



slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

CSCI 3753 Operating Systems Summer 2020

Christopher Godley

PhD Student

Department of Computer Science

University of Colorado Boulder



University of Colorado
Boulder



Inter-Process Communication

Inter-Process Communications (communications *between* processes)

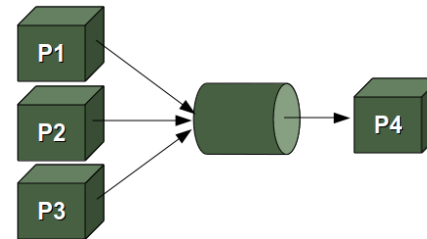
- Signals / Interrupts

- Notifying that an event has occurred



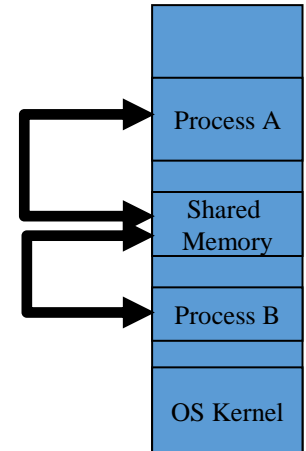
- Message Passing

- Pipes
- Sockets



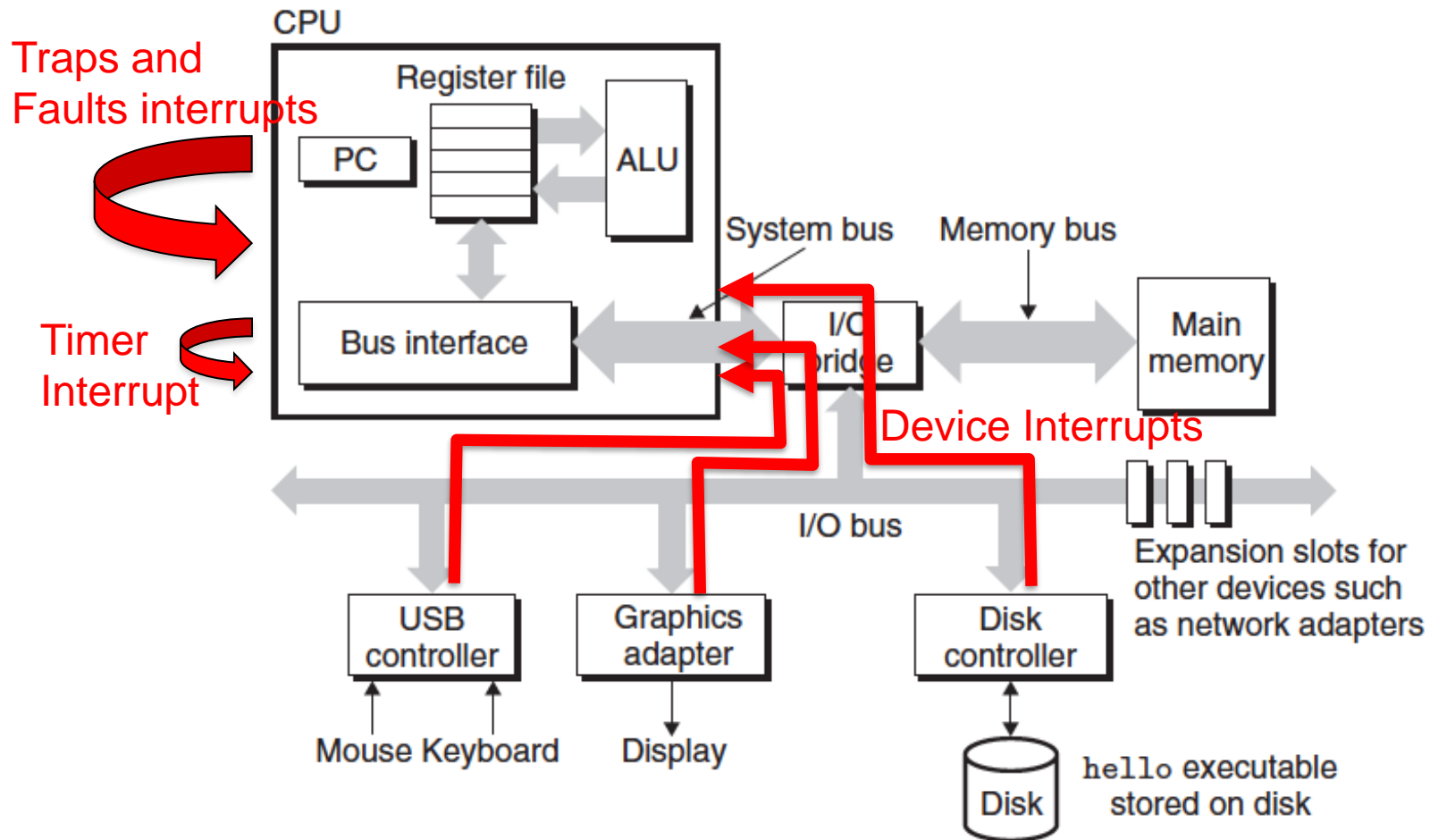
- Shared Memory

- Race conditions
- Synchronization



- Remote Procedure Calls

Linux Signals and Interrupts



Linux Signals and Interrupts

If a process ***catches*** a signal, it means that it includes code that will take appropriate action when the signal is received.

If the signal is not caught by the process, the kernel will take **default action** for the signal.

We will revisit these when we discuss process synchronization

Signal Name	Number	Description
SIGHUP	1	Hangup (POSIX)
SIGINT	2	Terminal interrupt (ANSI)
SIGQUIT	3	Terminal quit (POSIX)
SIGILL	4	Illegal instruction (ANSI)
SIGTRAP	5	Trace trap (POSIX)
SIGIOT	6	IOT Trap (4.2 BSD)
SIGBUS	7	BUS error (4.2 BSD)
SIGFPE	8	Floating point exception (ANSI)
SIGKILL	9	Kill(can't be caught or ignored) (POSIX)
SIGUSR1	10	User defined signal 1 (POSIX)
SIGSEGV	11	Invalid memory segment access (ANSI)
SIGUSR2	12	User defined signal 2 (POSIX)
SIGPIPE	13	Write on a pipe with no reader, Broken pipe (POSIX)
SIGALRM	14	Alarm clock (POSIX)
SIGTERM	15	Termination (ANSI)

Signal Name	Number	Description
SIGSTKFLT	16	Stack fault
SIGCHLD	17	Child process has stopped or exited, changed (POSIX)
SIGCONT	18	Continue executing, if stopped (POSIX)
SIGSTOP	19	Stop executing(can't be caught or ignored) (POSIX)
SIGTSTP	20	Terminal stop signal (POSIX)
SIGTTIN	21	Background process trying to read, from TTY (POSIX)
SIGTTOU	22	Background process trying to write, to TTY (POSIX)
SIGURG	23	Urgent condition on socket (4.2 BSD)
SIGXCPU	24	CPU limit exceeded (4.2 BSD)
SIGXFSZ	25	File size limit exceeded (4.2 BSD)
SIGVTALRM	26	Virtual alarm clock (4.2 BSD)
SIGPROF	27	Profiling alarm clock (4.2 BSD)
SIGWINCH	28	Window size change (4.3 BSD, Sun)
SIGIO	29	I/O now possible (4.2 BSD)
SIGPWR	30	Power failure restart (System V)

