Inter Process Communication

What to Expect?

- W's of Inter Process Communication
- Various IPC mechanisms in Linux
  - Pipes
  - FIFOs
  - Message Queues
  - Shared Memory & Process Semaphores

Why separate IPC mechanisms?

- Processes are non-sharing entities
- But, we may need communication
- Already looked at one mechanism
  - Signals
- But those are very basic
  - Mostly asynchronous
  - Connection-less
  - No data communication
- Separate mechanisms are answer to all those

What is IPC?

- Inter Process Communication
- Mechanism whereby one process communicates with another process
- Communication = Exchange of Data
- Directions
  - Uni-directional
  - Bi-directional
  - Multi-directional

Example for IPC

- Print the filenames in a directory
- 'ls' outputs the filenames
- 'lpr' prints its input
- But ls & lpr are separate processes
- What are the solutions?
  - Put into a file → ls > file; lpr < file
  - Setup an IPC, say a pipe (|) → ls | lpr

Types of IPC mechanisms

- Pipe: Allows the flow of data in one direction only
- Named Pipe: Pipe with a specific name
- Message Queues: Message passing using a queue
- Shared Memory & Semaphore
  - Shared Memory allows the interchange of data through a defined area of memory
  - Semaphore is used in solving problems associated with synchronization and avoiding race conditions during sharing
- Mapped Memory: Similar to shared memory, but use file instead of memory
- Why so many?
  - Based on the requirements, flexibility & benefits

Pipes

W's of a Pipe

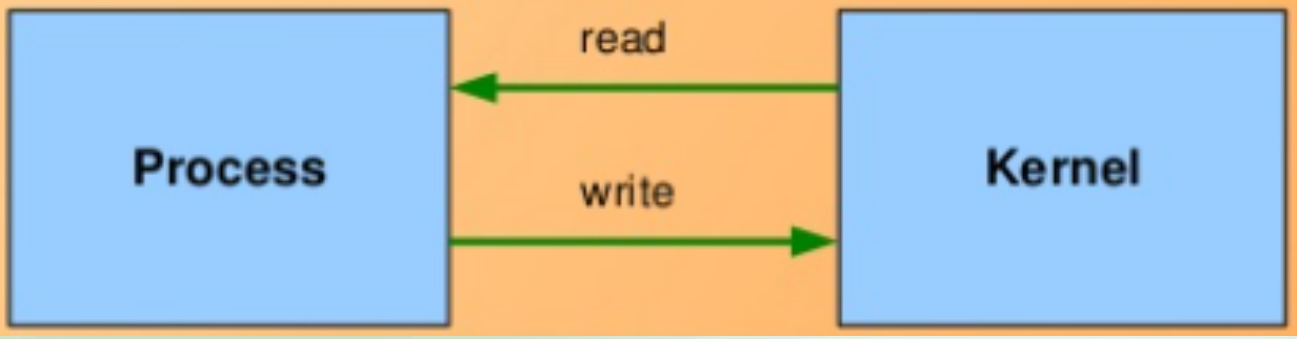
- Connects 2 ends to allow transfer
- One-way Serial Communication Channel
- One end: Output (write end)
- Another end: Input (read end)
- Limited Capacity
- Provides automatic Synchronization
  - Read blocks on no data
  - Write blocks on full data

Pipe Usage

- On Shell
  - Using |
- In Programming / C
  - Creation
    - Using popen(), pclose()
    - Using system call pipe()
  - Communication as with any other fd
    - Using system calls read(), write(), ...

Pipe in a Process

- Pipe Creation: Kernel sets up two file descriptors
  - 1st for Read: Path for obtaining the data
  - 2nd for Write: Path for data input
- Pipe to communicate with itself !!!
  - May be among threads of a process



Making Sense with Pipes

- Info: Child process inherits parent's open file descriptors
- This Info + Pipe Creation = IPC within the family
- Final Critical Decision
  - Remember pipes are half-duplex
  - So, who communicates with whom?

Multi-Process Communication

- Usual system calls operating on file descriptors
- Send data to the pipe: write()
- Retrieve data from the pipe: read()
- Though calls like lseek(), don't work

Pipes Extras & Limitations

- Two-way Pipes
  - Open two pipes
  - Assign the file descriptors, appropriately
- pipe() call must PRECEDE a call to fork()
- Pipes are only for related processes
  - That is processes with a related ancestry
- What about unrelated processes?
- There are special type of pipes called ...

