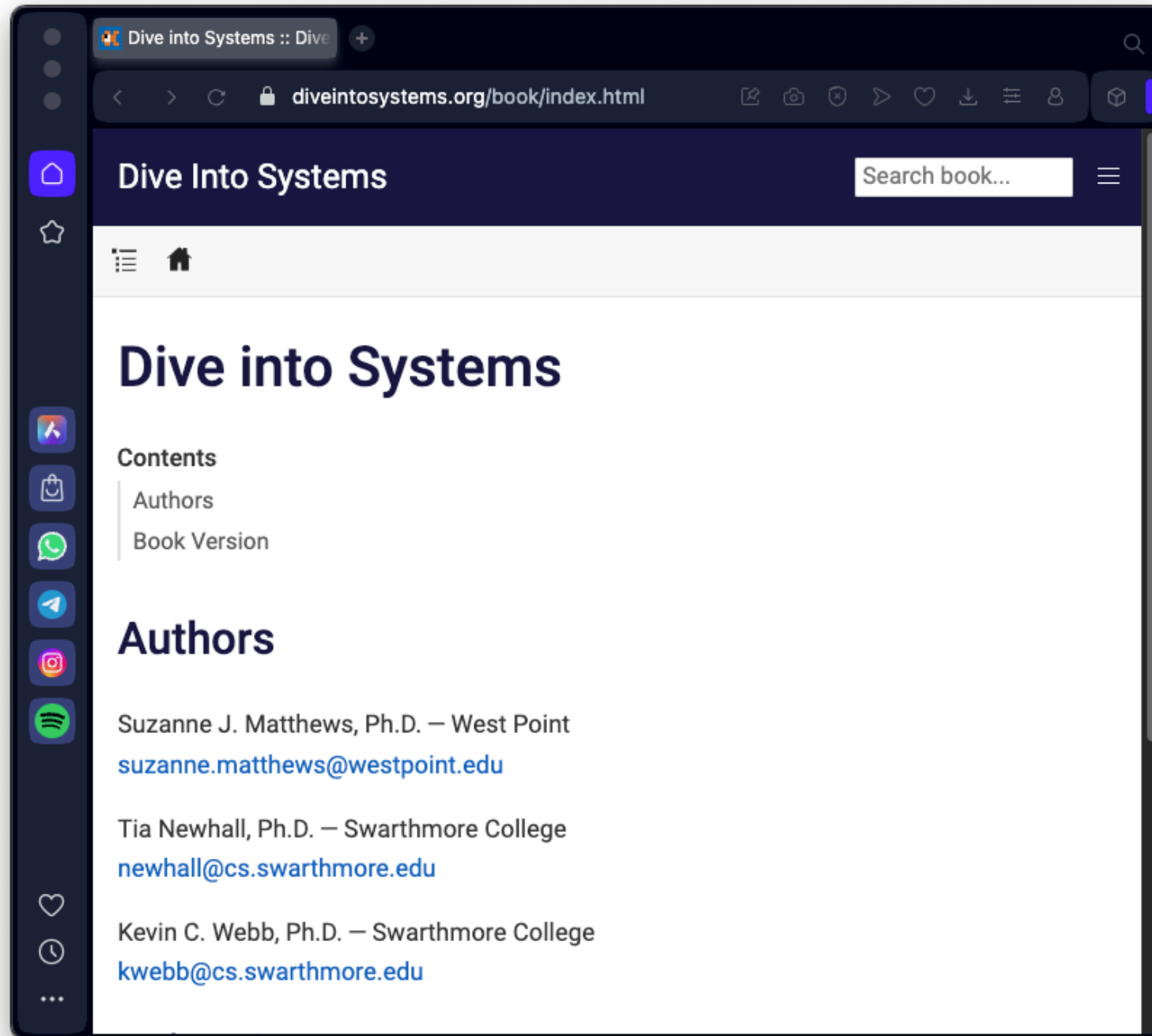


13. The Operating System

For COMSC 142

Free online textbook



- <https://diveintosystems.org/book/index.html>

Topics

13.1. Booting and Running

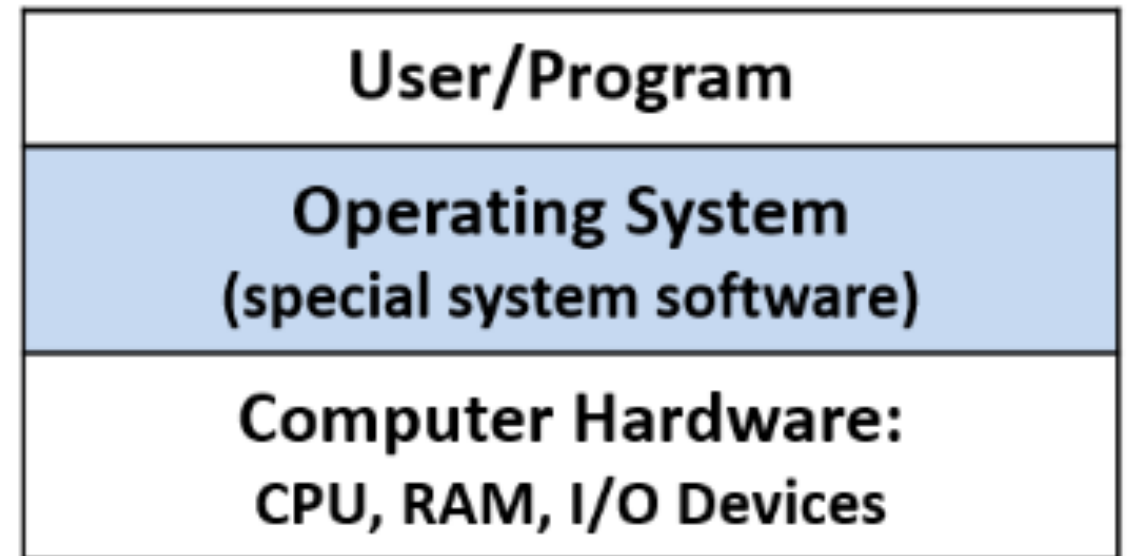
13.2. Processes

13.3. Virtual Memory

13.4. Interprocess Communication

13. The Operating System

- Such as Windows or Linux
- Manages hardware
- Supports initiating programs
 - Loads program into RAM
 - Starts CPU running its code
- Implements **multiprogramming**
 - More than one program can run at the same time
 - OS shares resources, including CPU, among the programs
 - When one program is waiting, another can proceed
- **Process**
 - An abstraction of a running program



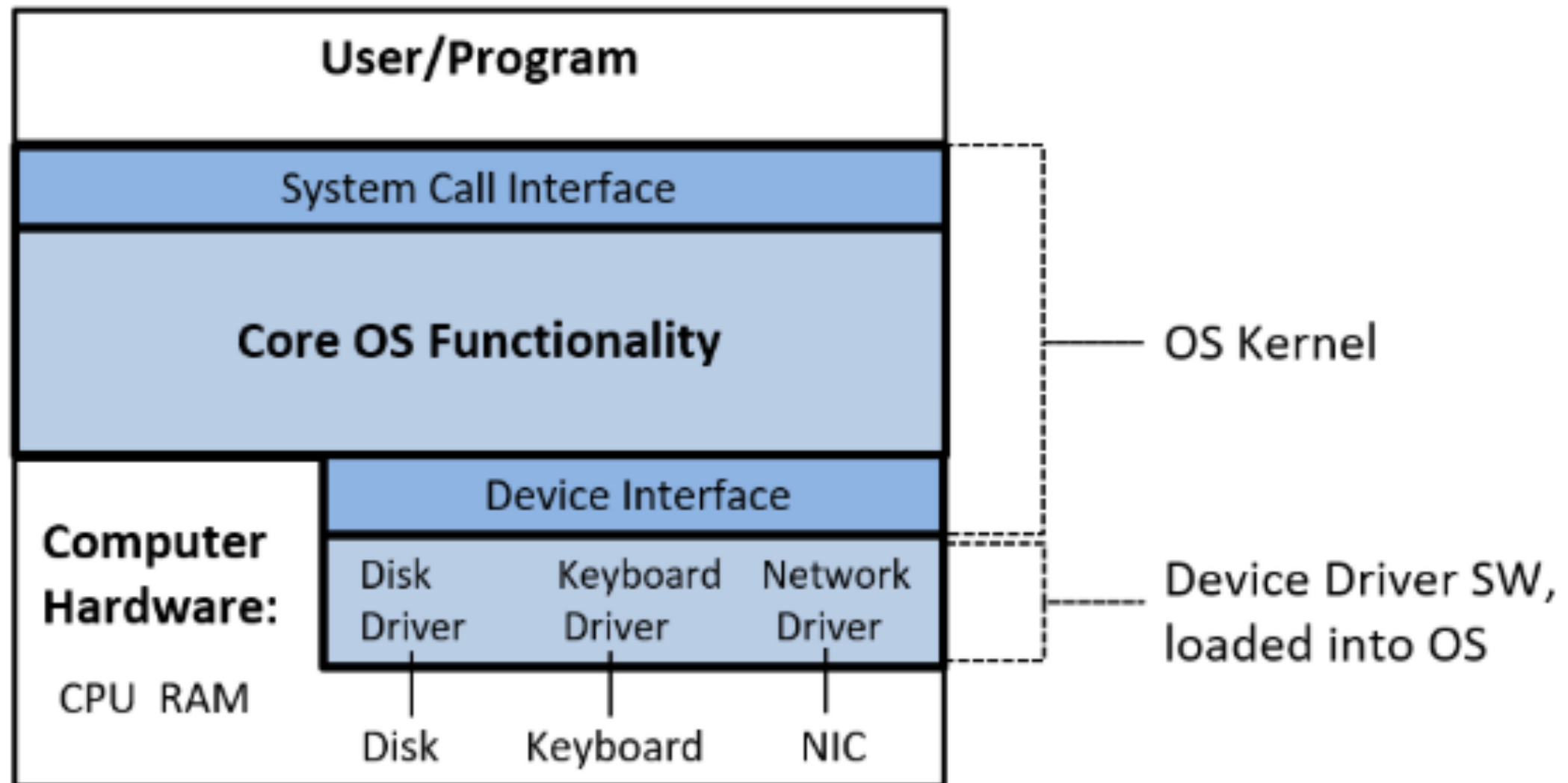
The Kernel

- Core OS functionality
 - Managing hardware
 - Managing OS abstractions exported to users
 - Such as files
 - Implementing interfaces to applications and devices
- **Mechanisms**
 - Enable the hardware to run processes
- **Policies**
 - Govern abstractions, such as deciding which process gets CPU time next

System Calls

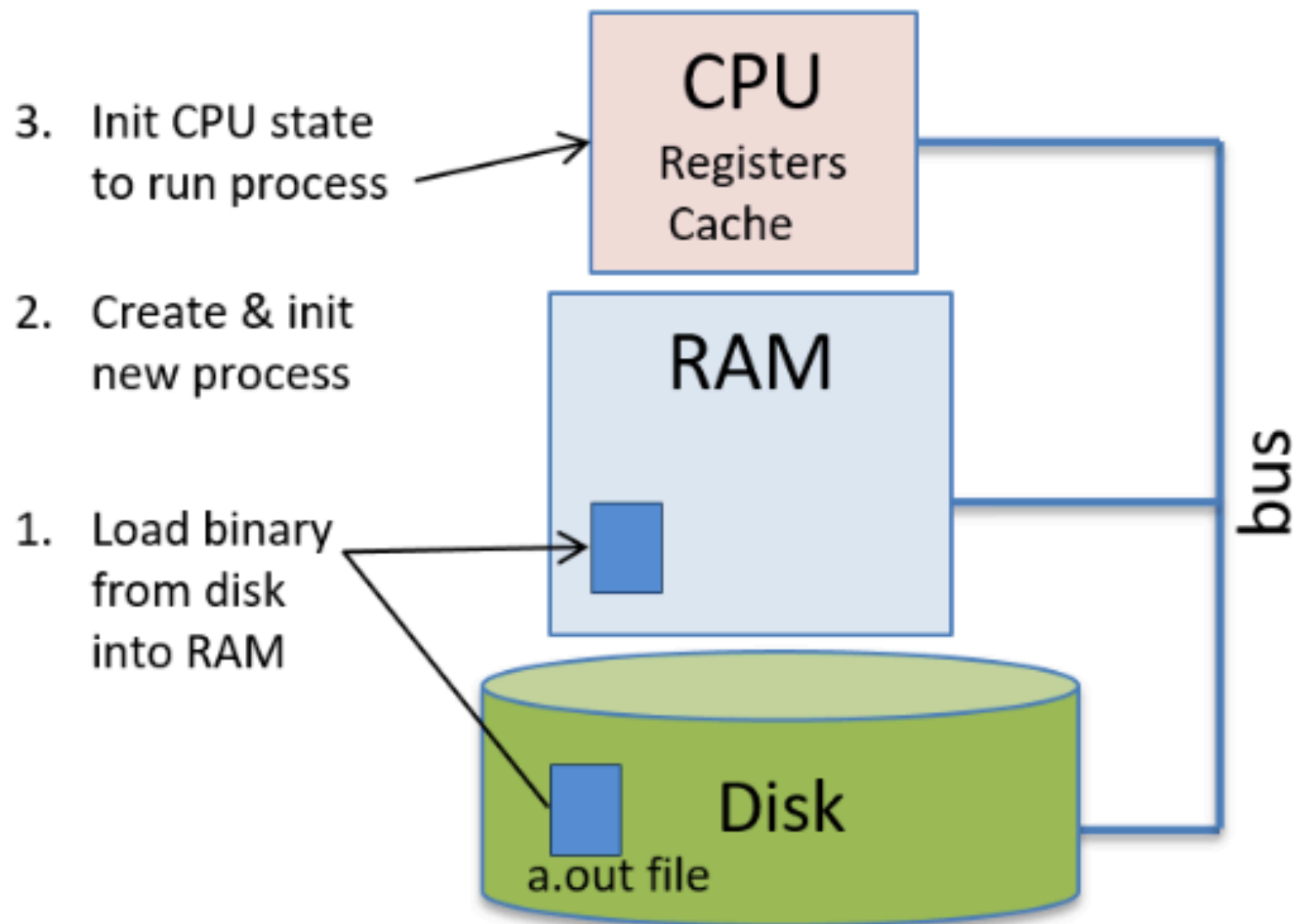
- A programming interface for users of the system
- Part of the kernel
- Example:
 - Get time of day from the **gettimeofday** system call

Kernel Components



13.1. Booting and Running

Starting a Program Running on System



13.1.1. OS Booting

- **Booting**
 - Runs firmware (nonvolatile code)
 - **BIOS** (Basic Input/Output System) or
 - **UEFI** (Unified Extensible Firmware Interface)
 - Loads boot block into RAM and starts it running
- That loads the rest of the OS from disk
 - Discovers and initializes hardware resources
 - Initializes data structures and abstractions to make the system ready for users

