Some OS Basics

Some notes adopted from Bryant and O'Hallaron

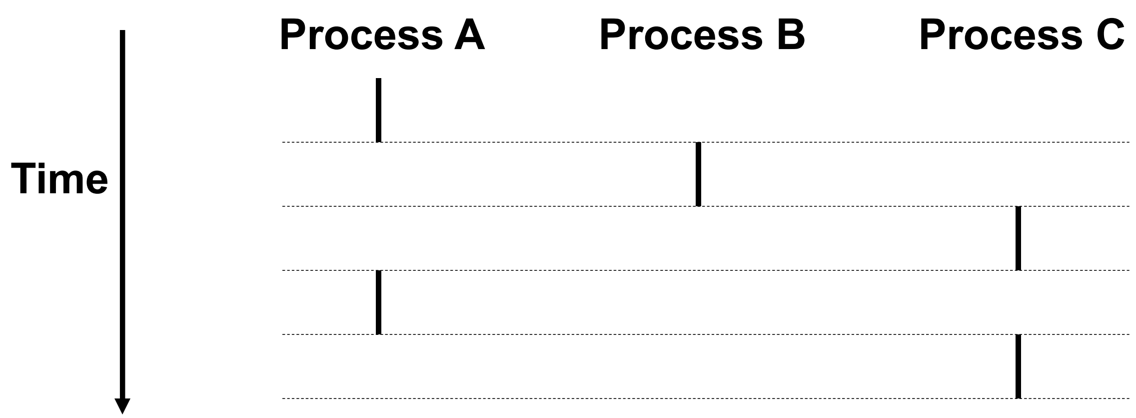
Role of The Operating System?

Processes

- Def: A process is an instance of a running program.
  - One of the most profound ideas in computer science.
  - Not the same as “program” or “processor”
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.
- How are these Illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

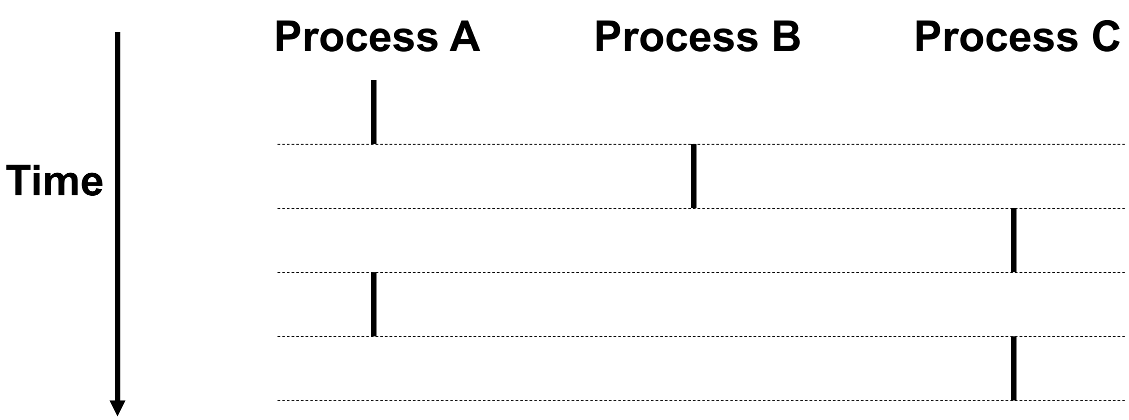
Logical Control Flows

Each process has its own logical control flow



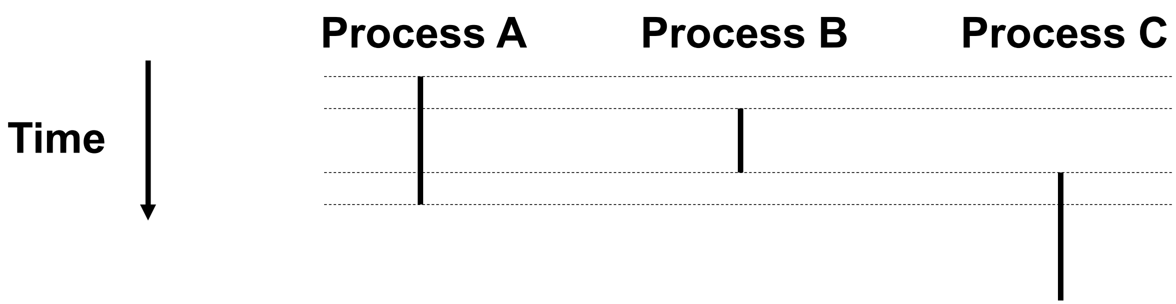
Concurrent Processes

- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time.
- Otherwise, they are *sequential.*
- Examples:
  - Concurrent: A & B, A & C
  - Sequential: B & C



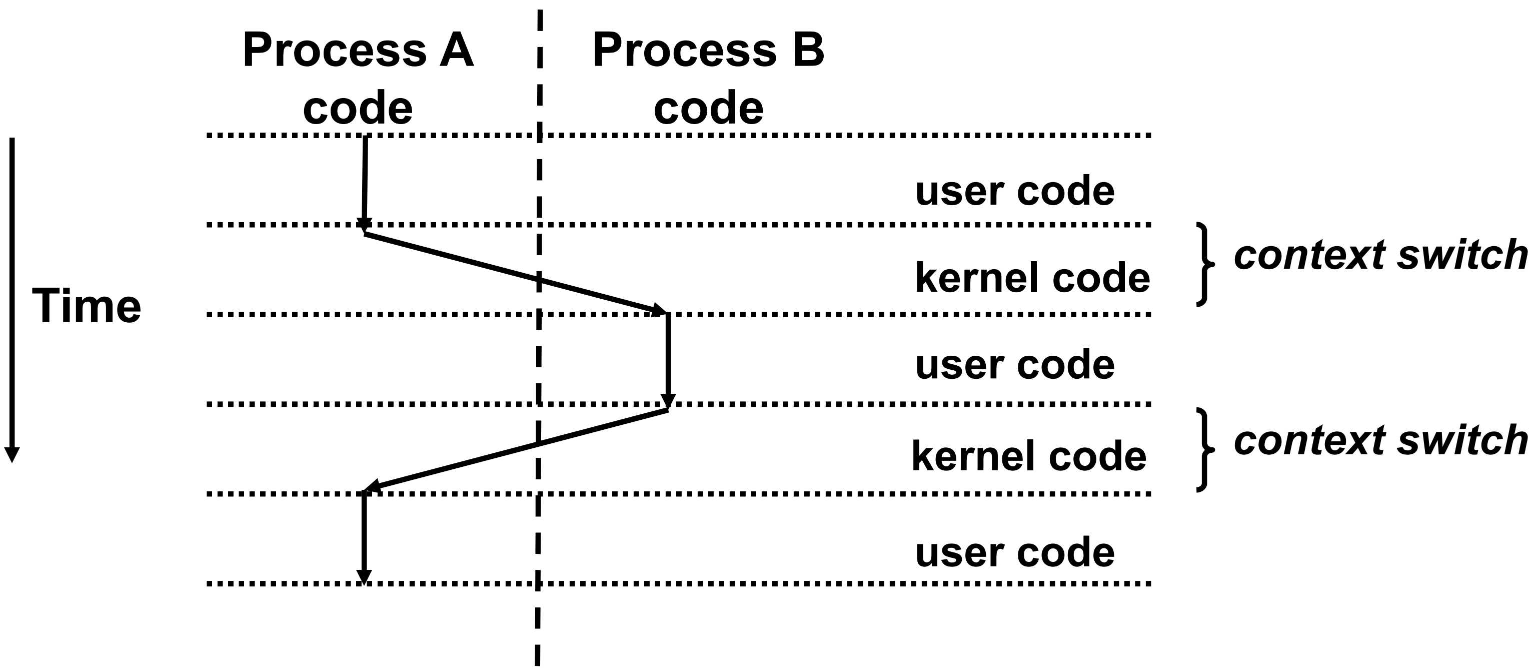
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time.
- However, we can think of concurrent processes are running in parallel with each other.



Context Switching

- Processes are managed by a shared chunk of OS code called the *kernel*
- Control flow passes from one process to another via a *context switch.*



Process: Traditional View

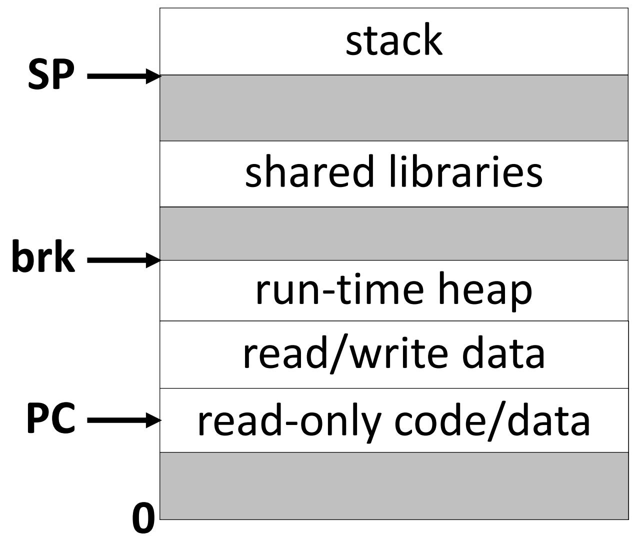
- Process = process context + code, data, and stack

Process context

Program context:
Data registers
Condition codes
Stack pointer (SP)
Program counter (PC)

