# INTER-PROCESS COMMUNICATION

Tanzir Ahmed
CSCE 313 Spring 2020

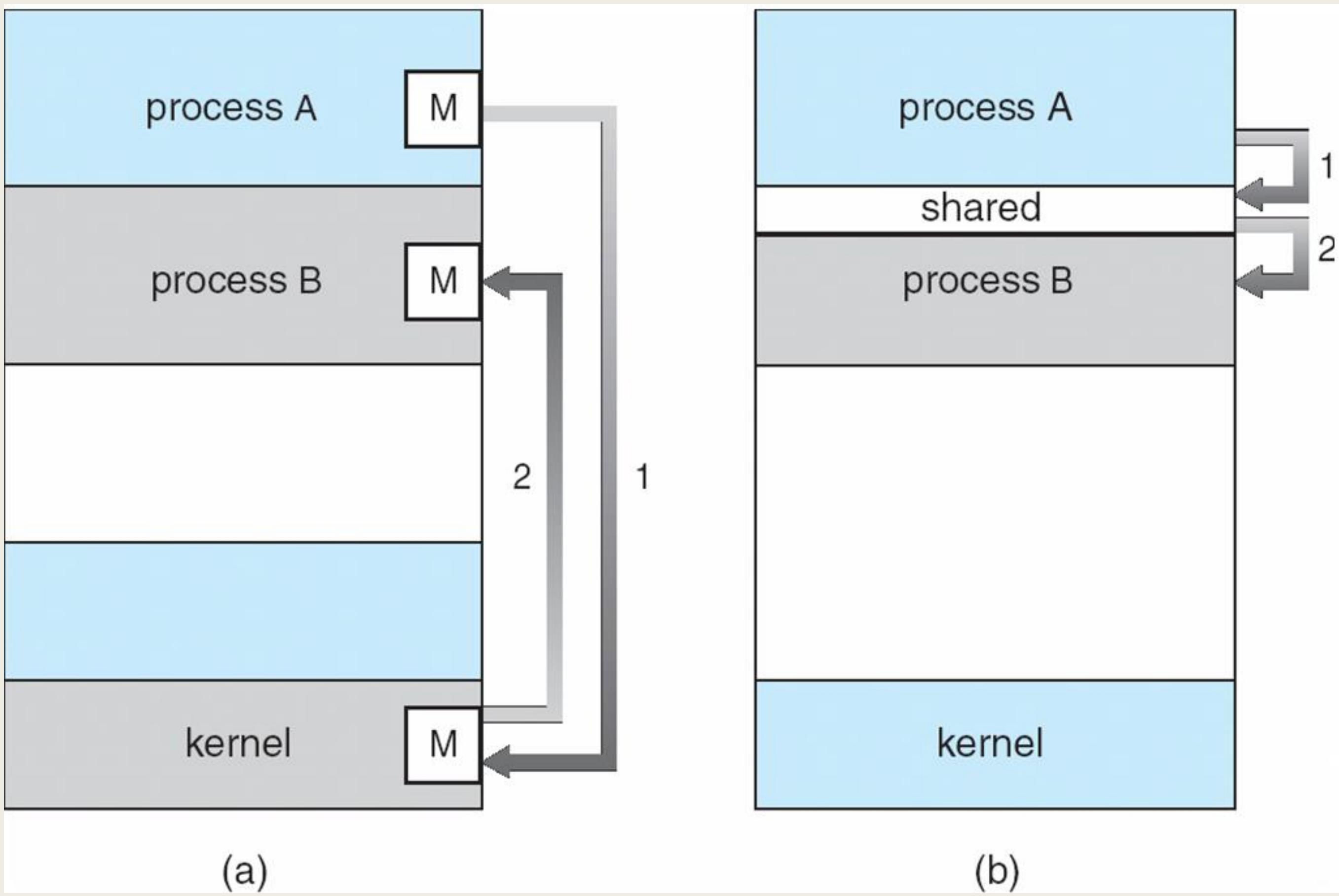
# Inter-Process Communication

- IPC Methos
  - *Pipes and FIFO*
  - *Message Passing*
  - *Shared Memory*
  - *Semaphore Sets*
  - *Signals*
- References:
  - *Beej's guide to Inter Process Communication for the code examples (https://beej.us/guide/bgipc/)*
  - *Understanding Unix/Linux Programming, Bruce Molay, Chapters 10, 15*
  - *Advanced Linux Programming Ch 5*
  - *Some material also directly taken or adapted with changes from Illinois course in System Programming (Prof. Angrave), UCSD (Prof. Snoeren), and USNA (Prof. Brown)*

