ECE437/CS481

M02C: PROCESSES & THREADS INTERPROCESS COMMUNICATIONS

Chapter 3.4-3.5

Xiang Sun

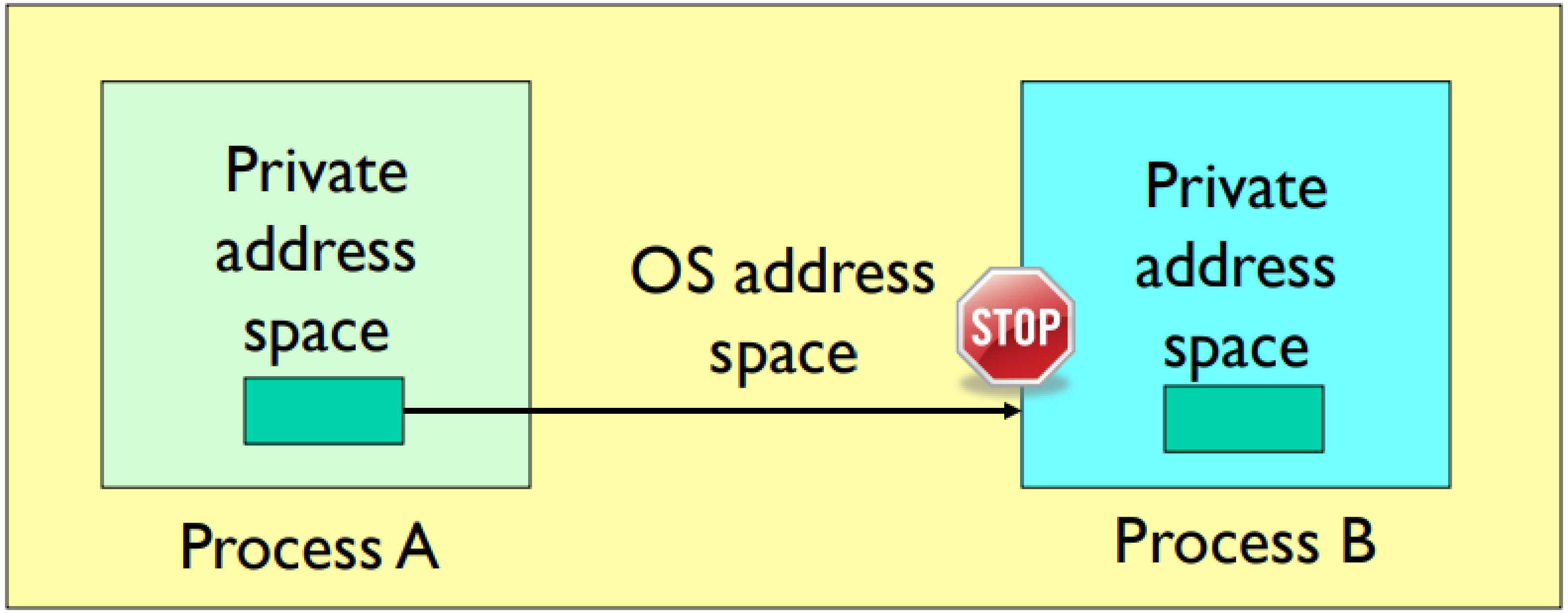
The University of New Mexico

Interprocess Communications

- Processes within a system may be?
  - Independent or Cooperative
- Cooperative process
  - Can affect or be affected by other processes, i.e., sharing data among processes
  - Benefits
    - ✓ Computation speedup
    - ✓ Convenience
- Cooperating processes need inter-process communications (IPC)
  - Shared memory
  - Message passing
  - Signals

