MONASH University

Information Technology

FIT2100 Operating Systems

INTER PROCESS COMMUNICATION - PART 2

Week 11
Semester 2 2024
(Reading: Tanenbaum: Chapter 2 and Stallings: Chapter 7, 8)

Dr Charith Jayasekara
Faculty of Information Technology

OUTLINE

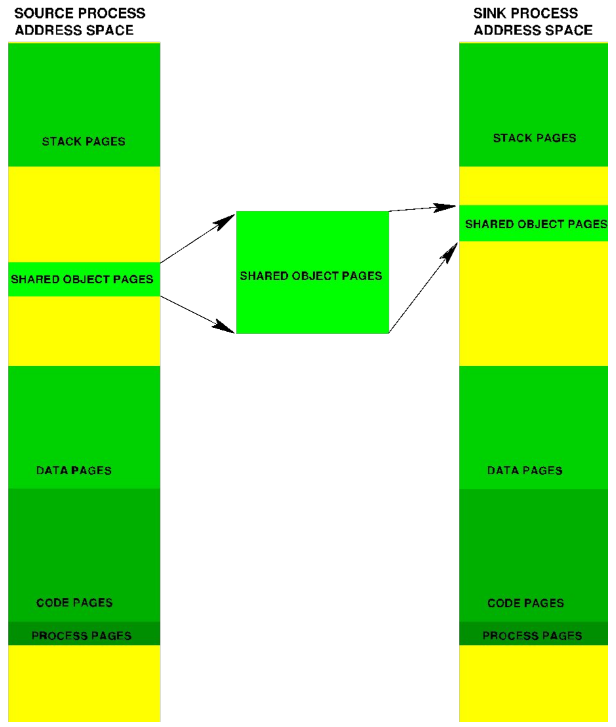
- Inter Process Communication (IPC)
- Message Queues
- Concurrency Implications
- Signals
- Shared Memory
- Stream-based IPC: Unnamed and Named Pipes
- Sockets

MONASH University

LEARNING OUTCOMES

- Upon the completion of Week 11, you should be able to:
  - Discuss various **mechanisms** for interprocess communication (IPC).
  - Understand the **implications** of different IPC mechanisms for **synchronization** and **concurrency**.

SHARED MEMORY (SHM)



SHARED MEMORY (SHM)

WHAT IS SHARED MEMORY

- A **segment in virtual memory** that is allocated to multiple processes
  - A memory segment **may be shared** among multiple processes
  - Like with multi-threading, **multiple processes** can access each other's memory
  - It does not happen automatically like with multi-threading
    - Must **manually** request a new block of shared memory to be allocated.
    - Required **number of bytes** must be **specified**.
    - Must manually assign a **data structure** (e.g. **struct** or **array**) to the shared block of bytes.
  - There is **no concurrency protection** offered. Mutual exclusion mechanisms like semaphores must be used.
  - Using multiple processes with shared memory can be **more robust** than using multiple threads. e.g. if one process crashes, the parent process can restart it.
    - e.g. tabs in Firefox web browser and Google Chrome

MONASH University

SHARED MEMORY (SHM)
TWO WAYS TO USE SHARED MEMORY

- ❒ Linux supports two traditional unix approaches

- ❒ SYSTEM V STYLE (FROM OLD SVR4 UNIX SYSTEMS)
  - Library functions in `<sys/shm.h>` include `shmget(…)` and `shmat(…)`
  - Processes are required to have knowledge of a common **key** value in order to get access to the same segment.
  - The key value is an integer that might be stored in a file for other processes to look up, or simply generated prior to forking a child process.

- ❒ POSIX STYLE (NEWER 'STANDARD' APPROACH)
  - Library functions in `<sys/mman.h>` include `shm_open(…)` and `mmap(…)`
  - A **filename** in the filesystem is associated with a shared memory segment.
  - When opened, the **file descriptor** can be read or written to just like a file.
  - Can also be mapped to a logical memory address using `mmap(…)`

MONASH University

SHARED MEMORY (SHM)