# Inter-Process Communication

- OS provides generic mechanisms to communicate
  - ***Un-named and Named Pipes***
    - Explicit communication channel
    - Explicit Synchronization: read() operation is blocking
  - ***Message Passing:*** *explicit communication channel provided through send()/receive() system calls*
    - Explicit Synchronization through send() and recv()
  - ***Shared Memory:*** *multiple processes can read/write same physical portion of memory; implicit channel*
    - Implicit channel
    - No OS mediation required once memory is mapped
    - No synchronization between communicating threads
      - *There is no read/send or write/recv available*
      - *How to know when data is available?*

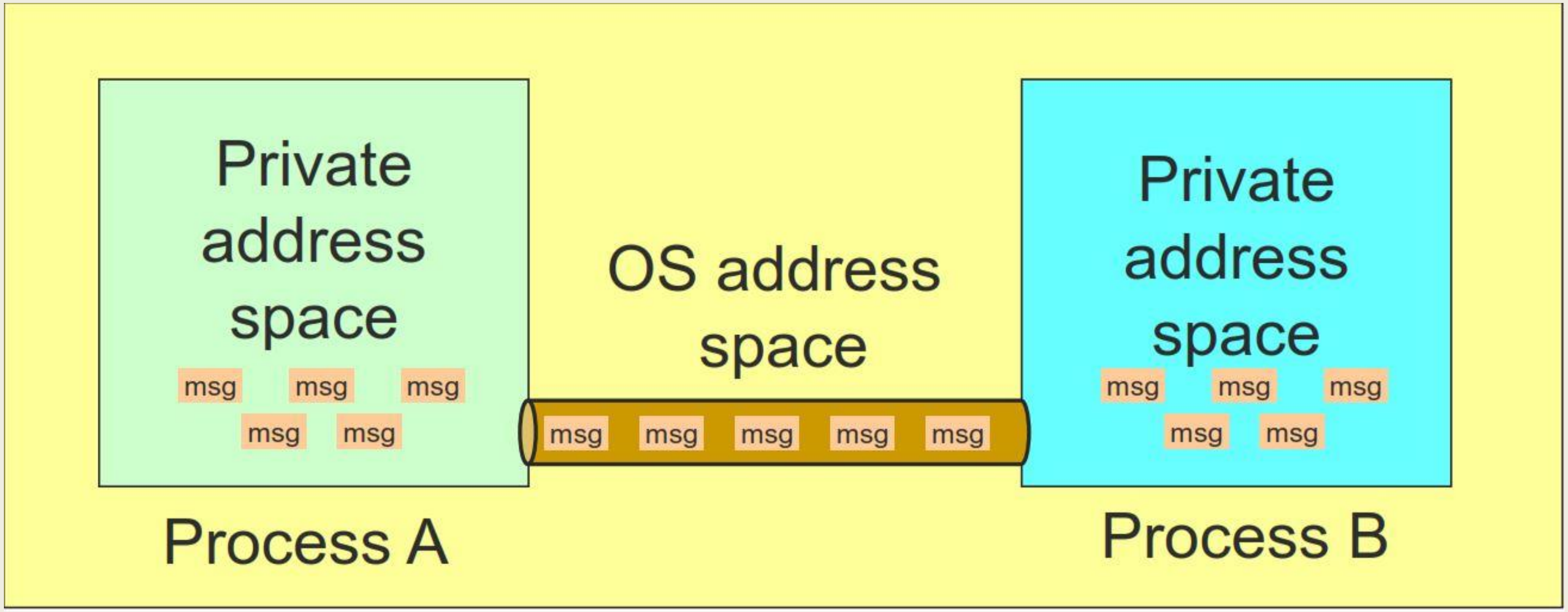
# IPC Fundamental Communication Models




**Example: pipe, fifo, message, signal**

**Example: shared memory, memory mapped file**

# Communication Over a Pipe



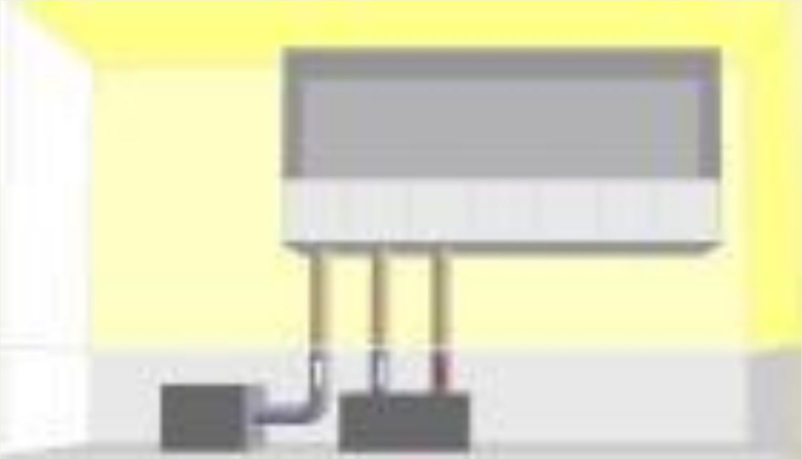
# Unix Pipes (aka Unnamed Pipes)

```
int pipe(int fildes[2])
```