13.1.2. Getting the OS to Do Something: Interrupts and Traps

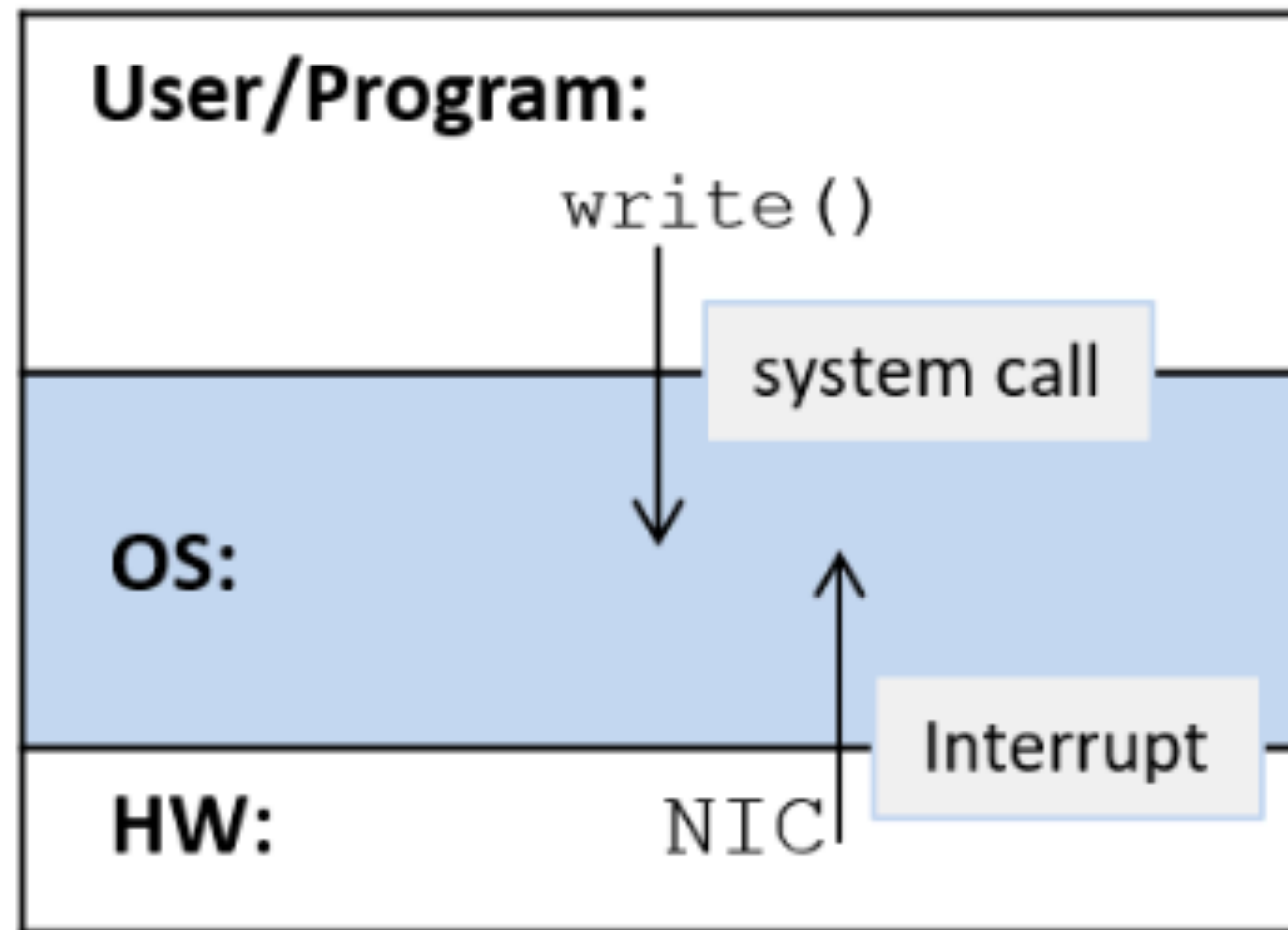


Figure 2. In an interrupt-driven system, user-level programs make system calls, and hardware devices issue interrupts to initiate OS actions.

Interrupts and Traps

- Interrupts that come from the hardware layer
 - such as when a NIC receives data from the network,
 - are called ***hardware interrupts***, or just ***interrupts***
- Interrupts that come from the software layer
 - such as when an application makes a system call
 - are called ***traps***
- Exceptions from either layer may also interrupt the OS
 - Such as disk read error, or divide by zero

Example: Write system call

```
GNU nano 3.2      hello.asm

section .text
global _start

_start:

    mov     edx, len
    mov     ecx, msg
    mov     ebx, 1
    mov     eax, 4
    int     0x80

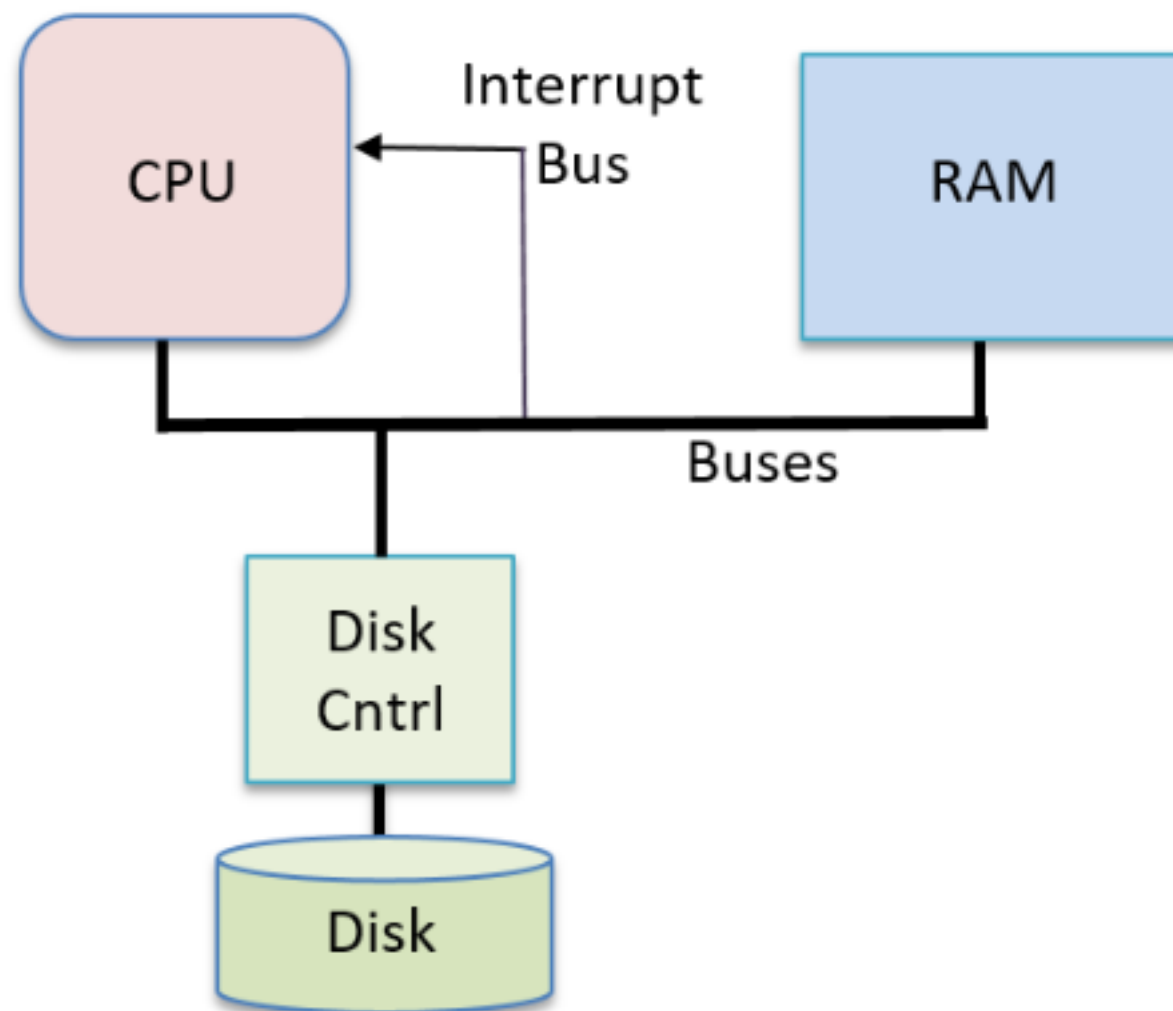
    mov     eax, 1
    int     0x80

section .data

msg     db     "Hello World!"
len     equ    $ - msg
```

Interrupt Bus

- Sends a signal from hardware to the CPU
- CPU runs *handler* code



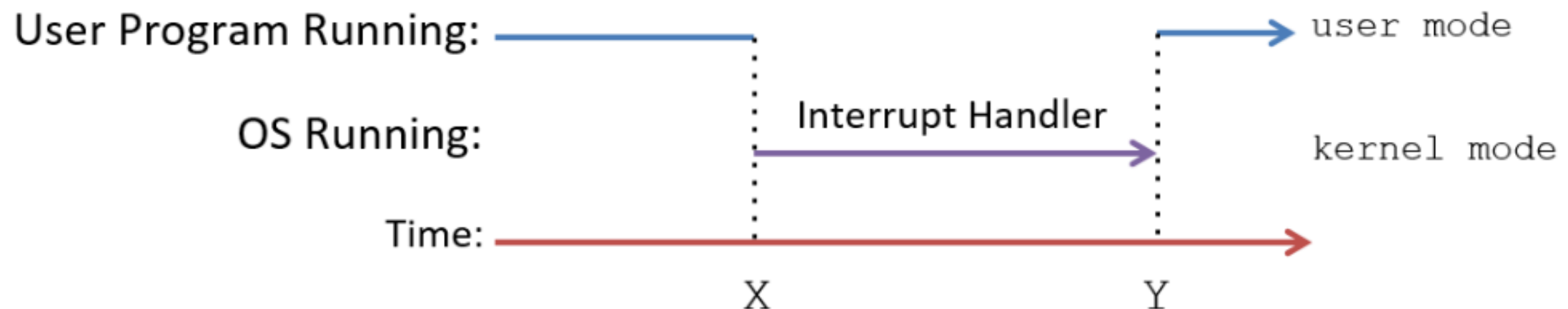
User Mode and Kernel Mode

- **User Mode**

- Restricted access to hardware
- Cannot access OS instructions or data
- Can only access memory allocated to it by the OS

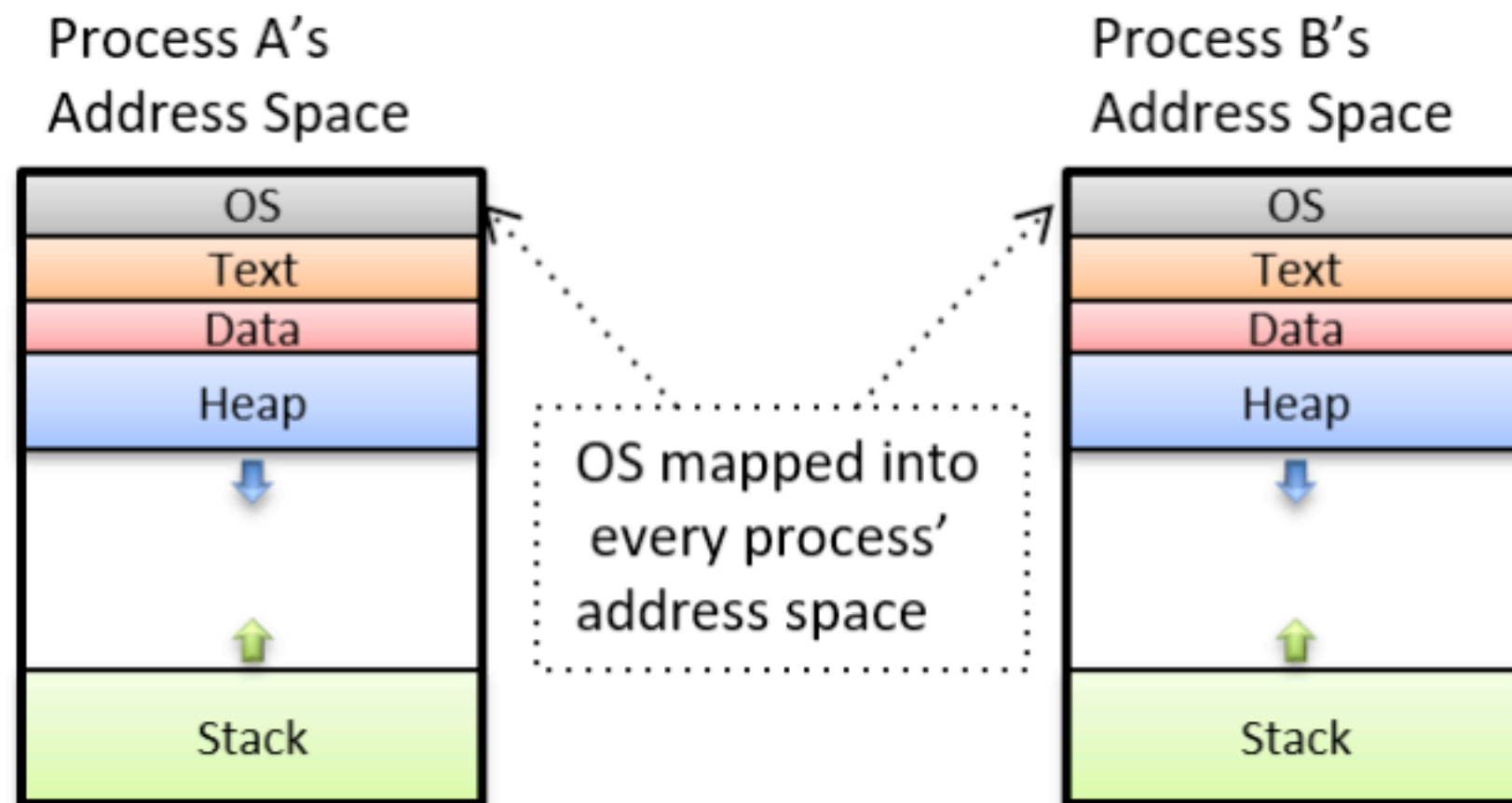
- **Kernel Mode**

- Can execute any instructions
- Can access any memory location
- Can access all hardware directly



Process Address Space

- The Kernel was mapped into the top of every process's address space
- This system was replaced in 2018 to mitigate the Meltdown hardware exploit



13.2. Processes

Processes

- **A process**
 - represents an instance of a program running in the system, which includes
 - the program's binary executable code, data, and execution **context**
- The **context** tracks the program's execution
 - maintaining its register values, stack location, and the instruction it is currently executing
- **Multiprogramming** systems
 - support multiple processes at the same time

Lone View

- OS isolates processes from one another
- Gives each process the illusion that it's controlling the entire machine

13.2.1. Multiprogramming and Context Switching

- **Timesharing**
 - OS gives each process a **time slice** or **quantum**
 - a few milliseconds of CPU time
- Processes run **concurrently**
 - their executions overlap in time