SYSTEM V VS POSIX

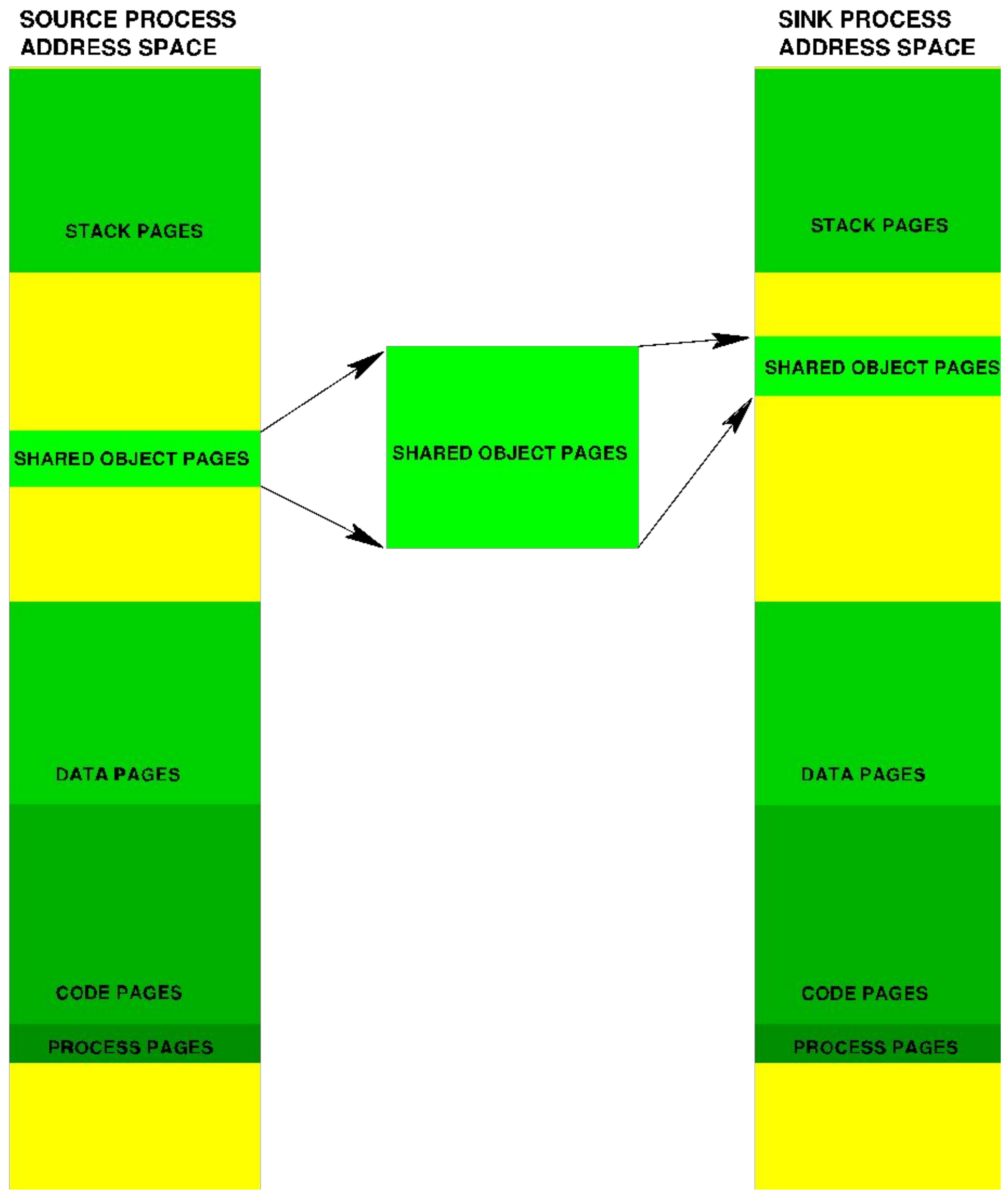
BEWARE!

- For either shared memory approach, there is **NO** built-in concurrency protection
    - Shared memory has the same **hazards** as multithreading.
- Mutual exclusion mechanisms such as **semaphores**, **mutexes**, etc. must be used when accessing shared data.
- These mutual exclusion resources can be stored in the shared memory itself.

SHARED MEMORY (SHM)

WHAT IS SHARED MEMORY

- Example shows a shared object between two processes, using shared memory
- The MMU is configured to map the block of physical memory of the specified size into respective virtual addresses in both processes
- Typical SHM APIs permit shared objects to be read, written, or both read and written
- Because access is controlled by the VM system and its MMU, security is usually good
- Bandwidth between processes is the bandwidth to main memory in the host platform – ***SHM is FAST!***



MONASH University

SHARED MEMORY (SHM)

ADVANTAGES AND DISADVANTAGES OF SHM

- **Advantage**: Shared memory is simple to implement (if hardware permits)
- **Advantage**: Shared memory has exceptional bandwidth
- **Advantage**: Shared memory permits concurrent IPC with many processes

- **Disadvantage**: Shared memory has no structure - burden on programmer
- **Disadvantage**: Shared memory APIs may differ between OS types

SHARED MEMORY (SHM)

EXAMPLE: UNIX SHARED MEMORY (Curry, 2014, p134)

- Shared memory is described *shared memory id*, a segment is described by a `shmid_ds` structure declared in `sys/shm.h` and `sys/types.h`:

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* permissions          */
    int    shm_segsz;         /* size of seg          */
    sde_t  shm_seg;           /* seg descriptor       */
    ushort shm_lpid;          /* last shmop           */
    ushort shm_cpid;          /* pid of creator       */
    ushort shm_nattch;        /* cur # attached       */
    ushort shm_cnattch;       /* # in mem attached */
    time_t shm_atime;         /* last shmat time      */
    time_t shm_dtime;         /* last shmdt time      */
    time_t shm_ctime;         /* last chg time        */
};
```

- To create or access a segment, you must execute a `shmget` system call
- To get or modify the attributes of an existing segment, you must execute a `shmctl` system call
- To use a segment, you must execute a `shmat` system call
- To detach from a segment, you must execute a `shmdt` system call

MONASH University

SHARED MEMORY (SHM)

EXAMPLE: UNIX SHARED MEMORY (Curry, 2014, p134)

- Curry provides a simple server program to demonstrate shared memory operations:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27
main()
{
    char c;
    int shmid;
    key_t key;
    char *shmat();
    char *shm, *s;
    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;
    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
```

