CS2106
Process Management
Inter-Process Communication

Lecture 4

Overview

- Inter-process Communication
  - Motivation
  - Common communication mechanisms:
    - Shared memory
    - Message passing
    - Pipes (Unix-specific)
    - Signal (Unix-specific)

Inter-Process Communication (**IPC**)

- It is hard for cooperating processes to share information
  - Memory space is independent!
  - Inter-Process Communication mechanisms (IPC) is needed

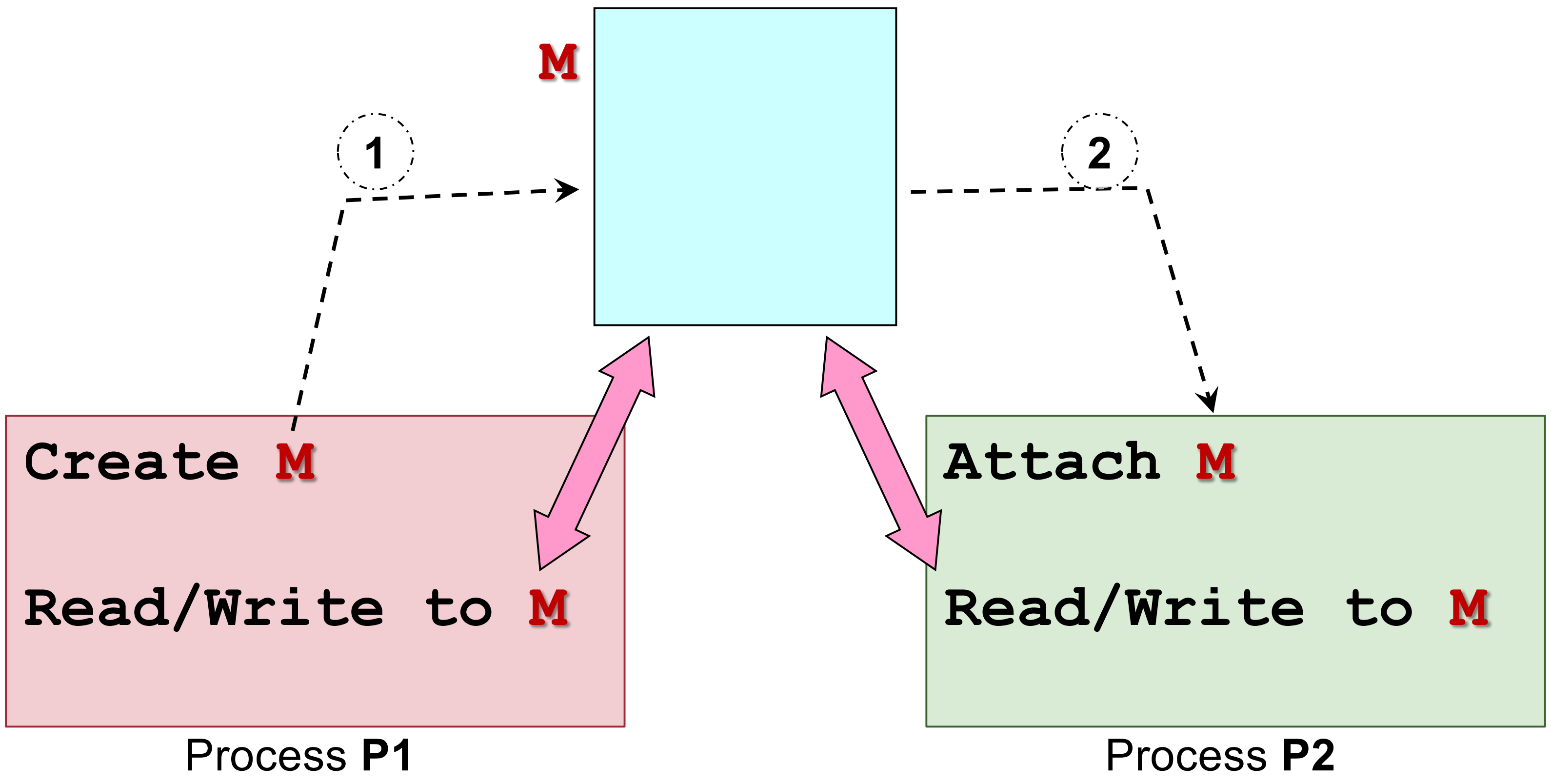
- Two common IPC mechanisms:
  - Shared-Memory and Message Passing

- Two Unix-specific IPC mechanisms:
  - Pipe and Signal

Shared-Memory

- Communication happens through reads/writes to shared variables
- General Idea:
  - Process $\mathbf{P_1}$ creates a shared memory region **M**
  - Process $\mathbf{P_2}$ attaches memory region **M** to its own memory space
  - $\mathbf{P_1}$ and $\mathbf{P_2}$ can now communicate using memory region M
    - **M** behaves very similar to normal memory region
    - Any writes to the region can be seen by all other parties
- The same model is applicable to multiple processes sharing the same memory region

Shared-Memory: Illustration



- OS is involved in step 1 and 2 only

Shared-Memory: Pros and Cons

- **Advantages:**
  - Efficient:
    - Only the initial steps (e.g., Create and Attach shared memory region) involves OS
  - Ease of use:
    - Shared memory region behaves the same as normal memory space
    - i.e., information of any type or size can be written easily
- **Disadvantages:**
  - Requires Synchronization:
    - Shared resource ➔ Need to synchronize accesses (more later)
    - Without synchronization: data races (a.k.a. race conditions) → incorrect behavior
  - Implementation is usually harder

Example

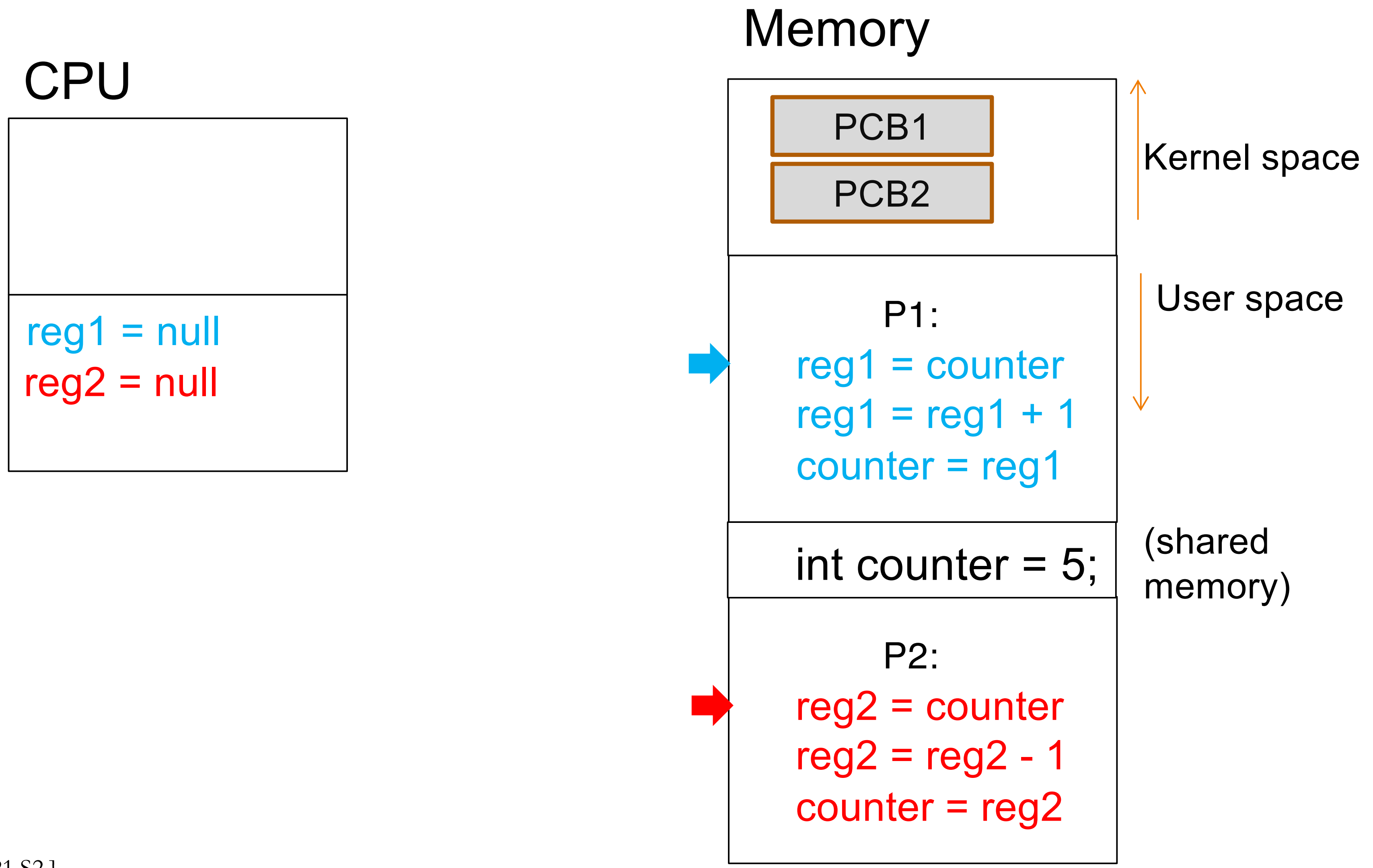
compiled:

```
int counter=5;
//shared variable

void incCounter(){
      counter++;
}

void decCounter(){
      counter--;
}

Process P0: incCounter();
Process P1: decCounter();
```

