

# Inter-process communication (IPC)

CS350

Kartik Gopalan

# Some simple forms of IPC

- Parent-child

- Command-line arguments,
- `wait(...)`, `waitpid(...)`
- `exit(...)`

- Reading/modifying common files

- Servers commonly use 'pid' file to determine other active servers.

- Signals

- Event notification from one process to another

# Some more forms of IPC...

- Shared Memory

- Common piece of read/write memory.
- Needs synchronization for access

- Semaphores

- Locking and event signaling mechanism between processes

- Pipes

- Uni-directional (if used cleanly)
- `'ps -aux | more'`

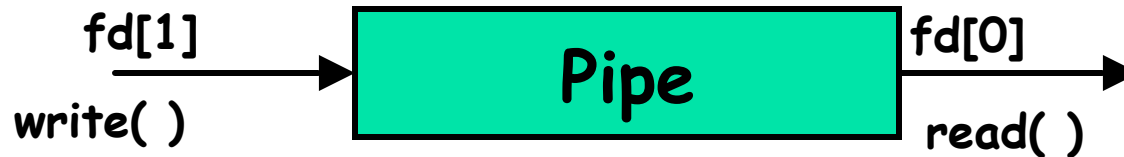
- Sockets

- Bi-directional
- Not just across the network, but also between processes.

# Pipes

# Pipe Abstraction

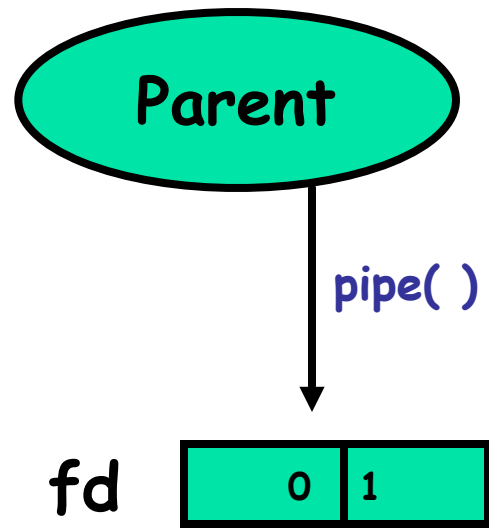
- Write to one end, read from another
- `pipe( )`



# Pipe provides a byte-stream abstraction

- You can read and write at arbitrary byte boundaries.
  - E.g. Byte lengths sequence written
    - 10, 10, 10, 10
  - byte lengths sequence read
    - 5, 15, 15, 5
- As opposed to **message abstraction**, which provides explicit message boundaries.
  - E.g. network packets