MONASH University

SHARED MEMORY (SHM)

EXAMPLE: UNIX SHARED MEMORY (Curry, 2014, p134)

```
    /*
     * Now we attach the segment to our data space.
     */
    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;
    /*
     * Finally, we wait until the other process
     * changes the first character of our memory
     * to '*', indicating that it has read what
     * we put there.
     */
    while (*shm != '*')
        sleep(1);
    exit(0);
}
```

- ❑ Curry also provides a simple client program to demonstrate shared memory operations (next page)

SHARED MEMORY (SHM)

EXAMPLE: UNIX SHARED MEMORY (Curry, 2014, p134)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27
main()
{
    int shmid;
    key_t key;
    char *shmat();
    char *shm, *s;
    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;
/*
     * Locate the segment.
     */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
```

SHARED MEMORY (SHM)

EXAMPLE: UNIX SHARED MEMORY (Curry, 2014, p134)

```
/*
     * Now we attach the segment to our data space.
     */
    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');
    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';
    exit(0);
}
```

- End of example

STREAM-BASED IPC:
UNNAMED PIPES AND FIFOS

STREAM-BASED IPC

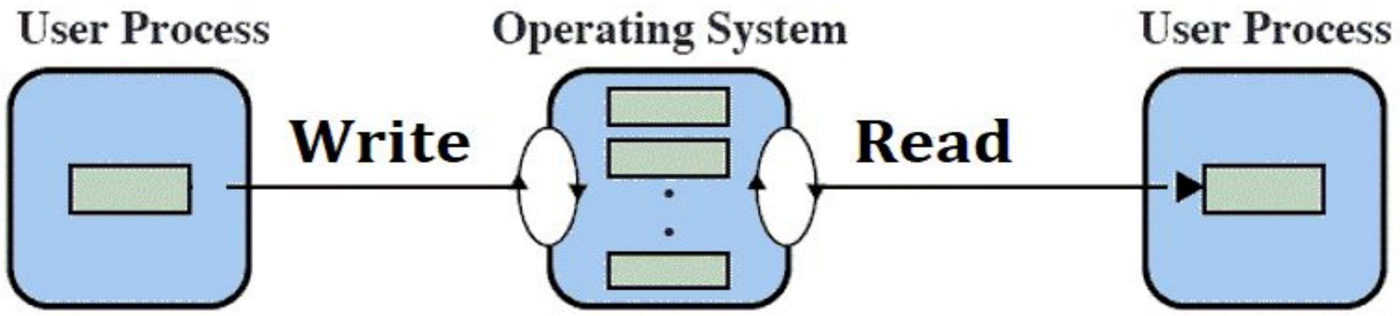
WHAT ARE PIPES?

- Bytes go in one end, come out at the other end
- A **buffer** maintained within the kernel
  - A pipe is a resource where information written in at one ‘end’ can be read out at the other ‘end’.
  - Slower than SHM since all data must pass through the kernel.
  - A pipe is a bit like a file but with **two** file descriptors. One file descriptor is for reading only, while the other is for writing only.
- A pipe **enforces** synchronisation between reader and writer
  - As long as only one process reads, and only one process writes to pipe, concurrency protection is guaranteed.
    - ➢ First-in-first-out behavior
    - ➢ The reader cannot read past what has been written, and the writer cannot overwrite data in the pipe before it has been read
    - ➢ The kernel may implement a **circular buffer** to make this happen.

MONASH University

STREAM-BASED IPC
CIRCULAR BUFFERS

- A circular buffer is a **set** of buffers accessed in a **‘circular’** way
  - The stream of bytes spans multiple buffers (character arrays)
  - At any one time, one buffer is selected for reading while another is being written to.
  - **Writer** fills buffers while the **reader** empties them, before moving on to the next buffer in the circle.
  - If the reader ‘catches up’ to the writer, further reads will **block** until the next buffer becomes available (has been written to).
  - If the writer **catches up** to the reader, further writes will **block** until the next buffer becomes available for writing (has been completely read).



MONASH University

STREAM-BASED IPC

TWO KINDS OF PIPES: UNNAMED AND NAMED

❒ UNNAMED PIPES

- Can only be shared between **related** processes.
- Created using the **`pipe`** system call.
  - ➢ Creates a **pipe** and allocates **two file descriptors**.
  - ➢ After a **`fork`**, the child will inherit the parent's open file descriptors.
  - ➢ So one process can write to the pipe to talk to the other process.

❒ NAMED PIPES (also known as FIFOs)

- A special kind of pipe with a **filename** in the filesystem.
- Can be used between **unrelated** processes.
- A special 'file' created using the **`mkfifo`** system call.
- One process then opens the 'file' read-only, while the other process opens the same file 'write-only'.

PIPES

ADVANTAGES AND DISADVANTAGES OF PIPES

- **Advantage**: Pipes can be simple to implement
- **Advantage**: Pipes usually have good or excellent bandwidth (depends on OS)

- **Disadvantage**: Pipes have no structure - burden on programmer
- **Disadvantage**: Pipes force FIFO discipline upon IPC
- **Disadvantage**: Pipe APIs may differ between OS types

PIPES

EXAMPLE: UNIX PIPES (Curry, 2014, p100)

- Pipes can be accessed in *nix systems using the shell, but also using system calls (the shell syntax is often a wrapper for using the system calls from the command line)
- *The system call to create a pipe is named* `pipe`, called with an array of two integers that are the file descriptors for the created *pipe*.
- Cite (Curry, 2014): *"After creating the pipe, a program should spawn a child process. The parent reads data from the child on the first descriptor, and writes data to the child on the second descriptor. Similarly, the child reads data from the parent on the first descriptor, and writes data to the parent on the second descriptor. It is common for the child to have its standard input and standard output connected to the first and second descriptors, respectively."*
- Curry provides a simple program to demonstrate pipe operations (next page)

PIPES

EXAMPLE: UNIX PIPES (Curry, 2014, p100)