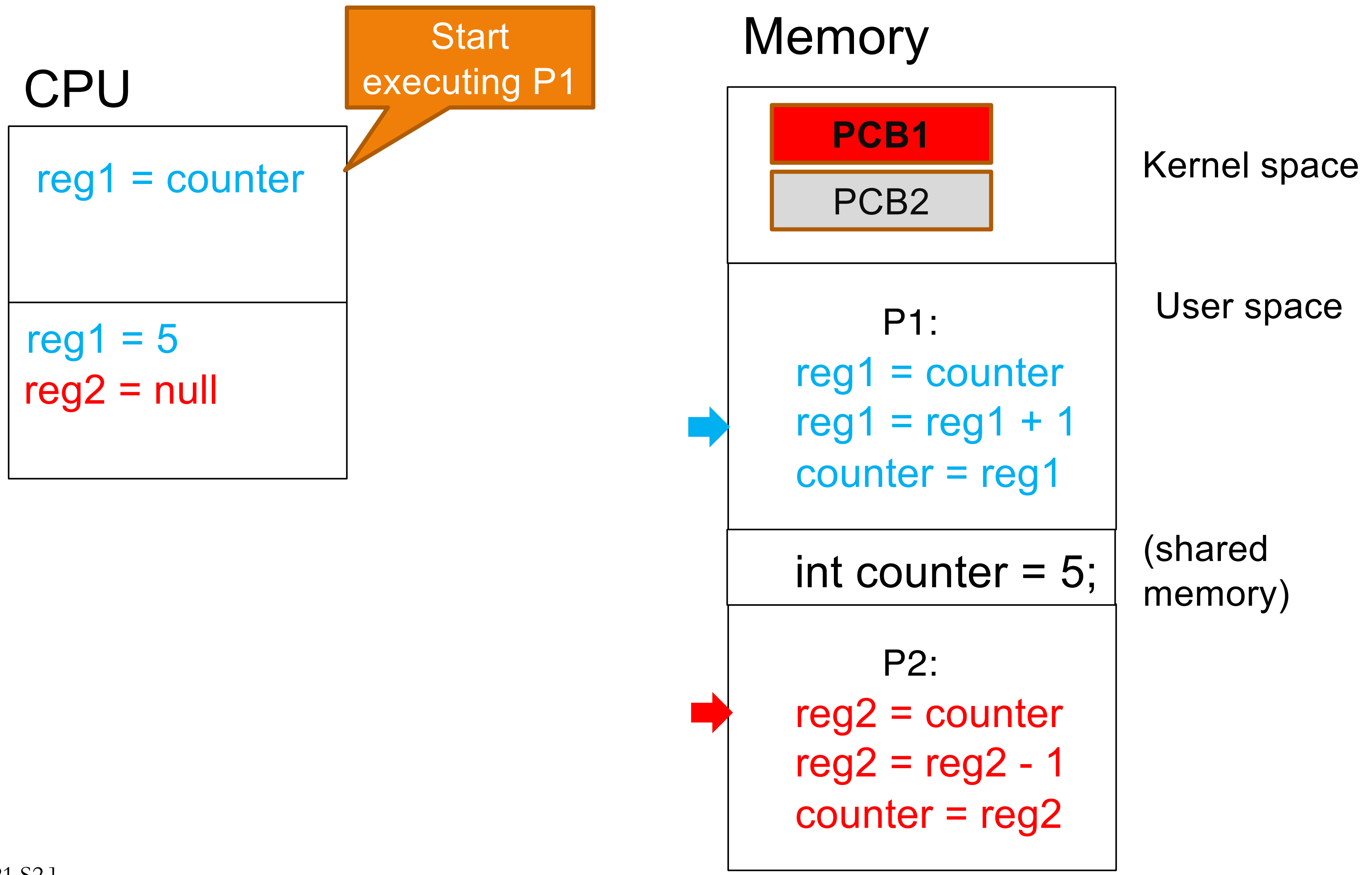
```
incCounter()
   load  reg1,counter
   add   reg1,reg1,1
   store reg1,counter
```

```
decCounter()
   load  reg2,counter
   sub   reg2,reg2,1
   store reg2,counter
```

Example

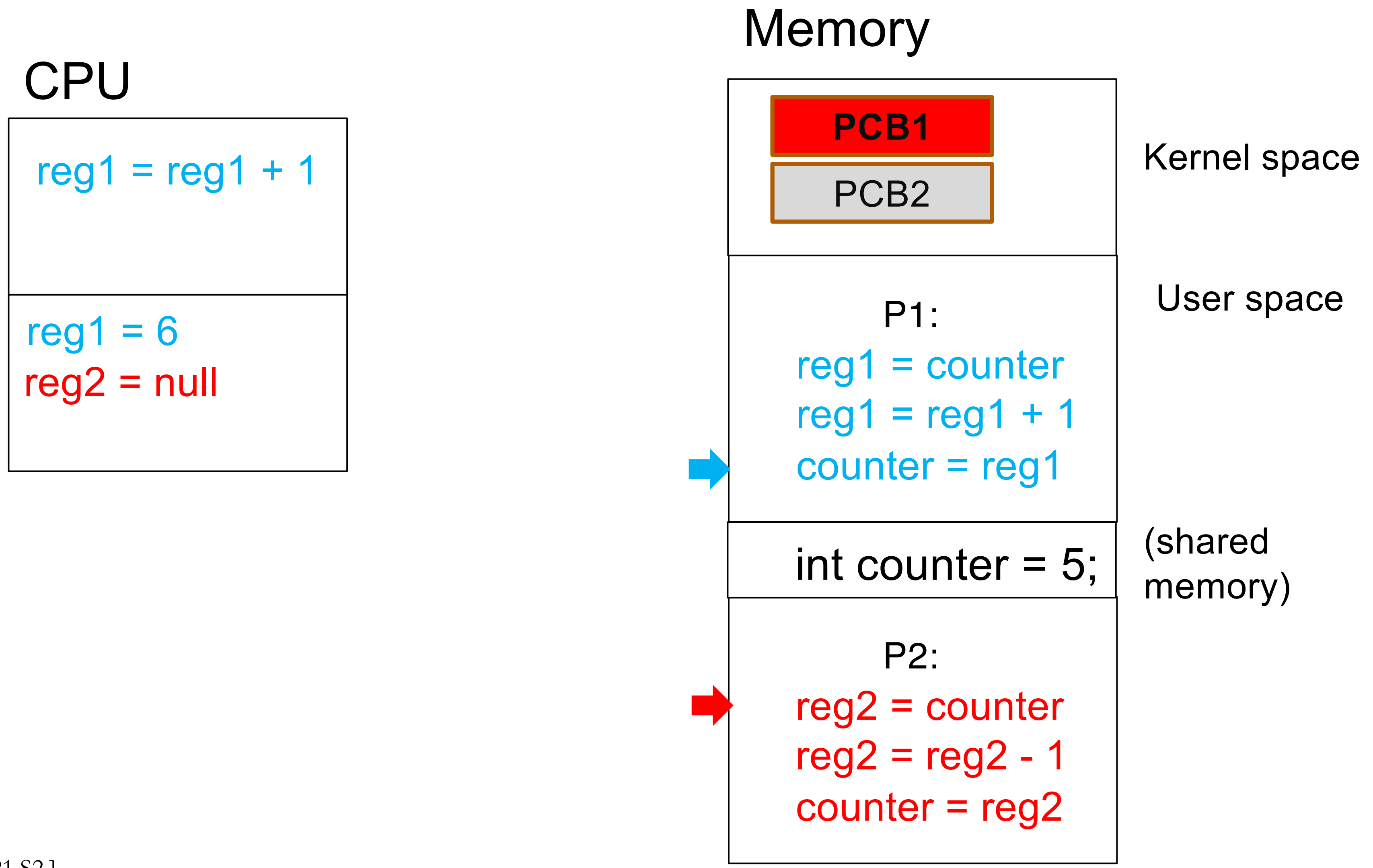

CPU
reg1 = null
reg2 = null
Memory
PCB1
PCB2
Kernel space
User space
P1:
reg1 = counter
reg1 = reg1 + 1
counter = reg1
int counter = 5;
(shared
memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