- Returns a pair of file descriptors
  - ***fildes[0] is connected to the read end of the pipe***
  - ***fildes[1] is connected to the write end of the pipe***
- **Create a message pipe**
  - *Data is received in the order it was sent*
  - *OS enforces mutual exclusion: only one process at a time*
  - *Processes sharing the pipe must have same parent in common*

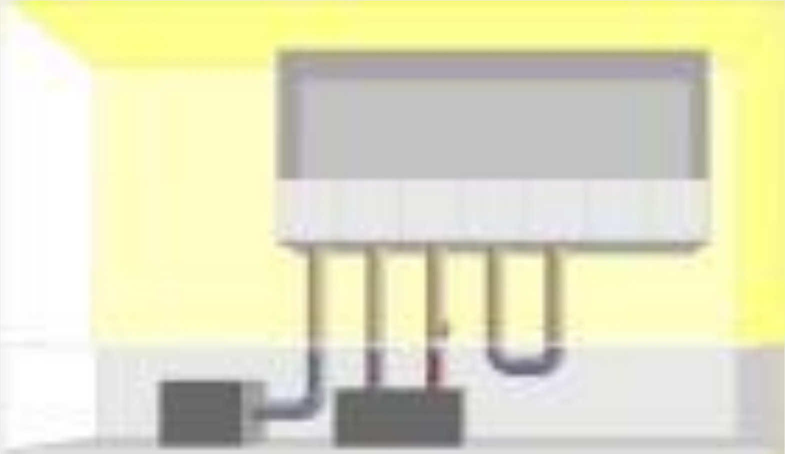
# Pipe Creation

BEFORE pipe



Process has some usual files open

AFTER pipe



Kernel creates a pipe and sets file descriptors

- BEFORE
  - *Shows standard set of file descriptors*
- AFTER
  - Shows newly created pipe in the kernel and the two connections to that pipe in the process

# IPC Pipe - Method

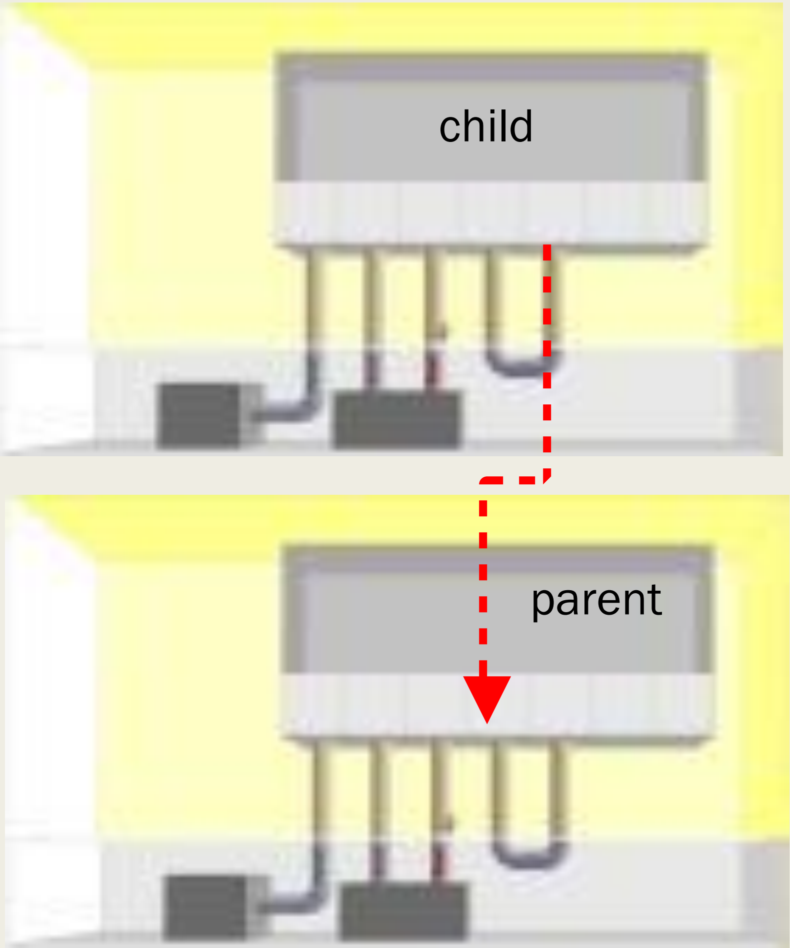
```c
#include <stdio.h>
#include <unistd.h>

void main ()
{
        char buf [10];
        int fds [2];
        pipe (fds);
        printf ("sending msg: Hi\n");
        write (fds[1], "Hi", 3);
        read (fds[0], buf, 3);
        printf ("Received msg: %s\n", buf);

}
```

