INTER-PROCESS COMMUNICATION

Tanzir Ahmed

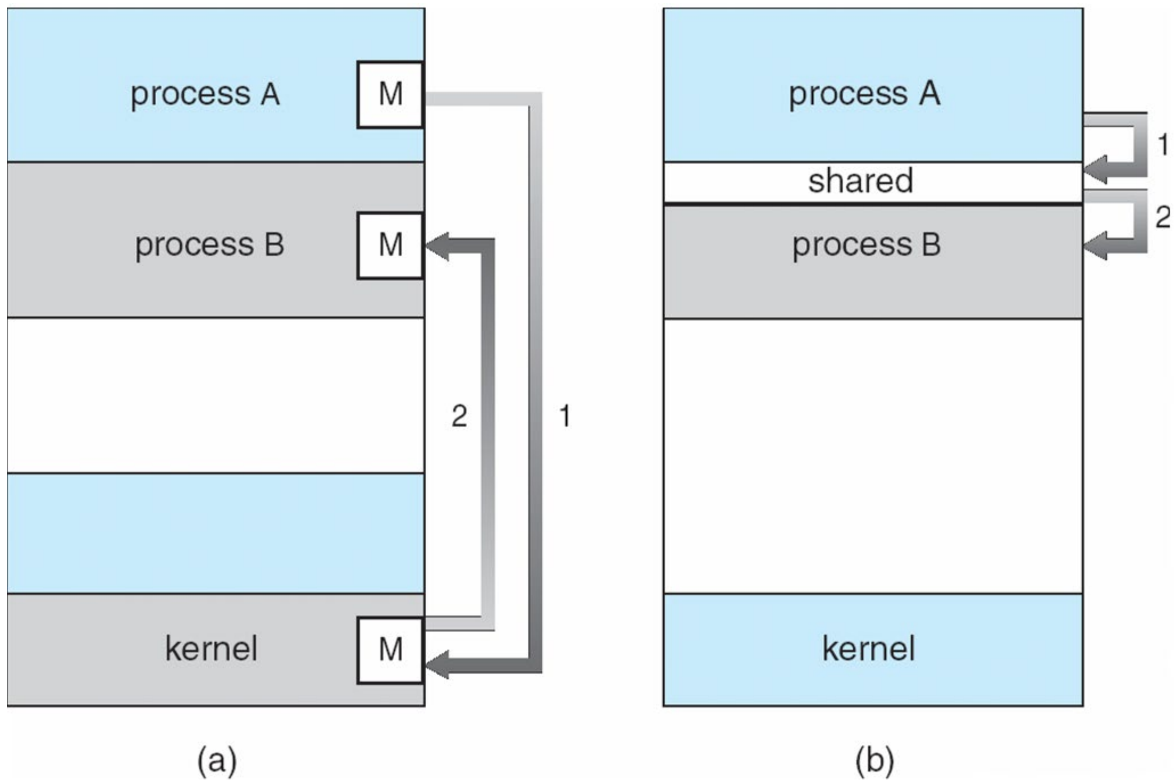
CSCE 313 Spring 2021

Inter-Process Communication

- IPC Methods
  - *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)*

IPC Fundamental Communication Models



Example: pipe, fifo, message, signal

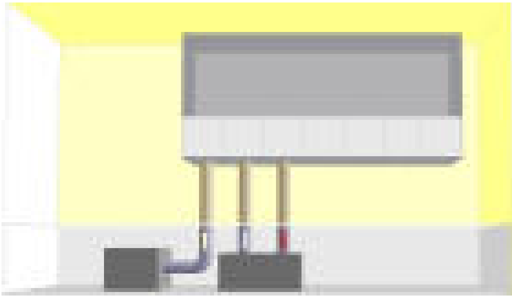
Example: shared memory, memory mapped file

