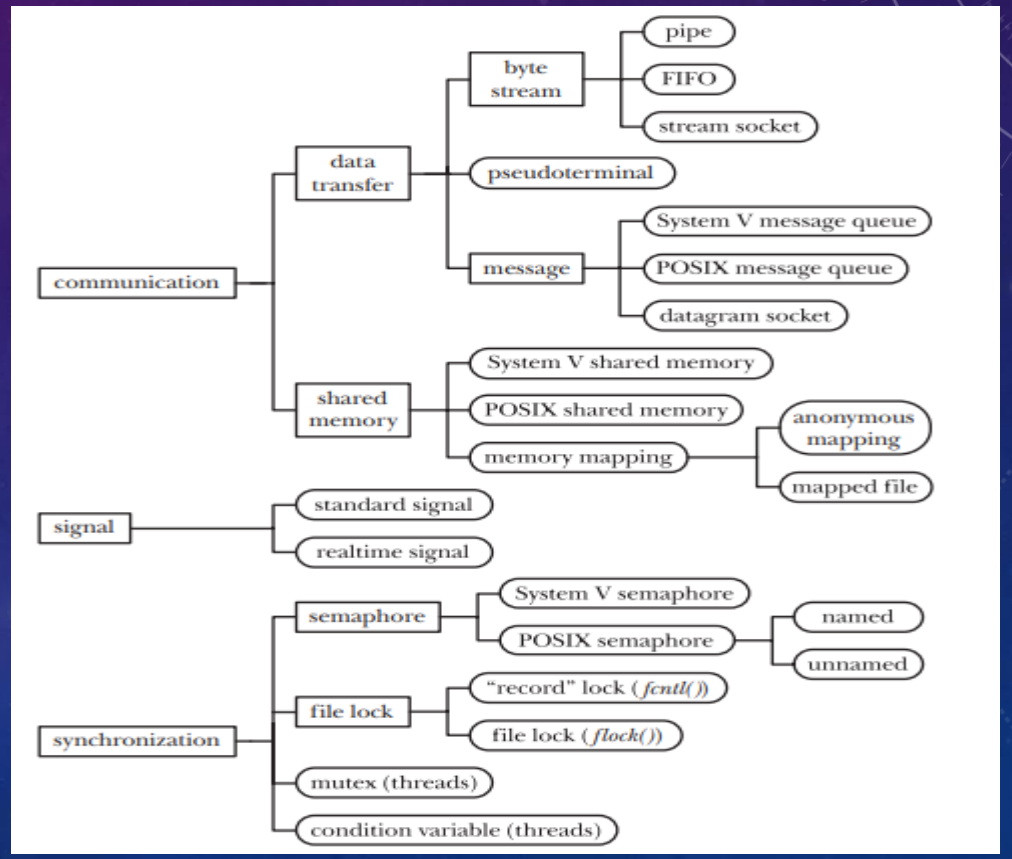
OVERVIEW IPC IN LINUX

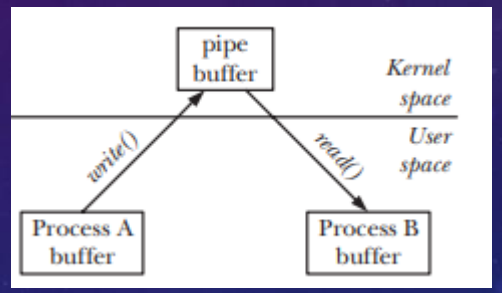
IPC CATAGORIES

- IPC is a method to exchange information between processes and threads
  - Communication – exchange data
  - Synchronization – synchronizing the actions of process
  - Signal – Sometimes use to Synchronization or Communication



COMMUNICATION ISSUE

- Various way to communication between 2 processes. There are 2 main categories
  - Data transfer
    - Writing/Reading – One process write data from user memory to kernel memory until other read (copy) data from kernel memory to user memory
  - Shared memory
    - Allow process to exchange information by placing it in memory region that shared between two processes
    - Kernel accomplish this by making page-table entries in each process point to the same page of RAM
    - Because communication doesn't require system call or copy data between kernel space and user space, shared memory can provide very fast communication