```
#include <stdio.h>
main()
{
    FILE *fp;
    int pid, pipefds[2];
    char *username, *getlogin();
    /*
     * Get the user's name.
     */
    if ((username = getlogin()) == NULL) {
        fprintf(stderr, "Who are you?\n");
        exit(1);
    }
    /*
     * Create the pipe.  This has to be done
     * BEFORE the fork.
     */
    if (pipe(pipefds) < 0) {
        perror("pipe");
        exit(1);
    }
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }
```

PIPES

EXAMPLE: UNIX PIPES (Curry, 2014, p100)

```
/*
 * The child process executes the stuff inside
 * the if.
 */
if (pid == 0) {
    /*
     * Make the read side of the pipe our
     * standard input.
     */
    close(0);
    dup(pipefds[0]);
    close(pipefds[0]);
    /*
     * Close the write side of the pipe;
     * we'll let our output go to the screen.
     */
    close(pipefds[1]);
    /*
     * Execute the command "mail username".
     */
    execl("/bin/mail", "mail", username, 0);
    perror("exec");
    exit(1);
}
```

PIPES

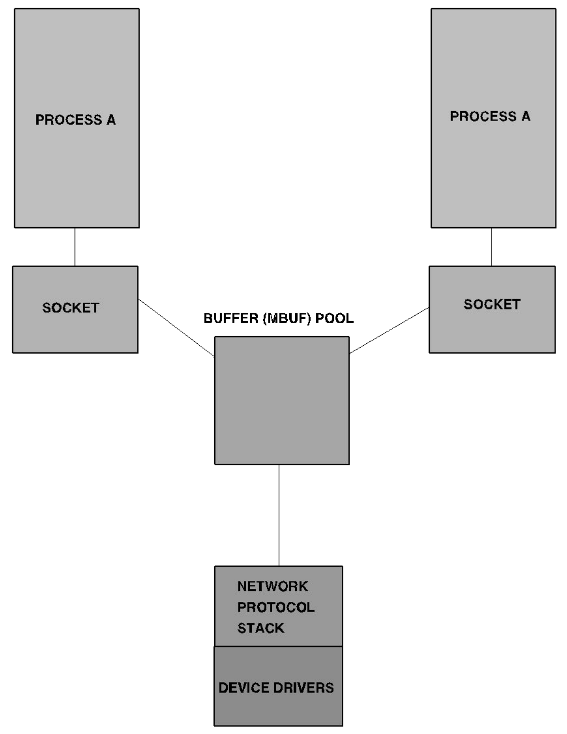
EXAMPLE: UNIX PIPES (Curry, 2014, p100)