Example



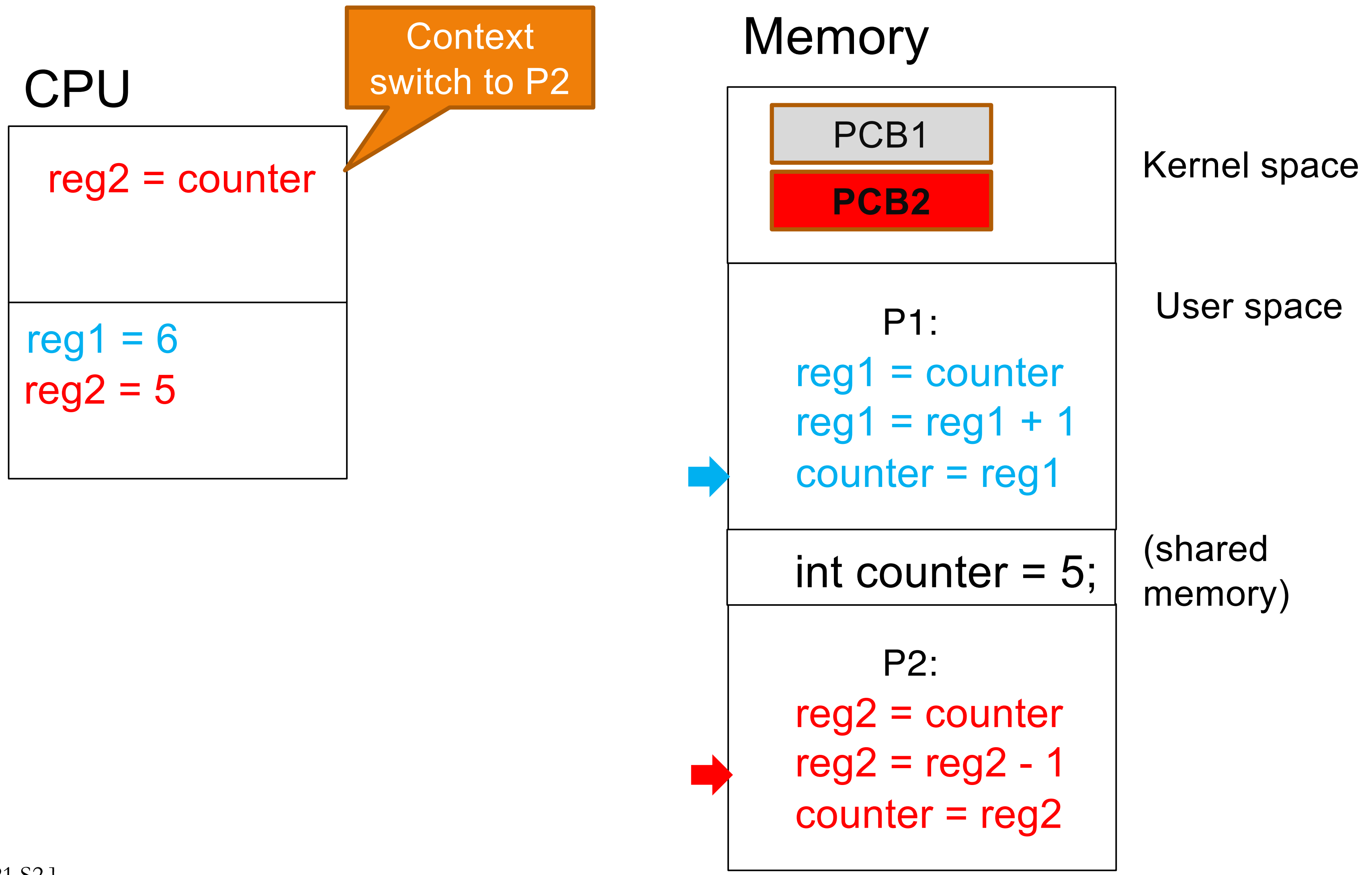
CPU
Start executing P1
reg1 = counter
reg1 = 5
reg2 = null
Memory
PCB1
PCB2
Kernel space
User space
P1:
reg1 = counter
reg1 = reg1 + 1
counter = reg1
int counter = 5;
(shared memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

Example

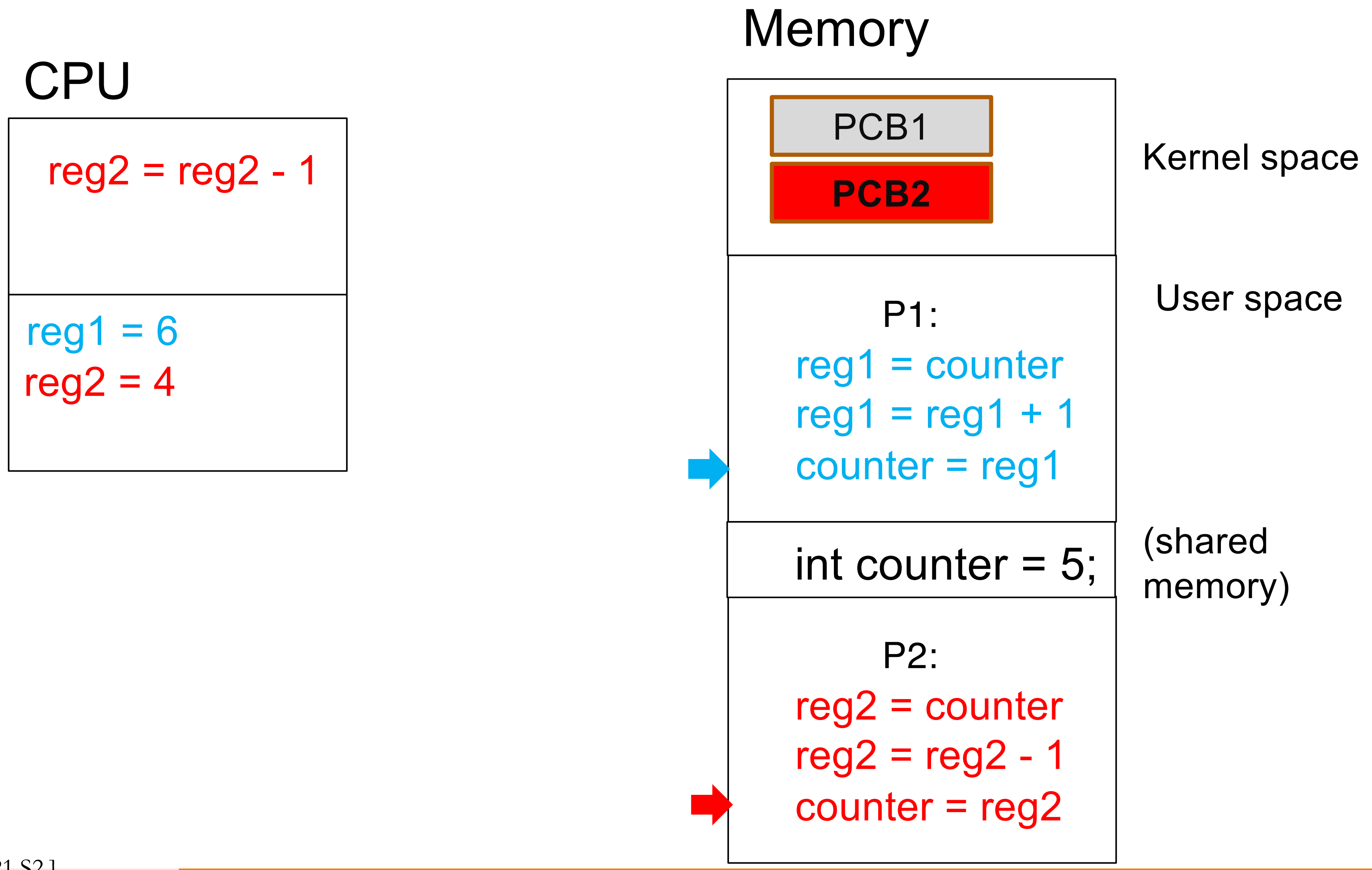


CPU
reg1 = reg1 + 1
reg1 = 6
reg2 = null
Memory
PCB1
PCB2
Kernel space
User space
P1:
reg1 = counter
reg1 = reg1 + 1
counter = reg1
int counter = 5;
(shared memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

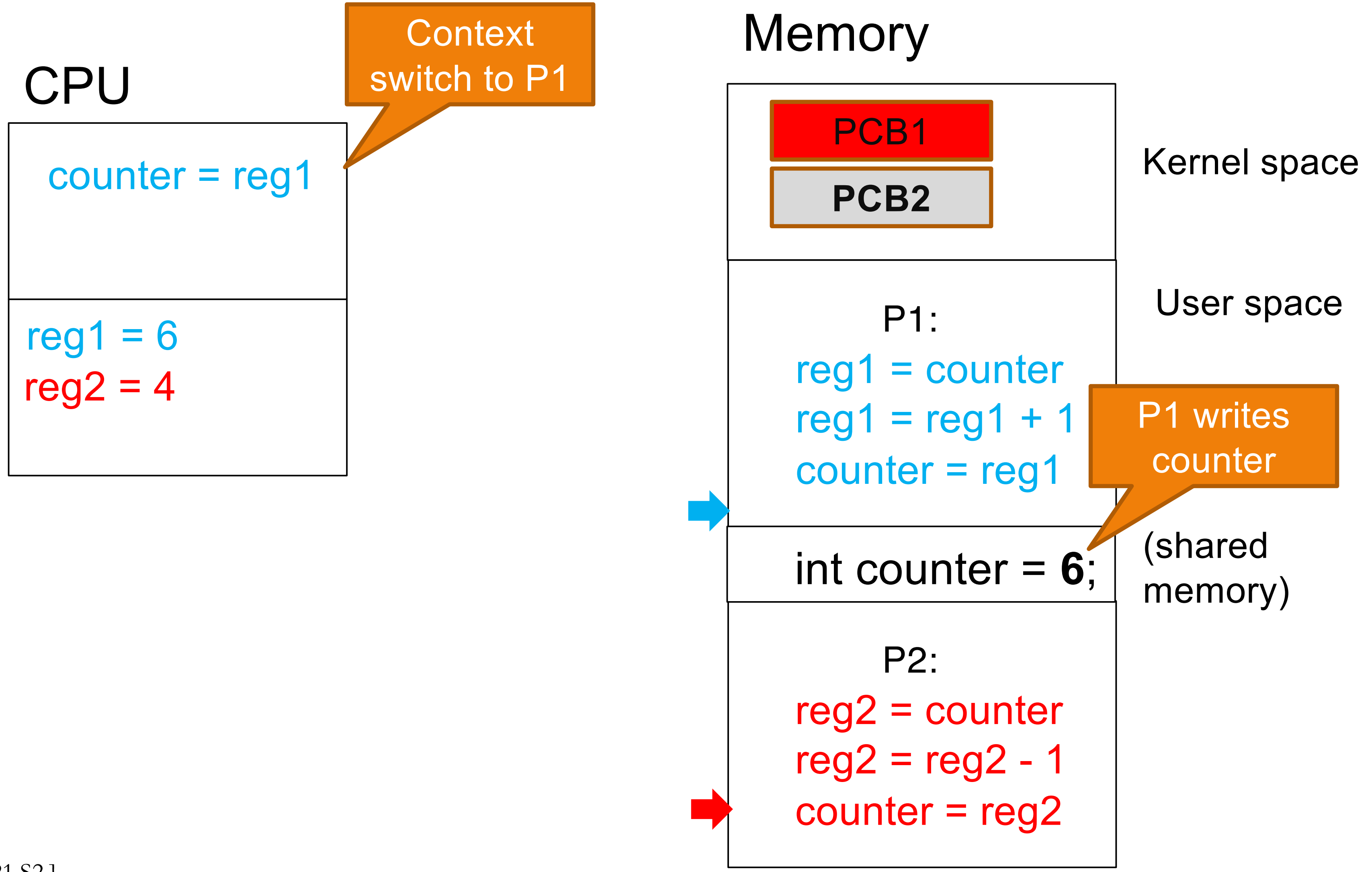
Example


CPU
reg2 = counter
reg1 = 6
reg2 = 5
Context switch to P2
Memory
PCB1
PCB2
Kernel space
User space
P1:
reg1 = counter
reg1 = reg1 + 1
counter = reg1
int counter = 5;
(shared memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

