Chapter 3: IPC, Pipes and Signals

CSCI 3753 Operating Systems

Instructor: Chris Womack

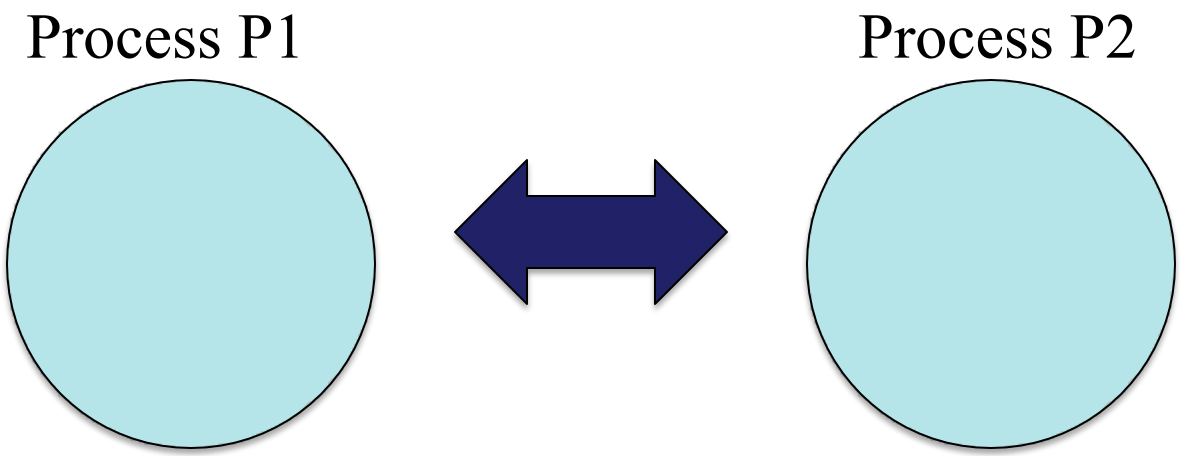
University of Colorado at Boulder

All material by Dr. Rick Han

Recap

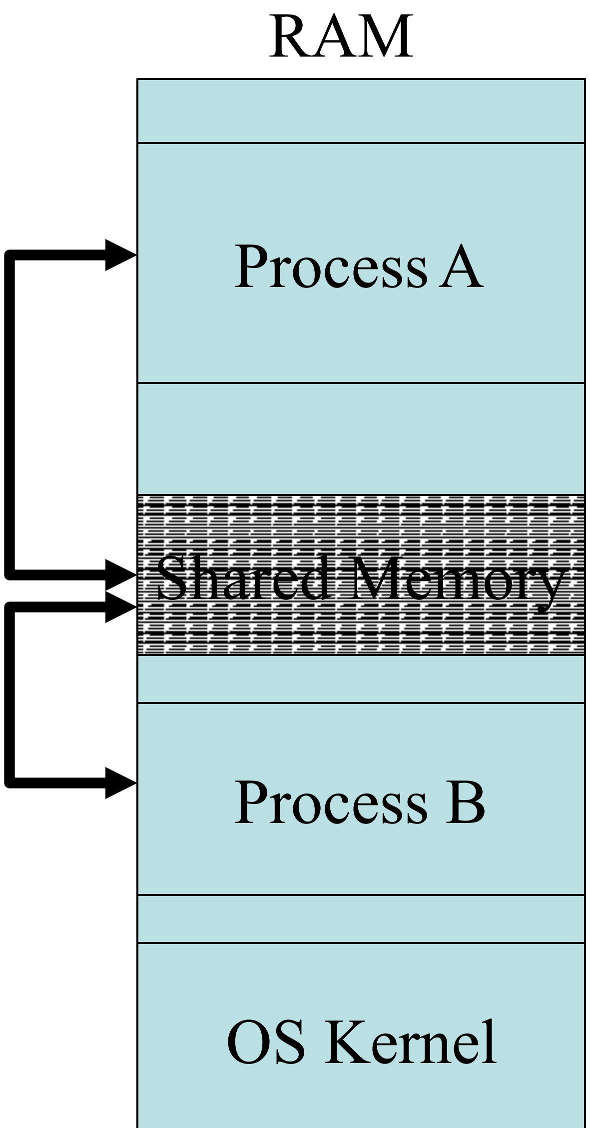
- Process Management
  - PCB State, /proc
  - Creation, Fork/Exec, Deletion
- Threads – unit of execution inside a Process
  - Shares address space (code, data, heap), but has its own register state, stack and PC
  - Faster, smaller, easier
  - Thread-safe code
  - Reentrant code
- User-space threads vs kernel threads

Inter-Process Communication (IPC)



