# Interprocess Synchronization and Communication

## Goal

To understand how these are implemented we need to look at what they are and why they are necessary.

## Parallelism

### What Is It?

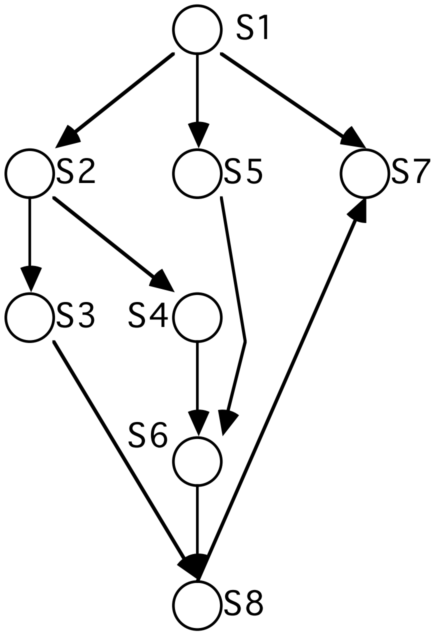
This is having statements execute simultaneously. Why? The system may have multiple CPUs or may have special units such as Floating Point Units which can compute at the same time as the CPU. *Problem*: some statements must be completed before others may be begun; for example, consider

```
1  a: = x+y
2  b: = z+1
3  c: = a-b
4  d: = c+1
```

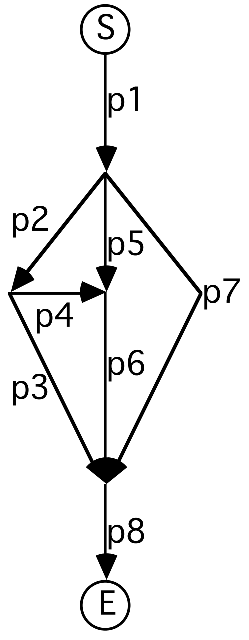
1 and 2 must finish before 3 can begin, but 1 and 2 can be done independently. This is a *precedence constraint* .

*Precedence graphs* and *process flow graphs* are used to examine processes for parallel work.
*example*: The following two graphs represent the same thing:

precedence graph



process flow graph

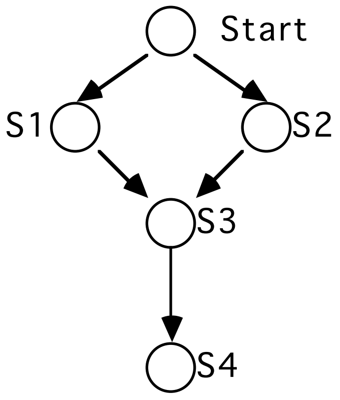


These are really equivalent (one focuses on statements, the other on processes). These graphs must be acyclic.

### Bernstein conditions

These describe when statements can be executed in parallel. Define

$R(S_i)$ = { variables whose value is referenced in statement $s_i$ }
$W(S_i)$ = { variables whose value is changed in statement $s_i$ }
For example, in the example above
$R(S_1) = \{x, y\}$ $R(S_2) = \{z\}$ $R(S_3) = \{a, b\}$ $R(S_4) = \{c\}$
$W(S_1) = \{a\}$ $W(S_2) = \{b\}$ $W(S_3) = \{c\}$ $W(S_4) = \{d\}$
precedence graph:



Now, Si and Sj may be done concurrently if the Bernstein conditions hold:
$W(S_i) \ldots W(S_j) = \varnothing$ and $R(S_i) \ldots W(S_j) = \varnothing$ and $W(S_i) \ldots R(S_j) = \varnothing$
In the above example, this agrees with the intuition.

