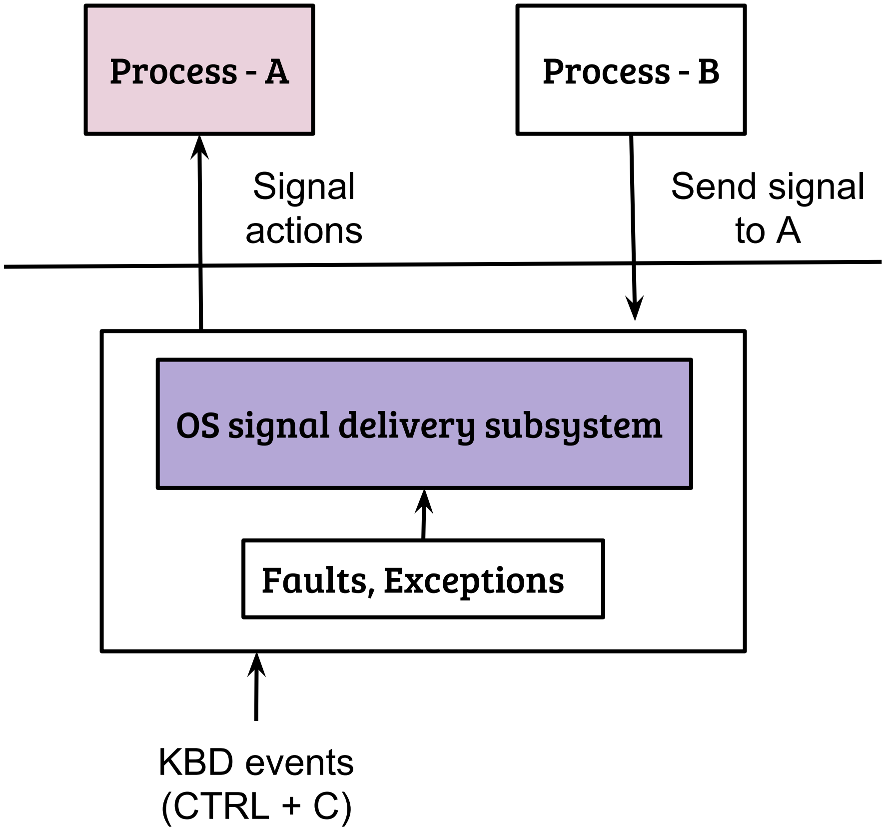
Operating Systems

Signals, Shell operations and IPC techniques

Debadatta Mishra, CSE, IITK

Why Signals?



- Termination of a process
- Process induced exceptions, div-by-zero, access to illegal memory area etc.
- Notification to process, SIGALRM, SIGCHLD
- Interprocess communication
- Custom actions on events

Signal semantics

Destination process

signal (signum, handler)

SIGHUP
SIGINT
SIGALRM
SIGUSR1
SIGCHLD
....

1. **SIG_IGN: Ignore the signal**
2. **SIG_DFL: Default action**
3. **Function address: custom handling**

- If signal handler not registered, process is terminated (mostly)
- SIGKILL and SIGSTOP → no custom actions
- How does the handler invoked?

Signal semantics

Source process

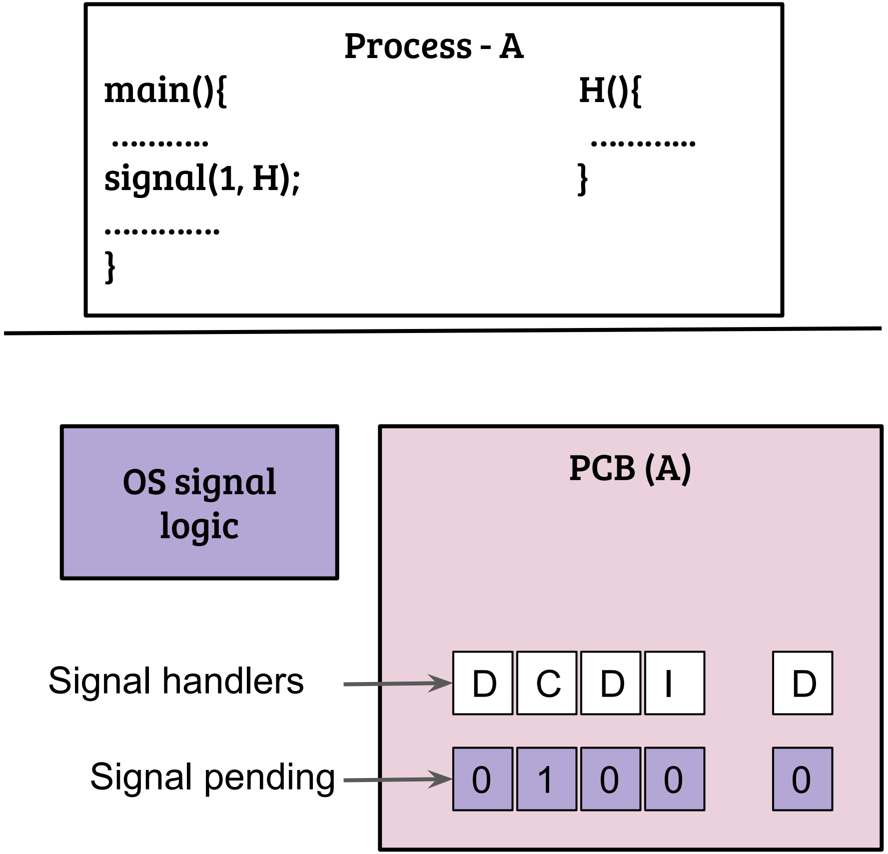
kill (pid, signum)

PID Of
target
process

SIGHUP
SIGINT
SIGALRM
SIGUSR1
SIGCHLD
....