Connects the two fds as pipe

```
compute-linux1 tanzir/code> ./a.out
sending msg: Hi
Received msg: Hi
```

# Unnamed Pipe Between Two Processes

```c
int main ()
{
   int fds [2];
   pipe (fds); // connect the pipe
   if (!fork()){ // on the child side
        sleep (3);
        char * msg = "a test message";
        printf ("CHILD: Sent %s\n", msg);
        write (fds [1], msg,
strlen(msg)+1);
   }else{
        char buf [100];
        read (fds [0], buf, 100);
        printf ("PRNT: Recvd %s\n", buf);
   }
   return 0;
}
```



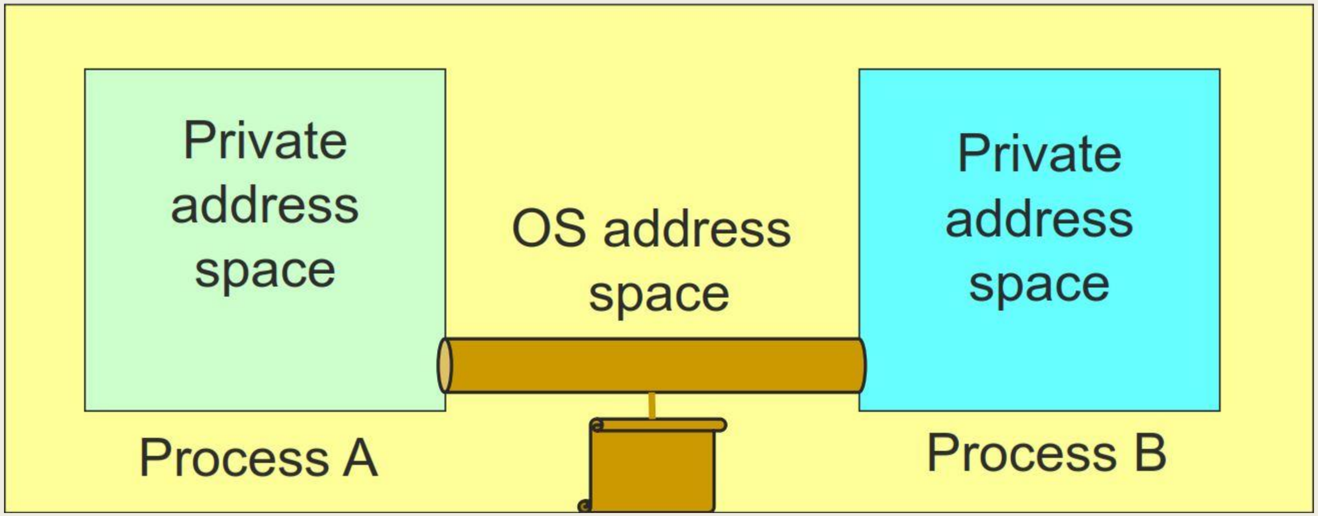
# Shell Piping: “ls -l | wc –l”

```
void main ()
{
   int fds [2];
   pipe (fds); // connect the pipe
   if (!fork()){ // on the child side
       close (fds[0]);  // closing unnecessary pipe end
       dup2 (fds[1], 1); // overwriting stdout with pipeout
       execlp ("ls", "ls", "-l", NULL);
   }else{
       close (fds[1]); // closing unnecessary pipe end
       dup2 (fds[0], 0); // overwrite stdin with pipe in
       execlp ("wc", "wc", "-l", NULL);
   }
}
```

# IPC- FIFO (named PIPE)

# FIFO

- A pipe (also called unnamed pipe) disappears when no process has it open
- FIFOs (also called named pipes) are a mechanism that allow for IPC that's like using regular files, except that the kernel takes care of synchronizing reads and writes, and
- <u>Data is never actually written to disk </u>(instead it is stored in buffers in memory) so the overhead of disk I/O (which is huge!) is avoided.



