Message Passing

OS Lecture 10

UdS/TUKL WS 2015

Communicating with Messages

Explicit communication: *sending* and *receiving* of messages via *mailboxes*.

```
System object:
mailbox_t mbox; // UNIX-like: typically a file descriptor
```

Producer/sender process:

```
char local_buf[1000];
prepare_message(local_buf);
send(local_buf, mbox);
```

Consumer/recipient process:

```
char local_buf[1000];

receive(local_buf, mbox);
processs_message(local_buf);
```

Why use explicit messages instead of shared memory?

- No side effects / no sharing → less error-prone
- **No trust** required: can *validate messages* before processing
- clear separation of *interface* and *implementation*
- enables/simplifies integration of *independently developed* components
- *distribution* can be transparent: receiver could be running on a different computer (→ *scaling out*)
- can interpose *proxies* for various reasons (logging, validating, filtering, load balancing, etc.)
- on machines with *non-uniform memory* (NUMA), sending messages can be more efficient

Communication Styles

There are two principle messaging patterns.

1. **One-way** communication: producer-consumer pattern
   - messages flow in one direction (like a `pipe`)
2. **Two-way** communication: like a conversation
   - messages flow back and forth
   - *peer-to-peer* or *client-server*

Example: Client & Server Communication

Pattern: *client* asks server to carry out a *named operation*, *server* replies with *result* (or *error*).

Example: Client & Server Communication

Pattern: *client* asks server to carry out a *named operation*, *server* replies with *result* (or *error*).

System objects: mailbox_t mbox1, mbox2;

Client process:

```
string_t response;
send("read /path/to/file", mbox1);
receive(response, mbox2);
```

Server process:

```
string_t command, answer;
receive(command, mbox1);
// ... decode command ...
// ... generate answer ...
send(answer, mbox2);
```

Client-Server Pattern vs. Procedure Calls

How is a *client-server* invocation/response pair similar to / different from *regular procedure calls*?

» **Similarities**: well-defined **parameters**; well-defined **result type**; referenced by **name**; defined **return address** (i.e., where to continue execution of client)

» **Differences**: cross-language procedure calls are difficult; procedure calls don't fail — either call returns or entire process crashes (all or nothing)

Remote Procedure Calls

Because client-server communication is so similar to procedure calls, the invocation of operations on a server is often abstracted as *remote procedure calls* (RPC).

- hides messaging: looks like a regular procedure call in the client program (but handling failure cases can be tricky)
- A library/framework/middleware/code generator transparently takes care of *marshalling* and *unmarshalling* procedure parameters (→ serializing into / deserializing from a *language-independent* message format)
- Examples: Google Protocol Buffers, Apache Thrift, CORBA, Java RMI, XML-RPC, SOAP, JSON-RPC, …

Message System Design Choices

While sending and receiving messages is conceptually simple, a large variety of semantics can be found in practice.

- What is being addressed?
- Buffering: what to do with messages that nobody is currently waiting to receive?
- What to do when an operation cannot be immediately carried out? (Example: buffer full)
- What do messages look like? Fixed size? Explicit message boundaries? Human readable?

Mailboxes vs. Processes

Do you send messages to abstract *mailboxes* or directly to *processes*?

1. One mailbox per process: send to *process name*: simple, but restrictive
   - e.g., UNIX signals like `SIGTERM`
2. Mailboxes are first-class entities: send to *mailbox name*
   - e.g., UNIX sockets representing ports like `localhost:8080`
   - can have multiple mailboxes per process
   - can share mailboxes between processes
   - can pass mailboxes to other processes

Buffering

What happens to logically sent, but not yet received messages?

1. *Dynamically sized* buffers: OS allocates as much memory as needed (or until it runs out)
2. *Fixed-size* buffers: up to a pre-determined limit, messages are copied & stored. What if no more space? Drop oldest? Reject latest?
3. *Single-message* buffers with *register semantics*: always keep the latest message (exactly one).
4. *No* buffering: message delivered only when receiving process is present (*rendezvous* communication).

Blocking vs. Non-Blocking Operations

What to do if intended operation cannot be *immediately* carried out?

» **Blocking receive**: return message if available, otherwise wait until message arrives.

» **Non-blocking receive**: return error / "buffer empty" if no message is available.

» **Blocking send**: copy message into mailbox; if necessary wait until space is available.

» **Non-blocking send**: return "full" (if buffered) or "recipient unavailable" (if rendezvous protocol)

Non-Blocking Rendezvous Protocols?

What happens if you combine *rendezvous* communication with *non-blocking* send and receive operations?

- rendezvous communication = message not buffered
- non-blocking send = sender doesn't wait for recipient to show up
- non-blocking receive = recipient doesn't wait for sender to show up
- Most likely outcome: *no communication at all*.
  → **Buffering** required, or one party has to **wait**.

Waiting for Messages

One mailbox, *one* waiting process? *One* mailbox, *many* waiting processes? *Many* mailboxes, *one* waiting process?

- Typically, *many* processes can wait on *the same* mailbox.
  - How are message distributed among recipients? FIFO? By chance?
- Modern systems also allow processes to *wait on several mailboxes at once*.
  - e.g., UNIX `select()` operation
  - logically returns first message to arrive in any of the mailboxes
  - useful for network services, windowing systems, etc.

Message Structure

Does the system enforce any particular message structure?

- unstructured *streams*: e.g., UNIX pipes, TCP, …
- explicit *message boundaries*, variable size: UDP
- fixed-size messages: e.g., *ring buffers*
- references to *protected objects* (e.g., access tokens)

Delivery Guarantees

Are messages always guaranteed to be delivered?

- no guarantees, *best effort*: e.g., UDP
- guaranteed delivery, but *out of order* possible: e.g., SCTP
- guaranteed, *in-order* delivery: e.g., TCP
- guaranteed, *all-or-nothing*: distributed transactions (→ distributed systems course)

Message Passing vs. Shared Memory

Which one would you rather use?

- fundamentally of equivalent power
  - can implement DSM over message-passing API
  - can implement message-passing API using shared memory
- result in very different styles of programming
  - personal preferences vary…
- In practice: *scalability*, *distribution*, & *efficiency* requirements force a combination of both styles ("right tool for the job").
- Both subject to *deadlock* risks…