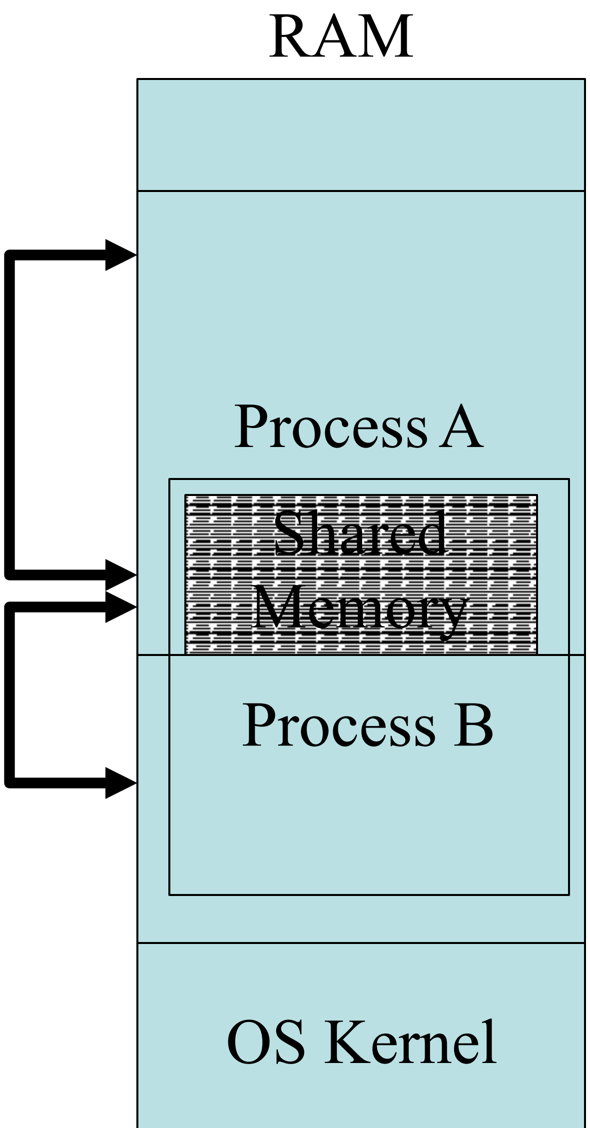
- Motivation
  - an application is split into multiple processes that need to share data
- How do two processes communicate?
  1. Shared Memory
  2. Message Passing

Shared Memory IPC



- OS creates a *shared memory buffer* between processes
- Advantages:
  - allows fast read/write by just using pointers
  - Enables high volume of reading/writing, e.g. moving large files between processes

Shared Memory IPC



- Common implementation is via virtual memory:
  - Page tables of both processes point to same pages in memory!
  - So shared memory is mapped into the address spaces of both processes, rather than being a separate piece
    - In practice, the shared memory is split into pages and scattered throughout main memory

Shared Memory Usage

- shmid = *shmget*(key name, size, flags)
  - Part of the POSIX API that creates a shared memory segment, using a name (key ID)
  - All processes sharing the memory need to agree on the key name in advance.
  - Creates a new shared memory segment if no such shared memory with the same name exists and returns handle to the shared memory.
  - If it already exists, then just return the handle.