## Parallel Programming Constructs

### *fork, join, quit*

The earliest parallel constructs were *fork* and *join*. *Fork* looks like `fork label` and splits the process into two, one of which begins at label and the other which falls through.
*example*:

```
         fork L;
         a := x+y;
         …
      L: b := z+1;
```

*Join* merges processes and has the form `join count label`; count is decremented and if zero, there is a branch to *label*: That is,

```
count := count - 1;
if count = 0 then goto label;
```

(Note: in some texts, process terminates instead of branches.)
*Quit* terminates process.
*example*:

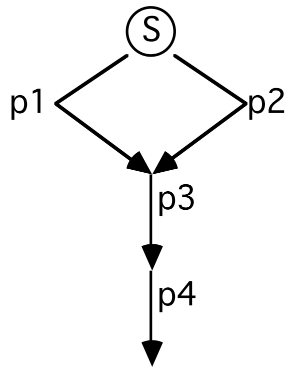
```
            count =2;
            fork dopar;
            a := x+y;
            go to donepar;
dopar:      b := z+1;
donepar:    join count, cont;
            quit;
cont:       c := a-b;
```

```
                    d := c+1;
```
As a second example, the program equivalent to the process flow graph above is:
```
          t6 := 2;
          t8 := 3;
          S1; fork p2; fork pt; fork p7; quit
      p2: S2; for L p3: fork p4; quit;
      p5: S5; join t6, p6; quit
      p7: S7; join t8, p8; quit
      p3: S3; join t8, p8; quit
      p4: S4; join t6, p6; quit
      p6: S6; join t8, p8; quit;
      p8: S8; quit
```
where Si is the program for pi.
*Strengths*: simple, powerful, easy to derive from precedence graphs
*Weakness*: clumsy, lots of gotos and goto like structures.

## *parbegin,parend*

Dijkstra suggested these as a goto-less mechanism; they bracket statements (or blocks of statements) to be done in parallel.
*example*:
```
                    parbebin
                         a: = x+y;
                         b: = z+1;
                    parend
                    c: = a-b;
                    d: = c+1;
```
*Strength*: easy to read, uses principles of modular programming, etc.
*Weakness*: not so powerful as the *fork-join-quit* set.

To see why, consider the idesa of *proper nesting*. Let
S(a,b) represent the serial execution of processes a and b
P(a,b) represent the parallel execution of processes a and b
Then a process flow graph is properly nested if it can be described by P, S, and functional composition.
*example*: the first example's process flow graph is



which is S(S(P(p1, p2), p3, p4)
But the second example cannot be so described.

**Claim** The second example is not properly nested.

**Proof** For something to be properly nested, it must be of the form S(pi, pj) or P(pi, pj) at the most interior level.

Clearly the example's most interior level is not P(pi, pj) as there are no constructs of that form in the graph.

In the graph, all serially connected processes pi and pj have at least 1 more process pk starting or finishing at the node nij between pi and pj; but if S(pi, pj) is in the innermost level, there can be no such pk (else a more interior P or S is needed, contradiction). Hence, it's not S(pi, pj) either.

Clearly, *parbegin-parend* works only when the process flow graph is properly nested.

## The problem with process interaction

This is best illustrated by an example, called the *bounded buffer producer/consumer problem*; we use a buffer of n items.

```
     var  buffer: array [0..n-1] of item;
          in, out: 0...n-1;
          counter: 0...n
producer: repeat
               make next p;
               while counter = n do (* nothing *);
               buffer [in] := next p;
               in := (in+1) mod n;
               counter: = counter + 1;
          until false;
consumer: repeat
               while counter = 0 do (* nothing *);
               next: = buffer[out];
               out: = (out + 1) mod n
               counter: = counter - 1;
          until false;
```

If each loop is executed separately, this works fine; but if intermingled ...