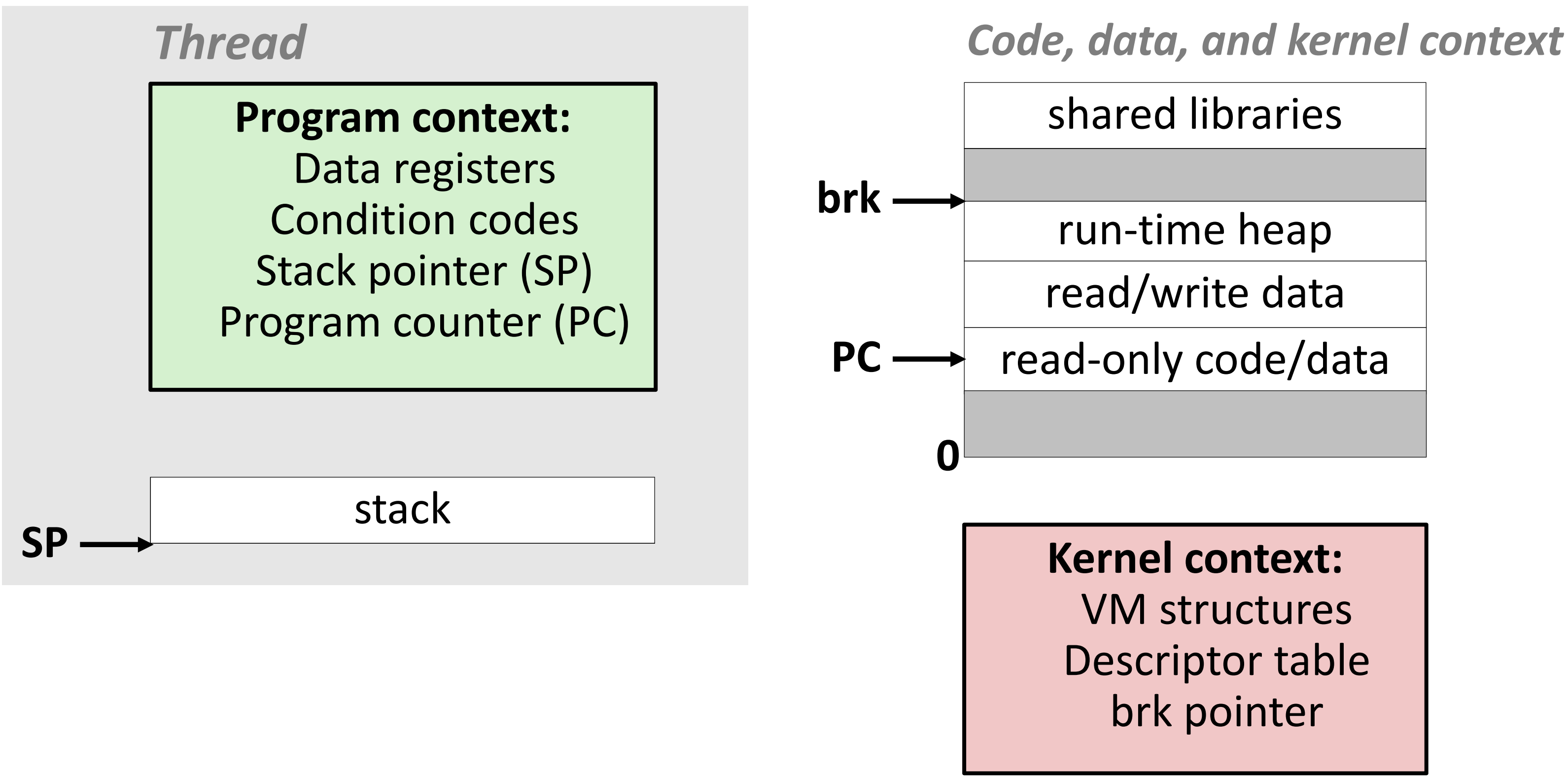
Kernel context:
VM structures
Descriptor table
brk pointer

Code, data, and stack

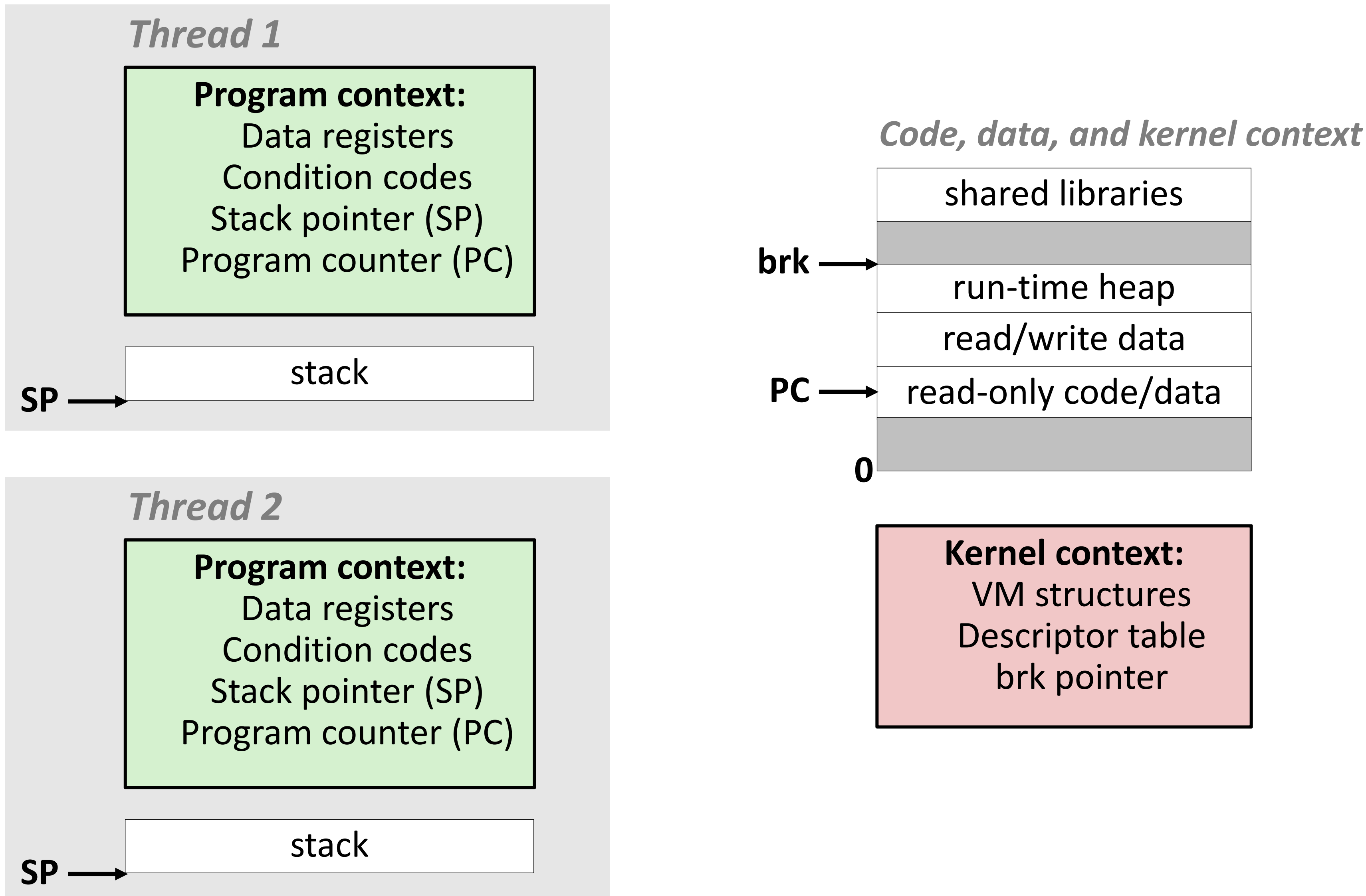


Process: Alternative View

- Process = thread + code, data, and kernel context



Process with Two Threads



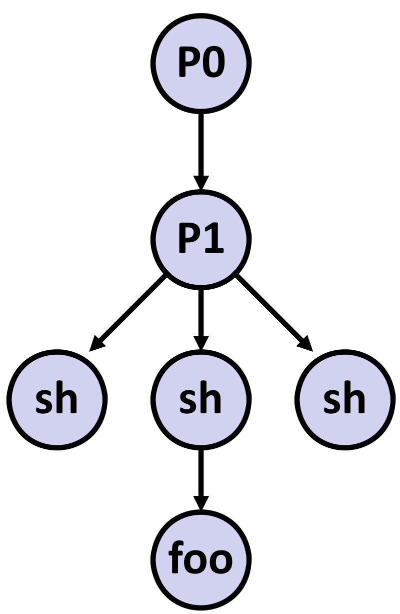
Threads vs. Processes

- Threads and processes: similarities
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched (scheduled) by the kernel
- Threads and processes: differences
  - Threads share code and data, processes (typically) do not
  - Threads are much less expensive than processes
    - Process control (creating and reaping) is more expensive as thread control
    - Context switches for processes much more expensive than for threads

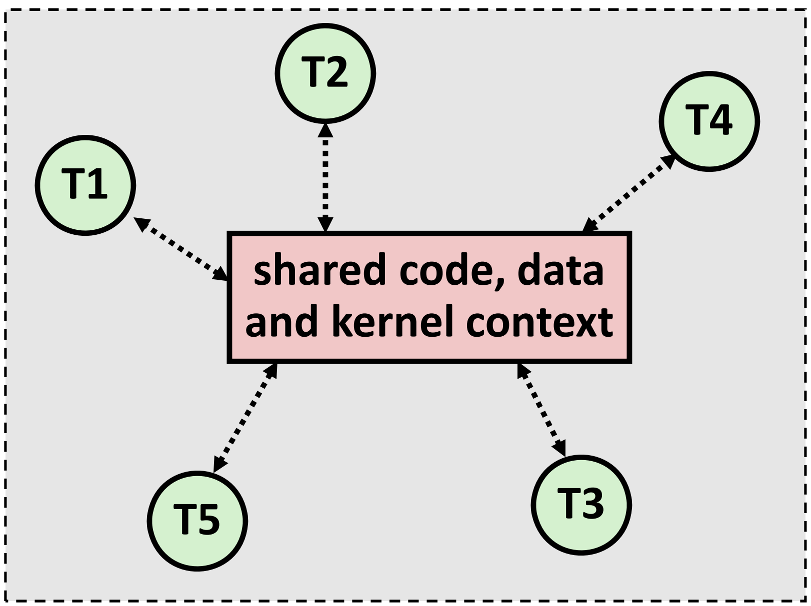
Threads vs. Processes (contd.)

