CMPE 220

Class 10 – Operating Systems - part 1

What is an Operating System?

Wikipedia: An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.

TechTerms: An operating system, or "OS," is software that communicates with the hardware and allows other programs to run. It is comprised of system software, or the fundamental files your computer needs to boot up and function.

HowToGeek: An operating system is the primary software that manages all the hardware and other software on a computer. The operating system, also known as an “OS,” interfaces with the computer’s hardware and provides services that applications can use.

What is an Operating System?

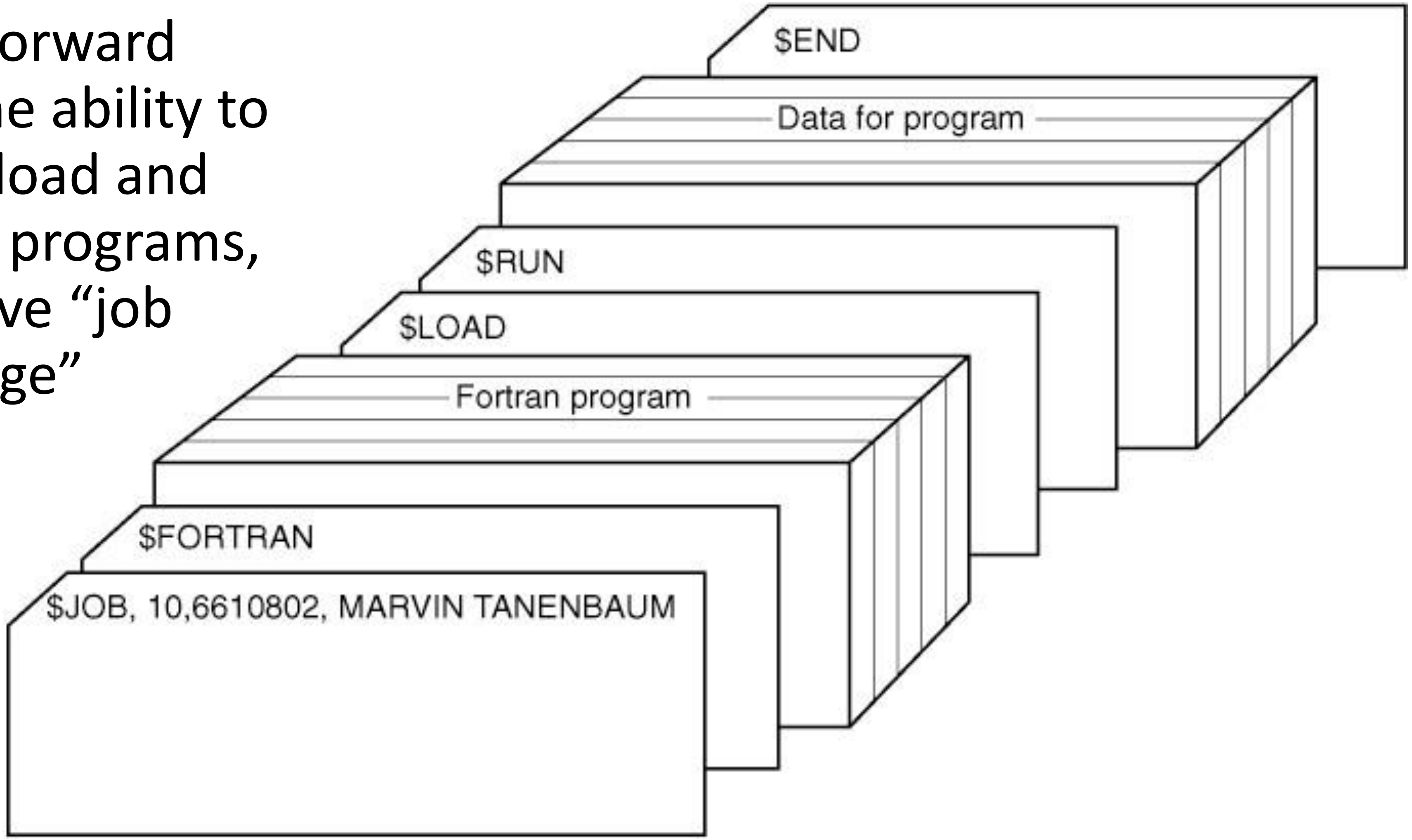
<table>
<tr><td><u>Applications</u>
Business programs, scientific programs, utility programs</td><td><u>System Software</u>
Compilers, assemblers, linkers, loaders, debuggers, databases</td></tr>
<tr><td colspan="2"><u>Operating System</u></td></tr>
<tr><td colspan="2"><u>Computer Hardware</u>
Instruction Set
I/O Architecture
Memory</td></tr>
</table>

History

- The first computers (1940s) did not have an operating system.
- Computers had a primitive *absolute loader* – often stored in some form of non-volatile memory
- Programs were loaded – typically from punched cards – and had complete control over the hardware
  - Programmers were responsible for memory management and I/O
- When the program was done (or crashed), the next user would take over the computer and load their program
- Microcomputers in the 1970s followed the same path, except programs were usually loaded from punched paper tape

History: Job Control Languages (JCL)

- The first step forward (1950s) was the ability to automatically load and run a series of programs, using a primitive “job control language” (JCL)



History: Job Control Programs

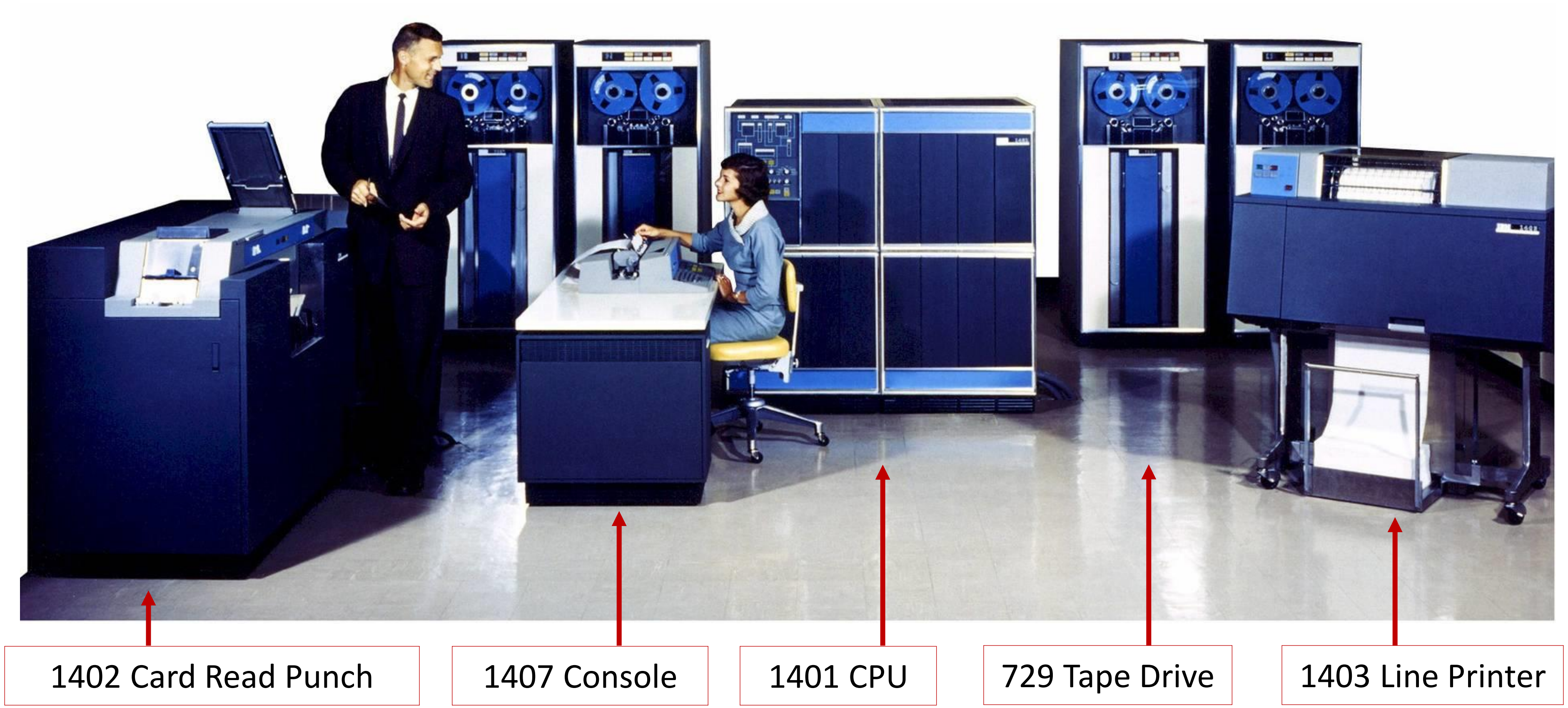
- This required a small “job control program” to remain **resident** in memory.
- This was a tradeoff. It used (precious) memory, but it made more efficient use of the computer
  - *Batch* processing
  - Moved on quickly when programs didn’t work
  - Did not require the programmer to be present
- This began a slippery slope
  - Saving work from programmers by adding functions to the resident job control program. Programming became simpler, but the control program got bigger.