Context Switching

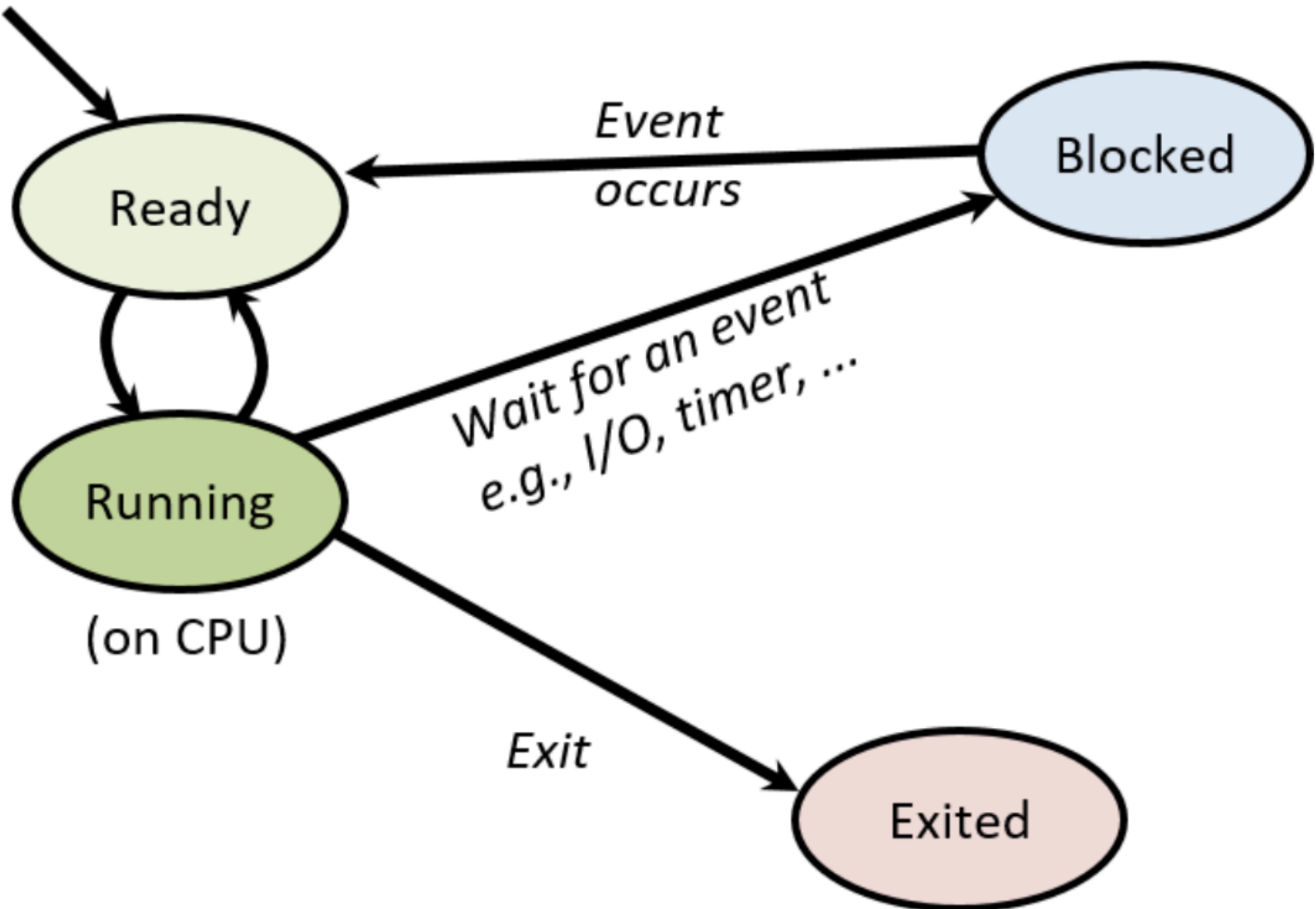
- OS saves **context** of the currently running process
 - register values (PC, stack pointers, general-purpose register, condition codes, etc.)
 - memory state
 - other states like open files
- OS restores the saved context from another process
 - resumes execution from the instruction where it left off

13.2.2. Process State

- **Process id (PID)**
 - unique identifier for a process
 - **ps** command shows PID values
- **Address space** information
- **Execution state**
 - CPU register values, stack location
- **Resources**
 - e.g., open files
- **Process state**
 - determines its eligibility for execution on the CPU

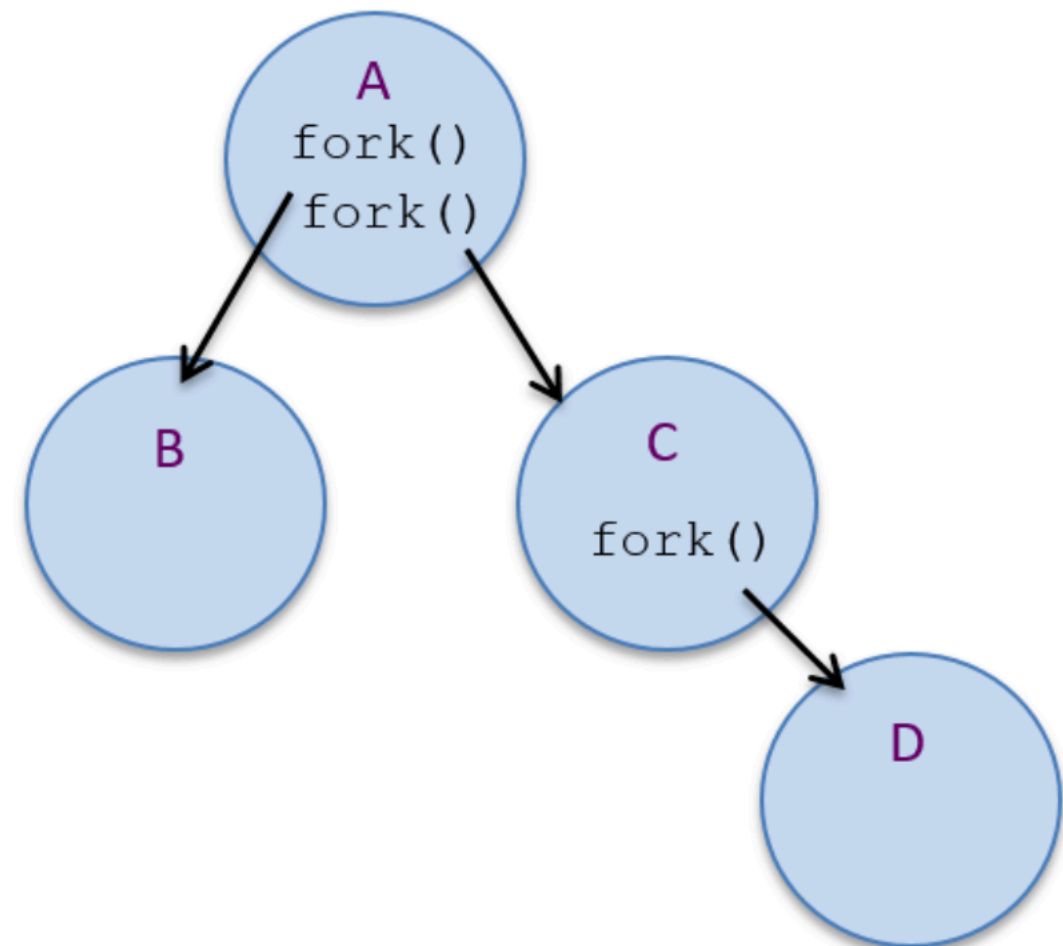
Process States

new process



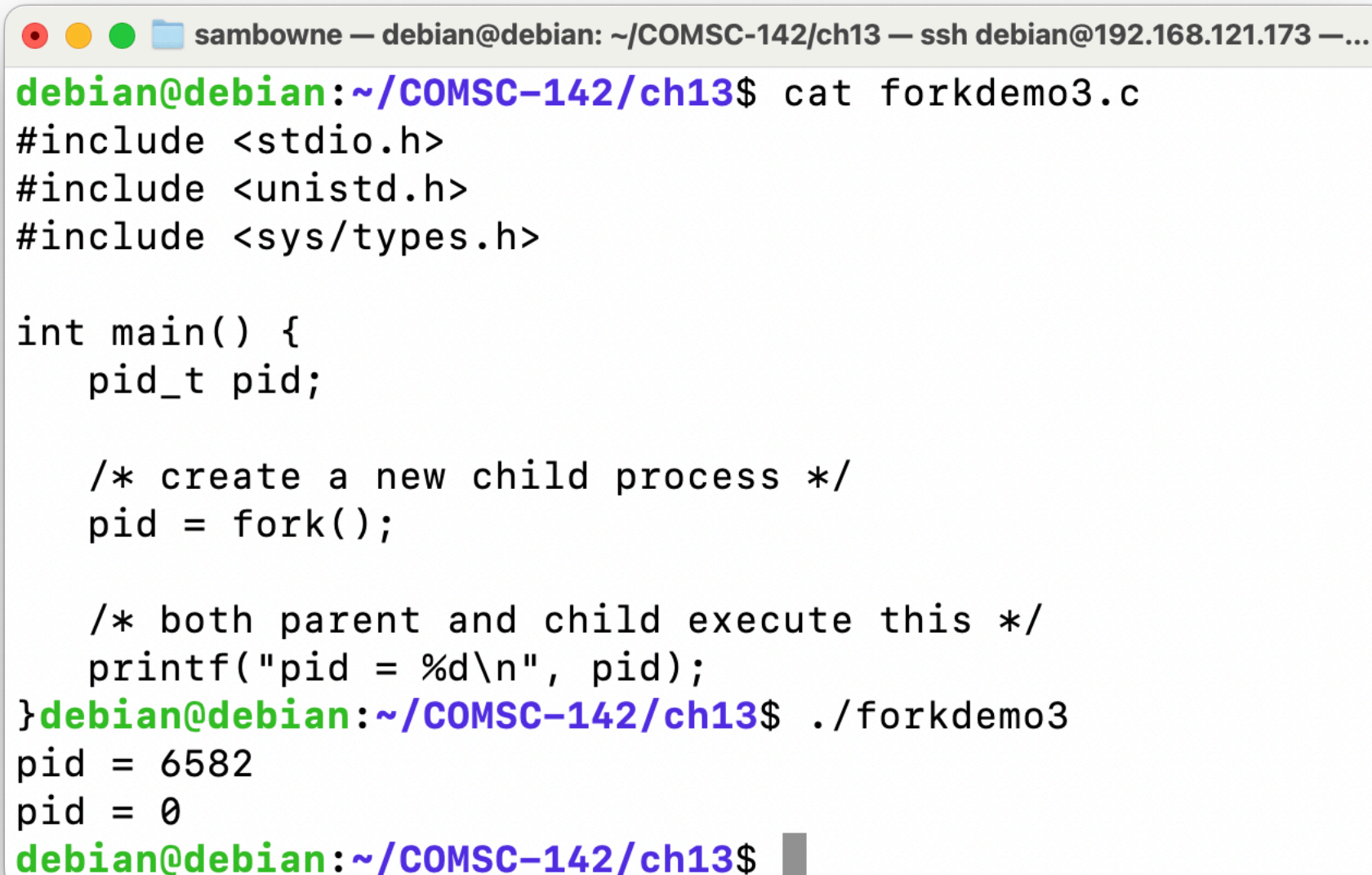
13.2.3. Creating (and Destroying) Processes

- The **fork** system call
 - creates a new process
- **Parent** process
 - the process calling **fork**
- **Child** process
 - the new process it creates
- **init**
 - The first process
 - Created at boot time



Fork Demo

- `wget https://samsclass.info/COMSC-142/proj/forkdemo3.c`
- `gcc -o forkdemo3 forkdemo3.c`



```
sambowne — debian@debian: ~/COMSC-142/ch13 — ssh debian@192.168.121.173 —...
debian@debian:~/COMSC-142/ch13$ cat forkdemo3.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

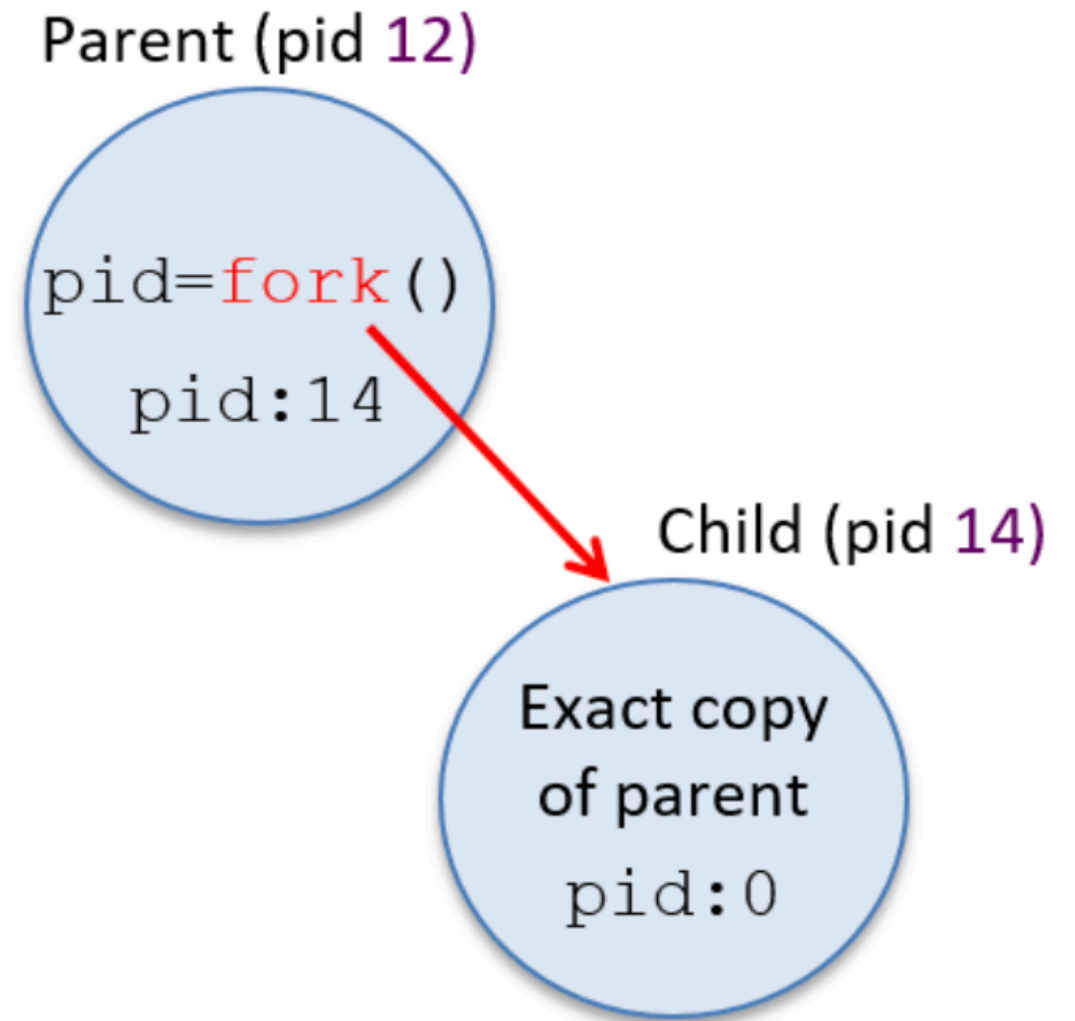
int main() {
    pid_t pid;

    /* create a new child process */
    pid = fork();

    /* both parent and child execute this */
    printf("pid = %d\n", pid);
}
debian@debian:~/COMSC-142/ch13$ ./forkdemo3
pid = 6582
pid = 0
debian@debian:~/COMSC-142/ch13$
```

Fork's Return Value

- fork returns 0 to the child process and the child's PID value (14) to the parent.



