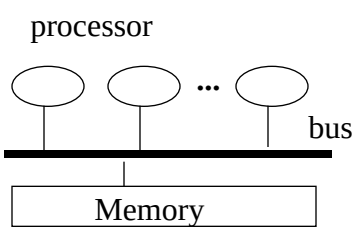
Concurrency

- **What is concurrent programming?**
- **Dining philosophers**
  - **Deadlock, livelock, fairness**
  - **Mutual exclusion**
- **Rendezvous in Ada**
  - **Ada syntax**

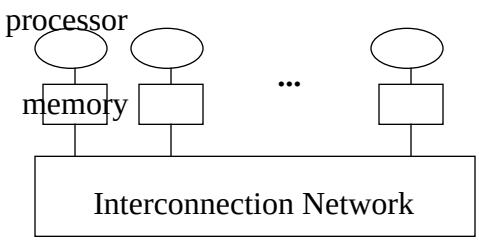
Concurrent Programming

- **Allows multiple threads of computation at same time**
- **Two general models of architecture**
  - **Shared memory and distributed memory**

Shared memory model



Distributed memory model (message passing)



Concurrent Programming

- **Hardware - *in parallel* means operations overlap in time performed**
- ***Concurrent* source operations means they can be, but need not be, executed in parallel**
  - **Potential for parallelism**
- ***Process* - a sequential computation with its own thread of control**
  - **Communication through shared variables or explicit messages**
- ***Event* - atomic action (uninterruptible)**
- ***Thread* of a process - sequence of events**

Concurrent Programming

- **Key issues**
  - **How concurrent processes synchronize and communicate with other processes?**
  - **Synchronization relates one thread to another in terms of exchange of control information**
  - **Communication usually implies an exchange of data**
  - **Unix pipes are examples of implicit synchronization. processes connected by stream of data**
    - **e.g., ls -lg * | more**

Concurrency as Interleaving

- ***Interleaving of threads*** **- possible orderings that maintain relative order of events within one thread**

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

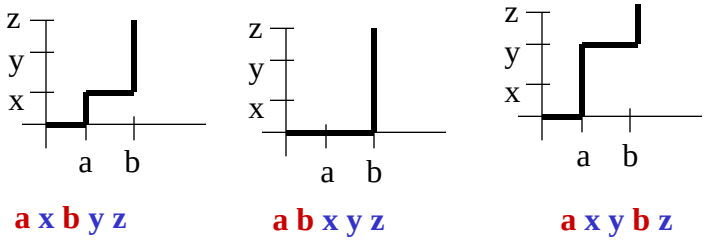
  a x b y z ***interleaving preserves relative order***

  a b x y z ***of events in any particular thread***

  a x y b z , etc.

Interleavings

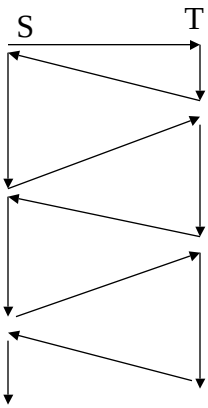
Geometric portrayal of interleavings



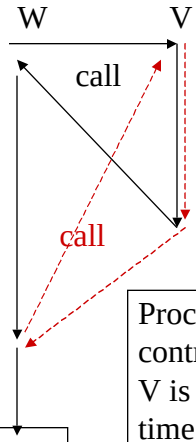
a x b y z **a b x y z** **a x y b z**

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

Coroutines vs Procedure Calls



Coroutine flow of control between S and T; communication always returns to where it last left off.



Procedure call flow of control between W and V; V is fully executed every time its called from W.

