

Linux Systems and Open Source Software

Inter-Process Communication (IPC)

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Fall 2022



Outline

- Overview
- Asynchrony
- Signals
- Pipes
- Shared Memory
- Sockets
- Command line tools



Overview

- Inter-process communication (IPC) refers to the communication between processes:
  - on the same machine,
  - on different machines

- In particular, IPC are the mechanisms offered by an operating system for managing shared data for the processes
- There are different approaches to IPC on Linux/Unix based systems:
  - ✓ Signals
  - ✓ Pipes
  - ✓ Shared memory
  - ✓ Sockets
  - Message queues
  - FIFOs
  - File-based IPC
  - …

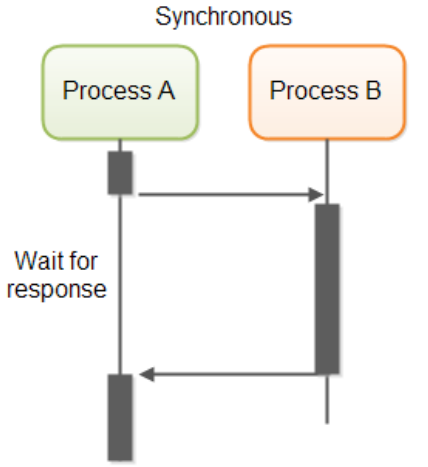
We will cover the four IPCs in this course

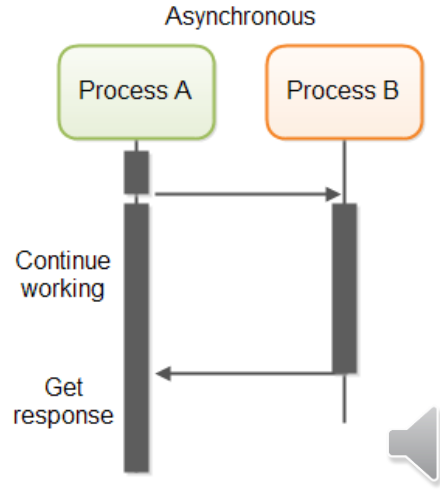
Courtesy of Inter-process communication, wikipedia



Asynchrony

- Asynchrony
  - the state of not being in synchronization
  - occurrence at different times
- In computing, it means the occurrence of events independent of **the main program flow** and ways to deal with such events
  - These may be "outside" events regarding the main flow,
    - such as the arrival of signals, or actions instigated by a program
  - that take place concurrently with program execution, without the program blocking to wait for results
- Example
  - **Synchronous**: Process A ***blocks and waits*** the completion of the processing done by Process B
  - **Asynchronous**: Process A ***continues its work*** **while** Process B performing the job that are needed by Process A (Concurrent processing)





Courtesy of Asynchrony (computer programming), Wikipedia
What is Synchronous and Asynchronous; Collins dictionaries; Boost application performance using asynchronous I/O



Asynchrony (Cont'd)

- Execution schemes:
  - **Synchronous execution**: the execution happens in a single series
  - **Asynchronous execution**: one process execution overlaps the other; i.e., they are running concurrently

```
Synchronous execution:
|--------A--------|
                  |--------B--------|
```

```
Asynchronous execution:
|--------A--------|
    |--------B--------|
```

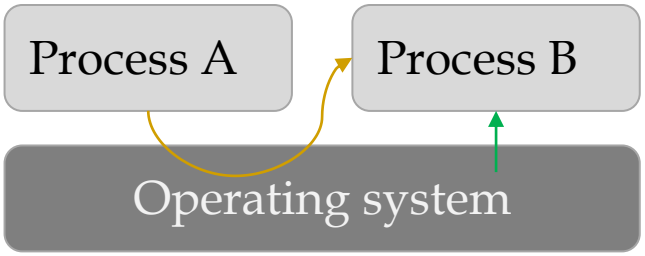
- Scenarios for asynchrony:
  - I/O operations
    - I/O operations like file I/O, network I/O, database I/O, (which is generally just network I/O), and web service calls benefit from asynchrony
    - Anytime that your code is making a database call, or a web service call or a network call or talking to the file system, that can be done asynchronously while that operation is in progress
  - Performing multiple operations in parallel
    - When you need to do different operations in parallel, for example, making a database call, web service call and any calculations, then you can use asynchrony