Fork Demo

- `wget https://samsclass.info/COMSC-142/proj/forkdemo4.c`

```
sambowne — debian@debian: ~/COMSC-142/ch13 — ssh debian@192.168.121.173 — 73x26
int main(void) {

    pid_t pid, mypid;

    printf("A\n");

    pid = fork();    /* create a new child process */

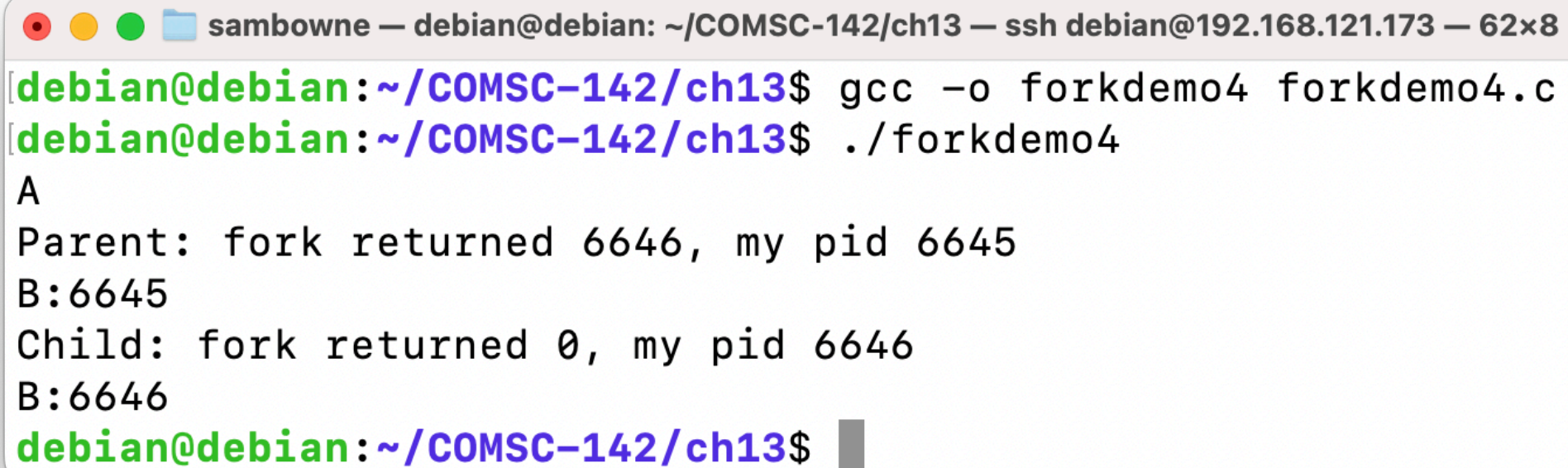
    if(pid == -1) { /* check and handle error return value */
        printf("fork failed!\n");
        exit(pid);
    }

    if (pid == 0) { /* the child process */
        mypid = getpid();
        printf("Child: fork returned %d, my pid %d\n", pid, mypid);
    } else { /* the parent process */
        mypid = getpid();
        printf("Parent: fork returned %d, my pid %d\n", pid, mypid);
    }

    printf("B:%d\n", mypid);

    return 0;
}
[debian@debian:~/COMSC-142/ch13$ ./forkdemo4]
```


Fork Demo



A terminal window titled "sambowne — debian@debian: ~/COMSC-142/ch13 — ssh debian@192.168.121.173 — 62x8". The prompt is "debian@debian:~/COMSC-142/ch13\$". The user enters "gcc -o forkdemo4 forkdemo4.c" and then "./forkdemo4". The program outputs "A", "Parent: fork returned 6646, my pid 6645", "B:6645", "Child: fork returned 0, my pid 6646", and "B:6646". The prompt returns to "debian@debian:~/COMSC-142/ch13\$".

```
sambowne — debian@debian: ~/COMSC-142/ch13 — ssh debian@192.168.121.173 — 62x8
[debian@debian:~/COMSC-142/ch13$ gcc -o forkdemo4 forkdemo4.c ]
[debian@debian:~/COMSC-142/ch13$ ./forkdemo4 ]
A
Parent: fork returned 6646, my pid 6645
B:6645
Child: fork returned 0, my pid 6646
B:6646
debian@debian:~/COMSC-142/ch13$
```

Possible Order of Outputs

- Parent and child run independently

Table 1. All Six Possible Orderings of Example Program Output

Option 1	Option 2	Option 3	Option 4	Option 5	Option 6
A	A	A	A	A	A
Parent...	Parent...	Parent...	Child...	Child...	Child...
Child...	Child...	B:12	Parent...	Parent...	B:14
B:12	B:14	Child...	B:12	B:14	Parent...
B:14	B:12	B:14	B:14	B:12	B:12

13.2.4. exec

- Overlays the calling process's image with a new image from a binary executable file.

```
int execvp(char *filename, char *argv[]);
```

execvp Demo

- `wget https://samsclass.info/COMSC-142/proj/execvpdemo.c`

```
sambowne — debian@debian: ~/COMSC-142/ch13 — ssh debian@192.168.121.173 — 69x21

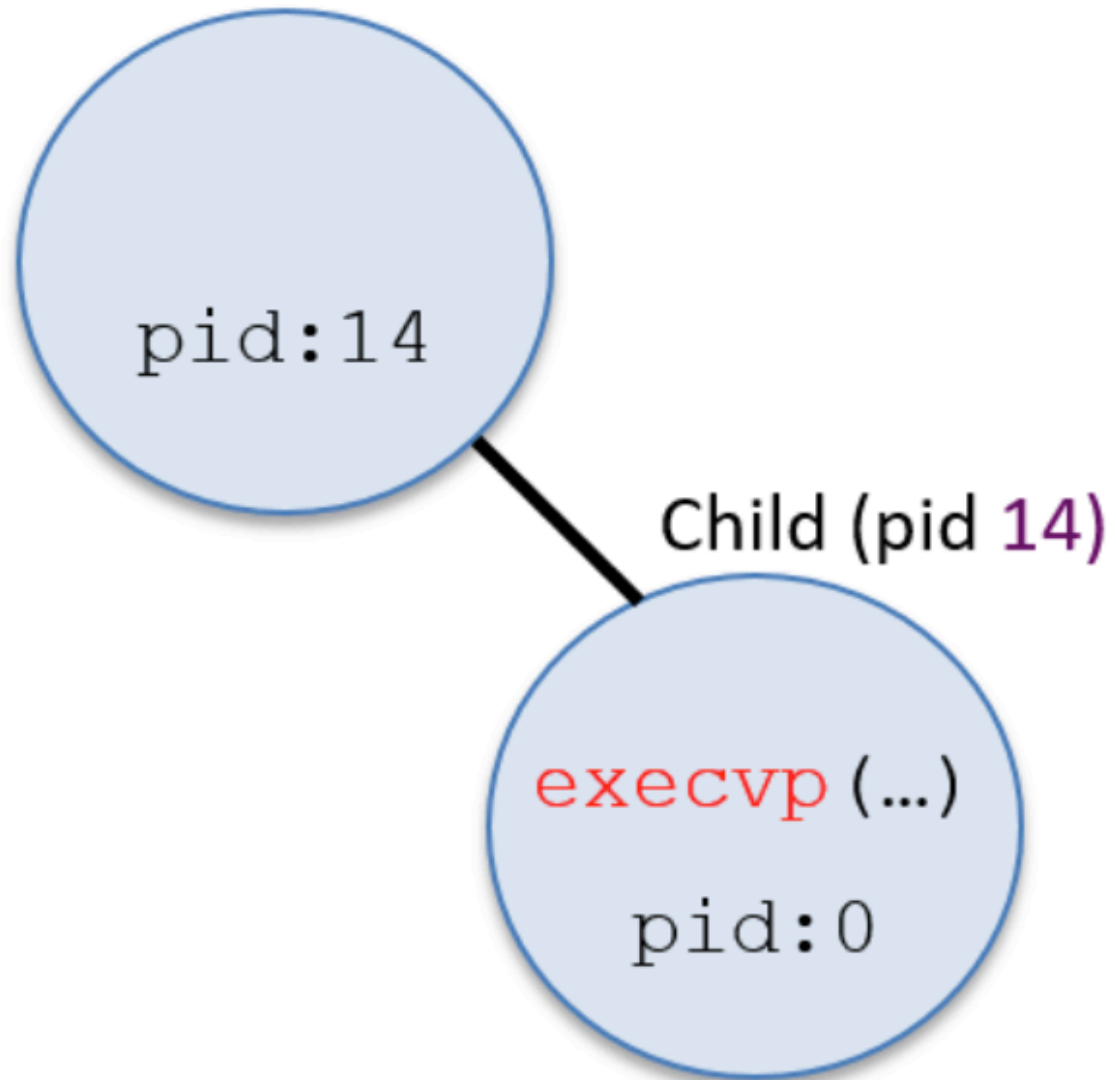
int main() {
    pid_t pid;
    int ret;
    char *argv[2];

    argv[0] = "ls";
    argv[1] = NULL;

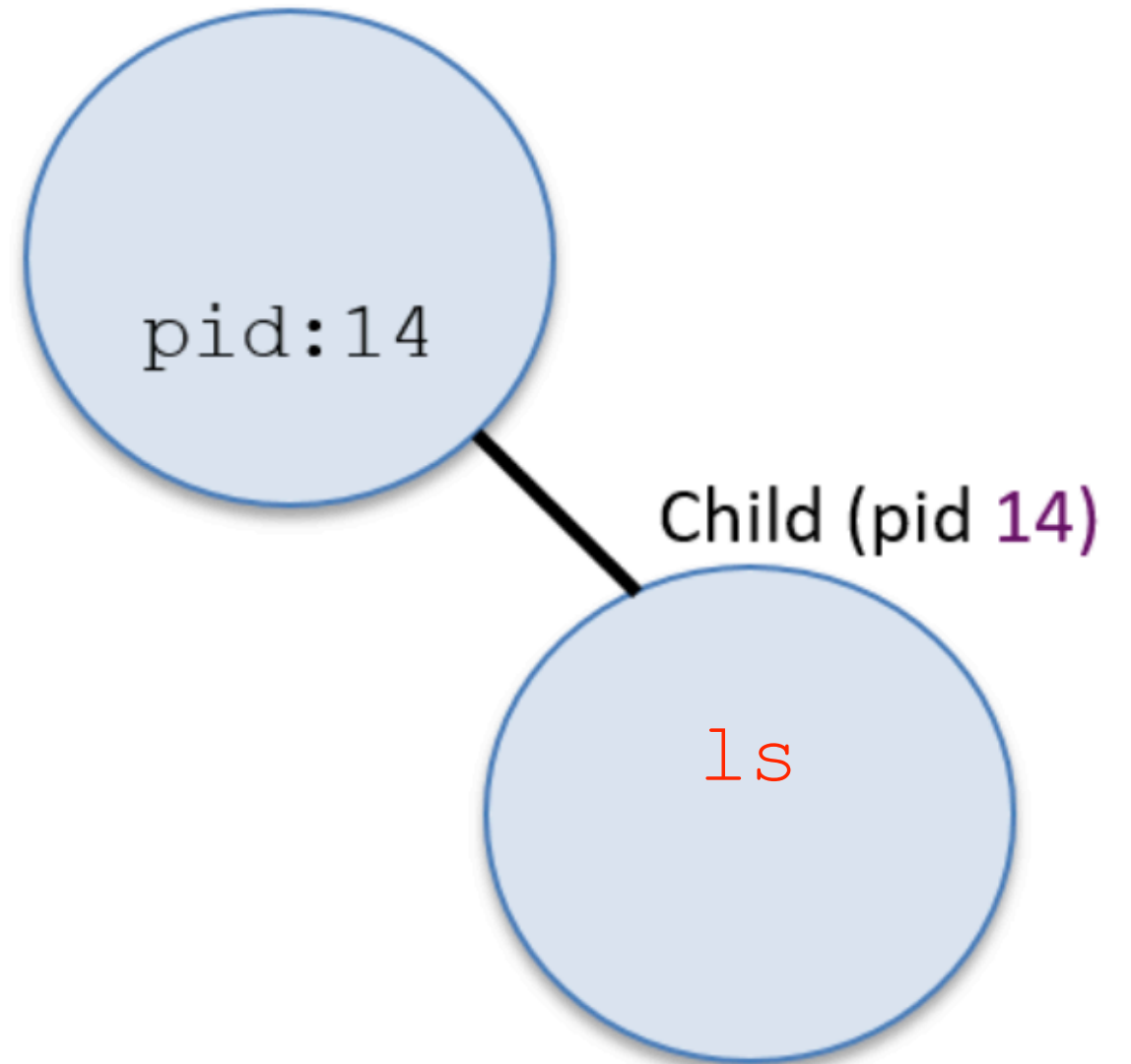
    pid = fork();
    if (pid == 0) { /* child process */
        ret = execvp("ls", argv);
        if (ret < 0) {
            printf("Error: execvp returned!!!\n");
            exit(ret);
        }
    }
}

debian@debian:~/COMSC-142/ch13$ ./execvpdemo
debian@debian:~/COMSC-142/ch13$ execvpdemo  execvpdemo.c  forkdemo3
forkdemo3.c  forkdemo4  forkdemo4.c
debian@debian:~/COMSC-142/ch13$
```

Parent (pid 12)

