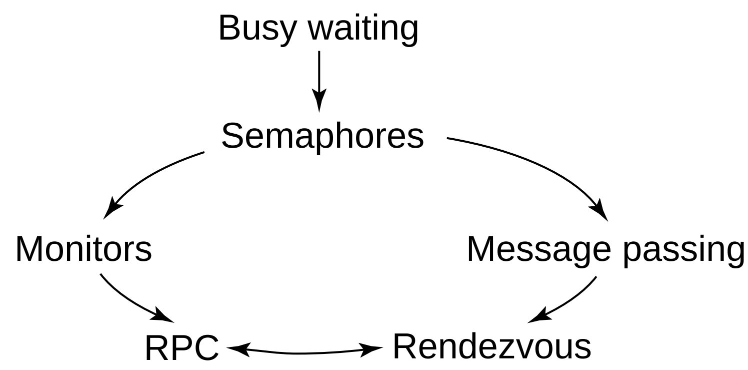
Distributed Programming

- Distributed memory architectures.
- Concurrent programs here are usually called *distributed* programs.
- No shared variables.
  - No mutual exclusion necessary!
- Communicate and synchronize with *channels*:
  - one-way or two-way
  - synchronous (blocking) or asynchronous (nonblocking)
- Four basic mechanisms:
  - Chapter 7:
    - asynchronous message passing
    - synchronous message passing
  - Chapter 8:
    - RPC (remote procedure call)
    - rendezvous
- Chapter 9 describes several paradigms for distributed programming:
  - managers/workers, hearbeat, pipeline, probe/echo, broadcast, token passing, decentralized servers



Relationships between programming mechanisms.

- Monitors combine implicit exclusion with explicit signalling
- Message passing adds data to semaphore
- RPC and Rendezvous combine procedural interface of monitors with implicit message passing

Message Passing

Andrews, Chapter 07

- Asynchronous message passing: channels are like semaphores.
- `send` and `receive` are like `V` and `P`
- `receive` is *blocking*
- May want to avoid blocking: `empty(ch)`
  - use with caution: may be unreliable
- The number of queued "messages" is the value of the semaphore.
- We assume messages are atomic and delivery is reliable and error-free.
- Three important types:
  - **Mailboxes**: any process may send or receive.
  - **Input port**: one receiver, many senders.
  - **Link**: one receiver, one sender.

```
chan input(char), output(char [MAXLINE]);
process Char_to_Line {
  char line[MAXLINE]; int i = 0;
  while (true) {
    receive input(line[i]);
    while (line[i] != CR and i < MAXLINE) {
      # line[0:i-1] contains the last i input characters
      i = i+1;
      receive input(line[i]);
    }
    line[i] = EOL;
    send output(line);
    i = 0;
  }
}
```

Figure 7.1 Filter process to assemble lines of characters.

- Channels like this are called *mailboxes*

Filters: Sorting

```
process Sort {
    receive all numbers from channel input;
    sort the numbers;
    send the sorted numbers to channel output;
}
```

- Suitable for "heavyweight" processes.
- Alternative: network of lightweight *merge* processes.
  - suitable for direct implementation in hardware.
- Construct a *merge network*.

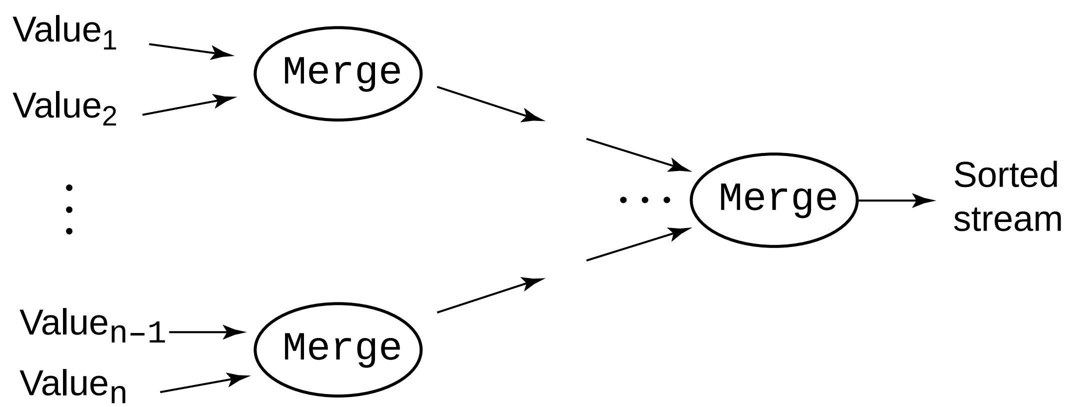


Figure 7.3 A sorting network of **`Merge`** processes.