```
/*
     * The parent executes this code.
     */
    /*
     * Close the read side of the pipe; we
     * don't need it (and the child is not
     * writing on the pipe anyway).
     */
    close(pipefds[0]);
    /*
     * Convert the write side of the pipe to stdio.
     */
    fp = fdopen(pipefds[1], "w");
    /*
     * Send a message, close the pipe.
     */
    fprintf(fp, "Hello from your program.\n");
    fclose(fp);
    /*
     * Wait for the process to terminate.
     */
    while (wait((int *) 0) != pid)
        ;
    exit(0);
}
```

- End of example

MONASH University

STREAM ORIENTED IPC



STREAM ORIENTED IPC IN CONTEXT

- ❑ As noted, IPC exists in many forms, but can be broadly divided into four categories:
  1. *Signals that employ discrete and limited message formats between processes*
  2. *Shared Memory IPC that relies on the Virtual Memory system to map pages or segments between processes;*
  3. *Message Passing IPC in which the kernel carries discrete messages (or signals) between processes;*
  4. *Stream Oriented IPC in which the kernel provides a stream channel between two processes with FIFO properties;*
- ❑ Stream Oriented IPC is the model and the programming abstraction mostly employed in networked applications;
- ❑ Aside from pipes, the two most commonly used stream oriented APIs are BSD Sockets and SVR4 STREAMs - pipes can be implemented as a wrapper around a stream IPC scheme like a Socket
- ❑ Sockets have become used most frequently as they provide a uniform mechanism for stream oriented IPC between processes on a single host, or processes on different hosts connected by a network, providing immense flexibility and portability

MONASH University

SVR4 STREAMS vs BSD Sockets (Kopp, 1996)

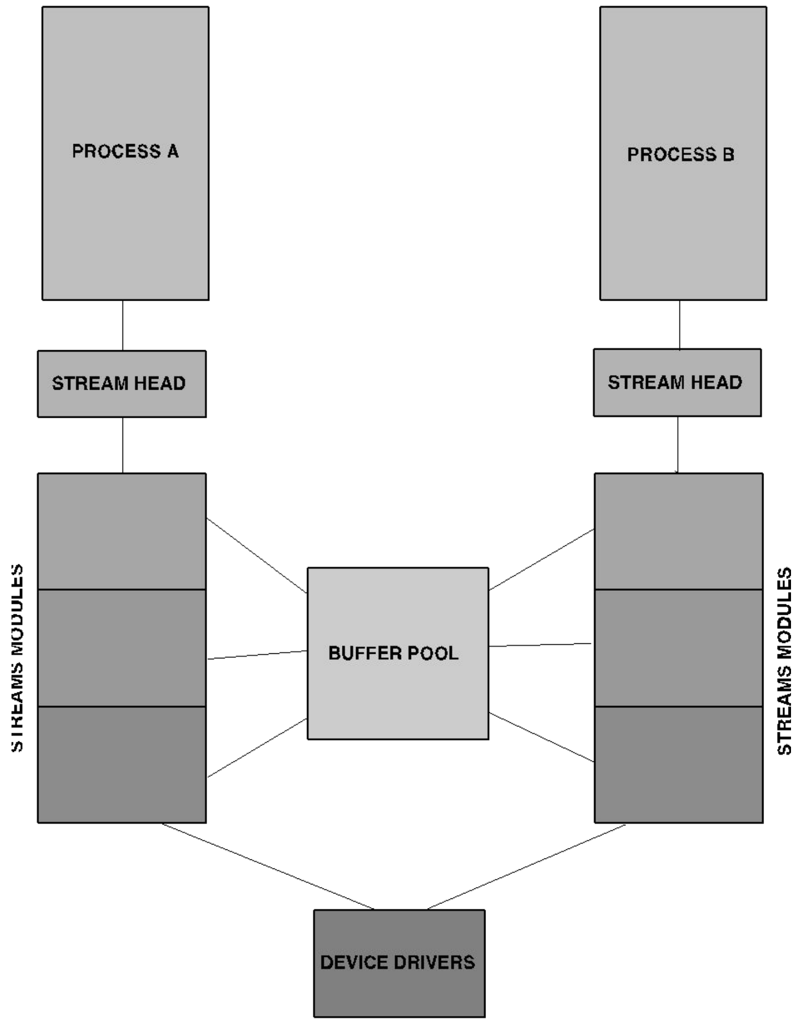


Fig.2.2 STREAMS Transport - Conceptual Model

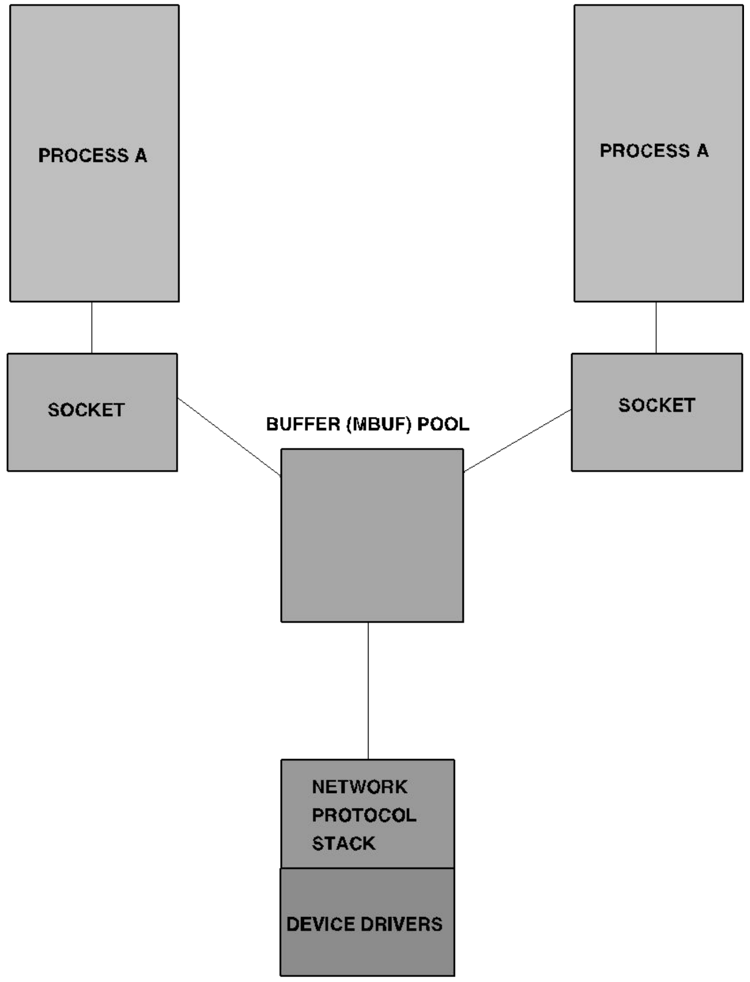


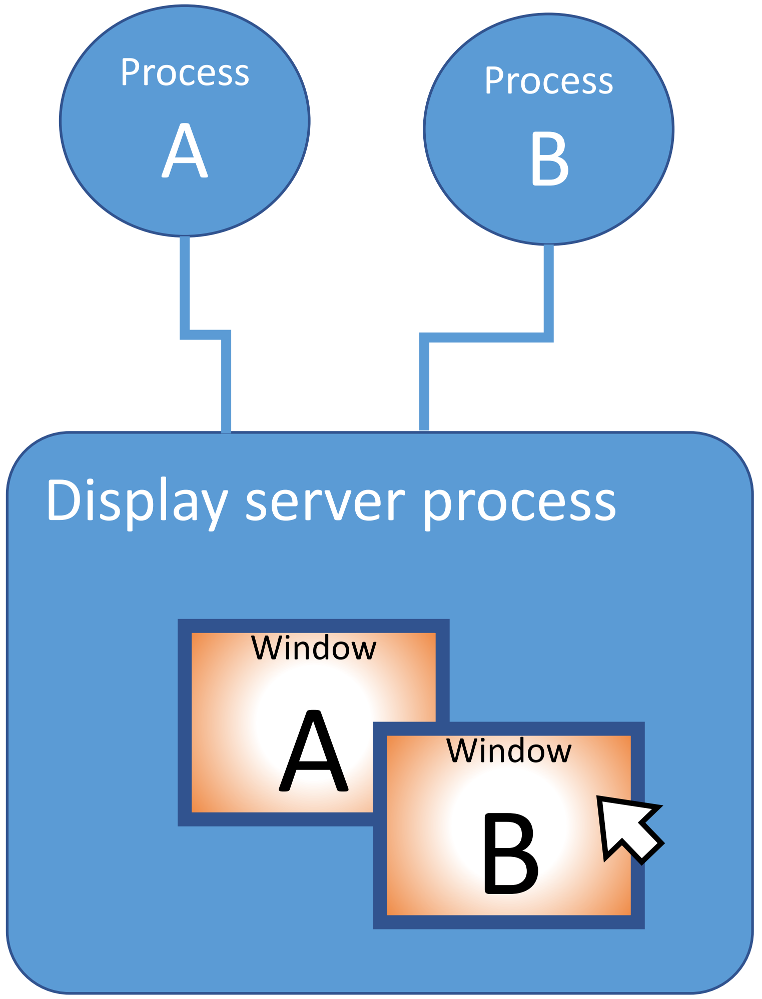
Fig 2.4 BSD Socket Transport - Conceptual Model

SOCKETS

SOCKETS

EXAMPLE: DISPLAY SERVER UTILITY

- Only **one** process can control the display hardware at a time
- How does a desktop system like ***Xorg*** work?
  - One utility called the **display server** takes responsibility for drawing all graphics on the screen.
  - Applications (**clients**) connect to the display server and send instructions for what should be drawn on the screen. (Through an API.)
  - The display server interprets instructions from different applications to produce a montage of **windows** on the screen.
  - When the user interacts with the mouse and keyboard, the display server figures out which **client** to send the information back to.



If all processes were allowed to write to the display directly, the result would be a mess!

MONASH University

SOCKETS

SOCKETS IN LINUX

- ‘UNIX DOMAIN SOCKETS’
- How is a socket connection created?
  1. The server process creates a `socket`, which is mapped to a **filename** in the filesystem.
  2. Client processes must `connect` to this file to reach the server.
  3. The server receives information about a new connection request, and chooses to `accept` the connection.
  4. When accepted, the OS provides an open file descriptor resource to both the client and server process.
- Just like a pipe, a socket connection enforces **FIFO behaviour** when it is being read and written to concurrently.