Interprocess Communications

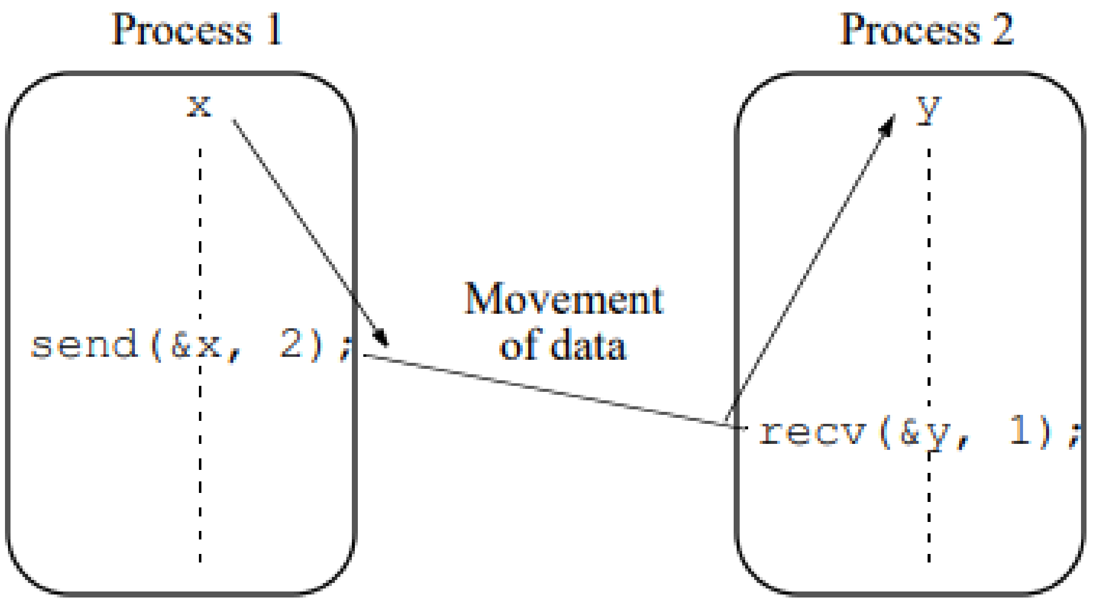
- IPC is not easy



  - Each process has a private memory space
  - No process can write to another process's space
  - How can we get data from process A to process B?

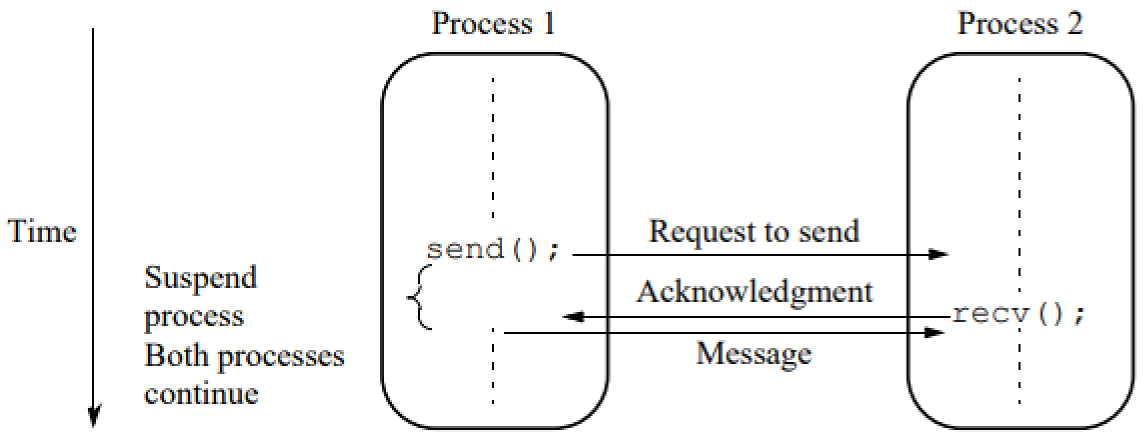
Message Passing

- Establish a link between two processes.
- Exchange messages via send/receive
- Different types of messaging passing:
  - ✓ Synchronous vs. Asynchronous
  - ✓ Direct vs. Indirect

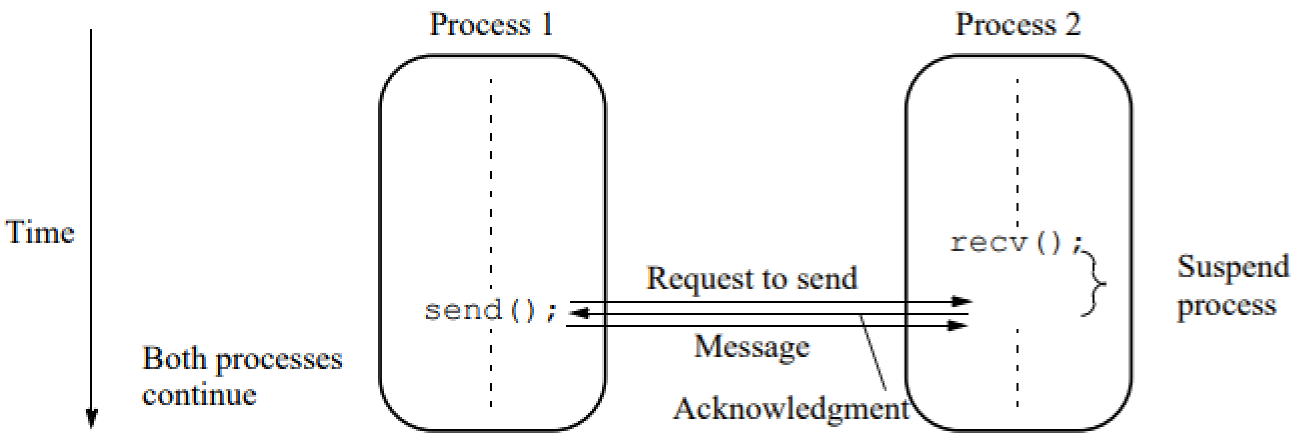


Message Passing

- Synchronous vs. Asynchronous Message Passing
  - Synchronous Message Passing without buffer
    - ✓ Both source and destination processes may suffer from a long suspension time.