A Typical Job

- A Fortran Compiler
  - 3 ½ boxes of punched cards
  - Each box = 2000 cards (about ten pounds each)
  - A 35 pound program!

Late 1950s: Recognizable Operating Systems

- IBM 1401 (1959)

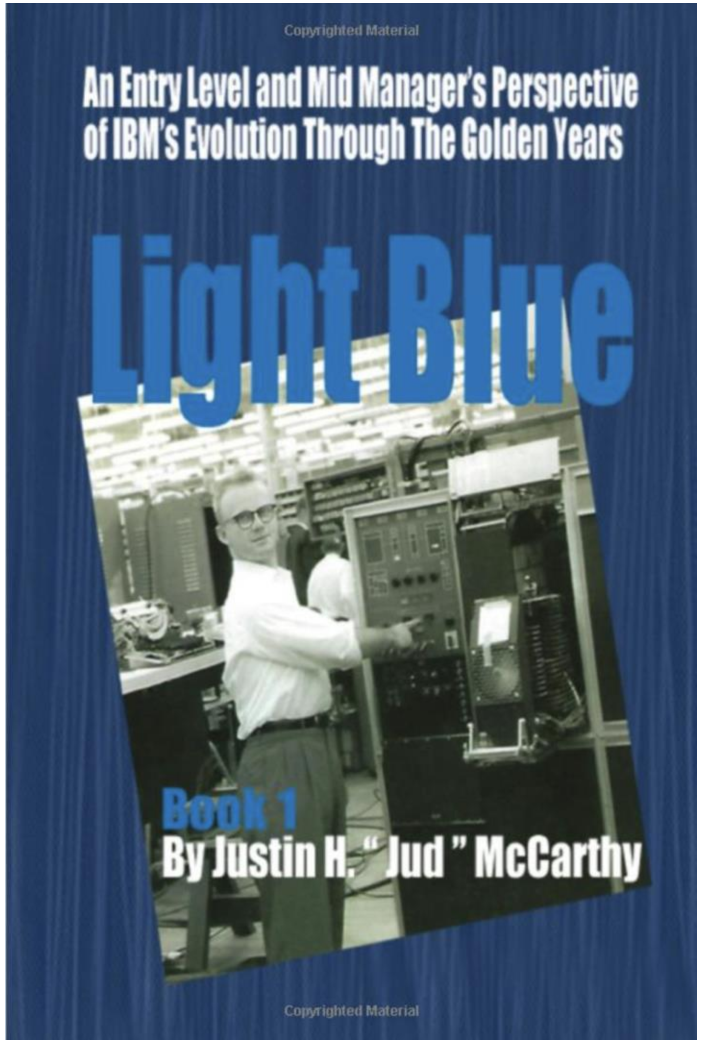


The IBM 1401: the Model T of Computing

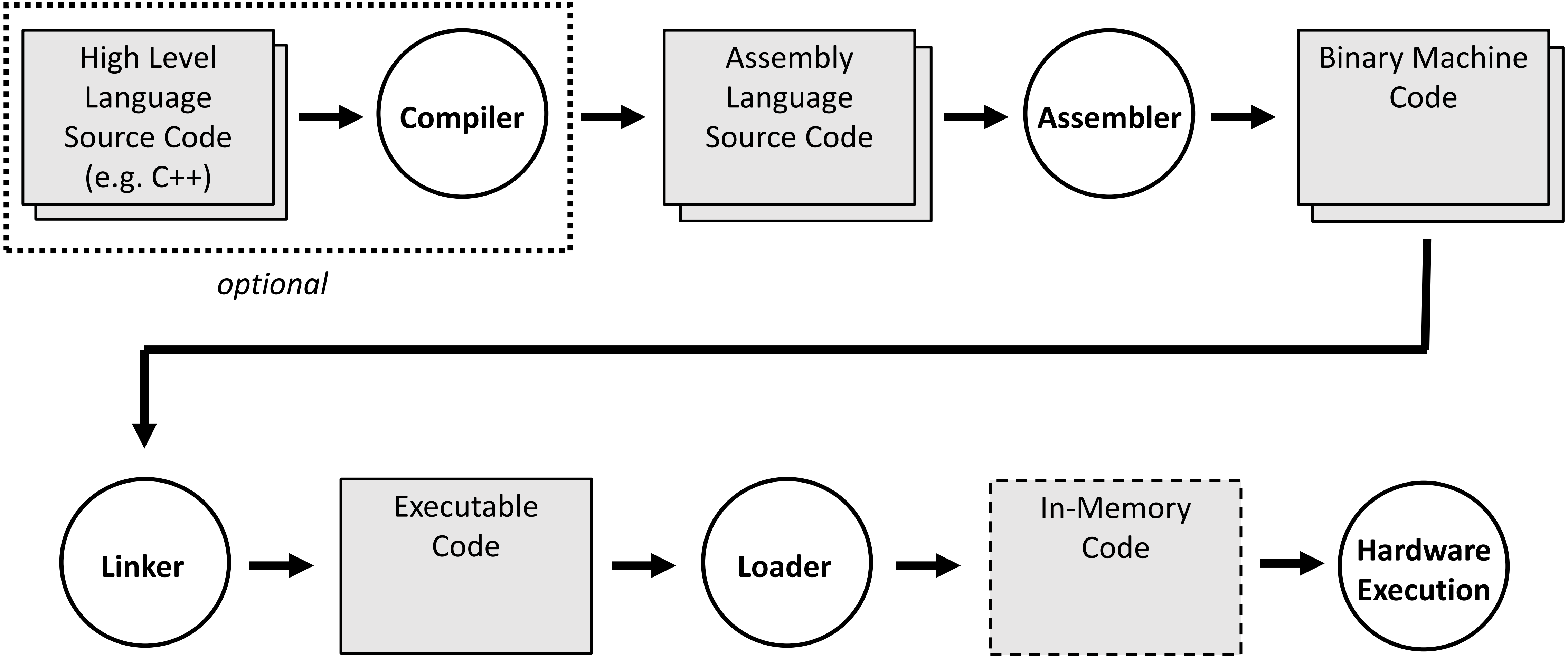
- 1959-1971
- Inexpensive
- Decimal (BCD) Arithmetic
- High Sales Volume: over 12,000 sold
  - By the mid-1960s, almost half the computers in the world were 1401s
- There is a fully restored and working IBM 1401 at the Computer History Museum in Mountain View

Interested in Early IBM History?

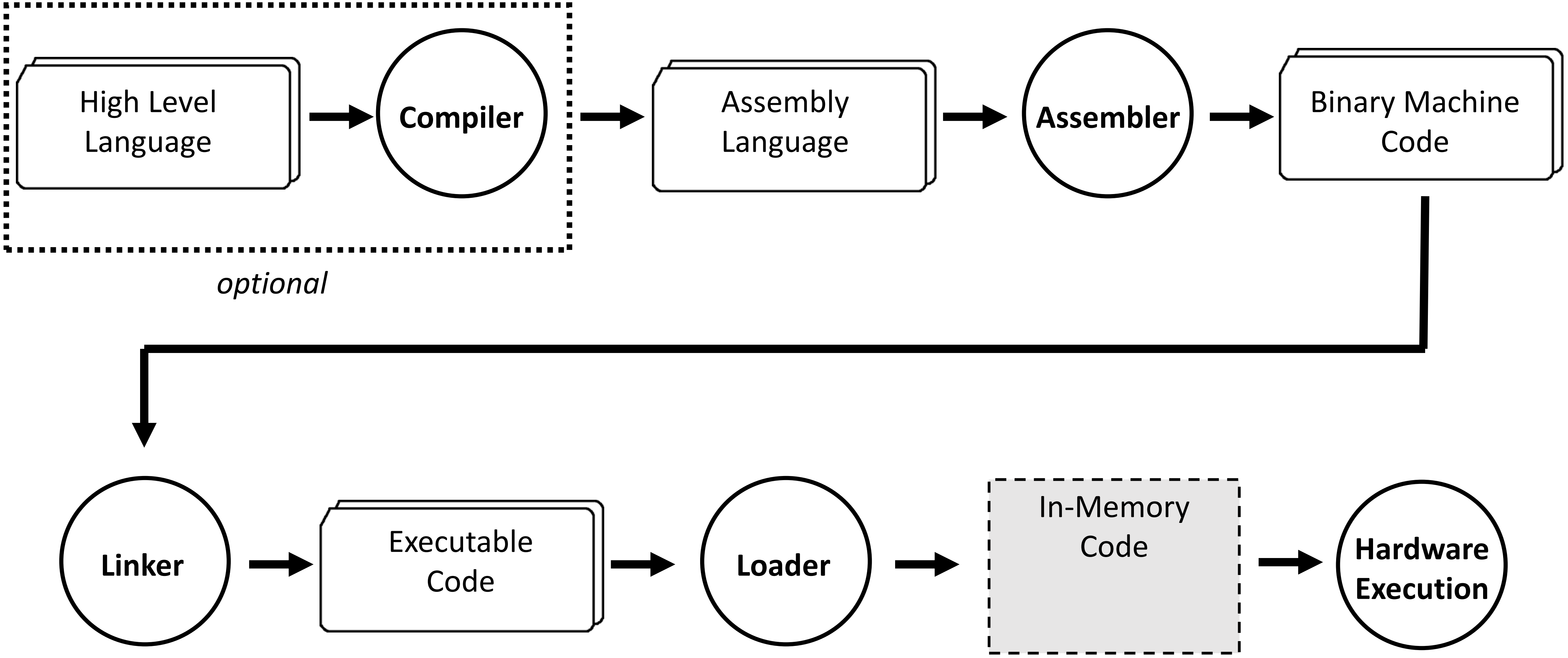
- Light Blue: An Entry Level and Mid Management Perspective of IBM's Evolution Through the Golden Years
  - Justin “Jud” McCarty
  - June, 2020