- Unlike a pipe, a socket connection can send data in both directions.
  - The kernel provides **two** circular buffers. One for bytes sent from the client to the server, and another for bytes sent from the server to the client.

MONASH University

BSD Socket Application Programming Interface

- Opening a socket connection requires the creation of the socket with a `socket()` system call, binding a socket address to the socket with a bind() call and initiating the connection with a `connect()` call.
- The `socket()` call returns an index into the process file table, termed a file descriptor in Unix/Linux/BSD.
- Once the connection is open, the programmer may use both socket specific calls or the established Unix/Linux/BSD `read()` and `write()` system calls.
- The BSD Socket has become the *de facto* standard low level programming interface for networked IPC, although in most contemporary applications it is hidden below other protocols;
- Nearly all operating systems will provide a BSD socket API, although some will use the newer POSIX standard.

Socket Function Prototypes

- `int socket(int domain, int type, int protocol);`
- Where the domain can be IPV4, IPV6 or UNIX (local to host);
- `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);`
- Where the arguments define the interface;
- `int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);`
- Where `*serv_addr` is the address of the server being connected to;
- Once the stream is established between two processes on two hosts, traffic can be sent or received using `send()`, `recv()`, `write()`, `read()` system calls;
- The socket is shut down using a `close()` call.

PROGRAMMING WITH SOCKETS

Programming with the `socket()` API (I)

- Sockets can be used for IPC between two processes, that reside on one or two hosts, which is useful for debugging code; sockets assume the client-server model, so one process is always a server process, that responds to requests from a client process;
- This requires that the server process be running before the client can access it;
- There are necessary include files for the socket API, that define the API interface types:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Server programming with the `socket()` API (II)

- ❑ The necessary starting point is always to set up a server process, which must set up a receiving socket;
- ❑ Several steps are necessary:

1. Use the `socket()` system call to create a socket;
2. The socket needs an address so the client can find it, and this is done by binding an address to the socket using the `bind()` system call – for the Internet a *port number* is used;
3. To activate the socket so it listens for incoming clients, a `listen()` system call is required;
4. An incoming client connection must be accepted using the `accept()` system call;

- ❑ *Socket programming in Java or other languages typically employs a language specific wrapper layered over the C language system calls!*

Server programming with the `socket()` API (III)

- The next step is always to set up a client process, which must set up a sending socket;
- Several steps are necessary:

1. Use the `socket()` system call to create a socket;