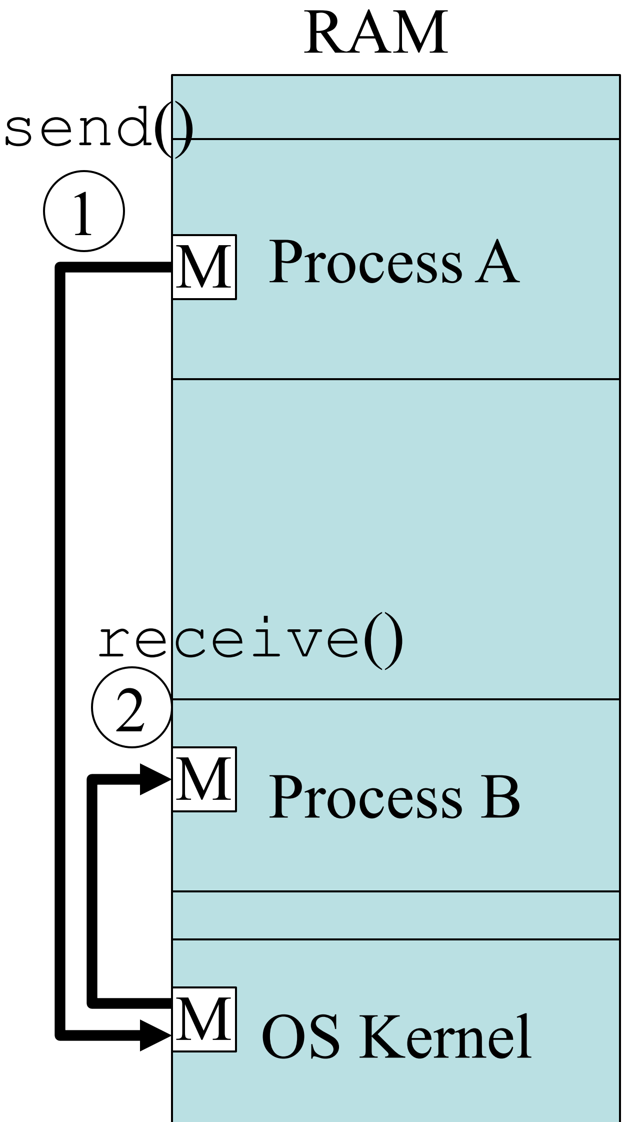
Shared Memory Usage

- shm_ptr = *shmat*(shmid, NULL, 0)
  - to attach a shared memory segment to a process' address space
  - This association is also called binding
  - Reads and writes now just use shm_ptr
- *shmctl*()
  - modify control information and permissions related to a shared memory segment, & to remove a shared memory segment

Shared Memory IPC Limitations

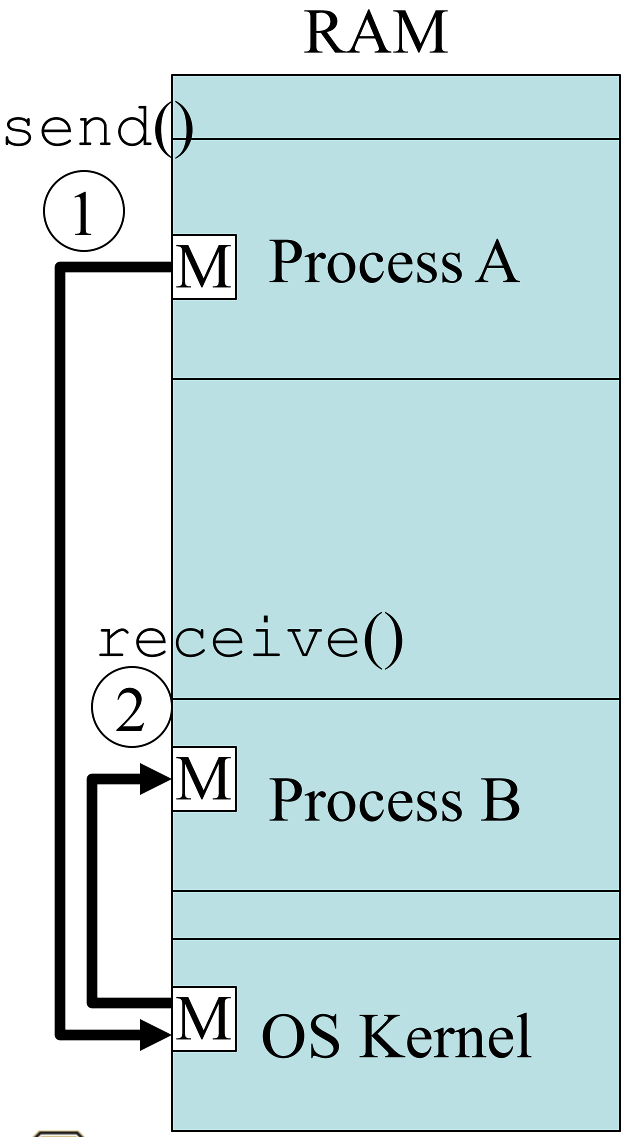
- Problem: shared access to the same memory introduces potential race conditions
  - need to synchronize access
  - Producer-Consumer example
    - if two producers write at the same time to shared memory, then they can overwrite each other's data
    - if a producer writes while a consumer is reading, then the consumer may read inconsistent data

Message Passing IPC



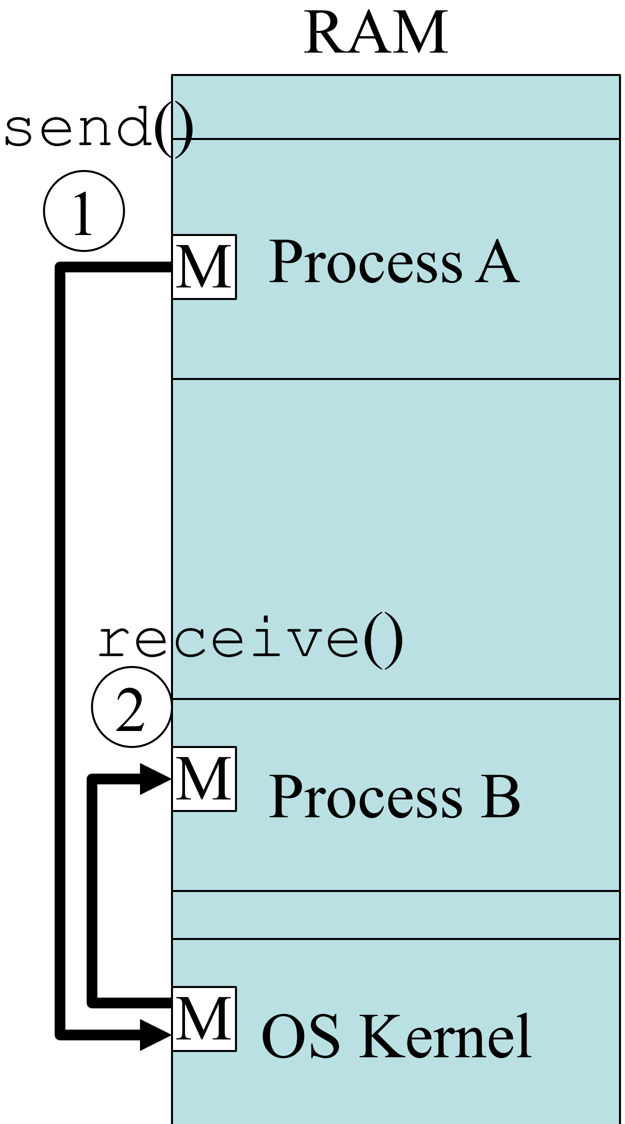
- Used `send()` and `receive()` to communicate messages between processes
- Special "ports" are used for communicating messages
- Typically used to pass small messages
- Indirect message-passing is shown
  - Alternative approach passes messages directly between processes
- can be blocking/synchronous or non-blocking/asynchronous