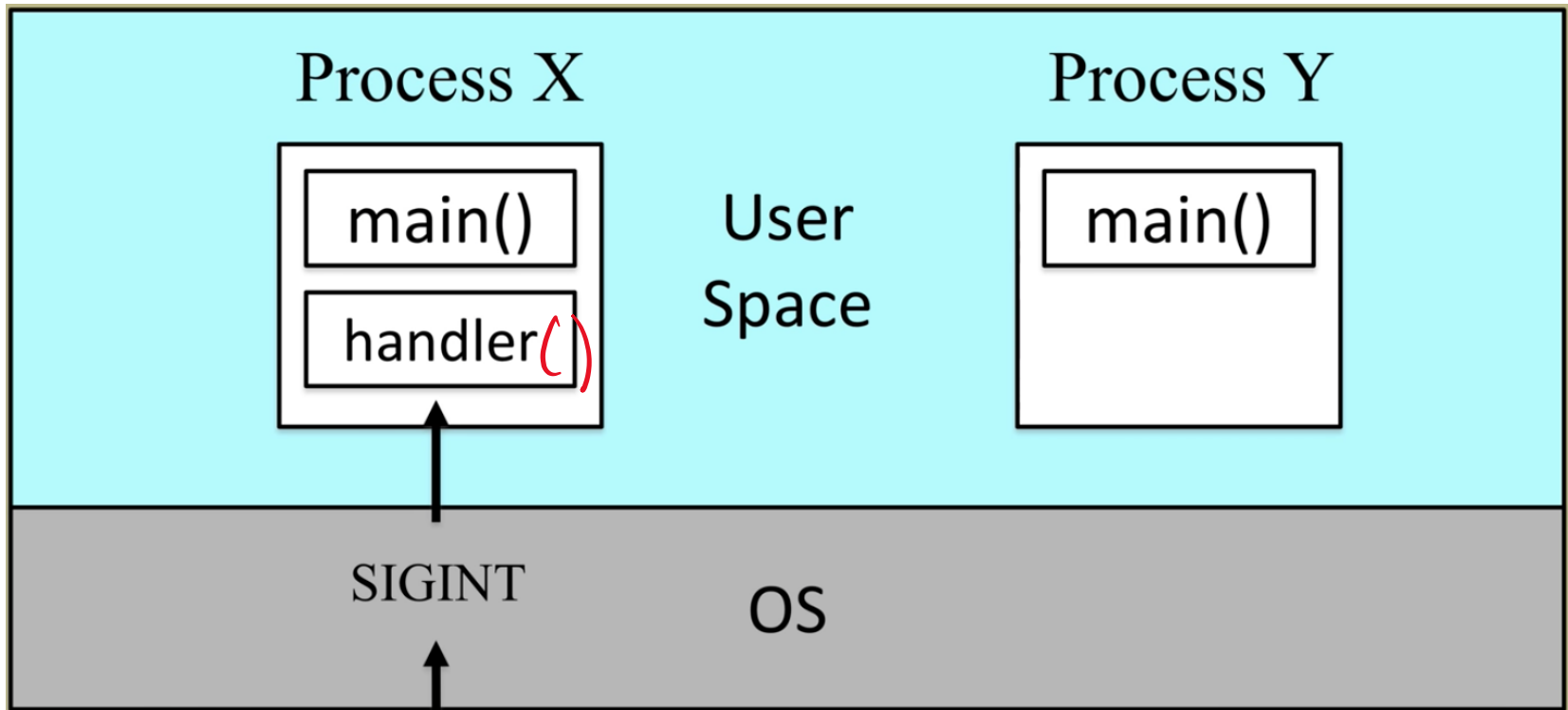
There is not much information that can be passed through an interrupt, only that an event has occurred

Commonly used 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

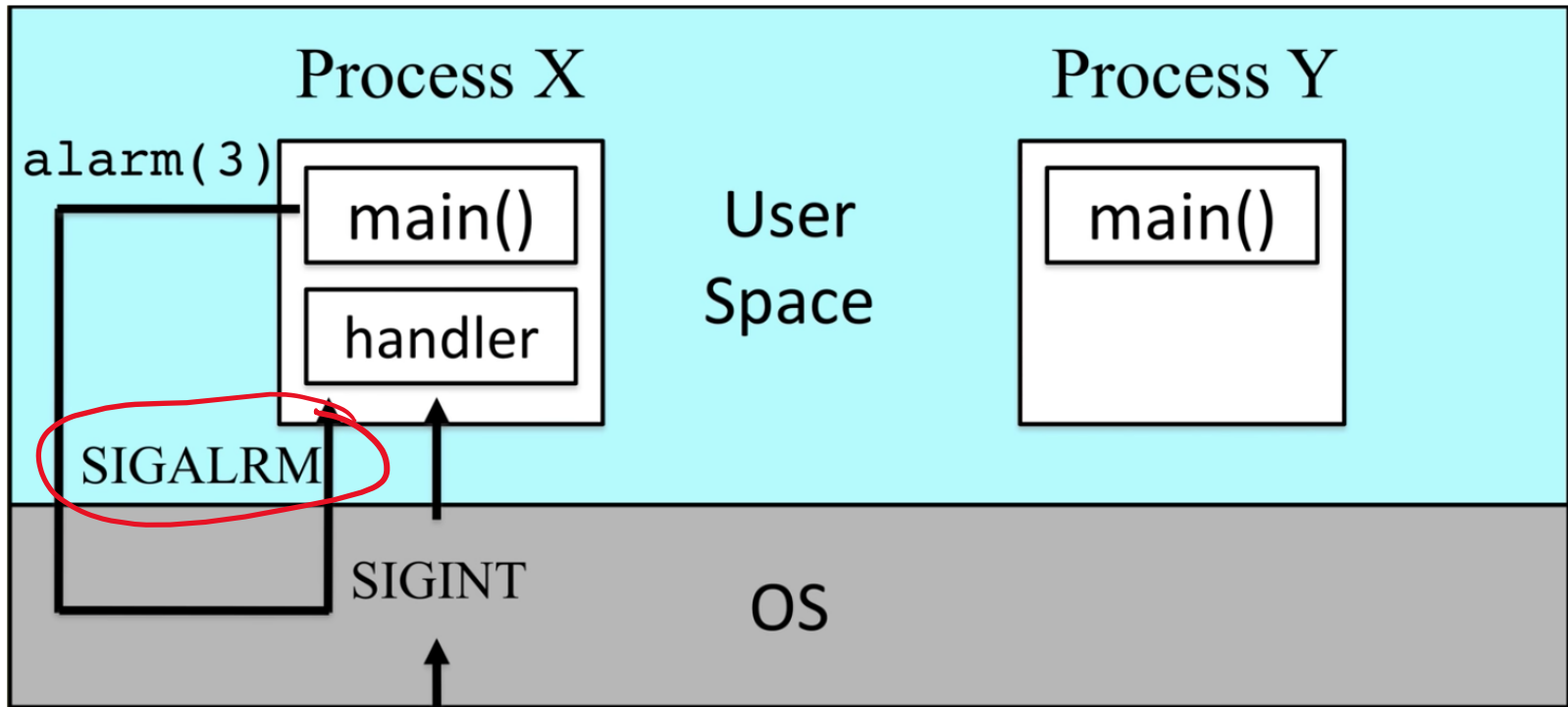


Signals allow limited IPC



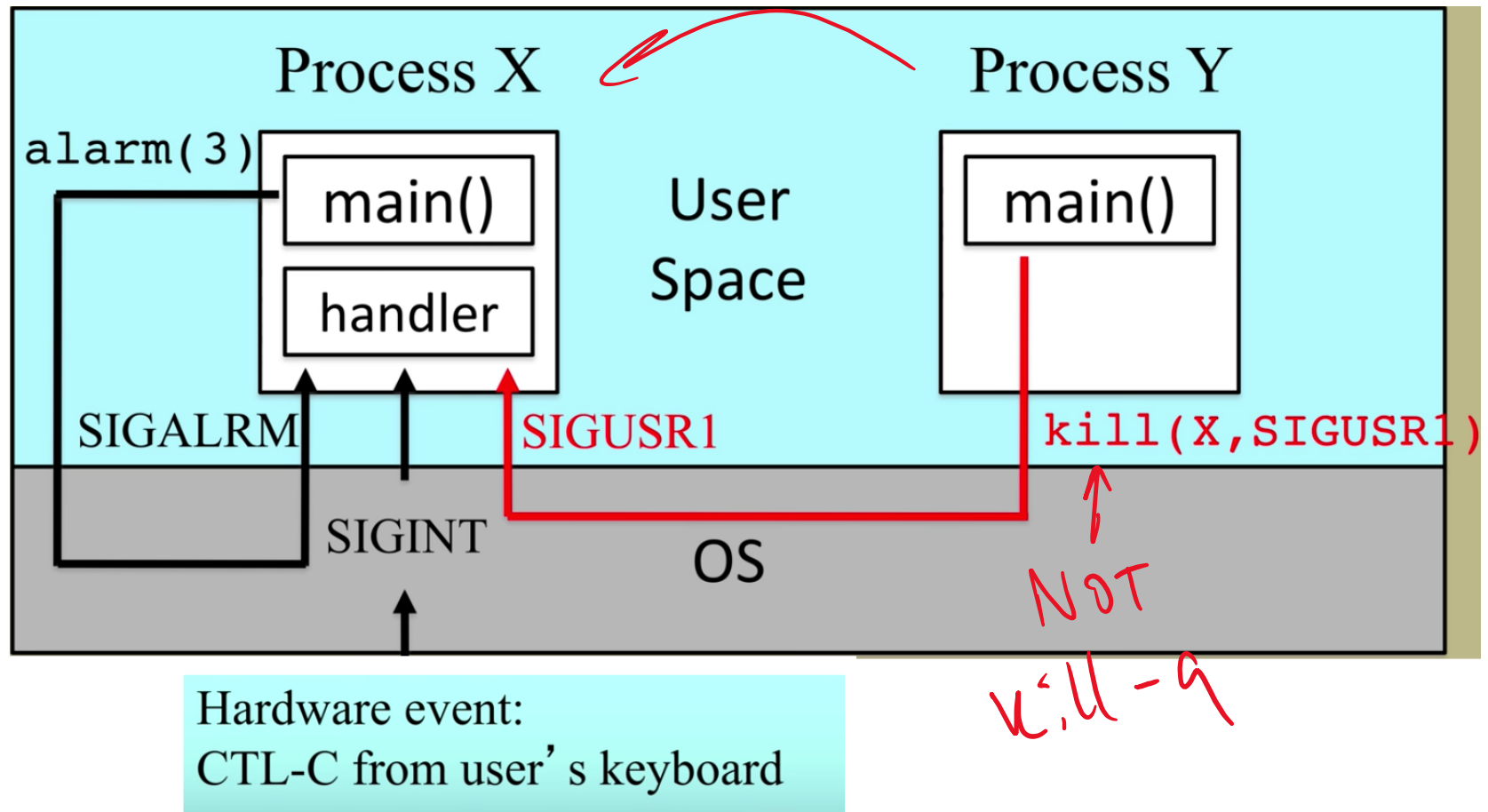
Hardware event:
CTL-C from user's keyboard

