Concurrent Programming

Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University

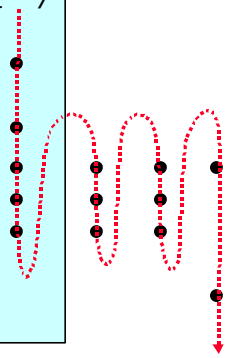
May, 2001

CONCURRENT PROGRAMMING

- **Operations in the source text are *concurrent* if they could be, but need not be, executed in parallel. Operations that occur one after the other, ordered in time, are said to be *sequential*.**
- **The fundamental concept of concurrent programming is the notion of a *process*, which corresponds to a sequential computation, with its own thread of control.**
- **The *thread* of a sequential computation is the sequence of program points that are reached as control flows through the source text of the program.**

THREAD

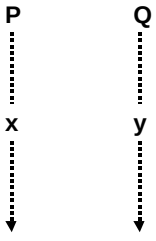
```
#include <stdio.h>

/* copy input to output */
main() {
   int c;

   c = getchar();
   while (c != EOF) {
       putchar(c);
       c =getchar();
   }
   putchar('\n');
}
```



INTERACTIONS BETWEEN PROCESSES

***Communication*: involves the exchange of data between processes, either by an explicit message or through the values of shared variables.**

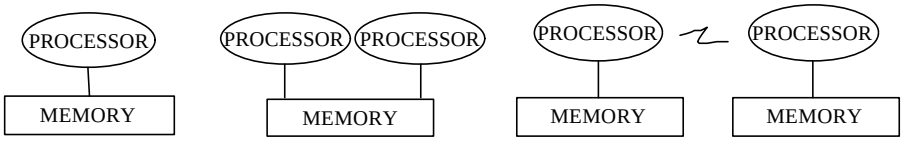
***Synchronization*: relates the thread of one process with that of another.**



Synchronization can be used to constraint the order in which P reaches x and Q reaches y.

Interactions between processes can also be visualized in terms of *competition* and *cooperation* between processes.

HARDWARE ARCHITECTURES



(a) Single processor (b) Shared memory (c) Distributed machine

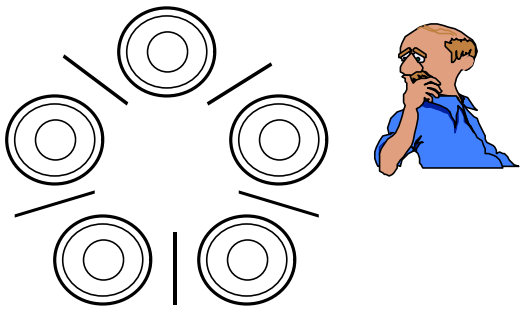
AN ADA PROGRAM

```
with text_io; use text_io;    -- import character input/output procedures
procedure hello is
begin
        put_line("hello world");
end hello;
```

AN ADA PROGRAM

```
with text_io; use text_io;
procedure identify is
      task p;                    -- task specification for p
      task body p is
      begin
            put_line("p");
      end p;
      task q;                    -- task specification for q
      task body q is
      begin
            put_line("q");
      end q;
begin                            -- procedure body sets up parent of p and q
      put_line("r");
end identify;
```


THE DINING PHILOSOPHERS



The design issues:

- **how to avoid deadlock and livelock?**
- **how to guarantee fairness (avoid starvation)?**

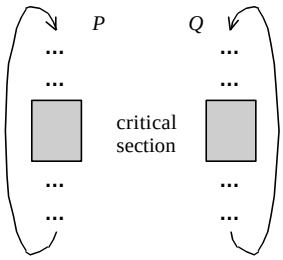
CRITICAL SECTIONS

A *critical section* in a process is a portion or section of code that must be treated as an atomic event.

Two critical sections are said to be *mutually exclusive* because their execution must not overlap.

In a sequential language, the following two statements increment x by 3, but in a concurrent language, the following two statements may be executed in parallel by two different threads, thereby producing three different outcomes.