Building Software



Building Software: 1950s



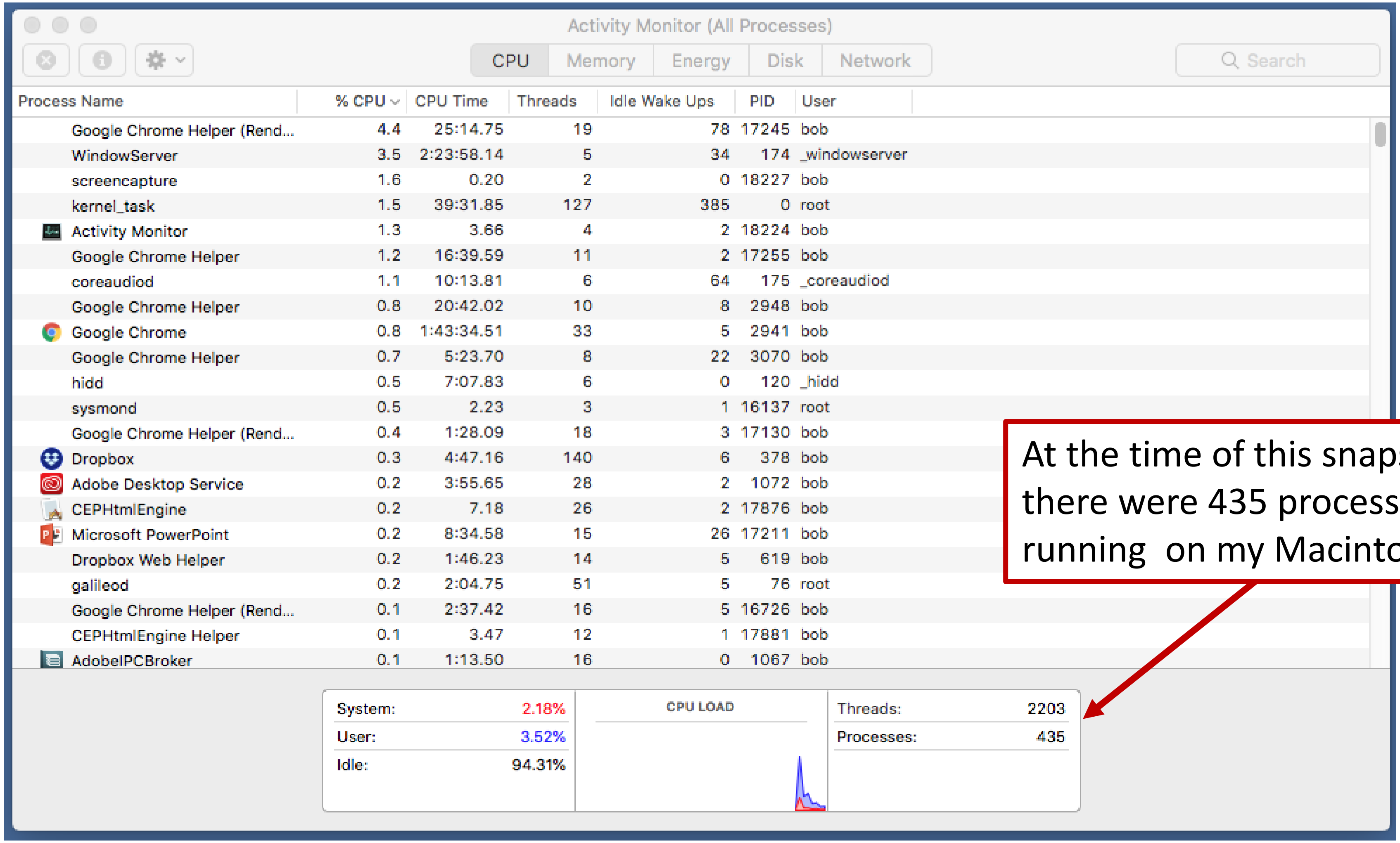
What Does a Modern Operating System Do?

1. Process Management
    - Interprocess Communications
2. Input / Output (I/O) Management
3. Memory Management
4. File System Management
5. System Functions and Kernel Mode
6. User Interaction – (maybe)

(1) Process Management

- A modern computer runs many programs at the same time
- Each instance of a running program is called a *process*
  - A program is just code
  - A process is code, data, and state information *running on a computer*
- Each process has its own address space… in effect, it behaves as if it is the only program running
- *It's important to understand that the operating system itself runs as one or more of these processes*

Processes – Mac OS: Activity Monitor


Activity Monitor (All Processes)

CPU | Memory | Energy | Disk | Network

Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
Google Chrome Helper (Rend...	4.4	25:14.75	19	78	17245	bob
WindowServer	3.5	2:23:58.14	5	34	174	_windowserver
screencapture	1.6	0.20	2	0	18227	bob
kernel_task	1.5	39:31.85	127	385	0	root
Activity Monitor	1.3	3.66	4	2	18224	bob
Google Chrome Helper	1.2	16:39.59	11	2	17255	bob
coreaudiod	1.1	10:13.81	6	64	175	_coreaudiod
Google Chrome Helper	0.8	20:42.02	10	8	2948	bob
Google Chrome	0.8	1:43:34.51	33	5	2941	bob
Google Chrome Helper	0.7	5:23.70	8	22	3070	bob
hidd	0.5	7:07.83	6	0	120	_hidd
sysmond	0.5	2.23	3	1	16137	root
Google Chrome Helper (Rend...	0.4	1:28.09	18	3	17130	bob
Dropbox	0.3	4:47.16	140	6	378	bob
Adobe Desktop Service	0.2	3:55.65	28	2	1072	bob
CEPHtmlEngine	0.2	7.18	26	2	17876	bob
Microsoft PowerPoint	0.2	8:34.58	15	26	17211	bob
Dropbox Web Helper	0.2	1:46.23	14	5	619	bob
galileod	0.2	2:04.75	51	5	76	root
Google Chrome Helper (Rend...	0.1	2:37.42	16	5	16726	bob
CEPHtmlEngine Helper	0.1	3.47	12	1	17881	bob
AdobeIPCBroker	0.1	1:13.50	16	0	1067	bob

System:	2.18%
User:	3.52%
Idle:	94.31%

CPU LOAD

Threads:	2203
Processes:	435

At the time of this snapshot, there were 435 processes running on my Macintosh

Processes – POSIX: ps –axl command

```
bob$ ps -axl
  UID   PID  PPID        F CPU PRI NI       SZ    RSS WCHAN     S             ADDR TTY           TIME CMD
    0     1     0     4004   0  37  0  4330580  14228 -      Ss                  0 ??         5:13.39 /sbin/launch
    0    59     1     4004   0   4  0  4306056    916 -      Ss                  0 ??         0:11.62 /usr/sbin/sy
    0    60     1     4004   0  37  0  4333164  10864 -      Ss                  0 ??         0:14.06 /usr/libexec
    0    64     1     4004   0  20  0  4296532   3304 -      Ss                  0 ??         0:02.98 /System/Libr
    0    65     1     4004   0  37  0  4339484   1904 -      Ss                  0 ??         0:03.19 /usr/libexec
    0    66     1  1004004   0  50  0  4356220   4368 -      Ss                  0 ??         1:06.17 /System/Libr
    0    70     1     4004   0   4  0  4311292  10024 -      Ss                  0 ??         0:00.59 /Library/Fra
    0    71     1     4004   0   4  0  4358892   6264 -      Ss                  0 ??         0:10.08 /usr/sbin/sy
   55    73     1     4004   0   4  0  4333692   4924 -      Ss                  0 ??         0:02.59 /System/Libr
    0    74     1     400c   0  37  0  4336288   5476 -      Ss                  0 ??         0:11.68 /usr/libexec
    0    75     1     4004   0  37  0  4331460   4864 -      Ss                  0 ??         0:38.00 /System/Libr
    0    76     1     4004   0  20  0  8929896  15168 -      Ss                  0 ??         2:04.89 /Library/App
    0    79     1     4004   0  37  0  4413840  18912 -      Ss                  0 ??         0:45.90 /usr/libexec
    0    83     1     4004   0  37 10  4336048   6708 -      SNs                 0 ??         0:02.35 /usr/libexec
    0    84     1  1004004   0  50  0  4504488  37392 -      Ss                  0 ??        48:37.12 /System/Libr
    0    87     1     4004   0   4  0  4304388    164 -      Ss                  0 ??         0:00.02 firmwaresync
    0    91     1     4004   0  37  0  4330644   2864 -      Ss                  0 ??         0:03.30 /usr/libexec
    0    93     1     4004   0   4  0  4348432  23776 -      Ss                  0 ??         0:34.61 /usr/libexec
    0    94     1     4004   0   4  0  4305144  14416 -      Ss                  0 ??         0:00.42 /System/Libr
    0    98     1     4004   0  37  0  4345252   8284 -      Ss                  0 ??         0:43.12 /usr/libexec
    0   101     1     4004   0   4  0  4331928   2632 -      Ss                  0 ??         0:00.25 /System/Libr
    0   102     1     4004   0   4  0  4338264  10756 -      Ss                  0 ??         0:22.02 /System/Libr
    0   104     1     4004   0  37  0  4338644   5944 -      Ss                  0 ??         0:38.26 /usr/sbin/se
  205   105     1     4004   0   4  0  4337968   6628 -      Ss                  0 ??         0:08.41 /usr/libexec
  244   108     1     4104   0  20  0  4305248   3104 -      Ss                  0 ??         0:00.07 /usr/libexec
    0   109     1     4004   0  20  0  4304884    188 -      Ss                  0 ??         0:00.02 autofsd
    0   111     1     4004   0   4  0  4336212   7104 -      Ss                  0 ??         0:05.38 /usr/libexec
   74   113     1     4044   0  31  0  5372540   1544 -      Ss                  0 ??         0:21.59 /usr/local/m
    0   114     1  1004004   0   4  0  4370036   3332 -      Ss                  0 ??         0:00.38 /System/Libr
  504   115     1 80004104   0  63  0  4466888  39956 -      Ss                  0 ??         0:25.84 /System/Libr
    0   116     1     4004   0  37  0  4304496   1936 -      Ss                  0 ??         0:00.13 /System/Libr
    0   117     1     4004   0  37  0  4296464    240 -      Ss                  0 ??         0:00.01 /usr/sbin/Ke
```