- If pid == 0, signal is sent to all processes in the process group
- Must have permissions to send signals → same user or root user

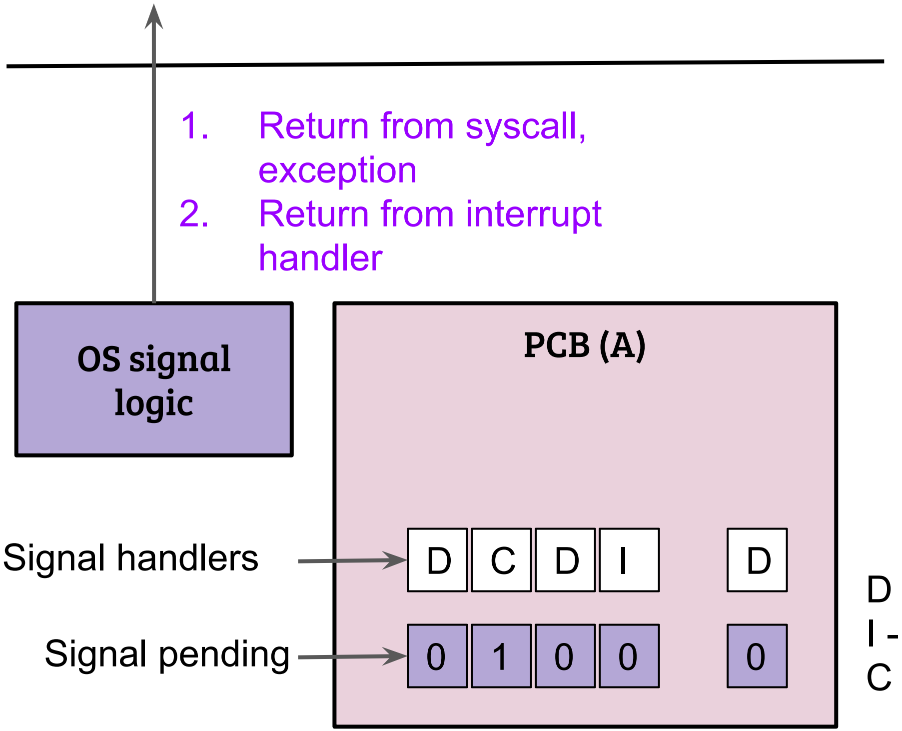
OS support for signals



- Signal pending is modified by either kill ( ) or OS
- Signal handlers are registered by the OS or the process
- Synchronous delivery is easy but not always possible
- Asynchronous delivery is complex

D - default
I - Ignore
C - Custom

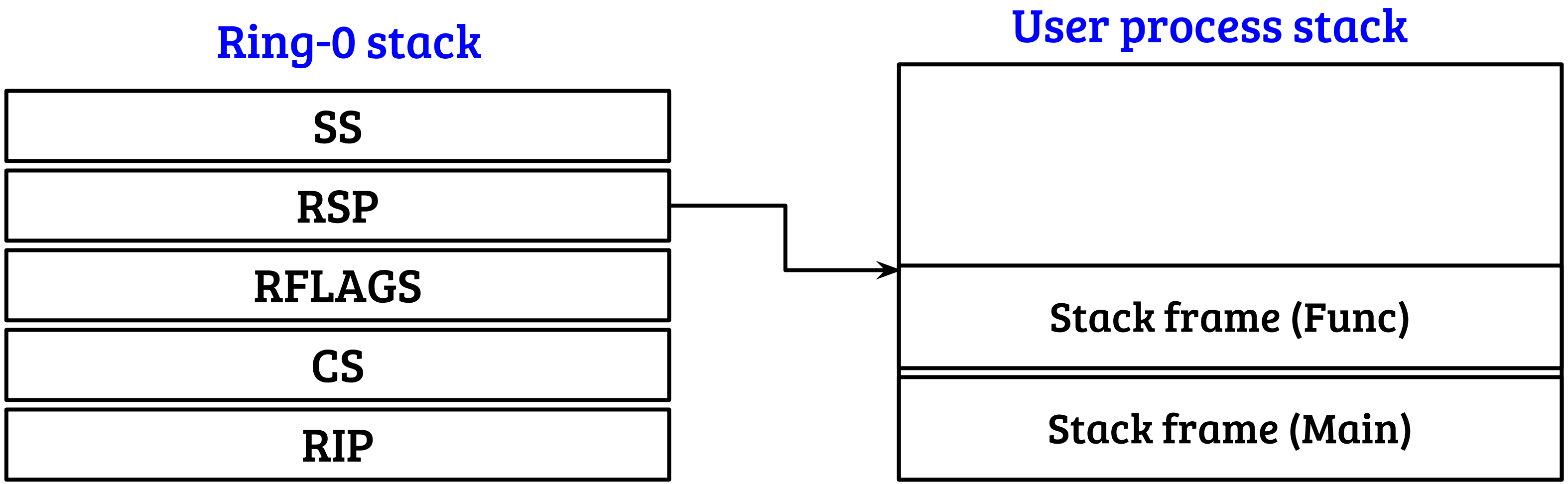
When to deliver signals?