Unnamed Pipes

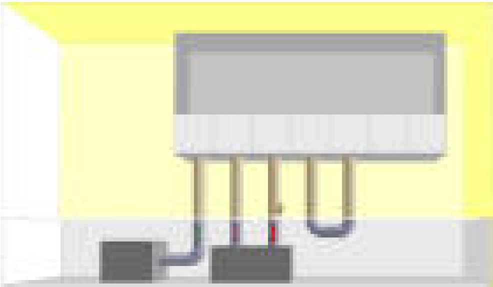
`int pipe(int fildes[2])`

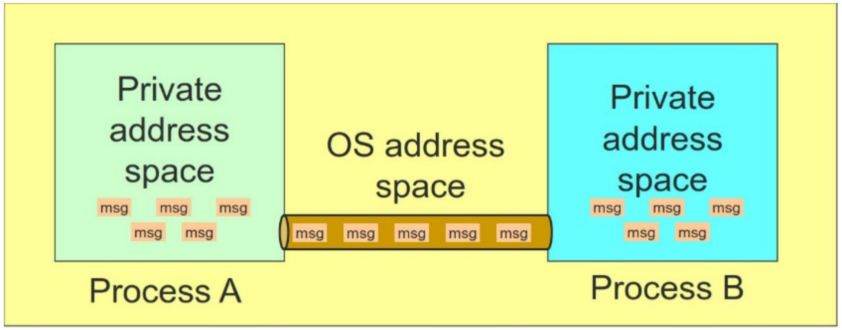
- Returns a pair of file descriptors
  - *fildes[0] is the read end and fildes[1] is the write end*
- **Create a message pipe**
  - *Data is received in the order it was sent*
  - *OS enforces mutual exclusion: only one process at a time*
  - *Processes sharing the pipe must have same parent in common*
  - *The space in between (in Kernel) is bounded (i.e., you cannot send unlimited msgs w/o receiving*

BEFORE



AFTER





IPC Pipe - Method

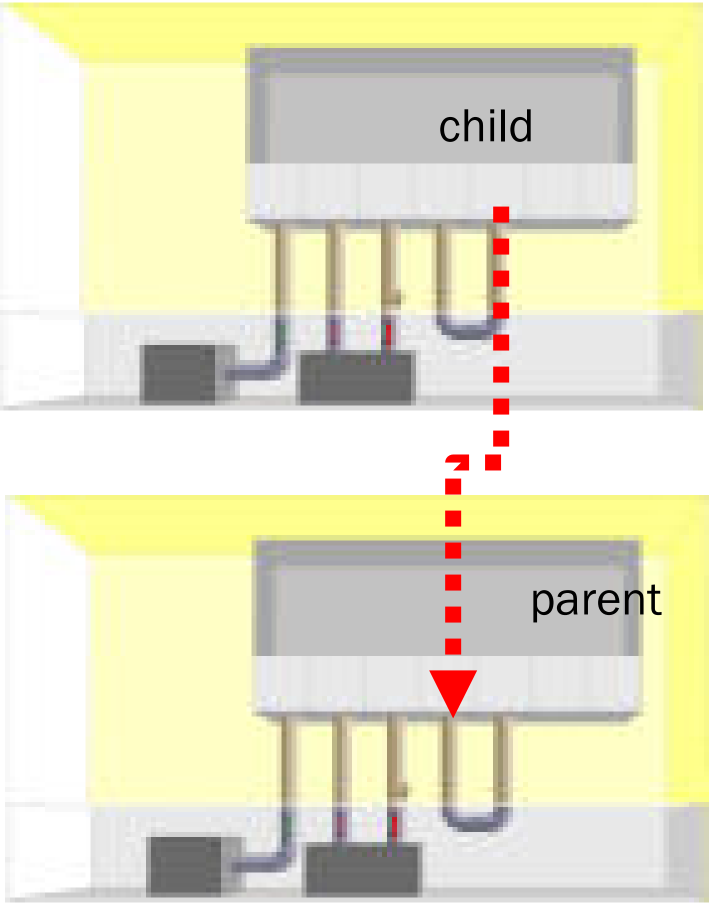
```c
#include <stdio.h>
#include <unistd.h>

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
```

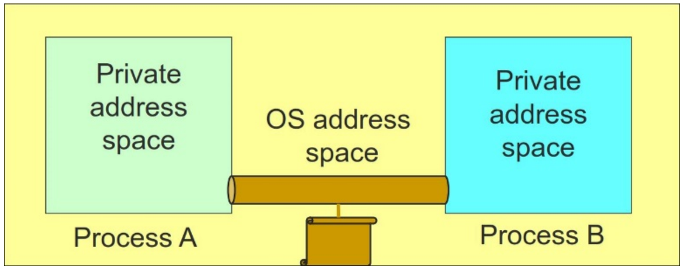
Unnamed Pipe Between Two Processes

```
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;
}
```



Named Pipes (FIFO)

