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
- Explicit Process Declarations

2.3 Basic Process Interactions

- Competition: The Critical Section Problem
- Cooperation

2.4 Semaphores

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

2.5 Event Synchronization

Processes

- A process is the activity of executing a program on a CPU. Also, called a task.
- 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

Advantages of Process Structure

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

Defining/Instantiating Processes

- Need to
  - Define what each process does
  - Specify precedence relations: when processes start executing and stop executing, relative to each other
  - Create processes

Specifying precedence relations

- Process-flow graphs (unrestricted)
- Properly nested expressions/graphs (also known as series-parallel graphs)

Process flow graphs

- Directed graphs
- Edges represent processes
- Vertices represent initiation, termination of processes

Examples of Precedence Relationships (Process Flow Graphs)

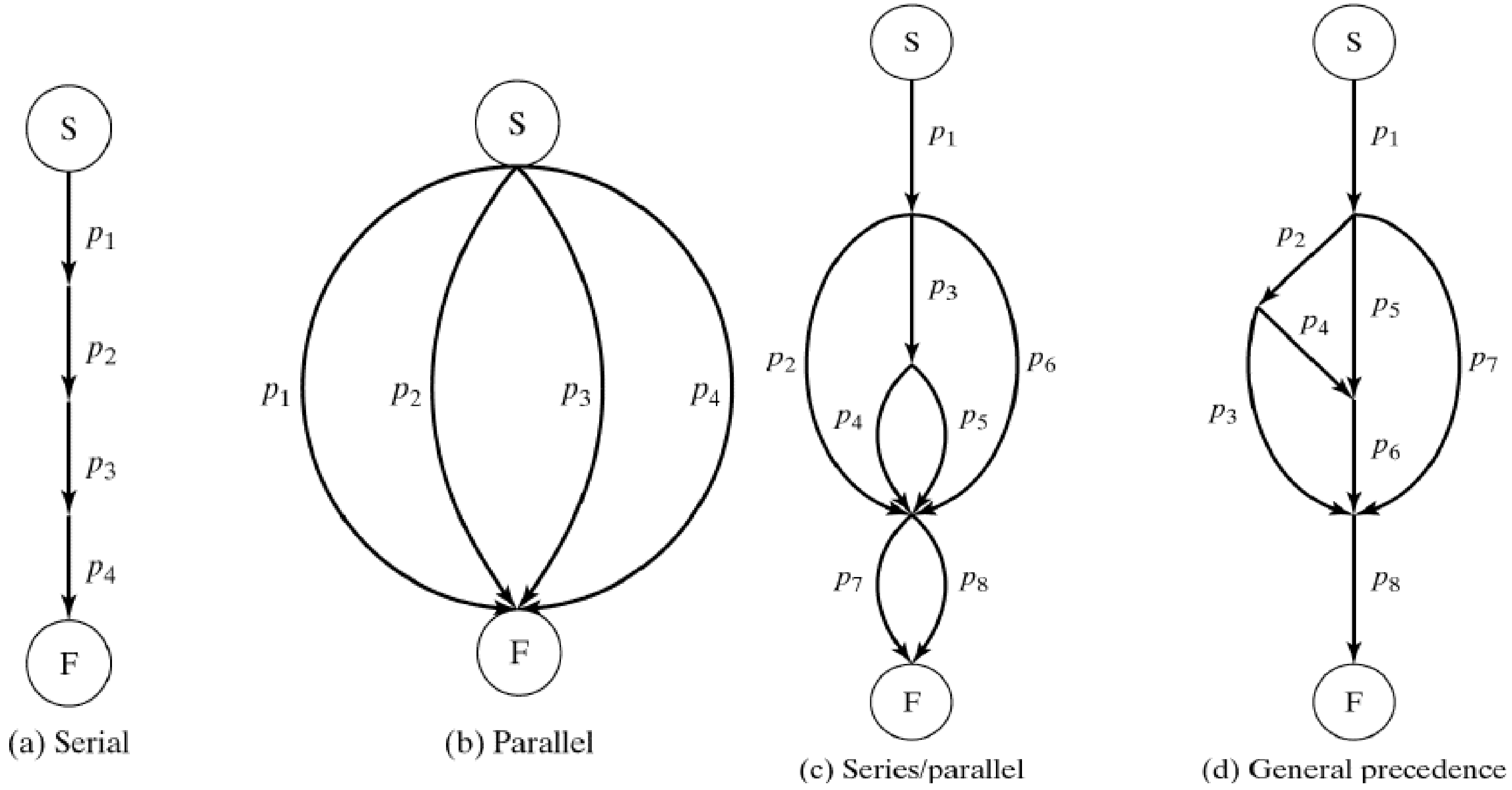
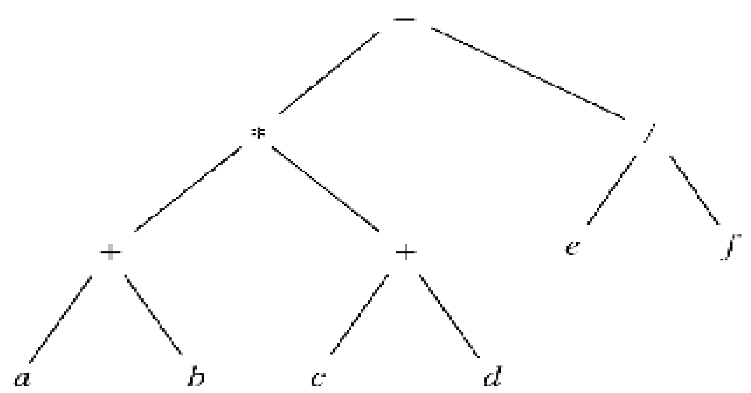