DATA TRANSFER

- There are 3 categories
  - Byte stream
    - Sequence of byte, reader can read any number of bytes regardless the size of block written by the writer
    - Pipe, FIFO, TCP socket
  - Message
    - An read operation read whole message with fixed size. It is not possible read a part of a message or read multiple message in one operation
    - POSIX message queue, UDP socket
  - Pseudoterminal
    - In special case one process use terminal other process like ssh

- Some additional features
  - Read operation can involved by multi-reader, but reads are destructive, one reader consume data, data become not available to all remaining reader
  - Synchronization between the reader and the writer processes is atomic. If reader try to read data that writer is not yet fished his job, the reader block until data available

SHARED MEMORY

- Shared memory
  - POSIX shared memory and memory mapping
  - Shared memory need to synchronization between provider(writer) and consumer(reader). Semaphore is introduced to take the responsibilities
  - Data place to shared memory is visible to all readers

- Synchronization
  - Semaphores
    - Is kernel maintain integer whose values is never permitted to fall below 0
    - A process want to access shared data, it try to decrease the integer
      - 1->0 process have right to access data
      - 0->-1 process block until other process increase integer
    - A process want leave shared data area, it try to increase the integer
      - 0->1 blocked process wake up and it can access shared data area
      - 1 -> it ok
  - File lock
    - Synchronization method explicitly designed to coordinate action of processes that operating the same file
  - File lock

COMPARING IPC FACILITIES

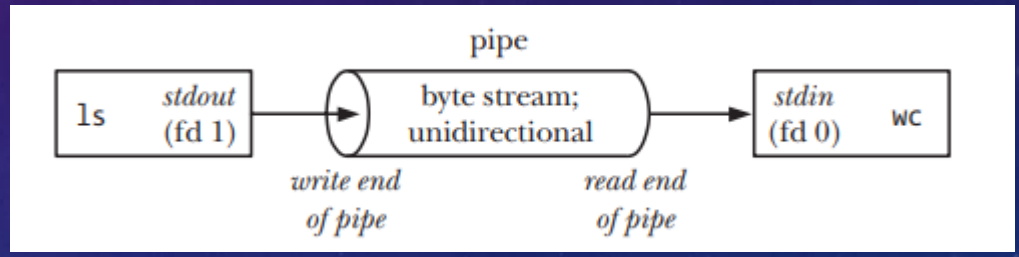
- Properties of various IPC

Facility type	Name used to identify object	Handle used to refer to object in programs
Pipe	no name	file descriptor
FIFO	pathname	file descriptor
UNIX domain socket	pathname	file descriptor
Internet domain socket	IP address + port number	file descriptor
System V message queue	System V IPC key	System V IPC identifier
System V semaphore	System V IPC key	System V IPC identifier
System V shared memory	System V IPC key	System V IPC identifier
POSIX message queue	POSIX IPC pathname	*mqd_t* (message queue descriptor)
POSIX named semaphore	POSIX IPC pathname	*sem_t* * (semaphore pointer)
POSIX unnamed semaphore	no name	*sem_t* * (semaphore pointer)
POSIX shared memory	POSIX IPC pathname	file descriptor
Anonymous mapping	no name	none
Memory-mapped file	pathname	file descriptor
flock() lock	pathname	file descriptor
fcntl() lock	pathname	file descriptor

PIPES

- Shell create two process and execute ls and wc
  - Pipe create to send output of ls to wc
  - Pipe allow data flow from one process to other
  - Writing process (ls) write data to its standard output (fd 1) and reading process (wc) read data from standard input (fd 0) and pipe jointed to them until two process unaware the existence of pipe

```
$ ls | wc -l
```



