in parallel**
  - **Usually assumed to have access to same stack frame**
  - **Found in PLs SR, Algol68, Occam**
  - **Task parallelism**




```
x = 5;
par begin
  P(2),
  y = x+2,
  x = 3,
  Q(33,'a')
par end
```


Concurrency Mechs, cont.

- *Parallel loop*
  - **Define a loop with all its iterations executing in parallel**
  - **For safety, can't have any dependences between loop iterations**
    - **E.g., if we had `a[j] = a[j-1]` then the calculation on iteration j depends on iteration j-1.**
    - **Parallelizing FORTRAN compilers do analysis to check for this type of condition before transforming programs to this form**
    - **Data parallelism**

```
forAll(i=5 to 10)
  a[i]= 3*b[i];
  a[i+1]= 2+a[i];
end forAll;
```

Concurrency Mechs, cont.

- **Task launch at procedure 'elaboration' or call**
  - Tasks in Ada and SR are created when declaring procedure is invoked
  - Procedure can't finish until all tasks are completed (barrier synchronization like co-begin, co-end)
  - task parallelism

```
procedure P{
  task T  is … end T;
  begin --P
    ...
  end --P
```


Concurrency Mechs, cont.

- **Explicit fork/join - explicit, executable thread creation**
  - Threads are objects that are created dynamically anywhere in the executing program
  - Can create arbitrary patterns of concurrency
  - Fork creates thread; join allows thread to wait for previously created thread
  - In Ada (as a type), Modula-3, Java, SR

```
class myThread extends Thread {…}
…
myThread first = new myThread();
```

Concurrency Mechs, cont.

- **Remote procedure call (implicit receipt)**
  - Idea of migrating a computation to another processor
  - Need to gather arguments and code and transfer
  - Then execute code on other processor and return results back to original machine
  - Execution is taking place in another address space from the thread creator

Message Passing

- **Distributed systems communication**
- **Naming communication partners**
  - Explicit process naming (1 to 1 communication)
  - Port naming (receiver has named ports to which senders can send messages; n to 1 communication)
  - Channel naming (both senders and receivers name channels for communic; n to n communication)

Message Passing

- **Sending information**
  - **– Problem: how much may this block the caller**
  - **– No-wait send: sender blocks for no more than a small bounded amount of time; messages are copied by runtime mechanism which is responsible for delivery**
  - **– Synchronization send: Sender waits until message is received**
  - **– Remote-invocation send: Sender waits until receives reply**

Message Passing

- **Receiving information**
  - **– Explicit receive: e.g., Ada accept**
  - **– Implicit receive: new thread is created to deal with the receive**