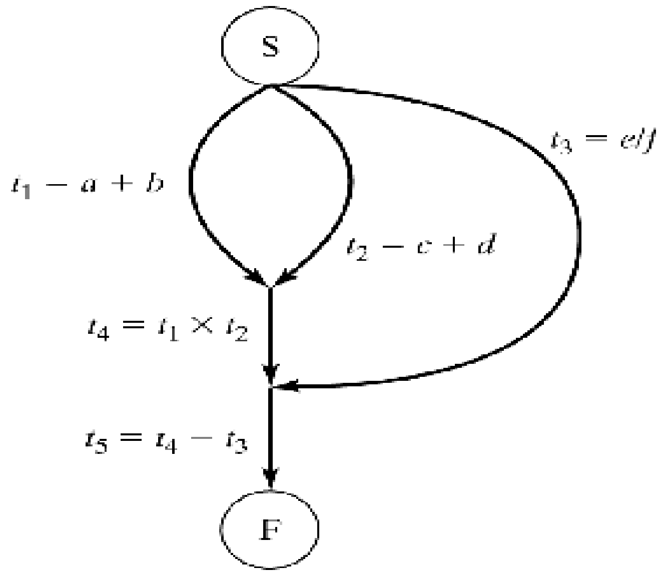
Figure 2-1

Process flow graphs

$(a + b) * (c + d) - (e / f)$ gives rise to



Expression tree



Process flow graph

Figure 2-2

(Unrestricted) Process flow graphs

- Any directed acylic graph (DAG) corresponds to an unrestricted process flow graph, and conversely
- May be too general (like unrestricted goto in sequential programming)

Properly nested expressions

- Two primitives, which can be nested:
  - Serial execution
    - Expressed as S(p1, p2, …)
    - Execute p1, then p2, then …
  - Parallel execution
    - Expressed as P(p1, p2, …)
    - Concurrently execute p1, p2,
- A graph is properly nested if it corresponds to a properly nested expression

Examples of Precedence Relationships (Process Flow Graphs)