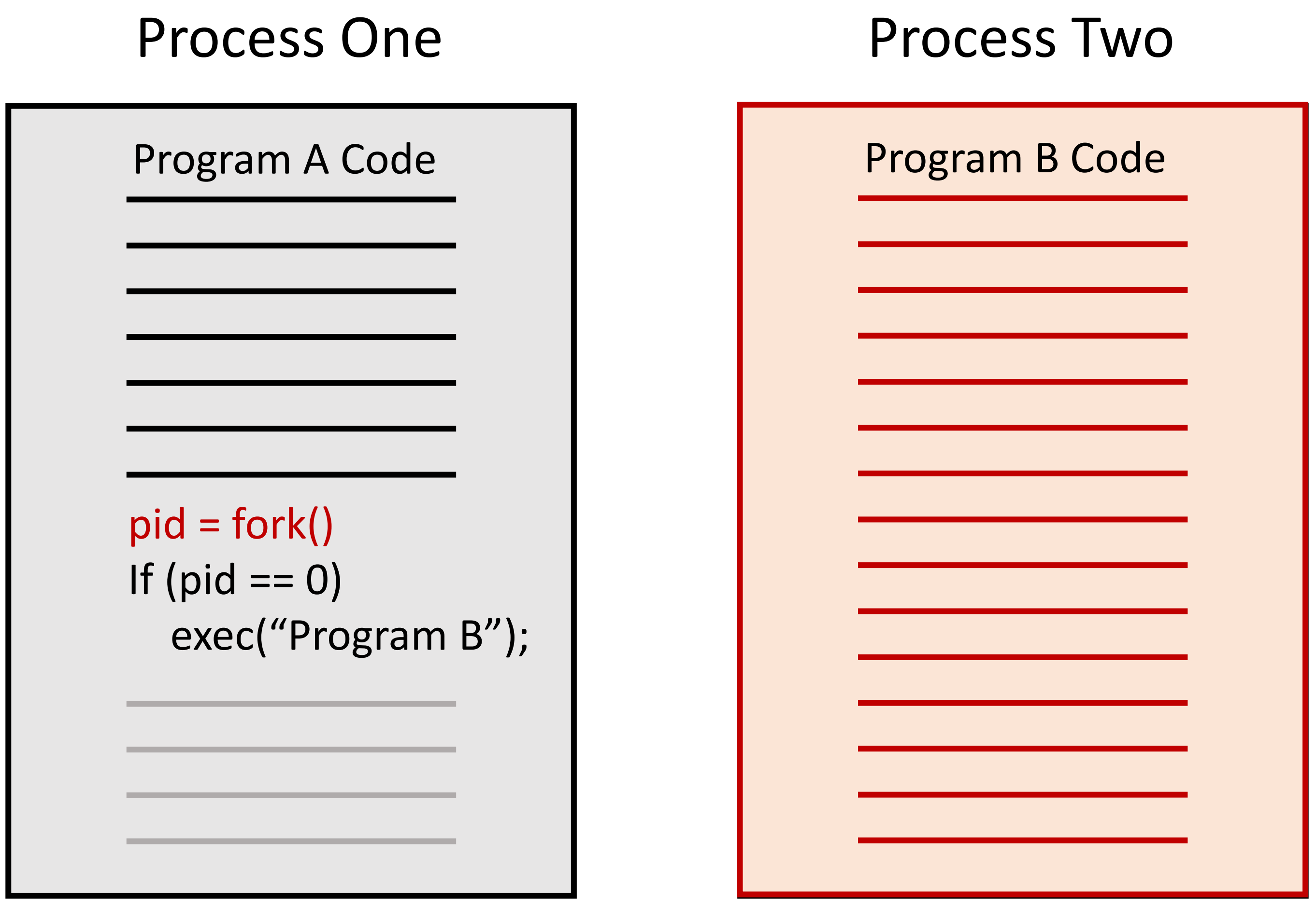
How Processes Are Created

- A system function call is required to create a new process
- A system function call is required to invoke the loader, load a program, and start execution
- There may be two separate calls, or a single call to do both
- In POSIX systems:
  - The fork() system call duplicates the currently running process – leaving TWO processes running the identical code, with identical states
  - The exec() system call loads an executable into the current process
  - For one program to “launch” another, it first calls fork(), and then the child process calls exec()

Fork() and Exec()

```
If ((pid = fork()) < 0) { // After this call there should be 2 processes running
        printf("Fork failed\n");
        exit(-1);
}
If (pid == 0) { // This is the child process
        exec( "/users/robert/fibonnacci" );  // Invoke the loader
        printf("Exec failed\n");
        exit(-1);
}
// Parent process continues
printf("Process ID of child is %d\n", pid);
```

Fork() and Exec()



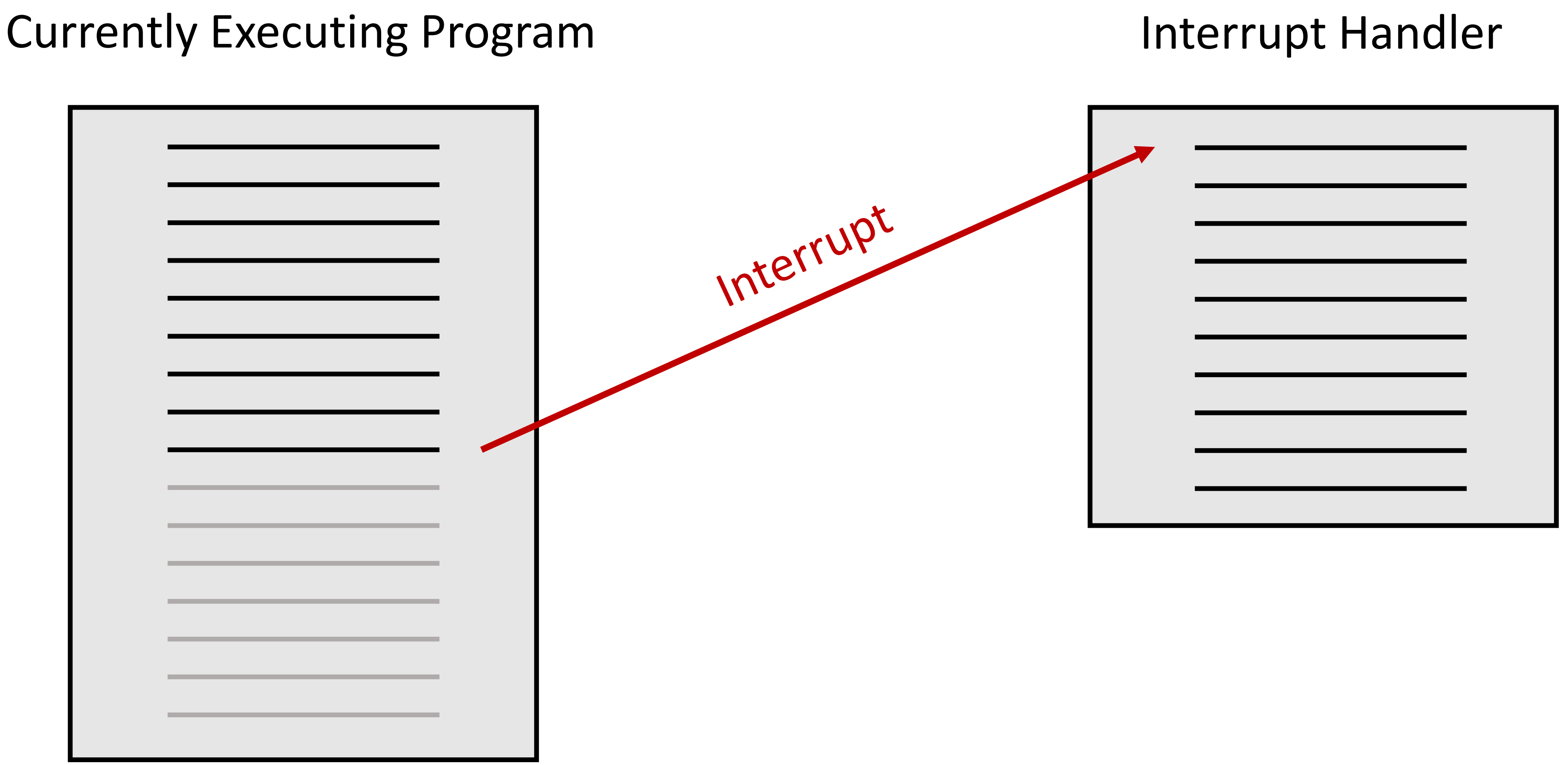
How Processes are Terminated

- The program may call a system function, such as the POSIX *exit()* function, that does some cleanup and deletes the process
- When a program terminates without calling exit(), the system automatically executes some code that does the same thing
  - Alluded to in earlier lectures
- With appropriate permissions, one process may force the deletion of another process by calling a system function such as the POSIX *kill()* function
- The system may also have security features that will automatically kill processes that use too much CPU, memory, etc.