2. The socket needs to be connected to the server process, and this is done by connecting to the server socket using the `connect()` system call – for the Internet a host *IP address* and *port number* must be used;
3. Once the connection has been accepted by the server, data can be sent and received using `read()` and `write()` system calls;

- Socket programming in C language is the simplest means available for IPC over network connections, but also exposes all of the details of the connection;

Domain parameters for the `socket()` API

- The `int socket(int domain, int type, int protocol)` call has three parameters (example MacOSX 10.10), `domain`:

1. `PF_LOCAL` Host-internal protocols, formerly called `PF_UNIX`,
2. `PF_UNIX` Host-internal protocols, deprecated, use `PF_LOCAL`,
3. `PF_INET` Internet version 4 protocols,
4. `PF_ROUTE` Internal Routing protocol,
5. `PF_KEY` Internal key-management function,
6. `PF_INET6` Internet version 6 protocols,
7. `PF_SYSTEM` System domain,
8. `PF_NDRV` Raw access to network device

- NB local host `PF_LOCAL/PF_UNIX` versus network host `PF_INET/PF_INET6!`

Type parameters for the `socket()` API

- The `int socket(int domain, int type, int protocol)` call has some important parameters (example MacOSX 10.10);
- Defines for the `int type` argument:

1. `SOCK_STREAM` sequenced, reliable byte streams (TCP)
2. `SOCK_DGRAM` connectionless, unreliable (UDP)
3. `SOCK_RAW` internal network protocols and interfaces
4. `SOCK_SEQPACKET` sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length
5. `SOCK_RDMNB` not implemented

- NB the `int protocol` argument is usually by default set to 0, although some systems may support a range of protocols.

Caveats for the `socket()` API

- The `socket()` API may be the simplest API for providing stream (TCP) or datagram (UDP) IPC, but it also requires that the programmer understand choices in protocols as well as the port addressing scheme;
- The programmer has to manage the state of the connection, and handle all of the errors;
- The API permits seamless use of socket connections local to the host (if Unix/BSD/Linux OS), or to remote hosts;
- The socket appears to the programmer as an `int sockfd` file descriptor, which is one layer of abstraction below the `file' used by the libc stdio *nix / C language abstraction;
- The `fwrite()` and `fread()` *stdio* calls are wrappers for the `int write()` and `int read()` *nix/POSIX system calls, and operate on a `FILE` struct that presents as a file;

Example Client-Server – HTTP over Sockets

- ❑ HTTP (Hypertext Transfer Protocol) is the most widely used protocol on the W3 and is a good example of a protocol built on top of the BSD Socket API;
- ❑ When a browser (client) intends to make a request of a web server (server), it opens a socket connection over the Internet to the web server;
- ❑ The browser then sends a HTTP Method message to the web server, for instance:
- ❑ `GET /mypath/to/myfile/blogs.html HTTP/1.0`
- ❑ The socket connection is then closed, while the server processes the method request;
- ❑ Once processing is complete, the web server opens a socket connection to the client, and responds with a message, header and MIME encoded body:
  `HTTP/1.0 404 Not Found`

Client-Server – HTTP over Sockets [200 OK]

- HTTP Response: `HTTP/1.0 200 OK`
- HTTP Header: `Last-Modified: Tue, 12 Jul 2011 21:59:59 GMT`
- HTTP Body:

```
Content-Type: text/html
Content-Length: 512
<!DOCTYPE doctype PUBLIC "-//w3c//dtd html 4.0
  transitional//en"> <html> <head> <meta
  http-equiv="Content-Type" content="text/html;
  charset=UTF-8"> <meta name="Author" content="Carlo
  Kopp"> <meta name="GENERATOR" content="Mozilla/4.55
  [en] (X11; U; Linux 2.4.2 i386) [Netscape]">
  <title>Carlo Kopp's Homepage</title> </head> <body
```

Client-Server – HTTP over Sockets

```
… More page content …
<center><!-- E N D T R A I L E R T A B L E --></center>
  </span><!-- M4 Macro --> </body> </html>
```

- Once the Body is transferred, the socket connection is then closed;
- HTTP is widely used to support other mechanisms used in distributed computing;
- Secure HTTP (SHTTP) employs a more complex connection mechanism due to the use of TLS or SSL encryption layers;
- As HTTP lacks mechanisms to handle multiple servers concurrently, it is a good example of a basic client server protocol.

Example: Linux Socket Integration [Stallings Ch.17]

INET Domain Socket running over a TCP/IP Network

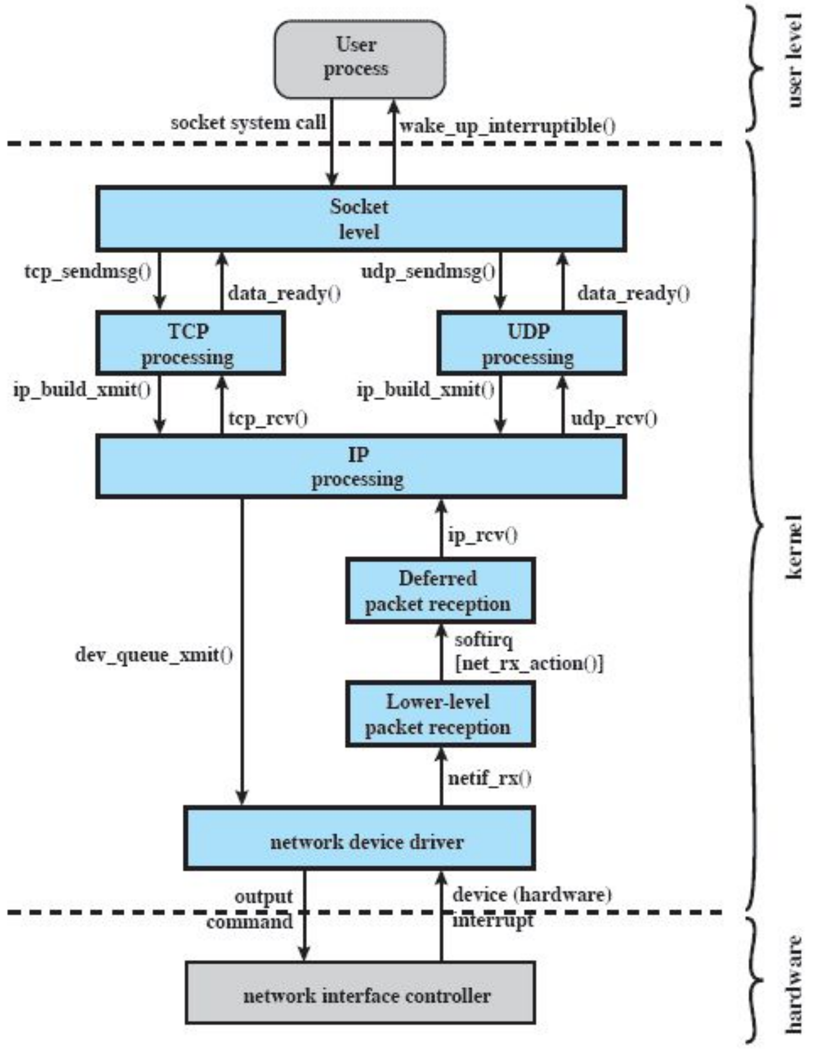


Figure 17.7 Linux Kernel Components for TCP/IP Processing

Summary

- So far we have discussed
  - Several mechanisms for interprocess communication.
    - Signals are the most primitive
    - Other approaches include pipes, sockets and message queues
  - The implications for synchronisation and mutual exclusion, of using IPC tools in an environment where concurrent processes share data.
  - Several mechanisms for interprocess communication.
  - The implications for synchronisation and mutual exclusion, of using IPC tools in an environment where concurrent processes share data.
- Next week
  - OS Security
- Reading
  - Stallings (7th Edition):
    - Chapter 5 sections 5.5, 5.6.
    - Chapter 6 sections 6.7, 6.8.
  - Further reading: Curry, Unix Systems Programming for SVR4, Chapter 13 – IPC, Chapter 10 – Signals
  - Further reading: Curry, Using C on the UNIX System, Chapters 9, 11