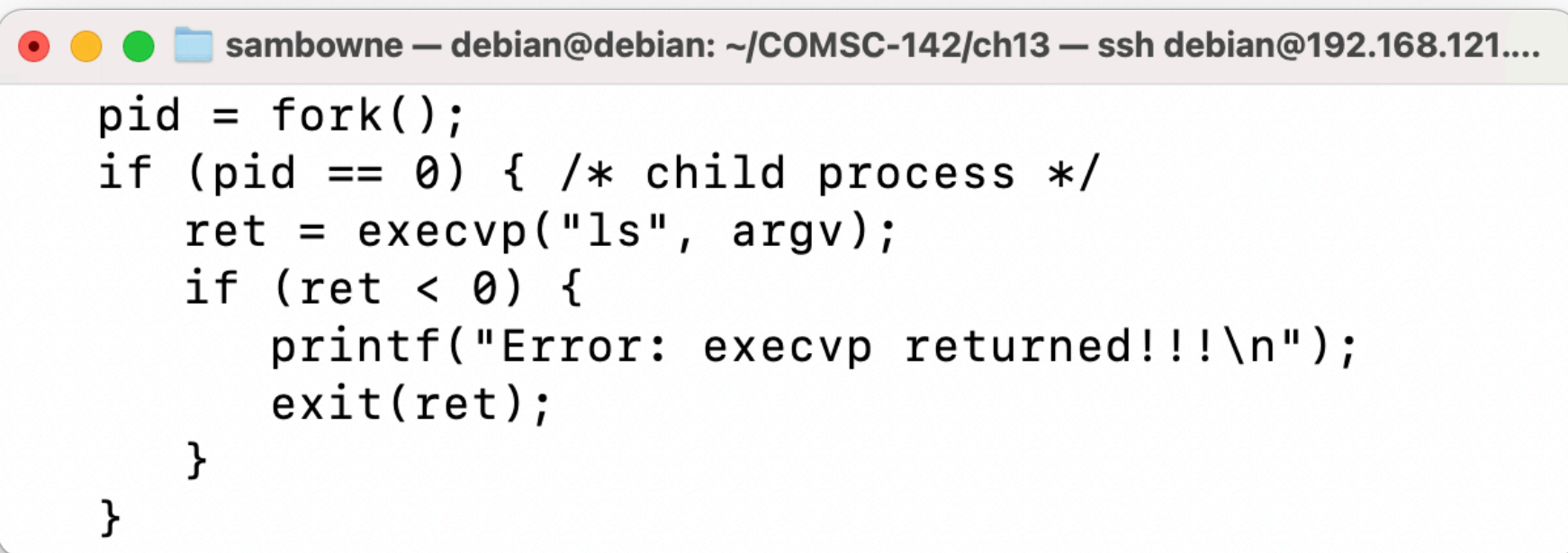


Parent (pid 12)



execvp Return

- The program should not return from execvp
- It should be replaced by the new executable
- With the same pid

A terminal window with a title bar containing three colored circles (red, yellow, green) and a folder icon, followed by the text "sambowne — debian@debian: ~/COMSC-142/ch13 — ssh debian@192.168.121....". The terminal displays a C code snippet. The code starts with "pid = fork();" followed by an "if (pid == 0) {" block for the child process. Inside this block, it calls "ret = execvp("ls", argv);". Then, it has another "if (ret < 0) {" block containing "printf("Error: execvp returned!!!\n");" and "exit(ret);". The "if (ret < 0) {" block is closed with "}", and the "if (pid == 0) {" block is closed with "}". Finally, the entire code block is closed with a closing curly brace "}" at the bottom.

```
pid = fork();
if (pid == 0) { /* child process */
    ret = execvp("ls", argv);
    if (ret < 0) {
        printf("Error: execvp returned!!!\n");
        exit(ret);
    }
}
```

Kahoot!

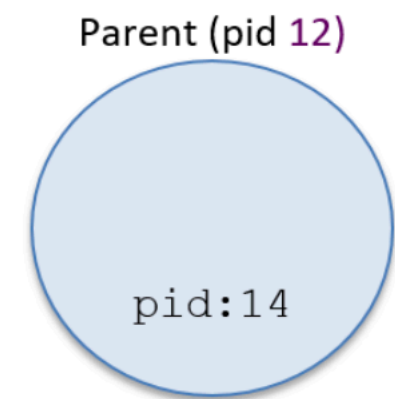
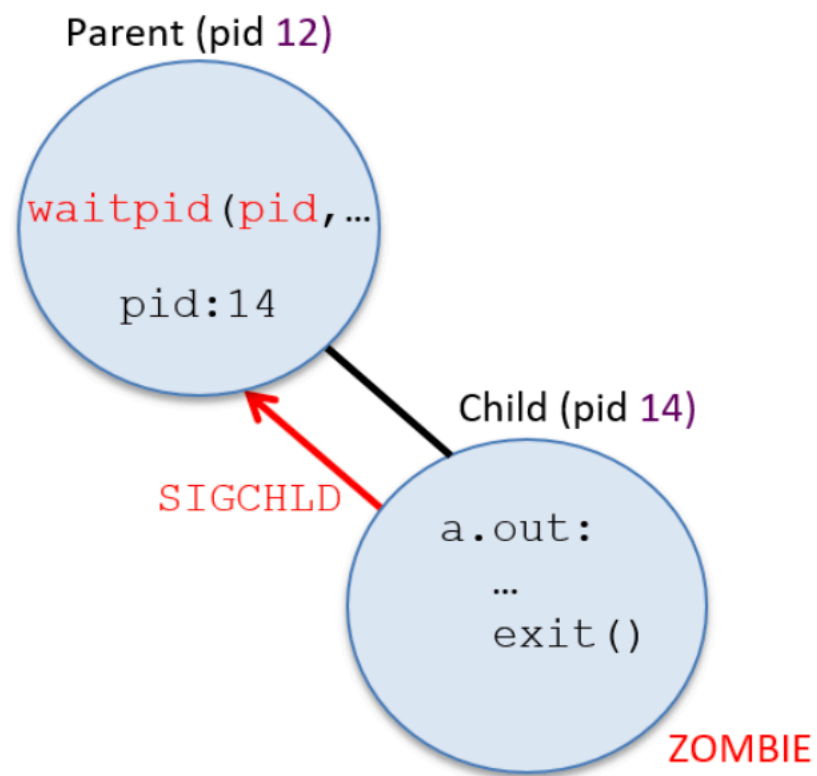
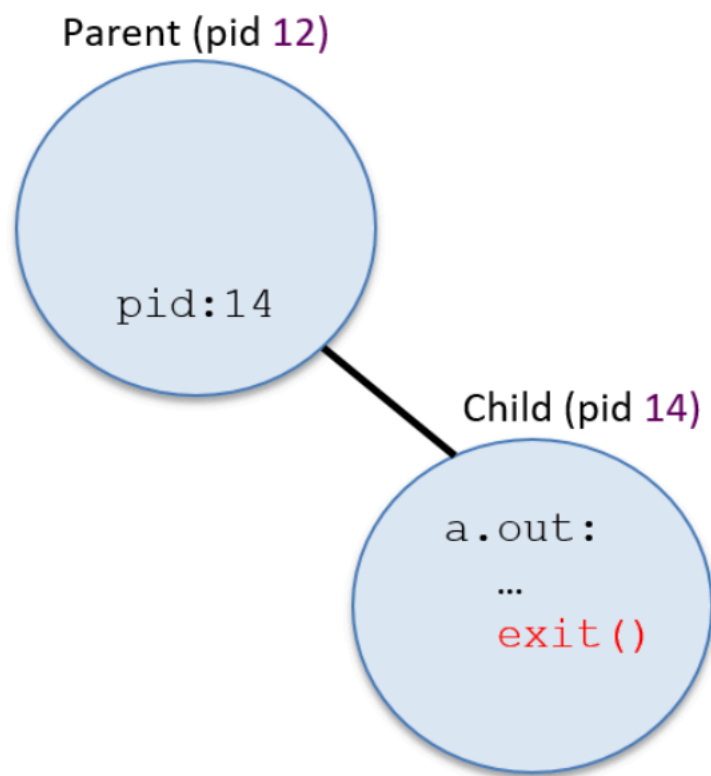
Ch13b

13.2.5. exit and wait

- Processes can be triggered to exit by:
 - Completing all of its application code
 - returning from its main function leads to a process invoking the exit system call
 - Perform an invalid action
 - such as dividing by zero or dereferencing a null pointer
 - Receiving a signal
 - from the OS or another process
 - telling it to exit
- In fact, dividing by zero and NULL pointer dereferences result in the OS sending the process SIGFPE and SIGSEGV signals telling it to exit

Signals

- Signals are software interrupts
 - **SIGSEGV** for null pointer dereference
 - **SIGKILL** from a parent calls **exit** in a child process
 - Pressing Ctrl+C sends **SIGINT**, which also calls **exit**
 - After calling **exit**, the OS delivers a **SIGCHLD** signal to the parent process
 - The child becomes a **zombie** process
 - it moves to the Exited state and can no longer run on the CPU
 - A parent process **reaps** its zombie child (cleans up the rest of its state from the system) by calling the **wait** system call



Blocking

- If the child is still executing when the parent process calls **wait**
 - the parent **blocks** until the child exits
 - (the parent enters the **Blocked** state waiting for the **SIGCHLD** signal event to happen)
- The blocking behavior of the parent can be seen if you run a program (a.out) in the foreground of a shell—the shell program doesn't print out a shell prompt until a.out terminates, indicating that the shell parent process is blocked on a call to **wait**, waiting until it receives a **SIGCHLD** from its child process running a.out.



```
sambowne — debian@debian: ~ — ssh debian@192.168.121.173 — 54x5
[debian@debian:~$
[debian@debian:~$
[debian@debian:~$
[debian@debian:~$ sleep 10
```

The image shows a terminal window with a title bar that reads "sambowne — debian@debian: ~ — ssh debian@192.168.121.173 — 54x5". The terminal content shows four consecutive shell prompts "[debian@debian:~\$". The fourth prompt is followed by the command "sleep 10". A grey cursor block is visible at the end of the fourth line, indicating the shell is waiting for the command to complete before printing the next prompt.

Backgrounding

- If the parent implements a **SIGCHLD** signal handler that contains the call to **wait**
 - then the parent only calls **wait** when there is an exited child process to **reap**
 - and thus it doesn't block on a **wait** call.
- This behavior can be seen by running a program in the background in a shell (a.out &).

```
sambowne — debian@debian: ~ — ssh debian@192.168.121.173 — 54x8
[debian@debian:~$ sleep 10 &
[1] 7468
[debian@debian:~$ ps
  PID TTY          TIME CMD
 7352 pts/2        00:00:00 bash
 7468 pts/2        00:00:00 sleep
 7469 pts/2        00:00:00 ps
debian@debian:~$
```

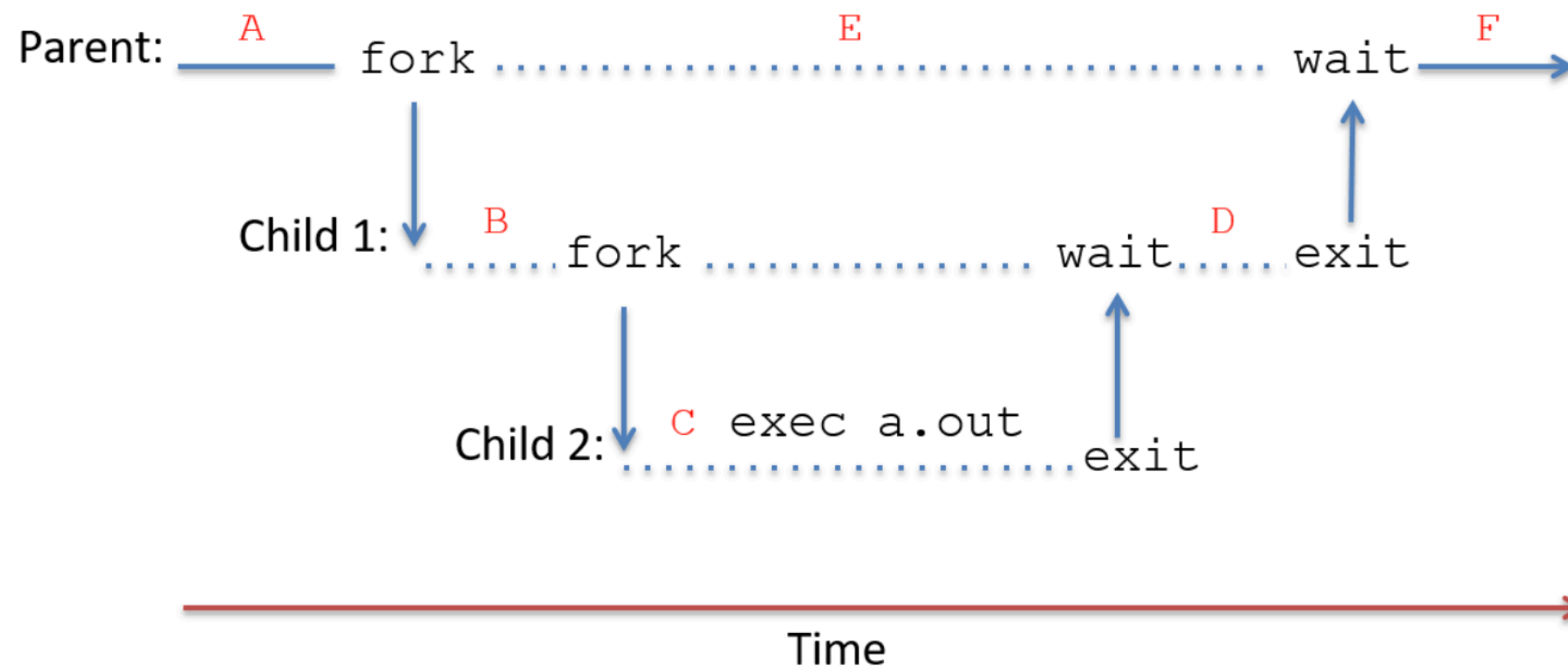
Example

```
pid_t pid1, pid2, ret;
int status;

printf("A\n");

pid1 = fork();
if (pid1 == 0 ) {           /* child 1 */
    printf("B\n");

    pid2 = fork();
    if (pid2 == 0 ){        /* child 2 */
        printf("C\n");
        execvp("a.out", NULL);
    } else {                /* child 1 (parent of child 2) */
        ret = wait(&status);
        printf("D\n");
        exit(0);
    }
} else {                    /* original parent */
    printf("E\n");
    ret = wait(&status);
    printf("F\n");
}
```