Message Passing IPC



- Advantage: doesn't require synchronization
  - Blocking send() allows OS to serialize writes, mitigating race conditions
- Disadvantage: Slow
  - OS is invoked via a system call for each IPC operation to pass control signaling and possibly data as well
- Message Passing IPC types: pipes, UNIX-domain sockets, Internet domain sockets, message queues, and remote procedure calls (RPC)

Message Passing IPC



- Message Passing IPC types:
  - UNIX-domain sockets
  - Internet domain sockets
  - Pipes
  - Signals
  - message queues
  - remote procedure calls (RPC)

Using Sockets for UNIX IPC

Sockets are an example of message-passing IPC. Created in UNIX using socket() call:!

```
sd = socket(int domain, int type, int protocol);!
```

socket descriptor

= PF_UNIX for local Unix domain sockets (local IPC)
= PF_INET for Interne sockets (but c an still achieve local communication by specifying localhost address as destination)

= SOCK_STREAM fo r reliable in-order delivery of a byte stream
= SOCK_DGRAM for delivery of discrete messages

= 0 usually to select default protocol associated with a type

Using Sockets for UNIX IPC (2)

- Each communicating process will first create its own socket, usually SOCK_STREAM.
- For UNIX domain sockets (PF_UNIX domain):
  - Used only for local communication only among a computer's processes
  - Emulates reading/writing from/to a file
  - Each process `bind()`'s its socket to a filename:

```
bind(sd, (struct sockaddr *)&local, length);!
```

socket descriptor

data structure containing unique unused file name, e.g. "/users/rick/myipcsocketfile"

Using Sockets for UNIX IPC (3)

- Usually, one process acts as the server, and the other processes connect to it as clients!

Server code:

```
sd = socket(PF_UNIX,SOCK_STREAM,
   0)!
bind(sd,…)!
```

bind and connect must use same file name! !

```
listen()
```
for connect requests

```
sd2 = accept()
```
a connect request

```
!
recv(sd2,…)/send(sd2,…)!
```

IPC

Client code:

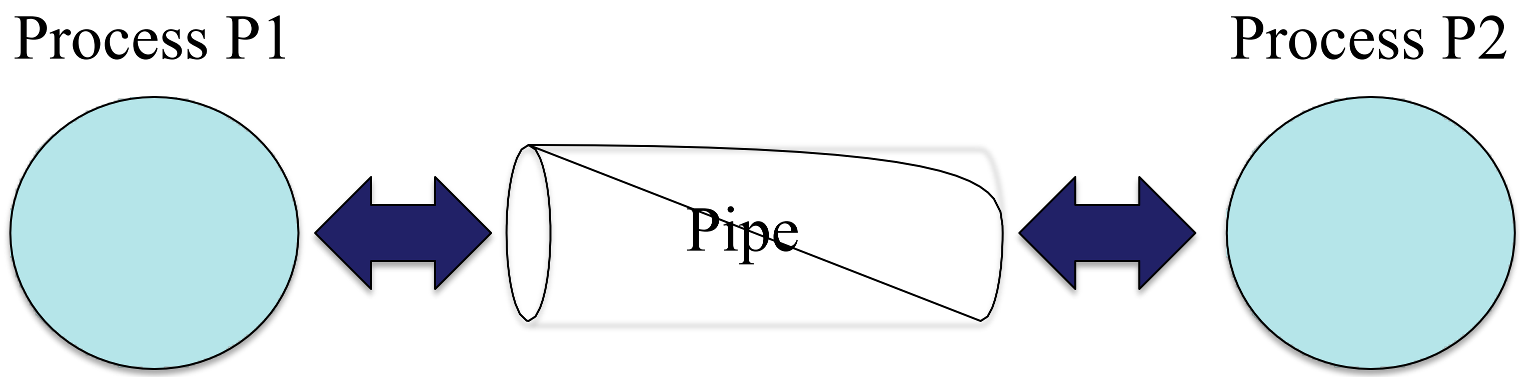
```
sd = socket(PF_UNIX,
   SOCK_STREAM,0)!
!
connect(sd,…)
```
to server

```
!
!
!
recv(sd,…)/send(sd,…)!
```