(a) When send() occurs before recv()



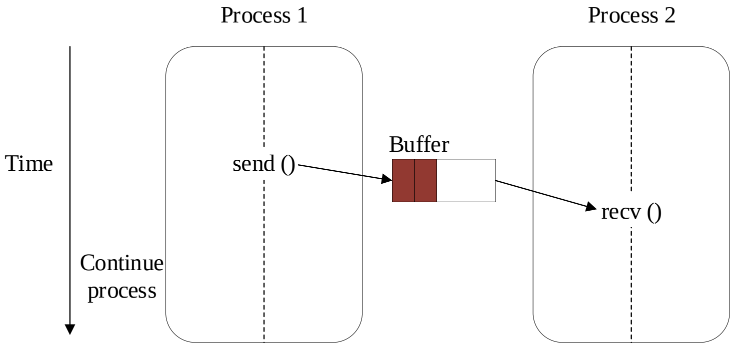
(b) When recv() occurs before send()

Message Passing

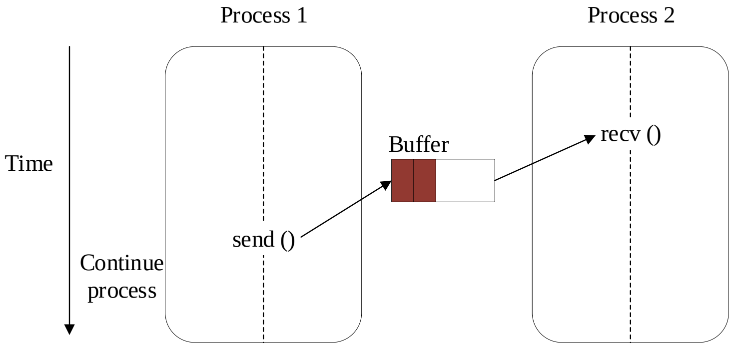
❑ Synchronous vs. Asynchronous Message Passing

➢ Synchronous Message Passing with buffer—Semi-synchronous

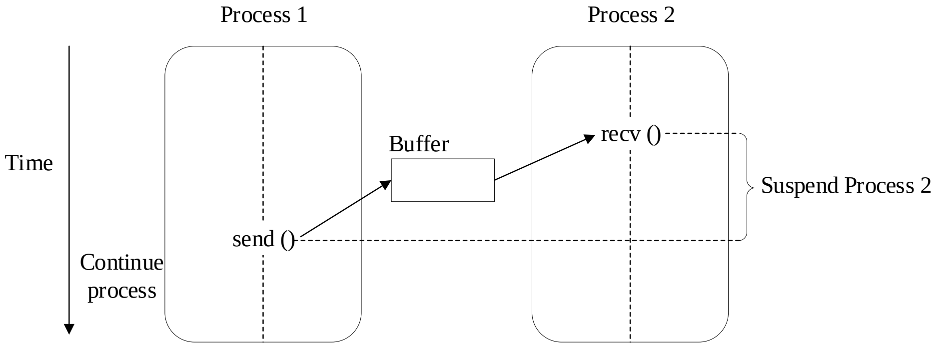
✓ A FIFO buffer is created to store messages.



a) when send() occurs before recv()