Example

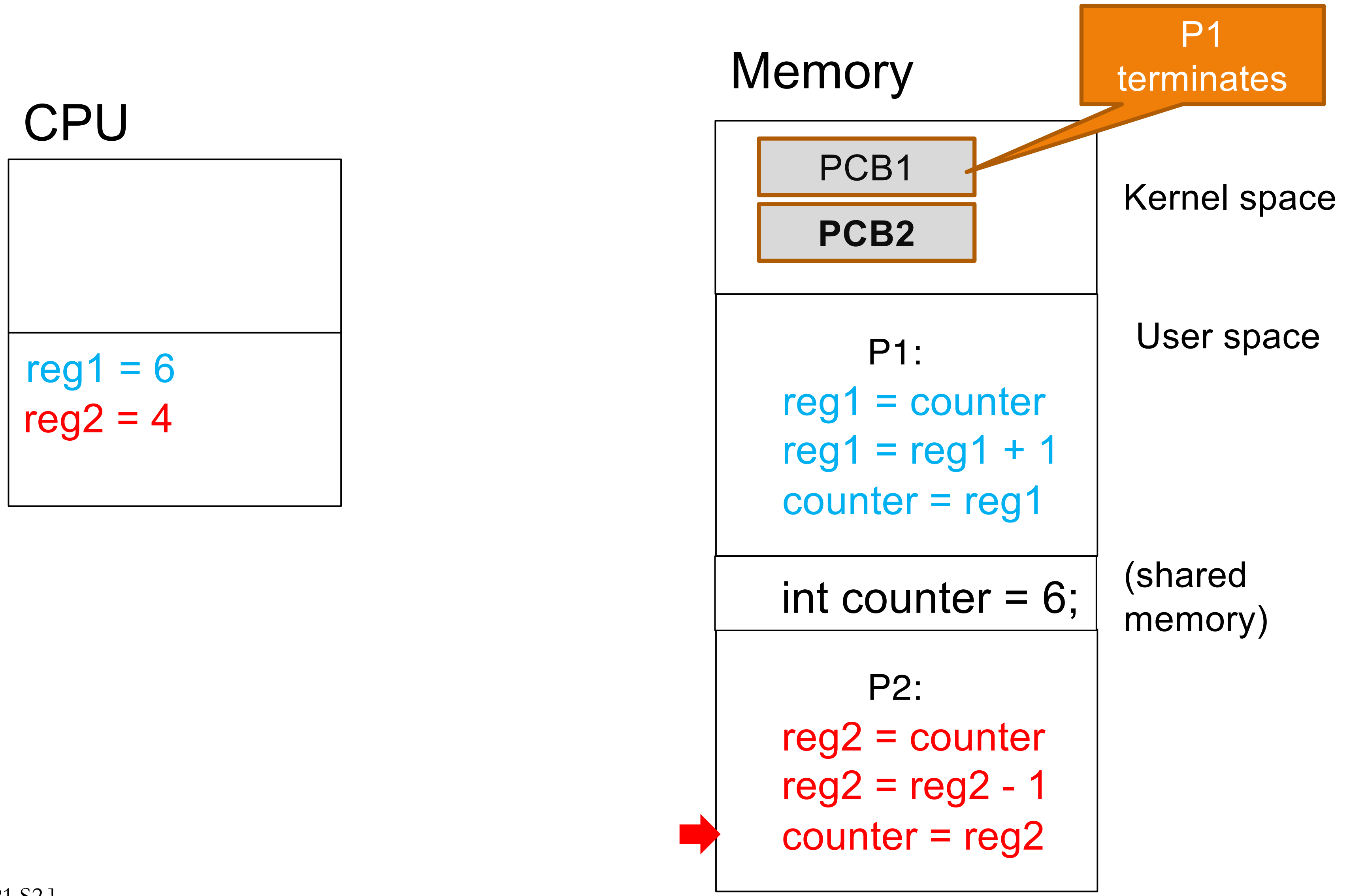

CPU
reg2 = reg2 - 1
reg1 = 6
reg2 = 4
Memory
PCB1
PCB2
Kernel space
User space
P1:
reg1 = counter
reg1 = reg1 + 1
counter = reg1
int counter = 5;
(shared
memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

Example



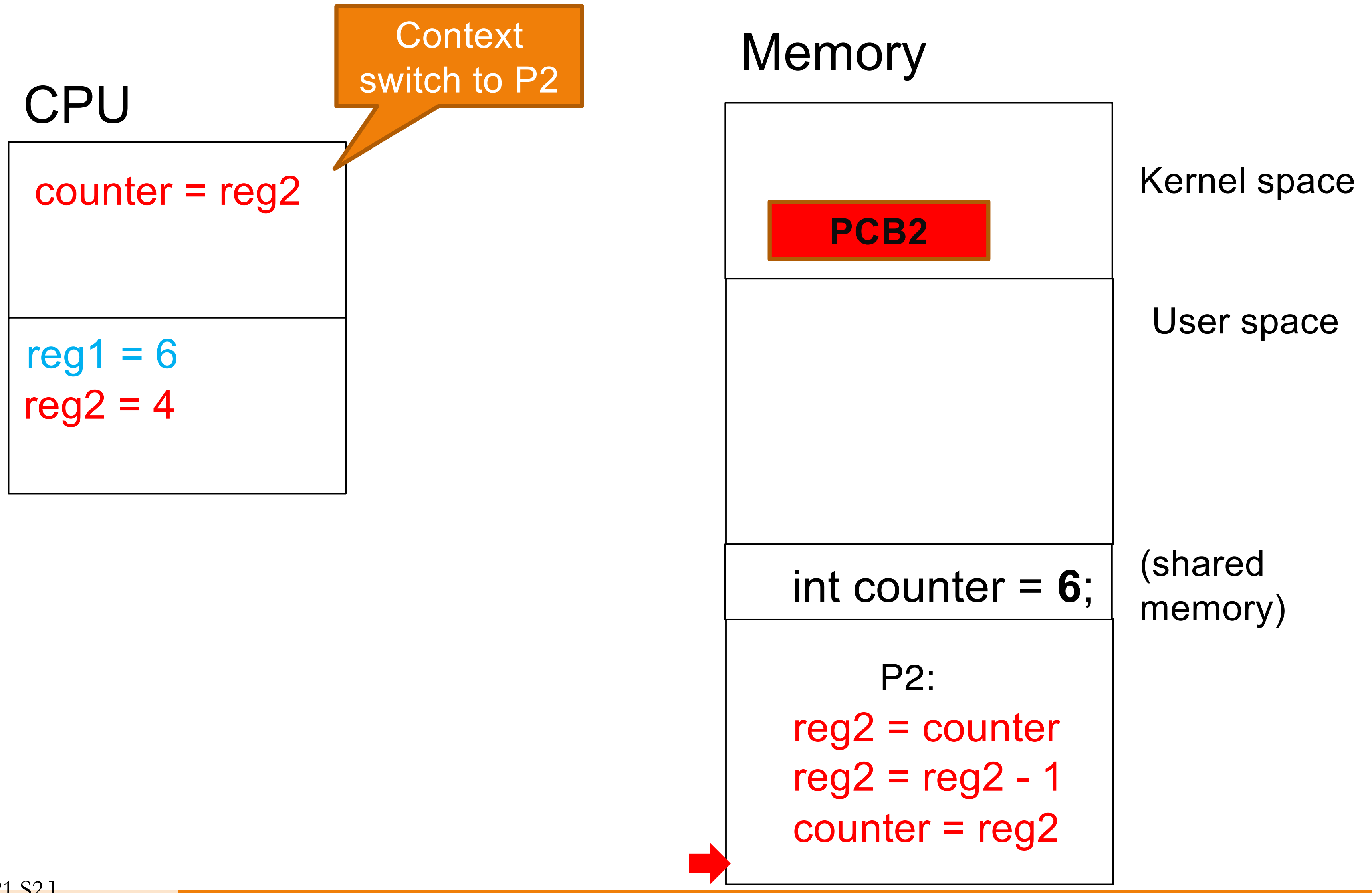
CPU
Context switch to P1
counter = reg1
reg1 = 6
reg2 = 4
Memory
PCB1
PCB2
Kernel space
User space
P1:
reg1 = counter
reg1 = reg1 + 1
counter = reg1
P1 writes counter
int counter = 6;
(shared memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