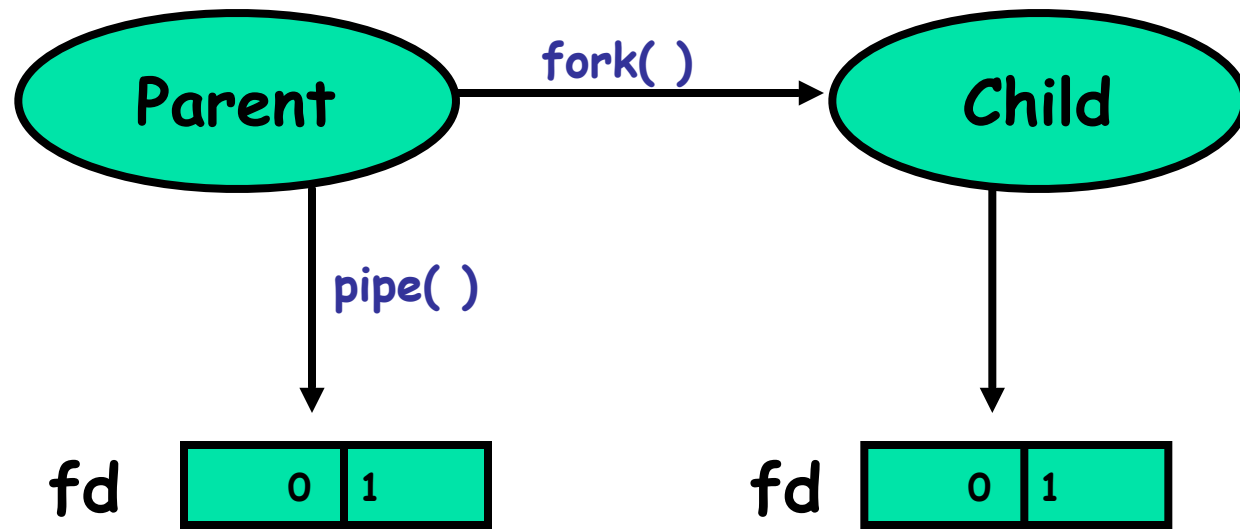
# Parent-child communication using pipe



**Here's an example.**

<http://www.cs.binghamton.edu/~kartik/examples/pipe1.c>

# Parent-child communication using pipe

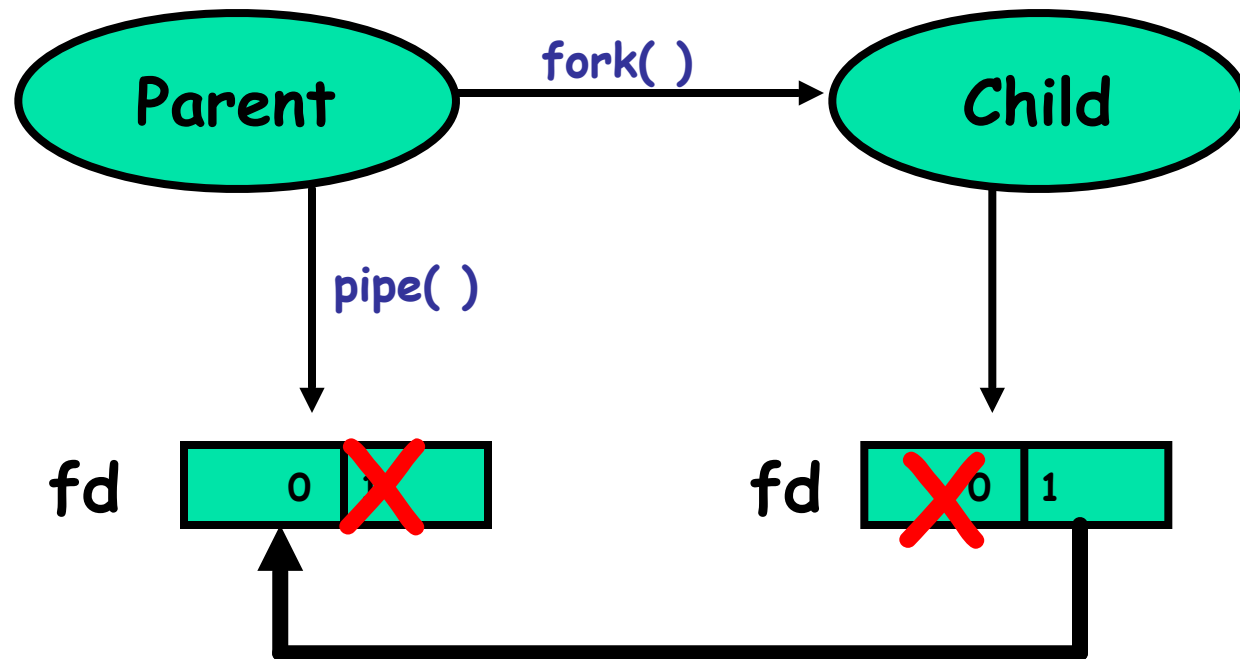


**Here's an example.**

<http://www.cs.binghamton.edu/~kartik/examples/pipe1.c>



# Parent-child communication using pipe

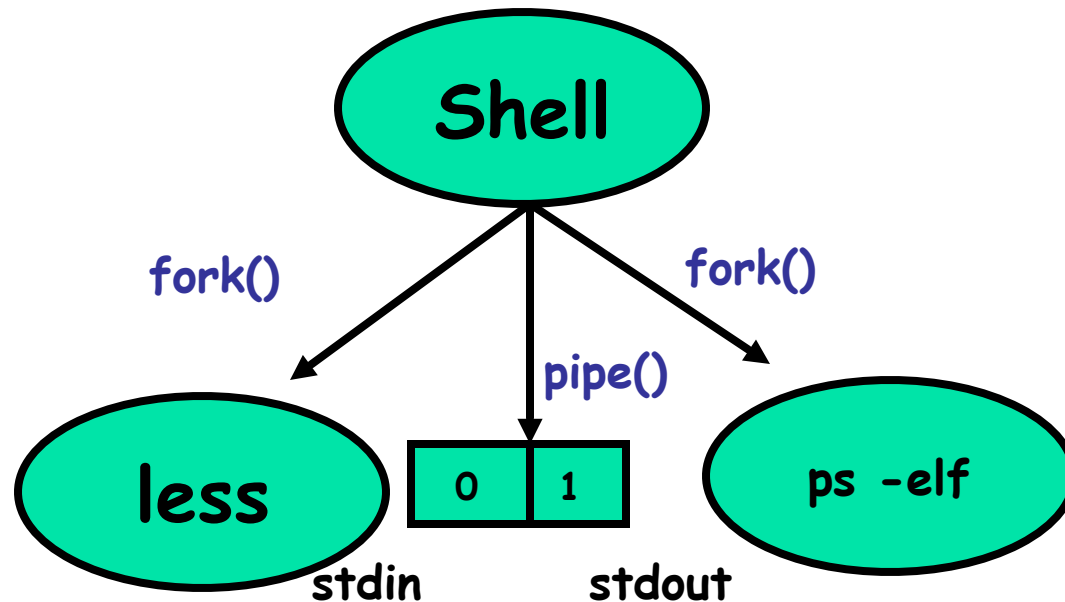