b) when send() occurs after recv() with non-empty buffer



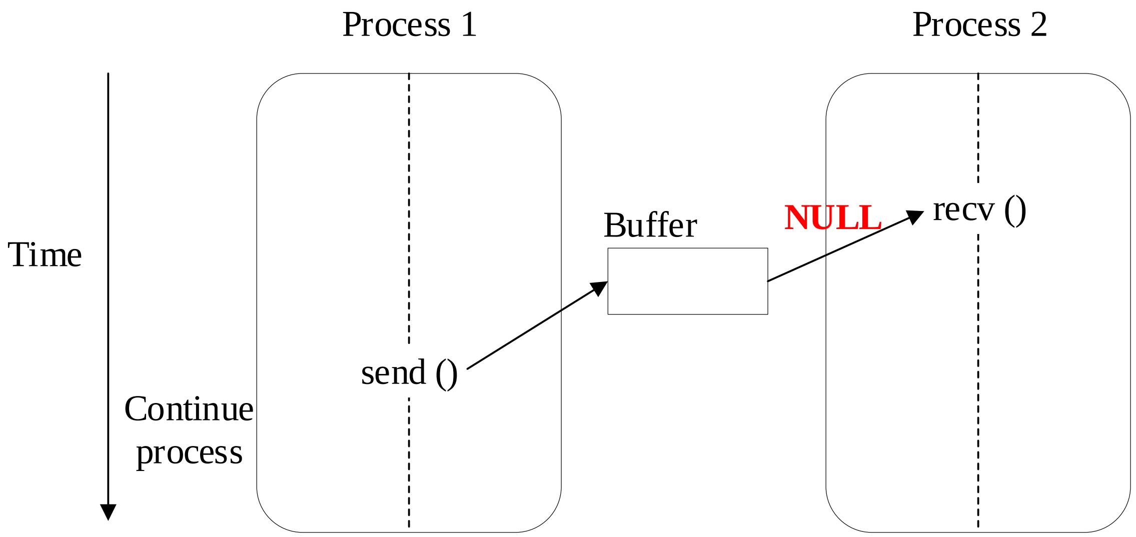
c) when send() occurs after recv() with empty buffer

Message Passing

❑ Synchronous vs. Asynchronous Message Passing

➢ Asynchronous Message Passing

- ✓ A FIFO buffer is created to store messages.
- ✓ Return "null" if the buffer is empty.



❑ Synchronous (no buffer case) vs. Asynchronous Message Passing

Source Process Side

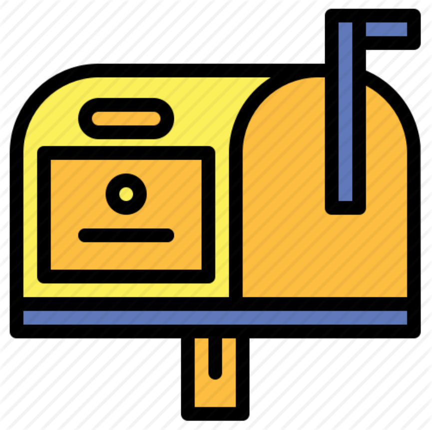
- Synchronous
  - ✓ Will be suspended until ack has been received
- Asynchronous
  - ✓ Suspend/Block on full buffer
  - ✓ Completion
    1. Require the source process to check status of a message (POLLING)
    2. Notify or signal the source process by the destination process

Destination Process Side

- Synchronous
  - ✓ Will be suspended on recv() before send()
- Asynchronous
  - ✓ Never block, but return null if buffer is empty

Message Passing

❑ Direct vs. Indirect Message Passing



- Direct message passing, PID based
  - ✓ send (P, message) – send a message to process P
  - ✓ receive(Q, message)- receive a message from process Q
- Indirect message passing, mailbox based
  - ✓ send(A, message) – send a message to mailbox A.
  - ✓ receive(A, message)-receive a message from mailbox A.
  - ✓ Each mailbox has a unique ID.
  - ✓ A mailbox can be used for multiple sender-receiver pairs for multicasting messages. That is, mailbox allows one-to-one, one-to-many, many-to-one, many-to-many communications.

Direct vs. Indirect Message Passing

- Problem of one-to-many or many-to-many communications for applying one mailbox
  - ✓ P1, P2, and P3 share mailbox *A*.
  - ✓ P1 sends two different messages to P2 and P3 via mailbox *A*, respectively.
  - ✓ Who gets the which message?

- Solutions
  - ✓ Allow a mailbox to be used by only one sender-receiver pair per communication.
  - ✓ Require lock/unlock a mailbox before using it.

Pipe

❑ Unix/Linux Pipe

- ➢ Provide processes with a simple way of passing massages
  - ✓ Just like a virutal pipe that can store a limited amount of data in a FIFO Manner
  - ✓ Constant "PIPE_BUF" defined as 5120 B, the max # of bytes a pipe may hold.
- ➢ How to communicate:
  - ➢ One process writes to the pipe, another process reads from the pipe.
  - ➢ If a process attempts to write to a full pipe, it will block.
  - ➢ If a process attempts to read from an empty pipe, it will block.