D - default
I - Ignore
C - Custom

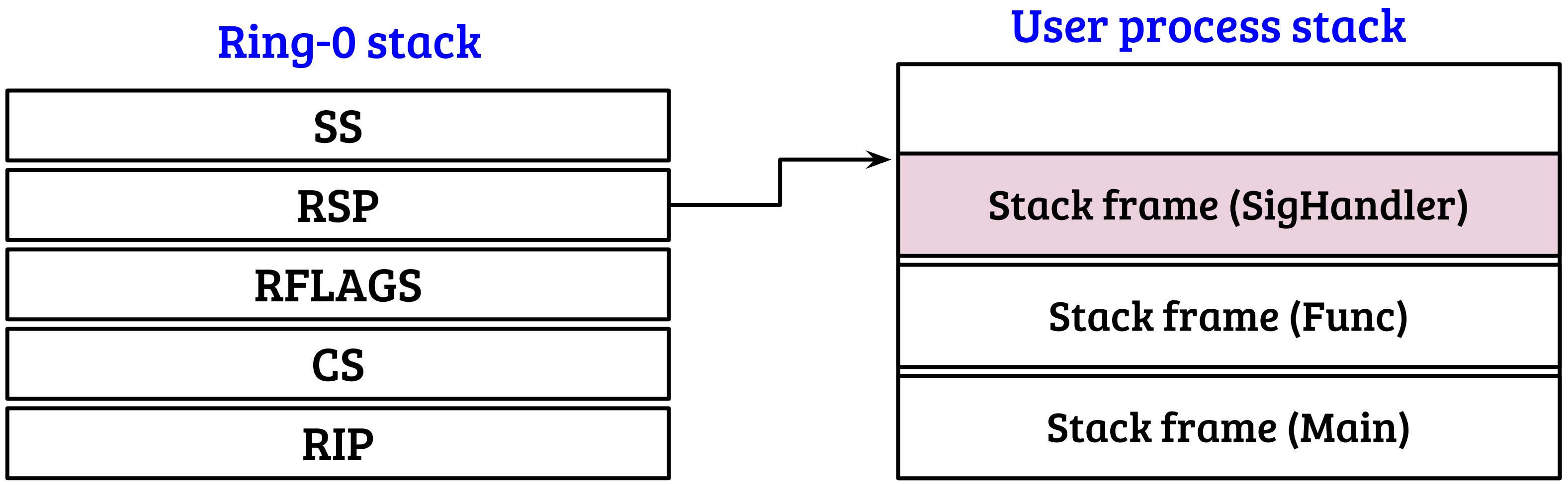
- Check for signal pending bit
- Invoke handler actions → default, ignore or custom
- For custom handlers, execution resumes after the handler is finished.
- How?

Signal delivery



- Assume that the program was interrupted when it was executing "Func"
- The OS can modify the user stack and the RIP. What values?

Signal delivery

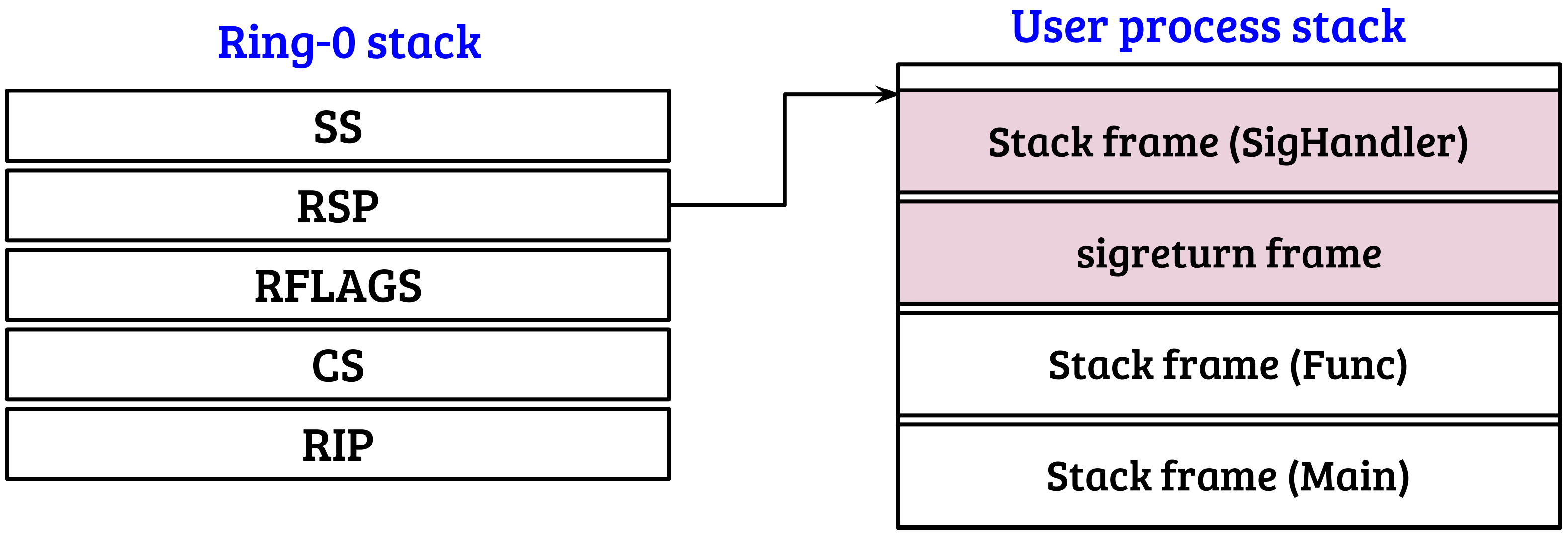


- Assume that the program was interrupted when it was executing “Func”
- The OS can modify the user stack and “mimic” a function call