**Here's an example.**

<http://www.cs.binghamton.edu/~kartik/examples/pipe1.c>

# Filters in shell command-line

`ps -elf | less`

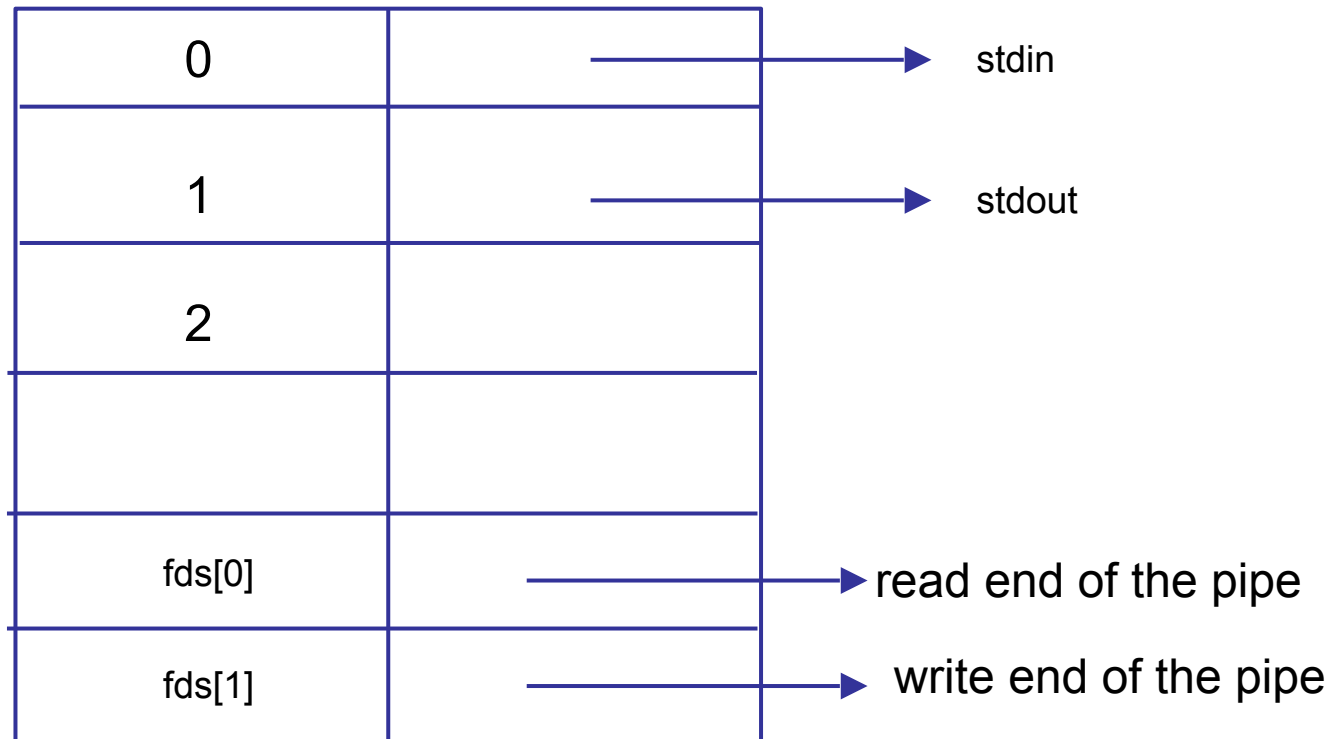


**Here's an example.**

<http://www.cs.binghamton.edu/~kartik/examples/pipe2.c>

# Understanding fds: File-Descriptor Table

- Each process has a file-descriptor table
- One entry for each open file
- “File” = regular files, stdin, stdout, pipes, I/O devices etc.