IPC via Internet domain sockets

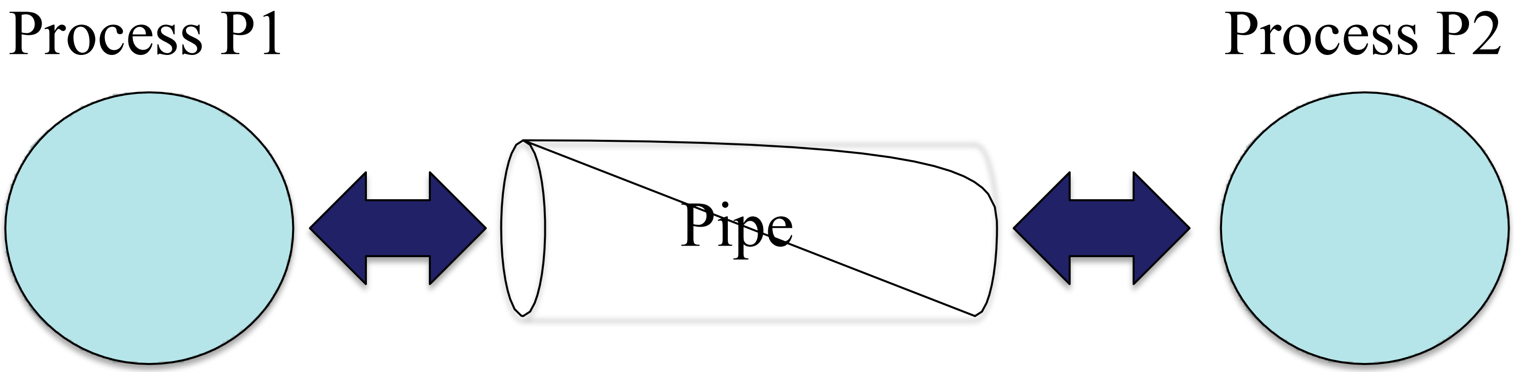
- Similar to Unix domain sockets,
  - Configure the socket with domain PF_INET
  - Set destination to *localhost* (say 127.0.0.1) instead of the usual remote Internet IP address
  - Choose *a well-known* port # that is shared between processes, i.e. P1 and P2 know this port # in advance
    - similar to a well-known file name
  - Both processes then send() and receive() messages via this port and socket
  - Arguably more portable than UNIX-domain sockets
  - May be slower than UNIX-domain sockets because messages traverse network's layered stack

IPC via Pipes



- Process 1 writes into one end of the pipe, & process 2 reads from other end of the pipe
  - e.g. "ls | more"
  - Form of IPC similar to message-passing but data is viewed as a stream of bytes rather than discrete messages
  - was one of UNIX's original forms of IPC

IPC via Pipes



- essentially FIFO buffers accessed like file I/O
  - so standard read()/write() for files can be used
  - Asynchronous/non-blocking send() and blocking/ synchronous receive()
- Ordinary pipes are one-way
  - To create two-way pipes, use two opposite one-way pipes