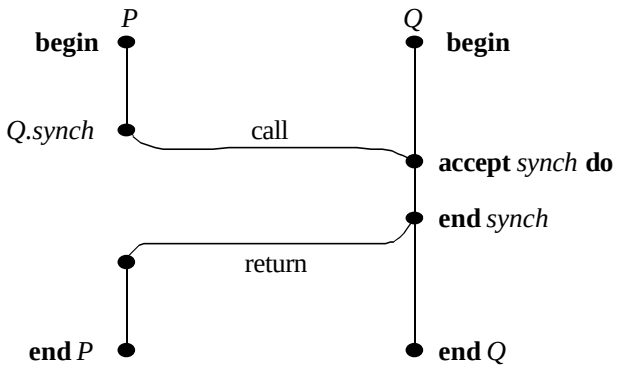
```
x:=x+1;        x:=x+2
```



A RENDEZVOUS

A rendezvous combines two events:

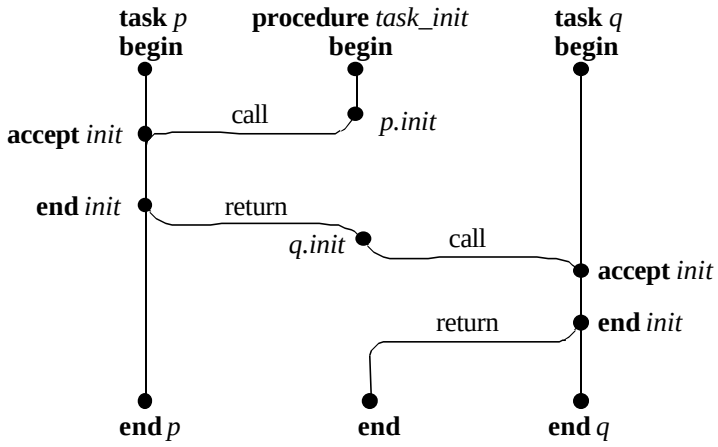
1. **A call within a client process P.**
2. **Acceptance of the call by the server process Q.**



RENDEZVOUS `init` INITAILIZES TASKS OF TYPE `emitter`

```
with text_io; use text_io;
procedure task_init is
  task type emitter is
     entry init(c : character)
  end emitter;
  p, q : emitter;
  task body emitter is
     me : character;
  begin
     accept init(c : character) do
          me := c;
     end init
     put(me); new_line;
  end emitter
begin
  p.init('p');
  q.init('q');
  put('r'); new_line;
end task_init;
```


THREADS FO THE TASKS SET UP BY THE PROGRAM



SELECTIVE ACCEPTANCE

The select construct in Ada allows a server to offer a selection of services to its clients.

```
select
   accept deliver_milk do
      …
   end deliver_milk;
or
   accept deliver_juice do
      …
   end deliver_juice;
end select;
```


GUARDED SELECTIVES

Alternatives in a select command can also be guarded.

```
select
   when notfull ⇒ accept enter(c : in character) do
      …
   end enter;
or
   when notempty ⇒ accept leave(c : out character) do
      …
   end leave;
end select;
```

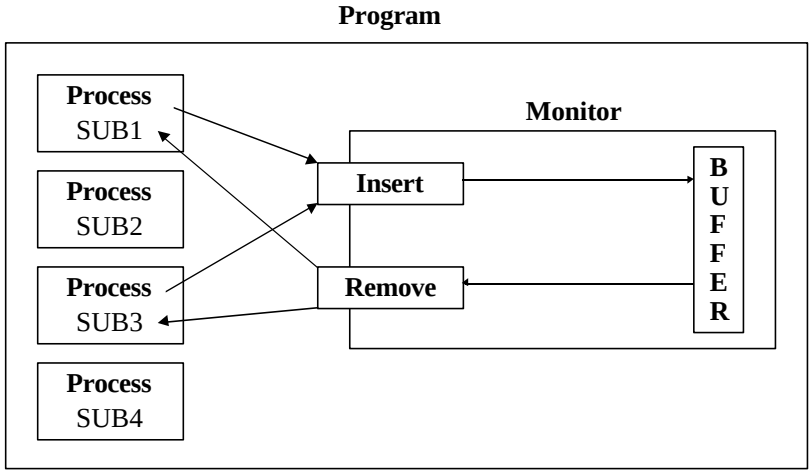
DYNAMIC CREATION OF TASKS THROUGH ACCESS TYPES

```
with text_io; use text_io;
procedure pointers is
  task type emitter is
     entry init(c : character)
  end emitter;
  type emitter_ptr is access emitter;
  p, q : emitter;
  task body emitter is
     me : character;
  begin
     accept init(c : character) do
          me := c;
     end init;
     put(me); new_line;
  end emitter
```


DYNAMIC CREATION OF TASKS THROUGH ACCESS TYPES (cont.)

```
begin
  p := new emitter;
  q := new emitter;
  p.init('p');
  q.init('q');
  put('r'); new_line;
end pointers;
```