- Processes form a tree hierarchy
- Threads form a pool of peers
  - Each thread can kill any other
  - Each thread can wait for any other thread to terminate
  - Main thread: first thread to run in a process

Process hierarchy

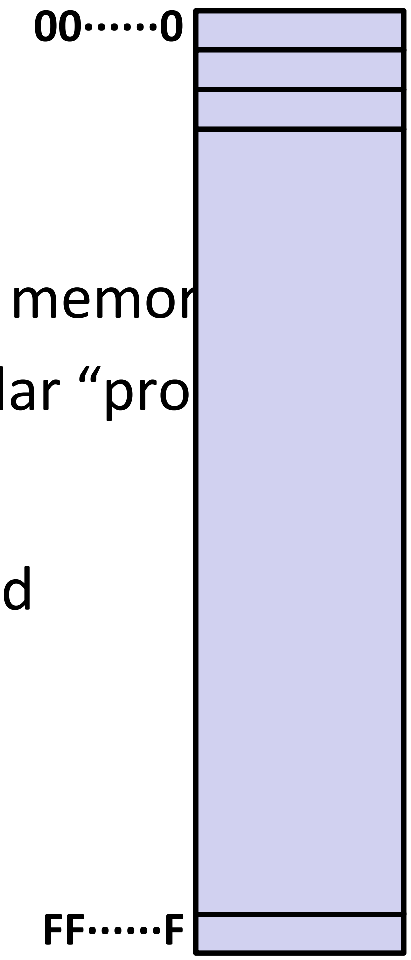


Thread pool



Virtual Memory (Previous Lectures)

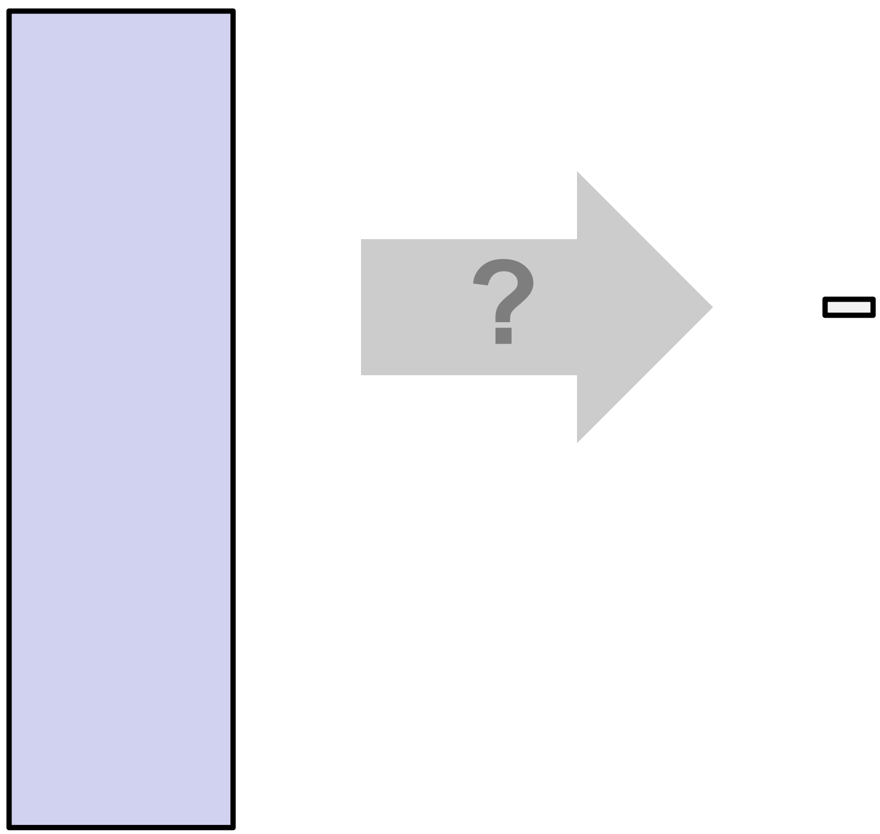
- Programs refer to virtual memory addresses
  - movl (%ecx),%eax
  - Conceptually very large array of bytes
  - Each byte has its own address
  - Actually implemented with hierarchy of different memor
  - System provides address space private to particular “pro
- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space
- But why virtual memory?
- Why not physical memory?



Problem 1: How Does Everything Fit?

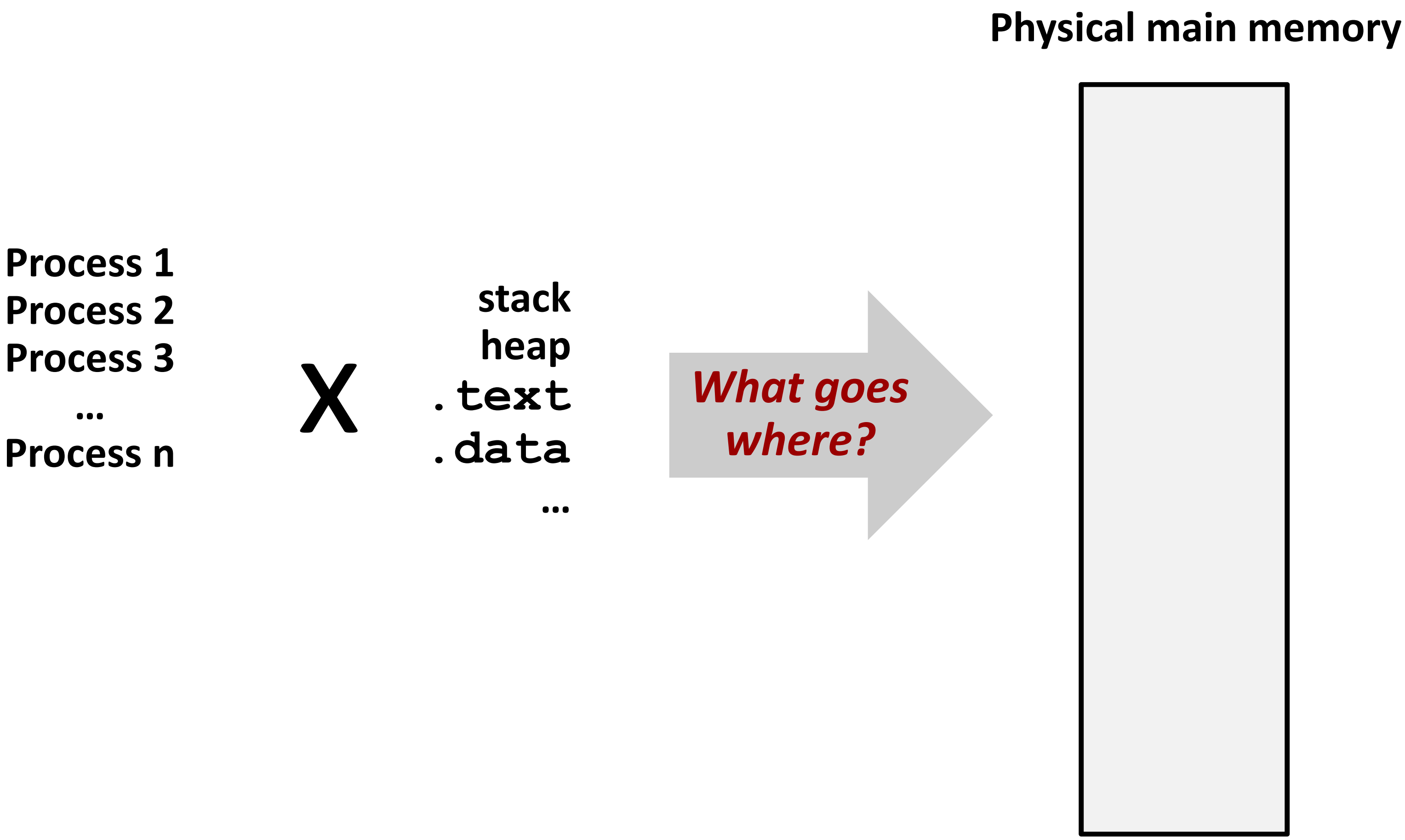
64-bit addresses:
16 Exabyte

Physical main memory:
Few Gigabytes

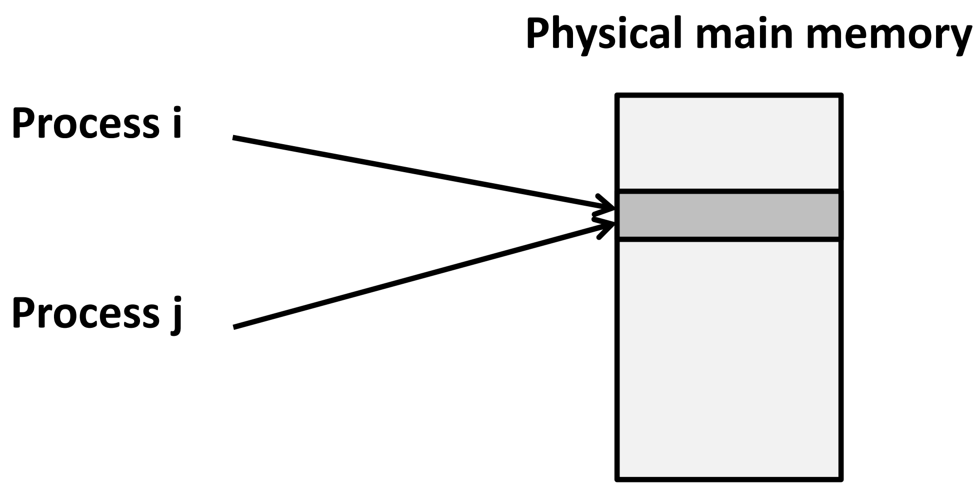


And there are many processes ….

