2. Processes and Interactions

2.1 The Process Notion

2.2 Defining and Instantiating Processes

- Precedence Relations
- Implicit Process Creation
- Dynamic Creation With fork And join

2.3 Basic Process Interactions

- Competition: The Critical Section Problem
- Cooperation

2.4 Semaphores

- Semaphore Operations and Data
- Mutual Exclusion
- Producer/Consumer Situations

Processes

- A **process** is the activity of executing a program on a CPU.
- Conceptually…
  - Each process has its own CPU
  - Processes are running concurrently
- **Physical** concurrency = **parallelism**
  - This requires multiple CPUs
- **Logical** concurrency = **time-shared** CPU
- Processes **cooperate** (shared memory, messages, synchronization)
- Processes **compete** for resources

Why Processes?

- **Hardware-independent** solutions
  - Processes cooperate and compete correctly, regardless of the number of CPUs
- **Structuring** mechanism
  - Tasks are isolated with well-defined interfaces

How to define/create Processes?

- Need to:
  - Define **what** each process does (the program)
  - **Create** the processes (data structure/PCB)
    - Subject of another chapter
  - Specify precedence relations:

    when processes **start** and **stop** executing, relative to each other

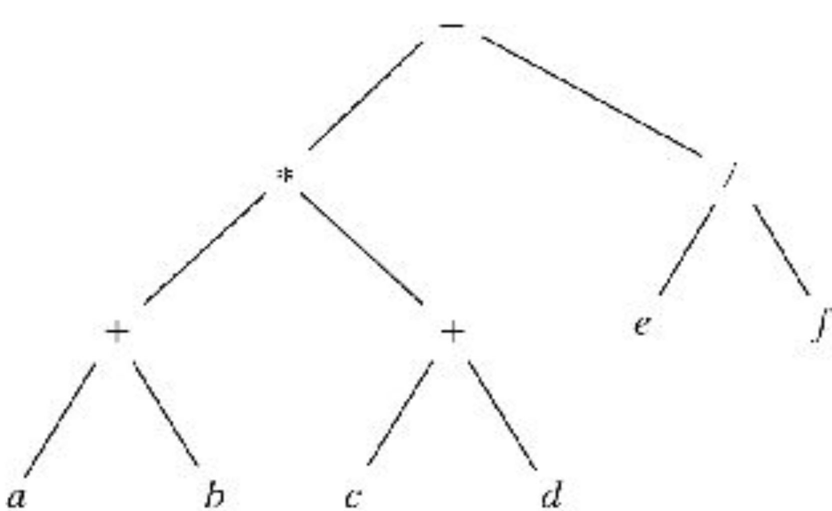
Specifying precedence relations

- A general approach: **Process flow graphs**
  - Directed acyclic graphs (DAGs)
  - **Edges** = processes
  - **Vertices** = starting and ending points of processes

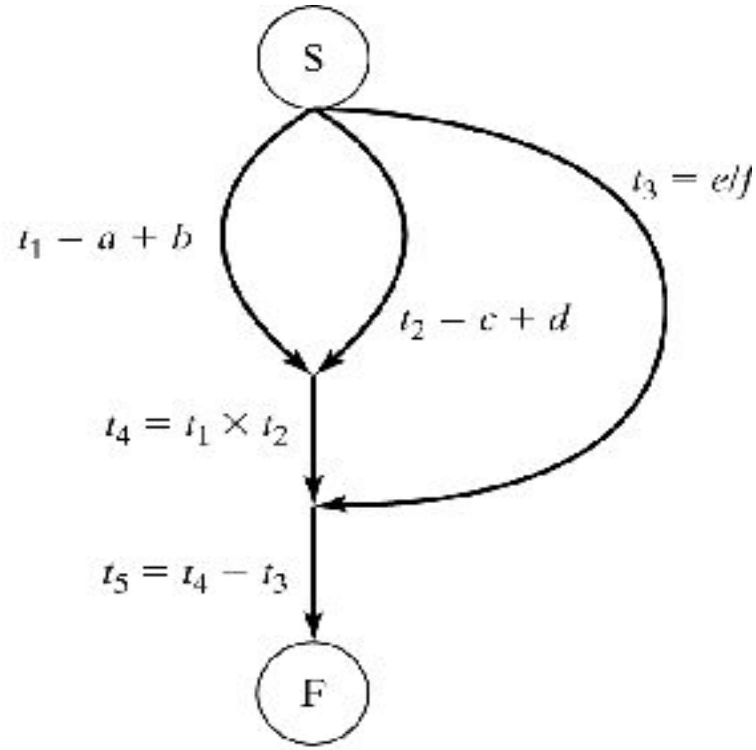
Process flow graphs

Example: parallel evaluation of arithmetic expression:

