Inter Process Communication

Revision of process related concepts

- PCB, struct proc
- Process lifecycle – different states
- Queues/Lists of processes
- What is “Blocking”
- Event driven kernel

Before IPC, let's learn more about file related system calls

- Redirection

  ls > */tmp*/file

  cat < /etc/passwd

  how does this work?

- File descriptors are inherited across fork

IPC: Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
    - Information sharing, e.g. copy paste
    - Computation speedup, e.g. matrix multiplication
    - Modularity, e.g. chrome – separate process for display, separate for fetching data
    - Convenience ,
- Cooperating processes need interprocess communication (IPC)
- Broadly Two models of IPC
    - Shared memory (examples: shared memory, pipes, ...)
    - Message passing (examples: send/recv on socket, send-recv messages, ...)

Shared Memory Vs Message Passing

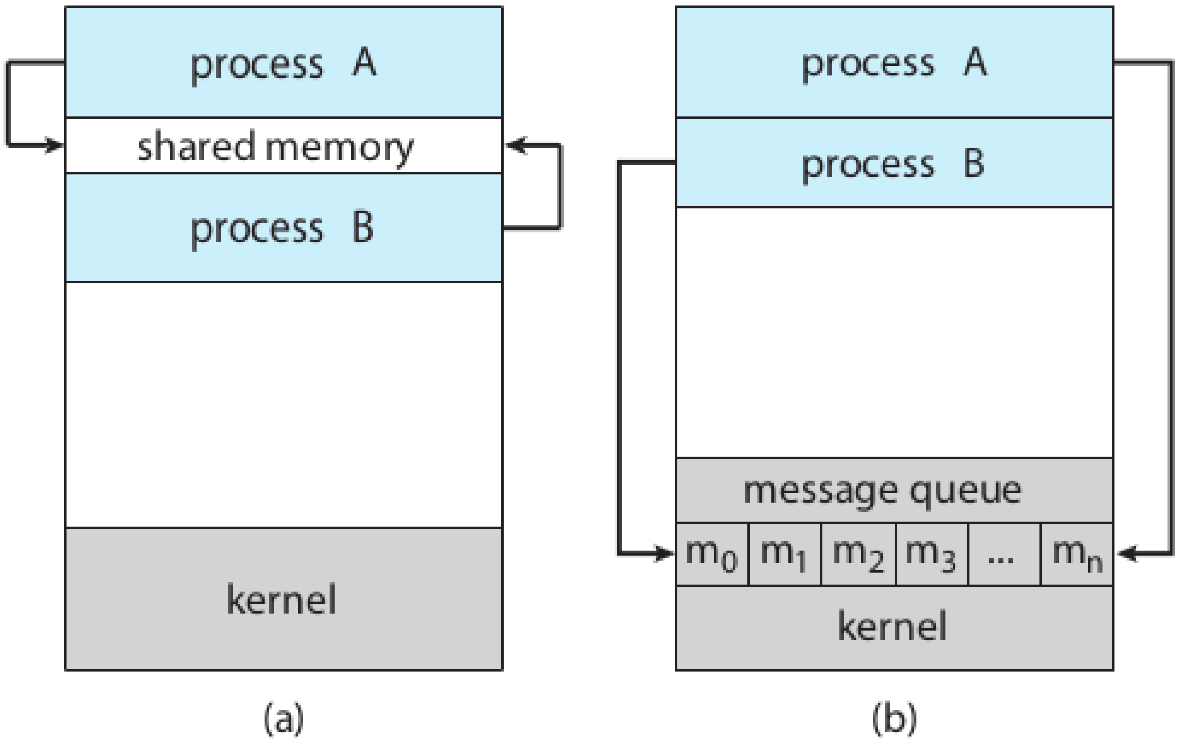


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

- Each requires OS to provide system calls for
  - Creating the IPC mechanism
  - To read/write using the IPC mechanism
  - Delete the IPC mechanism
- Note: processes communicating with each other with the help of OS!

Typical code of shared memory type of solutions

Process P1

```
x = getshm(ID)
*x = 100;
```

Process P2

```
x = getshm(ID)
y = *x
```

Typical code of message passing type solutions

Process P1

```
send(P2, x)
or
broadcast(x)
```

Process P2

```
y = receive(P1)
or
y = receive(anyone)
```

Example of co-operating processes: Producer Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - unbounded-buffer places no practical limit on the size of the buffer
  - bounded-buffer assumes that there is a fixed buffer size

Example of co-operating processes: Producer Consumer Problem

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Can only use BUFFER_SIZE-1 elements

Example of co-operating processes: Producer Consumer Problem

- **Code of Producer**

```
while (true) {
      /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE count)  ==
   out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

Example of co-operating processes: Producer Consumer Problem

- **Code of Consumer**