- Original values can be lists sorted by "mediumweight" processes.
- Nodes can wait for all values and then merge them.
- More parallelism if they merge as values come in.

```
chan in1(int), in2(int), out(int);
process Merge {
  int v1, v2;
  receive in1(v1);  # get first two input values
  receive in2(v2);
  # send smaller value to output channel and repeat
  while (v1 != EOS and v2 != EOS) {
    if (v1 <= v2)
      { send out(v1); receive in1(v1); }
    else   # (v2 < v1)
      { send out(v2); receive in2(v2); }
  }
  # consume the rest of the non-empty input channel
  if (v1 == EOS)
    while (v2 != EOS)
      { send out(v2); receive in2(v2); }
  else   # (v2 == EOS)
    while (v1 != EOS)
      { send out(v1); receive in1(v1); }
  # append a sentinel to the output channel
  send out(EOS);
}
```

Figure 7.2 A filter process that merges two input streams.

- **Active Monitor**

```
chan request(int clientID, types of input values);
chan reply[n](types of results);

process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) {   ## loop invariant MI
    receive request(clientID, input values);
    code from body of operation  op;
    send reply[clientID](results);
}

process Client[i = 0 to n-1] {
  send request(i, value arguments);     # "call" op
  receive reply[i](result arguments);   # wait for reply
}
```

Figure 7.4 Clients and server with one operation.

- Like a monitor with one method: `op`
- Client would call `m.op(values, results)`
- Except, here clients can do other things between `send` and `recieve`.
- Clients need to send ID.
- Monitor invariants become loop invariants.

```
type op_kind = enum(op1, ..., opn);
type arg_type = union(arg1, ..., argn);
type result_type = union(res1, ..., resn);
chan request(int clientID, op_kind, arg_type);
chan reply[n](res_type);
process Server {
  int clientID; op_kind kind; arg_type args;
  res_type results; declarations of other variables;
  initialization code;
  while (true) {    ## loop invariant MI
    receive request(clientID, kind, args);
    if (kind == op1)
      { body of op1; }
    ...
    else if (kind == opn)
      { body of opn; }
    send reply[clientID](results);
  }
}
process Client[i = 0 to n-1] {
  arg_type myargs; result_type myresults;
  place value arguments in myargs;
  send request(i, opj, myargs);    # "call" opj
  receive reply[i](myresults);     # wait for reply
}
```

- No condition variables yet.

Figure 7.5 Clients and server with multiple operations.

```
monitor Resource_Allocator {
  int avail = MAXUNITS;
  set units = initial values;
  cond free;   # signaled when a process wants a unit
  procedure acquire(int &id) {
    if (avail == 0)
      wait(free);
    else
      avail = avail-1;
    remove(units, id);
  }
  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail = avail+1;
    else
      signal(free);
  }
}
```

Figure 7.6 Resource allocation monitor.

- Passing the condition:

  good for translating into server code.

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitid);
chan reply[n](int unitID);