Interrupts

- A multiprocessing operating system relies on *interrupts*
- An interrupt is a signal sent to the processor that interrupts the current process
- It may be generated by a hardware device or a software program
- Types of interrupts
    - Timer
    - I/O state change (such as operation complete)
    - A signal from one process to another

What an Interrupt Does



Saving the Process State

- In order to allow us to resume execution of the current process, the interrupt handler must save the state (such as the registers) of the currently executing program in a data structure called a **Process Control Block (PCB)**
- The is a PCB for each process

Contents of a Process Control Block (PCB)

Process management	Memory management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit maps for signals	
	Various flag bits	

Process Scheduling

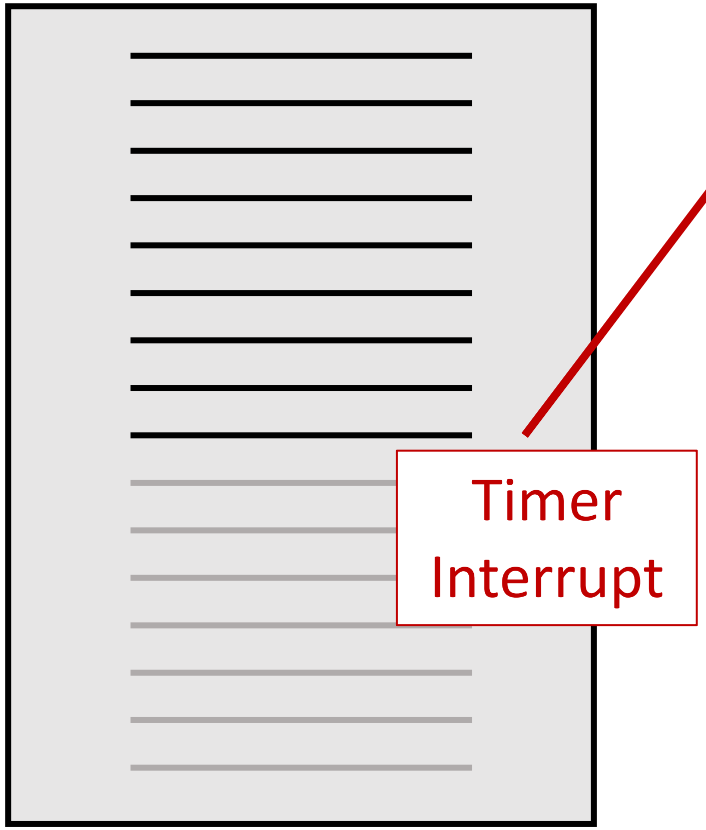
- The operating system makes sure that every program gets some time to run.
- **Scheduler:** a component of the operating system that determines which process runs next
- **Dispatcher:** The task of switching control to another process is called *dispatching*, and the code that accomplishes this is called the *dispatcher*
- **Time Slicing:** The operating system uses a *timer interrupt* to periodically regain control so that it can switch from one process to another

Scheduling Algorithms

- **Round Robin:** All processes get an equal time-slice, in order
- **Priority Scheduling:** Some processes may be scheduled as higher priority, and get more or longer time slices
  - The original Lunar Lander had a single computer
  - The process that adjusted the attitude control jets ran at a higher priority than the process that updated the display
- **Adaptive Scheduling:** adjust scheduling based on process performance
  - A system may give less time to “CPU hogs”
- Many other algorithms are possible

Switching Processes

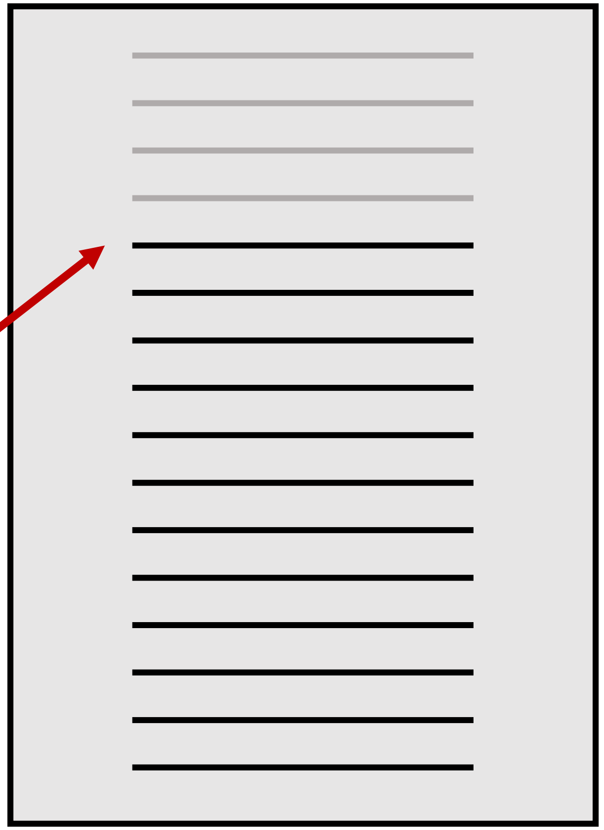
Currently Executing Process



Interrupt Handler

- Save state of current program in its PCB
- **Scheduler** determines next process to execute
- **Dispatcher** Restores state of next program from its PCB, and…
- Transfers control to next program

Next Process



Process Switching and I/O

- Typically, when a program starts an I/O operation, it can't proceed until the operation is complete
- In terms of processor speed, I/O operations take a really, really long time
- The system **scheduler** will not give control to a process that is waiting for I/O.
- We say that the process is *blocked* – or in an *I/O wait state*

What Happens When a Program Initiates I/O

- The program calls an operating system routine to start the I/O operation
- The operating system routine will:
  - Save the process state in its PCB
  - Set a flag in the process's PCB to indicate that the program is in an *I/O wait state*
  - Initiate the I/O operation
  - Call the **scheduler** to determine which process to execute next
  - Call the **dispatcher** to give control to that process

When an I/O Operation Completes

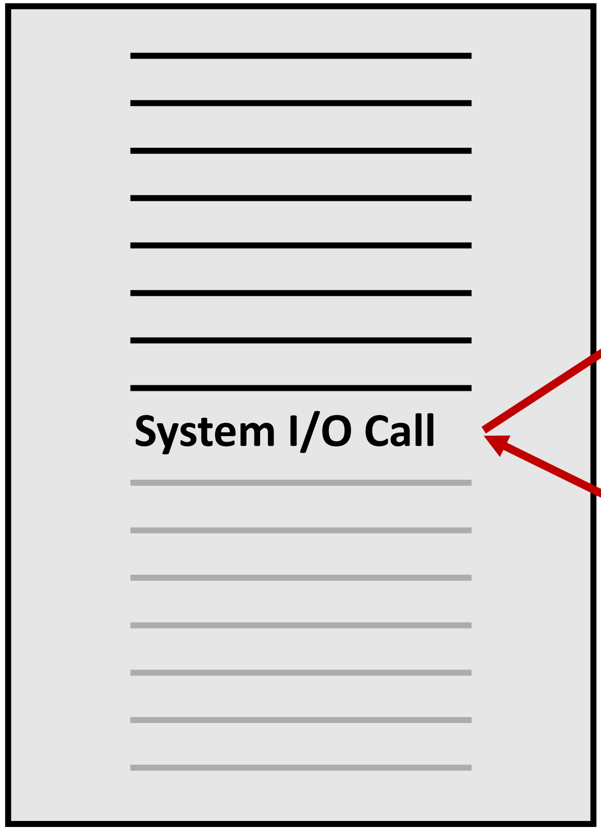
- A completed I/O operation generates an I/O Interrupt
- The interrupt handler gains control, and:
  - Saves the state of the currently executing process, whatever it may be
  - Determines which process initiated the I/O operation that just completed
  - Sets the status in that process's PCB to indicate it is *ready*
  - Calls the **scheduler** to determine which process to execute next
  - Calls the **dispatcher** to give control to that process
- Note that the "next process" *may or may not* be the process that initiated the I/O operation
- I/O completion simply makes that process *ready,* or eligible for scheduling