# Handling long chain of filters

## — Recursive approach

- create a pipe
- fork a child
- redirect stdin and/or stdout as necessary
- fork another child for next level of recursion with a shorter command
- exec the command for the current level

# Being careful with read()/write()

- `read(fds[0], buf, 6);`

- Doesn't mean read will return with 6 bytes of data!
- It could be less. Why?

- Some reasons

- `read()` could reach end of input stream (EOF).
- Other endpoint may abruptly close the connection
- `read()` could return on a signal.

- So you MUST incorporate error handling with every I/O call (actually with any system call)

# Error handling...

You **must**

- First check the return value of **every** read(...)/write(...) system call.
- Then either...
- Wait to read/write more data  
OR
- Handle any error conditions

More convenient to write a wrapper function

/\* Write "n" bytes to a descriptor. \*/

ssize\_t **written**(int fd, const void \*vptr, size\_t n)

```
{
    size_t      nleft;
    size_t      nwritten;
    const char  *ptr;

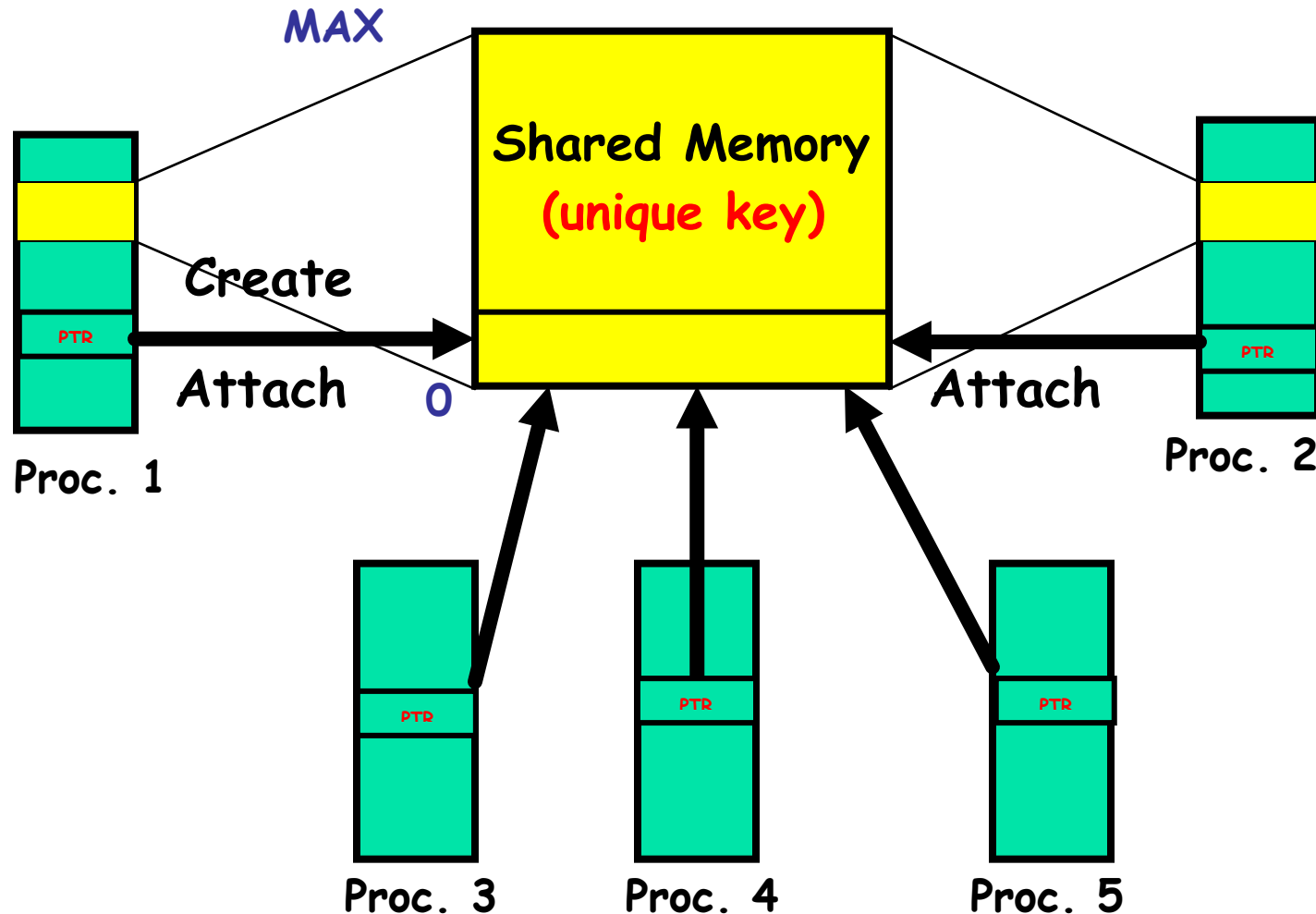
    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) <= 0) {
            if (errno == EINTR)
                nwritten = 0; /* call write() again */
            else return(-1); /* error */
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return(n);
}
```