Suppose `counter` is 3, and loot at `counter: = counter + 1` and `counter: = counter - 1`. These might compile into the following:

```
counter: = counter + 1            counter: = counter - 1
C1   r1: = counter;               P1   r2 : = counter;
C2   r1: = r1 + 1;                P2   r2: = r2 - 1;
C3   counter: = r1;               P3   counter: = r2;
```

depending on how these sets of statements intermingle, counter could have one of three values:

```
               P1 P2 C1 C2 P3 C3     counter = 4
               P1 P2 C1 C2 C3 P3     counter = 6
               P1 P2 P3 C1 C2 C3     counter = 5
```

Why? 2 processes manipulated `counter` simultaneously. Clearly, we need to ensure just one process does.

## The Critical Section Problem and Its Solutions

### Critical Section Problem

A *critical section* is a block of code that only one process at a time can execute; so, when one process is in its critical section, no other process may be in its critical section.
*Problem*: design a protocol to solve this problem.
*Generic description of framework*: Every solution will have the following layout:

```
entry section
critical section
exit section
remainder section
```

### Requirements to be a Solution

1. *Mutual exclusion*; at most one process in the critical section at a time.
2. *Progress* ; if no process in the critical section, and some wish to enter, then only processes **not** in remainder section can take part in deciding whichj one enters.
3. *Bounded Wait*; a bound on the number of times other processes are allowed to enter the critical section after a process asks to enter its critical section and before it is allowed to.

*Assumption made implicitly*: each process runs at nonzero speed, but **no** assumption is made as to relative speed.

### Two-Process Solutions

We will use processes p0 and p1, also known as pi and pj, with either i = 0 and j = 1 or with i = 1 and j = 0. The current process is always pi; the other one, pj.

### Analyses of Sample Solutions

Our goal is to learn how to analyze a proposed solution. The best way to do this is by examples:

*Example 1.*

```
var  turn: 0..1;          (* whose turn it is *)
     while turn ≠ i do    … entry section
          (* nothing *);
                          … critical section
     turn := j;           … exit section
```

Is mutual exclusion satisfied? As `turn` can have only 1 value, it is.

As for progress, note the processes enter their critical section in alternate order; hence progress is not met.

*Example 2.*

```
    var  inCS: array[0..1] of booleans := false;
                     (* who is in critical section *)
         while inCS[j] do    … entry section
              (* nothing *);
         inCS[i] := true;
                             … critical section
         inCS[j] := false;   … exit section
```

Is mutual exclusion satisfied? Suppose pi and pj both execute the **`while`** statement at the same time. As both `inCS[i]` and `inCS[j]` are `false`, both proceed through, set `inCS[i]` and `inCS[j]` to `true`, and enter their critical section. So mutual exclusion is violated.

*Example 3.*

```
    var  interested: array[0..1] of booleans := false;
                (* who wants to enter critical section
    *)
         interested[i] := true;         … entry section
         while interested[j] do
              (* nothing *);
                             … critical section
         interested[j] := false;        … exit section
```

Is mutual exclusion satisfied? The only way into the critical section is if `interested[j]` is `false`; but if a process is in the critical section, `interested[j]` must be `true`. Hence at most one process can be in the critical section at a time, proving mutual exclusion holds.

Is progress satisfied? Suppose both processes hit the **`while`** loop at the same time. Then they loop forever. So progress is not satisfied.

*Example 4.*

```
    var  interested: array[0..1] of booleans := false;
                (* who wants to enter critical section
    *)
         turn: 0..1;
         interested[i] := true;         … entry section
         turn := j;
         while interested[j] and turn = j do
              (* nothing *);
                             … critical section
         interested[j] := false;        … exit section
```

Is mutual exclusion satisfied? Note first that pi enters its critical section only if `interested[j]` is `false` and `turn` is `i`. For both to be in the critical section, `interested[i]` and `interested[j]` are both `true`.

Both could not have passed through the **while** loop at the same time (as turn is i or j but not both), so one did the loop while the other did one of the preceding lines. Say pi does the preceding lines and pj is in the loop. After doing the first line in the entry section, interested[i] and interested[j] are both true, and turn is j, so pi stops at the **while** and the next time pj goes, it enters the critical section; in short, at most one process can be in the critical section. So mutual exclusion holds.