# FIFO vs PIPE

- A FIFO is like an unconnected garden hose lying on the lawn
  - *Anyone can put one end of the hose to his ear and another person can walk up to the hose and speak into the other end*
  - *Unrelated people may communicate through a hose*
  - *Hose exists even if nobody is using it*

# FIFO

- **It's part of the file system**
  - *It has a name and path just like a regular file.*
  - *Programs can open it for reading and writing, just like a regular file.*
  - *However, the file does not contain any data*
- **Works like a Bounded Buffer**
  - *Bytes travel in First-In-First-Out fashion: hence the name FIFO.*
  - *Reading process must wait for the writer*
  - *Writer must wait for the read operation once the buffer is full*

# FIFO - Problems

- We still need to agree on a name ahead of time – how to communicate that??

```
FIFORequestChannel rc ("control", ..){
  …
  mkfifo ("control", PERMS); // create
}
```

- Not concurrency safe within a process
  - *Like a file used by multiple processes/threads*
  - *Multiple threads writing can cause race condition*

# Using FIFO’s

- How do I create a FIFO
  - *mkfifo (name)*
- How do I remove a FIFO
  - *rm fifoname or unlink(fifoname)*
- How do I listen at a FIFO for a connection
  - *open (fifoname, O_RDONLY)*
- How do I open a FIFO in write mode?
  - *open(fifoname, O_WRONLY)*
- How do two processes speak through a FIFO?
  - *The sending process uses write and the listening process uses read. When the writing process closes, the reader sees end of file*

# FIFO DEMO

Writer

```
#define FIFO_NAME "test.txt"
int main(void)
{
  char s[300];
  int num, fd;
  mkfifo(FIFO_NAME, 0666); // create
  printf("Waiting for readers...\n");
  fd = open(FIFO_NAME, O_WRONLY); //open
  if (fd < 0)
    return 0;
  printf("Got a reader--type some
stuff\n");
  while (gets(s)) {
    if (!strcmp (s, "quit")) break;
    if ((num = write(fd, s, strlen(s)))
== -1)
      perror("write");
    else
      printf("SENDER: wrote %d bytes\n",
num);
  }
  //unlink (FIFO_NAME);
  return 0;
}
```