Signals allow limited IPC

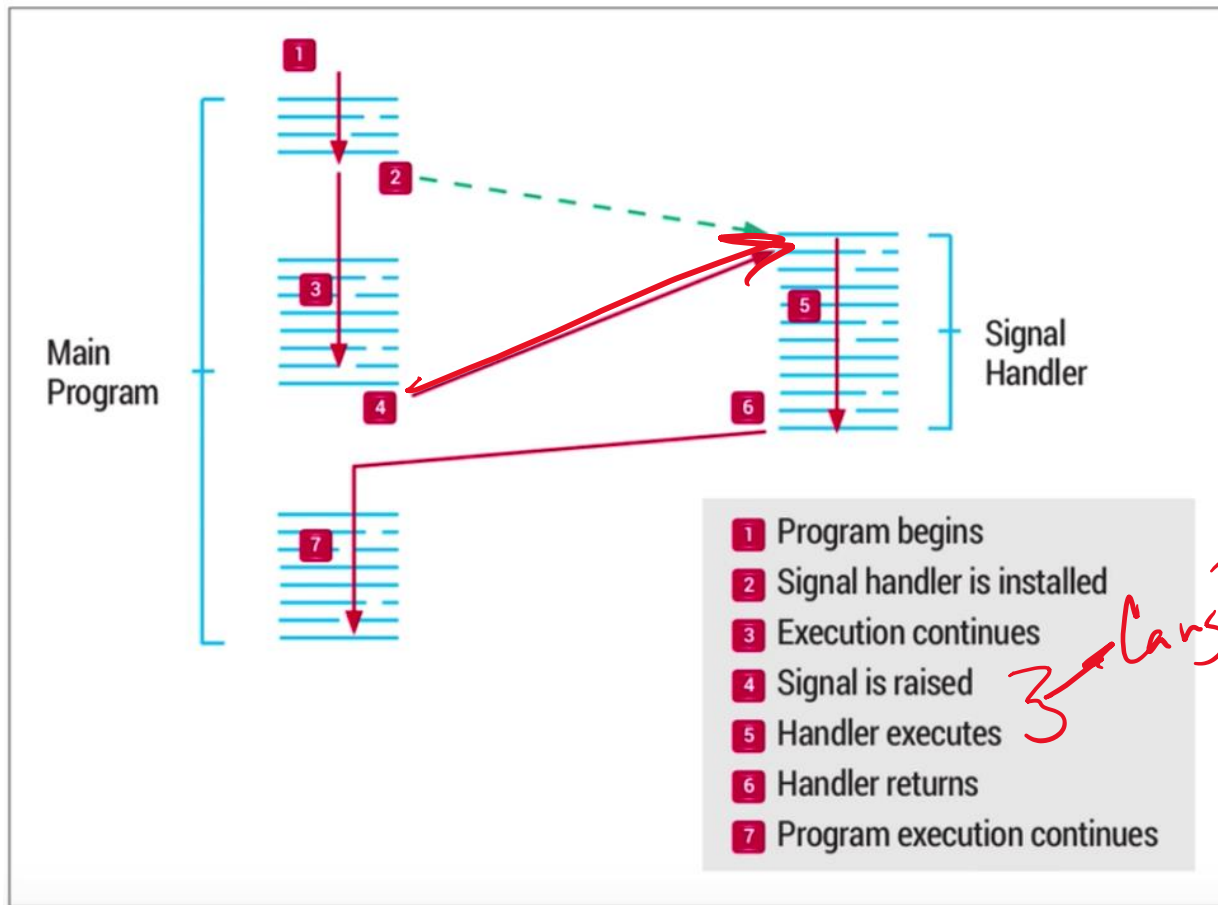


Hardware event:
CTL-C from user's keyboard

Signals allow limited IPC



Signal Handler



The handler resides in the same code base as the program.

It is usually a routine to be called when the signal is seen.

Signals

- Kernel-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 kernel, not directly from process to process.

Signals

- Kernel-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 kernel, not directly from process to process.
 - A process can send a signal to itself using a library call like `alarm()`



Signals and Race Conditions

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

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



Blocking versus Non-Blocking I/O

- Blocking system call
 - process put on wait queue until I/O read or write completes
 - I/O command succeeds completely or fails
- Non-blocking system call
 - a write or read returns immediately with partial number of bytes transferred (possibly zero),
 - e.g. keyboard, mouse, network sockets
 - makes the application more complex
 - not all the data may have been read or written in single call
 - have to add additional code to handle this, like a loop



Synchronous versus Asynchronous

- Synchronous will make the request I/O and not continue until the command is completed
 - often synchronous and blocking are used interchangeably
- Asynchronous returns immediately (like non-blocking)
 - often asynchronous and non-blocking are used interchangeably
 - but in asynchronous write I/O, at some later time, the full number of bytes requested is transferred
 - subtle difference with non-blocking definition
 - can be implemented using signals and handlers

Signals and Race Conditions

- Multiple signals can also have a race condition
 - signal handler is processing signal S1 but is interrupted by another signal S2
 - 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