Parent-Child Pipe Example

- Parent process uses `pipe`() system call to create pipe

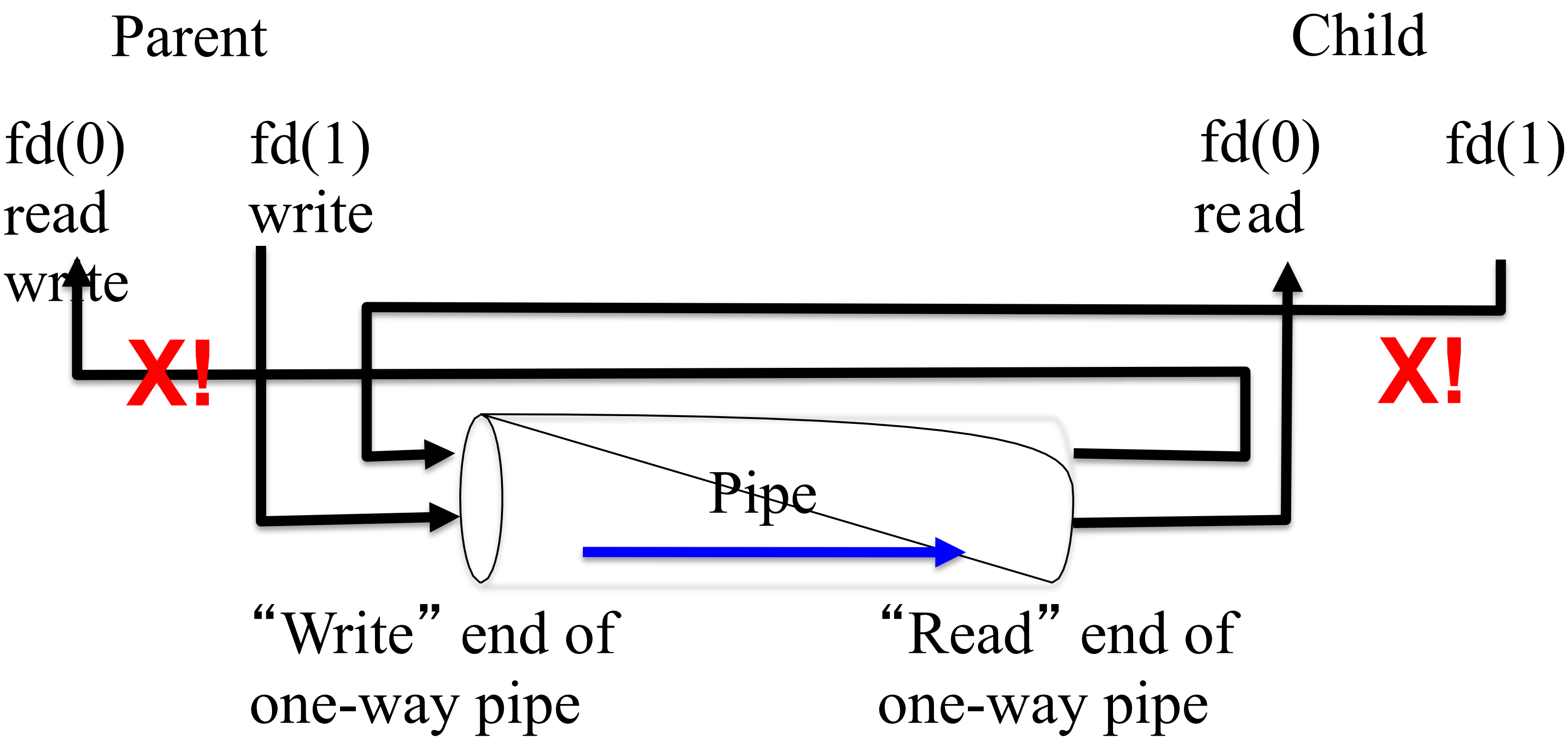
```
int pid;!
int piped[2];!
!
pipe(piped);!
pid = fork();!
if (pid==0) { /* child */!
    /* childs blocks on read */!
    read(piped[0], readdata, length);!
} elseif (pid>0) {  /* parent */!
    /* parent writes data to child */!
    write(piped[1], writedata, length);!
}!
```

piped[0] is file descriptor to read end of the pipe piped[1] is file descriptor to write end of pipe

Once there are *length* bytes, read returns, so as the parent sender streams bytes, the child reader can process them

Send message to child

IPC via Pipes (3)



- Chapter 3 textbook example is more detailed, e.g. closes the unused write fd for child and closes the unused read fd for parent