(a + b) * (c + d) - (e / f)



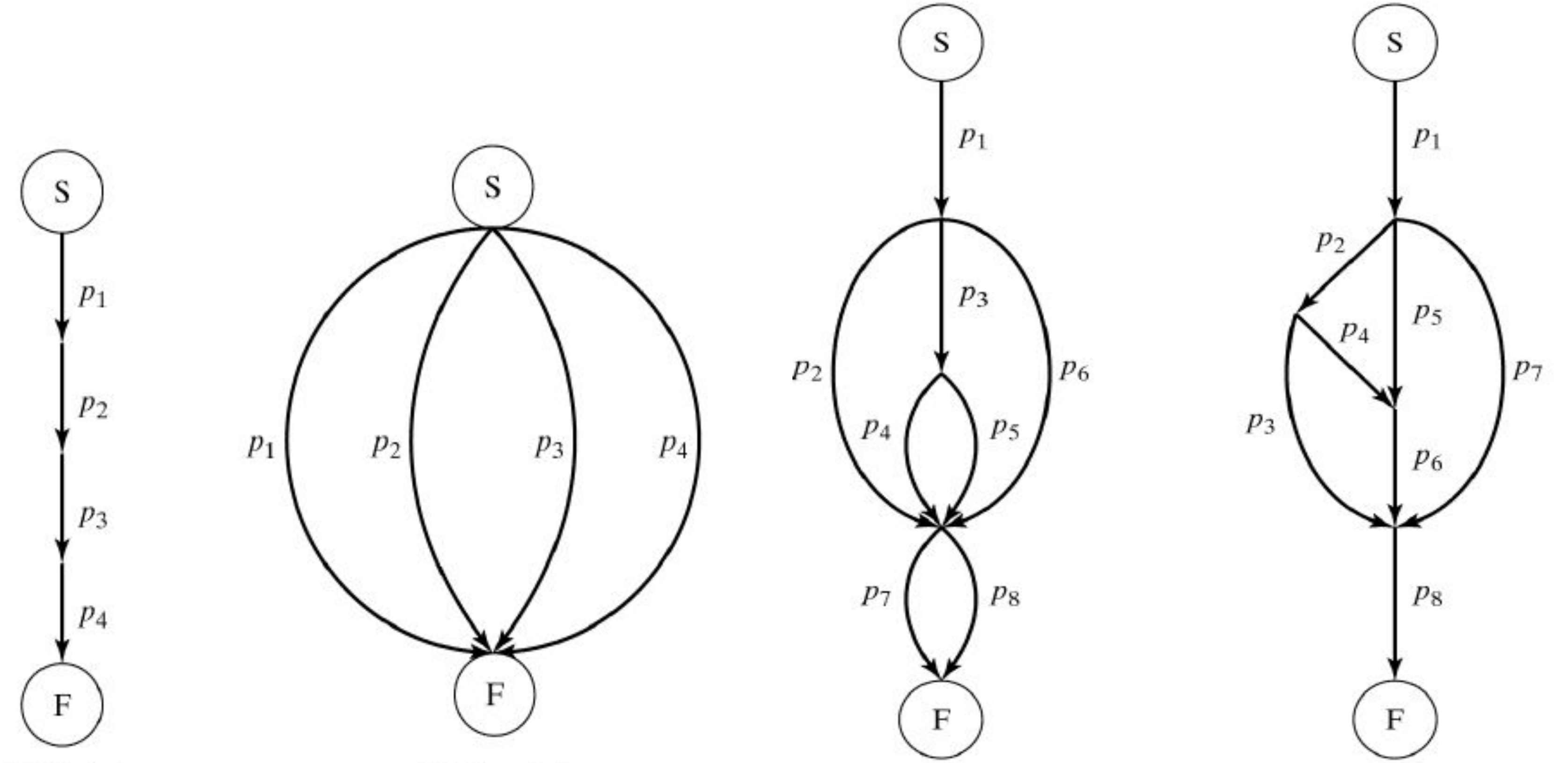
Expression tree



Process flow graph

Process flow graphs

Other examples of Precedence Relationships



(a) Serial (b) Parallel (c) Series/parallel (d) General precedence

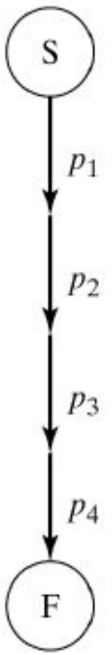
Process flow graphs (PFG)