Signal 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);
}
```

Register signal handler

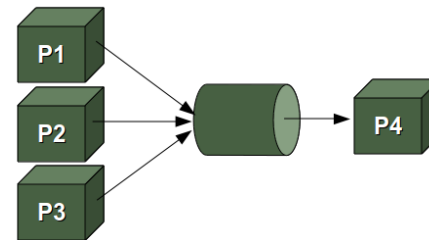
Cause first SIGALRM to be sent
to this process in 3 seconds

Infinite loop that get interrupted by signal
handling

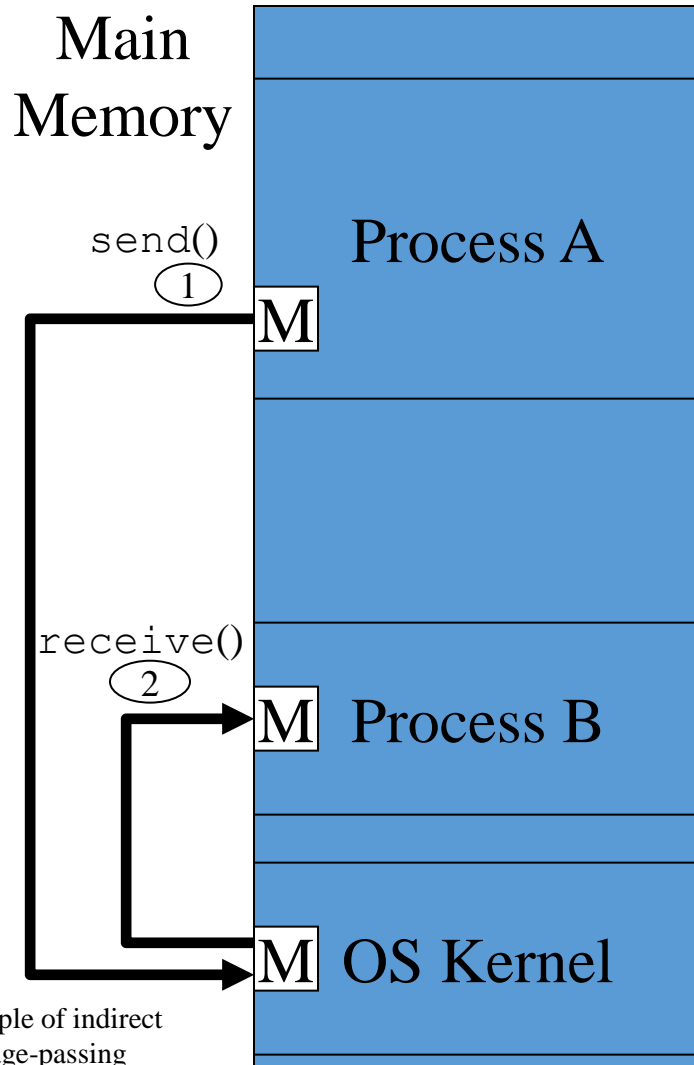


Inter-Process Communications (communications *between* processes)

- Signals
- Message Passing
 - Pipes
 - Sockets



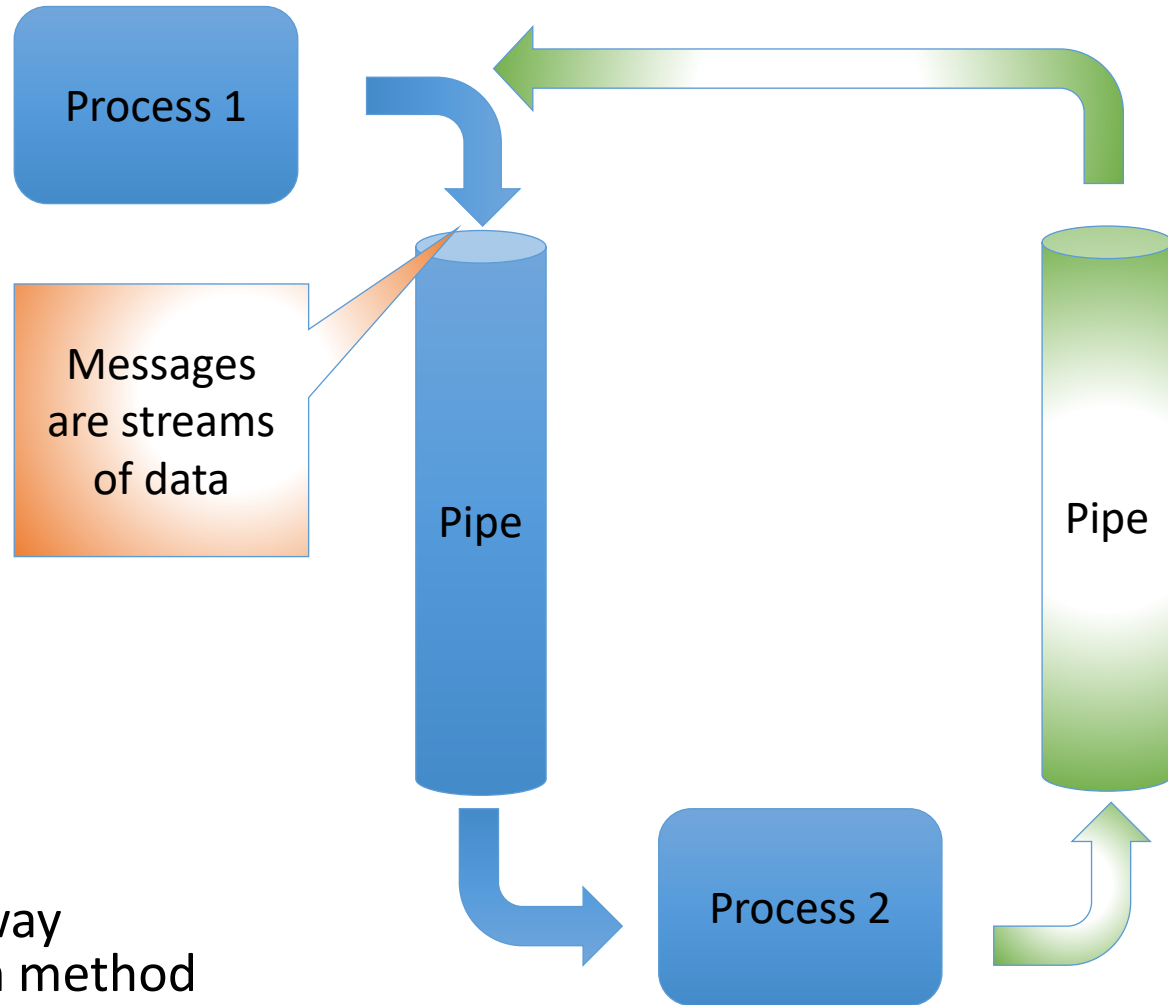
IPC Message Passing



- `send()` and `receive()` can be blocking/synchronous or non-blocking/asynchronous
- Used to pass small messages
- Advantage: OS handles synchronization
- Disadvantage: Slow
 - OS is involved in each IPC operation for control signaling and possibly data as well
- Message Passing IPC types:
 - Pipes
 - UNIX-domain sockets
 - Internet domain sockets
 - message queues
 - remote procedure calls (RPC)



IPC via Pipes



- Pipes are one way communication method

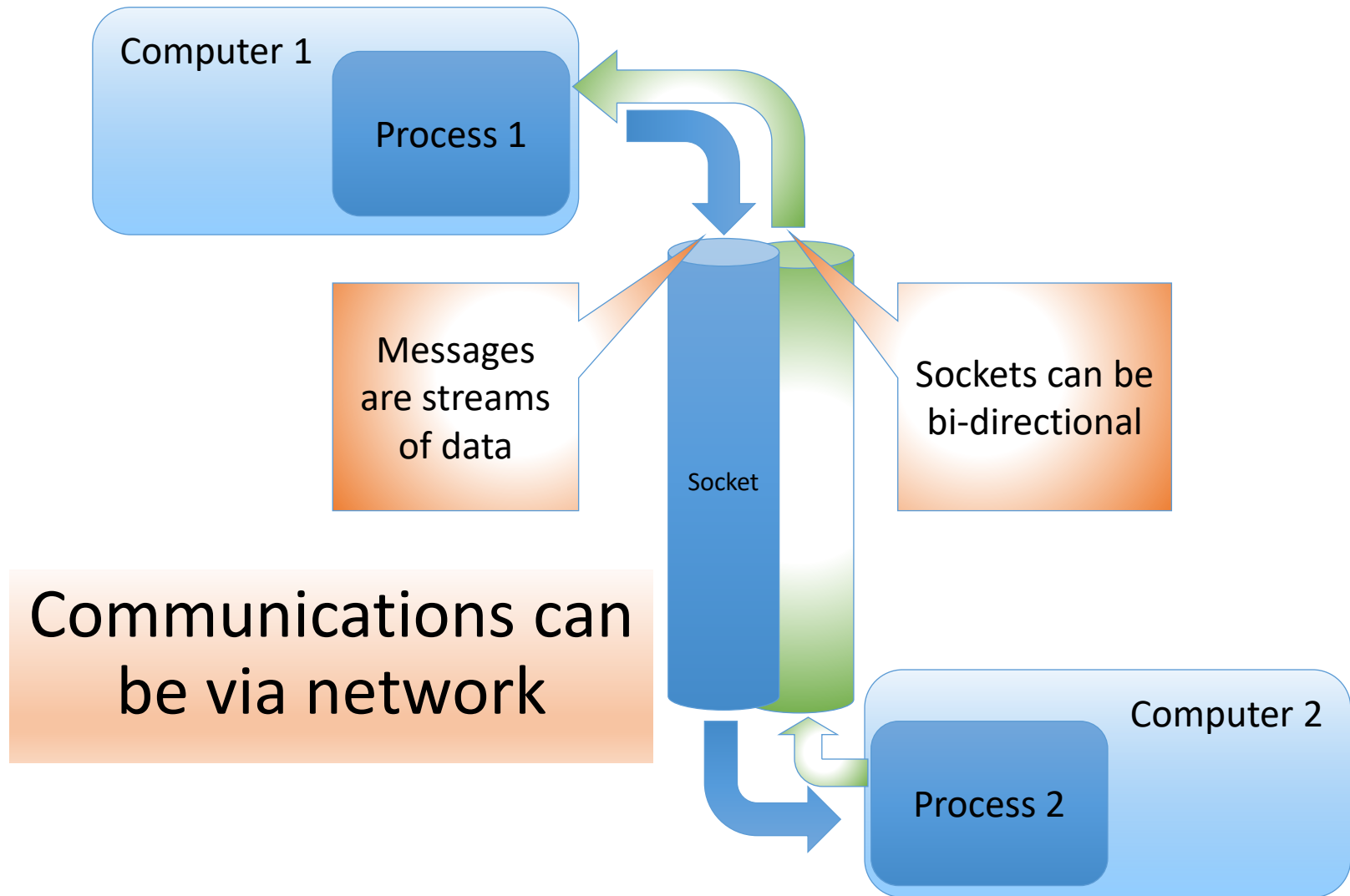


Named Pipes

- 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



IPC via Sockets



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 sockets
PF_INET for remote sockets

↖
0 to select default protocol
associated with a type


↘
SOCK_STREAM for reliable in-order delivery of a byte stream
SOCK_DGRAM for delivery of discrete messages

Using Sockets for UNIX IPC (2)


- 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/dave/my_ipc_socket_file"`



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, ...)
```

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

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

```
recv(sd2, ...) / send(sd2, ...)
```

Client code:

```
sd = socket(PF_UNIX, SOCK_STREAM, 0)
```

```
connect(sd, ...) to server
```

```
recv(sd, ...) / send(sd, ...)
```

bind and connect must
use same file name!