Courtesy of Asynchrony (computer programming), Wikipedia
What is Synchronous and Asynchronous



Signals

- A signal
  - is an ***asynchronous notification*** sent to a process (or to a specific thread) in order to notify it of an event that occurred
- Two types of signals:
  - Signals from Process A to B
  - Signals from OS to Process B



E.g., issue the **kill** command in console to ***kill*** a process

- Signals vs. Interrupts
  - the difference being that interrupts are mediated by the processor and handled by the kernel
  - while signals are mediated by the kernel (possibly via system calls) and handled by processes



What is a Signal?

SIGTSTP – 1 – Stop typed at terminal (Ctrl+Z)
SIGKILL – 9 – Kill a process, for realzies
SIGSEGV – 11 – Segmentation fault
SIGSTOP – 19 – Stop a process
SIGCONT – 18 – Continue if stopped

- Information about signals
  - → A signal is raised by a process (or a thread) to send the *numeric value* without containing any ***message***
  - Signals are unidirectional communications
  - Signals are defined by numeric values; usually less than 64 numbers
  - A process (or a thread) register a handler function
    - E.g., **signal()** or **sigaction()** to associate *a signal handler* with *the signal*
  - Default action will be taken if the signal handler is not registered
    - E.g., **SIGKILL** and **SIGSTOP** is the default signals to ***kill*** and ***stop*** the designated process, respectively; the two signals cannot be caught, blocked, or ignored



Workflow

- Workflow from figure below*
  1. Process A raises the signal
  2. OS interrupts the target process' normal execution flow (as shown at the right figure) to deliver the signal
  3. Target process B handles the registered signal
  4. Target process finished the signal handling
  5. OS returns the signal handling to Process A

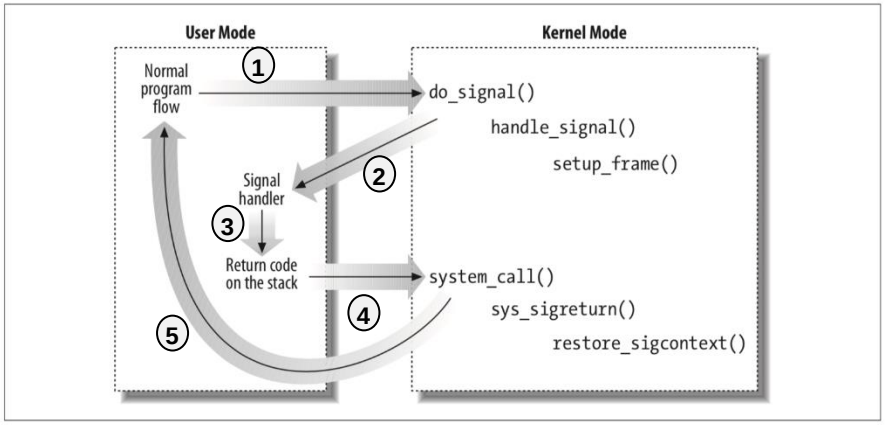
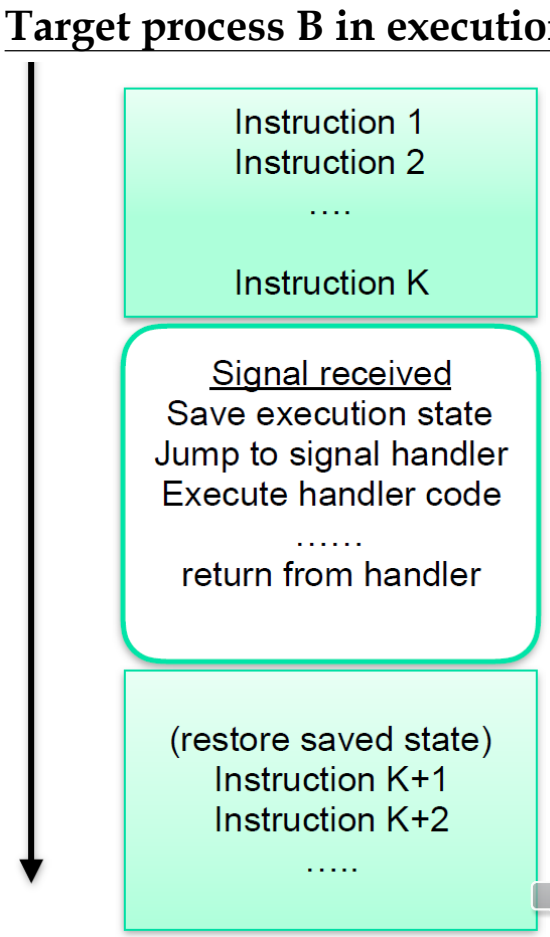