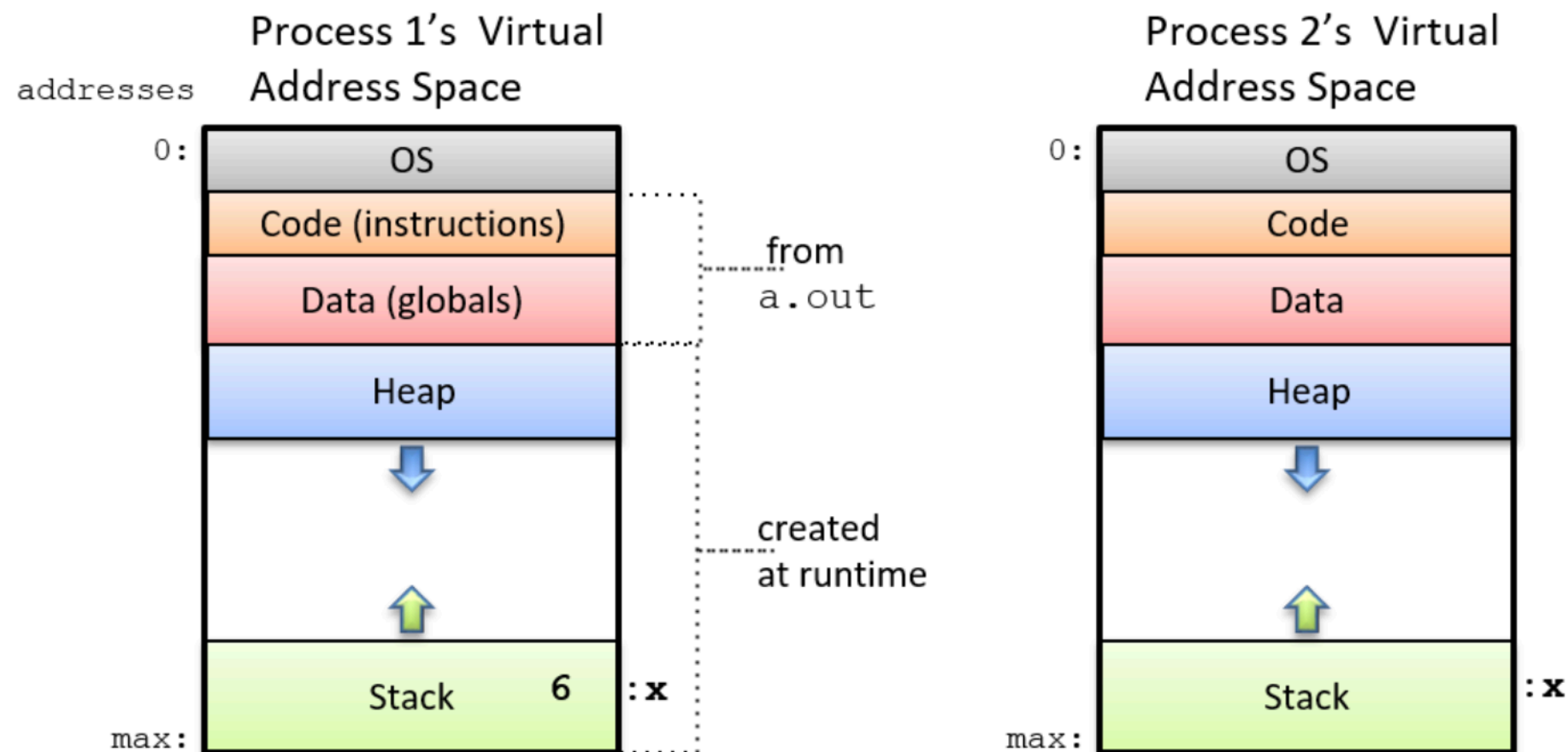
13.3. Virtual Memory

Virtual Memory

- Virtual memory is an abstraction that gives each process its own private, logical address space in which its instructions and data are stored—**lone view**
- Processes cannot access the contents of one another's address spaces
- Some parts of a process's virtual address space come from the binary executable file it's running (e.g., the text portion contains program instructions from the a.out file)
- Other parts of a process's virtual address space are created at runtime (e.g., the stack)

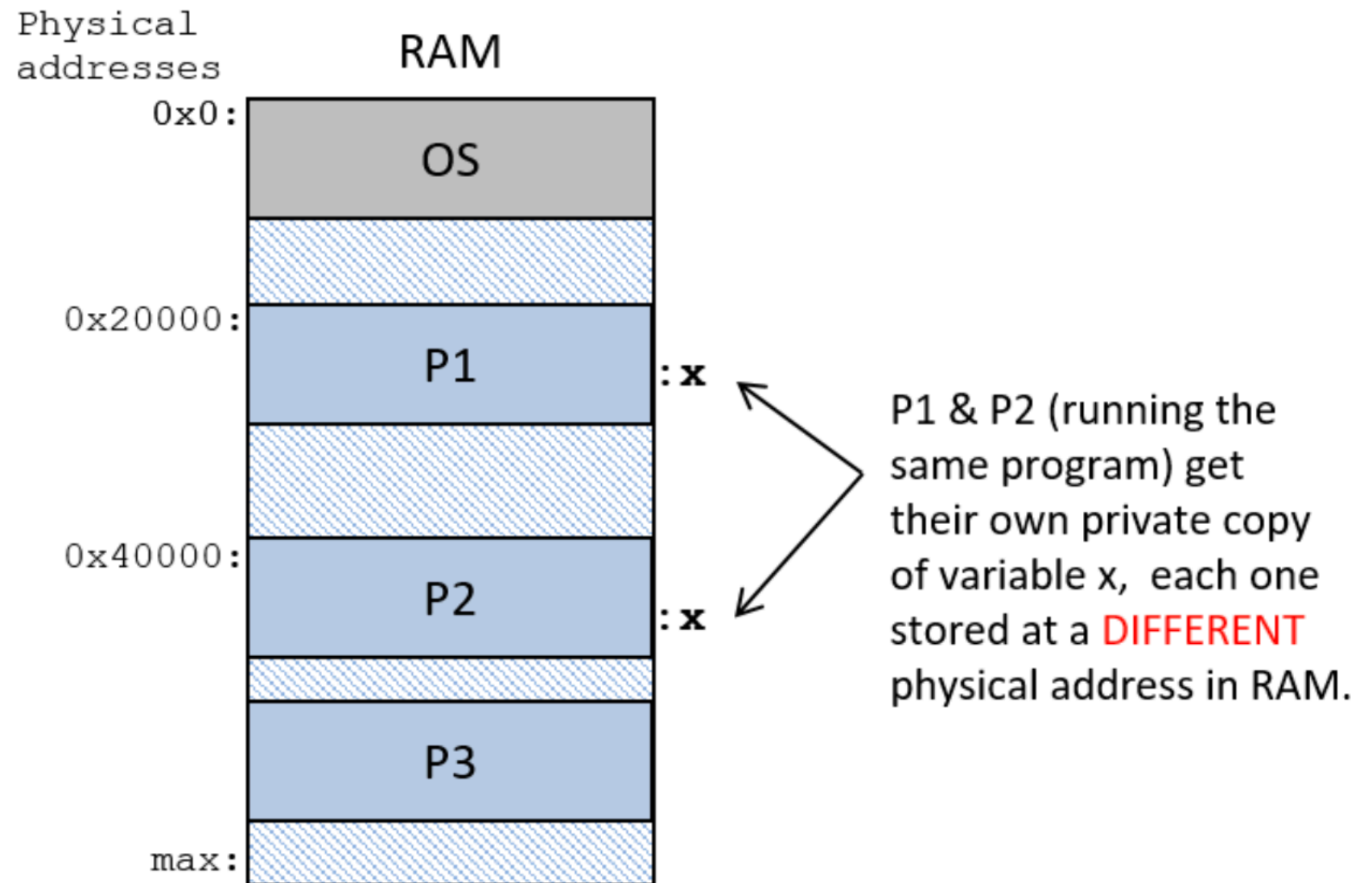
Two Processes

- From the same executable file
- Each instance of the variable **x** is independent

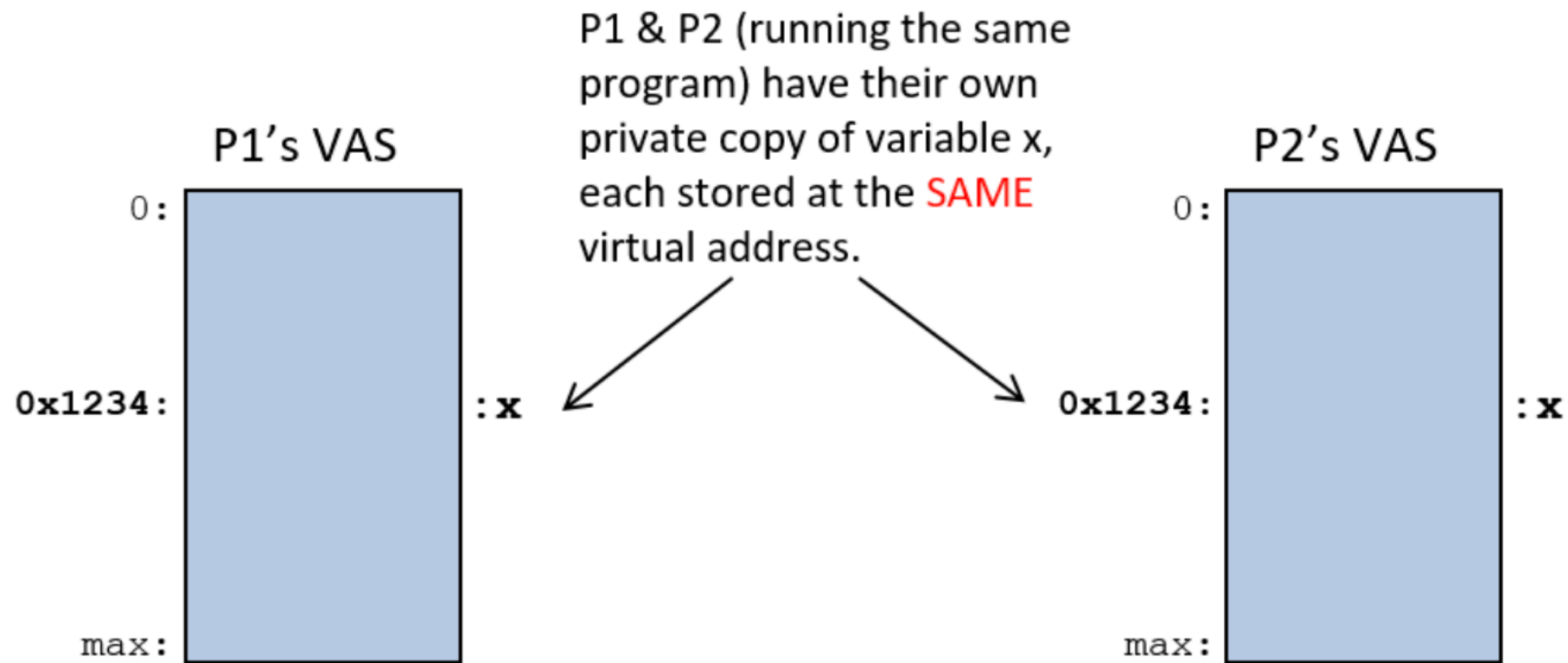


13.3.1. Memory Addresses

- **Virtual addresses**
 - locations in a process's virtual address space
- **Physical addresses**
 - locations in physical memory (RAM)