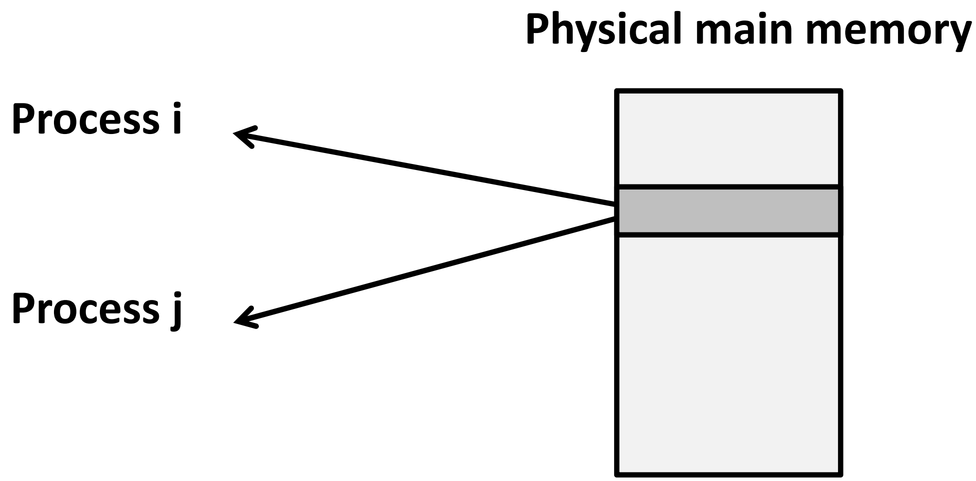
Problem 2: Memory Management



Problem 3: How To Protect

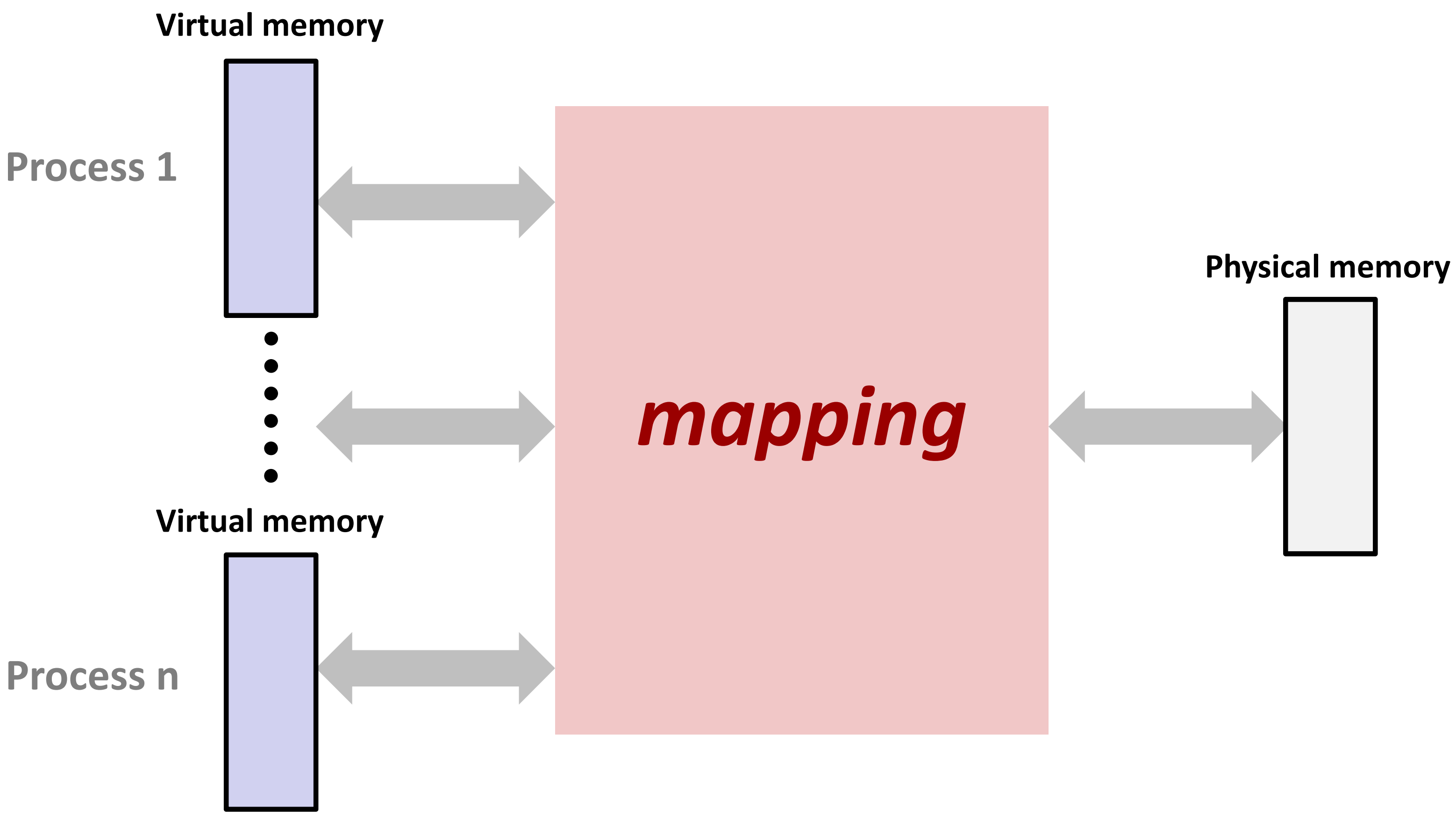


Problem 4: How To Share?



Solution: Level Of Indirection

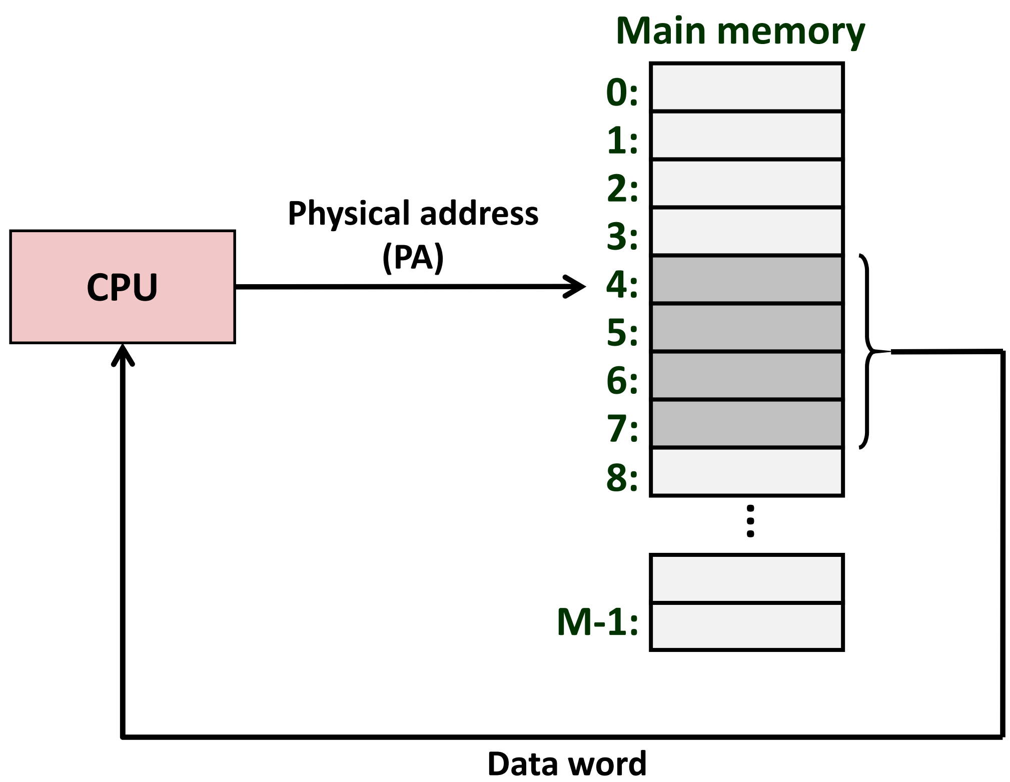
- Each process gets its own private memory space
- Solves the previous problems



Address Spaces

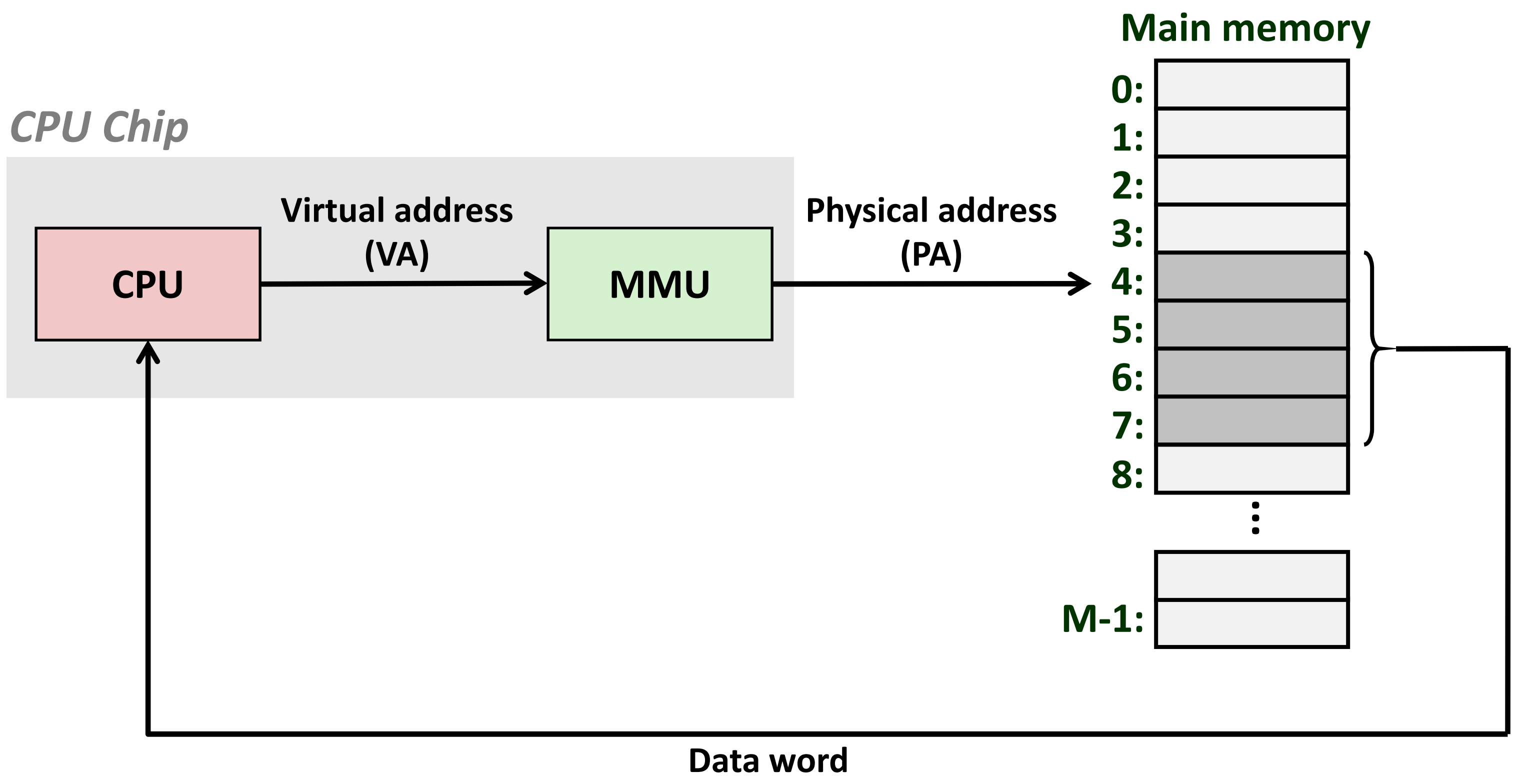
- Virtual address space
  - Set of N = 2n virtual addresses: {0, 1, 2, 3, …, N-1}
- Physical address space
  - Set of M = 2m physical addresses: {0, 1, 2, 3, …, M-1}
- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory:
  one physical address, one (or more) virtual addresses

