Concurrency

- What is concurrent programming?
- Problems of concurrent programming
  - *Liveness*:
  - *Safety*
- Models of concurrency
  - *shared memory/ dsitributed memory (message passing)*
- Issues*: communication, synchronization, definition*
- 3 examples: Unix pipes, Co-routines, rendezvous
- Concurrent programming techniques
- Survey of some PL features of concurrency

1

Motivation for (seemingly) parallel computation

- Different speeds for different computer components
  - I/O much slower than CPU -- data channels
- Different speeds between humans and computers in interactive systems
- When problem structure is naturally parallel:
  - discrete event simulation
  - web-page display: once layout of a web page is known, displaying different pictures, etc. can be done independently; and user can do other things while waiting for whole page

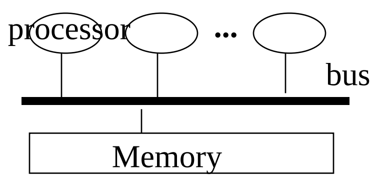
  "task parallelism"
- Numerical computations on huge arrays

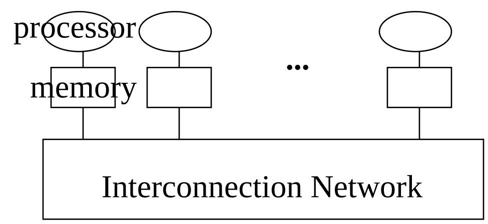
  A = B + C is same as A[j]=B[j]+C[j] <--- all can be done independently
  - "data parallelism"
- Multiprocessor hardware

2

Concurrent Programming

- **Allows multiple threads of computation at same time**
- **Two general models**
  - ***Shared memory (SM)***
  - ***Distributed memory (message passing) (DM)***





Concurrent Execution - terminology

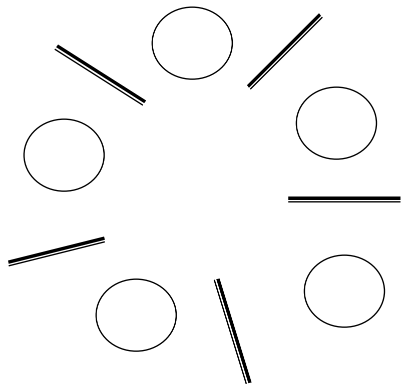
- **At hardware level, "in parallel" means operations overlapping in time**
- **In software, "concurrently" means operations that are *potentially* (but need not be) executed in parallel**

- ***Process*** **- a sequential computation with its own thread of control**
  - ***Event*** **-- atomic action (uninterrupted)**
  - ***Thread*** **of a process -- sequence of events**

- **Problems of concurrency:**
  - **Liveness: threads progress reasonably (e.g., no "deadlock")**
  - **Safety: getting the "right" answer (e.g., no "race conditions")**

a) LIVENESS: e.g. Dining Philosophers Problem

Philosophers eat
and talk at dinner.
To eat, a philosopher
must use 2 forks;
however, if her
neighbor is eating,
she cannot eat.
To think, a philosopher
puts down both her forks



Philosopher: process; fork:resource. Competition for resources.
Questions about algorithm:

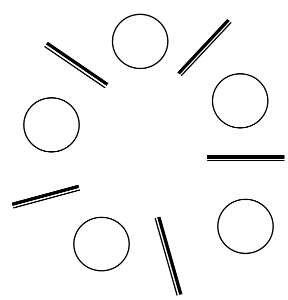
- fairness (can anyone starve?)
- can anyone eat?

5

CS314 © AB/BGR

Dining Philosophers-2

***Deadlock*: a chain of dependencies in which one process depends on a resource held by another process**



Each philosopher:

loop: pickup fork on right; (lock resource)
pickup fork on left;
eat for a while;
release forks; (unlock resource)
think for a while;
end loop;

May result in "pickup fork on right and wait for fork on left".