Coroutines

- **Represented by a closure that changes each time it runs (i.e., entry point and environment is updated)**
- **Used for discrete event simulation**
- **Provided by Simula, Modula-2**
- **Scott 8.6 commands**
  - ***detach* - creates coroutine object to which control can be later *transferr*ed and returns a reference to this coroutine to the caller**
  - ***transfer(param)* - saves the current coroutine (with program counter) and resumes the coroutine specified as *param***
  - ***resume(param)* - causes execution of coroutine param to start again**

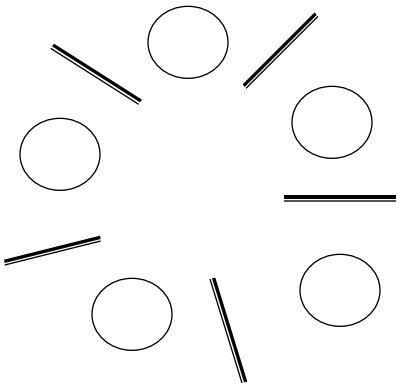
Iterator example exerpt, Scott p 477

```
Coroutine from_to_by(from, to, by:int; ref j: int; ref done: bool;
    caller:coroutine)
j := from
if by > 0 then { done:= (from  >= to)
                 detach
                 loop
                  { j += by
                    done := (j <= to)
                    transfer(caller)  //yield j
                   }
                }
end from_to_by
```

added small syntax changes for clarity

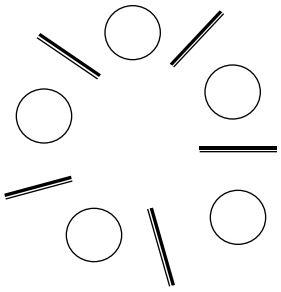
Dining Philosophers Problem

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



Dining Philosophers

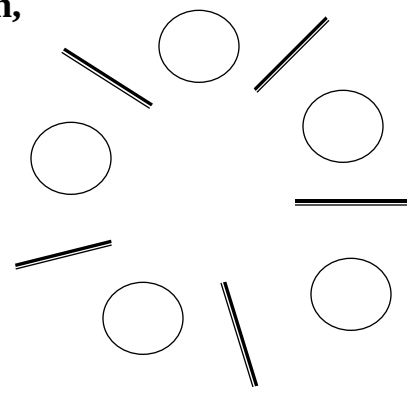
***Deadlock*: a chain of dependences in which one process depends on a resource held by another process**



Each philosopher:
loop: pickup fork on right; (lock resource)
pickup fork on left;
eat;
release forks; (unlock resource)
think;
end loop;
Results in "pickup fork on right and wait for fork on left".

Dining Philosophers

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



if all philosophers do:
pickup left fork;
release left fork;
pickup right fork;
release right fork;
…

Dining Philosophers

- ***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**
  - **Example with 2 philosophers**
    - **What if one philosopher reaches for fork on her left while other reaches for fork on her right (i.e., the same fork). then they both reach for the fork on the other side. This prevents deadlock by ordering requests for resources.** ***ordered resource usage***