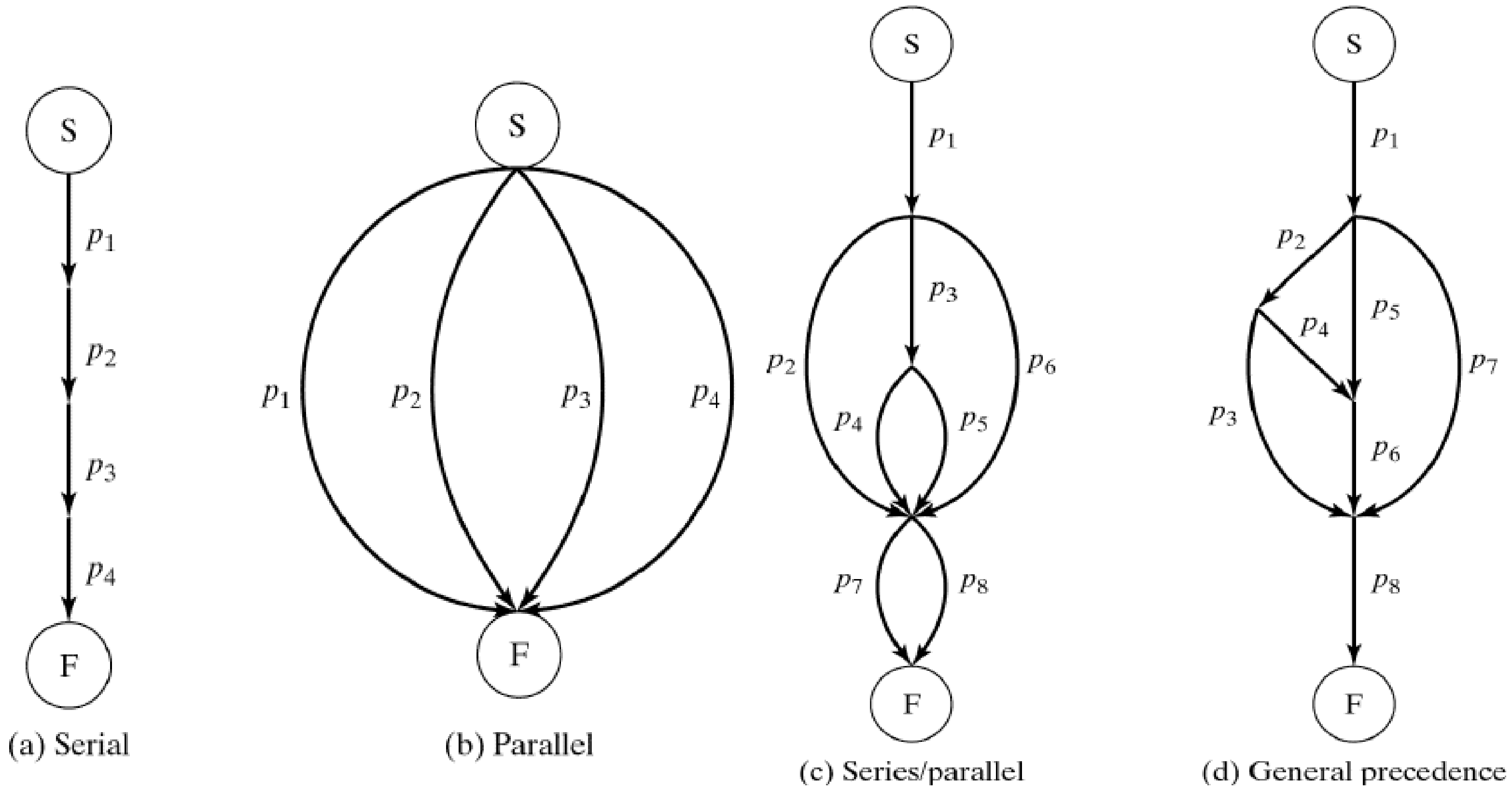
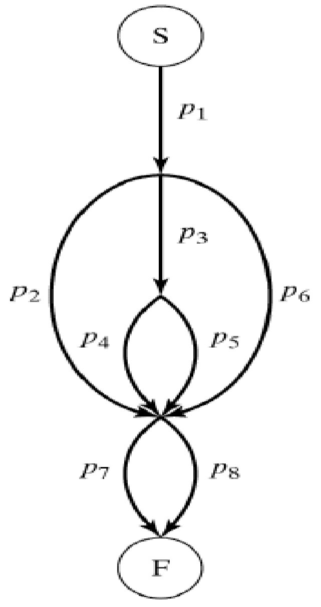


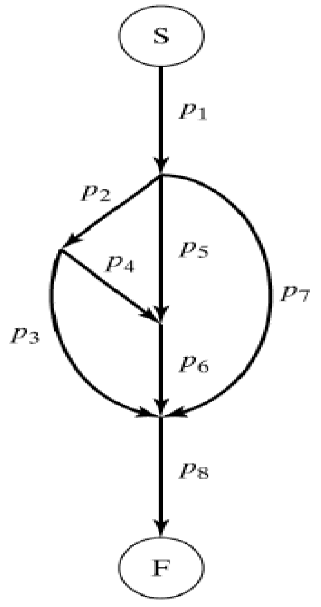
Figure 2-1

Properly nested 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

Process Creation

- Implicit process creation
  - cobegin // coend,
  - forall statement
- Explicit process creation
  - fork/join
  - Explicit process declarations/classes

Implicit Process Creation

- Processes are created dynamically using language constructs.
- Process is not explicitly declared or initiated
- cobegin/coend statement
- Data parallelism: forall statement