Signal handling: design choices

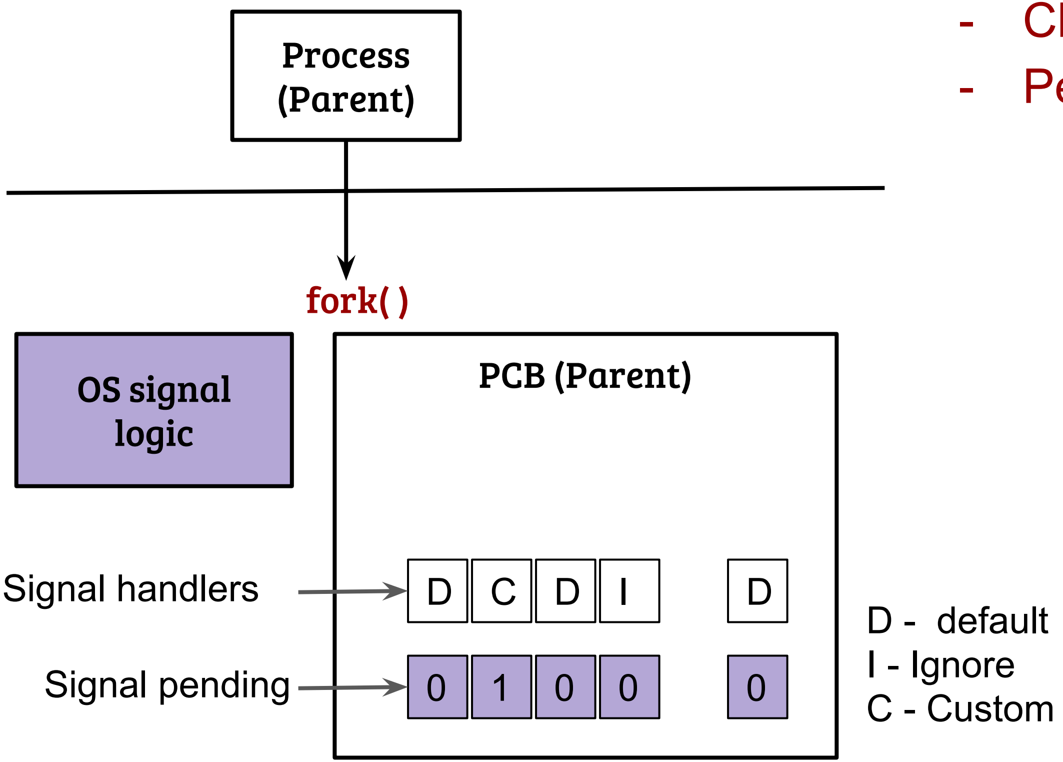
- Remove handler → Handle signal → Register signal handler again
- Don’t remove handler → New signal during handling → invoke the handler
- Temporarily disable signal → handle signal → re-enable signal

Signal delivery with handler notification (Linux)



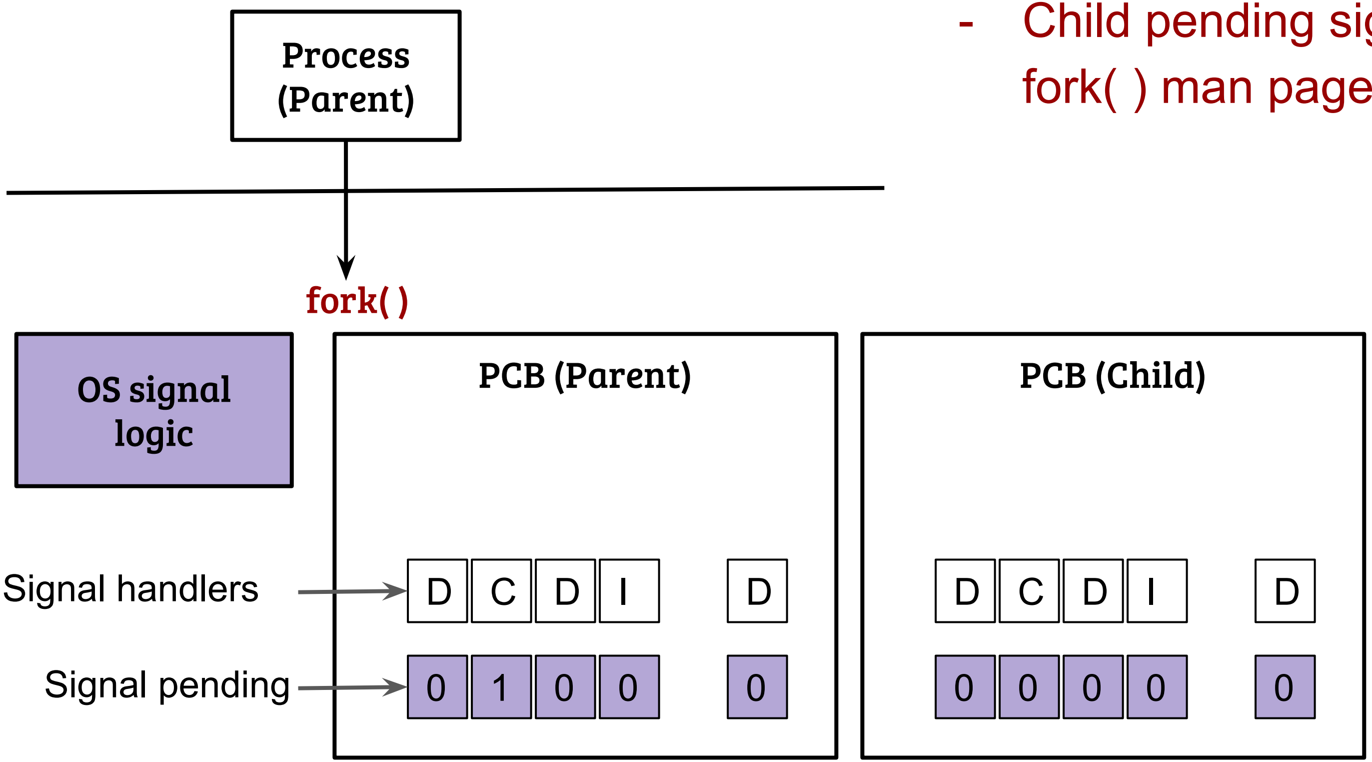
- Sigreturn function invokes a syscall to reenable signal
- OS restores the stack to original state

Signal and fork( )