Pipe

❑ Unix/Linux Pipe

- Unnamed/anonymous pipe: Piping between the parent & child processes.
  - ✓ before calling fork(), use pipe() to create an interprocess channel

```
#include <unistd.h>
int pipe(int fildes[2]);
```

  - ✓ The argument is an array indicating two file descriptors, i.e., fildes[0] and fildes[1]. If pipe() is successful, the array will contain two new file descriptors to be used for reading and writing on a specific pipe, respectively.

```
ssize_t write(int fildes[1], const void *buf, size_t nbyte);

ssize_t read(int fildes[0], void*buf, size_t nbyte)
```

Pipe

- Unix/Linux Pipe
  - Unamed pipe example.

```c
#include <unistd.h>
#include <stdio.h>
char msg1[20] = "Hello, world!"; char msg2[20] = "Bye, world!";
main() {
char inmsg[20]; int pipedes[2]; pid_t pid;
if (pipe(pipedes) < 0) { perror("pipe call failure"); exit(1); }
  switch (pid=fork()) {
  case -1: perror("fork call failure"); exit(2);
  case 0: // read from the pipe
        read(pipedes[0], inmsg, sizeof(msg1)); printf("%s\n",inmsg);
        read(pipedes[0], inmsg, sizeof(msg2)); printf("%s\n",inmsg); break;
  default: // write to the pipe
        write(pipedes[1], msg1, sizeof(msg1));
        write(pipedes[1], msg2, sizeof(msg2)); wait(NULL); break;
  }
}
```

```
shaun@shaun-VirtualBox:~$ ./pipe_test
Hello, world!
Bye, world!
```

Pipe

❑ Unix/Linux Pipe

- ➢ Named pipe: Piping between a server process & a client process (non parent-child).
  - ✓ A server process creates a pipe/file with a pipe/file name, writes into it. Then, a client process reads from the file using the same pipe/file name.
  - ✓ Use mkfifo() to create a named pipe

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

Unix/Linux Pipe

- Named pipe example: two processes A & B, process A writes a msg to a named pipe and process B reads it from the named pipe.

Operation permissions	Octal value
Read by user	00400
Write by user	00200
Read by group	00040
Write by group	00020
Read by others	00004
Write by others	00002

✓ Process A

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

main() {
int fd; extern int errno; char msg1[20] = "Hello, world!";
if (mkfifo("myfifo",0666) < 0 && errno != EEXIST) {
perror("mkfifo failure");
exit(1);
}
if ((fd = open("myfifo",O_WRONLY)) < 0) {
perror("open failure");
exit(2);
}
write(fd, msg1, sizeof(msg1));
}
```

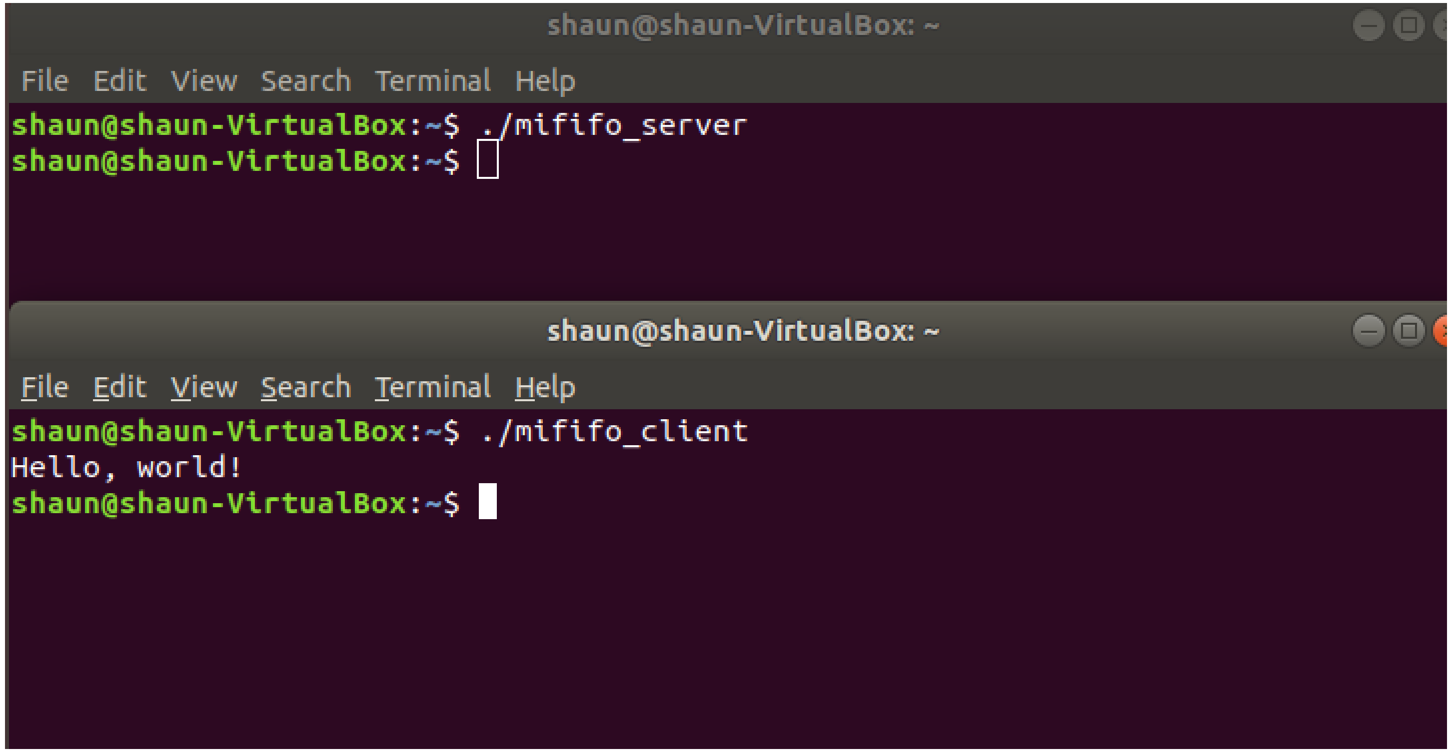
Pipe

❑ Unix/Linux Pipe

- ➢ Named pipe example: two processes A & B, process A writes msg to a named pipe and process B reads them from the named pipe.

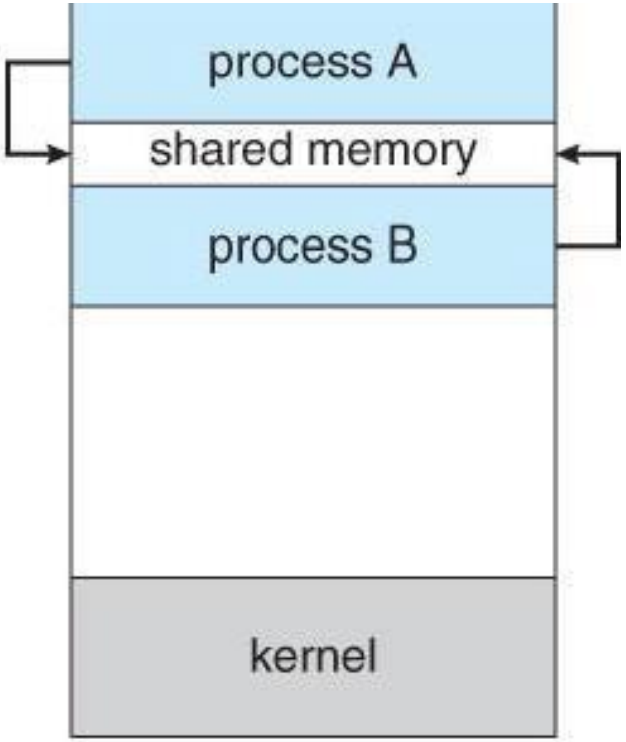
✓ Process B

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

main() {
int fd; extern int errno; char inmsg1[20];
if (mkfifo("myfifo",0666) < 0 && errno != EEXIST) {
perror("mkfifo failure");
exit(1);
}
if ((fd = open("myfifo",O_RDWR)) < 0) {
perror("open failure");
exit(2);
}
read(fd, inmsg1, sizeof(inmsg1));
printf("%s\n",inmsg1);
}
```