IPC



Inter-Process Communications (communications *between* processes)

- Signals
- Message Passing
 - Pipes
 - Sockets
- Remote Procedure Calls

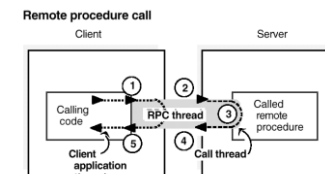
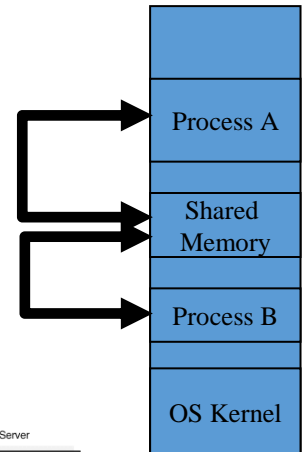
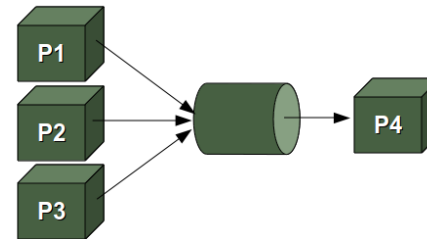
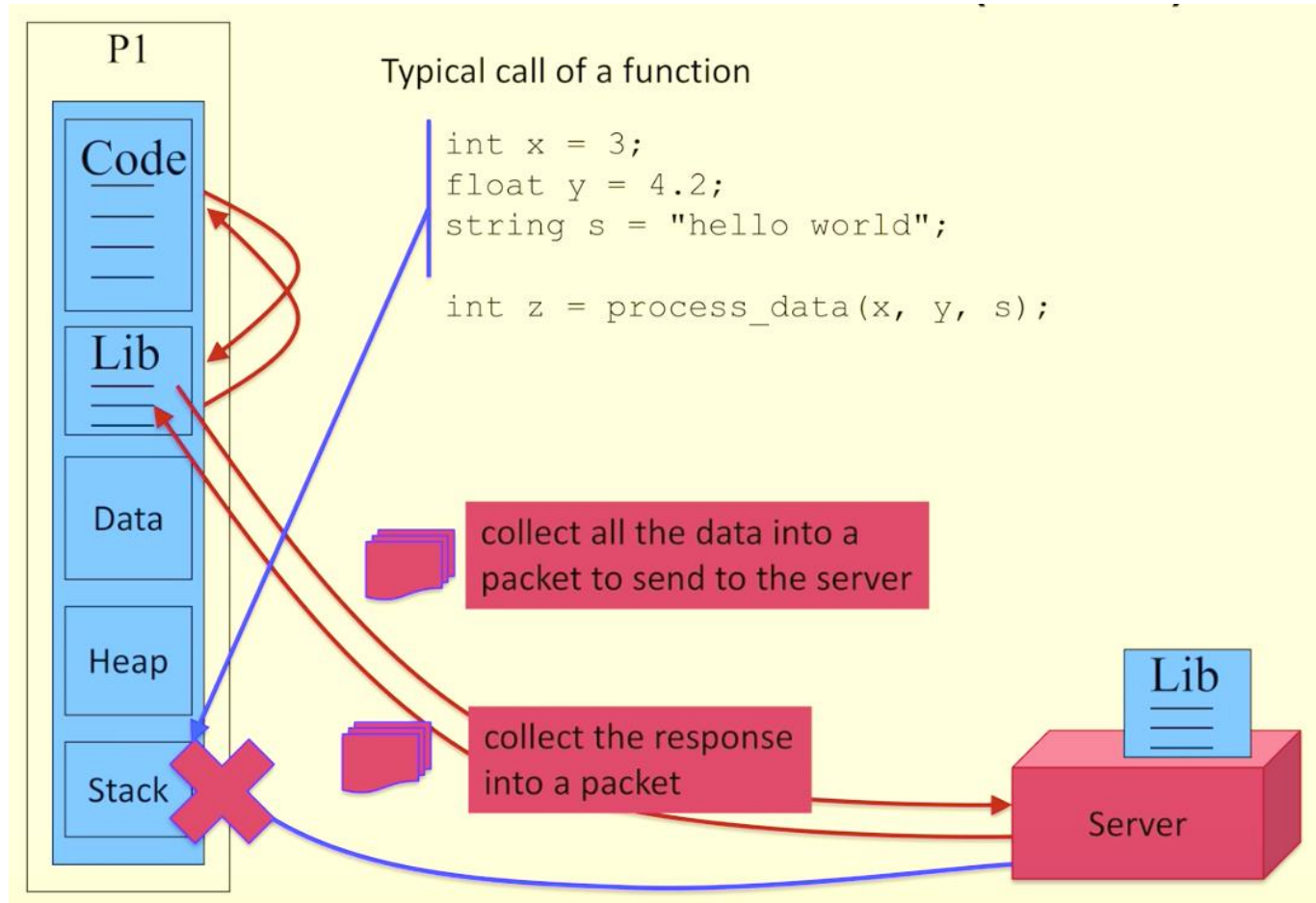


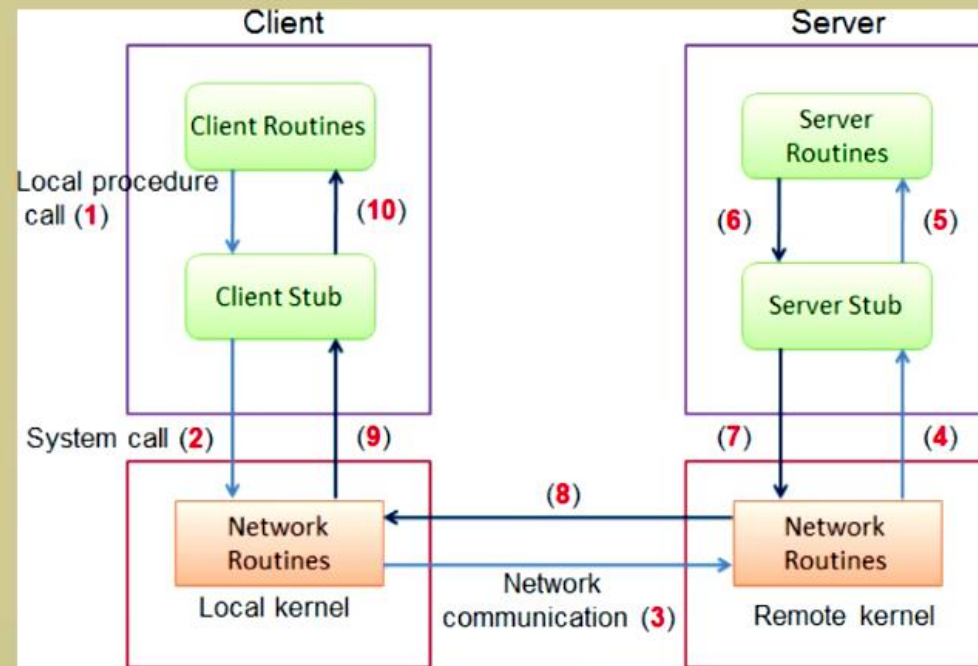
Figure 6-1 Execution Phases of an RPC Thread

Remote Procedure Call (RPC)