Reader

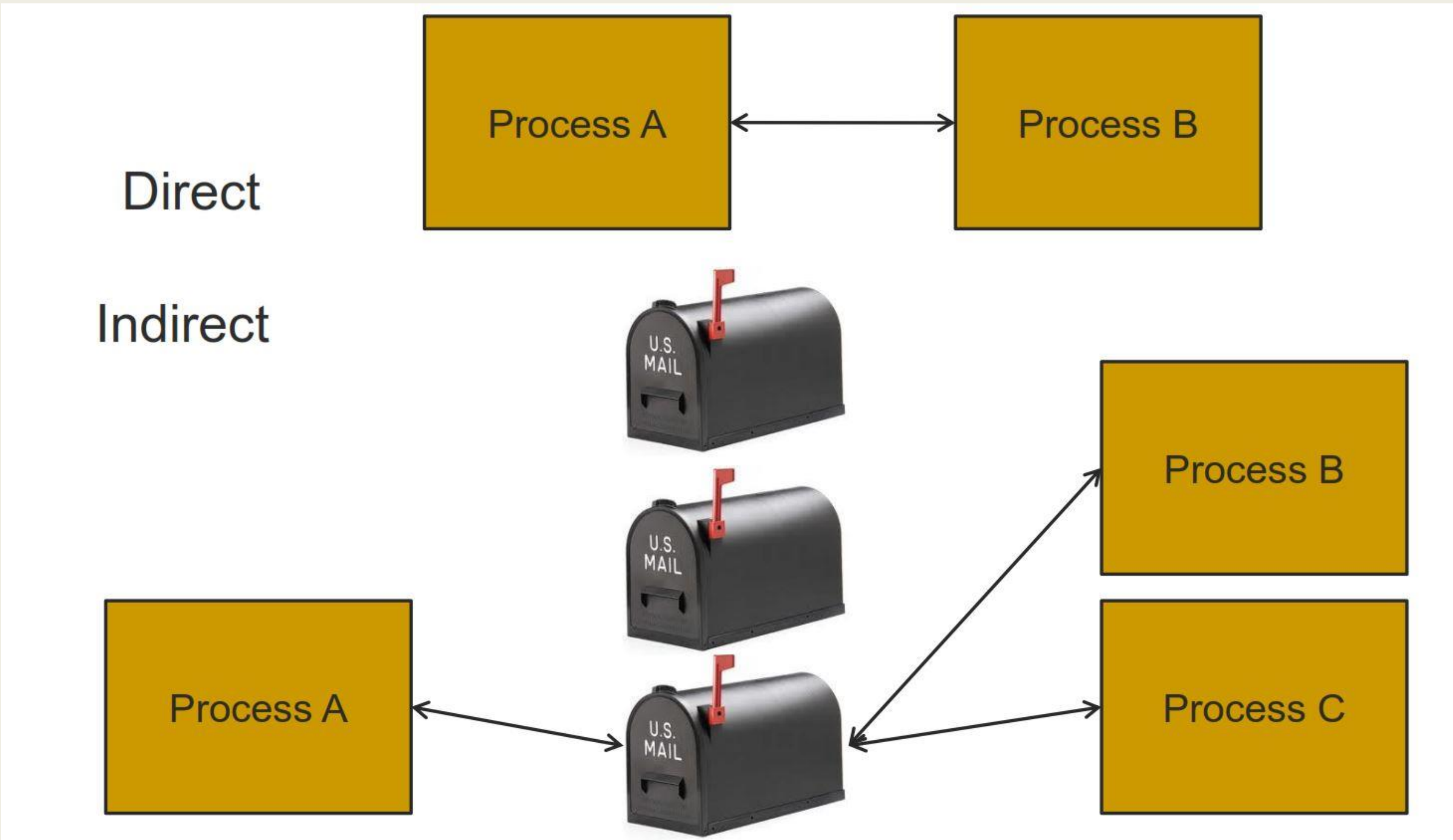
```
int main(void)
{
  char s[300];
  int num, fd;
  printf("waiting for writers...\n");
  fd = open(FIFO_NAME, O_RDONLY);
  printf("got a writer\n");
  do{
    if ((num = read(fd, s, 300)) == -1)
      perror("read");
    else {
      s[num] = '\0';
      printf("RECV: read %d bytes:
\"%s\"\n", num, s);
    }
  } while (num > 0);
  return 0;
}
```

# IPC: Message Queue

# Message Queue

- Mechanism for processes to communicate and to synchronize their actions
- IPC facility provides two operations:
  - ***send****(message)*
  - ***receive****(message)*
- If *P* and *Q* wish to communicate, they need to:
  - *establish a communication link between them*
  - *exchange messages via send/receive*

# Message Passing

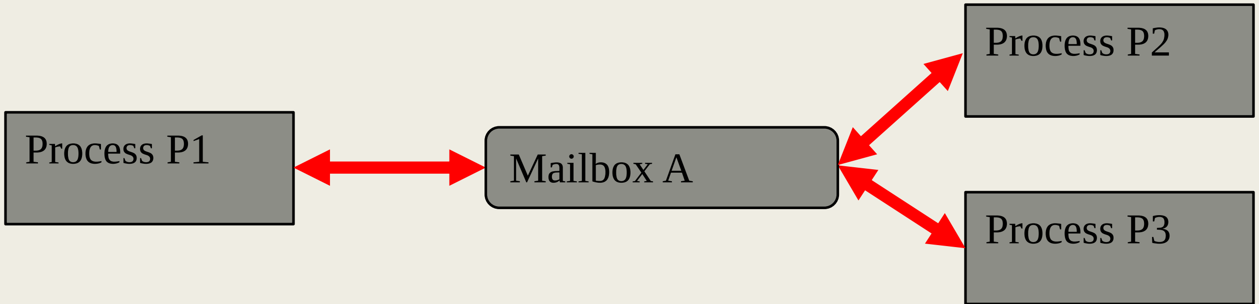


# Direct Message Passing

- 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 (or implicitly) while sending/receiving*
  - *A link is associated with exactly one pair of communicating processes*
  - *Between each pair, there exists exactly one link*
- Limitation: Must know the name or id of the other process

# Indirect Message Passing

- Messages are directed to and received from mailboxes (also referred to as ports)
  - *Mailbox can be owned by a process or by the OS*
  - *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*