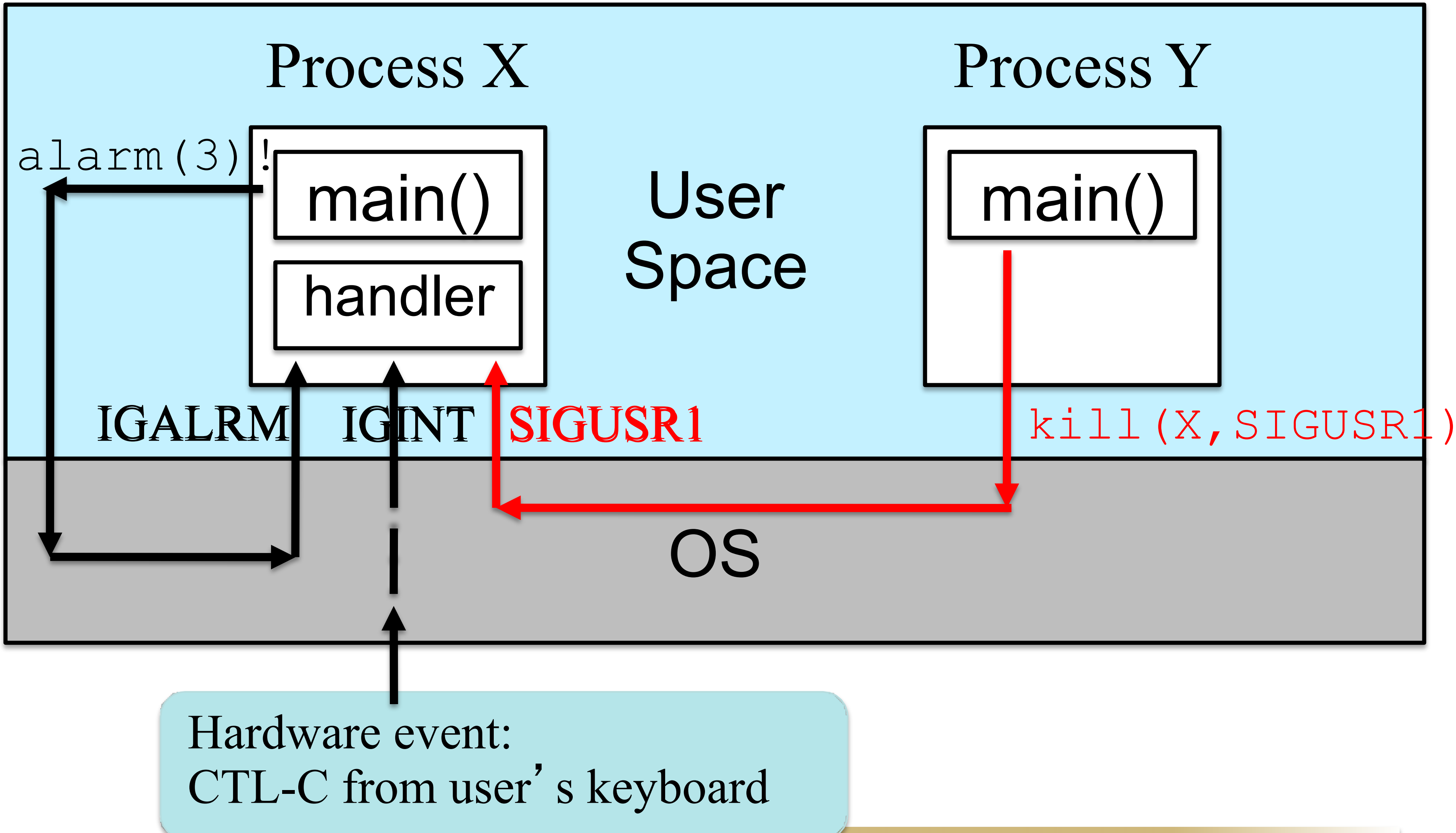
Named Pipes

- Traditional one-way or anonymous pipes only exist transiently between the two processes connected by the pipe
  - As soon as these processes complete, the pipe disappears
- Named pipes persist across processes
  - Operate as FIFO buffers or files, e.g. created using mkfifo(unique_pipe_name) on Unix
  - Different processes can attach to the named pipe to send and receive data
  - Need to explicitly remove the named pipe
  - See textbook for more info on named pipes

Signals as (Limited) IPC

- Signals allow a small numerical code to be sent to a process
  - This interrupts the normal control flow of a process, and is called exceptional control flow
    - Must register a *handler* to catch this signal using signal() or sigaction()
  - Usually OS-to-process communication
  - But some signals are useful in process-to-process communication
    - e.g. SIGUSR1, SIGCHILD, SIGKILL, etc.
  - No data can be sent, only the code, so this is very limited IPC

Signals as (Limited) IPC



Signals

- OS-to-process:
  - Kernel sets the numerical code in a process variable, then wakes the process up to handle the signal
- Process-to-process
  - Call `kill(process_id, signal_num)`!
    - e.g., `kill(Y,SIGUSR1)` sends a SIGUSR1 signal to process Y, which will know how to interpret this signal
    - Call still goes through OS, not directly from process to process.
  - A process can send a signal to itself using a library call like alarm()

Signals

- Signals expose low-level hardware exceptions to user processes
  - May be efficient to get a callback after a read() has completed for example
    - allows process to keep executing without polling to check for completion
  - Alternatives:
    - A process blocks on a read() and is informed of read's completion only when it is unblocked – inefficient!
    - A process polls a read() – also less efficient

Linux/UNIX Signals

Number	Name/Type	Event
2	SIGINT	Interrupt from keyboard (Ctrl-C)
8	SIGFPE	Floating point exception (arith. error)
9	SIGKILL	Kill a process
10, 12	SIGUSR1, SIGUSR2	User-defined signals
11	SIGSEGV	invalid memory ref (seg fault)
14	SIGALRM	Timer signal from alarm function
29	SIGIO	I/O now possible on descriptor

Signals

- Signals expose low-level hardware exceptions to user processes
  - May be efficient to get a callback after a read() has completed for example
    - allows process to keep executing without polling to check for completion
  - Alternatives:
    - A process blocks on a read() and is informed of read's completion only when it is unblocked – inefficient!
    - A process polls a read() – also less efficient