Figure 11-2. Catching a signal



Courtesy of Signal (IPC), Wikipedia; CS550 - Operating Systems, Kartik Gopalan

*Understanding the Linux Kernel, Daniel Pierre Bovet and Marco Cesati, O'Reilly Media; 3 edition (ISBN-10: 0596005652)



An Example: IPC between Two Processes

- Process A
  - Calls **signal()** to register that the signal **SIGUSR1** is handled by the function pointed by **&usr_handler**
- Process B
  - Invokes the call to **kill(),**
  - which is used to send signal to a process
    - E.g., **kill(300, SIGUSR1)** sends the signal **SIGUSR1** to the process w/ pid=**300**
- The **usr_handler()** in Process A is called to handle the signal

Process A (pid: 300)

```
int main() {
    ...
    signal(SIGUSR1, &usr_handler);
    ...
}
```

Process A (pid: 300)

```
int main() {
    ...
}
```

PC

Process B

```
kill(300, SIGUSR1);
```

```
int usr_handler() { …
```

NOTE: raise() function does the similar thing as **kill()**

NOTE: **raise()** function blocks and wait for the handler returns



You may like to know more about …

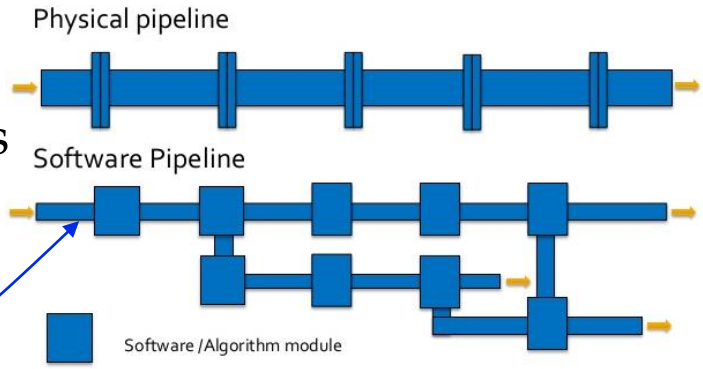
- Signals implementations
  - There is a variant of signals: POSIX reliable signals (hereinafter "standard signals"), POSIX real-time signals, ANSI C signal handling
  - Be aware of your chosen implementation; e.g., a real-time signal can keep a counter
- Nested signals
  - The **signal()** function **does not (necessarily) block other signals** from arriving while the current handler is executing
  - **sigaction()** **can block other signals** until the current handler returns
  - ✓ **sigaction()** is recommended to be used, especially in a multi-threaded environment
  - Check this page for more details (some corner cases)
  - Check the man pages for signal(7) for signals overviews, signal(2) for ANSI C signal handling, sigaction(2) for examining and changing a signal action
- **Async-signal-safe** functions
  - An async-signal-safe function is one that can be safely called from within a signal handler. Many functions are not async-signal-safe
  - In particular, **non-reentrant functions are generally unsafe to call from a signal handler**. The kinds of issues that render a function unsafe can be quickly understood when one considers the implementation of the STDIO library, whose functions are not async-signal-safe
  - When performing buffered I/O on a file, the stdio functions must maintain a statically allocated data buffer along with associated counters and indexes (or pointers) that record the amount of data and the current position in the buffer. Suppose that the main program is in the middle of a call to a stdio function such as printf(3) where the buffer and associated variables have been partially updated. If, at that moment, the program is interrupted by a signal handler that also calls printf(3), then the second call to printf(3) will operate on inconsistent data, with unpredictable results