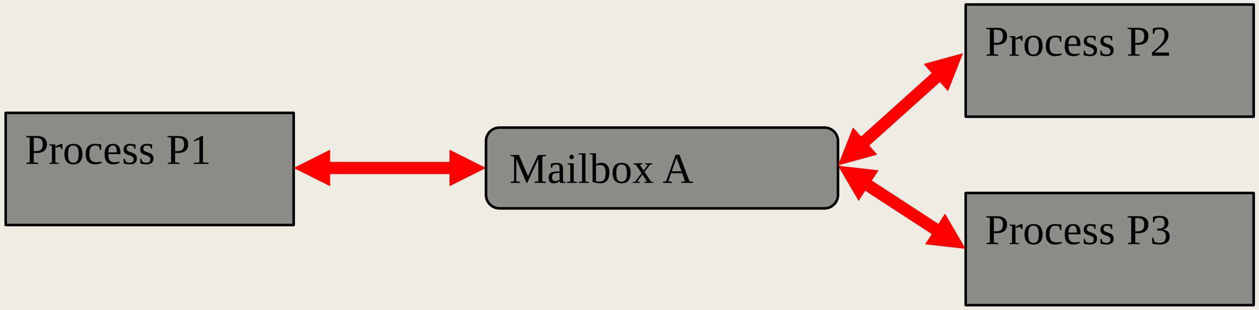


# Indirect Message Passing

- 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



# Synchronization

- 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*
    - Does not wait for it
    - A receive may “fail” several times before it succeeds at the end

# Operations on Message Queues

```
mqd_t mq_open(const char *name, int oflag,
              mode_t mode, struct mq_attr *attr);
```

```
int mq_send(mqd_t mqdes, const char *msg_ptr,
       size_t msg_len, unsigned int msg_prio);
```

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
       size_t msg_len, unsigned int *msg_prio)
```

```
int mq_close(mqd_t mqdes)
```

# Message Queue – Example

```
send(){
  mqd_t mq = mq_open("/testqueue", O_RDWR|O_CREAT, 0664, 0);
  if (mq_send(mq, av[1], strlen (av[1]) + 1, 0)<0){
       perror ("MQ Send failure");
       exit (0);
  }
  printf ("MQ Put: %s\n", av [1]);
  return 0;
}
```

```
recieve(){
  mqd_t mq = mq_open("/testqueue",O_RDWR|O_CREAT, 0664,0);
  struct mq_attr attr;
  mq_getattr (mq, &attr);  // get attribute
  char *buf = (char*)malloc (attr.mq_msgsize);
  mq_receive(mq, buf, attr.mq_msgsize, NULL);
  printf ("MQ Receive Got: %s\n", buf);
  //clean-up
  mq_close(mq);
  //mq_unlink("/testqueue"); // remove from Kernel
  return 0;
}
```

# IPC: Shared Memory

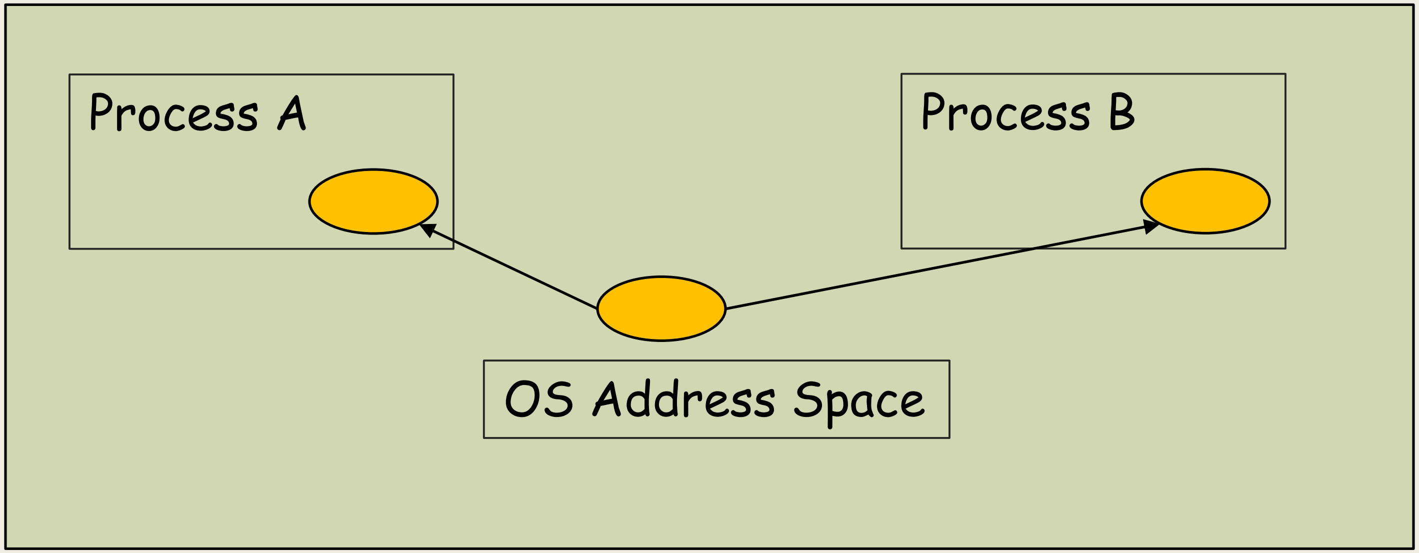
# Shared Memory

- How does data travel through a FIFO?
  - *‘write’ copies data from process memory to kernel buffer and then ‘read’ copies data from a kernel buffer to process memory*
- If both processes are on the same machine, then they may not need to copy data in and out of the kernel
  - *They may exchange or share data by using a shared memory segment*
  - *Shared memory is to processes what global variables are to threads*

# Shared Memory

- Processes share the same segment of memory directly
  - *Memory is mapped into the address space of each sharing process*
  - *Memory is persistent beyond the lifetime of the creating or modifying processes (until deleted)*
- But, now the processes are on their own for synchronization
  - ***Mutual exclusion must be provided by** processes using the shared memory*
  - ***Semaphores** – to ensure Producer-Consumer relation also has to be now done by the respective processes*

# Shared Memory




- Processes request the segment
- OS maintains the segment
- Processes can attach/detach the segment

# Facts about Shared Memory Segments

- A shared memory segment has a name, the name must start with “/”
- A shared memory segment has an owner and permission bits
- Processes may “map” or “unmap” a segment, obtaining/removing a pointer to the segment
- reads and writes to the memory segment are done via regular pointer operations

# Shared Memory – POSIX functions

- **`shm_open`**: creates a shared memory segment
- **`ftruncate`**: sets the size of a shared memory segment
- **`mmap`**: maps the shared memory object to the process's address space
- **`munmap`**: unmaps from process's address space
- **`shm_unlink`**: removes the shared memory segment from the kernel
- **`close`**: closes the file descriptor associated with the shared memory segrment

# Shared Memory Example