Returning Control to the I/O Caller

- Eventually, the dispatcher will return control to the process that initiated the I/O operation
- At that time, the system I/O function will return, passing back the results of the I/O operation

I/O Processing

Executing Program



System I/O Function

- Save the process state in its PCB
- Set *I/O wait state*
- Initiate the I/O operation

I/O Completion Interrupt

- Set status in original caller’s PCD to ready

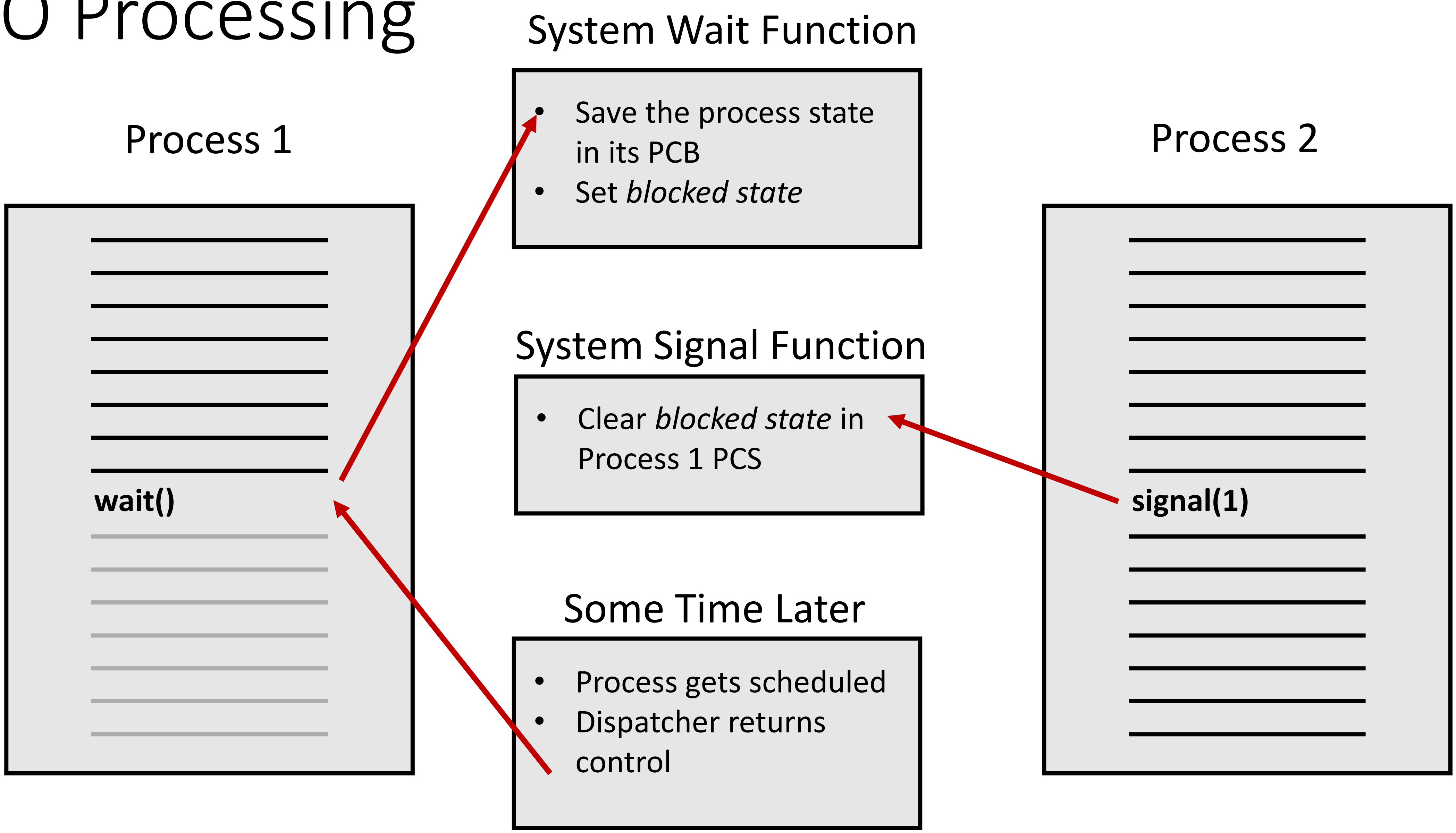
Some Time Later

- Process gets scheduled
- Dispatcher returns control, along with I/O results

Process Synchronization and Communication

- A process can pause, waiting for the completion of another process using the *wait()* function call
  - A POSIX shell may launch a child process to run a program from the command line, and wait for its completion
- **Inter-Process Communication (IPC):** A process may make a system call to send a signal to another process, or to set up a signal handler to receive signals
- Just as with I/O wait states, these waits are indicated by a special *blocked* status in the PCB, preventing the process from being scheduled until the condition is met

I/O Processing



(2) Input / Output (I/O) Management

- On early computers, I/O was performed by the processor:

```
WAITING     TD      DEVICE        ; wait until device is ready
            JEQ     WAITING
            RD      DEVICE        ; get a byte from device
            STA     BUFFER, X     ; store byte in buffer
```

- Of course, today we don’t tie up the processor waiting for each byte!

Adding a Primitive I/O Subsystem

- System functions to READ and WRITE data
  - READ(device, buffer, count);
  - WRITE(device, buffer, count);
  - The calling process is placed in an *I/O wait state*
  - Functions place the parameters in an I/O Control Block (IOCB)
- An *I/O Process*
  - The I/O process contains a loop: for each IOCB:
    - Check to see if the device is ready
    - If so, READ or WRITE the next byte
    - If the I/O operation is complete, set the status to *ready* in the calling process's PCB

A Modern I/O Subsystem

- On modern systems, I/O is handled by hardware, rather than relying on the processor for byte-by-byte transfers
- System functions to READ and WRITE data
  - READ(device, buffer, count);
  - WRITE(device, buffer, count);
  - The calling process is placed in an *I/O wait state*
  - Functions place the parameters in an I/O Control Block (IOCB)
  - Functions initiate the I/O transfer using dedicated I/O controllers

A Modern I/O Subsystem - Continued

- We do not need an *I/O process*
- The I/O controllers will generate an interrupt when the transfer is complete
- The interrupt handler will set the status to *ready* in the calling process's PCB

(3) Memory Management

<u>Physical Addressing</u>

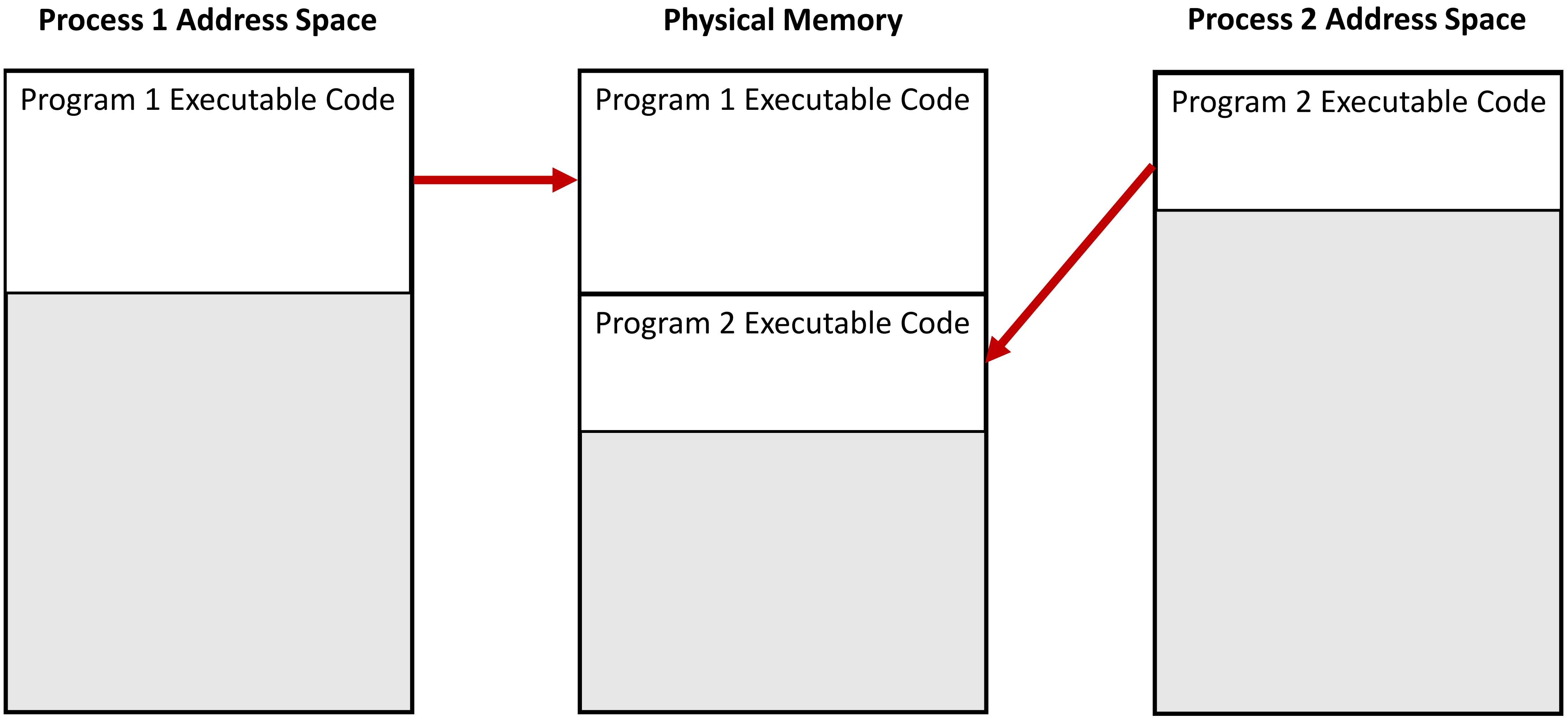
- Early computers did not have any form of address translation
- Addresses used in the program exactly correspond to physical memory addresses
- The size of physical memory was limited to the address range of the instruction set
- To load multiple programs or code blocks into memory, the system required a *relocating loader*
- Microcomputers in the 1970s and embedded computers today use physical addressing