A System Using Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



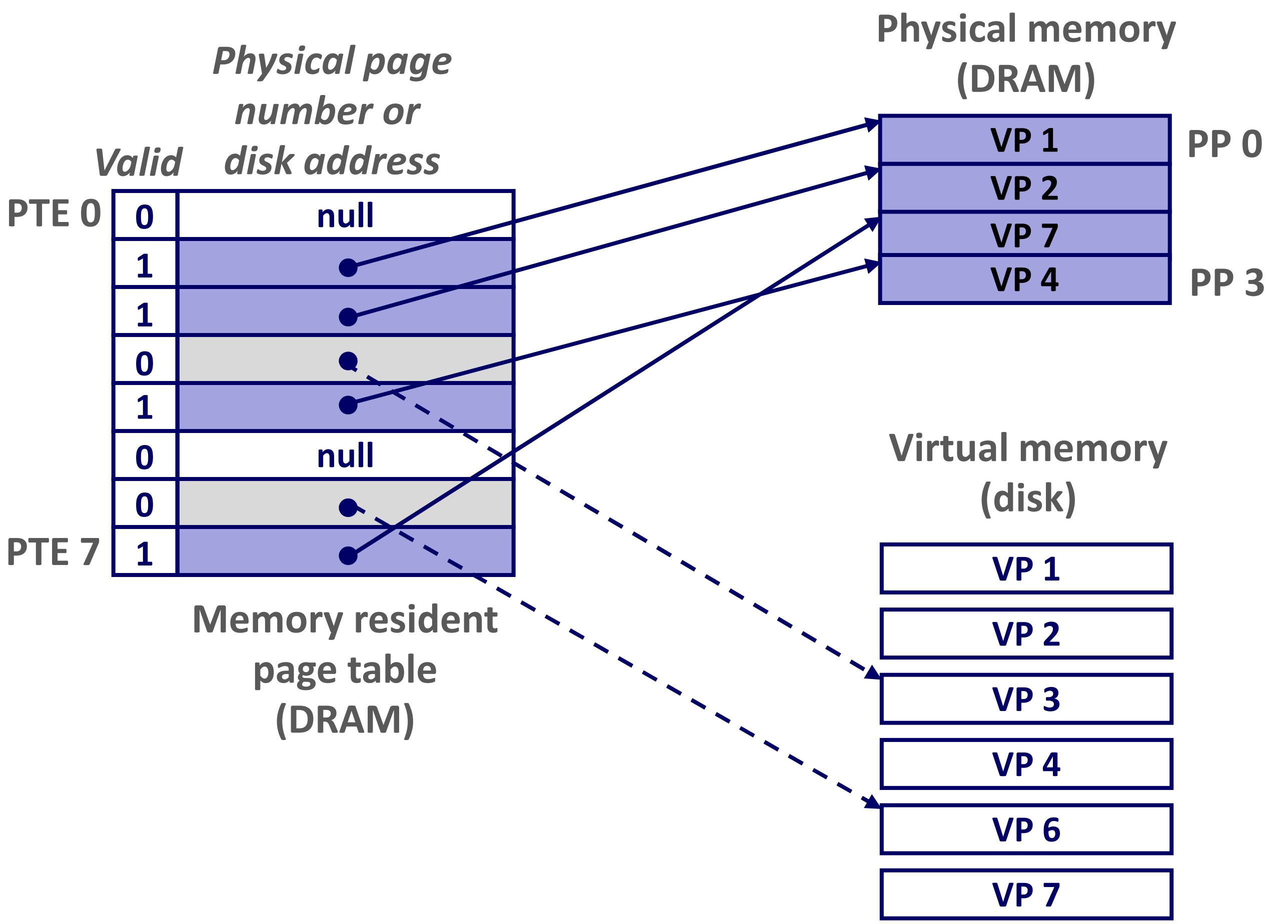
- Used in all modern desktops, laptops, workstations
- One of the great ideas in computer science
- MMU checks the cache

Why Virtual Memory (VM)?

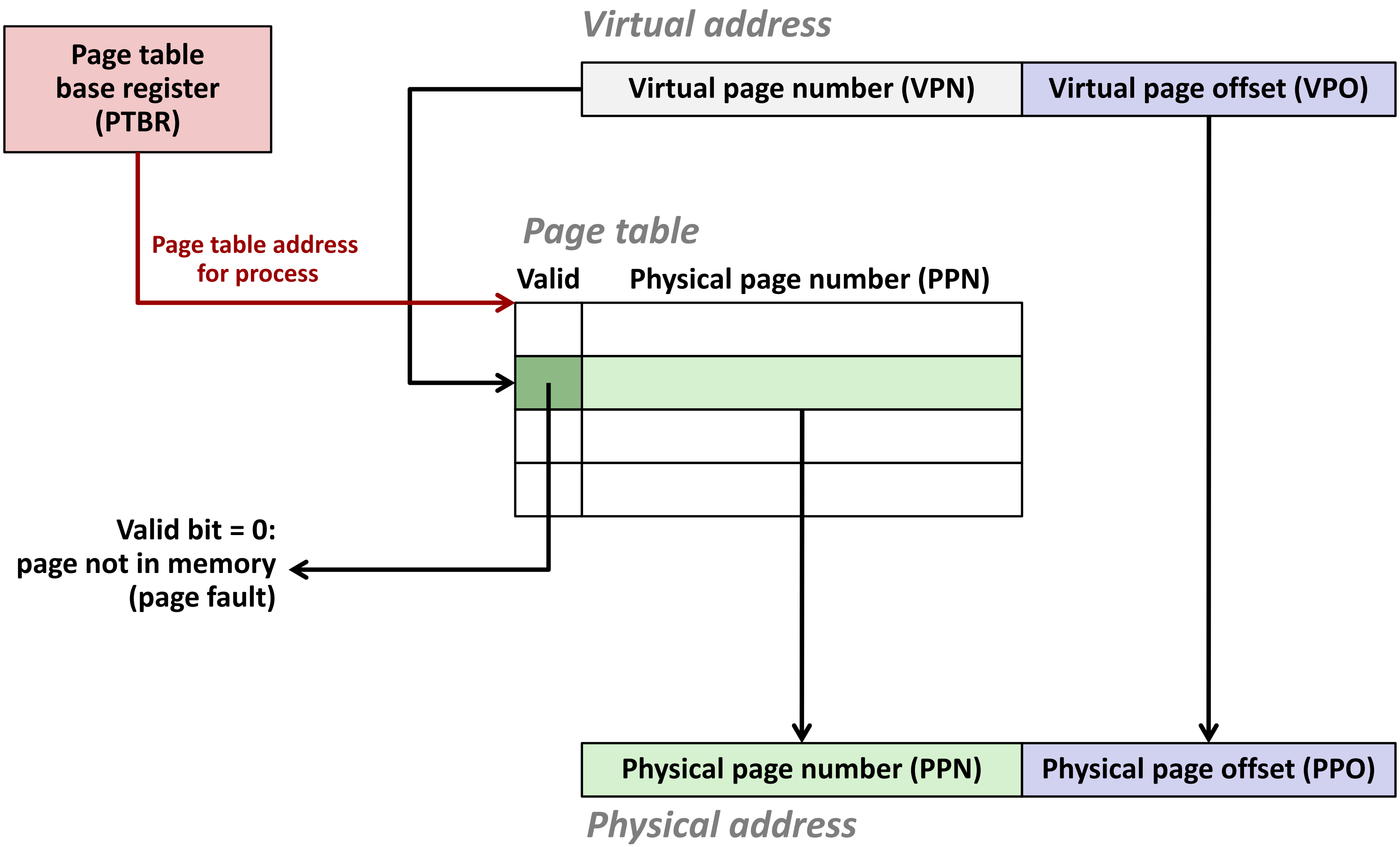
- Efficient use of limited main memory (RAM)
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed
- Simplifies memory management for programmers
  - Each process gets the same full, private linear address space
- Isolates address spaces
  - One process can't interfere with another's memory
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions

Address Translation: Page Tables

- A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages. Here: 8 VPs
  - Per-process kernel data structure in DRAM



Address Translation With a Page Table



Exceptions

Some notes adopted from Bryant and O'Hallaron

Control Flow

- Computers do Only One Thing
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions
  - This sequence is the system's physical *control flow* (or *flow of control*).

