11 Communication with Messages

Up until now, discussion has been about communication using shared data. *Messages* provide for communication without shared data. One process or the other owns the data, never two at the same time.

Message = a piece of information that is passed from one process to another.
Mailbox = a place where messages are stored between the time they are sent and the time they are received.
Operations:

- Send: copy a message into mailbox. If the mailbox is full, wait until there is enough space in the mailbox.
- Receive: copy message out of mailbox, delete from mailbox. If the mailbox is empty, then wait until a message arrives.

Is there really no shared data?

There are two general styles of message communication:

- 1-way: messages flow in a single direction (Unix pipes, or producer/consumer, or stream).
- 2-way: messages flow back-and-forth (remote procedure call, or client/server, or request/response).

Producer & consumer example:

```
Producer:
int msg1[1000];
while (1)
{
    – prepare msg1 –
    send(msg1, mbox);
}
```

```
Consumer:
int msg2[1000];
while (1)
{
    receive(msg2, mbox);
    – process msg2 –
}
```

In this implementation the buffer recycling is implicit, whereas it was explicit in the semaphore implementation.

Client & Server example:

```
Client:
char response[1000];
send("read rutabaga", mbox1);
receive(response, mbox2);

Server:
char command[100];
char answer[1000];
receive(command, mbox1);
– decode command –
– read file into answer –
send(answer, mbox2);
```

Note that this looks a lot like a procedure call&return. Analogs between procedure calls and message operations:

- Parameters: read rutabaga
- Result: buffer2
- Name of procedure: mbox1
- Return address: mbox2. Note that this is hardwired in this example. How could we make it variable?

Why use messages?

- Many kinds of applications fit into the model of processing a sequential flow of information, including all of the Unix filters.
- The communicating parties can be totally separate, except for the mailbox:
  - Less error-prone, because no invisible side effects: no process has access to another's memory.
  - They might not trust each other (OS vs. user).
  - They might have been written at different times by different programmers who knew nothing about each other.
  - They might be running on different processors on a network, so procedure calls are out of the question;
  - Or on a machine with non-uniform memory access, where it is expensive to access another processor's memory.

Message systems vary along several dimensions:

- Relationship between mailboxes and processes:
  - One mailbox per process, use process name in send, no name in receive (simple but restrictive) [RC4000, V].
  - No strict mailbox-process association, use mailbox name (can have multiple mailboxes per process, can pass mailboxes from process to process, but trickier to implement) [Unix].
- Extent of buffering:
  - Buffering (more efficient for large transfers when sender and receiver run at varying speeds).
  - None – rendezvous protocols (simple, OK for call-return type communication, know that message was received).
- Blocking vs. non-blocking ops:
  - Blocking receive: return message if mailbox isn't empty, otherwise wait until message arrives.

  - Non-blocking receive: return message if mailbox isn't empty, otherwise return special "empty" value.
  - Blocking send: wait until mailbox has space.
  - Non-blocking send: return "full" if no space in mailbox.

  What happens with rendezvous protocols and non-blocking operations? Show how either sender, receiver, or data must wait.

- Additional forms of waiting:
  - Almost all systems allow many processes to wait on the same mailbox at the same time. Messages get passed to processes in order.
  - A few systems allow each process to wait on several mailboxes at once (e.g. *select* in UNIX). The process gets the first message to arrive on any of the mailboxes. This is quite useful. Network services, window systems are examples.
- Constraints on what gets passed in messages:
  - None: just a stream of bytes (Unix pipes).
  - Enforce message boundaries (send and receive in same chunks).
  - Protected objects (e.g. process id of sender, or a token for a mailbox).

How do the following mechanisms relate to the above classifications?

- Condition variables
- Unix pipes

Messages and shared-data approaches are equally powerful, but result in very different-looking styles of programming. Most people find shared-data approach easier to work with. But, it is easier to make message-passing programs perform well on machines that don't have shared memory.