Courtesy of signal-safety(7) — Linux manual page; COMP 790: OS Implementation, Don Porter

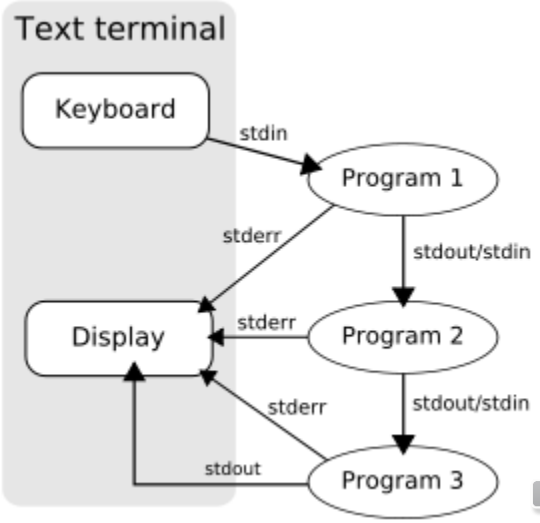


Pipes

- Software pipeline
  - A chain of software processing modules (e.g., processes, threads, and functions), is arranged in a way so that the output of each element is the input of the next
  - Usually some amount of ***buffering*** is provided between consecutive elements

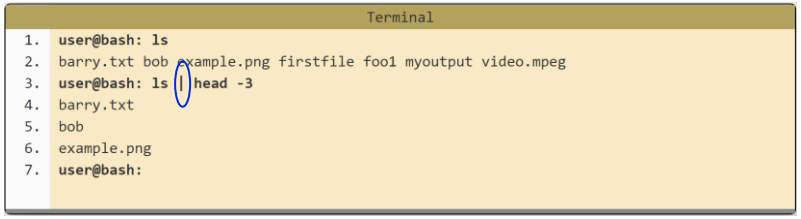


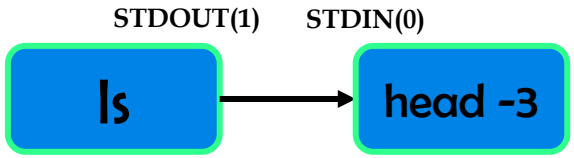
- A pipeline is a mechanism for inter-process communication using message passing
  - A pipeline is a set of processes chained together by their standard streams,
  - so that the output text of each process (STDOUT) is passed directly as input (STDIN) to the next one



Data Streams & Pipes

- Each program has three data streams connected to it
  - STDIN (0)
    - Standard input (data fed into the program)
  - STDOUT (1)
    - Standard output (data printed by the program, defaults to the terminal)
  - STDERR (2)
    - Standard error (for error messages, also defaults to the terminal)





- The *pipe operator* is '|'
  - found above the backslash '\' key on most keyboards
  - feeding the output from the program on the left as input
  - to the program on the right
- The above figures show that
  - the output of **ls** is piped to the input for "**head -3**"
- The example uses an *anonymous pipe*,
  - where data written by one process is buffered by the operating system until it is read by the next process, and
  - this uni-directional channel *disappears* when the processes are completed

Courtesy of Pipeline (Unix), wikipedia; Piping and Redirection!

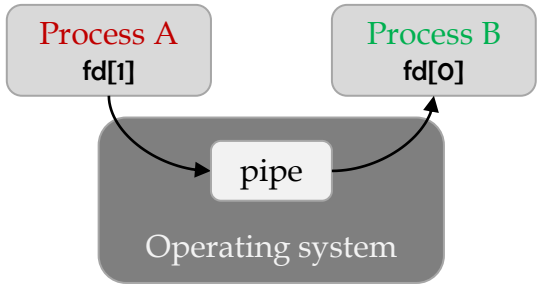


A Pipe

- Properties
  - A pipe is usually a unidirectional data channel
  - Only used between processes having a common ancestor process
  - Can have multiple readers/writers

- Create a pipe with **pipe()**
  - which returns the array **pipefd** with two ***file descriptors*** referring to the read/write ends of the pipe
  - **pipefd[0]** refers to the read end of the pipe
  - **pipefd[1]** refers to the write end of the pipe
  - Use **read()** to get data from the read end of the pipe
  - Use **write()** to write data to the write end of the pipe
  - Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe