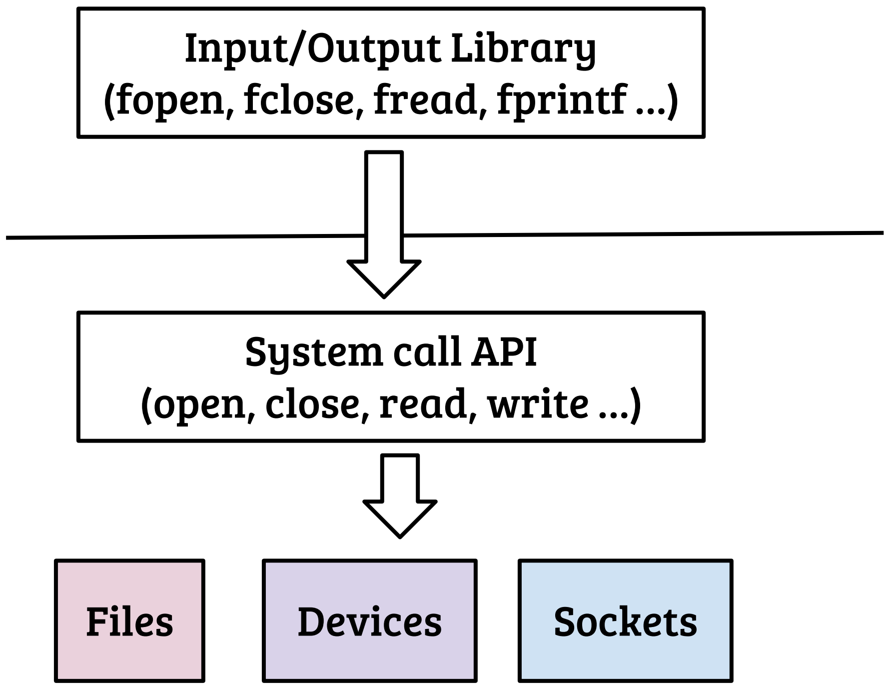
- Child inherits the signal handlers
- Pending signals?

Signal and fork( )



- Child pending signals is empty -- fork( ) man page. Why?

System calls (open, close, read, write)



- User process identify files through a file handle a.k.a. file descriptors
- In UNIX, the POSIX file API is used to access files, devices, sockets etc.

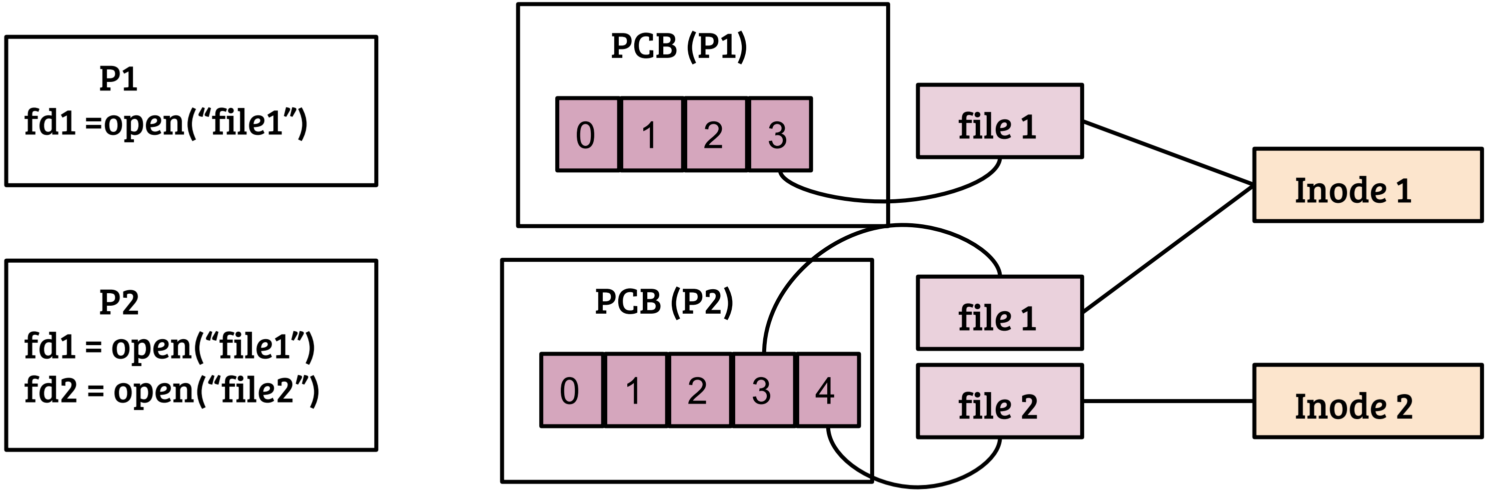
File system calls: getting a handle

Open: Provides a handle to a file

int open (char *path, int flags, mode_t mode)

- Access mode specified in flags : O_RDONLY, O_RDWR, O_WRONLY
- Access permissions check
- On success, a file descriptor (int) is returned
- If flags contain O_CREAT, mode specifies the file creation mode
- Open fds remain open across exec( ), can be changed by setting O_CLOEXEC flag

Process view of file



- Per-process file descriptor table with pointer to "file" object
- fd → file (many-to-one)
- file → inode (many-to-one)
- On fork( ), child inherits open file handles
- 0, 1, 2 are STDIN, STDOUT and STDERR, respectively

Read and Write

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

- fd → file handle
- buf → user buffer as read destination
- count → #of bytes to read

read ( ) returns #of bytes actually read, can be smaller than count

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

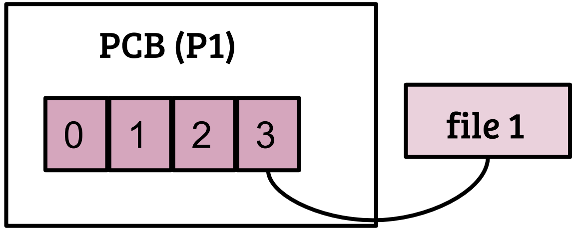
- Similar to read

Duplicate file handles (dup and dup2)

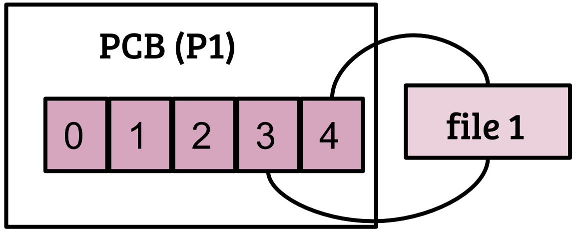
P1
fd1 =open(“file1”)

+ dup(fd1)

Before dup( )



After dup( )



- Lowest available duplicate file handle is returned
- Example:

  close(1); dup(fd)

- dup2(old, new) performs both steps in one system call

Shell redirection

- Example: ls > tmp.txt
- How implemented?

