Chap 2. Inter-Process Communication

- IPC primitives
  - send/receive
  - send/receive can be blocking or non-blocking
  - Communication can be synchronous or asynchronous
  - Communication can also be transient or persistent
  - Sockets (Java and Unix)

2.1 IPC primitives

Message: | Header | Body |
---|---|---|
- Send (destination, &msg);
- Receive (source, &buf);
- destination and source can be process id, port (with single receiver), or mailbox (multiple receivers)
- Blocking vs. non-blocking
  - Non-blocking: returns after ‘minimal’ delay; only local operation is involved
  - Blocking receive: blocks until message available
  - Blocking send: different definitions

Communication system

Return from a send

1. Copied to kernel
2. Transmitted
3. Received by dest. kernel
4. Received by dest. process

- User buffer can be reused;
- may not exist
- Acknowledgement from destination kernel
- Remote process called receive

Blocking send if or later (external operation involved)
Non-blocking send if or earlier (returns after local operation)

Comments

- In non-blocking send/receive, interrupt can be used to inform calling process when operation is complete (e.g., Unix SIGIO)
- Non-blocking receive can simulate blocking receive (busy wait), but not vice versa (unless extra thread is used)
- Non-blocking receive is not very useful (you cannot proceed without message)

Comparison

- Blocking
  - Advantages: Ease of use and low overhead of implementation
  - Disadvantage: low concurrency
- Non-blocking
  - Advantages: Flexibility, parallel operations
  - Disadvantages: Programming tricky. Program is timing-dependent (interrupt can occur at arbitrary time, and execution irreproducible)

Use blocking versions with multiple threads
Using blocking operation without penalty

- Some threads may be blocked while others continue to be active

Practice

- Many OSs support both blocking and non-blocking versions of send/receive
- With blocking version, timeout option is often available
- Blocking send may also block on full buffer (no more space in send buffer)

Implementation considerations

- Time-out is especially important for inter-machine communication, due to possible failure of communication or remote machine
- Copying to local kernel takes time, but facilitates buffer reuse by sender
- If destination is on same machine, send-by-reference, is most efficient if memory can be shared, but access must be controlled after send
  - Copy-on-write

Communication

- Synchronous communication if send blocks until some response returns from destination (i.e., at least until step (c))
- Asynchronous: not synchronous
- Overloaded terms!
- Transient
  - Both sender and receiver must be up and running
- Persistent
  - Sender and receiver need not be running at same time

Transient Communication

- Asynchronous communication
- Receipt-based synchronous communication

- Delivery-based synchronous communication
- Response-based synchronous communication
**Persistent Communication**

(a) Persistent asynchronous communication
(b) Persistent synchronous communication

**Shared memory**
- `shmid = shmat ( size, … )`
- `addr = shmat ( shmid, shmatr, … )` (attach)
  - `shmat` is suggested address (or 0)
- `shmdt ( shmid )` (detach)

**Copy on write**
- Page table of sender
- Page table of receiver
- Copy-on-write
  - read-only (hardware level) → write-protection fault on write attempt

**Implementation of message system (intra-machine messages)**
- Message Q:
  - Head → Tail
  - `Next msg`
  - Sender pid
  - Length
  - `Msg body`
- Message block:
  - `Msg body`

**Send (destination, &msg)**
- If `destination` not known
  - (return “receiver unknown”)
- Else call `memoryallocate` to get system space for `MsgBlk`;
- Copy sender process id, message length, body (`msg`), into `MsgBlk`;
- Add `MsgBlk` at head of receiver’s `msg Q`;

**System V Unix example**
- Create message queue
  - `msgQid = msgget ( )`
- Send
  - `msgsnd ( msgQid, &buf )`
- Receive
  - `msgrcv ( msgQid, &buf )`
Unix pipes
- Pipe
  - Between two related processes
  - FIFO of bounded length (normally 4KB)
  - No message boundaries
- Named pipe
  - Almost like file: name and permissions (but created by mknnod)
  - Can be accessed by unrelated processes
  - Persistent

Mach example
- Mach IPC plays central role in Mach OS.
- Mach port
  - msgs are sent to/received from ports. Only threads in owner task that owns port can receive them.
  - Data structure: owner task, pointer to msg Q, pointer to Q of threads blocked on it.
- Capability list
  - List of port numbers with associated rights.

Kernel ports
- On creation each task and threads are given some kernel ports, e.g.,
  - thread_self: thread can create new port by sending msg to this port
  - task_notify: to receive msg from kernel.
- Bootstrap: provides access to Name Server, through which task can obtain send rights to ports of publicly available servers.

Mach Net Message Server (NMS)

NMS details
- Send rights
  - Network port
  - Send rights
- Receiver task
  - Receive msg from NMS
  - Network address

Text Fig. 18.6