Physical control flow

Time

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

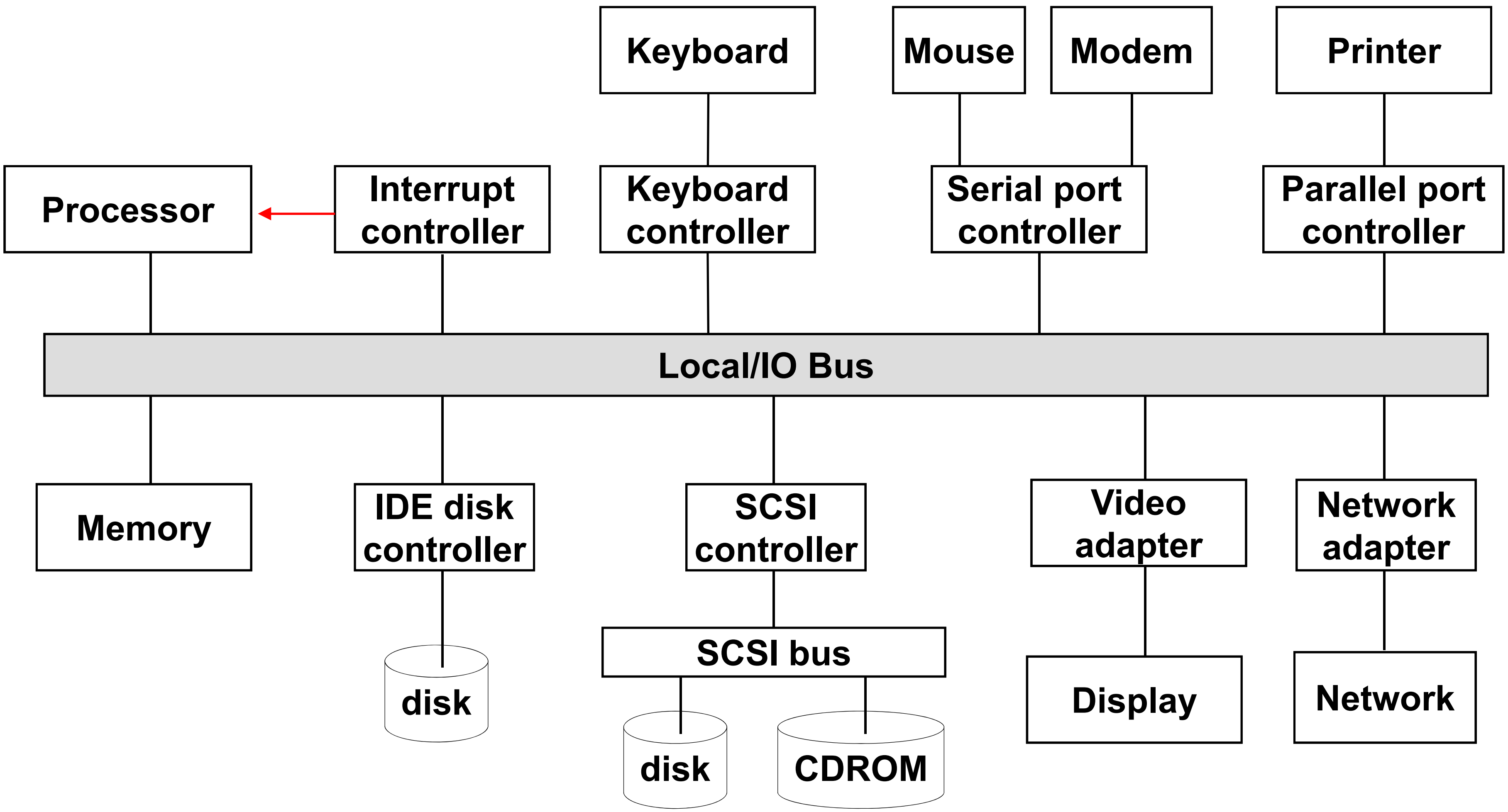
Altering the Control Flow

- Up to Now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return using the stack discipline.
  - Both react to changes in program state.
- Insufficient for a useful system
  - Difficult for the CPU to react to changes in system state.
    - data arrives from a disk or a network adapter.
    - Instruction divides by zero
    - User hits ctl-c at the keyboard
    - System timer expires
- System needs mechanisms for “exceptional control flow”

Exceptional Control Flow

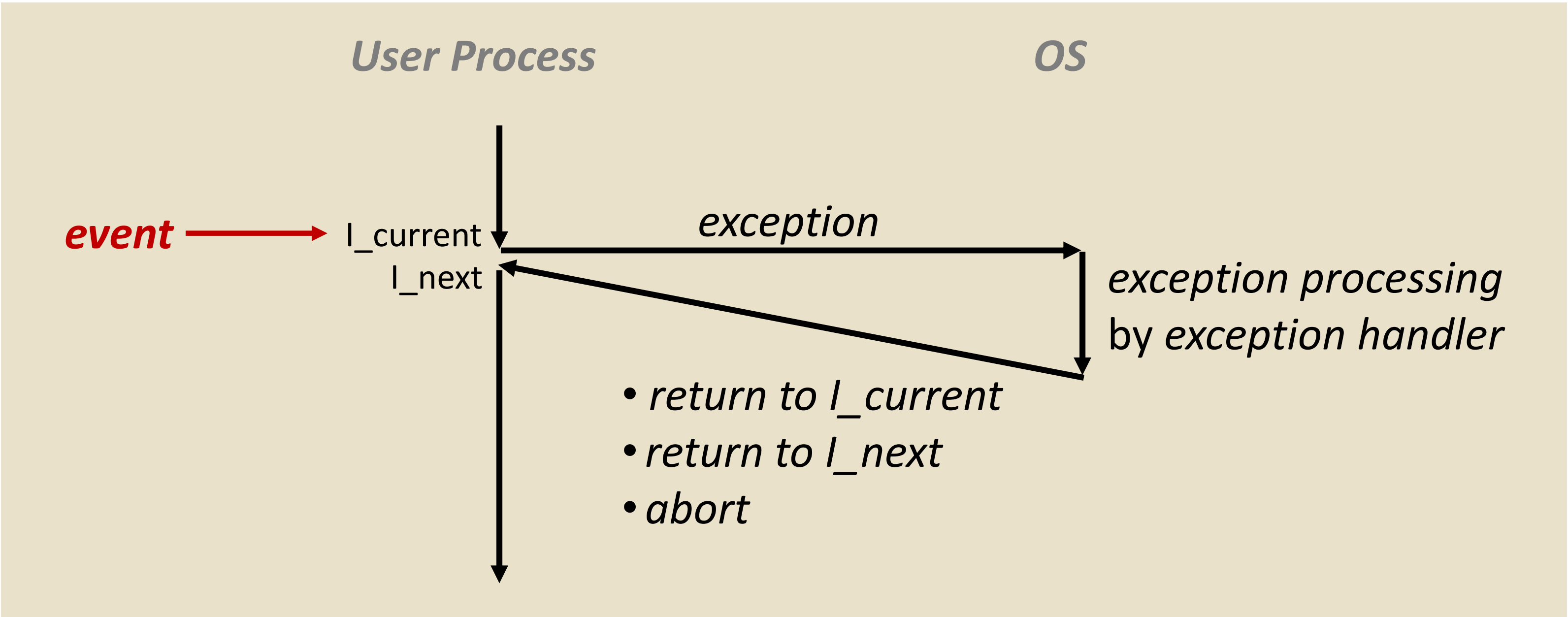
- Mechanisms for exceptional control flow exists at all levels of a computer system.
- Low level Mechanism
  - exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher Level Mechanisms
  - Process context switch
  - Signals
  - Nonlocal jumps (setjmp/longjmp)
  - Implemented by either:
    - OS software (context switch and signals).
    - C language runtime library: nonlocal jumps.

System context for exceptions



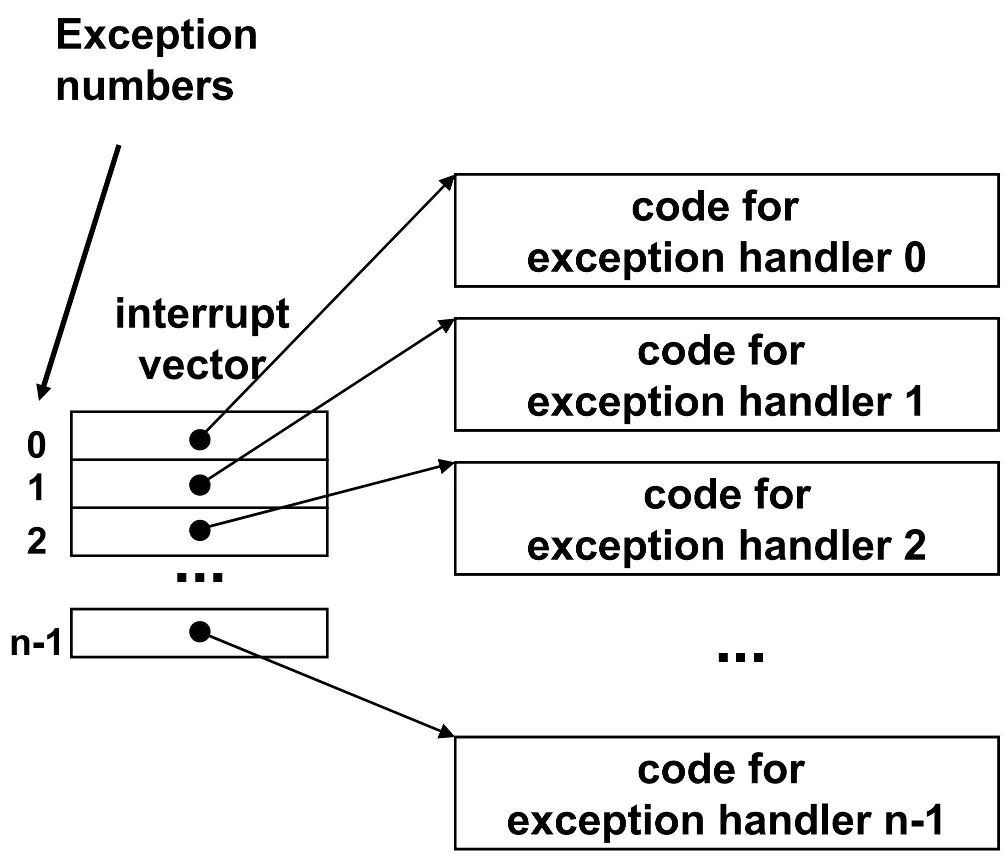
Exceptions

- An exception is a transfer of control to the OS in response to some event (i.e., change in processor state)