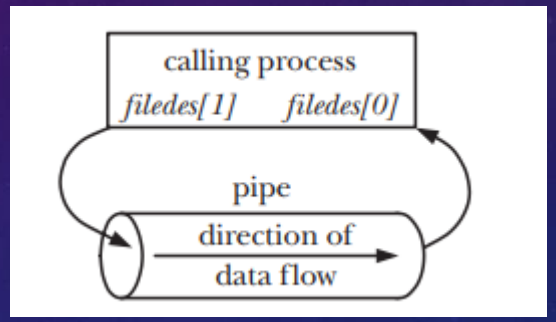
PIPE PROPERTIES

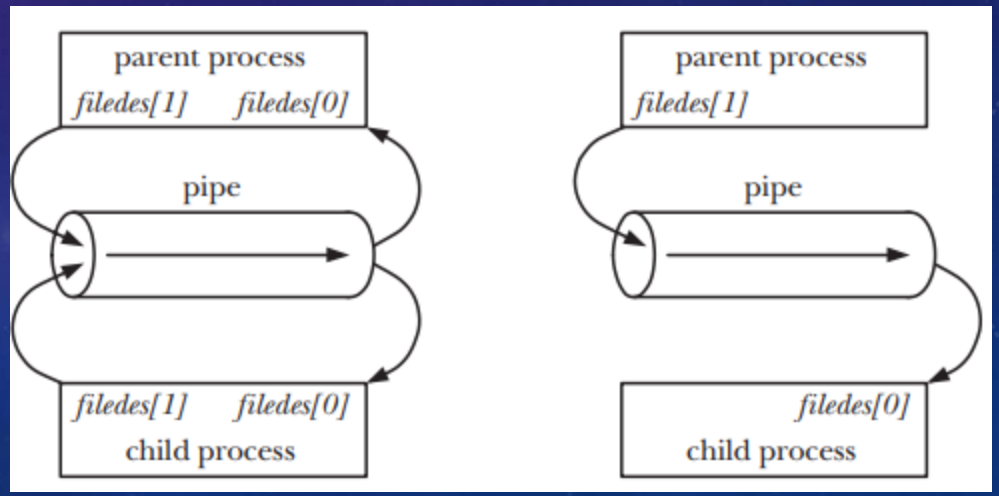
- Pipe is a byte stream
  - No have concept of message or message boundaries when using pipe
  - Reading process read block of data in any size regardless of the size of blocks written by writing process
  - Data passes through the pipe sequentially – byte read from pipe in exactly the order they are written
  - Attempt to read from pipe that currently empty block until at least one byte has been written to pipe
  - If the write end of pipe closed, then a process reading from pipe will see end-of-file (read 0)
- Pipe is unidirectional
  - Data can only travel one direction through pipe
  - One pipe use for writing and other for reading
- Pipe is atomic in writing
  - If multiple process write to pipe, data guarantee still in order if number byte written larger than PIPE_BUF(4096-page size)
- Pipe is atomic in writing
  - Pipe is simply a buffer maintained in kernel memory. This buffer has limit capacity and when pipe full , write to pipe block until the reader removes data from pipe

CREATE AND USE PIPE

- Pipe created
  - filedes[0] for read data from pipe and filedes[1] for write data to pipe
  - Pipe and fork
- Parent and child communicate unidirectional
  - Close unused file descriptors
  - If two processes try to simultaneously read from a pipe, we can't be sure which process will be the first succeed, two process race for data
  - Preventing such races need to synchronization mechanism
  - If multi-processes write to pipe, it it typical to have only a single writer

```
int pipe(int filedes[2]);
```





CLOSING UNUSED PIPE FILE DESCRIPTOR

- Pipe allow communication between related processes
  - It is normally two process communicated each other is parent and child,
  - Two process that have relationship can be used pipe for communication such as process and grandchild or two siblings process