Shell redirection

- Example: ls > tmp.txt
- How implemented?

- fd = open (“tmp.txt”)
- close(1); close(2); // close STDOUT and STDERR
- dup(fd); dup(fd) // 1→ fd, 2 → fd
- Invoke exec( )

Shell: ls | wc -l

- Output of “ls” is input to “wc -l”
- How implemented?

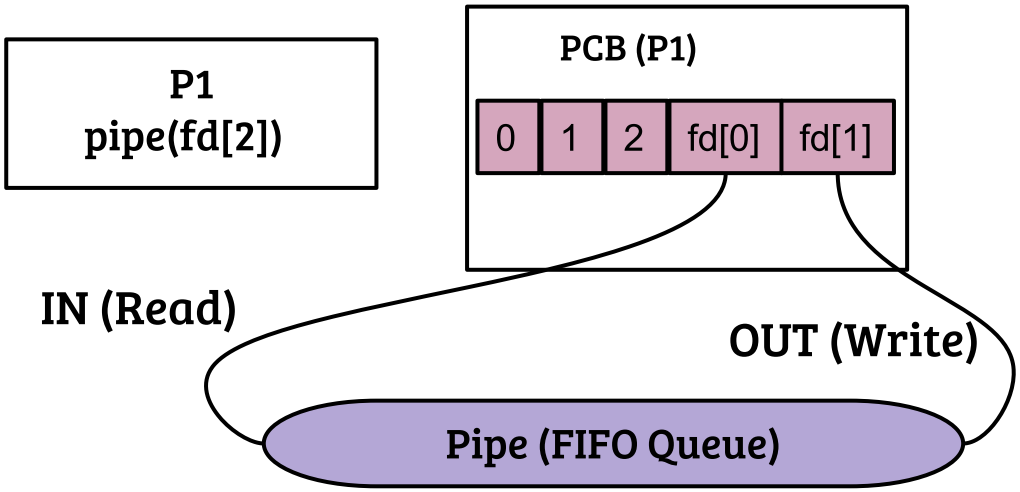
- Option 1

  temporary intermediate file + dup(0)

- Option 2

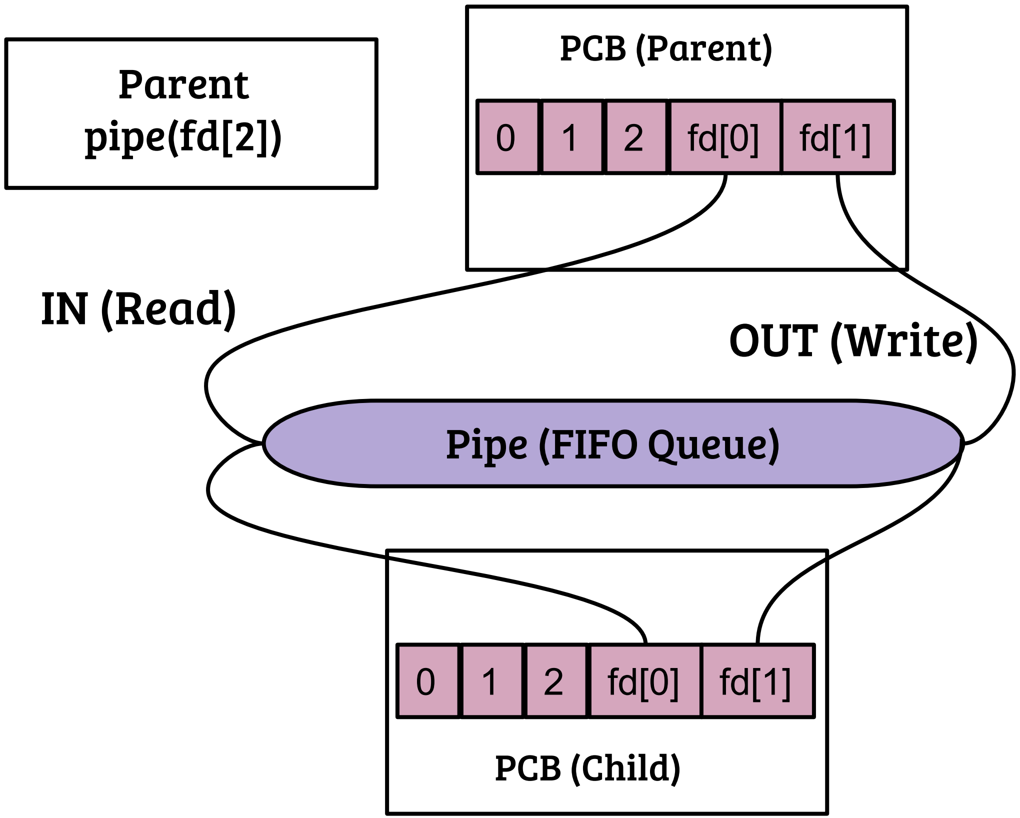
  Shared buffer between “ls” and “wc” process