A pipe example code (source)




```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void)
{
    int pipefd[2];
    char buf[30];
    pipe(pipefd);
    if (!fork()) { // Process A (Child process)
        printf(" CHILD: writing to the pipe\n");
        write(pipefd[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else { // Process B (Parent process)
        printf("PARENT: reading from pipe\n");
        read(pipefd[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }
    return 0;
}
```

- Each process has its own **pipefd**
- Refer to fork() and file-descriptor table for details

The output of the code

```
PARENT: reading from pipe
 CHILD: writing to the pipe
 CHILD: exiting
PARENT: read "test"
```



File Manipulations

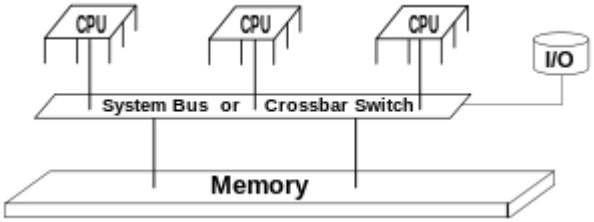
- From **stdio.h**, you uses the APIs to manipulate a file
  - **fopen()**: Opens the file with the filename
  - **fwrite()**: Writes the given data to the given stream
  - **fread()**: Reads the data from the given stream
  - **fclose()**: Closes the stream of the file
- You could also use the Linux system calls to manipulate a file
  - **open()**: Opens the file with the filename
  - **write()**: Writes the given data to the given stream
  - **read()**: Reads the data from the given stream
  - **close()**: Closes the stream of the file

- You can search online to find the differences between **fopen()** and **open()**

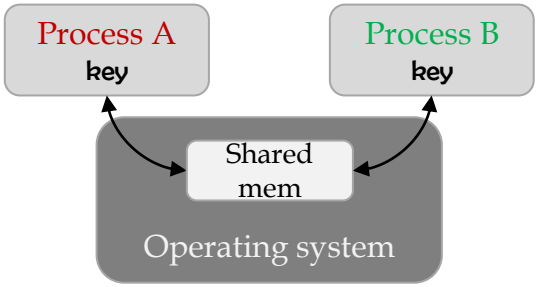


Shared Memory

- Computer hardware
  - Shared memory refers to a block of random access memory (RAM) that can be accessed by several different processors (or cores) in a multiprocessor computer system
  - It is also very important on multicore systems



- Computer software
  - Shared memory is a method of exchanging data between processes running at the same time
  - One process will create an area in RAM where other processes can access
  - A very ***fast*** way of ***bi-directional communications*** opposed to other mechanisms of IPC such as named pipes, Unix domain sockets
  - It is ***less scalable***, as for example the communicating processes must be **running on the same machine**



Courtesy of Shared memory, wikipedia



Shared Memory APIs

- Shared memory allows ***two unrelated processes*** to access the same logical memory
  - It is probable that most implementations of shared memory arrange for the memory being shared between different processes to be **the same physical memory**
- Concurrent accesses
  - There are ***no automatic facilities*** to prevent a second process from starting to read the shared memory before the first process has finished writing to it
  - ***It's the responsibility of the programmer to synchronize access***
    - ***E.g., with semaphores***

- APIs
  - **shmget()**: create shared memory
  - **shmat()**: attach the created shared mem to the address space of the calling process so as to access the memory
  - **shmdt()**: detach the shared memory from the calling process
  - **shmctl()**: is the catchall for various shared memory operations, e.g., destroying the created memory