MONITOR



A PROGRAM USING A MONITOR TO CONTROL ACCESS TO A SHARED BUFFER

```
type databuf =
  monitor
    const bufsize = 100;
    var buf : array [1..bufsize] of integer;
              next_in;
              next_out        : 1..bufsize;
              filled          : 0..bufsize;
              sender_q;
              receiver_q      : queue;
  procedure entry deposit(item : integer);
    begin
    if filled = bufsize
              then delay(sender_q);
    buf[next_in] := item;
    next_in := (next_in mod bufsize) + 1;
    filled := filled + 1;
    continue(receiver_q);
    end;
```

A PROGRAM USING A MONITOR TO CONTROL ACCESS TO A SHARED BUFFER

```
procedure entry fetch(var item : integer);
  begin
    if filled = 0
        then delay(receive_q);
    item := buf[next_out];
    next_out := (next_out mod bufsize) + 1;
    filled := filled – 1;
    continue(sender_q)
  end;

begin
   filled := 0;
   next_in := 1;
   next_out := 1;
end;
```

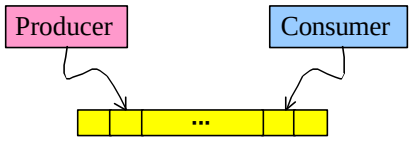

A PROGRAM USING A MONITOR TO CONTROL ACCESS TO A SHARED BUFFER

```
type producer = process(buffer : databuf);
  var newvalue : integer;
  begin
  cycle
    -- produce newvalue --
    buffer.deposit(newvalue);
  end
  end;

type consumer = process(buffer : databuf)
  var stored_value : integer;
  begin
  cycle
    buffer.fetch(stored_value);
    -- consume stored_value --
  end
  end;
```

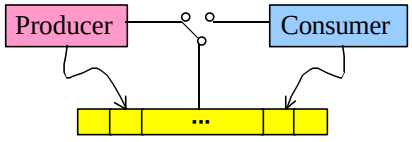
A PROGRAM USING A MONITOR TO CONTROL ACCESS TO A SHARED BUFFER

```
-- type declarations --

var new_producer   : producer;
    new_consumer   : consumer;
    new_buffer     : databuf;
begin
init new_buffer, new_producer(new_buffer),
    new_consumer(new_buffer);
end;
```


SOLUTIONS TO THE PRODUCER-CONSUMER PROBLEM

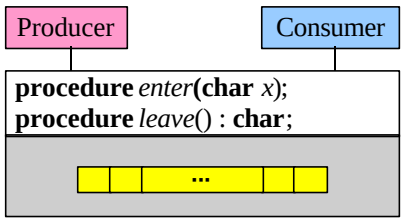


(a) Direct access

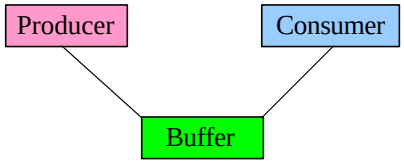


(b) Synchronized direct access

SOLUTIONS TO THE PRODUCER-CONSUMER PROBLEM (cont.)



(c) Access through a monitor



(d) The buffer as a separate process

PSEUDOCODE FOR UNSYNCHRONIZED ACCESS TO THE BUFFER

```
with text_io; use text_io;
procedure direct is
    size : constant integer := 5;
    buf : array(0..size-1) of character;
    front, rear : integer := 0;
    function notfull return boolean is … end notfull;
    function notempty return boolean is … end notempty;
    task producer;
    task body producer is
        c : character;
    begin
        while not end_of_file loop
                if notfull then
                        get(c);
                        buf(rear) := c;
                        rear := (rear +1) mod size;
                end if
        end loop;
    end producer
```

PSEUDOCODE FOR UNSYNCHRONIZED ACCESS TO THE BUFFER (cont.)

```
task consumer;
task body consumer is
    c : character;
begin
    loop
        if notempty then
            c := buf(front);
            front := (front+1) mod size;
            put(c);
        end if;
    end loop;
end consumer;

begin
    null;
end direct;
```


SEMAPHORES: MUTUAL EXCLUSION

A *semaphore* is a construct that has an integer variable value and supports two operations:

1. If ***value*** ≥ 1, then a process can perform a ***p*** operation to decrement the value by 1. Otherwise, a process attempting a ***p*** operation waits until the value becomes greater than or equal to 1.
2. A process can perform a ***v*** operation to increment variable ***value*** by 1.