- FIFOs are a mechanism that allow for IPC that's to some degree similar to using regular files
  - *Because you have to* ***`open()`*** *the pipe file to start a communication and then* ***`read()/write()`*** *to/from it*
  - *FIFOs files are persistent in disk, just like regular files.*
- However, it is also very different from using regular files in the sense that <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
  - *Filename is only used for system-wide visibility/scope of the pipe, not for containing data*
- FIFOs are similar to unnamed pipes because:
  - *They are unidirectional: 1 side can only either read or write*
  - *Mechanism is a Kernel-managed bounded queue, just like unnamed pipes*



FIFO - Problems

- The processes 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

- `mkfifo (name):` create a FIFO
- 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)`*
- Can someone open in both read and write mode?
  - *No. That makes it directional, but efficient*

FIFO DEMO

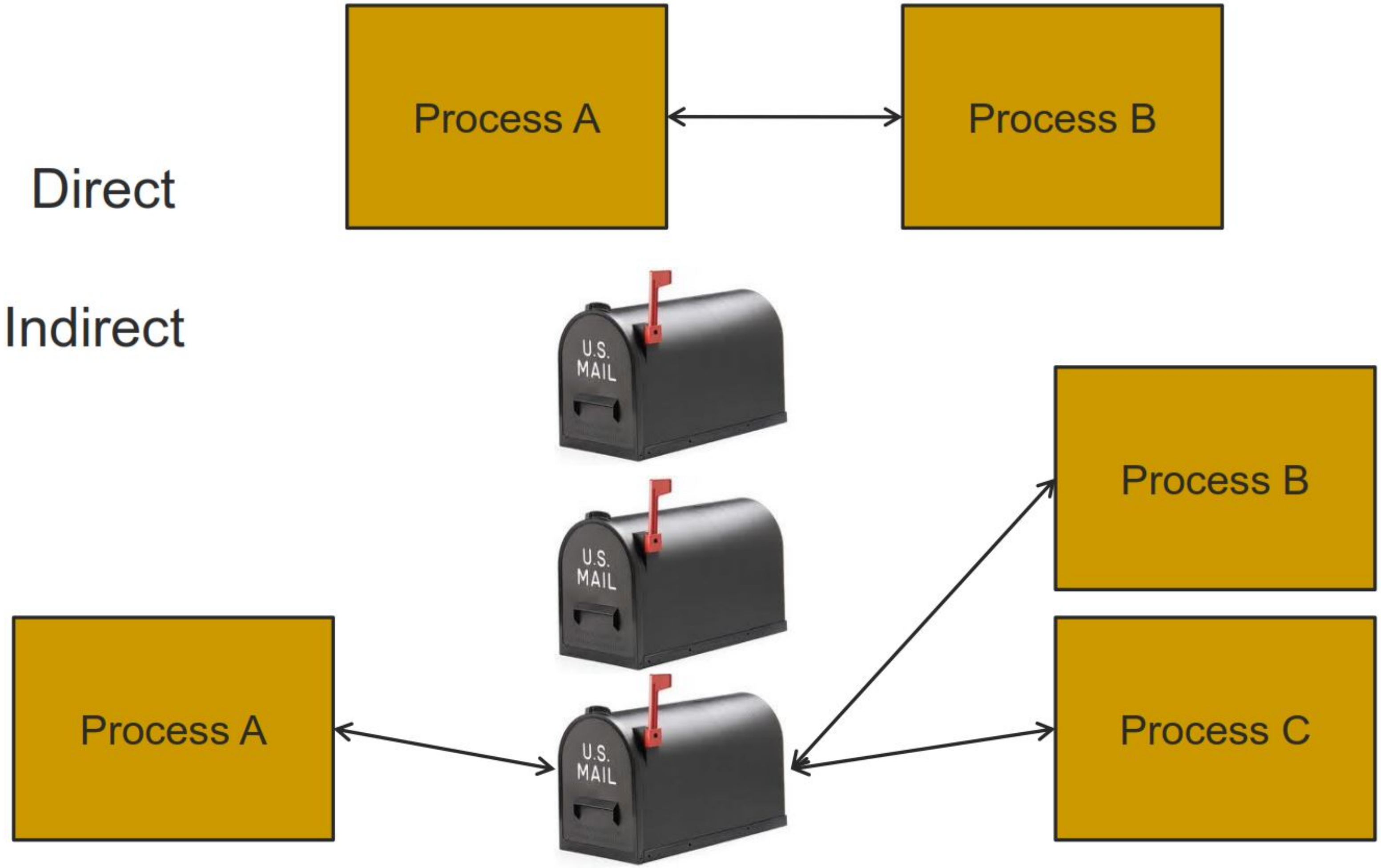
Writer

```c
#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