Illustration of shared memory

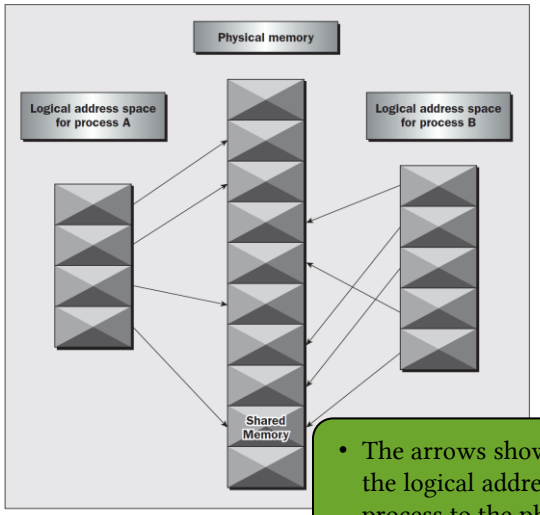


Figure 14-2

- The arrows show the mapping of the logical address space of each process to the physical memory



Shared Memory APIs (shmget)

- Processes use the same ***key*** to uniquely specify which shared memory to access

A shared memory example code (source)

```
key_t key;
int shmid;
key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

- Generate the ***key*** via **ftok()**
  - Which takes **a file path** and **an arbitrary character** to generate ***a probably-unique key***
  - Processes must generate the same key by passing the same parameters to **ftok()** for accessing the same memory
- Create/obtain the shared memory ***identifier*** via the key using **shmget()**
- **int shmget (key_t key, size_t size, int shmflg);**
  - the **key** generated by **ftok()**
  - the **size** in bytes of the shared memory
  - **shmflg** is the permissions of the segment bitwise-OR with IPC_CREAT
  - It is ok to use IPC_CREAT every time; it will simply connect you if the segment already exists



Shared Memory APIs (shmat)

- A process should attach itself to the created shared memory by **shmat()** before using it

- **void *shmat(int shmid, void *shmaddr, int shmflg);**
  - **shmid** is the shared memory ID returned from **shmget()**
  - **shmaddr** is the attaching address; if it is NULL, the system chooses a suitable (unused) page-aligned address to attach the memory
  - **shmflg** is a bit-mask argument for usage, e.g., **SHM_EXEC**, **SHM_RDONLY**

A shared memory example code (source)

```
key_t key;
int shmid;
char *data;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);

if (data == (char *)(-1))
    perror("shmat");
```

- **shmat()** returns the **void** pointer to the shared memory
  - The program treats the pointer (**data**) as a **char** pointer



Shared Memory APIs (Read, Write, shmdt and shmctl)

- Read/Write from/to the shared memory as ordinary **char** pointer
  - E.g., **data[0] = 'a';**
- Besides, it is more efficient to use those functions that can handle the **char** pointer to help:
  - **printf()**
    E.g., **printf("shared contents: %s\n", data);**
  - **gets()**
    E.g., **printf("Enter a string: "); gets(data);**
- Detach the calling process from the shared memory
  - **int shmdt(void *shmaddr);**
    - **shmaddr** is the address returned by **shmat()**
- Destroy the shared memory
  - E.g., **shmctl(shmid, IPC_RMID, NULL);**
  - **int shmctl(shmid, cmd, buf);**
    - **shmid** is the id returned from **shmget()**
    - **cmd** is the action to take; e.g., **IPC_RMID** for deleting the shared memory segment
    - **buf** points to the structure containing the modes and permissions for the shared memory; it is **NULL** when you try to destroy the shared memory
  - Be aware of destroying the memory segment while someone attaches to it



Concurrency Issues on the Shared Memory

- *Multiple processes modifying the shared memory segment
  - it is possible that certain **errors** could crop up when *updates to the segment occur simultaneously*
  - **It is the responsibility for the programmers to handle the concurrent accesses issues**

- Protecting the shared resources is very important in operating systems
  - *Semaphores* could be used to lock the shared memory segment while a process is writing to it
  - The code, which you want to ensure that a single process/thread has exclusive assess to the resource, is called a *critical section*
  - You can refer to the References to find the related information

**An example code of concurrent access to the shared memory