Are bounded wait and progress both satisfied? First, notice that pi is blocked from entering the critical section only if it is stuck at the while loop, which means that interested[j] is true and turn is j. If pj is not in the entry or critical sections, interested[j] is false and pi goes in. If pj is at the **while** statement in the entry section, turn is either i or j, and the process with index the same as turn will go in. Now, once pi leaves the critical section, interested[i] is false and pj can go in (or vice versa). If pj resets interested[j] is true, then turn will be set to i and pi goes in. This means that only the processes in the entry, exit, or critical sections affect which process goes in, giving progress. Further, at most one additional entry by pj will occur if both request entry at the same time, so bounded waiting holds.

Hence this is a solution to the critical section problem; in fact, it is Peterson's Solution.

## N-process Solution

Lamport's Bakery Algorithm (see handout) is one solution.

Is mutual exclusion satisfied? let Pi be in the critical section. If Pk ($k \neq i$) has chosen number[k] $\neq 0$, (number [i],i) < (number [k],k). Now, suppose Pk trying to enter, and Pi is in the critical section. When Pk runs with j = i, it sees number [i] $\neq 0$ and (n[i], i) < (n[k], k) and therefore loops.

Are bounded wait and progress satisfied? Yes, as processes enter the critical section on FIFO basis.

## Hardware Solutions

The critical section problem is easier to solve using hardware instructions, for example the atomic test-and-set instruction:

```
function TaS(var Lock: boolean): boolean
begin
     TaS: = Lock;
     Lock = True;
end;
```

which is used to solve the N-process problem (see handout)

All of the above solutions (software and hardware) have problems:

1. busy waiting; the CPU does nothing in such a way that no-one else can use it
2. complex, not easily generalizable; for example, Peterson's solution does not easily generalize to N processes.

To solve these, we can define other hardware constructs. For example:

## semaphores

### semaphores

This is a non-negative integer variable S that can be operated on in 3 ways.

1. *initialization* operation sets initial value atomically:
```
                S := n;
```
2. *signal* or *V* operation increments value atomically:
```
                S := S + 1;
```
3. *wait* or *P* operation blocks until S is non-zero, then decrements value; the whole thing is atomic:
```
                while S = 0 do block;
                S := S - 1;
```

Mathematically, let:

| | |
|---|---|
| ns | number of signals |
| nw | number of completed wait operations (*i.e.*, not suspended) |
| iv | initial value |
| vs | value of semaphore |

Then for a semaphore mutex, the definitions give:

$$\mathrm{vs}(\mathtt{mutex}) = \mathrm{iv}(\mathtt{mutex}) + \mathrm{ns}(\mathtt{mutex}) - \mathrm{nw}(\mathtt{mutex})$$

but as vs ≥ 0, we also have

$$\mathrm{nw}(\mathtt{mutex}) < \mathrm{ns}(\mathtt{mutex}) + \mathrm{iv}(\mathtt{mutex})$$

Both of these are always true, and hence are *invariants*. So, for mutual exclusion:
```
        P(mutex); (* critical section  *); V(mutex);
```
If the initial value of mutex is 1, then the second invariant gives

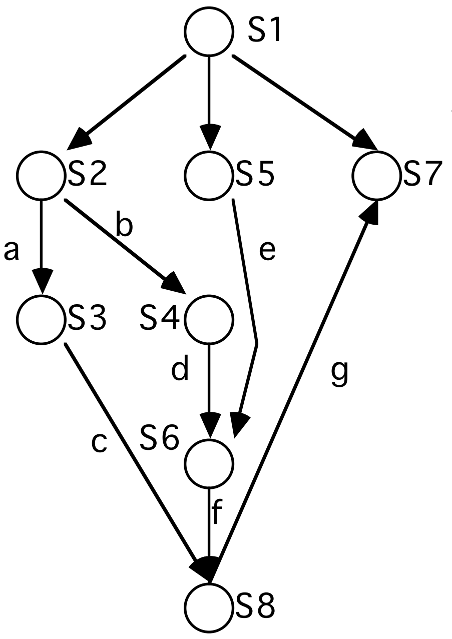
$$\mathrm{nw}(\mathtt{mutex}) < \mathrm{ns}(\mathtt{mutex}) + 1$$

meaning that at most 1 process will be in the critical section at a time (giving mutual exclusion).