process Allocator {
  int avail = MAXUNITS; set units = initial values;
  queue pending;  # initially empty              ⇐ Queue needed for wait
  int clientID, unitID; op_kind kind;
  declarations of other local variables;
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind == ACQUIRE) {
       if (avail > 0) {  # honor request now
          avail--; remove(units, unitID);
          send reply[clientID](unitID);
       } else   # remember request
          insert(pending, clientID);
    } else {   # kind == RELEASE
       if empty(pending) {  # return unitID to units
          avail++; insert(units, unitid);
       } else {  # allocate unitID to a waiting client
          remove(pending, clientID);
          send reply[clientID](unitID);
       }
    }
  }
}
process Client[i = 0 to n-1] {
  int unitID;
  send request(i, ACQUIRE, 0)    # “call” request
  receive reply[i](unitID);
  # use resource unitID, then release it
  send request(i, RELEASE, unitID);
   ...
}
```

- Only works *without* the “`while(!B) wait(cv);`” pattern.

Figure 7.7 Resource allocator and clients.

Other monitor code

- `wait` in a loop:
  - queue requests, as before
  - add code for what actions need to be taken when client wakes
  - different code for different wait statements depend on client info when queued
- Unconditional `signal`:
  - check queue
  - if nonempty, process it *after* finishing the `signal`
  - in other words: signal-and-continue
- Several exercises explore these problems.

Monitors *vs.* message passing

- There is a *duality* between monitors and message passing servers.
- On shared memory machines, monitors are usually used.
  - Operating systems use monitors.
- On distributed systems, message passing is used.
- RPC and rendezvous (Chapter 8) strengthen the duality between monitors and message passing.

Monitor-Based Programs	*Message-Based Programs*
permanent variables	local server variables
procedure identifiers	**`request`** channel and operation kinds
procedure call	**`send request(); receive reply`**
monitor entry	**`receive request()`**
procedure return	**`send reply()`**
`wait` statement	save pending request
`signal` statement	retrieve and process pending request
procedure bodies	arms of case statement on operation kind

Table 7.1 Duality between monitors and message passing.

- Shared memory favors Monitors
  operating systems
- Distributed memory favors Message passing
  networks, clusters

Skipping §7.3.2 and §7.3.3

- More complex examples of active monitors.

Interacting peers: each process needs maximum of all numbers

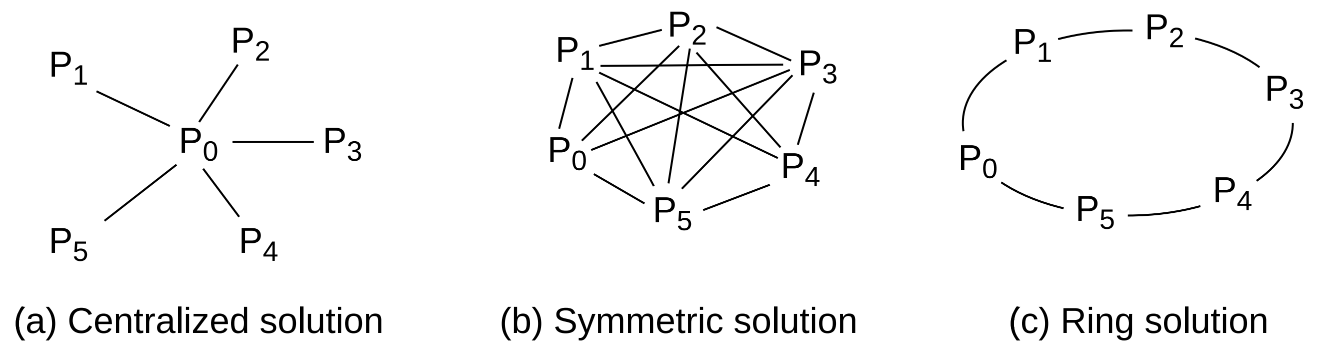


Figure 7.14 Communication structures of the three programs.

- Centralized solution: $2(n-1)$ messages
  with broadcast: n messages
  not symmetric, one process does most of the work
- Symmetric solution: $n(n-1)$ messages
  with broadcast: n messages
  SPMD: single program multiple data
- Ring solution: $2(n-1)$ messages
  almost symmetric, one process starts and ends
  long delay, but efficient if other work needs to be done

```
chan values(int), results[n](int smallest, int largest);
process P[0] {   # coordinator process
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # gather values and save the smallest and largest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
  # send the results to the other processes
  for [i = 1 to n-1]
    send results[i](smallest, largest)
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  send values(v);
  receive results[i](smallest, largest);
}
```

Figure 7.11 Exchanging values: centralized solution.

```
chan values[n](int);
process P[i = 0 to n-1] {
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # send my value to the other processes
  for [j = 0 to n-1 st j != i]
    send values[j](v);
  # gather values and save the smallest and largest
  for [j = 1 to n-1] {
    receive values[i](new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
}
```

Figure 7.12 Exchanging values: symmetric solution.

- Easiest to program.
- Creates largest number of messages.
- Message traffic frequently cancels out gains from parallelism.