```
#define TEXT_SZ 2048
struct shared_use_st {
   int written_by_you;
   char some_text[TEXT_SZ];
};

int running = 1;
...
shared_memory = shmat(shmid, (void *)0, 0);
...
shared_stuff = (struct shared_use_st *)shared_memory
shared_stuff->written_by_you = 0;
while(running) {
   if (shared_stuff->written_by_you) {// The producer writes data...
      printf("You wrote: %s", shared_stuff->some_text);
      sleep( rand() % 4 ); /* make the other process wait for us ! */
      shared_stuff->written_by_you = 0;
      if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
         running = 0;
      }
   }
}
...
```

Shared data is protected!!!

- A **pair** of the programs used to handle the concurrent access issue (we only show one of the program pair, consumer, here)
- The data structure **shared_use_st** uses a flag to indicate the memory is being written or not
  - If yes (**shared_stuff->written_by_you == 1**), the other process has to wait until the writing is finished
- If you are interested in this topic, you can read more about it**

Courtesy of Beej's Guide to Unix Interprocess Communication; *Concurrency (computer science), Wikipedia; **Ch. 14 (p. 591), Beginning Linux Programming;



Sockets (Unix Domain Sockets)

Overview

- ***Bi-directional communications***, exchanging data in different *domains*
  - E.g., LAN and Internet are examples of the domains
  - Unix sockets share some of the same interfaces as Internet sockets
- Communications taken place between processes on the same machine or different machines
- Sockets read/write data through the socket interface,
  - which is similar to pipes that read/write data through the file interface
  - Socket descriptors vs. File descriptors

Socket Programming

- The ***client*** and ***server*** programs are involved in the socket programming

Workflow

- The server waits for the connections from the client(s), which is called ***listen***
- Each client tries to ***connect*** to the server
- Once the server ***accepts*** the connection request made by the client, the client and the server are able to exchange data via **send()** and **recv()**
- Shutdown the connection by calling **close()** or **shutdown()**



Server Program Workflow

1. Call to **socket()** to create the socket
   - **int socket(int domain, int sockettype, int protocol);**
     - **domain** is the nature of communication, e.g., AF_INET for IPv4 Internet domain, and AF_UNIX for UNIX domain
     - **sockettype** is the communication characteristics, e.g., SOCK_STREAM for sequenced, reliable, bi-directional, connection-oriented byte streams, and SOCK_SEQPACKET for fixed-length, sequenced, reliable, connection-oriented messages
     - **protocol** is the default protocol for the given domain and socket type
       - Usually set to 0, meaning that TCP is used for SOCK_STREAM socket in the AF_INET
2. Call to **bind()** to link the created socket descriptor **s** with the address **local** in the UNIX domain
   - *Address* is a special file on the system **"/home/beej/mysocket"**
3. Instruct the socket to **listen()** for incoming connections
   - The second argument, **5**, is the number of incoming connections that can be queued before accept() is called

An example code of socket server program (source)

```
struct sockaddr_un {
    unsigned short sun_family; /* AF_UNIX */
    char sun_path[108];
}
unsigned int s, s2;
struct sockaddr_un local, remote;
int len;
s = socket(AF_UNIX, SOCK_STREAM, 0);

local.sun_family = AF_UNIX; /* local is declared before socket() ^ */
strcpy(local.sun_path, "/home/beej/mysocket");
unlink(local.sun_path);
len = strlen(local.sun_path) + sizeof(local.sun_family);
bind(s, (struct sockaddr *)&local, len);
listen(s, 5);
```

Server Program Workflow (Cont'd)



4. accept() a connection from a client create the socket
   - This function returns a new socket descriptor **s2** for further data exchanges via send() and recv()
   - The original descriptor **s** is used for listening

5. Handle the connection via send() and recv(), and loop back to accept()
   - Actually, it communicates with the client via send() and recv(), close() the connection, and accept() a new connection (what are done in the *for-loop*)

- Check the socket man page for details
  - Definitions of other sockets functions can be found within the socket man page

An example code of socket server program (source)