Cobegin/coend statement

- syntax: cobegin C_1 // C_2 // … // C_n coend
- meaning:
  - All C_i may proceed concurrently
  - When *all* of the C_i's terminate, the statement following the cobegin/coend can proceed
- cobegin/coend statements have the same expressive power as S/P notation
  - S(a,b) ≡ a; b (sequential execution by default)
  - P(a,b) ≡ cobegin a // b coend

cobegin/coend example

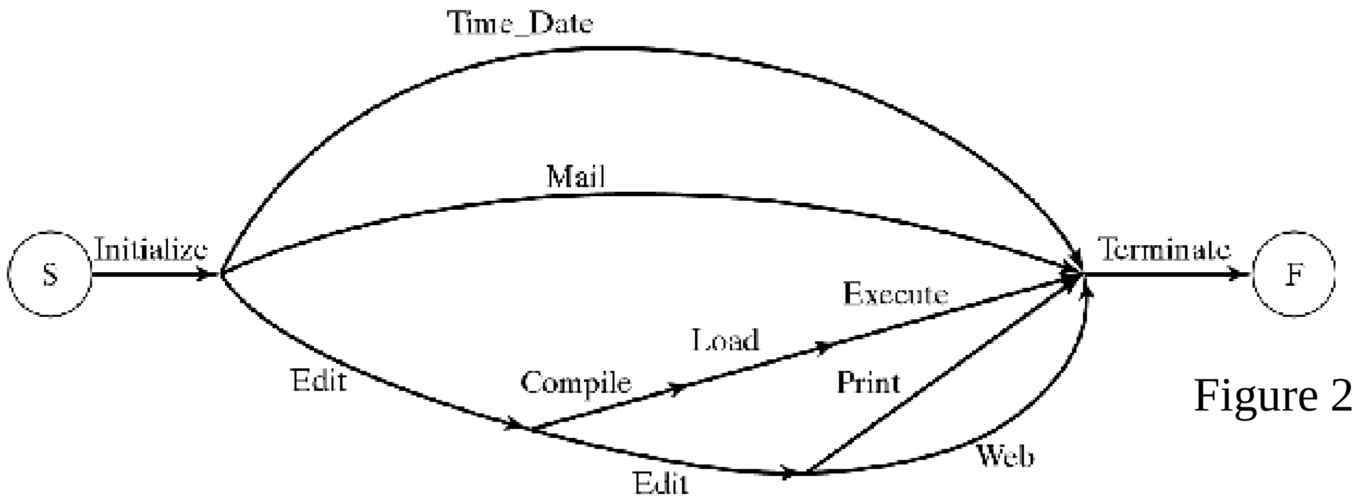


Figure 2-4

```
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

```
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

Explicit Process Creation

- Using fork/join
- Explicit process declarations/classes

Explicit program creation: fork/join

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

The *fork* and *join* primitives

- Syntax: fork x

  Meaning: create new process that
  begins executing at label x

- Syntax: join t,y

  Meaning:

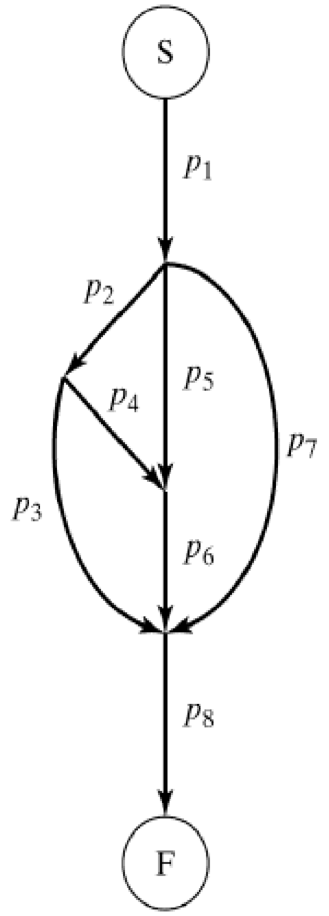
```
t = t–1;
if (t==0) goto y;
```

The operation *must be indivisible*. (Why?)

fork / join 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;
```



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
```

Thread creation in Java

- Define a runnable class

```
Class MyRunnable implements runnable
{  …
   run() {…}
}
```

- Instantiate the runnable, instantiate and start a thread that runs the runnable

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Process Interactions

- Competition/Mutual Exclusion
  - Example: Two processes both want to access the same resource.
- Cooperation
  - Example:

$$\textit{\textbf{Producer}} \rightarrow \textit{\textbf{Buffer}} \rightarrow \textit{\textbf{Consumer}}$$

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

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

- Problem statement:

```
cobegin
p1: while(1) {CS_1; program_1;}
    //
p2: while(1) {CS_2; program_2;}
    //
    ...
    //
pn: while(1) {CS_n; program_n;}
coend
```