❑ Shared Memory

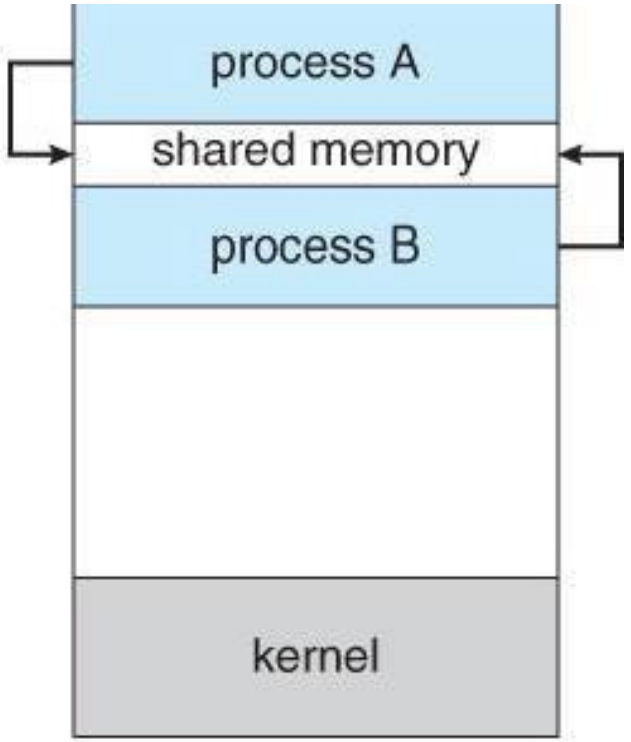
- Data is exchanged by placing it in memory pages shared by multiple processes.
- The shared memory pages may not be physically contiguous.
- What is the difference between shared memory and asynchronous message passing?
  - ✓ The buffer in asynchronous message passing only temporarily stores the data. The data are removed once they are delivered. Processes are unable to modify the data in the buffer.
  - ✓ The shared memory pages could store the data until the shared memory pages are deleted. Processes are able to modify the data in the shared memory pages.