6

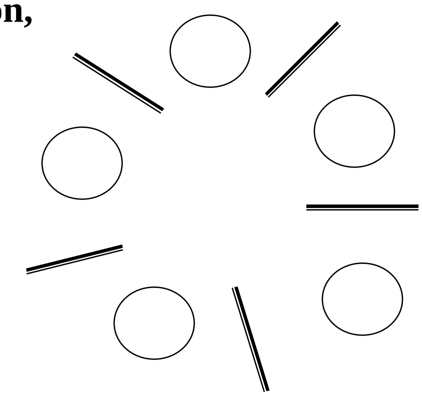
CS314 © AB/BGR

Dining Philosophers - 3

***Livelock*: continuing execution, but without progress**

if all philosophers do:

pickup left fork;
release left fork;
pickup right fork;
release right fork;
…



7

Dining Philosophers - 4

- ***Fairness:*** **any process that wishes to execute can do so in a finite amount of time**
  - **So every philosopher should get a chance to eat in a fair algorithm**

8

b) SAFETY

Define concurrency as *Interleaving of threads*: possible orderings that maintain relative order of events within one thread

{ a; b}   {x; y; z}

a x b y z ***interleaving preserves relative order***
a b x y z ***of events in any particular thread***
a x y b z , etc.

CS314 © AB/BGR

Safety problems

- **Two processes P and Q**
  - **P = { x=x+1; }**
  - **Q= { x=x+2; }**
  - **Consider each statement as an atomic event.**
  - **Execute P and Q concurrently: with interleaving, order does not matter - effect is add 3 to x**
- **Two processes P' and Q'**
  - **P' = { t = load(x); store(x , t + 1);}**
  - **Q' = {s = load(x); store(x , s + 2);}**
  - **/* P and Q translated into assembler; event are assembler ops */**
  - **IF we desire the same effect as running P and Q (in either order)**
  - **– Some interleavings ok:**
    - **t=load(x); store(x,t+1); s=load(x); store(x,s+2);**
  - **– Others do not produce expected final result:**
    - **t=load(x); s=load(x); x=s+2; store(x,t+1);**
    - ***/*at end, x is incremented by 1 only */***

CS314 © AB/BGR

Ensuring Safety with interleaving

- ***Mutual exclusion*: many processes share a resource (e.g., variable) but only 1 allowed to "use" it at a time.**
  - **To ensure safety, program for a process includes statements for acquiring and releasing resources appropriately**
- ***Critical section*: section of code that must be executed as if it is atomic (usually involves shared data)**

  e.g., [t=load(x); store(x,t+1);]
  - **Each thread executes its critical section completely before another thread can enter its own critical section (including the one containing access to the same shared data --> solves mutual exclusion).**
  - **An interleaving is considered *safe* if the events in every designated critical section are executed atomically/contiguously.**
  - **To ensure safety, programs mark critical sections**
- ***GOAL: allow as much concurrency as possible***

  e.g.

  R = { y = 7; t = load(x); store(x , t + 1);}

  S = {s = load(x); store(x , s + 2); z= 3;}

Alternate Safety Definition

- ***Database serializability***
  - **Two processes T1 and T2 *execute serially* if 1st process executes completely before the other one begins**
  - **Any serial execution of the processes is considered to produce a correct result : T1;T2 or T2;T1**
  - ***Serializablity* criterion: an interleaving of the steps of T1 and T2 is considered *safe* if its final effect is the same as that of some serial execution of the processes**

Serializable Executions

P {t:= x; x:= t+1;}		Q {u :=x; y:= 2*u;}	
P, Q	*t:= x;*	x == 0	
	x:= t+1;	x==1	
	u:=x;		safe
	y:= 2*u;	y==2 and x==1	
Q, P	u:=x;	x==0	
	y := 2*u;	y==0	
	t:= x;		safe
	x:= t+1;	x==1 and y==0	
??	u:= x;	x==0	?? is considered safe because results mirror Q,P
	t:= x;		
	y:= 2*u;	y==0	
	x:= t+1;	x==1 and y==0	