Signals and Race Conditions

- Signals are an *asynchronous* signaling mechanism in UNIX
  - A process never knows when a signal will occur
  - Its execution can be interrupted at any time.
  - A process must be written to handle this asynchrony. Otherwise, could get race conditions.

```
int global=10;!
handler(int signum) {!
global++;!
}!
main() {!
signal(SIGUSR1,handler);!
while(1) {global--;}!
}!
!
```

Both the main control flow and exceptional control flow change the global variable – could have a *race condition*

Signals and Race Conditions

- In addition, if there are multiple signals, can have a race conditions
  - i.e. if a signal handler is processing signal S1 but is interrupted by another signal S2, then could have a race condition inside the handler.
    - In the previous example, we'd have global++ happening in two handlers in rapid succession, could lead to unpredictable results.
  - The solution is to *block* other signals while handling the current signal.
    - Use `sigprocmask`() to selectively block other signals
    - A blocked signal is pending
    - There can be at most one pending signal per signal type, so signals are not queued

Supplementary Slides

Signals

- Receiving signals:
  - when a kernel is returning from some exception handler, it checks to see if there are any pending signals for a process before passing control to the process
  - The user may register a signal `handler()` via the `signal()` function. If no handler is registered, then default action is typically termination
  - `signal(signum, handler)` function is used to change the action associated with a signal
    - if handler is SIG_IGN, then signals of type signum are ignored
    - if handler is SIG_DFL, then revert to default action, usually termination
    - otherwise if there is a user-defined function `handler`, then call it to handle the signal.

Signals

- Invocation of the signal handler is called *catching the signal.* We use this term interchangeably with *handling the signal*
- the user-specified signal handler executes in the context of the affected process
  - The kernel calls the user-specified handler, and passes control to user space. When the handler is done (call returns), control returns to the kernel.
  - Note: the next time the process executes, it will resume back at the instruction where the process was originally interrupted/signaled

Signals

- Portable signal handling: `sigaction`() is a standard POSIX API for signal handling
  - allows users on Posix-compliant systems such as Linux and Solaris to specify the signal-handling semantics they want, e.g. whether sys call is aborted or restarted
  - Is more advanced/expressive than the `signal`() function
- `sigaction`(signum, struct sigaction*act, struct sigaction *oldact)
- each struct sigaction can define a handler, e.g. action.sa_handler = handler;
  - use sigaction() to define the handler
  - Also need to set up the timer – could use *setitimer* instead of alarm. A SIGALRM is delivered when timer expires.

Signals

- More generally, when a process catches a signal of type signum=k, the handler installed for signal k is invoked with a single integer argument set to k
- This argument allows the same handler function to catch different types of signals.

Signaling Example

- A process can send SIGALRM signals to itself by calling the alarm function
  - alarm(T seconds) arranges for kernel to send a SIGALRM signal to calling process in T seconds
  - see code example next slide
    - #include<signal.h>
    - uses `signal` function to install a signal handler function that is called asynchronously, interrupting the infinite while loop in main, whenever the process receives a SIGALRM signal
    - When handler returns, control passes back to main, which picks up where it was interrupted by the arrival of the signal, namely in its infinite loop

Signaling Example

```
#include <signal.h>
int beeps=0;

void handler(int sig) {
    if (beeps<5) {
        alarm(3);
        beeps++;
    } else {
        printf("DONE\n");
        exit(0);
    }
}

int main() {
    signal(SIGALRM, handler);
    alarm(3);

    while(1) { ; }
    exit(0);
}
```

Signal handler, passed signal #

cause next SIGALRM to be sent to this process in 3 seconds. Assume that SIGALRM is the only signal handled. Otherwise, would need a *case* statement in handler for other signals.

register signal handler

cause first SIGALRM to be sent to this process in 3 seconds

infinite loop that gets interrupted by signal handling

Signals

- a process can selectively block the receipt of certain signals using the function sigprocmask()
  - when a signal is blocked, it can be delivered, but the resulting pending signal will not be received until the process unblocks the signal. Like masking interrupts.
- A signal that has been sent but not yet received is called a *pending* signal.
  - Use `sigpending`() to get a list of pending signals.
  - At any point in time, there can be at most one pending signal of a particular type (signal number)
  - A pending signal is received at most once

Signals

- For each process, the kernel maintains
  - the set of pending signals in the `pending` bit vector, and
  - the set of blocked signals in the `blocked` bit vector

Signals

- Handling multiple signals
  - choose to handle the lower number signals first
  - pending signals are blocked,
    - e.g. if a 2nd SIGINT is received while handling 1st SIGINT, the 2nd SIGINT becomes pending and won't be received until after the handler returns
  - pending signals are not queued
    - there can be at most one pending signal of type k, e.g. if a 3rd SIGINT arrives while the 1st SIGINT is being handled and the 2nd SIGINT is already pending, then the 3rd SIGINT is dropped
  - system calls can be interrupted
    - slow system calls that are interrupted by signal handling may not resume in some systems, and may return immediately with an error