2) Synchronization:
```
        A           B
       ...         ...
    P(mutex)    V(mutex)
       ...         ...
```
If the initial value of mutex is 0, process A will block at P(mutex) until process B reaches V(mutex); and nw(mutex) ≤ ns(mutex).

*example*: Recall the precedence graph

precedence graph



```
S1;
parbegin
 begin S2; V(a); V(b); end;
 begin P(a); S3; V(c); end;
 begin P(b); S4; VB(d); end;
 begin S5; V(e); end;
 begin P(d); p(e); S6; V(f); end;
 begin S7; V(g); end;
 begin P(c); P(f); P(g); S8; end;
parend;
```

The implementation is usually First-In-First-Out queues; the first process blocked is the first one allowed to continue.

## Testing Synchronization Primitives

### Bounded Buffer Problem

Processes communicate through a buffer of fixed length. The *producer* inserts items and the *consumer* extracts them.
*Problem*: prevent buffer overflow or underflow.
Assume:

| | |
|---|---|
| n | size of buffer |
| d | number of items deposited |
| e | number of items extracted |

Obviously, $0 \le d - e \le n$ must hold.
From the order of operations:

in the producer: $ns(\texttt{full}) \le d \le nw(\texttt{empty})$
in the consumer: $ns(\texttt{empty}) \le e \le nw(\texttt{full})$

Hence

$$d \le nw(\texttt{empty}) \le ns(\texttt{empty}) + n \le e + n$$
$$e \le nw(\texttt{full}) \le ns(\texttt{full}) \le d$$

Combining, we have

$$e \le d \le e + n$$

### Readers - Writers Problem

Here, a file is to be shared among several concurrent processes, each of which is only interested in reading (the *readers*) or writing (the *writers*).
*Problem*: to enforce Bernstein's conditions.
The nub is that writers need exclusive access to the file. This suggests two variants, based on priority of the writers:
*The first readers-writers problem*: Readers have priority. Hence even if a writer is waiting for the file, a new readers may start reading the file provided another reader is also currently reading the file. This means that writers may starve
*The second readers-writers problem*: Writers have priority. Hence once a writer is waiting for the file, no new readers may start reading.
A solution to the first problem is in the handout.

### The Dining Philosophers Problem

Five philosophers are dining at a circular table. They have five plates (one in front of each philosopher) and five forks, one between each plate. They alternate between thinking and using both their right and left forks to eat.
*Problem*: prevent starvation and deadlock.
*Example 1 of a Potential Solution*:
Each philosopher picks up her left fork first:

```
var  fork:  array [0..4] of semaphore: = 1,1,1,1,1
repeat     (* philosopher i *)
     P(fork[i]);
     P(fork[i + 1 mod 5]);
          (* eat *)
     V(fork[i]);
     V(fork[i + 1 mod 5]);
          (* think *)
until false
```

This leads to deadlock.