- Examples:
  div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C

Interrupt Vectors

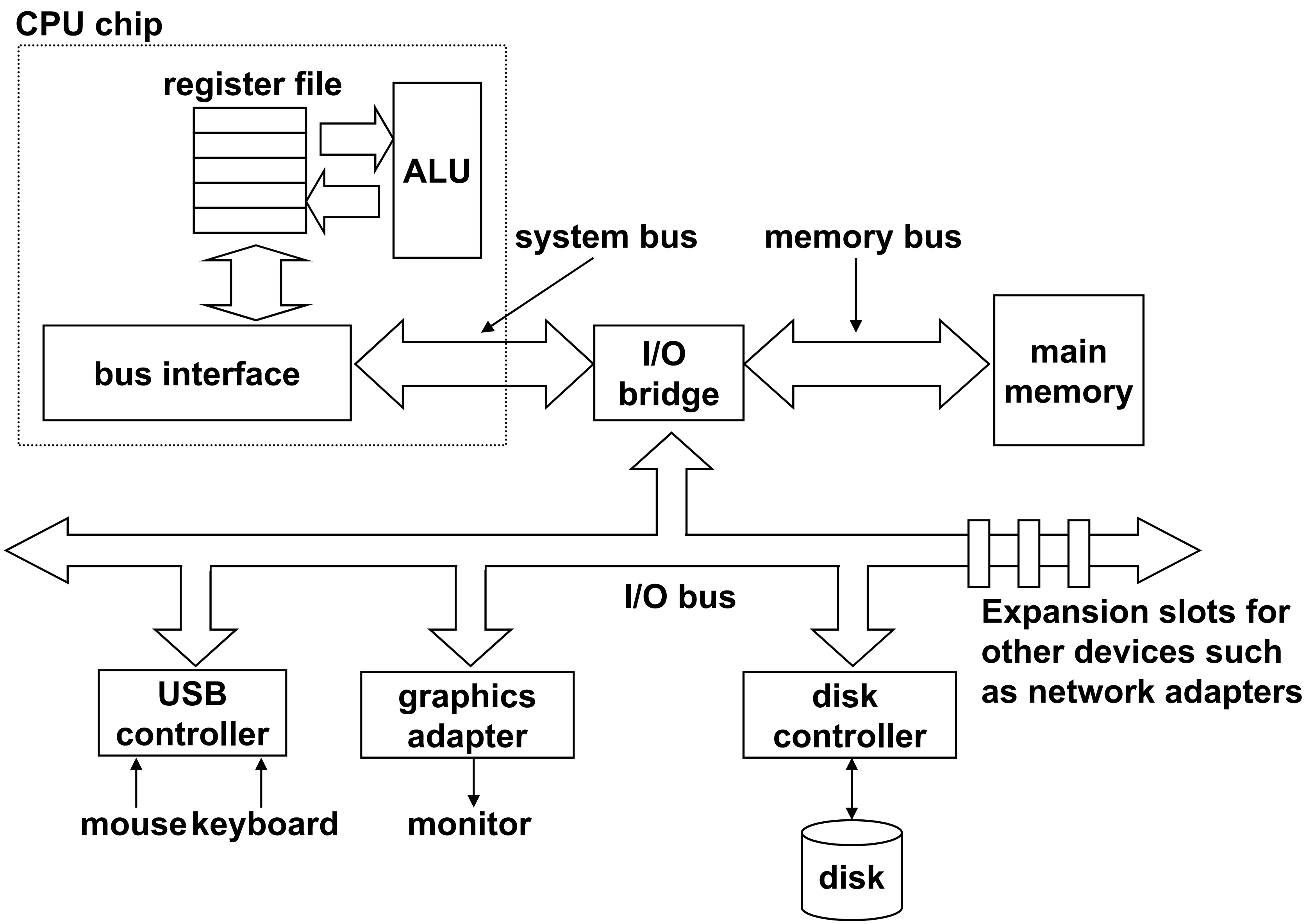


- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler k is called each time exception k occurs.

Asynchronous Exceptions (Interrupts)

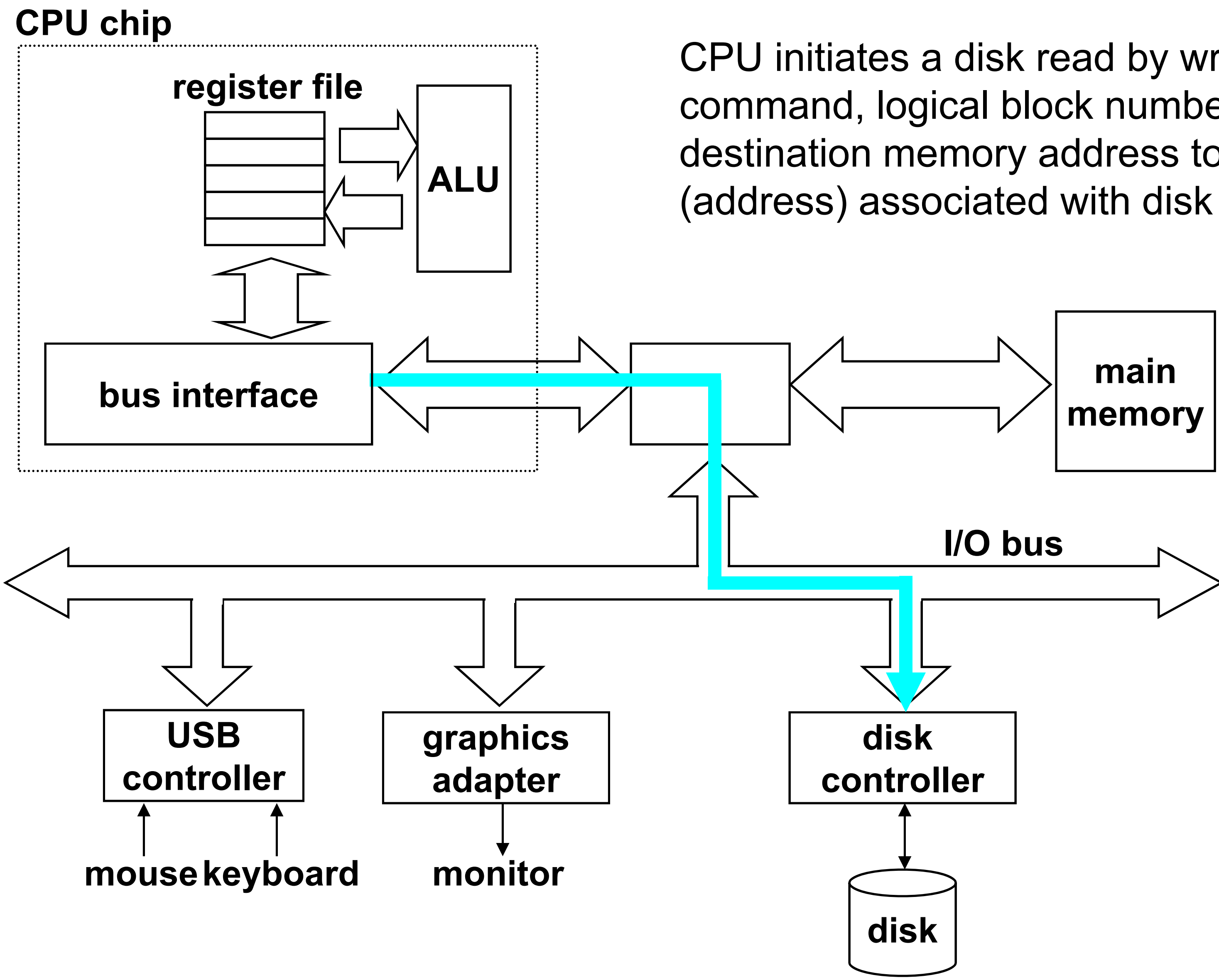
- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - handler returns to "next" instruction.
- Examples:
  - I/O interrupts
    - hitting ctl-c at the keyboard
    - arrival of a packet from a network
    - arrival of a data sector from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting ctl-alt-delete on a PC

A Typical Hardware System



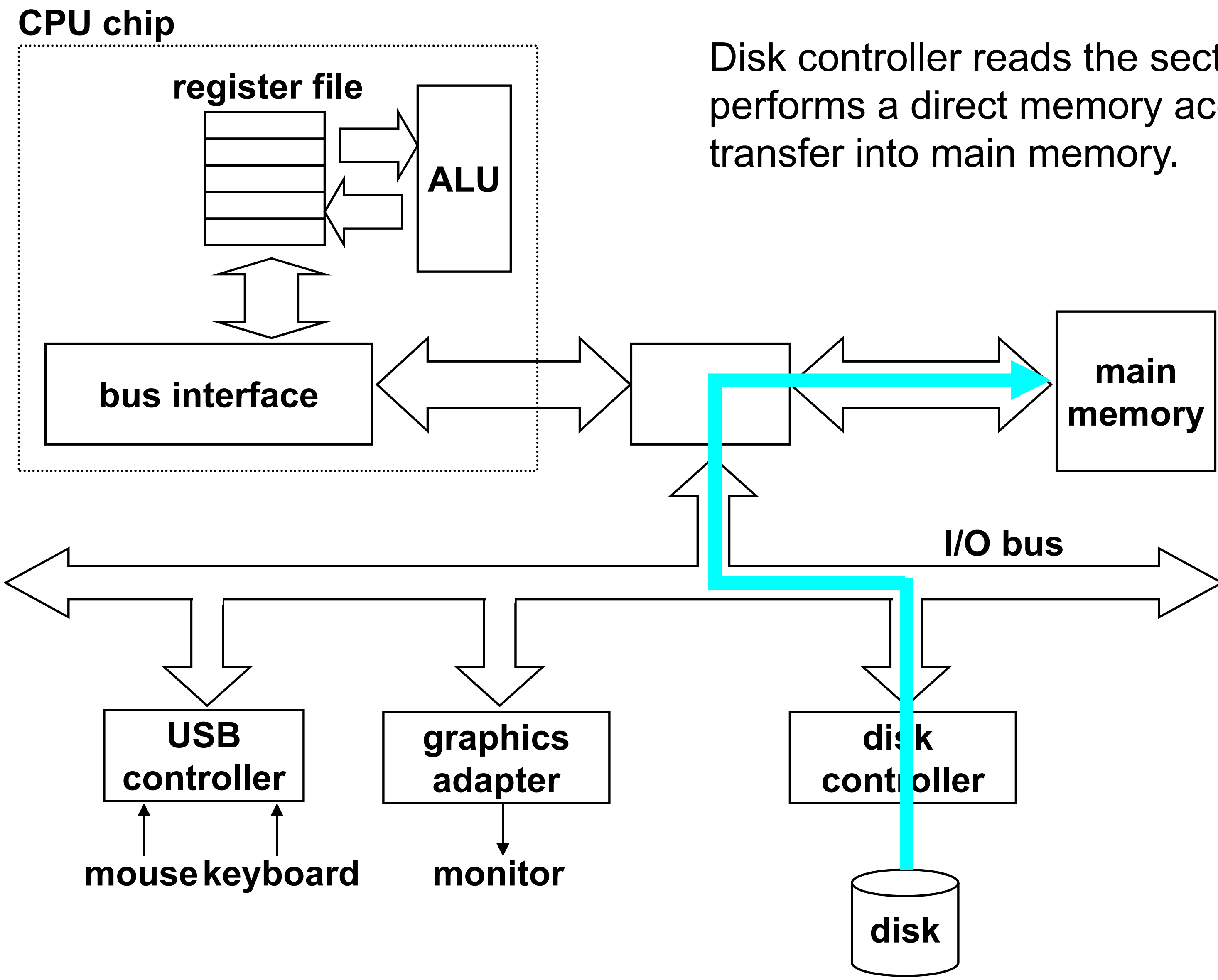
Reading a Disk Sector: Step 1

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.



