Topic 7: Message Communication

- Readings for this topic: Section 3.4-3.6
- Communication using shared data vs without shared data.

- Components of message communication:
  - Message = a piece of information that is passed from one process to another.
  - Mailbox (Port) = a place where messages are stored until they are received.

- Operations:
  - Send: copy a message into mailbox. What if the mailbox is full?
  - Receive: copy message out of mailbox, delete from mailbox. What if empty?

- Is there really no sharing?

- Two general styles of message communication:
  - 1-way: messages flow in a single direction (Unix pipes, or producer/consumer style).
  - 2-way: messages flow back-and-forth (remote procedure call, or client/server style).

- Producer & consumer example (1-way):

  ```c
  Producer:

  int msg1[1000]; // area to prepare the stuff to send
  while (true)
  {
    -- prepare msg1 --
    send(msg1, mbox);
  }
  ```
Consumer:

```c
int msg2[1000]; // place to receive a message
while (true)
{
    receive(msg2, mbox);
    -- process msg2 --
}
```

- Client & Server example (2-way):

  **Client:**
  ```c
  char response[1000];
  send("read my®le", mbox1);
  receive(response, mbox2);
  ```

  **Server:**
  ```c
  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. Local Procedure Call (LPC) facility in Windows XP (Ch.22; PP 804-805). Analogs between procedure calls and message operations:

  - Parameters: request message (read my®le)
  - Result: return message (contents of my®le)
  - Name of procedure: mbox1
  - Return address: mbox2.

- Any problem with it? Why?
• Why use messages?
  • Many applications fit into the model of processing a sequential flow of information.
  • The communicating parties can be totally separate, except for the mailbox:
    • Less error-prone, because no invisible side effects.
    • They might not trust each other (OS vs. user).
    • They might have been written at different times by different programmers.
    • They might be running on different processors on a network, so ...

• Different styles of message passing systems: they vary along several dimensions

  • Relationship between mailboxes and processes:
    • One mailbox per process, use process name in send, no name in receive.
    • No strict mailbox-process association, use mailbox name.

  • Extent of buffering:
    • Buffering (efficient transfers when sender and receiver run at different rates).
    • None -- rendezvous protocols (simple; OK for call-return type communication).
    • Desirable features of rendezvous protocols?

  • Waiting (blocking vs. non-blocking ops):
    • Blocking receive: return message; if empty, wait until message arrives.
    • Non-blocking receive: return message; if empty, return special ‘‘empty’’ value.
    • Blocking send: wait until mailbox has space.
    • Non-blocking send: return ‘‘full’’ if no space in mailbox.

  • Additional forms of waiting:
    • Many processes wait on the same mailbox at the same time.
    • One process wait on several mailboxes at once (e.g. select in UNIX).
• Constraints on messages:
  • None: just a stream of bytes (Unix pipes).
  • Enforce message boundaries (send and receive in same chunks).
  • More constraints (e.g. process id of sender): security issues

• How do the following mechanisms relate to the above classifications?
  • Semaphores
  • Condition variables

• Are messages and shared-data approaches equally powerful?
• Do they result in very different-looking styles of programming?
• Which one is easier to work with for most people?