UNIX pipe( ) system call



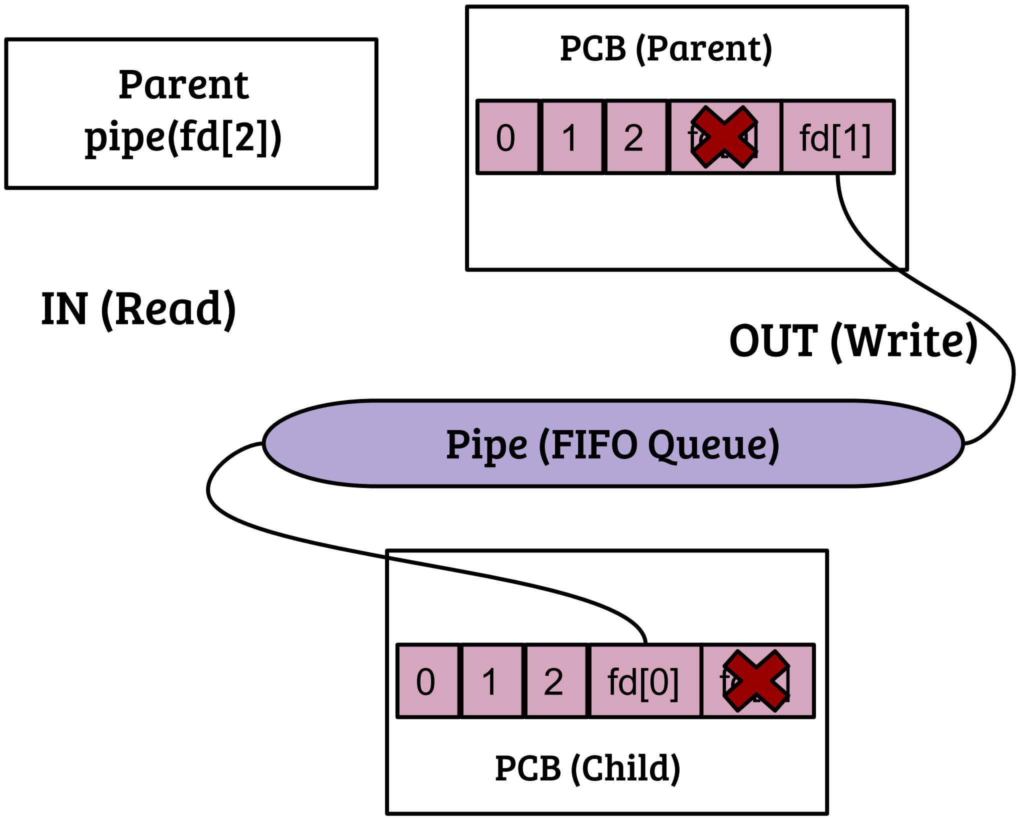
- pipe( ) takes array of two FDs as input
- fd[0] is the read end of the pipe
- fd[1] is the write end of the pipe
- Implemented as a FIFO queue in OS

UNIX pipe( ) with fork( )



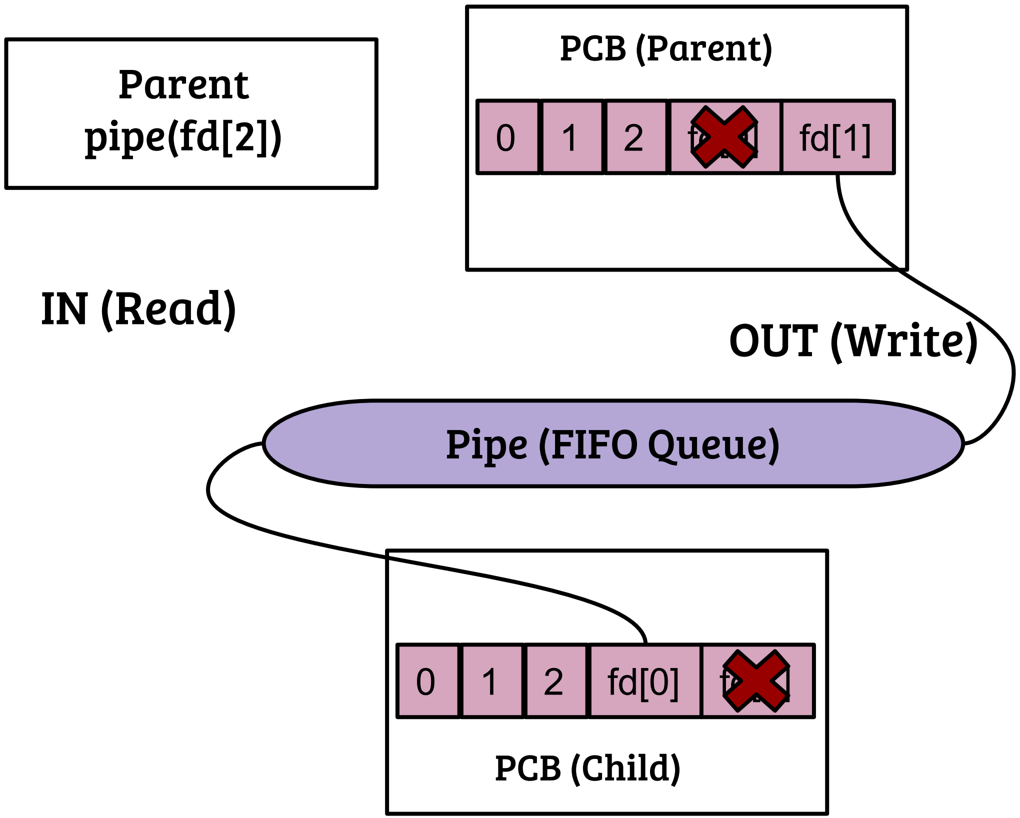
- fork( ) duplicates the file descriptors
- close( ) one end of the pipe, both in child and parent
- Result: a queue between parent and child

UNIX pipe( ) with fork( )