Example



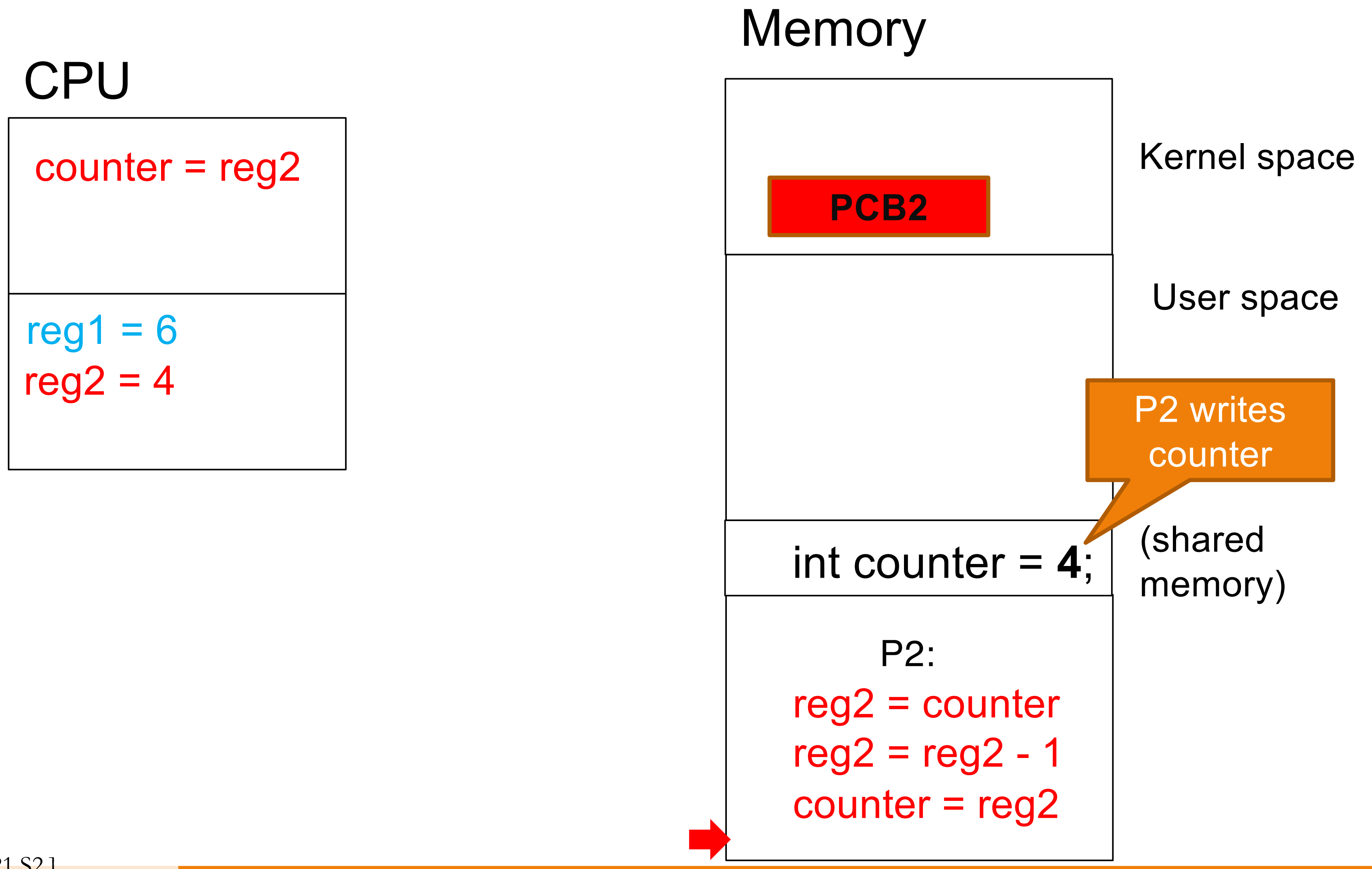
CPU
reg1 = 6
reg2 = 4
Memory
P1
terminates
PCB1
PCB2
Kernel space
User space
P1:
reg1 = counter
reg1 = reg1 + 1
counter = reg1
int counter = 6;
(shared
memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

Example



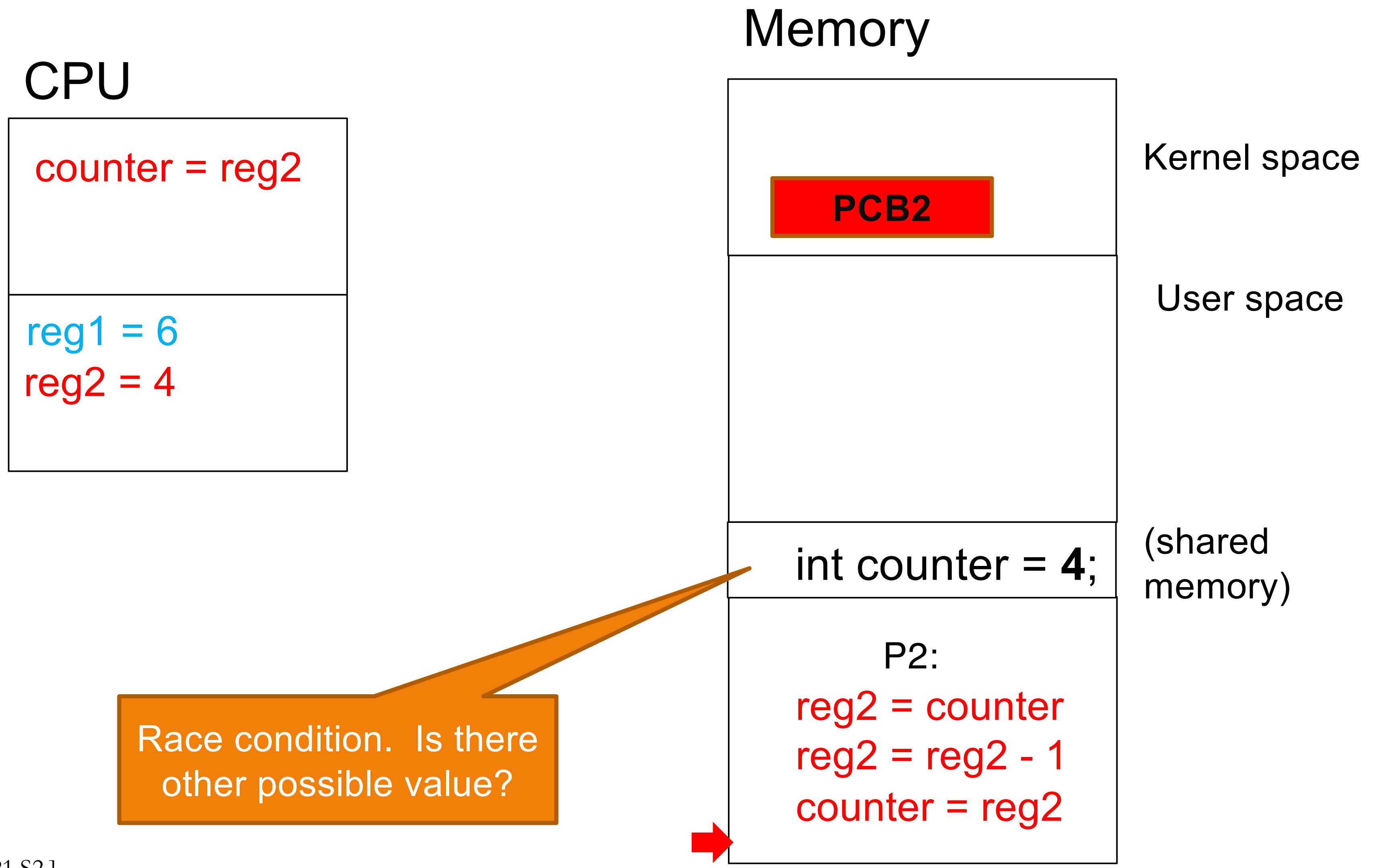
CPU
counter = reg2
Context switch to P2
reg1 = 6
reg2 = 4
Memory
PCB2
Kernel space
User space
int counter = 6;
(shared memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

Example



CPU
counter = reg2
reg1 = 6
reg2 = 4
Memory
PCB2
Kernel space
User space
P2 writes counter
int counter = 4;
(shared memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2

Example


CPU
counter = reg2
reg1 = 6
reg2 = 4
Memory
PCB2
Kernel space
User space
int counter = **4**;
(shared
memory)
P2:
reg2 = counter
reg2 = reg2 - 1
counter = reg2
Race condition. Is there other possible value?

Problems in the example

```
incCounter()
   load   reg1,counter
   add    reg1,reg1,1
   store  reg1,counter
```

```
decCounter()
   load   reg2,counter
   sub    reg1,reg1,1
   store  reg1,counter
```

- The final outcome depends on the interleaving of read and write operations to shared variable counter

- How many interleavings are possible between two processes? 4!/2*2 = 6
  - inc-LD, inc-ST, dec-LD, dec-ST → counter = *5*
  - inc-LD, dec-LD, inc-ST, dec-ST → counter = *4*
  - inc-LD, dec-LD, dec-ST, inc-ST → counter = *6*
  - dec-LD, dec-ST, inc-LD, inc-ST → counter = 5
  - dec-LD, inc-LD, dec-ST, inc-ST → counter = 6
  - dec-LD, inc-LD, inc-ST, dec-ST → counter = 4

Are they all OK? No!