Partitioned Memory

<u>Partitioned Addressing</u>

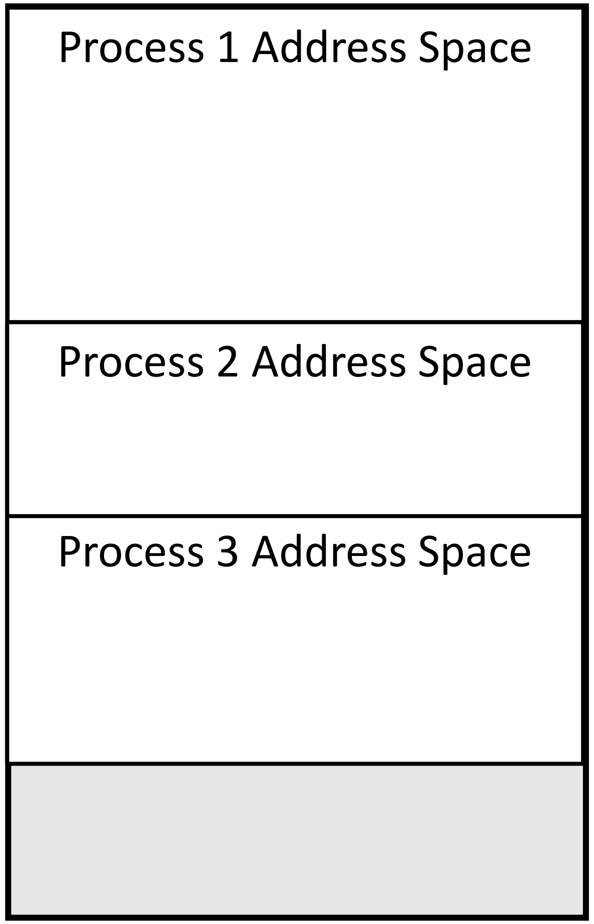
- Partitioned Addressing requires hardware support – a Memory Management Unit (MMU)
- A “base address” register that can be set when a process is dispatched, and possibly start and end addresses to provide memory protection
- Memory references are translated in hardware by adding the “base address register” to determine the address in physical memory
- The amount of memory available to any process may be limited by the instruction set, but physical memory may be much larger

Partitions

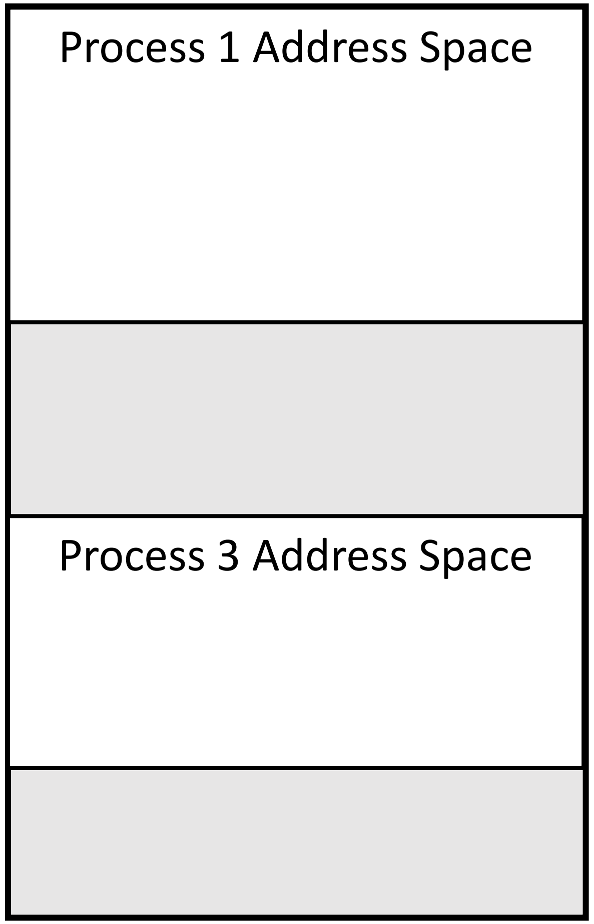

Process 1 Address Space
Physical Memory
Process 2 Address Space
Program 1 Executable Code
Program 1 Executable Code
Program 2 Executable Code
Program 2 Executable Code

Memory Fragmentation

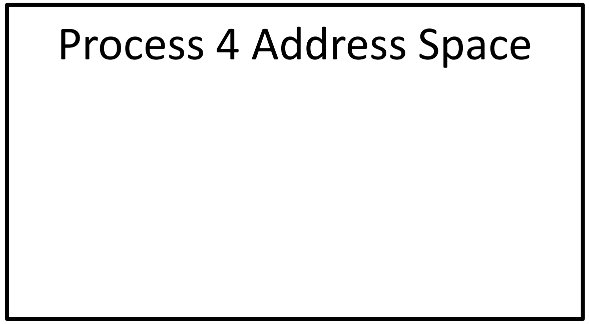
Starting State



Process 2 Ends

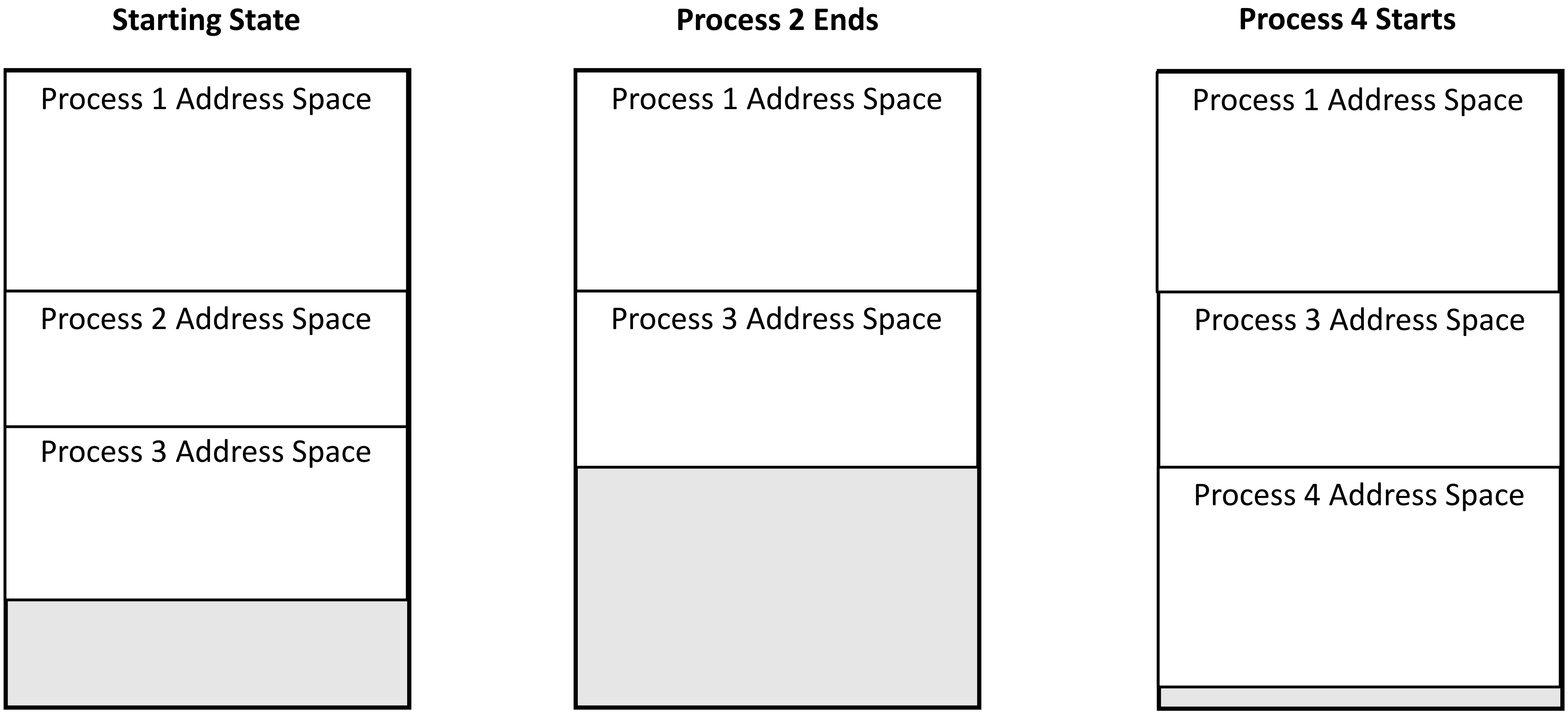


Process 4 Doesn’t Fit!