```
chan values[n](int smallest, int largest);
process P[0] {  # initiates the exchanges
  int v;   # assume v has been initialized
  int smallest = v, largest = v;  # initial state
  # send v to next process, P[1]
  send values[1](smallest, largest);
  # get global smallest and largest from P[n-1] and
  #   pass them on to P[1]
  receive values[0](smallest, largest);
  send values[1](smallest, largest);
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  # receive smallest and largest so far, then update
  #   them by comparing their values to v
  receive values[i](smallest, largest)
  if (v < smallest)
      smallest = v;
  if (v > largest)
      largest = v;
  # send the result to the next processes, then wait
  # to get the global result
  send values[(i+1) mod n](smallest, largest);
  receive values[i](smallest, largest);
}
```

- Solution inherently linear, but simple
- Very low communicaiton overhead, useful if processes have other work to do.

Figure 7.13 Exchanging values using a circular ring.

Synchronous message passing

- `synch_send` blocks until message received.
- Buffer space bounded.
  - Can even be zero.
- Concurrency reduced.
- More prone to deadlock.
  - Client/server and producer/consumer OK.
  - Interacting peers very difficult.

```
channel values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    do some computation;
    synch_send values(data[i]);
  }
}

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    do some computation;
  }
}
```

Producer/consumer example using synchronous message passing.

- A delay at *every* exchange.
- With asynchronous messages there might be *no* delay.
- Can insert a buffer process in the middle.

Deadlocks with synchronized messages

- No problem with producer/consumer, pipelines.
- No problem with client/server.
- Problems with interacting peers.

```
channel in1(int), in2(int);

process P1 {
  int value1 = 1, value2;
  synch_send in2(value1);
  receive in1(value2);
}

process P2 {
  int value1, value2 = 2;
  synch_send in1(value2);
  receive in2(value1);
}
```

Exchanging values with synchronous message passing.

- Deadlocks
- Which of the three exchange-values solutions work with `asynch_send`?

- No problem with asynchronous send.
- Asynchronous send scales to more than two processes.

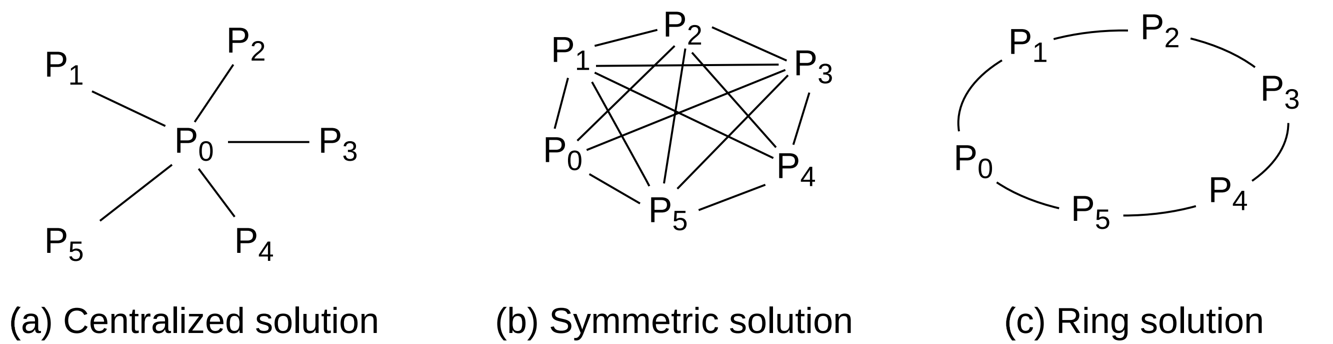


Figure 7.14 Communication structures of the three programs.

- Asynchronous send works in all 3 of the patterns but one. Which?

Tradeoff between synchronous and asynchronous

- Asynchronous uses more memory.
- Asynchronous enables more concurrency.
- Synchronous more prone to deadlock.
- Memory is cheap, time and bugs are expensive:
  - asynchronous wins

Skipping §7.6 and §7.7

MPI

- SPMD style
- Also covered in Pacheco book available here:
  `http://ezproxy.library.wwu.edu/login?url=http://proquest.safaribooksonline.com/?uicode=wwu`
- Compiling:
  `mpicc -g Wall -o myprogram myprogram.c`
- Running:
  `mpiexec -n 2 ./myprogram`

```
#include <mpi.h>

main(int argc, char *argv[]) {
  int myid, otherid, size;
  int length = 1, tag = 1;
  int myvalue, othervalue;
  MPI_Status status;