```c
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;
}
```

Message Queues



Added Features of MQ

- Supports Priority of messages, effectively changing the FIFO order of the messages (note pipe/FIFO messages are only FIFO)
- Multiple processes can read from or write into the same message queue – not possible in FIFO
- MQ does not require the ends be simultaneously connected
  - *FIFO requires both processes connected*
- Allows for asynchronous delivery of messages
  - *The recipient process sets up notification of message receipt using* `mq_notify()`

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)
```

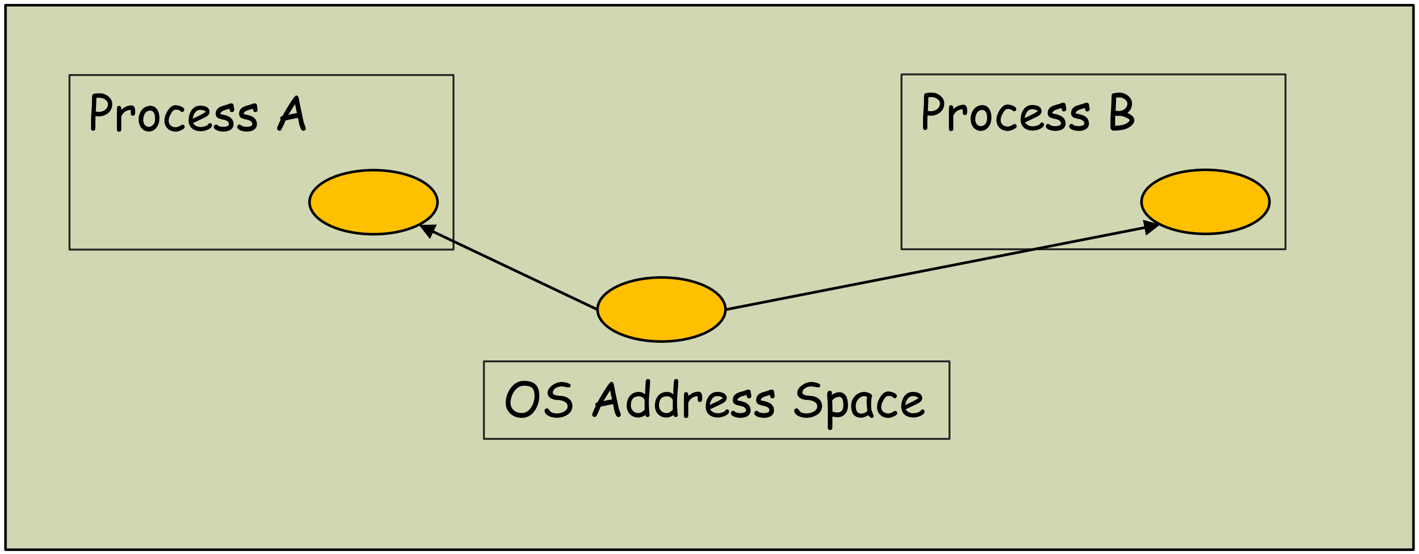
```
int mq_close(mqd_t mqdes)
```

Message Queue – Example

```
send(char* msg){
  mqd_t mq = mq_open("/testqueue", O_RDWR|O_CREAT, 0664, 0);
  if (mq_send(mq, msg, strlen (msg) + 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;
}
```

Shared Memory



- Processes request the segment
- OS maintains the segment – it persists w/o any processes connected
- Processes can `map/unmap` the segment
- Synchronization is now up to the processes
  - *No send/receive functions, must use "Kernel Semaphores"*

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);
}
```

Kernel Semaphores

- How do we synchronize processes?
  - *We will again need semaphores, but this time Kernel Semaphores*
  - *They are visible to separate processes who do not share address space*
- Operations on Kernel Semaphore:
  - ***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 PA3
  - ***sem_wait()*** *waiting for an event*
  - ***sem_post()*** *signaling that an event has occured*
- Find out more from **sem_overview(7)** in man pages