A *binary semaphore* is a semaphore whose value is constrained to be either 0 or 1. If the value of a binary semaphore is 1, then a process attempting a *v* operation on it is suspended until its value becomes 0. In other words, the *p* and *v* operations on a semaphore must be performed alternately.

IMPLEMENTATIONS OF SEMAPHORES

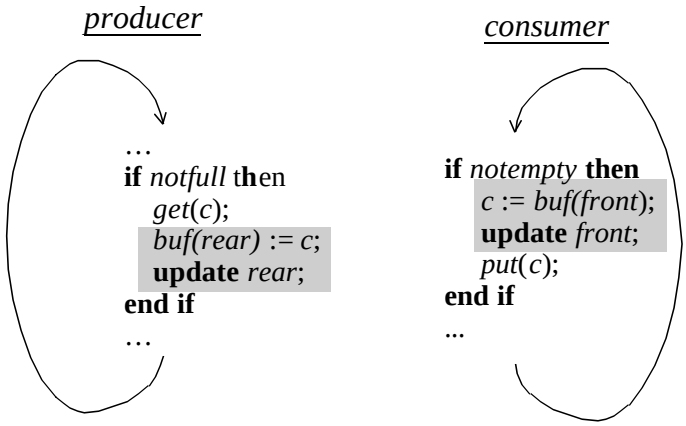
```
task type binary_semaphore is
   entry p;
   entry v;
end binary_semaphore;

task body binary_semaphore is
begin
   loop
      accept p;
      accept v;
   end loop;
end binary_semaphore;
```


MUTUAL EXCLUSION

Mutual exclusion can be implemented by enclosing each critical section between the operations *s.p* and *s.v*, where *s* is a binary semaphore:

process Q	*process R*
…	…
s.p;	*s.p*;
critical section for *Q*;	critical section for *R*;
s.v	*s.v*
…	…

THE PRODUCER AND CONSUMER AS CYCLIC PROCESSES WITH CRITICAL SECTIONS



A SEMAPHORE AS A TASK IN ADA

```
task body semaphore is
    value : integer;
begin
    accept init(n : integer) do                    -- initialization
        value := n;
    end init;
    loop
        select
                when value ≥ 1 ⇒                   -- p operation
                        accept p do
                                value := value – 1;
                        end p;
        or      accept v do                        -- v operation
                        value := value + 1;
                end v;
        end select;
    end loop;
end semaphore;
```

USE OF THE SEMAPHORES *filling, emptying,* AND *critical*

```
task body producer is
   c : character;
begin
   while not end_of_file loop
      get(c);
      filling.p;
         critical.p;
            buf(rear) := c;
            rear := (rear+1) mod size;
         critical.v;
      emptying.v;
   end loop;
end producer;
```

```
task body consumer is
   c : character;
begin
   loop
      emptying.p;
         critical.p;
            c = buf(front);
            front := (front+1) mod size;
         critical.v;
      filling.v;
       put(c);
   end loop;
end consumer;
```

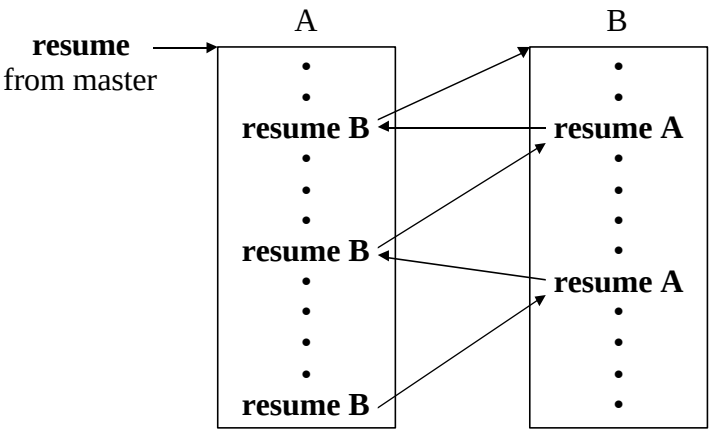

A MONITOR FOR A BOUNDED BUFFER

```
monitor buffer is
      buf : …;
      procedure enter(c : in character);
      begin
            if buffer full then wait(filling);        -- block producer
            enter c into buffer;
                  …
            signal(empty);                            -- unblock consumer
      end enter;
      procedure leave(c : out character);
      begin
            if buffer empty then wait(emptying);      -- block consumer
            c := next character;
                  …
            signal(filling);                          -- unblock producer
      end leave;
begin
      initialize private data;
end buffer;
```