  /* initialize MPI and get own id (rank) */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if (myid == 0) {
    otherid = 1; myvalue = 14;
  } else {
    otherid = 0; myvalue = 25;
  }
  MPI_Send(&myvalue, length, MPI_INT, otherid,
           tag, MPI_COMM_WORLD);
  MPI_Recv(&othervalue, length, MPI_INT, MPI_ANY_SOURCE,
           tag, MPI_COMM_WORLD, &status);
  printf("process %d received a %d\n", myid, othervalue);

  MPI_Finalize();
}
```

Figure 7.17 MPI program to exchange values between two processes.

Other MPI primitives

- `MPI_Barrier`
  returns when all processes have called it.
- `MPI_Bcast`
  send a copy to all processes (including sender).
- `MPI_Scatter`
  scatter an array `a` of size `size` elements by sending a message containing `a[i]` to every process `i` in the group.
- `MPI_Gather`
  gather messages from every process and store them in an array of `size` elements.
- `MPI_Reduce`
  gather values from every process and reduce them to one value. Reduction operators include `MPI_SUM`, `MPI_MAX`, *etc.*
- `MPI_Allreduce`
  same as `MPI_Reduce` except that all processes receive a copy.

```
chan values(int), results[n](int smallest, int largest);

process P[0] {   # coordinator process
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # gather values and save the smallest and largest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
  # send the results to the other processes
  for [i = 1 to n-1]
    send results[i](smallest, largest)
}

process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  send values(v);
  receive results[i](smallest, largest);
}
```

- Could use MPI_Gather and MPI_Bcast

Figure 7.11 Exchanging values: centralized solution.

```
chan values[n](int);
process P[i = 0 to n-1] {
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # send my value to the other processes
  for [j = 0 to n-1 st j != i]
    send values[j](v);
  # gather values and save the smallest and largest
  for [j = 1 to n-1] {
    receive values[i](new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
}
```

Figure 7.12 Exchanging values: symmetric solution.

- Could use two calls to `MPI_Allreduce`

Networks and Sockets

- TCP (Transmission Control Protocol)
  - Same semantics as a channel.
- UDP (Unreliable Datagram Protocol)
  - Used where speed more important than reliability.
  - Games, for example.
- Built on top of IP
- FTP and HTTP built on top of TCP
- Available in most languages, Java illustrated in text.
- Covered in detail in the network class.

```
// Read a file and send it back to a client
import java.io.*; import java.net.*;

public class FileReaderServer {
 public static void main(String args[]) {
  try {
   // create server socket and
   // listen for connection on port 9999
   ServerSocket listen = new ServerSocket(9999);

   while (true) {
     System.out.println("waiting for connection");
     Socket socket = listen.accept();  // wait for client
     // create input and output streams to talk to client
     BufferedReader from_client =
       new BufferedReader(new InputStreamReader
         (socket.getInputStream()));
     PrintWriter to_client = new PrintWriter
         (socket.getOutputStream());

     // get filename from client and check if it exists
     String filename = from_client.readLine();
     File inputFile = new File(filename);
     if (!inputFile.exists()) {
       to_client.println("cannot open " + filename);
       to_client.close(); from_client.close();
       socket.close();
       continue;
     }

     // read lines from filename and send to the client
     System.out.println("reading from file " + filename);
     BufferedReader input =
       new BufferedReader(new FileReader(inputFile));
     String line;
     while ((line = input.readLine()) != null)
       to_client.println(line);
     to_client.close(); from_client.close();
     socket.close();
  }}
  catch (Exception e) // report any exceptions
    { System.err.println(e); }
}}
```

Figure 7.18 A file reader server in Java.

```
// Get file from RemoteFileServer and print on stdout
import java.io.*; import java.net.*;

public class Client {
 public static void main(String[] args) {
  try {
   // read command-line arguments
   if (args.length != 2) {
     System.out.println("need 2 arguments");
     System.exit(1);
   }
   String host = args[0];
   String filename = args[1];

   // open socket, then input and output streams to it
   Socket socket = new Socket(host,9999);
   BufferedReader from_server =
     new BufferedReader(new InputStreamReader
       (socket.getInputStream()));
   PrintWriter to_server = new PrintWriter
       (socket.getOutputStream());

   // send filename to server, then read and print lines
   // until the server closes the connection
   to_server.println(filename); to_server.flush();
   String line;
   while ((line = from_server.readLine()) != null) {
     System.out.println(line);
   }
  }
  catch (Exception e)    // report any exceptions
    { System.err.println(e); }
}}
```

Figure 7.19 A file reader client in Java.