Race Conditions

- The system behavior depends on the exact interleaving
  - Systems that have race conditions are incorrect
  - The impact is context-dependent (could be catastrophic)

- Possibly huge number of interleaving scenarios!
  - Grows exponentially w.r.t. the number of processes, shared variables, and operations on shared variables

- Some interleavings are OK, some are not.
  - **How to avoid bad interleavings, while allowing as many good ones as possible? → Synchronization! (next week)**

POSIX Shared Memory in *nix

- Basic steps of usage:
  1. Create/locate a shared memory region **M**
  2. Attach **M** to process memory space
  3. Read from/Write to **M**
     - Written values visible to all process that share **M**
  4. Detach **M** from memory space after use
  5. Destroy **M**
     - Only one process need to do this
     - Allowed only if **M** is not attached to any process

Example: Master program (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int shmid, i, *shm;

    shmid = shmget( IPC_PRIVATE, 40, IPC_CREAT | 0600);

    if (shmid == -1){
        printf("Cannot create shared memory!\n");
        exit(1);
    } else
        printf("Shared Memory Id = %d\n", shmid);

    shm = (int*) shmat( shmid, NULL, 0 );
    if (shm == (int*) -1){
        printf("Cannot attach shared memory!\n");
        exit(1);
    }
```

The master program create the shared memory region and wait for the "worker" program to produce values before proceeding.

Step 1. Create Shared Memory region.

Step 2. Attach Shared Memory region.

Example: Master program (2/2)

```
    shm[0] = 0;

    while(shm[0] == 0){
        sleep(3);
    }

    for (i = 0; i < 3; i++){
        printf("Read %d from shared memory.\n", shm[i+1]);
    }

    shmdt( (char*) shm);
    shmctl( shmid, IPC_RMID, 0);

    return 0;
}
```

The first element in the shared memory region is used as "control" value in this example (0: values not ready, 1: values ready).

The next 3 elements are values produced by the worker program.

Step 4+5. Detach and destroy Shared Memory region.

Example: Worker program

```
//similar header files
int main()
{   int shmid, i, input, *shm;

    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);

    shm = (int*)shmat( shmid, NULL, 0);
    if (shm == (int*)-1) {
        printf("Error: Cannot attach!\n");
        exit(1);
    }

    for (i = 0; i < 3; i++){
        scanf("%d", &input);
        shm[i+1] = input;
    }
    shm[0] = 1;