❑ Shared Memory

- Once a shared-memory segment is created, other processes that wish to communicate using this shared-memory segment mush attach it to their address spaces.

- Normally, the OS tries to prevent one process from accessing another process's memory. However, shared memory is an exception.



❑ Shared Memory

- Shared memory has four operations:

  - ✓ int shmget(key_t key, size_t size, int shmflg): get/create a shared memory segment or obtain the identifier of a previously created shared memory segment. shmget() returns an ID for the shared memory segment upon successful completion.

  - ✓ void *shmat(int shmid, void *shmaddr, int shmflg): attach the shared memory segment to the memory space of the process.

  - ✓ int shmdt(void *shmaddr): detach the process from the shared memory segment.

  - ✓ int shmctl(int shmid, int cmd, struct shmid_ds *buf): destroy/delete the shared memory segment (where cmd=IPC_RMID and *buf=NULL).

❑ Shared Memory Example—Share_mem_wrt

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27

main()
{
    char c;
    int shmid;
    char *shm, *s;

    key_t key = ftok("shmfile",65);
    /*Create the segment.*/
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    /*Now we attach the segment.*/
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    /*Now put some things into the memory for the other process to read.*/
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

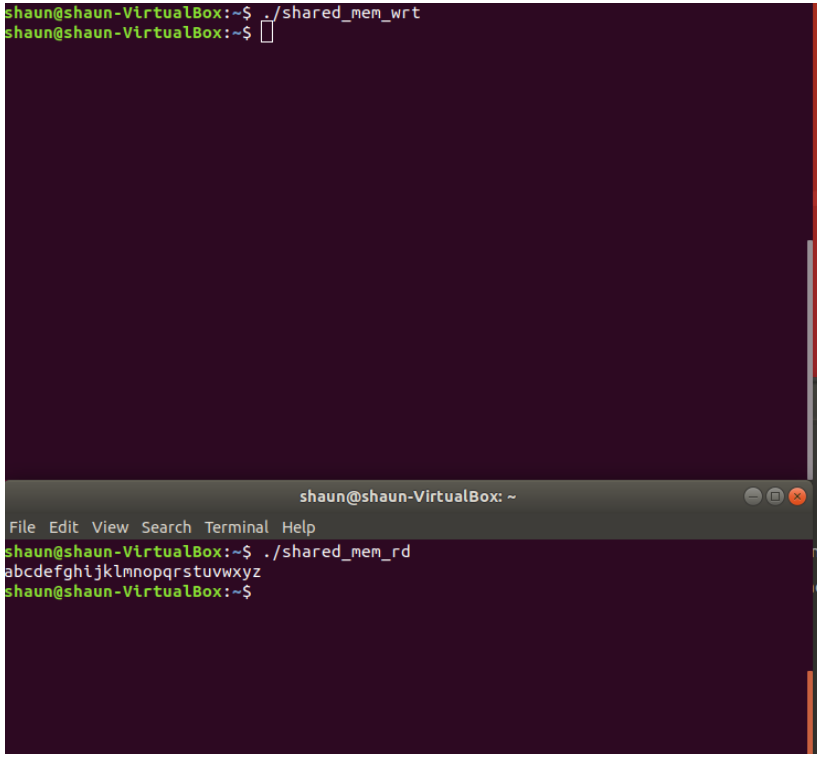
Operation permissions	Octal value
Read by user	00400
Write by user	00200
Read by group	00040
Write by group	00020
Read by others	00004
Write by others	00002

The process waits until the other process changes the first character of shared memory to '*', indicating that some process has read what we put there.

Shared Memory

❑ Shared Memory Example—Share_mem_rd

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27

main()
{
    int shmid;
    char *shm, *s;

    key_t key = ftok("shmfile",65);
    //Locate the segment.
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    //Now we attach the segment
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    //Now read what the server put in the memory.
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    *shm = '*';
    exit(0);
}
```



Finally, the process changes the first character of the segment to '*', indicating the process has read the segment.

IPC keys

- IPC keys

  - An IPC key is the number/ID to identify an IPC object. An IPC object can be
    - ✓ msg--a message queue
    - ✓ mbox-a mailbox
    - ✓ sem—a semaphore (to be covered later)
    - ✓ shm—a shared-memory segment

  - An IPC key allows an IPC object to be shared among different processes.

  - Q: We have the ID of the created shared memory (which is the return value of shmget()). Why we have to use IPC key to identify the created shared memory?

IPC keys

- ftok(): a library function of mapping characters (normally, a file's pathname) into a key.

- Synopsis/Syntax:

```
include <sys/types.h>
key_t ftok(const char *path, int id);
```

  - The ftok() function returns a key based on path and id that is usable in subsequent calls to msgget, semget and shmget.

  - The ftok() function will return the same key value for all paths that name the same file, when called with the same id value, and will return different key values otherwise.

Signals

- Unix/Linux signals
  - What is a signal?
    - ✓ A short message sent to inform a process or a group of processes about certain event.
    - ✓ A simple IPC, but not for data/information exchanging.
  - Who sends a signal to whom
    - ✓ Normally, initiated by one process, and deliver to another process (or process group) via the kernel.

Unix/Linux signals

- How to respond?

  ✓ When a process receives a signal, it can handle it in three ways:

1. By default --- the OS takes care of it
2. User-defined actions --- the process specifies which routine to execute when a certain signal is received and can be caught.
3. By ignoring --- no action taken upon receipt of signal.

Signals

❑ Unix/Linux signals

- Different kinds of signals.
  - ✓ Each of them are associated with an integer/signal number (1,2,3…).
  - ✓ Use the command "kill –l" to see a list of signals.

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Signals

❑ Unix/Linux signals

- Different ways of sending signals—Keyboard
  Special keys are interpreted by OS as requests to send signals to the process.

1. Ctrl-C
   - Send signal SIGINT to the running process.
   - By default, SIGINT causes the process to gracefully terminate.
   - SIGINT can be caught. Thus, the process can define the related routine to handle the SIGINT signal.
   - SIGINT can be ignored.

2. Ctrl-Z
   - send signal SIGTSTP to the running process.
   - By default, SIGTSTP causes the process to suspend execution.
   - SIGTSTP can be caught and ignored.

Signals

❑ Unix/Linux signals

➢ Different ways of sending signals—<u>Shell command</u>
Use various command to send signal to a particular process identified by PID

1. kill –INT 1234 (kill –2 1234)
   ➢ Send signal SIGINT to the process with PID=1234.
   ➢ By default, SIGINT causes the process to terminate.

2. Kill –CONT 1234
   ➢ Send signal SIGCONT to the process with PID=1234.
   ➢ By default, SIGCONT causes the process to resume execution.

Signals

❑ Unix/Linux signals

- Different ways of sending signals—System call
  Use various command to send a signal to a particular process identified by PID

1. kill(pid_t pid, int sig)
    - Use system call “kill” to send a signal from one process to another process.
    - e.g., kill (1234, SIGINT).
2. abort()
    - Send SIGABRT to itself
3. alarm(int secs)
    - Send SIGALRM to itself when the interval expires

Unix/Linux signals

- **Signal handler**—a software interrupt being invoked when the process receives the signal.
  - When the signal is sent to the process, OS stops the execution of the process, and "forces" it to call the signal handler function.
  - Signals are not reliable
    - A signal can be ignored/blocked: the signal will not be delivered until it is later unblocked.
    - Two special signals (i.e., SIGKILL and SIGSTOP) cannot be ignored, blocked, and caught.

Signals

❑ Unix/Linux signals

- **Default signal handler**—OS sets up a set of default signal handlers for your program
- **User-defined signal handler**—Users define their signal handlers for their programs
  - POSIX signal handling function:

    int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact)

    1st argument specifies any signal (except SIGKILL & SIGSTOP) that wants to be caught.
    2nd argument specifies new action for the caught signal.
    3rd argument saves the previous action for the caught signal.

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

❑sigaction() example

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

void userDef_signalHandler(int sig)
{
  printf("oh, got a signal %d\n", sig);
  int i = 0;
  for (i = 0; i < 5; i++)
  {
      printf("signal func %d\n", i);
       sleep(1);
  }
}


int main()
{
   struct sigaction act;
   act.sa_handler = userDef_signalHandler;
   sigemptyset(&act.sa_mask);
   sigaddset(&act.sa_mask, SIGQUIT);
   act.sa_flags = 0;

   sigaction(SIGINT, &act, 0);

   struct sigaction act_2;
   act_2.sa_handler = userDef_signalHandler;
   sigemptyset(&act_2.sa_mask);
   act.sa_flags = 0;
   sigaction(SIGQUIT, &act_2, 0);

   while(1)
   {
    }
    return;
}
```

```
shaun@shaun-VirtualBox:~/OS_code/IPC$ ./sigaction
^Coh, got a signal 2
signal func 0
signal func 1
signal func 2
signal func 3
signal func 4
^\oh, got a signal 3
signal func 0
signal func 1
signal func 2
signal func 3
signal func 4
```