FIFOs

W's of a FIFO

- A Pipe with a name in the file system
- Also called Named Pipe
- Properties identical to those of a Pipe
  - including the Serial Communication
  - which is now termed as First-In-First-Out
- Except that now processes can be unrelated
- Achieved by it being a device special file
  - which persists irrespective of Process existence
  - And any process can operate on it

FIFO Usage

- On Shell
  - Creation: mknod, or mkfifo
  - Communication: cat, echo, ...
- In Programming / C
  - Creation
    - System Call: mknod(), open(), close()
    - Library Function: mkfifo(), fopen(), fclose()
  - Communication
    - System Calls: read(), write(), ...
    - Library Functions: fread(), fwrite, fputs, ...

Experiment on Shell

- mkfifo /tmp/fifo
- ls -l /tmp/fifo
- cat < /tmp/fifo

  <See the text>

Open another shell

- cat > /tmp/fifo

  <Type some text>
- rm /tmp/fifo

Shared Memory

W's of Shared Memory

- Rawest Communication Mechanism
- Piece of Memory which is Shared
  - Between two or more processes
- Fastest form of IPC
  - As no unnecessary copying of data
  - And no other overheads – not even synchronization :)
- Synchronization is Users responsibility
  - As desired by him
  - Process Semaphores is the Mechanism provided

Linux Memory Model

- All accesses in Linux are through virtual addresses
- Each Process has its “own” Virtual Address Space
  - Which is split into Virtual Pages
- Each Process maintains a mapping table
  - From its Physical Pages to these Virtual Pages
  - For access to its actual data
- And mappings of multiple processes can point to the same physical page
  - Enabling the sharing of “actual” memory

Shared Memory Usage

- Shell Way
  - Create: ipcmk -M <size> [-p <mode>]
  - Get Information: ipcs [-m [-i id]]
  - Remove: ipcrm [-M <shmkey> | -m <shmid>]
- Programming Way in C (using sys calls)
  - Physical Memory Allocation: shmget()
  - Mapping / Attaching to Virtual Memory: shmat()
  - Access through usual (Virtual) Address Dereferencing
  - Unmapping / Detaching from Virtual Memory: shmdt()
  - Physical Memory Deallocation & Other Shared Memory Control Operations: shmctl()

Physical Memory Allocation

- Done by only one of the sharing Processes
  - int shmget(key_t key, size_t size, int shm_flg);
- Returns
  - Shared Memory id on success
  - -1 on error (and sets errno)
- key: System-wide unique id for a Shared Memory
  - Same as in Message Queue
  - Process gets a particular shared memory id using its creation key
- size: Number of bytes to allocate or get for the shared memory
- shm_flg: operation | permissions
  - IPC_CREATE, IPC_EXCL

Shared Memory Attachment

Mapping to Process Virtual Address Space

- Done by all the sharing Processes to Access the Shared Memory
  - void *shmat(int shm_id, const void *shm_addr, int shm_flg);
- Returns
  - Virtual Start Address of the attached Shared Memory on success
  - MAP_FAILED i.e. (void *)(-1) on error (and sets errno)
- shm_id: Shared Memory id returned by shmget()
- shm_addr
  - Requested process virtual start address
  - NULL enables Kernel to choose one
- shm_flg
  - SHM_RND: round down shm_addr to a multiple of the page size
  - SHM_RDONLY: shared memory will be only read, not written
- Children created by fork() inherit attached shared segments

Shared Memory Detachment

Unmapping from Process Virtual Address Space

- Done by all the Processes, which did attach, or inherit
  - int shmdt(const void *shm_addr);
- Returns: 0 on success; -1 on error (and sets errno)
- shm_addr: Process virtual address map returned by shmat()
- Call to _exit() or exec*() automatically detaches
- Detachment may deallocate the shared memory
  - If done by the last process
  - And already marked to deallocate

Need for Synchronization

- We already talked
  - Shared Memory has no Access Coordination
  - It has to be handled by us – the User
- Moreover, now we observed
  - The issue with the Example
- So, now its time to look into resolving it
- That's where we go to the next topic ...

Specialized Semaphores

- Binary semaphore
  - Semaphore controlling a single resource “availability”
  - Possible values
    - 0 → Unavailable
    - 1 → Available
  - Typically initialized to 0
- Mutex
  - Specialized Binary Semaphore
  - Used for a “Usable” rather than a “Consumable” Resource
  - Process using it, only releases it
    - Giving a notion of ownership
  - Typically initialized to 1