    shmdt( (char*)shm );
    return 0;
}
```

Step 1. By using the shared memory region id directly, we skip ***shmget***() in this case.

Step 2. Attach to shared memory region.

Write 3 values into shm[1 to 3]

Let master program know we are done!

Step 4. Detach Shared Memory region.

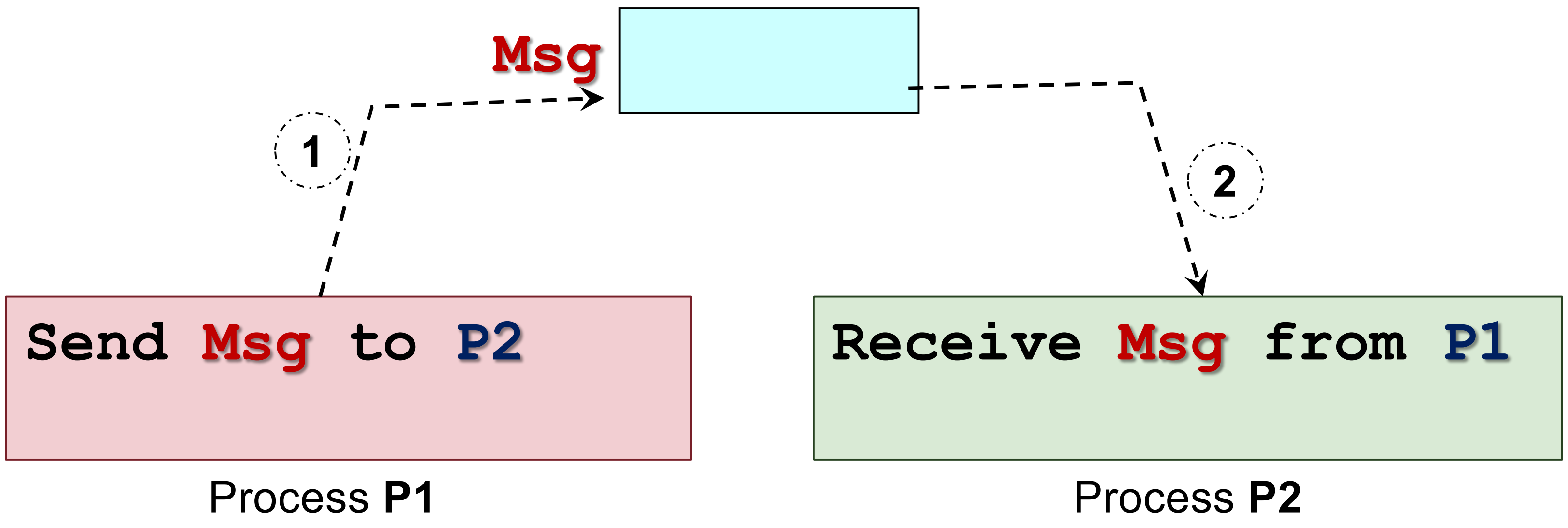
You have 1,023,428 messages waiting…

MESSAGE PASSING

Message Passing

- Explicit Communication through exchange of messages
- General Idea:
  - Process $\mathbf{P_1}$ prepares a message **M** and sends it to Process $\mathbf{P_2}$
  - Process $\mathbf{P_2}$ receives the message **M**
  - Message sending and receiving are usually provided as system calls
- Additional properties:
  - **Naming**
    - How to identify the other party in the communication
  - **Synchronization**
    - Implicit: through the blocking behavior of the sending/receiving operations

Message Passing: Illustration



- The `Msg` have to be stored in kernel memory space
- All send/receive operations must go through OS (i.e., a system call)

Naming Scheme: **Direct Communication**

- Sender/Receiver of message **explicitly name the other party**

- Example:
  - **`Send(P`$_2$`, Msg)`**: Send Message **`Msg`** to Process **`P`**$_2$
  - **`Receive(P`$_1$`, Msg)`**: Receive Message **`Msg`** from Process **`P`**$_1$

- Characteristics:
  - One link per pair of communicating processes
  - Need to know the identity of the other party

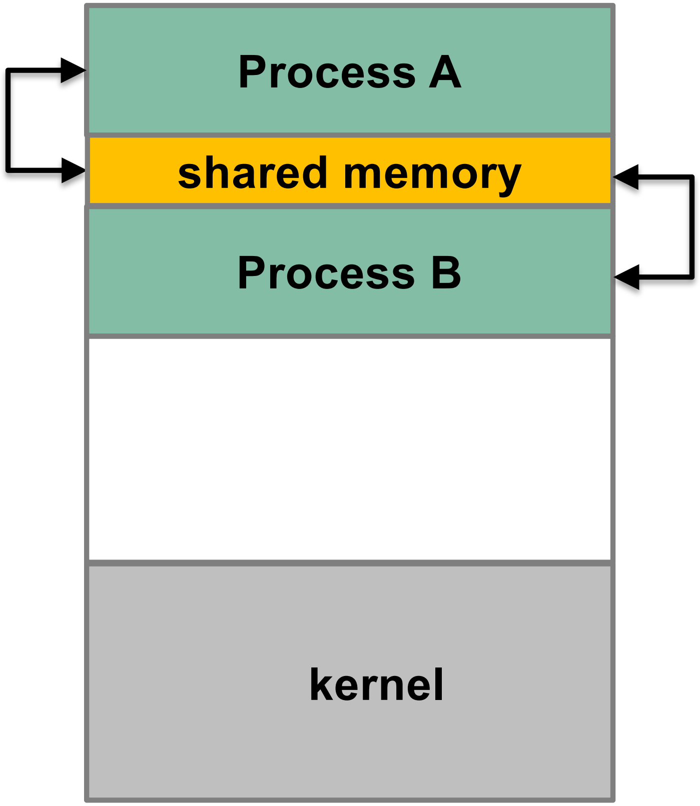
Naming Scheme: **Indirect Communication**

- Message are sent to / received from message storage:
  - Usually known as ***mailbox*** or ***port***

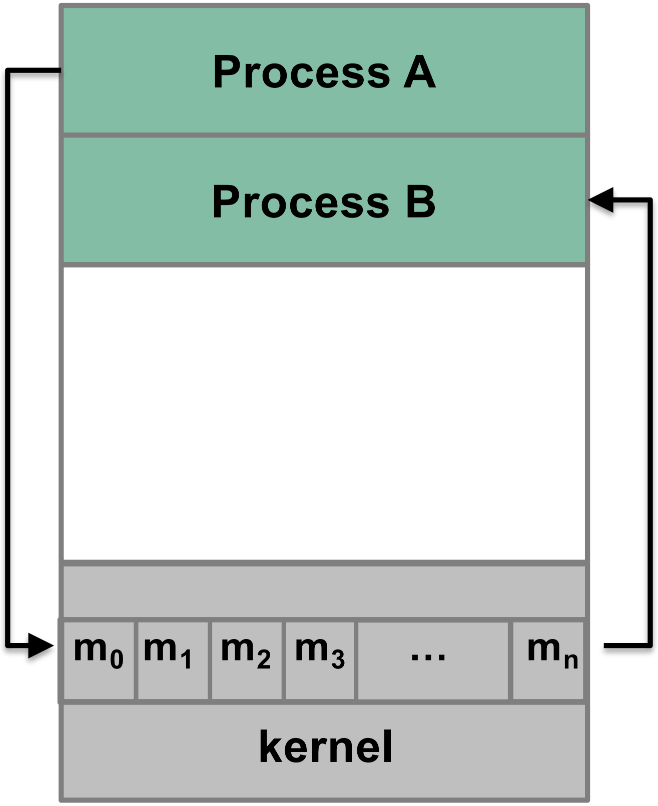
- Example:
  - **`Send( MB, Msg )`** : Send Message **`Msg`** to Mailbox **`MB`**
  - **`Receive( MB, Msg )`** : Receive Message **`Msg`** from Mailbox **`MB`**

- Characteristics:
  - One mailbox can be shared among a number of processes

Shared Memory versus Message Passing



Shared memory



Message passing

Two Synchronization Behaviors

- **Blocking Primitives** (Synchronous):
  - Send(): Sender is blocked until the message is received
  - Receive(): Receiver is blocked until a message has arrived

- **Non-Blocking Primitives** (asynchronous):
  - Send(): Sender resume operation immediately
  - Receive(): Receiver either receive the message if available or some indication that message is not ready yet

Synchronization Model: Receiver

- Blocking receive
  - Most common
  - Receiver must wait for a message it is not already available

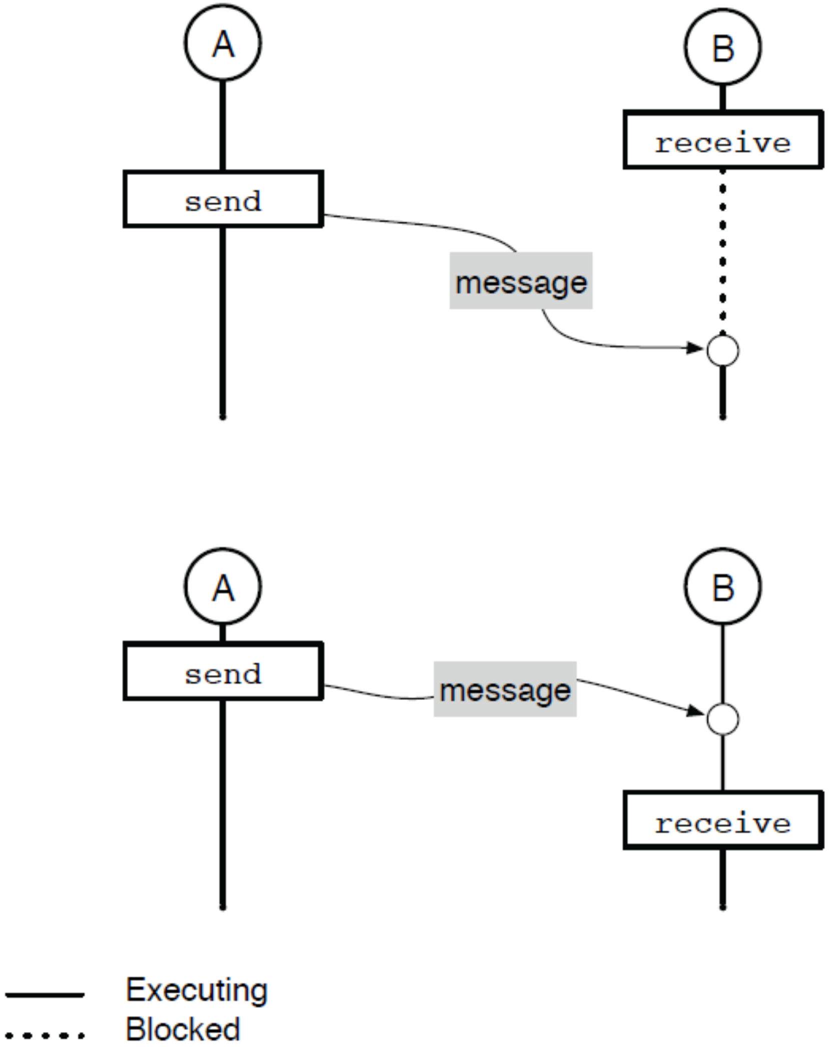
- Non-blocking receive
  - Checks whether a message is available
  - If message is available, retrieves it and moves on
  - If message not available, continues without a message

- We will assume blocking receive from now on

Non-blocking Send = Asynchronous Message Passing

- Sender is never blocked even if the receiver has not yet executed the matching `receive()`
- System buffers the message (up to certain capacity).
- `receive()` performed by the receiver later will be completed immediately.
- Asynchronous MP is generally a good choice, BUT:
  - Too much freedom for the programmer; program is complex to understand.
  - **Finite buffer size means system is not truly asynchronous** (sender waits when the buffer is full or returns with error).

Asynchronous Message Passing



Scenario 1:

Message sent after `receive()` call.
Sender always continues.
Receiver is blocked waiting for message

Scenario 2:

Message sent before `receive()` call.
Sender always continues.
`receive()` gets data immediately

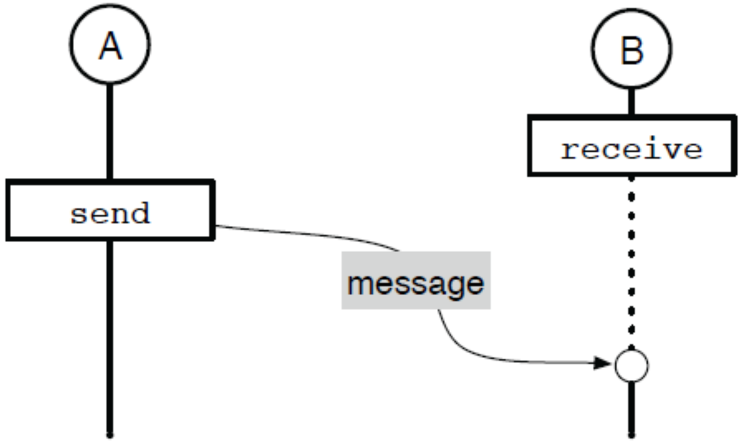
Message Buffers

- Intermediate buffers between sender and receiver in asynchronous communication
- Under OS control → no synchronization necessary
- Large buffer decouples sender and receiver making them less sensitive to variations in execution; they do not wait for each other unnecessarily
- No amount of buffering helps when the sender is always faster than the receiver
- User needs to declare in advance the capacity of the mailbox mailbox is created
  - Sender blocks if the buffer is completely full OR
  - Sender returns immediately with an error

Synchronous Message Passing

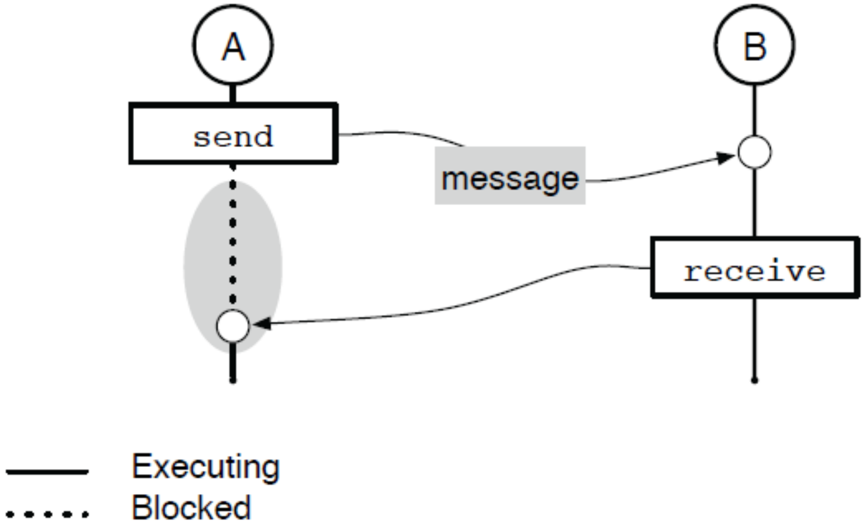
- Also known as rendezvous
- Sender invoking `send()` is blocked till the receiver performs the matching `receive()`
- Sender forced to wait till the receiver is ready → no intermediate buffering required
  - Message can be kept by the sender till the receiver is ready and then directly copied into the receiver address space

Synchronous Message Passing



Scenario 1:

Message sent after `receive()` call
Sender does not have to block.
Receiver is blocked waiting for message



Scenario 2:

Message sent before `receive()` call
Sender is blocked.
Sender is blocked waiting for receiving.

Message Passing: Pros and Cons

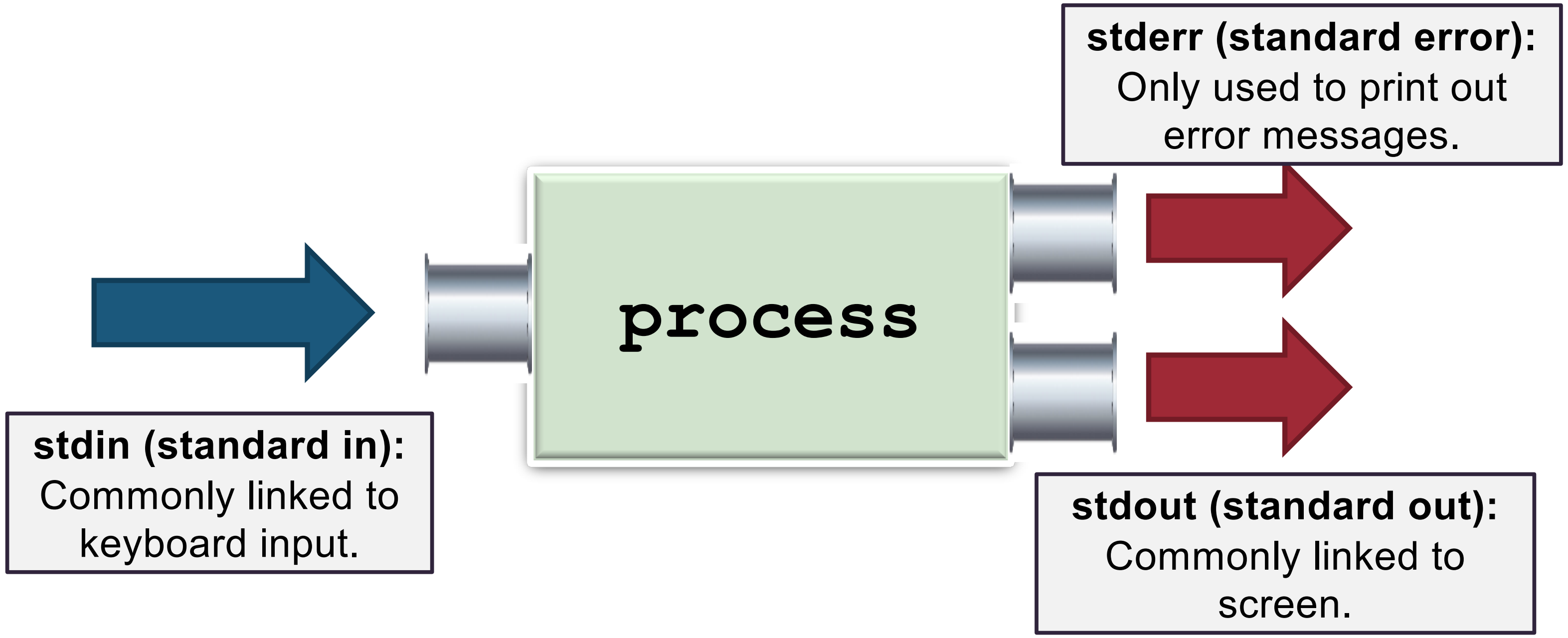
- **Advantages:**
  - Portable:
    - Can easily be implemented on different processing environment, e.g. distributed system, wide area network etc
  - Easier synchronization:
    - E.g. when synchronous primitive is used, sender and receiver are implicitly synchronized
- **Disadvantages:**
  - Inefficient:
    - Usually require OS intervention
  - Harder to use:
    - Message are usually limited in size and/or format

Plumber needed! Leaking pipes all around!

UNIX PIPES

Pipes: Communication channels

- In Unix, a process has 3 default communication channels:

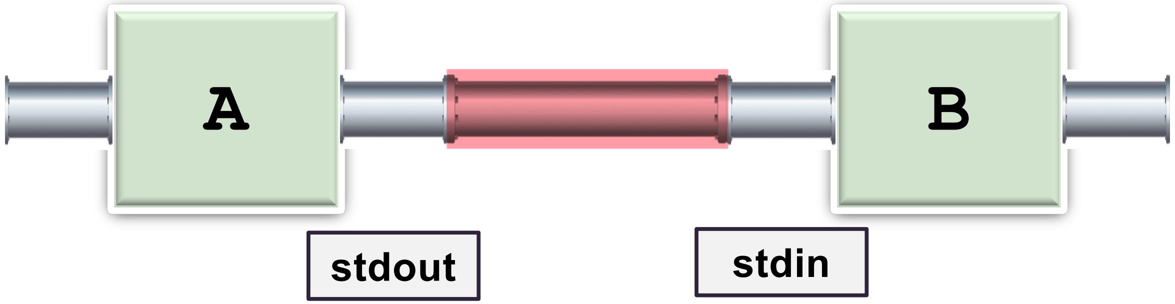


- Example:
  - In a typical C program, printf() uses stdout, scanf() uses stdin.

Piping in Shell

- Unix shell provides the "|" symbol to link the input/output channels of one process to another, this is known as **piping**

- For example (" A | B"):



- The output of A (instead of going to screen) directly goes into B as input (as if it come from keyboard)

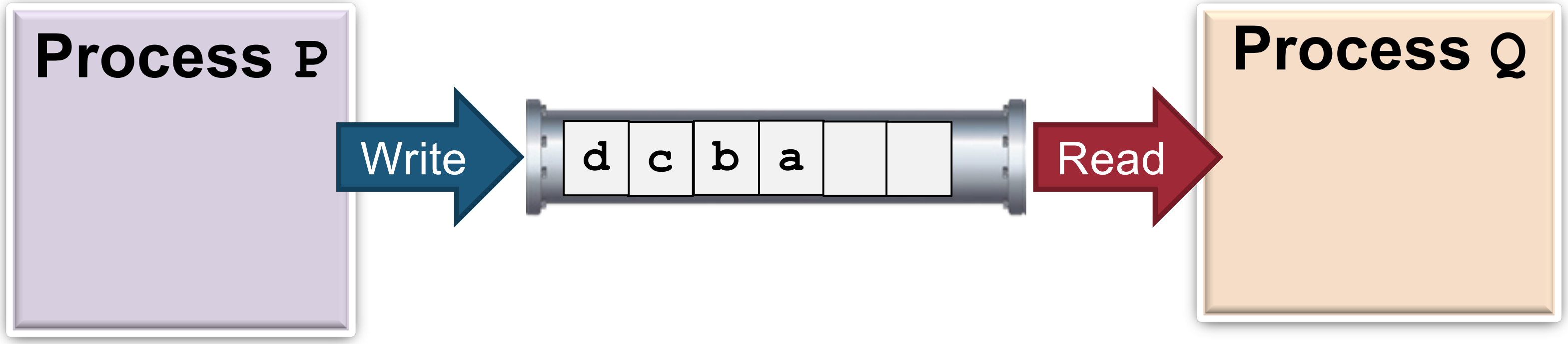
Unix Pipes

- One of the earliest IPC mechanism

- General Idea:
  - A communication channel is created with 2 ends:
    - 1 end for reading, the other for writing
    - Just like a water pipe in the real world

- The piping “|” in shell is achieved using this mechanism internally

Unix Pipes: as a IPC Mechanism



- A pipe can be shared between two processes
- A form of Producer-Consumer relationship
  - P produces (writes) **n** bytes
  - Q consumes (reads) **m** bytes
- Behavior:
  - Like an anonymous file
  - FIFO ➔ must access data in order

Unix Pipes: **Semantic**

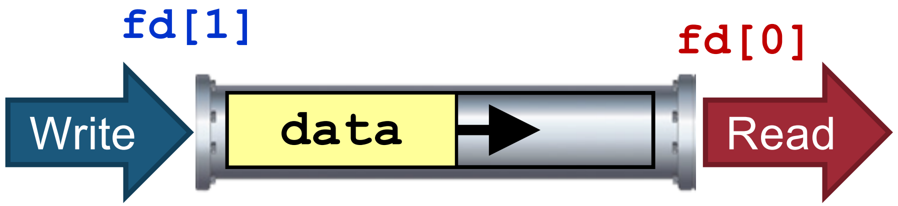
- Pipe functions as **circular bounded byte buffer** with **implicit synchronization**:
  - Writers **wait** when buffer is **full**
  - Readers **wait** when buffer is **empty**

- Variants:
  - Can have multiple readers/writers
    - The normal shell pipe has 1 writer and 1 reader
  - Depends on Unix version, pipes may be **half-duplex**
    - **unidirectional**: with one write end and one read end
  - Or **full-duplex**
    - **bidirectional**: any end for read/write

Unix Pipe: **System Calls**

Header File	`#include <unistd.h>`
Syntax	`int pipe( int fd[] )`

- Returns:
  - 0 to indicate success; !0 for errors
  - An array of file descriptors is returned:
    - fd[0] == reading end
    - fd[1] == writing end



Unix Pipes: Example Code