```
char* my_shm_connect(char* name, int len){
    int fd = shm_open(name, O_RDWR|O_CREAT, 0644 );
    ftruncate(fd, len);  //set the length to 1024, the default
is 0, so this is a necessary step
    char *ptr = (char *) mmap(NULL, len, PROT_READ|PROT_WRITE,
MAP_SHARED, fd, 0);// map
    if (fd < 0){
        perror ("Cannot create shared memory\n");
        exit (0);
    }
    return ptr;
}
```

```
void send(char* message){
    char    *name    = "/testing";
    int len = 1024;
    char* ptr = my_shm_connect (name, len);

    strcpy(ptr, message); // putting data by just copying
    printf ("Put message: %s\n", message);
    close(fd);  // close desc, does not remove the segment
    munmap (ptr, len); // this is a bit redundant,
}
```

# Shared Memory Example- contd

```
void receive(){
        char    *name    = "/testing";
        int len = 1024;
        char* ptr = my_shm_connect (name, len)

        printf ("Got message: %s\n", ptr);
        shm_unlink (name); //this removes the segment
from Kernel, this is a necessary clean up
        exit(0);
}
```

# Shared Memory - Summary

- It’s great because it does not allocate extra memory
  - *3-times less memory (In others you have 1 copy in kernel, 2 others in the individual processes)*
- But a huge problem looms
  - *Who does synchronization?*
  - *There is no guarantee of order between sending and receiving processes*
  - *The receiver process can run first finding nothing or stale data in the buffer*

# Kernel Semaphores

- We have learned how to synchronize multiple threads using Semaphores and Locks
- But how do we synchronize multiple processes?
  - *We will again need semaphores, but this time Kernel Semaphores*
  - *They are visible to separate processes who do not share address space*

# Kernel Semaphore Operations

- **`sem_open(name, …)`** to create or connect to a semaphore
  - *The name argument must start with a "/"*
- **`sem_close ()`** closes a semaphore
  - *It does not destroy it from Kernel*
- **`sem_unlink()`** removes from Kernel
  - *Must be put in the destructor for PA5*
- **`sem_wait()`** is equivalent to Semaphore::P() operation
- **`semd_post()`** is equivalent to Semaphore::V() operation

- Find out more from **`sem_overview(7)`** in man pages