We have enough memory… but it’s not contiguous

Memory Fragmentation – Dynamic Relocation

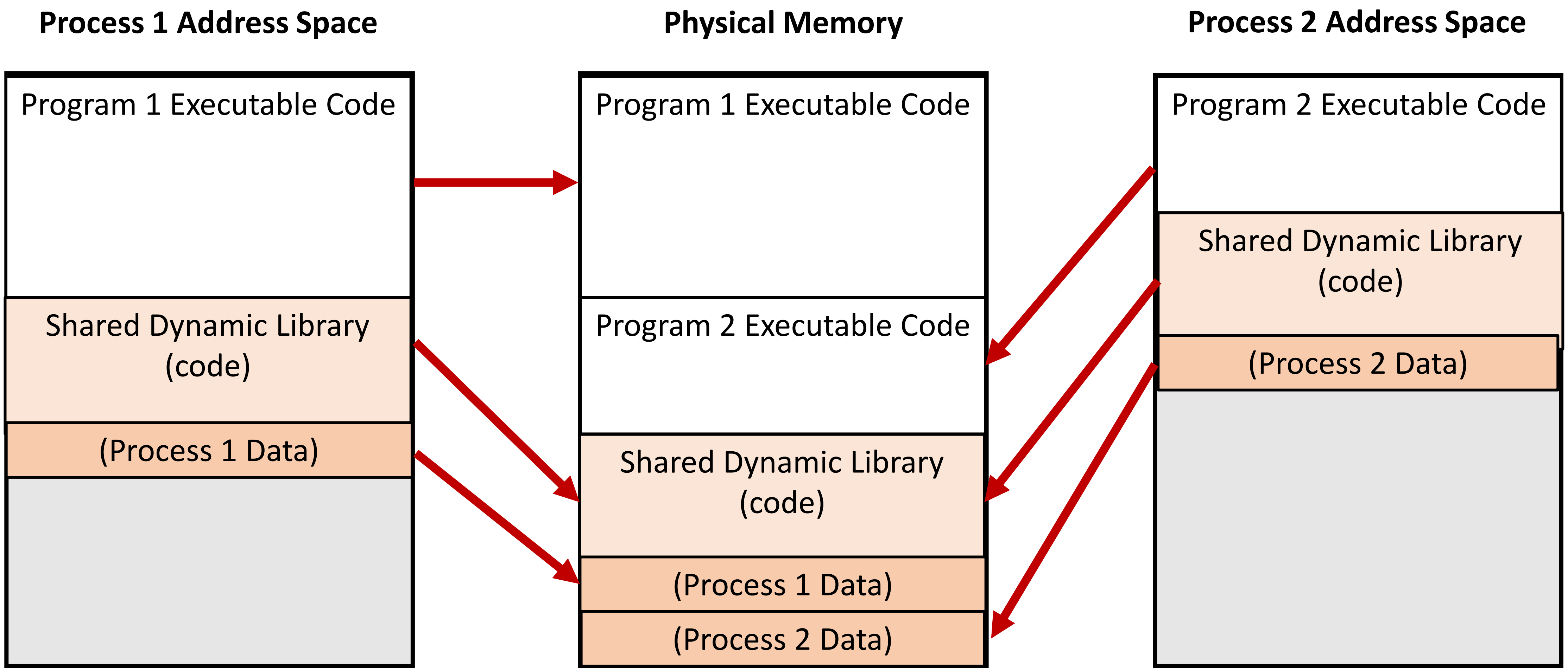


Memory Protection

- *Memory Protection* prevents one process from reading or writing memory used by another process
- In addition to a *base address*, the MMU may support a *start address* and *end address.*
- For each memory reference, the MMU adds the base address, and determines if the resulting physical address live within the bounds of the start and end
    - If not, an error interrupt is generated

Getting Fancy – Modern MMUs

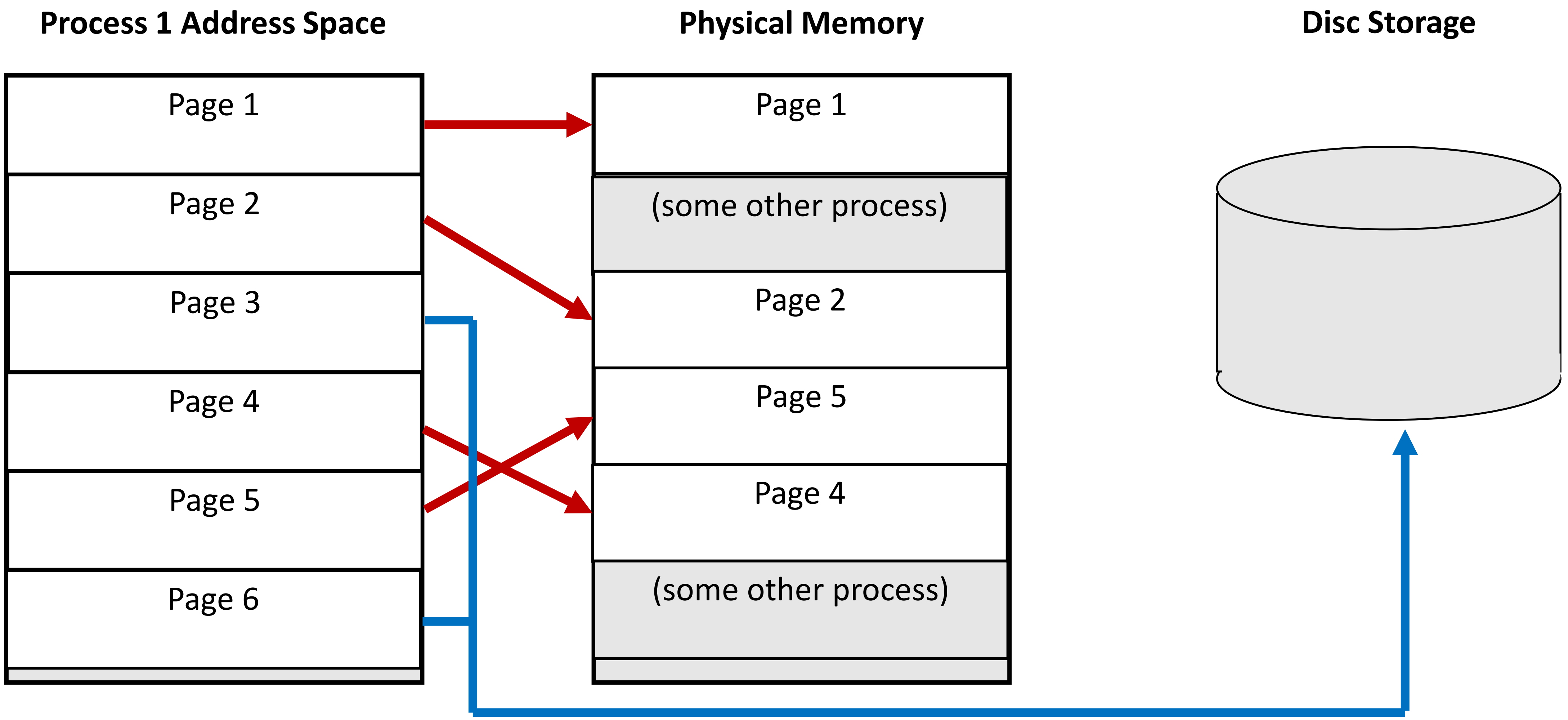
- Separate base addresses for code, data, and shared memory



Virtual Memory

- **Virtual Memory:** a system of software and hardware that allows portions of a program’s memory to be temporarily cached on disk, to minimize the requirements for physical memory.
- Allows the operating system to load and execute programs with memory requirements that total far more than the available physical memory.
- Virtually memory systems divide the memory address space into *pages*, each of which is separately mapped by the MMU.
- Not all pages reside in physical memory. Pages that are not loaded in physical memory are cached on disk.

Virtual Memory Mapping



Virtual Memory Mapping

- Requires a separate *base address* for each *page* in the process's address space.
- The MMU maps each memory reference to the corresponding location in physical memory by adding the base address.
- If the page does not reside in physical memory, the MMU will generate a *page fault* interrupt.
  - The process will be placed in an I/O wait state
  - The requested memory page will be loaded from disk into physical memory
  - This may require a page that is currently in physical memory to be moved to disk (swapped out) in order to free up space

Choosing the Page to Swap Out

- If a page is already on disk, and the copy in memory has not been modified, it doesn’t actually need to be written to disk – so it’s a good candidate to free up
- Other algorithms include:
  - FIFO – First In, First Out
  - LRU – Least Recently Used
  - LFU – Least Frequently Used

Why the Algorithm is Important

- Going to disk to “swap in” a page (and possibly “swap out” a page to free up memory) dramatically slows program execution
- If a RAM access took one SECOND, a disk access would take one DAY
- A system that has insufficient physical memory needs to swap frequently, resulting in *thrashing*, a condition in which the system is essentially unable to perform useful work because of constant page swaps.