## Abstraction

### Problem with Semaphores

The problem is similar to that of *fork/join/quit*. P and V are too low-level. Also, semaphores combine blocking with counting these are distinct functions, which could be kept separate. Finally, they are murder to debug; try typing a P for a V and then figure out where you did it!
*Solution*: build some higher level constructs

### Abstract Datatypes

The theory of programming languages gives a technique to do this. A *class* is a module giving representation of an abstract data type
*example*:
```
type stack = class
     var  stack: array [1.. 100] of integer;
          top: integer;
     procedure entry push(x: integer);
     begin
          if (checkpush()) then
          begin
               stack[top] := x;
               top := top + 1;
          end;
     end;
     procedure entry pop(var x: integer);
     begin
          if (checkpop()) then
          begin
               x := stack[top];
               top := top - 1;
          end;
     end;
     function checkpush(): boolean;
     begin
          checkpush := top <=100;
     end;
     function checkpop() = boolean,
     begin
          checkpop := top > 1;
     end;      (* initialization *)
     begin
          top := 1;
     end;
```
Then you declare an *instance* of this type by
```
var calcstack: stack;
```
The *type* is the template, or definition; the *instance* is a variable of that type.

## Monitors (Hoare)

Monitors are defined like a class, but they *guarantee* mutual exclusion; only 1 process may be active in the monitor. So, when using one:

1. access to the encapsulated resource should be possible **only** via the monitor
2. procedures in the monitor are mutually exclusive; when 1 process is executing within the monitor, other processes calling procedures within the monitor are delayed until that process leaves the monitor

So far, these are similar to critical regions. But to synchronize, define *condition variables* and 2 operations:

*wait* on a condition variable (`x.wait`) - block process, put it on queue associated with condition `x`

*signal* on a condition variable (`x.signal`) - if any process is blocked on condition `x`, unblock **one** of them; if not, ignore the signal.

Note that the *signal* operation is memoryless (that is, if no-one is blocked, the signal disappears and the next one to try to wait will **`wait`**); different than semaphores (not memoryless).

*Problem* with *signal*: suppose P1 is blocked on `x`, and P2 signals on `x`. Since only one process can be in the monitor at a time, one must block until the other is done. Two methods are followed:

1. P2 waits until P1 either leaves monitor or waits for a condition (Hoare). This gives simpler, more elegant proofs.
2. P1 waits until P2 either leaves monitor or waits for a condition (Lampson & Redell). The programming language Mesa uses this. One problem is that the "logical" condition for which P1 was waiting may no longer be true when P2 leaves monitor; hence under this scheme, the monitor must say
   ```
   while not B do x.wait;
   ```
   and not
   ```
   if not B do x.wait;
   ```

*example*: Binary semaphore with a monitor.

```
semaphore:    monitor;
        var  busy: boolean;
             notbusy: condition
        procedure entry P;
        begin
             if busy then
                  notbusy.wait;
             busy: = true;
        end;
        procedure entry V;
        begin
             busy: = false;
```

```
                        notbusy.signal;
                    end;
                    begin
                        busy:= false;
                    end.
```

*example*: producers/consumers problem (see handout)
*example*: first readers-writers problem (see handout)

*Implementation*: Here we consider Hoare's. We must assure:

1. execution of procedures in the monitor is mutually exclusive;
2. wait blocks the current process on corresponding condition
3. if a process exits or is blocked, and other processes are waiting to enter the monitor, one must be selected; priority goes to those blocked after issuing a signal, then those waiting to get in go, then those blocked on a condition variable.
4. if a process issues a signal, it must determine if any process is waiting on the corresponding condition; if so, current process is suspended and a waiting one activated; else, the signaller continues.

See handout for how it is done.

*Priority Waits*

Recall that signals restart processes in FIFO order, but sometimes this is not good; for example, think of waiting for a specific time of day. Hence define *priority wait* as:

c.**wait**(p)

where `p` is an integer and `c` a condition variable. If more than one process waiting on `c` is signalled, the one with the lowest value of `p` resumes.
*example*: alarm clock (see handout).

## Event Counters and Sequencers

These allow synchronization without mutual exclusion (but they can provide mutual exclusion too). There are two parts:

*Event counters* are non-decreasing integers beginning at 0. Three operations (here, `E` is an event counter):

| | |
|---|---|
| `advance(E)` | `E := E + 1` atomically; indicates an event of interest occurred |
| `read(E)` | `return(E)`; so, if `E` is n, at least n `advance(E)` operations occurred |
| `await(E,v)` | block until `E` has value `v` so this continues only when at least `v` `advance(E)` operations occurred. |

*Sequencers* also are non-decreasing integers beginning at 0. These are used to order events, and one operation only is defined (here, `S` is a sequencer):

`ticket(S)`

```
olds := S;
S := S + 1;
return(oldS);
```

executed atomically; this requires mutual exclusion so no two calls will return the same value.