13

CS314 © AB/BGR

Nonserializable Executions

• *P { t:= x; x:= t+1;}*		Q {u:=x; x:= u+2;}	
P,Q	*t:= x;*	x==0, t==0	
	x:=t+1;	x==1	
	u:= x;	u==1 •	safe
	x:= u+2;	x==3 •	
Q,P	u:= x;	x==0,u==0	
	x:= u+2;	x==2	
	t:=x;	t==2 •	safe
	x:=t+1;	x==3 •	
??	*t:=x;*	x==0,t==0	
	u:=x;	u==0 •	Not serializable, because outcome is not P,Q nor Q,P
	x:=u+2;	x==2	
	x:=t+1;	x==1 •	

14

CS314 © AB/BGR

Concurrent Programming PL Issues

- *Process description*
- *Thread creation/destruction*
- ***Communication:*** **relating one thread to another in terms of exchange of data**
  - **DM: send/receive <info> ("messages")**
  - **SM: shared variable access**
- ***Synchronization:*** **relating order of events in one thread to another (exchange of control information)**
  - **DM:often provided implicitly by wait for message**
  - **SM: usually programmed explicitly**

1. Unix Pipes

Processes connected through pipes

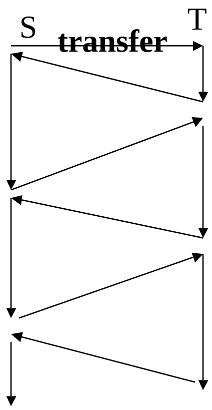
e.g., ***"give file names containing*** **march*****, one screen at a time"***

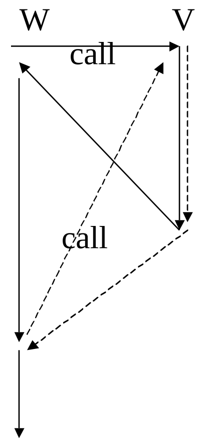
ls * | grep `march` | more

- ***according to definition*****: each operation creates as output a file, which is provided as input to next operation**
- **BUT: operations could be processes, connected by pipes streaming values from one to the other; process waits when pipe is empty, proceeds *concurrently with preceding one***
- **communication: *streams* of ascii chars grouped into lines**
- **implicit synchronization: process waits for next value in its input stream, if not yet available**

2. Coroutines (explicitly serial)

e.g., S reads lines, finds next word in line of text;
T processes a word *depending on context.*





Coroutine flow of control between S and T: control always returns to where it last left off *in both S and T*.(both have contexts)

Procedure flow of control between W and V; *V is fully executed*; then control returns to W, where it last left of.

Coroutines (cont'd)

Pseudo-concurrent:

- **Synchronization: explicit** `transfer` **statement**
- **Communication: globals** or **<u>ref</u> parameters** in call
- **Possible implementation: closures, "cactus stacks"**

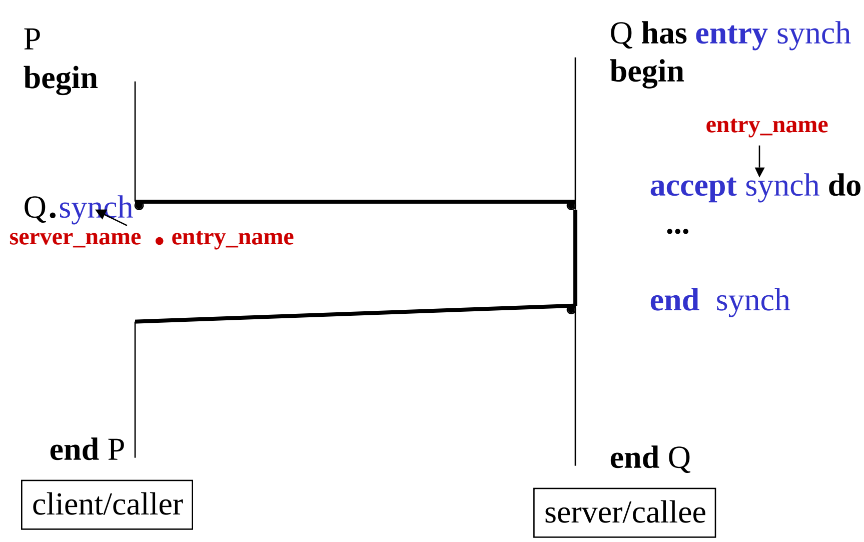
e.g.,

```
Coroutine from_to_by(from, to, by:int; ref j: int; ref done: bool; caller:coroutine)
    j := from
    done:= (from  >= to)
    detach  //wait till someone resumes (transfers back)
    loop
        { j += by
          done := (j >= to)
          transfer(caller)  //transfer  control to caller
        }
end from_to_by
```