- Challenge: devise programming language constructs to capture PFG
- Special case: **Properly Nested Graphs**
- A graph is properly nested if it corresponds to a properly nested **expression**, where
  - S(p1, p2, …) describes serial execution of p1, p2, …
  - P(p1, p2, …) describes parallel execution of p1, p2, …

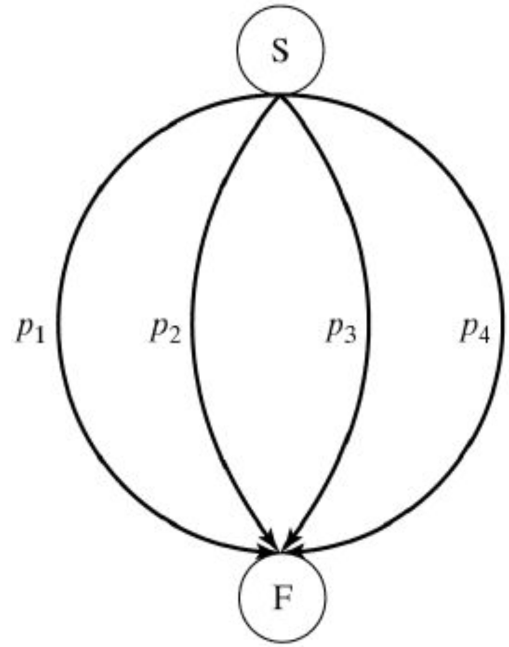
Process flow graphs

- Strictly **sequential** or strictly **parallel** execution

(a) S(p1, p2, p3, p4) (b) P(p1, p2, p3, p4)



(a) Serial



(b) Parallel

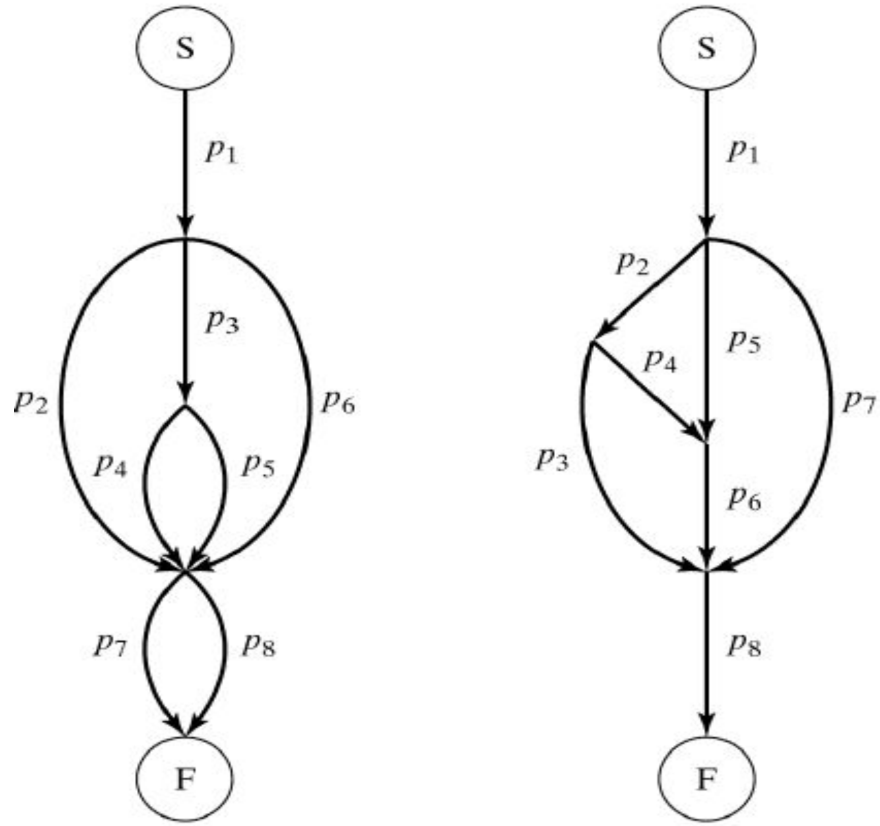
Process flow graphs

(c) corresponds to the properly nested expression:

S(p1, P(p2, S(p3, P(p4, p5)), p6), P(p7, p8))

(d) is not properly nested

– (proof: text, page 44)



(c) Series/parallel (d) General precedence

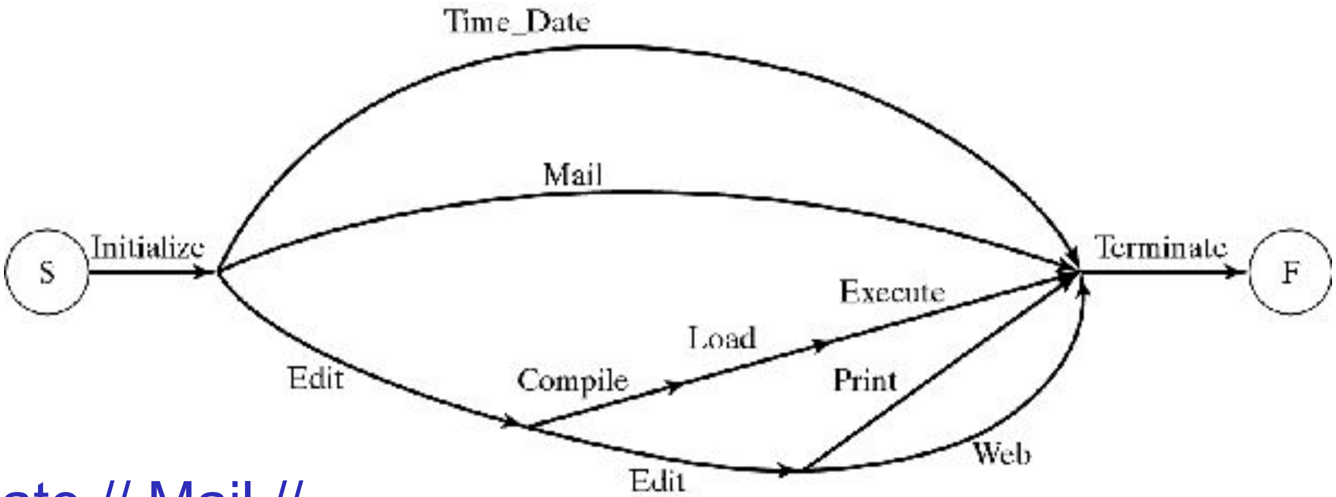
Language Constructs for Process Creation

- to capture properly nested graphs
  - **cobegin // coend**
  - **forall** statement
- to capture unrestricted graphs
  - **fork/join/quit**

cobegin/coend statements

- **syntax**: cobegin C_1 // C_2 // … // C_n coend
- **meaning**:
  - all C_i may proceed concurrently
  - when *all* C_i's terminate, next statement can proceed
- **cobegin/coend** are analogous to **S/P** notation
  - S(a,b) ≡ a; b (sequential execution by default)
  - P(a,b) ≡ cobegin a // b coend

cobegin/coend example




```
cobegin
  Time_Date // Mail //
    {   Edit;
        cobegin
          {   Compile; Load; Execute} //
          {   Edit; cobegin Print // Web coend}
        coend
    }
coend
```

Data parallelism

- **Same code** is applied to **different data**
- The **forall** statement
  - **syntax**: forall (parameters) statements
  - **meaning**:
    - Parameters specify set of data items
    - Statements are executed for each item concurrently

Example of forall statement

- Example: **Matrix Multiply** A=B*C

```
forall ( i:1..n, j:1..m )
{
   A[i][j] = 0;
   for ( k=1; k<=r; ++k )
      A[i][j] = A[i][j] + B[i][k]*C[k][j];
}
```