Mutual exclusion:

```
await (E, ticket (S));
…
advance(E);
```

*example*: producer-consumer problem (see handout)

## Shared Memory Synchronization

There are some cases where none of the above mechanisms are satisfactory:

1. Security considerations may prevent sharing memory
   *example*: each process must run in strict isolation, in its own logical space with all interactions under its own control; this is not possible with monitor, as any process with access to the monitor can get global data stored within the monitor.
2. It may not be possible to share memory
   *example*: in a distributed system, each processor may have its own local memory and so processes on different processors cannot share data.

In these cases a mechanism other than those based on shared memory must be used; these new schemes are called *message-based synchronization schemes*.

### Interprocess Communication (IPC)

Two primitives:

| | |
|---|---|
| send(p, msg) | transfers message msg to process p; special (implementation-dependent) values of p can be used to indicate that the message goes to all processes; this is called *broadcast*. |
| receive(q, msg) | obtains message msg from process q; special (implementation-dependent) values of q can be used to indicate that the message goes to all processes. |

The answers to four basic questions characterize send/receive primitives:

1. Does the sender wait until its message is accepted by the recipient, or does it continue processing?
   - if the sender blocks, the send is called *blocking* or *synchronous*
   - if the sender may proceed while the message is being delivered, send is *non-blocking* or *asynchronous*
2. What happens when a receive call is issued, but there is no message waiting?
   - if the process waits for a message to arrive, the receive is called *blocking* or *synchronous*
   - if the process continues, the receive is called *nonblocking* or *asynchronous*

   A related question is the size of the (system) queue used to hold messages in transit. This queue, associated with the connection or *link* between the two processes, has a capacity for a certain number of messages; the capacity is a property of the link. There are three different types of implementations:

1. A *zero capacity* link: the link cannot have any messages waiting; the sender *must* wait until the recipient gets the message, or the message is lost. It is most useful when the process transmits the message from a buffer within the process (called *rendezvous*).
2. A *bounded capacity* link: If the capacity is n, then at most n messages can be stored in the associated queue. If the queue is not full, the message is copied into the queue. If it is full, the sender must wait (or the message will be lost).
3. An *unbounded capacity* link: any number of messages can be stored in the associated queue.

3. Must the sender specify *exactly* 1 recipient, or can messages be sent to any (or all) of a number of recipients?
4. Must the recipient specify *exactly* 1 sender, or can messages be accepted from any (or all) of a number of senders?

*Naming*

There are two types:

1. The sender or recipient is specified; called *explicit naming* or *direct communication*.
   Relevant properties:
   - the link between pairs of processes wanting to communicate is established automatically; the processes need to know each other's identity *only*.
   - each link is associated with exactly 2 processes.
   - between each pair of communicating processes, there is exactly one link
   - the link is bidirectional

   *example*: the producer/consumer problem (see handout)
   A variant of this scheme is that the sender specifies the recipient, but the recipient gets messages from any sender; on return of the receive call, the process argument in the call is set to the name of the sending process
   *Problem*: lack of modularity, as if a process changes its name, all references to it must also be changed.
2. Messages are sent to *mailboxes* or drop boxes; called *implicit naming* or *indirect communication*.
   Relevant properties:
   - there is a link between a pair of processes only if there is something shared (like a mailbox)
   - a link is associated with any number of processes

- between each pair of communicating processes, there may be many links (specifically, 1 per mailbox)
- a link may be unidirectional or bidirectional

*example*: the producer/consumer problem (redo handout)

*Problem*: If two processes do a receive on a mailbox at the same time, who gets the message? Three possibilities:

1. Each link is associated with exactly 2 processes so the problem should never arise.
2. Only 1 process at a time may do a receive on a particular mailbox; in this case, the mailbox is called a *port*.
3. The system selects which process (but not both of them!) gets the message.