```
while (true) {
        while (in == out)
                ; // do nothing -- nothing to consume

      // remove an item from the buffer
      item = buffer[out];
      out = (out + 1) % BUFFER SIZE;
      return item;
}
```

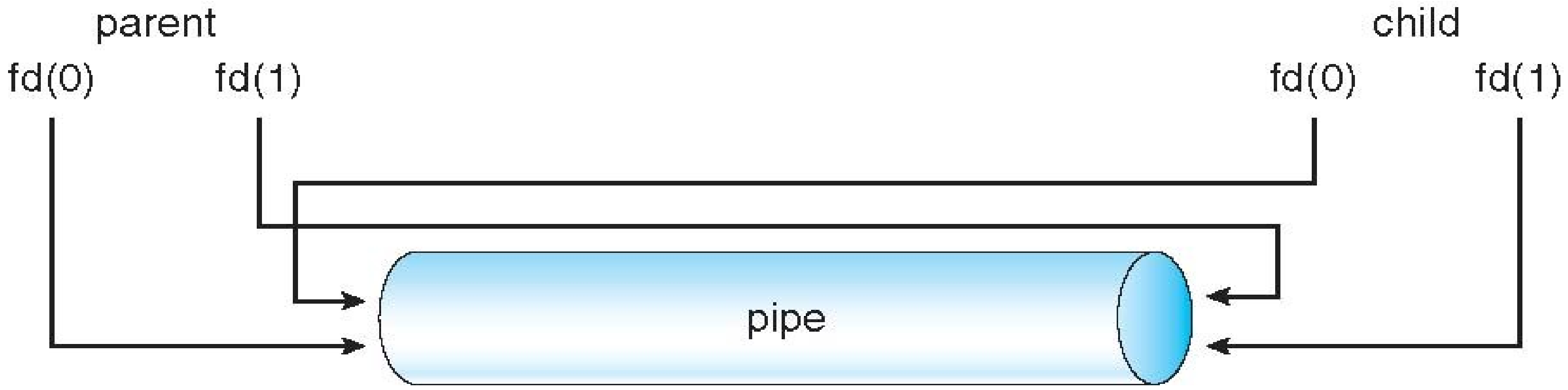
Pipes

Pipes for IPC

- **Two types**
  - **Unnamed Pipes or ordinary pipes**
  - **Named Pipe**

Ordinary pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Requires a parent-child (or sibling, etc) kind of relationship between communicating processes


parent
fd(0)
fd(1)
child
fd(0)
fd(1)
pipe

Named pipes

- Also called FIFO
- Processes can create a “file” that acts as pipe. Multiple processes can share the file to read/write as a FIFO
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems
- Is not deleted automatically by OS

Named pipes

- int mkfifo(const char *pathname, mode_t mode);
- Example

Shared Memory

System V shared memory

- Process first creates shared memory segment

```
segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

- When done, a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

Example of Shared memory
POSIX Shared Memory

- **What is POSIX?**
  - **Portable Operating System Interface (POSIX)**
  - **family of standards**
  - **specified by the IEEE Computer Society**
  - **for maintaining compatibility between operating systems.**
  - **API (system calls), shells, utility commands for compatibility among UNIXes and variants**

POSIX Shared Memory

- shm_open
- ftruncate
- Mmap
- See the example in Textbook

Message passing

Message Passing

- Message system – processes communicate with each other using send(), receive() like syscalls given by OS
- IPC facility provides two operations:
  - send(message) – message size fixed or variable
  - Receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Communication link can be implemented in a variety of ways

Message Passing using "Naming"

- **Pass a message by "naming" the receiver**
  - **A) Direct communication with receiver**
    - **Receiver is identified by sender directly using it's name**
  - **B) Indirect communication with receiver**
    - **Receiver is identified by sender in-directly using it's 'location of receipt'**

Message passing using direct communication

- Processes must name each other explicitly:
  - send (P, message) – send a message to process P
  - receive(Q, message) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Message passing using IN-direct communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Message passing using IN-direct communication

- **Operations**
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- **Primitives are defined as:**
  - send(A, message) – send a message to mailbox A
  - receive(A, message) – receive a message from mailbox A

Message passing using IN-direct communication

- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Message Passing implementation: Synchronization issues

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

Producer consumer using blocking send and receive

Producer

message next produced;

while (true) {

/* produce an item in next_produced */

send(next_produced);

}

Consumer

message next_consumed;

while (true) {

receive(next_consumed);

}

Message Passing implementation: choice of Buffering

- **Queue of messages attached to the link; implemented in one of three ways**
  1. **Zero capacity – 0 messages**
     - Sender must wait for receiver (rendezvous)
  2. **Bounded capacity – finite length of n messages**
     - Sender must wait if link full
  3. **Unbounded capacity – infinite length**
     - Sender never waits