Lecture 11, 12: Interprocess Communication (IPC)

OS Course (231)

What is IPC?

- It is a mechanism that allows processes to exchange data and coordinate their activities

- Processes and memory isolation
  - Each process has its own private memory space
  - IPC allows communication while preserving the isolation

- Cooperation between processes
  - Many processes might need to cooperate in order to accomplish a task
  - IPC provides efficient and safe mechanisms for the cooperation and coordination to take place

Mechanisms for IPC

- Pipes
  - Allow one-way or two-way communication between processes
- Shared memory
  - Allows multiple processes to access a common memory space, enabling fast data sharing
- Message queues
  - Provide a queue structure where messages can be sent by one process and read by another
- Signals
  - Used for sending simple notifications between processes
- Sockets
  - Enable communication between processes over a network

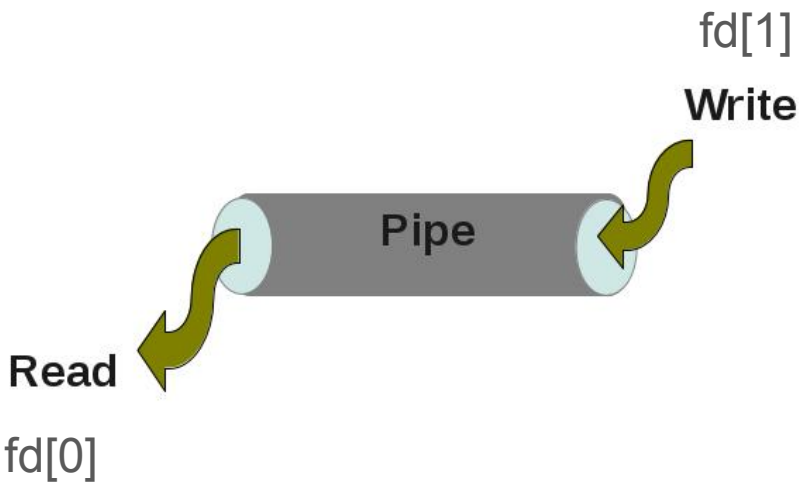
Why IPC is required?

- Data sharing
  - In many applications, processes need to share data
  - But each process has its own separate memory space
  - IPC provides mechanisms for processes to share data safely and efficiently
- Modular program design
  - Large programs can be divided into smaller, independent processes each handling a specific task
  - These processes can coordinate with each other using IPC mechanisms
- Concurrency and parallelism
  - Multiple processes often run concurrently. IPC allows synchronization and coordination between concurrently running processes.
- Resource sharing
  - Multiple processes may need access to the same resources (for eg., files). IPC provides mechanism to manage access to shared resources.
- Distributed systems
  - Processes running on different machines need to communicate and exchange data. IPC mechanisms enable such communication.

Pipes

- Allows data to be sent from one process to another
- It acts like a communication channel between processes, enabling unidirectional (or sometimes bidirectional) data flow
- Unidirectional - Data flows in one direction, from the writing process to the reading process
- Buffering - Pipes have a buffer that temporarily stores data until it is read by the receiving process.

Pipes



Pipes

- Blocking read and write by default. Nonblocking mode can also be enabled.
- When a process tries to read from a pipe, it will block (wait) if there is no data available in the pipe.
- When a process tries to write to a pipe that is full (the buffer has reached its limit), it will also block (wait) until there is space available in the pipe.

Pipes

Code demonstration

File Permissions

- Owner
  - Read (r) : 2^0 = 1
  - Write (w) : 2^1 = 2
  - Execute (x): 2^2 = 4
- Group
  - Read (r) : 2^0 = 1
  - Write (w) : 2^1 = 2
  - Execute (x): 2^2 = 4
- Others
  - Read (r) : 2^0 = 1
  - Write (w) : 2^1 = 2
  - Execute (x): 2^2 = 4

- What does a file permission of 775 mean?
  - Owner (7), Group (7), Others (5)
  - Owner and Group has read, write, execute permissions = 1+2+4 = 7
  - Others have read and execute permissions = 1+4 = 5

Named Pipes

- A named pipe is a special type of pipe that uses files for IPC
- Named pipe have a persistent space in the filesystem
- They can be used by unrelated processes
- Similar to anonymous pipes, named pipes also allow only half-duplex communication, i.e., the flow of data is only in one direction and it can only be read or written at a time.
- Reading is blocked if the pipe is empty. Writer is blocked if the pipe is full.

Pipes - Code Sample

Named Pipes

Code demonstration

Introduction to Signals

- A signal is a software interrupt delivered to a process by the OS.

- Used to notify processes about **events**.

- Sources of signals:
  - The OS (e.g., SIGSEGV, SIGKILL)
  - Another process (via kill() or sigqueue())
  - The process itself (raise())

- Each signal has
  - Default action (terminate, ignore, stop, continue)
  - Custom handler (defined by the program)

Default Actions for Common Signals

Signal	Default Action	Notes
SIGINT	Terminate	Sent by Ctrl + C in terminal
SIGTERM	Terminate	Graceful termination request
SIGKILL	Terminate (uncatchable)	Cannot be handled or ignored
SIGSEGV	Terminate + core dump	Invalid memory access (seg fault)
SIGABRT	Terminate + core dump	Abnormal termination (abort())
SIGCHLD	Ignore	Sent to parent when child exits
SIGSTOP	Stop (suspend)	Cannot be handled or ignored
SIGCONT	Continue (resume if stopped)	
SIGUSR1	Terminate	User defined signal #1
SIGUSR2	Terminate	User defined signal #2

Signals for IPC

- Signals can be used for basic IPC

- Eg., one process signals to the other to indicate
  - A message is ready
  - A task should start/stop

- User-defined signals: SIGUSR1 and SIGUSR2

- Limitation
  - Only type of event is sent (not bulk data)
  - Payloads are very small

Key System Calls

- Install a handler
  - signal(signum, handler)

- Send a signal
  - kill(pid, signum) -> send signal to the process indicated by the pid
  - raise(signum) -> send signal to the currently running thread
  - sigqueue(pid, signum, value) -> send signal with small data

- Wait for signals
  - pause() -> sleep until a signal arrives

The kill() misnomer

- kill() system call is used to send a signal
  - int kill(pid_t pid, int sig);

- Despite its name, it doesn’t always kill a process

- Behaviour depends on the signal
  - kill(pid, SIGKILL) -> process is terminated (uncatchable)
  - kill(pid, SIGUSR1) -> process runs its handler
  - kill(pid, 0) -> no signal sent; only checks if process exists

- Think of kill() as “send signal” rather than “kill process”

Code Demo