How do you create a mailbox? It can be done:

- within the process, by declaring the mailbox (like you declare a variable to create it); that process gets all messages sent to the mailbox, and when the process dies, the mailbox goes away.
- by the operating system, which provides system calls to create and delete a mailbox. Usually the ability to receive messages from a particular mailbox can be passed to another process via system calls. May require garbage collection.

Other issues include:

- communications delay; onbe solution is to send until a reply (*acknowledgement*) is received.
  *example*: Toth, internet higher level protocols (SMTP, TCP)
- process termination before message processed
  1. if a recipient process P1 is waiting for a message from a terminated process P2 using a blocking receive, P1 is blocked forever.
  2. if a sender process P1 sends a message to a terminated process P2 using a blocking send on a zero-capacity link, P1 is blocked forever.

  In both cases, the solution is to notify P1 that P2 has terminated, or terminate P1
- messages lost in transfer
  1. The operating system may be responsible for detecting this and retransmitting the message or notifying the sender.
  2. The sender may detectthis; it can resend the message.

  Timeouts are used to detect this; one problem is they may be too short, so messages are unnecessarily retransmitted.
- messages may be garbled or altered in transit
  These can be detected by using message integrity codes such as checksums or CRCs.

## Remote Procedure Calls (RPC)

The send/receive mechanism has the same problems as semaphores; they are too low-level, so the user must suce them like P and V and abandon the idea of procedures. Remote procedure calling provides this procedural interface.

*Programmer view*
A remote procedure call is just like a regular procedure call, except the procedure is in a separate address space and doesn't share global variables

*Implementation view*
Each remote procedure needs a separate process ; this process can be created by the call to the procedure, or it can be a permanent process (below, called an *RP_guard*) that reads parameters, runs the remote procedure, and returns its result using send and receive primitives:

$$\begin{tikzcd}
\text{caller process} & \text{RP\_guard process} \\
\text{send(RP\_guard, parameters)} \arrow[r] & \text{receive(caller, parameters)} \arrow[d] \\
 & \text{RP(parameters)} \arrow[d] \\
\text{receive(RP\_guard, results)} & \text{send(caller, results)} \arrow[l]
\end{tikzcd}$$

*Example*: the programming language ADA™
The *accept* statement designates a segment of code as a remote procedure:

```
    accept name(formal_parameter_list) do proc-body end
```

To call, the caller uses

```
                        name(actual_parameters)
```

- If the caller issues a call before the process containing the definition of the called function hits the corresponding *accept*, the caller blocks. When the process with the called function hits the *accept*, it executes the statement body, and sends the results to the caller. Then both go on.
- If the process containing the definition of the function hits *accept* first, it blocks until the caller issues the corresponding remote

procedure call; it then proceeds, sends the result to the caller, and goes on.

Thus the *accept* mechanism is like blocking receive with explicit naming. We would like to wait for any of several possible requests so that the remote procedure could be shared by many remote procedures (that is, one RP_guard procedure that will call on many different functions). The *select* statement does this; it associates with each *accept* a Boolean condition. If the Boolean condition is false when the *select* is executed, the corresponding *accept* cannot be done.

```
select    [when B1:] accept E1(…) do S1 end
or when B2:] accept E2(…) do S2 end
…
or [when Bn:] accept En(…) do Sn end
else      R
end;
```

If the *else* is omitted, and none of the Booleans are true, an error is generated. If more than one Boolean is true, the systems is assumed to choose among the possibilities according to a fair internal policy.

What characterizes a "fair internal policy?" Which process is given the CPU next? This is the province of *schedulers.*