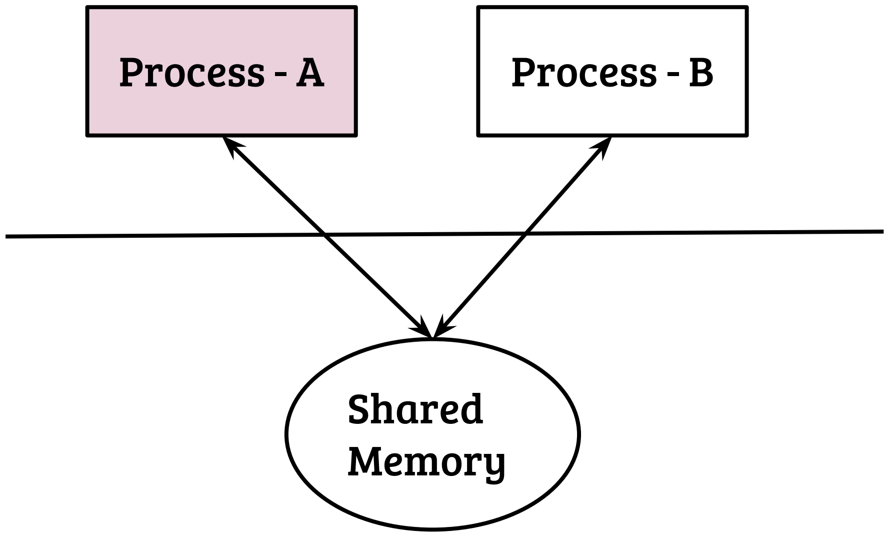
- fork( ) duplicates the file descriptors
- close( ) one end of the pipe, both in child and parent
- Result: a queue between parent and child

Shell piping : pipe( ) + dup( ) + fork( ) + exec( )



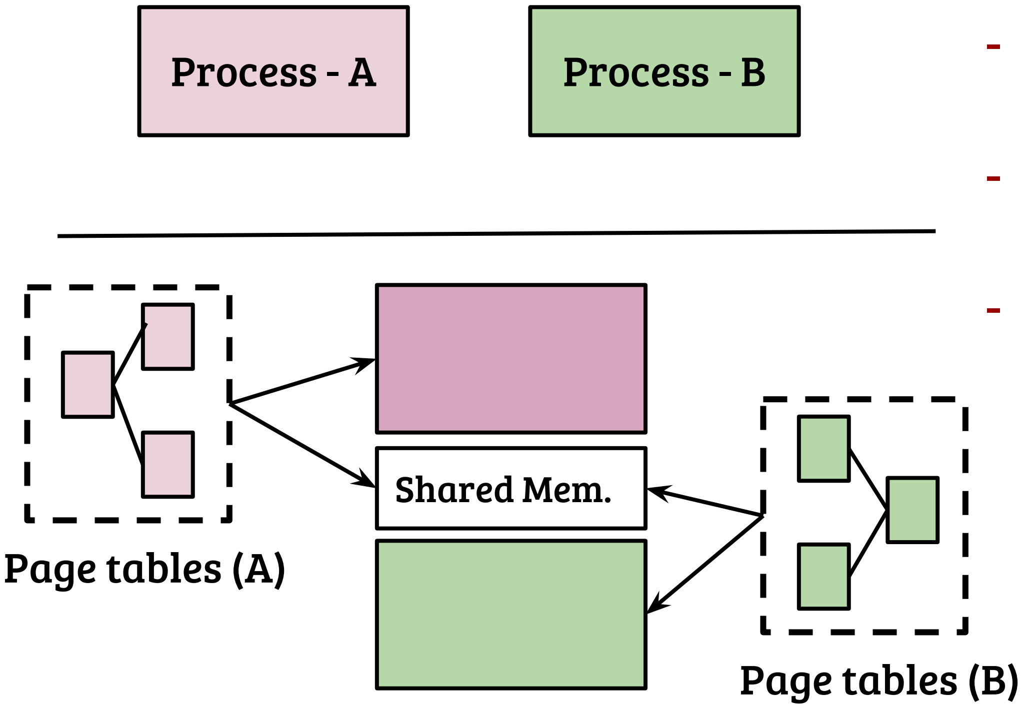
- pipe( ) followed by fork( )
- exec("ls") after closing STDOUT and duping OUT fd of pipe
- exec("wc") after closing STDIN and duping IN fd of pipe
- Result: input of "wc" is connected to output of "ls"

Shared memory



- Shared memory made accessible using virtual memory
- How?

Shared memory

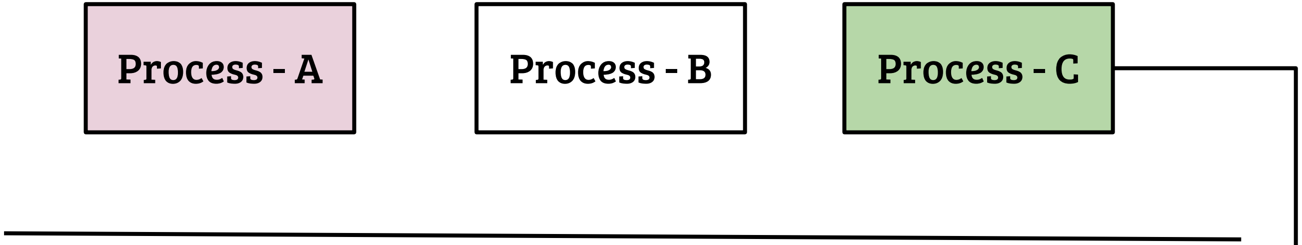


- A and B both map the shared region
- Is it required to be mapped to same VA in both processes?
- How shared memory regions are managed?

Shared memory design (API)

- How a shared memory region is created?
  - Must have a global identity
  - The OS maintains a list of shared regions
  - handle = shmget (key, size, flags = IPC_CREAT|0666)
- How any process gets an handle to a shared memory region?
  - Process must identify the shared region
  - The OS looks up in the list of shared regions
  - handle = shmget (key, size, flags = 0666)
- How any process gets the VA to the shared region?
  - Page table mapping inserted
  - shmat(handle, address_hint, flags)

Shared memory design (OS)



shmget(0x1234, 4096, 0666)

ID	PFNs, Size	Permissions	Users (processes)
0x1234	8000, 1	RD+WR	A, B

- OS looks up shared region in the global shared region table

Shared memory design (OS)

Process - A

Process - B

Process - C

shmat(handle)

ID	PFNs, Size	Permissions	Users (processes)
0x1234	8000, 1	RD+WR	A, B, C

- OS creates V to P mapping for process C

Shared memory across fork and exec

- The child inherits the attached shared memory segments
- On exec( ), all attached shared memory segments are detached
- On exit( ), all attached shared memory segments are detached