Iterator <from,from+by, from+by+by,...> as a coroutine. **"Yields" via reference parameter j; "terminates" via done;**

3. Rendezvous

- **2 threads, C and S**
- **Synchronization:**
  - ***one thread ('client'/C) offers a hand to S;***
  - ***the other thread ('server'/S) `accepts` hand (from anyone);***
  - ***when hands meet, shake ("rendezvous" - meeting)***
  - **Whichever one arrives first at the rendezvous has to wait for the other one.**
  - **Server may accept from multiple clients**
- **Communication : allow in/out parameters in "handshake" at rendezvous**
- **Mutual exclusion is enforced during rendezvous**
  - **Body of `accept` clause acts as critical section**



Ada Syntax

- *entry call*

  <server_name>•<entry_name>

- *entry*

```
accept <entry_name>(args)
 do
   { sequence of statements,
    executed without interruption}
end <entry_name>
```

- **non-deterministic choice of entries**

```
select
     accept Entr1 do
      …
      end Entr1;
   or
      accept Entr2(param) do
      …
      end Entr2;
end select
```

but only those are considered which are ready for rendezvous; so server blocks only if neither Entr1 or Entr2 are called at this point.

More Ada

- **Even more general: *guarded entries*:**

```
select when expr1 => accept X do
                     …
                     end X;
    or when expr2 => accept Y do
                     …
                     end Y;
end select;
```

  - ***all* guards evaluated when select is entered; an alternative is “open” if its guard is true**
  - **act as regular select, but only on the open alternatives**
    - **(if all alternatives are closed, error)**
  - **when blocked, look for new entries opening up**

An example: producer/consumer problem

- ***Producer*** **gets pieces of data (at some rate) and "massages" them, before passing them on**
- ***Consumer*** **"chews" the received data and then spits it out**
- **Each act at their own rate**
- **To allow producer to move faster (or ...), "bounded queue" may be available to hold values not yet chewed**

- **Problems to watch for:**
  - **safety: "dropped" input;**
  - **no concurrency**
  - **deadlock**

Bad Solution with no global variable

```
task producer
body{
    c: char;
    loop{
        get(c);
        c := massage(c);
        g := c}
```

```
global
variable
  g
```

```
task consumer
body{
    d: char;
    loop{
        d := g
        d :=chew (d);
        put(d);
        }
}
```

Problem:
producer can race ahead and overwrite a value before consumer gets to it:

g:= c1
g:=c2
d:=g

Good solution -- with a buffer *task*

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

```
task type producer
body{
      c: char;
      loop{
            get(c);
            c := massage(c);
            pool.add(c);
            }
}
```

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


CS314 © AB/BGR

Putting it all together (in Ada)

```
procedure main {
  task pool { ... };     //single process named buffer  launched at elaboration
begin
  c: consumer;   //proces c of task type consumer  launched  at elaboration;
     refers to global task called pool
  p: producer;
do
  null;
end;
```

***Trace*:**

- alternating producer and consumer
- faster producer
- faster consumer
- multiple consumers `c,c2,c3:consumer;`
- multiple producers `p,pA,:producer;`

CS314 © AB/BGR