```c
#define READ_END 0
#define WRITE_END 1

int main()
{
   int pipeFd[2], pid, len;
   char buf[100], *str = "Hello There!";

   pipe( pipeFd );
   if ((pid = fork()) > 0) { /* parent */
     close(pipeFd[READ_END]);
     write(pipeFd[WRITE_END], str, strlen(str)+1);
     close(pipeFd[WRITE_END]);
   } else {                   /* child */
     close(pipeFd[WRITE_END]);
     len = read(pipeFd[READ_END], buf, sizeof buf);
     printf("Proc %d read: %s\n", pid, buf);
     close(pipeFd[READ_END]);
    }
}
```

Unix Pipes: More to explore

- It is possible to:
  - Attach/change the standard communication channels (**`stdin`**, **`stdout`**, **`stderr`**) to one of the pipes
    - ➔ Redirect the input/output from one program to another!
- Unix system calls to explore:
  - **`dup()`**
  - **`dup2()`**
- Wikipedia article on **`dup()`** system call has a great program example

pssst! pssst!

UNIX SIGNAL

Unix Signal: Quick Overview

- A form of inter-process communication
  - An asynchronous notification regarding an event
  - Sent to a process/thread
- The recipient of the signal must handle the signal by:
  - A default set of handlers OR
  - User supplied handler (only applicable to some signals)
- Common signals in Unix:
  - Kill, Stop, Continue, Segmentation Fault Signal….

Example: **Custom Signal Handler**

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void myOwnHandler( int signo )
{
    if (signo == SIGSEGV){
        printf("Memory access blows up!\n");
        exit(1);
    }
}

int main(){
    int *ip = NULL;

    if (signal(SIGSEGV, myOwnHandler) == SIG_ERR)
        printf("Failed to register handler\n");

    *ip = 123;

    return 0;
}
```

User defined function to handle signal. In this example, we handle the "SIGSEGV" signal, i.e. the memory segmentation fault signal.

Register our own code to replace the default handler.

This statement will cause a segmentation fault.

Summary

- Common Inter Process Communication mechanisms:
  - Shared Memory
    - POSIX example
  - Message Passing
  - Unix Pipes
  - Unix Signals

References

- Modern Operating Systems (4th Edition)
  - Chapters 2.3.1 and 2.3.8
- Operating System Concepts (9th Edition)
  - Chapters 3.4 and 3.5
- Operating Systems: Three Easy Pieces
  - None!