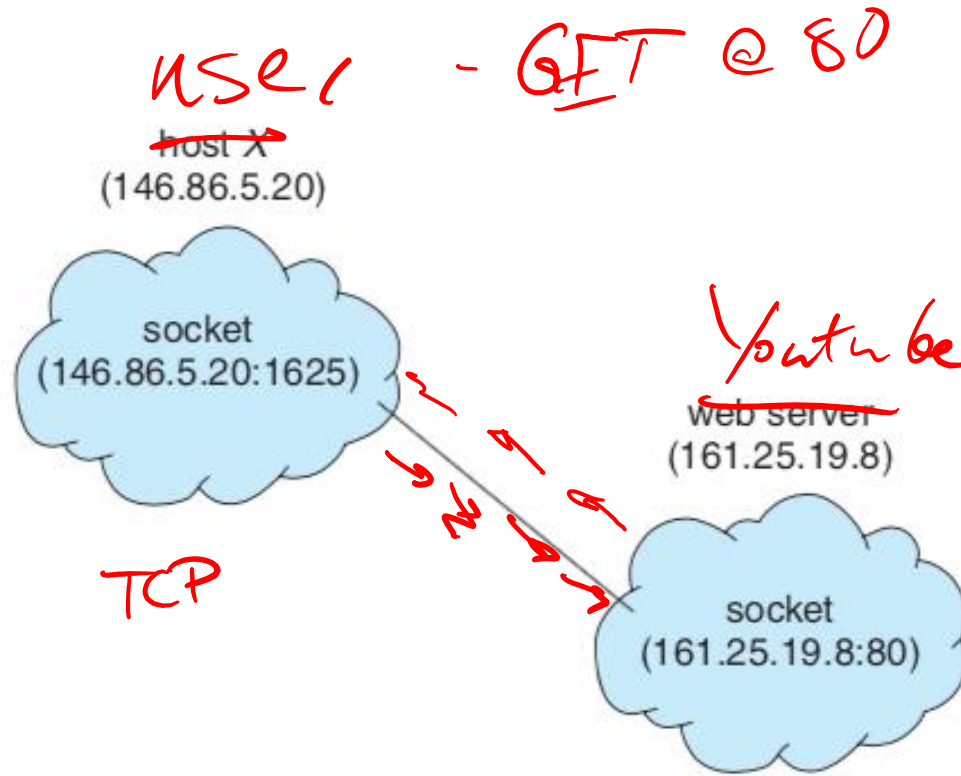
Remote Procedure Call (RPC)

1. Client makes a call to a function and passes parameters
2. The client has linked a stub for the function. This stub will marshal (packetize) the data and send it to a remote server
3. The network transfers the information to a server
4. A service listening to the network receives a request
5. The information is unmarshalled (unpacked) and the server's function is called
6. The results are returned to be marshalled into a packet
7. The network transfers the packet is sent back to the client
8. TCP/IP used to transmit packet



Example: Client-Server system

- Web server

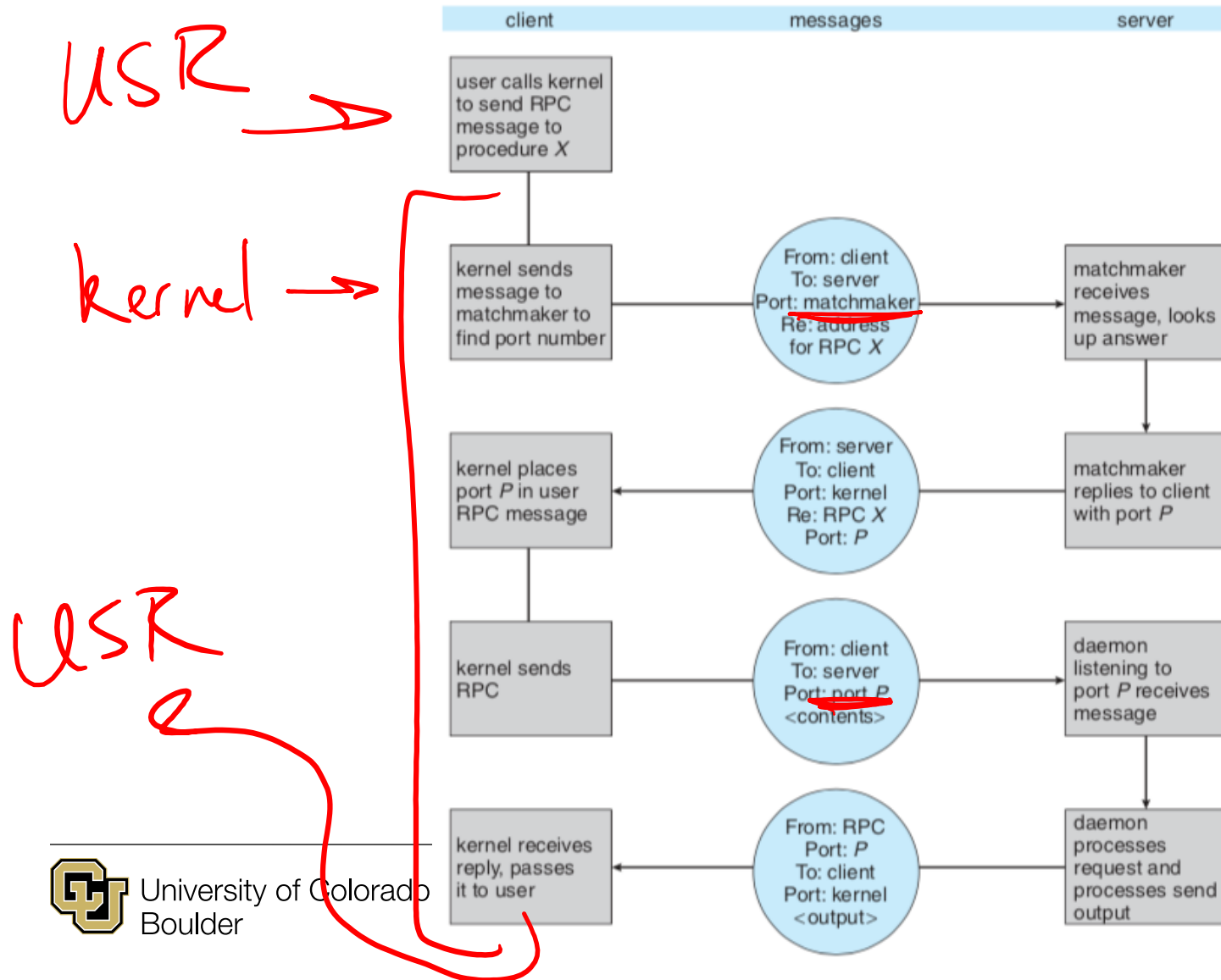


Remote Procedure Call (RPC)

- Can fail more often
- Can be executed more than once
 - Exactly once vs. at most once
 - Timestamped message record is needed for “at most once”
 - An ACK protocol is needed for “exactly once”
- Remote port selection
 - Static port => Pre-selected
 - Dynamic port => matchmaker is needed

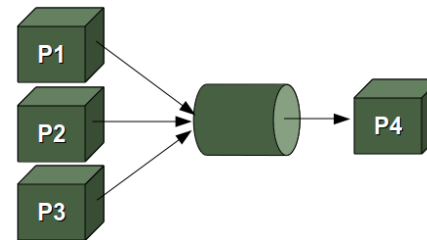


Remote Procedure Call (RPC)



Inter-Process Communications (communications *between* processes)

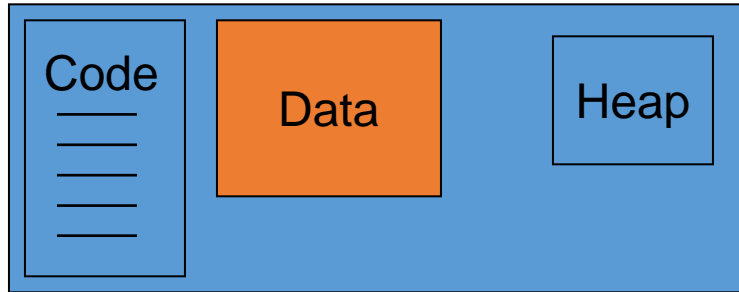
- Signals
- Message Passing
 - Pipes
 - Sockets
- Remote Procedure Calls
- Shared Memory