# Shared Memory, Semaphores

- Man pages : shmget, shmat, shmdt, shmctl, semget, semop, semctl

# Shared Memory

Common chunk of read/write memory among processes





# Creating Shared Memory

```
int shmget(key_t key, size_t size, int shmflg);
```

Example:

```
key_t key;  
int shmid;
```

```
key = ftok("<somefile>", 'A');
```

```
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

Here's an example.

[http://www.cs.binghamton.edu/~kartik/examples/shm\\_create.c](http://www.cs.binghamton.edu/~kartik/examples/shm_create.c)

# Attach and Detach Shared Memory

```
void *shmat(int shmid, void *shmaddr, int shmflg);  
int shmdt(void *shmaddr);
```

Example:

```
key_t key;  
int shmid;  
char *data;  
key = ftok("<somefile>", 'A');  
shmid = shmget(key, 1024, 0644);  
data = shmat(shmid, (void *)0, 0);  
// read or write something to data here.  
shmdt(data);
```

Here's an example.

[http://www.cs.binghamton.edu/~kartik/examples/  
shm\\_attach.c](http://www.cs.binghamton.edu/~kartik/examples/shm_attach.c)

# Deleting Shared Memory

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

```
shmctl(shmid, IPC_RMID, NULL);
```

## **Example:**

[http://www.cs.binghamton.edu/~kartik/examples/  
shm\\_delete.c](http://www.cs.binghamton.edu/~kartik/examples/shm_delete.c)

# Command-line IPC control

- `ipcs`

- Lists all IPC objects owned by the user

- `ipcrm`

- Removes specific IPC object

# Signals

# Signals Overview

- **Signal** is a notification to a process that an event has occurred.
  - Could come from another process or from the OS
- Type of event determined by type of signal
- Try listing all signal types using

```
% kill -l
```
- Some interesting signals
  - SIGCHLD, SIGKILL, SIGSTOP

# Handling Signals

- Signals can be **caught** – i.e. an **action** can be associated with them
  - `SIGKILL` and `SIGSTOP` cannot be caught.
- Actions to signals can be customized using **`sigaction(...)`** which associates a **signal handler** with the signal.
- Default action for most signals is to terminate the process
  - Except `SIGCHLD` and `SIGURG` are ignored by default.
- Unwanted signals can be ignored
  - Except `SIGKILL` or `SIGSTOP`
- Here's an example.
  - [http://www.cs.binghamton.edu/~kartik/examples/signals\\_ex.c](http://www.cs.binghamton.edu/~kartik/examples/signals_ex.c)

# More on SIGCHLD

- Sent to parent when a child process **terminates** or **stops**.
- If `act.sa_handler` is **SIG\_IGN**
  - **SIGCHLD** will be ignored (default behavior)
- If `act.sa_flags` is **SA\_NOCLDSTOP**
  - **SIGCHLD** won't be generated when children stop
- `act.sa_flags` is **SA\_NOCLDWAIT**
  - children of the calling process will not be transformed into zombies when they terminate.
- These need to be set in **sigaction()** **before** parent calls **fork()**



# Reading child's exit status without blocking on wait()

- Parent could install a signal handler for SIGCHLD
- Call `wait(...)` / `waitpid(...)` inside the signal handler

```
void handle_sigchld(int signo) {  
    pid_t pid;  
    int stat;  
  
    pid = wait(&stat); //returns without blocking  
  
    printf("child %d terminated\n", pid);  
}
```

- Here's an example.
  - <http://www.cs.binghamton.edu/~kartik/examples/sigchld.c>

# More information...

- Check ‘**man sigaction(...)**’
- Understand what happens when signal is delivered in the middle of a system call?
  - Different OSes have different behavior.
- Google for keywords “Unix Signals”
  - Tons of useful links

# References

- Unix man pages

- “Advanced Programming in Unix Environment” by Richard Stevens

- <http://www.kohala.com/start/apue.html>