```
struct sockaddr_un {
   unsigned short sun_family; /* AF_UNIX */
   char sun_path[108];
}
unsigned int s, s2;
struct sockaddr_un local, remote;
int len;
char str[100];

s = socket(AF_UNIX, SOCK_STREAM, 0);

local.sun_family = AF_UNIX; /* local is declared before socket() ^ */
strcpy(local.sun_path, "/home/beej/mysocket");
unlink(local.sun_path);
len = strlen(local.sun_path) + sizeof(local.sun_family);
bind(s, (struct sockaddr *)&local, len);
listen(s, 5);

for(;;) {
   len = sizeof(struct sockaddr_un);
   s2 = accept(s, &remote, &len);
   printf("Connected.\n");

   done = 0;
   do {
      n = recv(s2, str, 100, 0);
      if (!done)
         if (send(s2, str, n, 0) < 0) {
            perror("send");
            done = 1;
         }
   }while(!done);
   close(s2);
}
```



Client Program Workflow

- It is easier for the client program

1. Get a UNIX domain socket via **socket()**
2. Contact the server with **connect()**,
   - which links the created socket descriptor **s** with the address **remote** in the UNIX domain; similar to **bind()**
3. Use **send()** and **recv()** to exchange data
4. Call **close()** to terminate the connection **s**

- The function call **socketpair()** helps create a pair of already connected sockets
  - You can use the returned socket descriptor for IPC; check the links below for more information
  - It is programmed like a two-way pipe
- The right figure contains the **error handling codes**,
  - which are important to write the socket programs because I/O operations are not always performed as you expect

An example code of socket client program (source)

```
int s, t, len;
struct sockaddr_un remote;
char str[100];

if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
}
printf("Trying to connect...\n");

remote.sun_family = AF_UNIX;
strcpy(remote.sun_path, SOCK_PATH);
len = strlen(remote.sun_path) + sizeof(remote.sun_family);
if (connect(s, (struct sockaddr *)&remote, len) == -1) {
    perror("connect");
    exit(1);
}
printf("Connected.\n");
while(printf("> "), fgets(str, 100, stdin), !feof(stdin)) {
    if (send(s, str, strlen(str), 0) == -1) {
        perror("send");
        exit(1);
    }
    if ((t=recv(s, str, 100, 0)) > 0) {
        str[t] = '\0';
        printf("echo> %s", str);
    } else {
        if (t < 0) perror("recv");
        else printf("Server closed connection\n");
        exit(1);
    }
}
close(s);
```



Command Line Tools for IPC

- **ipcs**: List any of your created IPC objects on the system
- Actually, the command shows the objects for System V IPC
  - Including shared memory, message queue, and semaphore
  - List the specific type of the objects with the flag, **-m** (shared memory), **-q** (message queue), **-s** (semaphore)
- **ipcrm**: destroy the IPC objects
- Remove the specific type of the objects with the ***flag*** and the ***id***
  - flag: **-m** (shared memory), **-q** (message queue), **-s** (semaphore)
  - id: the **shmid** of created object

```
[ysmoon@cs1 16]$ gcc -o shm_wr shm_wr.c
[ysmoon@cs1 16]$ gcc -o shm_rd shm_rd.c
[ysmoon@cs1 16]$
[ysmoon@cs1 16]$ shm_wr
(1.4140), (3.1420), (1.7710)
[ysmoon@cs1 16]$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0xf1010101 131072     ysmoon     666        12         0

[ysmoon@cs1 16]$ shm_rd
(1.4140), (3.1420), (1.7710)
[ysmoon@cs1 16]$ shm_rd
(1.4140), (3.1420), (1.7710)
[ysmoon@cs1 16]$
[ysmoon@cs1 16]$ ipcrm -m 131072
[ysmoon@cs1 16]$
[ysmoon@cs1 16]$ shm_rd
세그멘테이션 오류 (core dumped)
[ysmoon@cs1 16]$
```



References

Books

- Beginning Linux Programming, 4th Edition, Neil Matthew, Richard Stones, Wiley, 2007 (ISBN-13: 978-0-470-14762-7)
- Understanding the Linux Kernel, 3rd Edition, Daniel Pierre Bovet and Marco Cesati, O'Reilly Media, 2005 (ISBN-13: 978-0596005658)
- Advanced Programming in the UNIX Environment, 3rd Edition, W. Richard Stevens, Stephen A. Rago, Addison-Wesley Professional, 2013 (ISBN-13: 978-0321637734)

Courses

- CS550 - Operating Systems, Kartik Gopalan, SUNY Binghamton
- COMP 790: OS Implementation, Don Porter, UNC Chapel Hill

Online resources

- Beej's Guide to Unix Interprocess Communication
- Beej's Guide to Network Programming



QUESTIONS?