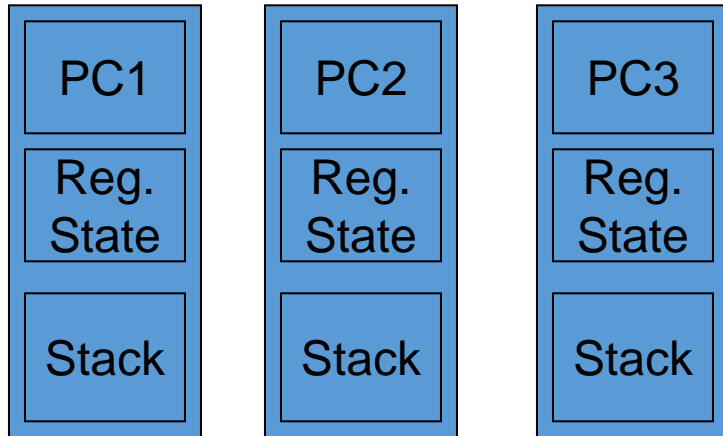
Multiple Threads

Main Memory

Process P1's Address Space

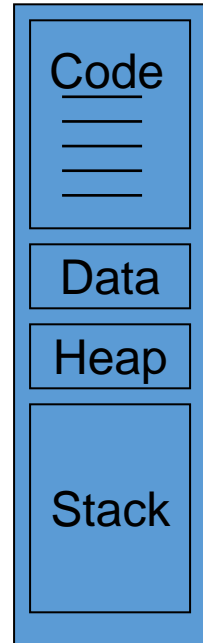


Thread 1 Thread 2 Thread 3

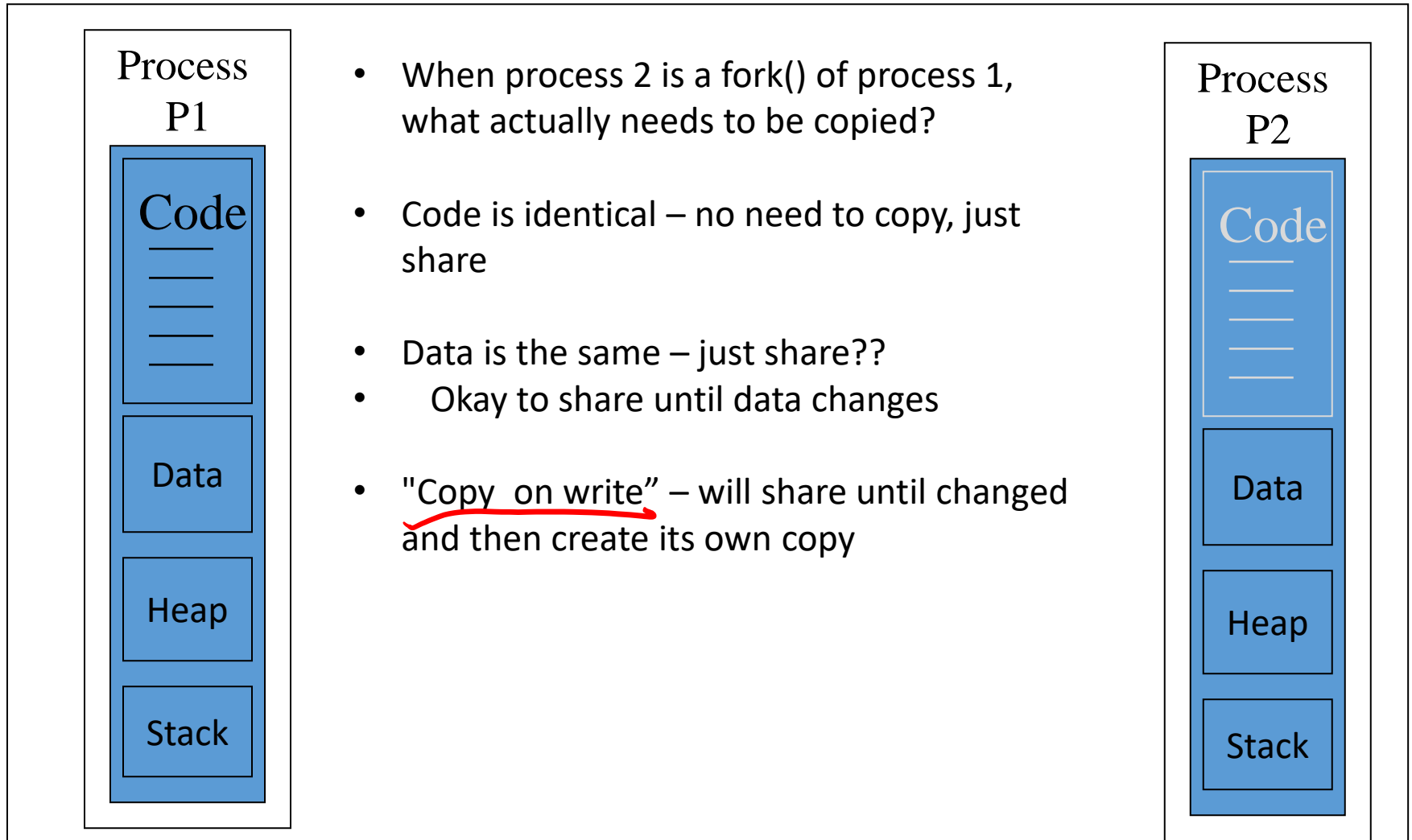


- Processes have separate Code, Data, Stack, & Heap
- Threads SHARE Code, Data, & Heap
- Threads have separate Stack
- Can processes share more?