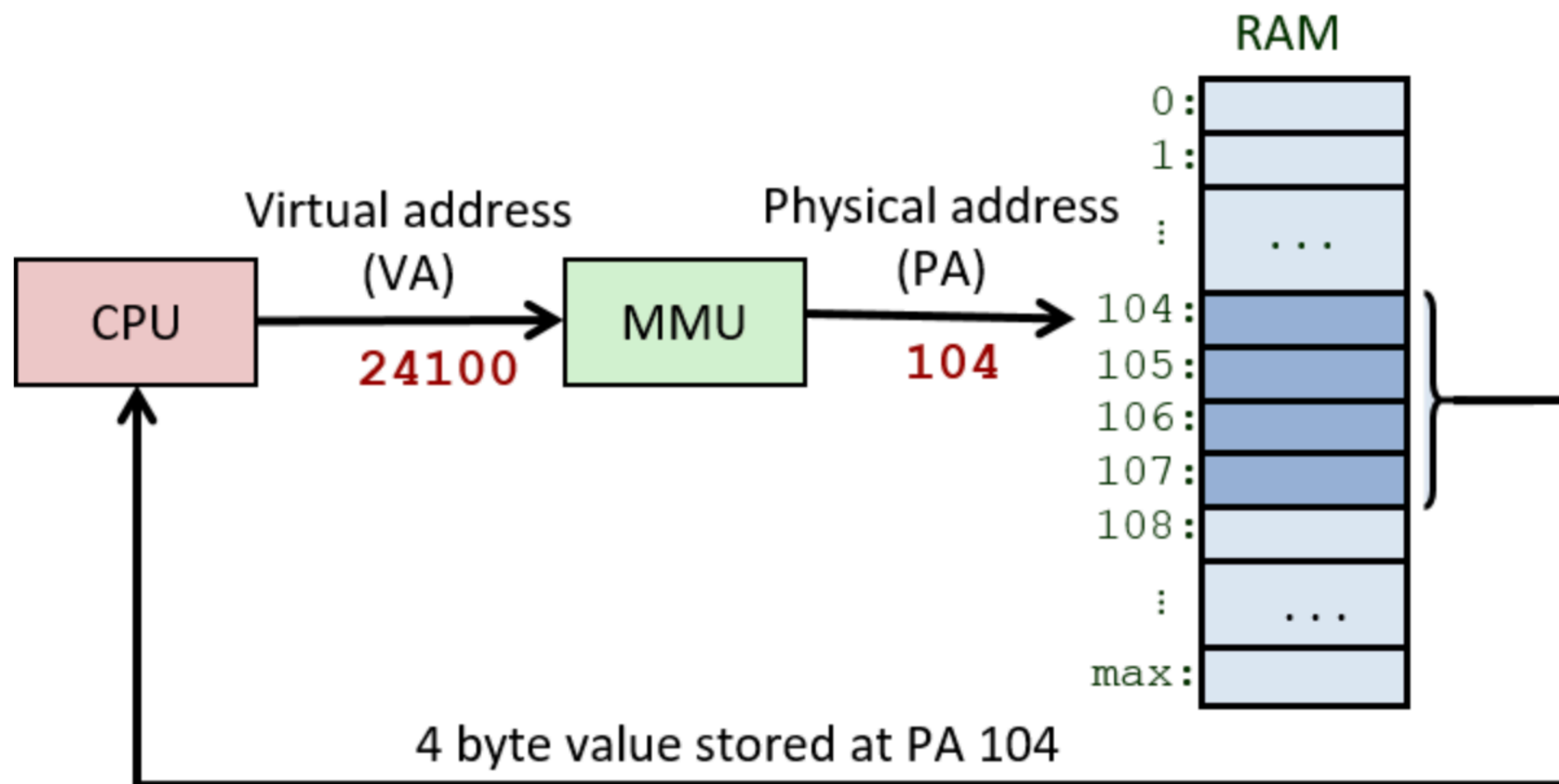


Virtual Addresses



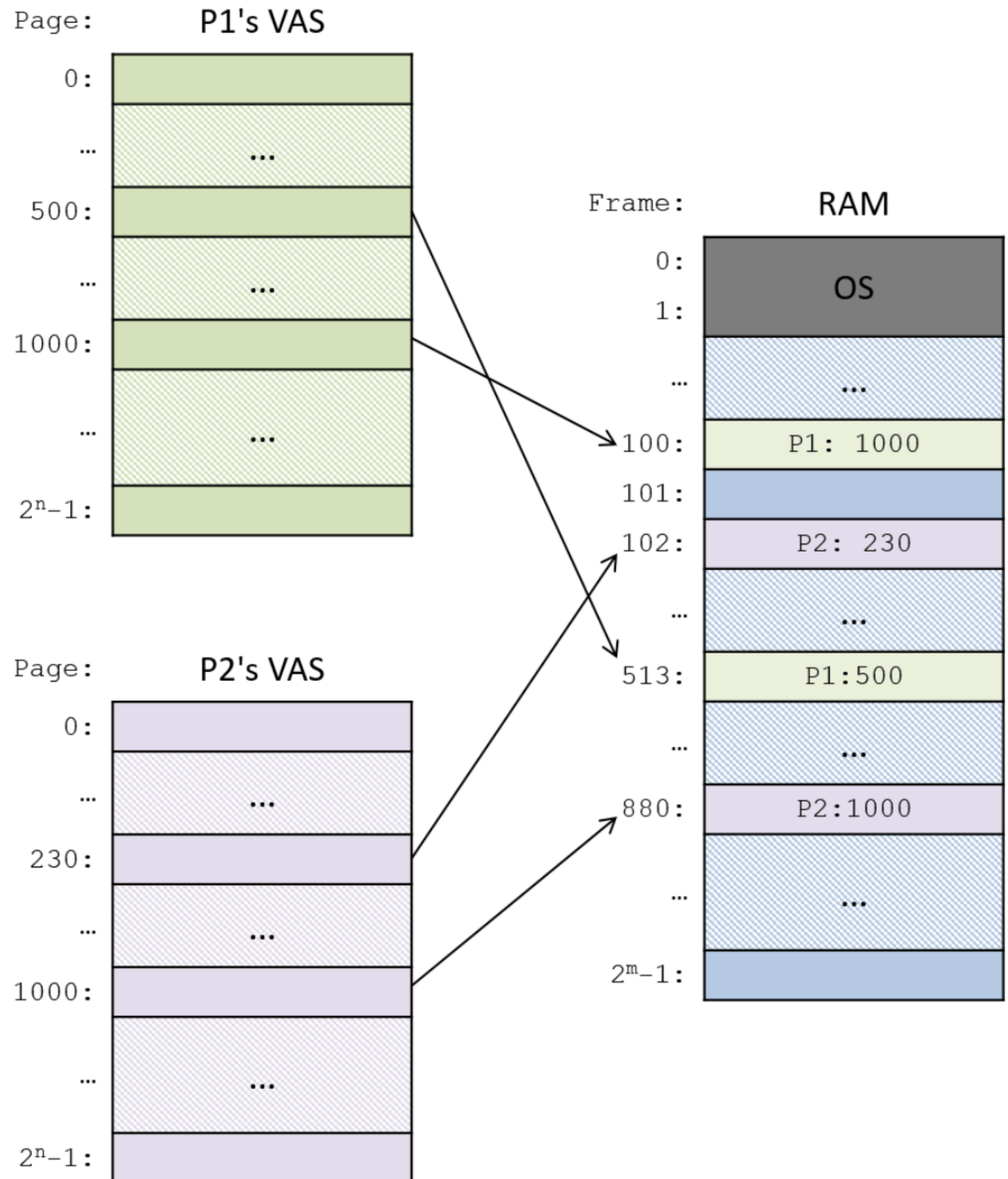
13.3.2. Virtual Address to Physical Address Translation

- Memory Management Unit (MMU) is the part of the computer hardware that implements address translation
- When the CPU needs to fetch data from physical memory, the virtual address is first translated by the MMU to a physical address that is used to address RAM.



13.3.3. Paging

- The OS divides the virtual address space of each process into fixed-sized chunks called pages
 - Typically 4 KB
- Physical memory is similarly divided by the OS into page-sized chunks called **frames**



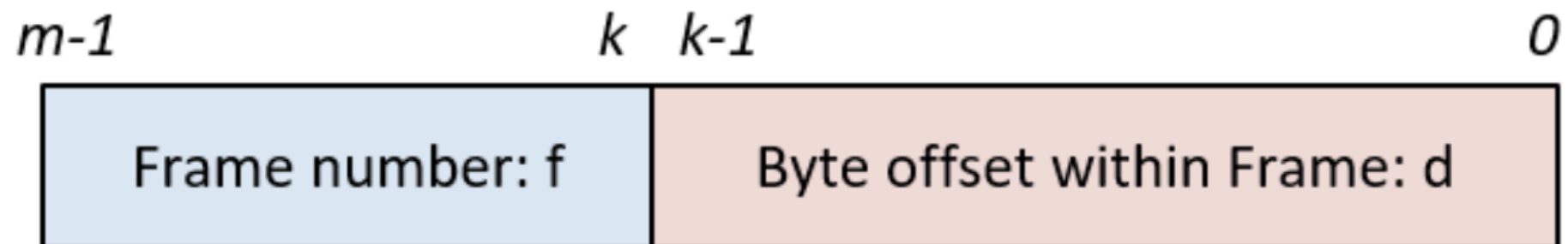
Virtual and Physical Addresses in Paged Systems

- Paged virtual memory systems divide the bits of a **virtual** address into two parts:
 - high-order bits specify the **page number** on which the virtual address is stored
 - low-order bits correspond to the **byte offset** within the page
- Physical addresses are also divided:
 - high-order bits specify the **frame number**
 - low-order bits specify the byte offset within the frame

Virtual Address Space of 2^n bytes, Page size 2^k bytes, VA bits:

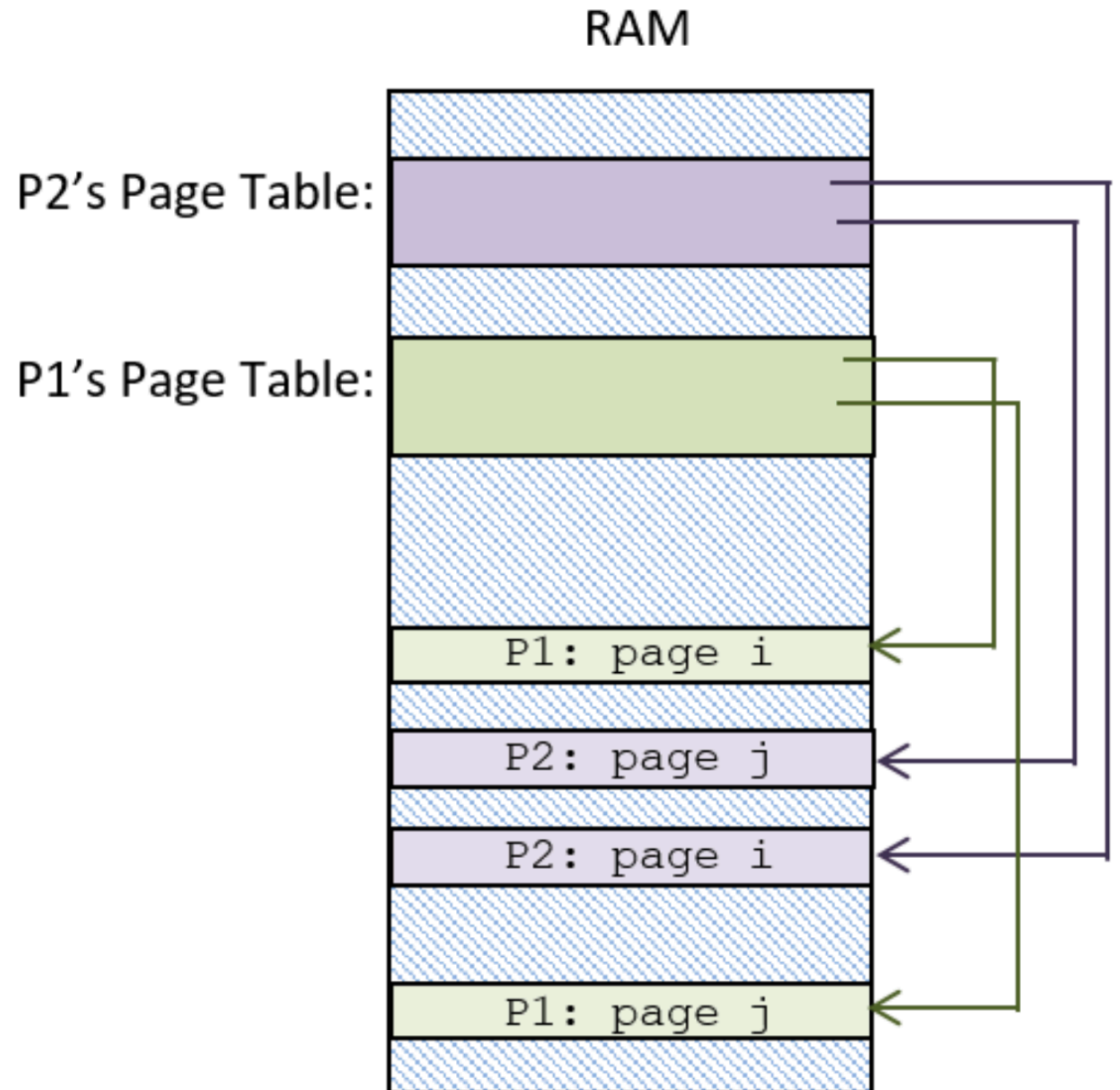


Physical Address Space of 2^m bytes, Page size 2^k bytes, PA bits:



Page Tables for Virtual-to-Physical Page Mapping

- The OS keeps a per-process **page table**
- Stores the process's virtual page number to physical frame number mappings



Page Table Entries

- For each page of virtual memory, the page table stores one **page table entry** (PTE)
 - contains the frame number of physical memory (RAM) storing the virtual page.
 - may also contain other info, such as a valid bit

Page Table Entry:

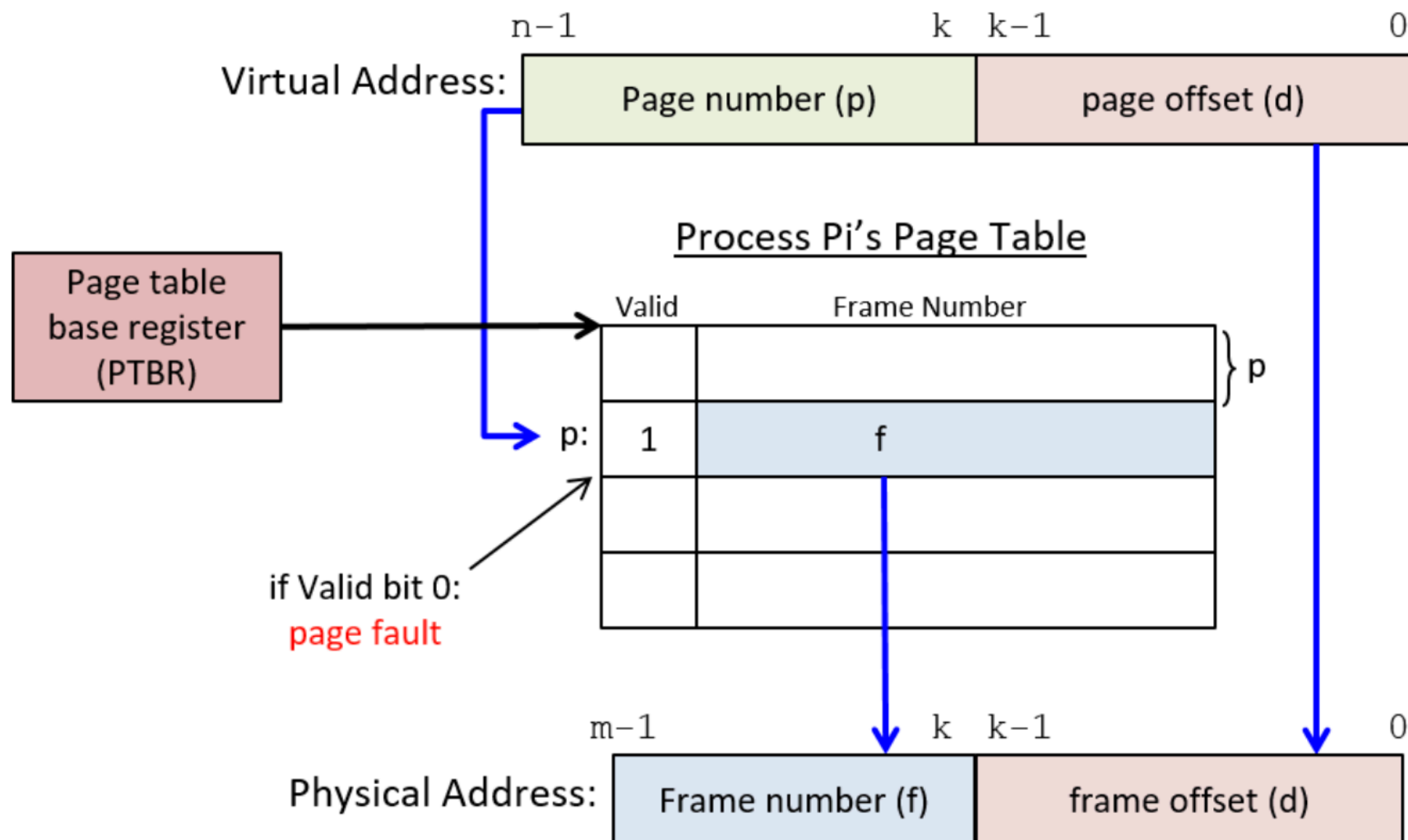
	Valid Bit	Frame Number
For Virtual Page P:	1	Physical Frame # (f) storing Virtual Page P

(ex) PTE for Virtual Page 6 if it is currently stored in RAM Frame 23:

PT[6]:	1	23
--------	---	----

Page Table Base Register (PTBR)

- RAM address of the running process's page table



13.3.4. Memory Efficiency

- Processes tend to access pages of their memory space with a high degree of **temporal** or **spatial locality**
- RAM as a cache for disk:
 - OS allows processes to run only having some of their virtual memory pages in RAM
 - Their other virtual memory pages remain on secondary storage devices, such as disk
 - OS only brings them into RAM when the process accesses addresses on these pages