- **Each inner product** is computed **sequentially**
- **All inner products** are computed in **parallel**

fork/join/quit

- **cobegin/coend**
  - limited to *properly nested graphs*
- **forall**
  - limited to *data parallelism*
- **fork/join/quit**
  - can express *arbitrary functional parallelism* (any process flow graph)

fork/join/quit

- **Syntax**: fork x

  Meaning: create new process that begins executing at label x

- **Syntax**: join t,y

  Meaning:

  ```
  t = t–1;
  if (t==0) goto y;
  ```

- **Syntax**: quit

  Meaning: terminate current process

fork/join/quit example

- A simple Example:
  - execute x and y concurrently
  - when both finish, execute z

```
          t = 2;
          fork L1; fork L2; quit;
      L1: x; join t,L3; quit
      L2: y; join t,L3; quit;
      L3: z;
```

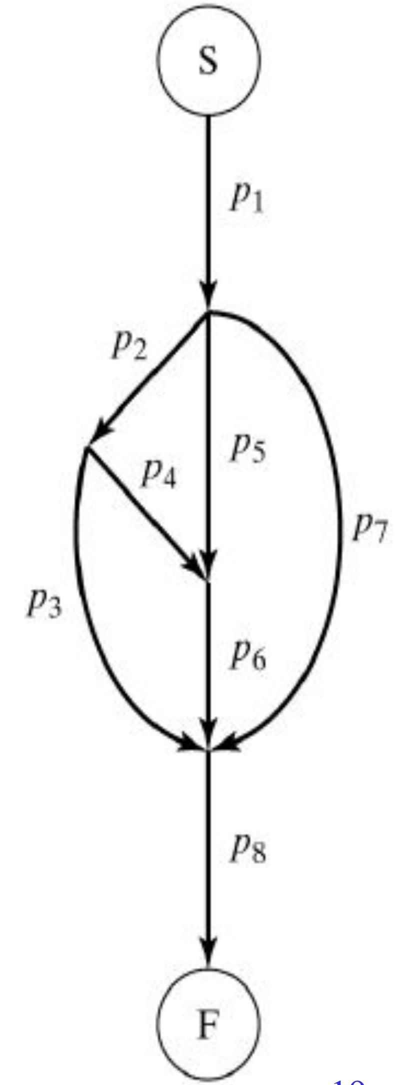
  - Better:

```
          t = 2;
          fork L2; x; join t,L3; quit;
      L2: y; join t,L3; quit
      L3: z;
```

fork/join/quit example

- Example: Graph in Figure 2-1(d)

```
        t1 = 2; t2 = 3;
        p1; fork L2; fork L5; fork L7; quit;
L2: p2; fork L3; fork L4; quit;
L5: p5; join t1,L6; quit;
L7: p7; join t2,L8; quit;
L4: p4; join t1,L6; quit;
L3: p3; join t2,L8; quit;
L6: p6; join t2,L8; quit;
L8: p8; quit;
```



Example: the Unix *fork* statement

- **procid = fork()**
- Replicates calling process
- Parent and child are identical except for the value of **procid**
- Use **procid** to diverge parent and child:

```
if (procid==0) do_child_processing
else do_parent_processing
```

Explicit Process Declarations

- Designate piece of code as a unit of execution
  - Facilitates program structuring
- Instantiate:
  - Statically (like `cobegin`) or
  - Dynamically (like `fork`)

Explicit Process Declarations

```
process p

   process p1
     declarations_for_p1
   begin ... end

   process type p2
     declarations_for_p2
   begin ... end

begin
   ...
   q = new p2;
   ...
end
```

Process Interactions

- **Competition**
  - Two processes both want to access the same resource
  - Example: write the same file, use the same printer
  - Requires **mutual exclusion**
- **Cooperation**
  - Two processes work on a common problem
  - Example: ***Producer*** → ***Buffer*** → ***Consumer***
  - Requires **coordination**

Process Interactions

- Competition: The **Critical Section** Problem

```
x = 0;
cobegin
p1: …
    x = x + 1;
    …
    //
p2: …
    x = x + 1;
    …
coend
```

- After both processes execute, we should have x=2, but …

The Critical Section Problem

- Interleaved execution (due to parallel processing or context switching)

```
p1: R1 = x;                    p2: …
                                   R2 = x;
    R1 = R1 + 1;
                                   R2 = R2 + 1;
    x = R1 ;
    …                              x = R2;
```

- x has only been incremented once. The first update (x = R1) is lost.