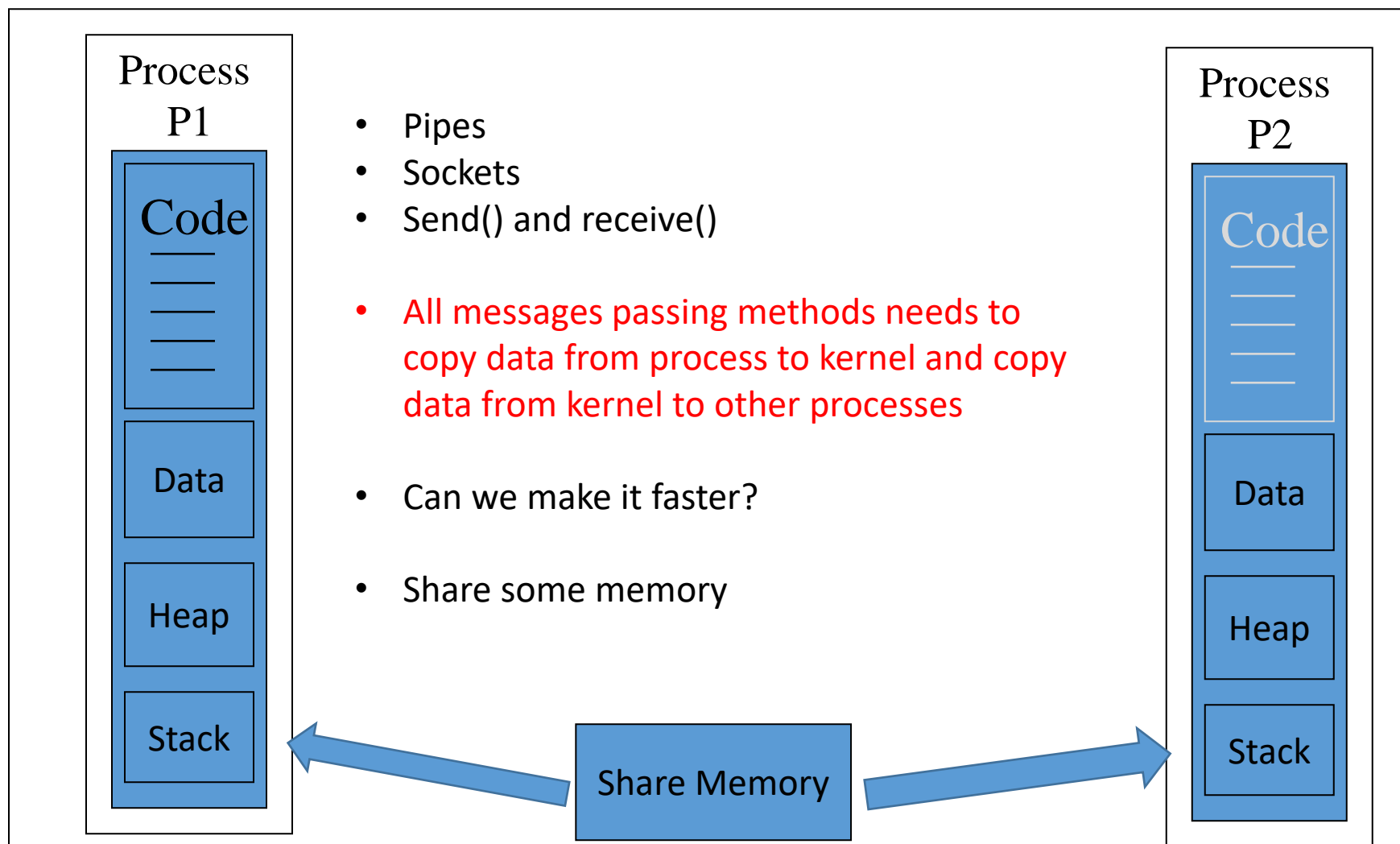
Process P2



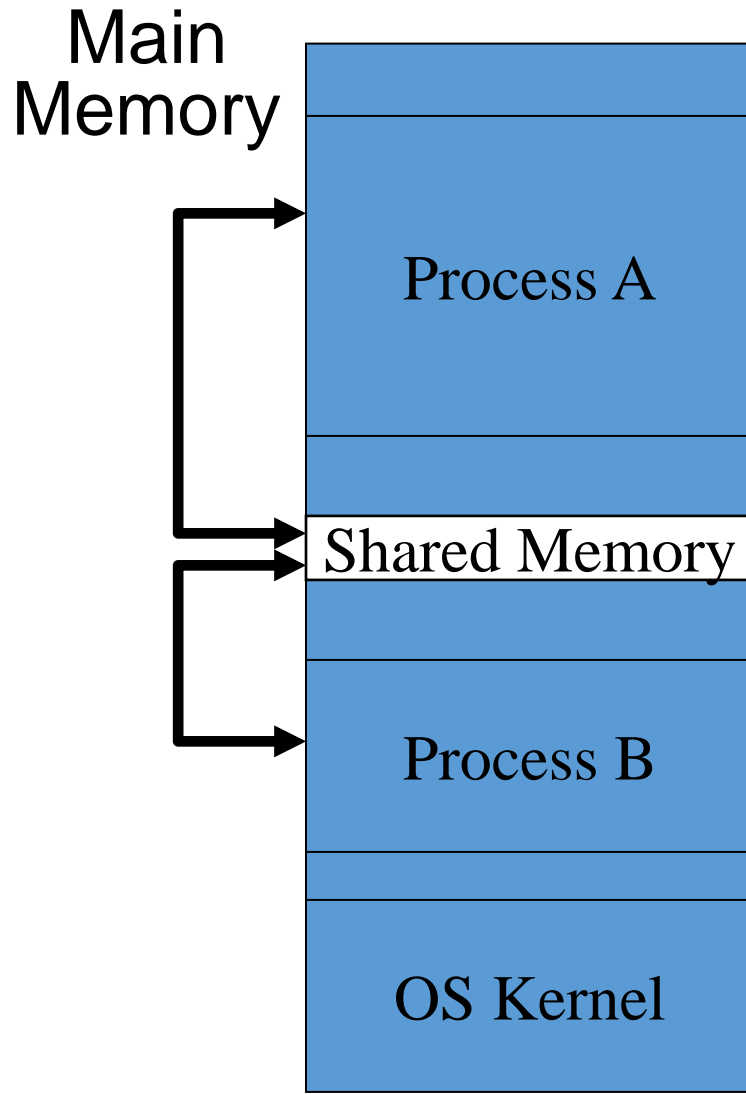
Duplication of a Process



Sharing Data with Multiple Processes



IPC Shared Memory



- OS provides mechanisms for creation of a shared memory buffer between processes (both processes have address mapped to their process)
- Applies to processes on the same machine
- Problem: shared access introduces complexity
 - need to synchronize access



IPC Shared Memory (Linux)

- `shmid = shmget (key name, size, flags)`
is part of the POSIX API
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.

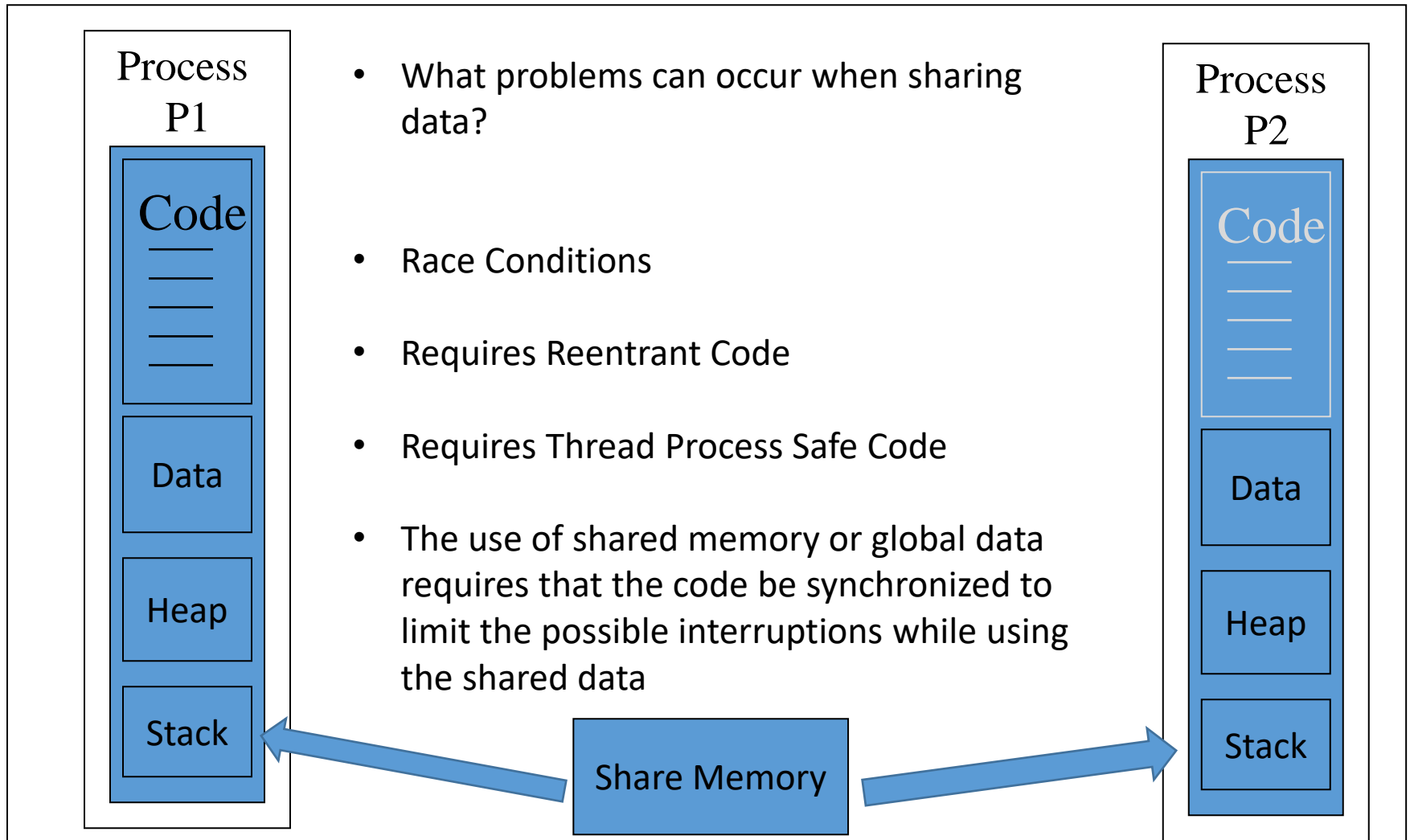
IPC Shared Memory (Linux)

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

Details about Linux support for shared memory IPC will be covered in recitation



Multiple Threads



Summary

- Definition of process and its states
- How processes are managed by the OS (PCB)
- Mechanisms for processes to be created, executed, and terminated
- Cooperating processes needs IPC
- IPC includes
 - Signaling/interrupts
 - Memory sharing
 - Message passing
 - Remote procedure call
- Smaller operating unit of OS is threads
 - Lighter weight unit with faster context switching
 - More overhead in managing them by the application programmer



Example: Shared memory

- Producer consumer system

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_consumed */
}
```

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
ftruncate(shm_fd, 4096);
mmap()
```



Example: Message passing

- Producer consumer system

```
message next_produced;  
  
while (true) {  
    /* produce an item in next_produced */  
  
    send(next_produced);  
}
```

```
message next_consumed;  
  
while (true) {  
    receive(next_consumed);  
  
    /* consume the item in next_consumed */  
}
```



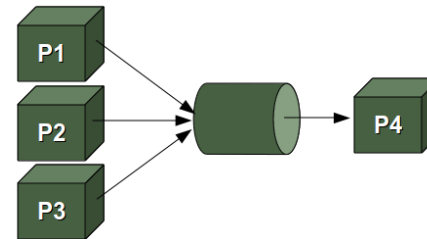
Inter-Process Communications (communications *between* processes)

- Signals / Interrupts
+ Notifying that an event has occurred



- Message Passing

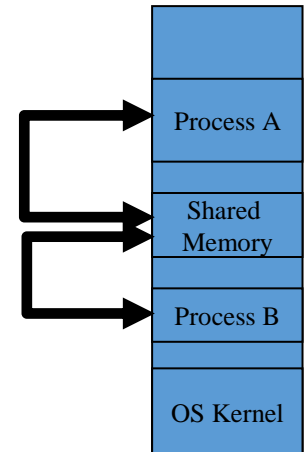
- Pipes
- Sockets



- Shared Memory

- Race conditions
- Synchronization

- Remote Procedure Calls



Why IPC?

- Information sharing
 - Shared file that can be accessed concurrently
- Computation speedup through sub-tasking
 - Parallel execution of subtasks => merge the sub-results
- Modularity.
 - Modular design fashion
- Convenient for users
 - Individual tasks to be running in parallel