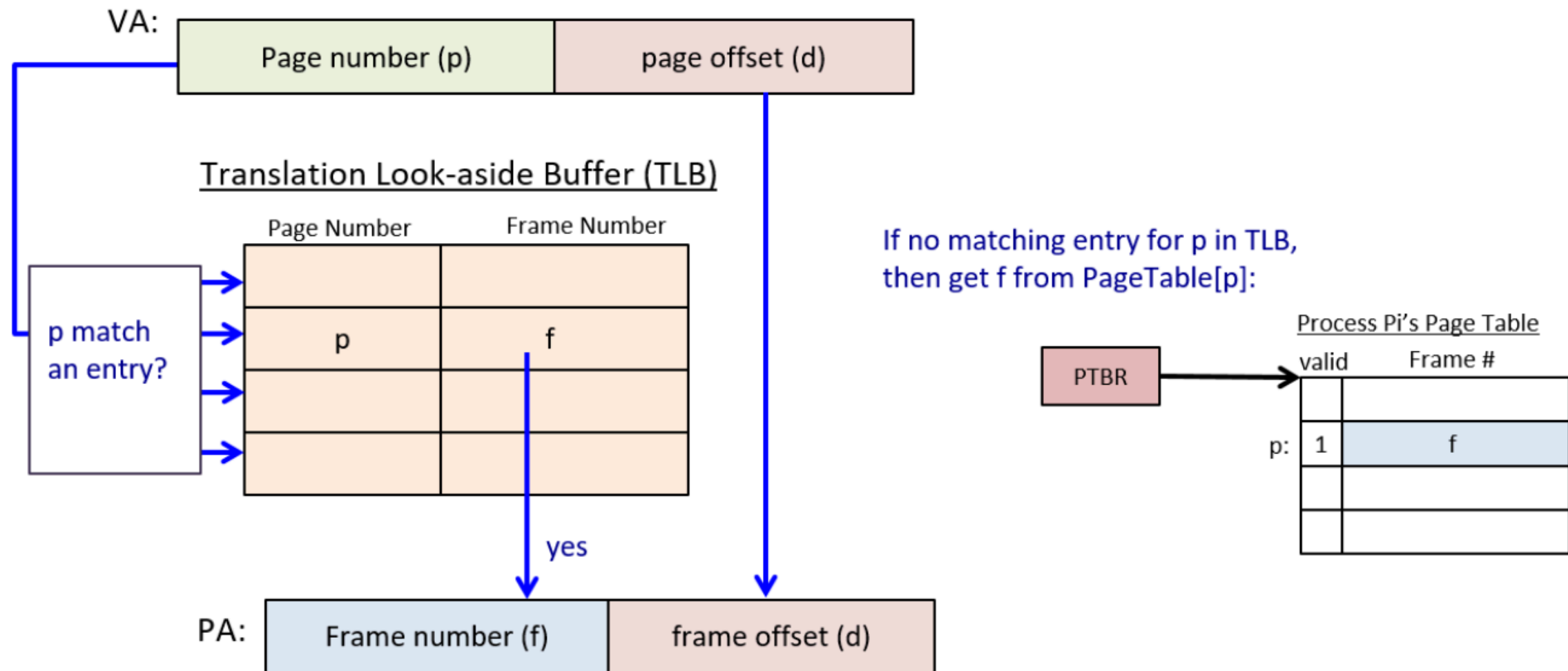
Page Fault

- A process tries to access a page that is currently not stored in RAM
- OS finds a free frame of RAM and loads the page into it
- A **page replacement policy**
 - used when free RAM is exhausted in the system
 - for example, an OS might implement the **least recently used** (LRU) policy
 - which replaces the page stored in the frame of RAM that has been accessed least recently

Making Page Accesses Faster

- In a paged virtual memory system
- every load and store to a virtual memory address requires two RAM accesses
 - Read the page table entry to get the frame number for virtual-to-physical address translation
 - Read or write the byte(s) at the physical RAM address
- twice as slow as in a system that supports direct physical RAM addressing
- **Translation Look-aside Buffer (TLB)** is a hardware cache that stores (page number, frame number) mappings

Translation Look-aside Buffer



13.4. Interprocess Communication

Three Methods

- **Signals**
 - very restricted form of interprocess communication
 - one process can send a signal to another process to notify it of some event
- **Message passing**
 - a process exchanges messages with another process
- **Shared memory**
 - a process shares all or part of its virtual address space with other processes

13.4.1. Signals

- **Signals** are similar to **hardware interrupts** and **traps**
 - but are different from both.
- **Trap** is a **synchronous** software interrupt
 - occurs when a process explicitly invokes a system call
- **Signals** are **asynchronous**
 - a process may be interrupted by the receipt of a signal at any point in its execution
- **Signals** also differ from asynchronous hardware interrupts in that they are triggered by software rather than hardware devices

Signals

Signal Name	Description
SIGSEGV	Segmentation fault (e.g., dereferencing a null pointer)
SIGINT	Interrupt process (e.g., Ctrl-C in terminal window to kill process)
SIGCHLD	Child process has exited (e.g., a child is now a zombie after running <code>exit</code>)
SIGALRM	Notify a process a timer goes off (e.g., <code>alarm(2)</code> every 2 secs)
SIGKILL	Terminate a process (e.g., <code>pkill -9 a.out</code>)
SIGBUS	Bus error occurred (e.g., a misaligned memory address to access an <code>int</code> value)
SIGSTOP	Suspend a process, move to Blocked state (e.g., Ctrl-Z)
SIGCONT	Continue a blocked process (move it to the Ready state; e.g., <code>bg</code> or <code>fg</code>)

Default Actions

- When a process receives a signal, one of several default actions can occur:
 - the process can terminate
 - the signal can be ignored
 - the process can be blocked
 - the process can be unblocked
- Application programmers, however, can change the default action of most signals and can write their own signal handler code

Signal Handlers

```
/* signal handler for SIGALRM */
void sigalarm_handler(int sig) {
    printf("BEEP, signal number %d\n.", sig);
    fflush(stdout);
    alarm(5); /* sends another SIGALRM in 5 seconds */
}

/* signal handler for SIGCONT */
void sigcont_handler(int sig) {
    printf("in sigcont handler function, signal number %d\n.", sig);
    fflush(stdout);
}

/* signal handler for SIGINT */
void sigint_handler(int sig) {
    printf("in sigint handler function, signal number %d\n.", sig);
    fflush(stdout);
    exit(0);
}
```

```
/* main: register signal handlers and repeatedly block
int main(void) {

    /* Register signal handlers. */
    if (signal(SIGCONT, sigcont_handler) == SIG_ERR) {
        printf("Error call to signal, SIGCONT\n");
        exit(1);
    }

    if (signal(SIGINT, sigint_handler) == SIG_ERR) {
        printf("Error call to signal, SIGINT\n");
        exit(1);
    }

    if (signal(SIGALRM, sigalarm_handler) == SIG_ERR) {
        printf("Error call to signal, SIGALRM\n");
        exit(1);
    }
}
```



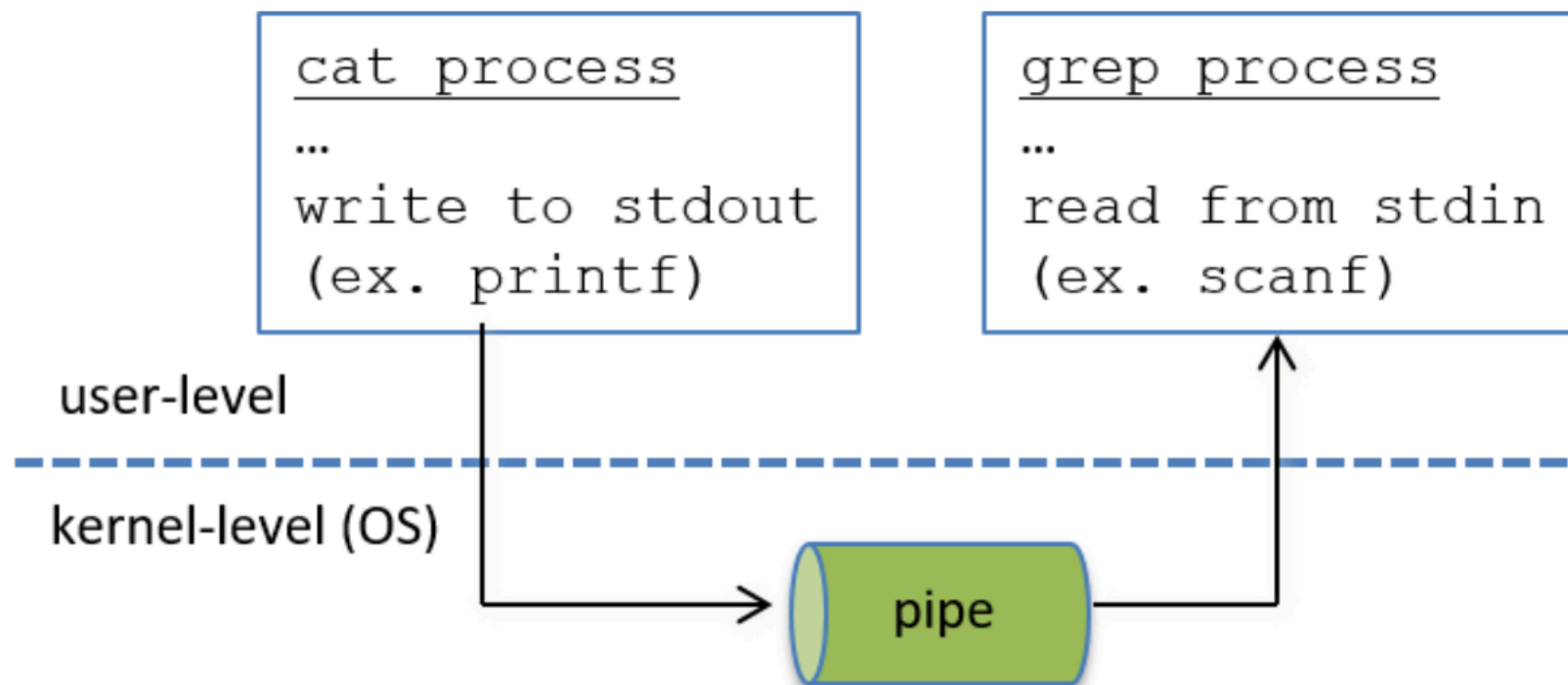
```
printf("kill -CONT %d to send SIGCONT\n", getpid())

alarm(5);  /* sends a SIGALRM in 5 seconds */

while(1) {
    pause(); /* wait for a signal to happen */
}
}
```

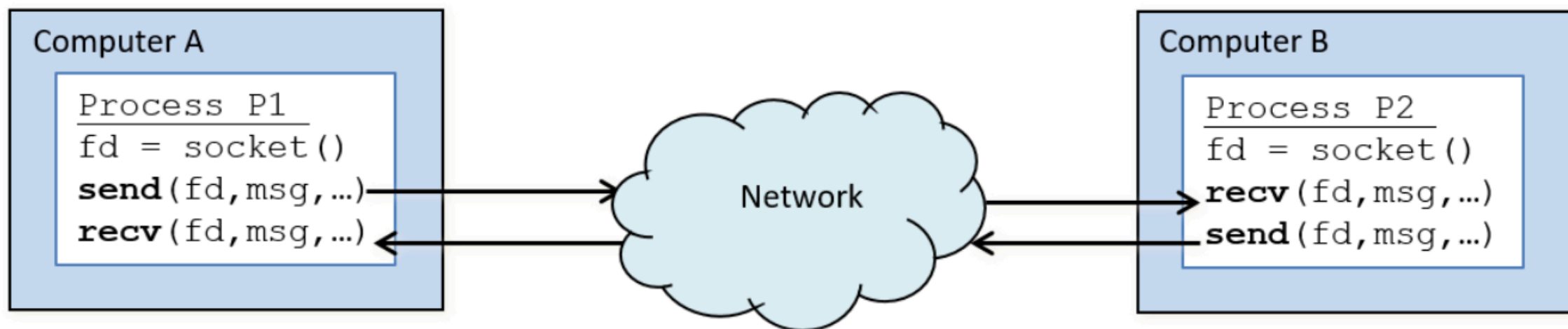
13.4.2. Message Passing

- **pipe** is a one-way communication channel for two processes running on the same machine
 - Example: **cat foo.c | grep factorial**



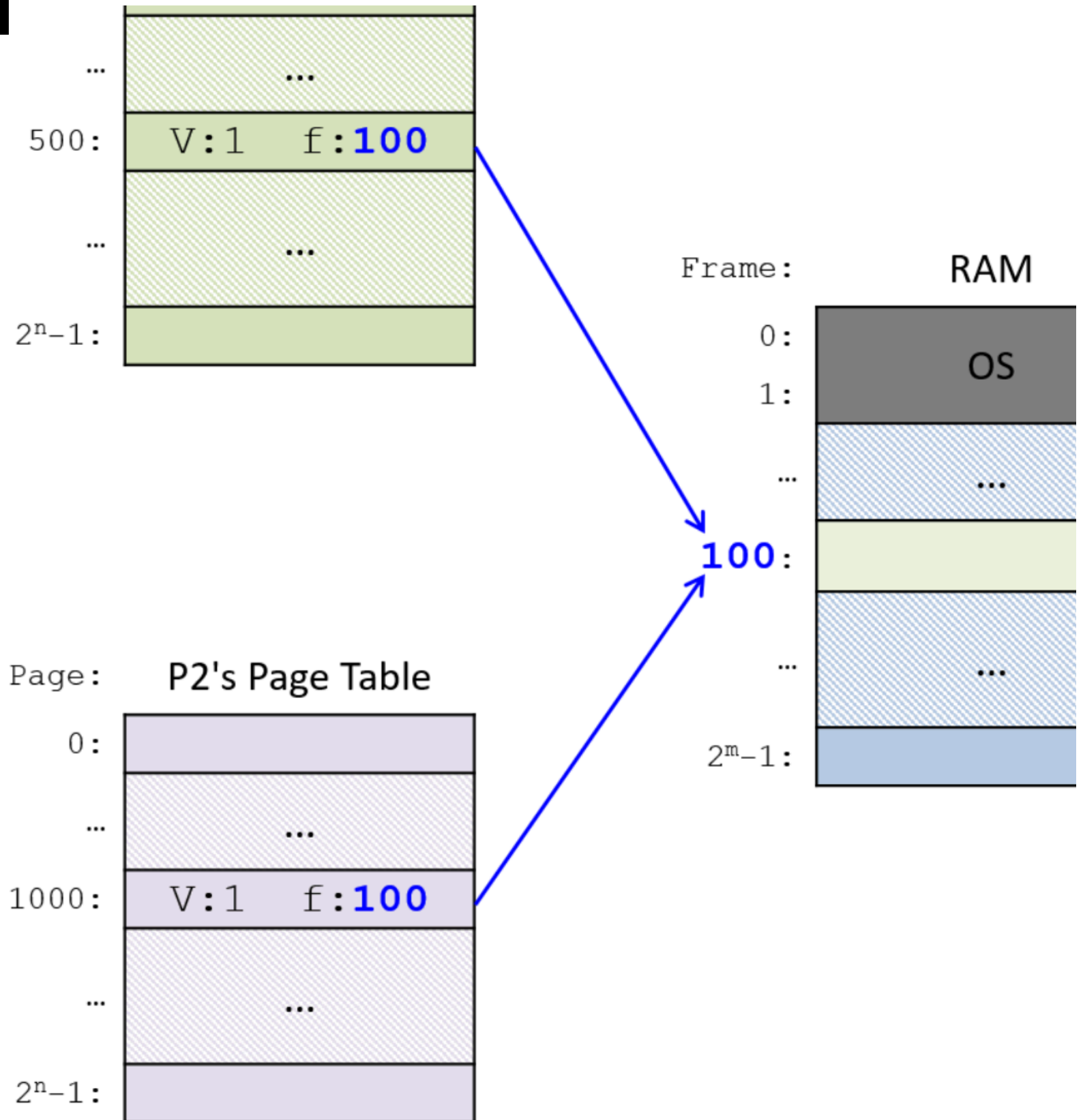
Sockets

- A **socket** is a two-way communication channel
- message passing is the only way in which processes on different computers can communicate



13.4.3. Shared Memory

- Entries in the page tables of two or more processes to map to the same physical frames



Kahoot!

Ch13b