The Critical Section Problem

- General problem statement:

```
cobegin
p1: while(1) {CS1; program1;}
    //
p2: while(1) {CS2; program2;}
    //
    ...
    //
pn: while(1) {CSn; programn;}
coend
```

- Guarantee **mutual exclusion**: At any time, at most one process should be executing within its critical section (CSi).

The Critical Section Problem

In addition to **mutual exclusion**, must also prevent **mutual blocking**:

1. Process **outside** of its CS must not prevent other processes from entering its CS *(no "dog in manger")*
2. Process must not be able to repeatedly reenter its CS and **starve** other processes *(fairness)*
3. Processes must not **block each other** forever *(no deadlock)*
4. Processes must not **repeatedly yield** to each other ("after you—after you") *(no livelock)*

The Critical Section Problem

- Solving the problem is subtle
- We will examine a few incorrect solutions before describing a correct one: Peterson’s algorithm

Attempt 1 (incorrect)

- Use a single turn variable:

```
int turn = 1;
cobegin
p1: while (1) {
      while (turn != 1);   /*wait*/
      CS1; turn = 2; program1;
      }
 //
p2: while (1) {
      while (turn != 2);   /*wait*/
      CS2; turn = 1; program2;
      }
coend
```

- Violates blocking requirement (1), “dog in manger”

Attempt 2 (incorrect)

- Use two variables: c1=1 when p1 wants to enter its CS. c2=1 when p2 wants to enter its CS.

```
int c1 = 0, c2 = 0;
cobegin
p1: while (1) {
      c1 = 1;
      while (c2);  /*wait*/
      CS1; c1 = 0; program1;
      } //
p2: while (1) {
      c2 = 1;
      while (c1);  /*wait*/
      CS2; c2 = 0; program2;
      }
coend
```

- Violates blocking requirement (3), deadlock.

Attempt 3 (incorrect)

- Like #2, but reset intent variables (c1 and c2) each time:

```
int c1 = 0, c2 = 0;
cobegin
p1: while (1) {
     c1 = 1;
     if (c2) c1 = 0;   //go back, try again
     else {CS1; c1 = 0; program1}
     }  //
p2: while (1) {
     c2 = 1;
     if (c1) c2 = 0;   //go back, try again
     else {CS2; c2 = 0; program2}
     }
coend
```

- Violates livelock (4) and starvation (2) requirements

Peterson's algorithm

- Processes indicate intent to enter CS as in #2 and #3 (by setting c1 or c2)
- After a process indicates its intent to enter, it (politely) tells the other that it will wait if necessary (using willWait)
- It then waits until one of the following is true:
  - The other process is **not trying** to enter; or
  - The other process has said that it **will wait** (by changing the value of the willWait variable.)
- Shared variable willWait is the key:
  - with #3: **both** processes can reset c1/c2 simultaneously
  - with Peterson: willWait can only have a **single** value

Peterson's Algorithm

```
int c1 = 0, c2 = 0, willWait;
cobegin
p1: while (1) {
      c1 = 1; willWait = 1;
      while (c2 && (willWait==1)); /*wait*/
      CS1; c1 = 0; program1;
      }
 //
p2: while (1) {
      c2 = 1; willWait = 2;
      while (c1 && (willWait==2)); /*wait*/
      CS2; c2 = 0; program2;
      }
coend
```

- Guarantees mutual exclusion *and* no blocking

Another algorithm for the critical section problem: the Bakery Algorithm

Based on “taking a number” as in a bakery or post office

1. Process chooses a number larger than the number held by all other processes
2. Process waits until the number it holds is smaller than the number held by any other process trying to get in to the critical section

Complication: there could be ties in step 1.