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

The Critical Section Problem

In addition to mutual exclusion, 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

Algorithm 1

- Use a single turn variable:

```
int turn = 1;
cobegin
p1: while (1) {
      while (turn != 1); /*wait*/
      CS_1; turn = 2; program_1;
      }
 //
p2: while (1) {
      while (turn != 2); /*wait*/
      CS_2; turn = 1; program_2;
      }
coend
```

- Violates blocking requirement (1), "dog in manger"

Algorithm 2

- 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*/
        CS_1; c1 = 0; program_1;
        } //
p2: while (1) {
        c2 = 1;
        while (c1); /*wait*/
        CS_2; c2 = 0; program_2;
        }
coend
```

- Violates blocking requirement (3), deadlock. Processes may wait forever.

Algorithm 3

- 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 {CS_1; c1 = 0; program_1}
      }  //
p2: while (1) {
      c2 = 1;
      if (c1) c2 = 0; //go back, try again
      else {CS_2; c2 = 0; program_2}
      }
coend
```

- Violates blocking requirements (2) and (4), fairness and livelock

Peterson's algorithm

- Processes indicate intent to enter CS as in #2 and #3 (using c1 and c2 variables)
- After a process indicates its intent to enter, it (politely) tells the other process that it will wait (using the willWait variable)
- It then waits until one of the following two conditions 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.)

Peterson's Algorithm

```
int c1 = 0, c2 = 0, willWait;
cobegin
p1: while (1) {
      c1 = 1; willWait = 1;
      while (c2 && (willWait==1)); /*wait*/
      CS_1; c1 = 0; program_1;
      }
 //
p2: while (1) {
      c2 = 1; willWait = 2;
      while (c1 && (willWait==2)); /*wait*/
      CS_2; c2 = 0; program_2;
      }
coend
```

- Guarantees mutual exclusion *and* no blocking
- Assumes there are only 2 processes

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

Code for Bakery Algorithm (First cut)

```
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)  {
     program_i

     // Step 1: choose a number
     x = 0;
     for (j=0; j < n; j++)
        if (j != i)  x = max(x,number[j]);
     number[i] = x + 1;

     // Step 2: 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]))
     CS_i

     number[i] = 0;
  }
```

Bakery algorithm, continued

- Complication: there could be ties in step 1. This would cause a deadlock (why?)
- Solution: if two processes pick the same number, give priority to the process with the lower process number.

Correct 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)  {

     program_i

     // Step 1: choose a number
     x = 0;
     for (j=0; j < n; j++)
        if (j != i)  x = max(x,number[j]);
     number[i] = x + 1;

     // Step 2: 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)))
     CS_i

     number[i] = 0;
  }
```

Software solutions to Critical Section problem

- Drawbacks
  - Difficult to program and to verify
  - Processes loop while waiting (busy-wait). Wastes CPU time.
  - Applicable to only to critical section problem: (competition for a resource). Does not address cooperation among processes.
- Alternative solution:
  - special programming constructs (semaphores, events, monitors, …)

Semaphores

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

  P(s): if **s>0**, decrement **s** and proceed;
  else wait until **s>0** and then decrement **s** and proceed

  V(s): increment **s** by 1
- Equivalent Semantics:

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

  V(s): s=s+1;
- The operations P and V are *atomic* (indivisible) operations

Notes on semaphores

- Invented by Dijkstra
- As we will see in Chapter 4, the waiting in the `P` operation can be implemented by
  - Blocking the process, or
  - Busy-waiting
- Etymology:
  - `P(s)`, often written `Wait(s)`; think "Pause":
    "P" from "*passaren*" ("pass" in Dutch) or from "*prolagan*," combining "*proberen*" ("try") and "*verlagen*" ("decrease").
  - `V(s)`, often written `Signal(s)`:
    think of the "V for Victory" 2-finger salute:
    "V" from "*vrigeven*" ("release") or "verhogen" ("*increase*").

Mutual Exclusion w/ Semaphores

```
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

- Cooperating processes must also synchronize
- Example: **`P1`** waits for a signal from **`P2`** before **`P1`** proceeds.
- Classic generic scenario:

  Producer → Buffer → Consumer

Signal/Wait with Semaphores

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

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

History

- Originally developed by Steve Franklin
- Modified by Michael Dillencourt, Summer, 2007
- Modified by Michael Dillencourt, Spring, 2009
- Modified by Michael Dillencourt, Winter, 2010
- Modified by Michael Dillencourt, Summer, 2012