Taller de IPC

Clase pública
Sistemas Operativos
2do Cuatrimestre de 2018

- Conceptos básicos de IPC.
- Comunicación vía `pipes`.
- Comunicación vía `sockets`.
- Enunciado del Taller.

Comunicación vía `pipes`

- *Pipe* (tubería): Canal de comunicación unidireccional entre dos procesos.
- Típico *pipe* en *bash*: **echo "sistemas es lo más" | wc -c**
  1. Se llama a `echo`, un programa que escribe su parámetro por **stdout**.
  2. Se llama a `wc -c`, un programa que cuenta cuántas palabras (-w) o bytes (-c) entran por **stdin**.
  3. Se conecta el **stdout** de `echo` con el **stdin** de `wc -c`.

Comunicación vía `pipes`

En la implementación en lenguaje C, se usan dos tipos de pipes:

- Con nombre (FIFOs): Son colas, y se utiliza la función *mkfifo()*.
- Sin nombre: Se utiliza la función (*syscall*):

int pipe(int pipefd[2]).

  - Para implementar un pipe sin nombre se usa un archivo **temporal** y **anónimo** que se aloja en memoria y actúa como un **buffer** para leer y escribir de manera **secuencial**.
  - Luego de ejecutar `pipe`, se tiene:
    - En `pipefd[0]` un **file descriptor** que apunta al extremo del `pipe` en el cual se **lee**.
    - En `pipefd[1]` otro **file descriptor** que apunta al extremo del `pipe` en el cual se **escribe**.

Comunicación vía `pipes` - sin nombre

Para leer de un **file descriptor** se usa:

ssize_t read(int fd, void *buf, size_t count);

- Intenta leer hasta *count* bytes desde el archivo con descriptor *fd* dentro del buffer que empieza en *buf*.

Para escribir un **file descriptor** se usa:

ssize_t write(int fd, const void *buf, size_t count);

- Escribe hasta *count* bytes desde el buffer que inicia en *buf* al archivo referenciado por el descriptor *fd*.

En ambos casos se retorna la cantidad de bytes leídos o escritos, ó -1 en caso de error.
Además, **write** y **read** son bloqueantes.

Comunicación vía Sockets

- Se utilizan para comunicar procesos que no se conocen entre sí.
- Los **sockets** pueden trabajar en distintos contextos (tipos de direccionamiento).
- En el contexto UNIX los sockets son nombres de archivo y sirven para comunicar dos procesos en una misma máquina.
- En el contexto de redes se conforman por una dirección IP y un puerto, y sirven para comunicar dos procesos de dos máquinas diferentes vía red.

Envío de paquetes con sockets

En redes existen dos tipos de conexiones: UPD (sin conexión - sin confirmación) y TCP (orientado a conexión).

Conexión UDP: Servidor (recibe mensajes)

1. **int socket(int domain, int type, int protocol);** Crear un nuevo **socket**.
2. **int bind(int fd, sockaddr* a, socklen_t len);** Asigna una dirección (nombre o IP y puerto) al **socket**.
3. **size_t recv(int s, void *buf, size_t l, int flags);** Lee un paquete

Cliente (envía mensajes)

1. **int socket(int domain, int type, int protocol);** Crear un nuevo **socket**.
2. **size_t send(int s, void *buf, size_t l, int flag);** Envía un paquete

Conexión TCP: Servidor:

1. **int socket(int domain, int type, int protocol);** Crear un nuevo **socket**.
2. **int bind(int fd, sockaddr* a, socklen_t len);** Asigna una dirección (nombre o IP y puerto) al **socket**.
3. **int listen(int fd, int backlog);** Se asigna una cola y se utiliza el **socket** para recibir conexiones entrantes.
4. **int accept(int fd, sockaddr* a, socklen_t* len);** Espera la próxima conexión de un cliente en el **socket** (y crea un nuevo socket para atenderla).

Cliente:

1. **int socket(int domain, int type, int protocol);** Crear un nuevo **socket**.
2. **int connect(int fd, sockaddr* a, socklen_t* len);** Conectarse a un **socket** remoto que debe estar escuchando.

Una vez que un cliente solicita conexión y esta es aceptada por el servidor puede comenzar el intercambio de mensajes con:

- **ssize_t send(int s, void *buf, size_t len, int flags);**
- **ssize_t recv(int s, void *buf, size_t len, int flags);**