Code for Bakery Algorithm

```
int number[n];  //shared array.  All entries initially set to 0
//Code for process i.  Variables  j and x are local (non-shared) variables
while(1)  {
   --- Normal (i.e., non-critical) portion of Program ---
   // choose a number
   x = 0;
   for (j=0; j < n; j++)
      if (j != i)  x = max(x,number[j]);
   number[i] = x + 1;
   // wait until the chosen number is the smallest outstanding number
   for (j=0; j < n; j++)
      if (j != i) wait until ((number[j] == 0) or (number[i] < number[j]) or
                                              ((number[i] = number[j]) and (i < j)))
   --- Critical Section ---
   number[i] = 0;
}
```

Software solutions to CS problem

- Drawbacks
  - Difficult to program and to verify
  - Processes loop while waiting (busy-wait).
  - Applicable to only to CS problem: competition. Does not address cooperation among processes.
- Need a better, **more general** solution:
  - semaphores
  - semaphore-based high-level constructs, such as monitors

Semaphores

- A **semaphore** s is a nonnegative integer
- Operations **P** and **V** are defined on s
- Semantics:

  P(s): while (s<1) /*wait*/; s=s-1

  V(s): s=s+1;
- The operations P and V are **atomic** (indivisible)
- If more than one process invokes P simultaneously, their execution is sequential and in arbitrary order
- If more than one process is waiting in P, an arbitrary one continues when s>0
- Assume we have such operations (chapter 3) …

Notes on semaphores

- Developed by Edsger Dijkstra
  http://en.wikipedia.org/wiki/Edsger_W._Dijkstra
- Etymology:
  - **`P(s)`**:
    "P" from "*passaren*" ("pass" in Dutch) or from "*prolagen*," which combines "*proberen*" ("try") and "*verlagen*" ("decrease")
  - **`V(s)`**
    "V" from "*vrigeven*" ("release") or "verhogen" ("*increase*")

Mutual Exclusion w/ Semaphores

- Assume we have P/V as defined previously

```
semaphore mutex = 1;
cobegin
p1: while (1) {
      P(mutex); CS1; V(mutex); program1;}
//
p2: while (1) {
      P(mutex); CS2; V(mutex); program2;}
//
...
pn: while (1) {
      P(mutex); CSn; V(mutex); programn;}
coend;
```

Cooperation

- Semaphores can also solve **cooperation** problems
- Example: assume that **`p1`** must wait for a signal from **`p2`** before proceeding.

```
semaphore s = 0;
cobegin
p1:  ...
     P(s);   /* wait for signal */
     ...
//
p2:  ...
     V(s);   /* send signal */
     ...
coend;
```

Bounded Buffer Problem

- Classic generic scenario:

  Producer → Buffer → Consumer

- Produce and consumer run **concurrently**
- Buffer has a **finite size** (# of elements)
- Consumer may **remove** elements from buffer as long as it is **not empty**
- Producer may **add** data elements to the buffer as long as it is **not full**
- Access to buffer must be **exclusive** (critical section)

Bounded Buffer Problem

```
semaphore e = n, f = 0, b = 1;
cobegin
Producer: while (1) {
   Produce_next_record;
   P(e); P(b); Add_to_buf; V(b); V(f);
   }
//
Consumer: while (1) {
   P(f); P(b); Take_from_buf; V(b); V(e);
   Process_record;
   }
coend
```

Events

- An *event* designates a change in the system state that is of interest to a process
  - Usually triggers some action
  - Usually considered to take no time
  - Principally generated through interrupts and traps (end of an I/O operation, expiration of a timer, machine error, invalid address…)
  - Also can be used for process interaction
  - Can be *synchronous* or *asynchronous*

Synchronous Events

- Process explicitly waits for occurrence of a specific event or set of events generated by another process
- Constructs:
  - Ways to define events
  - E.post (generate an event)
  - E.wait (wait until event is posted)

- Can be implemented with semaphores
- Can be "memoryless" (posted event disappears if no process is waiting).

Asynchronous Events

- Must also be defined, posted
- Process does not explicitly wait
- Process provides *event handlers*
- Handlers are evoked whenever event is posted

Event synchronization in UNIX

- Processes can signal conditions using asynchronous events:
  kill(pid, signal)
- Possible signals: SIGHUP, SIGILL, SIGFPE, SIGKILL, …
- Process calls sigaction() to specify what should happen when a signal arrives. It may
  - catch the signal, with a specified signal handler
  - ignore signal
- Default action: process is killed
- Process can also handle signals synchronously by blocking itself until the next signal arrives (pause() command).

Case study: Event synch. (cont)

- Windows 2000
  - WaitForSingleObject or WaitForMultipleObjects
  - Process blocks until object is signaled

object type	signaled when:
process	all threads complete
thread	terminates
semaphore	incremented
mutex	released
event	posted
timer	expires
file	I/O operation terminates
queue	item placed on queue