- Closing unused pipe file descriptor
  - Avoid limited file descriptor that a process can open
  - When a reading process close write descriptor, when writing process completes its output and close its write descriptor, reading process read end-of-file → reading process know end of communication and close pipe
  - If reading process is not close write descriptor and other close write descriptor, it isn't seek the end-of-file→ wait data never coming
  - When a process tries to write to a pipe which no process has open read descriptor, kernel send SIGPIPE to process and process terminated

PIPES AS A METHOD OF PROCESS SYNCHRONIZATION

- Example to using pipe
- Use example to explain Pipes as a Method of Process Synchronization

USING PIPES TO CONNECT FILTERS

- Back to ls | wc –l and understand how to implement it
  - Create two processes, one process invoke ls, other invoke wc and directive output of first process to input of second process
  - It is need to guarantee stdin is 0, stdout is 1 before use this technique

```
$ ls | wc -l
```

```
int pfd[2];

pipe(pfd);          /* Allocates (say) file descriptors 3 and 4 for pipe */

/* Other steps here, e.g., fork() */

close(STDOUT_FILENO);           /* Free file descriptor 1 */
dup(pfd[1]);                    /* Duplication uses lowest free file
                                   descriptor, i.e., fd 1 */
```

```
dup2(pfd[1], STDOUT_FILENO);    /* Close descriptor 1, and reopen bound
                                   to write end of pipe */
```

FIFO

- FIFO similar with pipe
- FIFO has name in file system and opened as a regular file
- Used for communication between unrelated process
  - When FIFO open, the IO system call read/write/close can use with FIFO
  - Data read has same order with data written
  - First in, first out
  - When FIFO close, outstanding data will be discarded

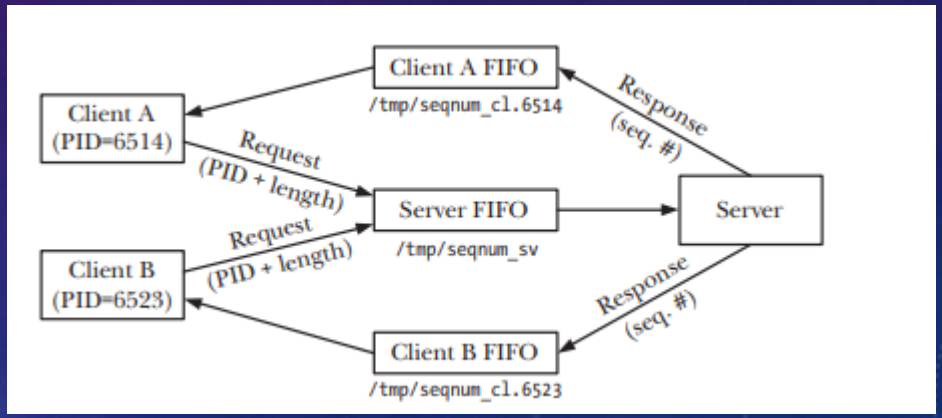
```
int mkfifo(const char *pathname, mode_t mode);
```

- Pathname is path to file on file system
- Mode is permission of file

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

A CLIENT-SERVER APPLICATION USING FIFOS

- Application overview
  - All clients send their request to server using a single server FIFO
    - Clients knows pathname of FIFO that use connect to server before open connection to server
  - Each client has its pathname to server send response back to client
    - Server don’t know which file use to connect to client, it get this information by parsing message from client
    - It is possible to all clients use a unique file for response but multiple clients read FIFO at the same time, it cause race data because it cannot predict which process get data
    - Client build its pathname base on its process ID and sent it to server



- Server requirement
  - Create server well known FIFO and open FIFO for reading
    - Server or client need to running first?
    - FIFO block open for read until open for write
  - Open more than one server for writing to avoid read end-of-file if all clients close the write
  - Ignore SIGPIPE to avoid terminate server
  - If server encounter an error in opening the client FIFO, it abandon client and continue server other request
  - It is a iterative server

- Client requirement
  - Open client FIFO to server can response to it
  - Open well known server FIFO and sending request including its pid
  - Read and print server response