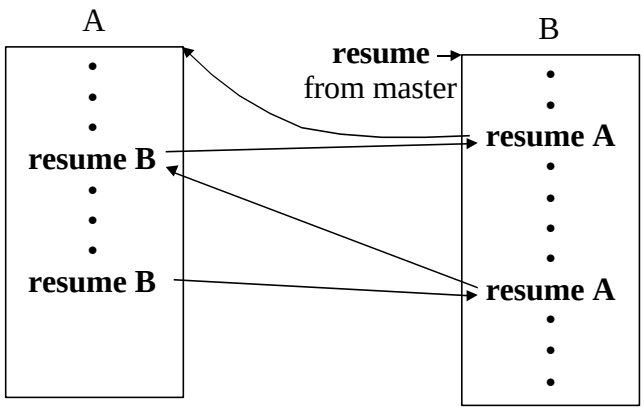
THE BUFFER AS A PROCESS

```
task body buffer is
  <data-declarations>
begin
  loop
    select
          when notfull ⇒
                accept enter(x : in integer) do
                      …
                end enter;
    or    when notempty ⇒
                accept leave(x : out integer) do
                      …
                end leave;
    end select
  end loop
end buffer;
```

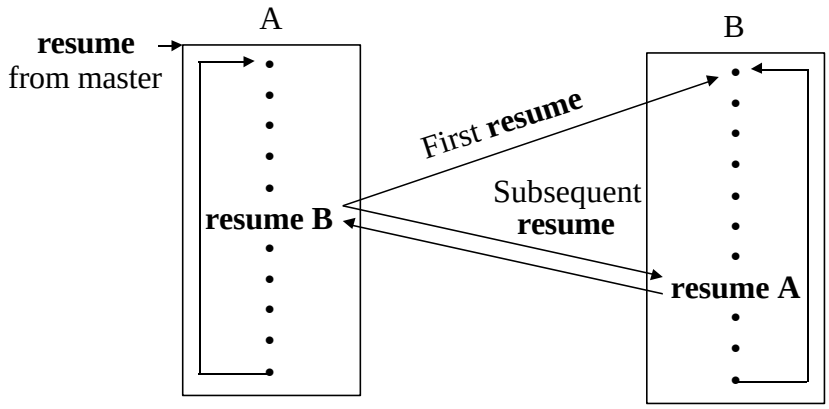

Two Possible Execution Control Sequences for Two Coroutines Without loops



Two Possible Execution Control Sequences for Two Coroutines Without loops



Coroutine execution sequence with loops



UNIX PROCESSES

A program can be any of several things:

- A file containing instructions and data used to initialize a process;
- An algorithm represented in the source code of some programming language, probably stored in a file;

A process, briefly, is a running program. Processes are resources that are managed by the operating system (OS).

THE COMPONENTS OF A PROCESS

- **the text (code) segment**
- **the user data segment (on modern Unix systems divided into the initialized and uninitialized (called bss) data segments)**
- **the system data segment**

The `fork` and `exec` System Calls

The exec system call is the only way a process begins execution; the fork system call is the only way to create a new process.

```
/* ignoring errors ... */
if (fork() == 0) {
    /* i am the child */
    exec("new program");
} else {
    /* i am the parent */
    wait();
}
```


The `exec` System Calls

`exec` in all its forms overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. There can be no return from a successful exec because the calling process image is overlaid by the new process image.

If `exec` returns to the calling process, an error has occurred; the return value is `-1` and `errno` is set to indicate the error.

The `system()` function forks to create a child process that in turn `execs` the shell in order to execute string. If the `fork()` or `exec()` fails, `system()` returns a value of `-1` and sets `errno`.

The `fork` System Calls

`fork` creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited. On success, the `PID` of the child process is returned in the parent's thread of execution, and a `0` is returned in the child's thread of execution. On failure, a `-1` will be returned in the parent's context, no child process will be created, and `errno` will be set appropriately.

Because the child process is an exact copy of the parent, both processes pick up execution from the next statement after `fork`. They share identical all resources the original one had at the time of fork()ing, but not any allocated later.

PROCESS CREATION IN UNIX

```
int pid;
int status = 0;
if (pid = fork()) {
/* parent */
…..
pid = wait(&status);
} else {
/* child */
…..
exit(status);
}
```

Another example