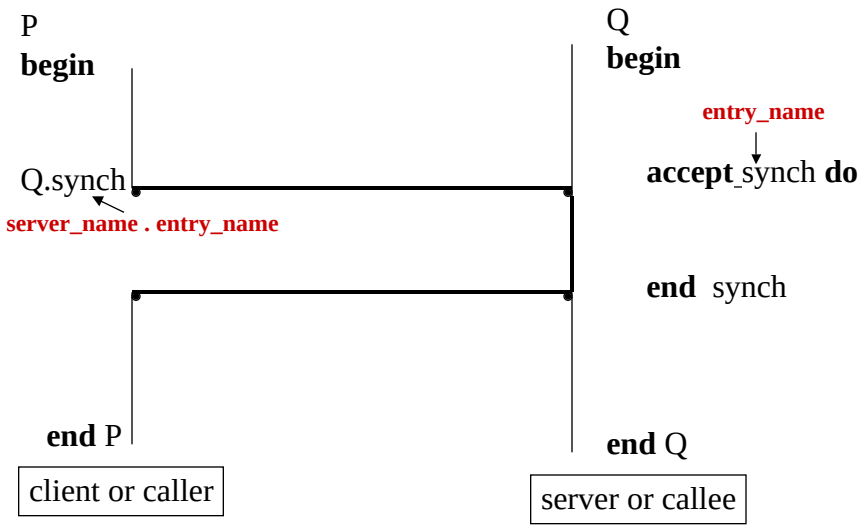
Shared Data

- ***Mutual exclusion*****: many processes share a resource, but only 1 can use it at a time.**
- ***Critical section*****: section of code that must be executed as if it is atomic (usually involves shared data)**
  - **Each thread executes its critical section completely before another thread can enter its related critical section (containing access to the same shared data).**
  - **Must make sure that 2 threads can never access shared data at the same time, one to read and one to write, or both to write.**

Ada Concurrency

- **Rendezvous mechanism**
  - **Symmetric, 1st thread to arrive has to wait for 2nd thread**
  - **Mutual exclusion is enforced during rendezvous**
    - **Body of accept clause acts as critical section**
  - **Handshake communication**
    - **Either client waits for server to answer OR**
    - **Server waits for anonymous client to call**

Ada Rendezvous



Ada Syntax

- **entry call**
  <server_name>.<entry_name>
- **entry**

```
accept <entry_name>
   {(<params>)} do
{ sequence of statements}
end <entry_name>
```

- **nondeterministic choice of entries**

```
select   accept X do
         …
         end X;
         accept Y do
         …
         end Y;
end select
```


More Ada Syntax

- **guarded entries: all guards (i.e., exprj) evaluated when select is entered; choice among the true guards is nondeterministic.**

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

Producer-Consumer Model

- **A model for managing explicit parallelism, communication between tasks**
  - **Producer -- reads in values one by one, "processing" them and then passing them on.**
  - **Consumer - uses values one by one, "chewing" them before printing them.**
  - **But they run at independent speeds!! This requires explicit synchronization between tasks**
- **Our job: to examine several examples of attempted synchronization and critique them**

First (incorrect) Attempt

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

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

global variable g;

Synch via Global Variable

Tasks use global variable *g* to communicate.
Result is essentially a sequential program.

```
task producer
body{
      c: char;
      loop{
            get (c);
            g := process (c);
            consumer.take;
            }
}
```

```
task consumer
      entry take;
body{
      d: char;
      loop{
            accept take do
                  d := g;
                  put (chew(d));
            end take;
            }
}
```


Synch via Local Variable

Tasks communicate through a parameter to the rendezvous with same effect of sequentializing execution.

```
task producer
body{
      c,g : char;
      loop{
            get (c);
            g := process (c);
            consumer.take (g);
            }
}
```

```
task consumer
      entry take(x: in char);
body{
      d: char;
      loop{
            accept take(d) do
            end take;
            put (chew(d));
            }
}
```

Semaphores

- Semaphores invented by Dijkstra, 1968
- Semaphore uses {value, p, v}
  - value is an integer variable
  - p : if value >=1 then a process can perform a p operation to decrement value by 1, else a process attempting a p must wait until value becomes >=1
  - v: a process can perform a v operation to increment value by 1

Binary Semaphores

- Value is always either 0 or 1
  - p: if value >= 1 then value -- ; otherwise, wait
  - v: value++
- Can implement in Ada using a task

Synch w. Binary Semaphore

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

```
task producer
body{
      c : char;
      loop{
            get (c);
            c := process (c);
            critical.p;
            g := c;
            critical.v;
            }
}
```

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

Use semaphore to protect accesses to g, which protect data transfer.

Questions

- **What if we wrote one critical section as?**
  critical.v … critical.p or
  critical.p … critical.p
  misplacing semaphore operations is hazardous!
- **What happens if there are multiple consumers and one producer?**
  - **This discipline may not work. May need to simulate size of inputs already seen in a non-binary semaphore and use ++/-- operations for p and v.**