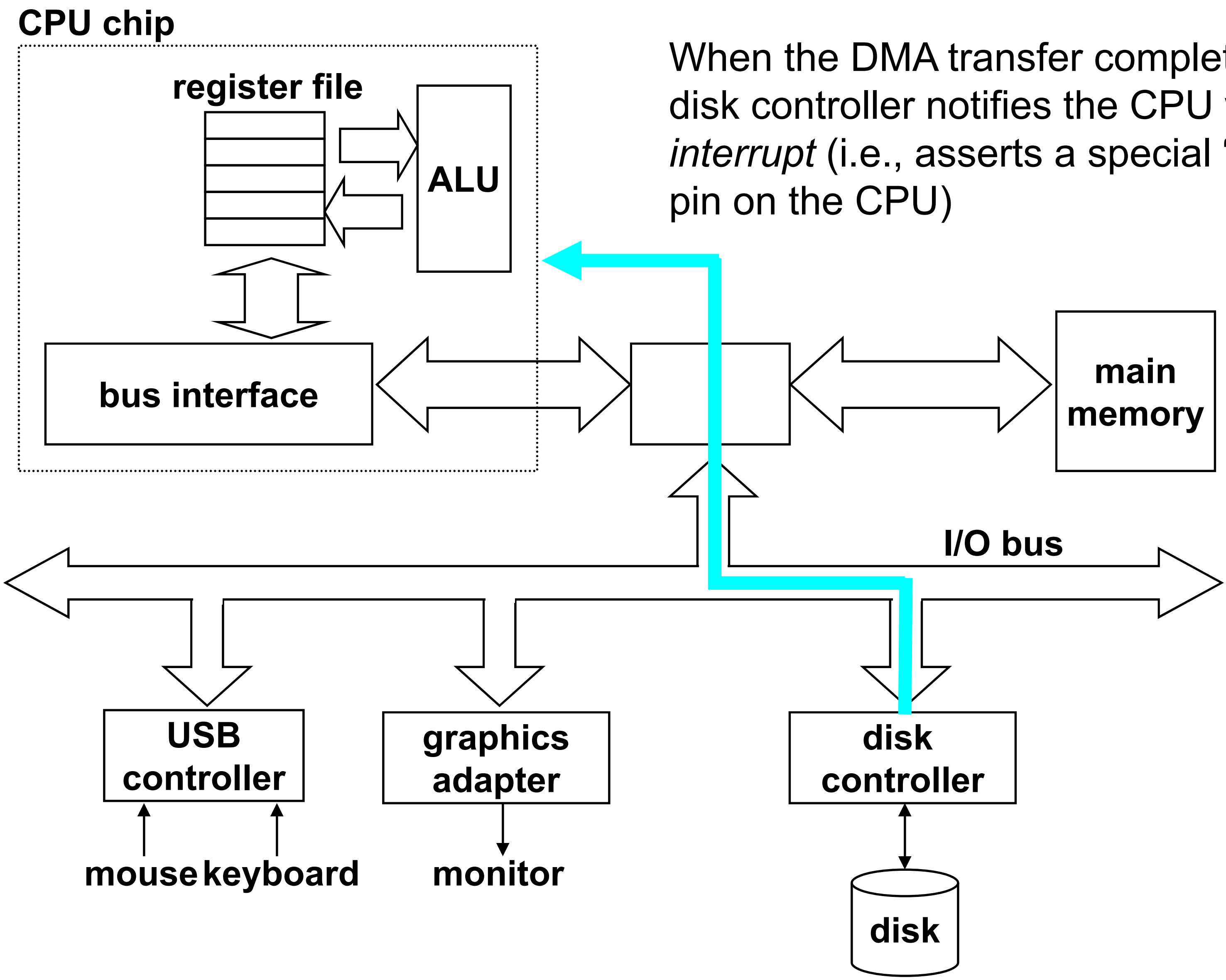
Reading a Disk Sector: Step 2

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.



Reading a Disk Sector: Step 3

When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)



Synchronous Exceptions

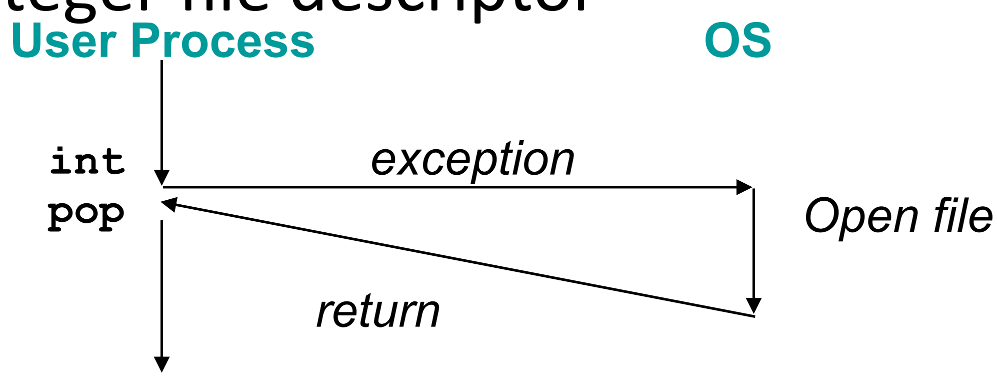
- Caused by events that occur as a result of executing an instruction:
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable).
    - Either re-executes faulting ("current") instruction or aborts.
  - Aborts
    - unintentional and unrecoverable
    - Examples: parity error, machine check.
    - Aborts current program

Trap Example

- Opening a File
  - User calls open(filename, options)

```
0804d070 <__libc_open>:
 . . .
 804d082:       cd 80                  int     $0x80
 804d084:       5b                     pop     %ebx
 . . .
```

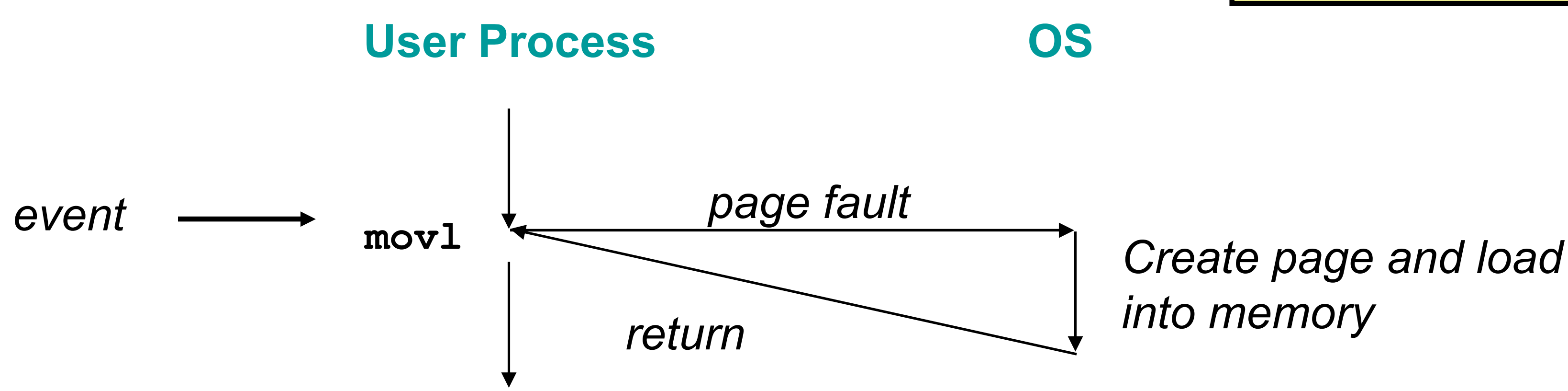
    - Function open executes system call instruction int
  - OS must find or create file, get it ready for reading or writing
  - Returns integer file descriptor



Fault Example #1

- Memory Reference
  - User writes to memory location
  - That portion (page) of user’s memory is currently on disk
  - Page handler must load page into physical memory
  - Returns to faulting instruction
  - Successful on second try

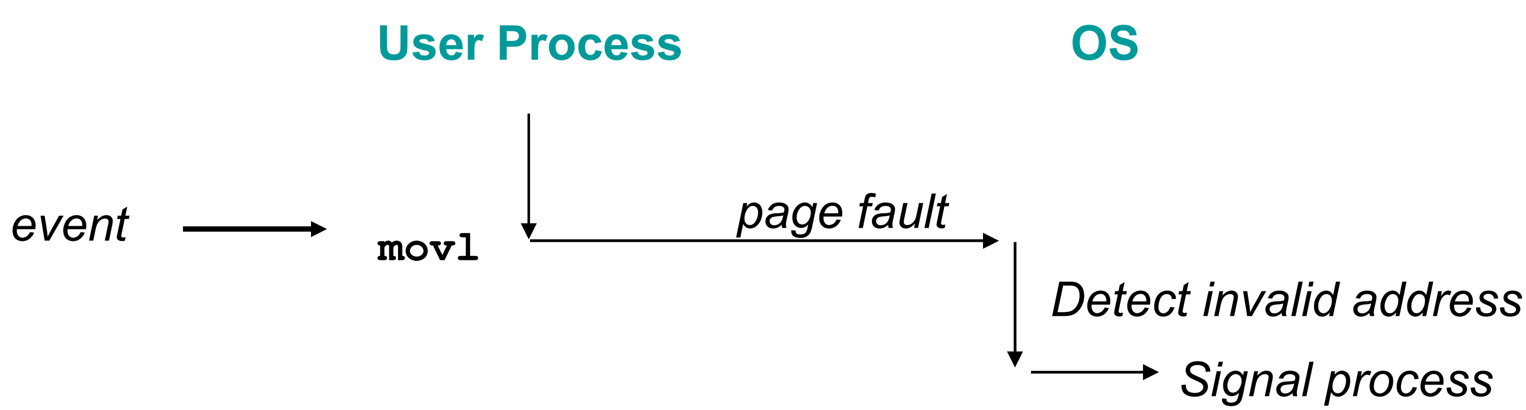
```
int a[1000];
main ()
{
    a[500] = 13;
}
```



Fault Example #2

- Memory Reference
  - User writes to memory location
  - Address is not valid
  - Page handler detects invalid address
  - Sends SIGSEG signal to user process
  - User process exits with “segmentation fault”

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```



Exception Table IA32 (Excerpt)

Exception Number	Description	Exception Class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-127	OS-defined	Interrupt or trap
128 (0x80)	System call	Trap
129-255	OS-defined	Interrupt or trap

Check pp. 183:
http://download.intel.com/design/processor/manuals/253665.pdf

Linkers

Some notes adopted from Bryant and O'Hallaron

Example C Program

main.c

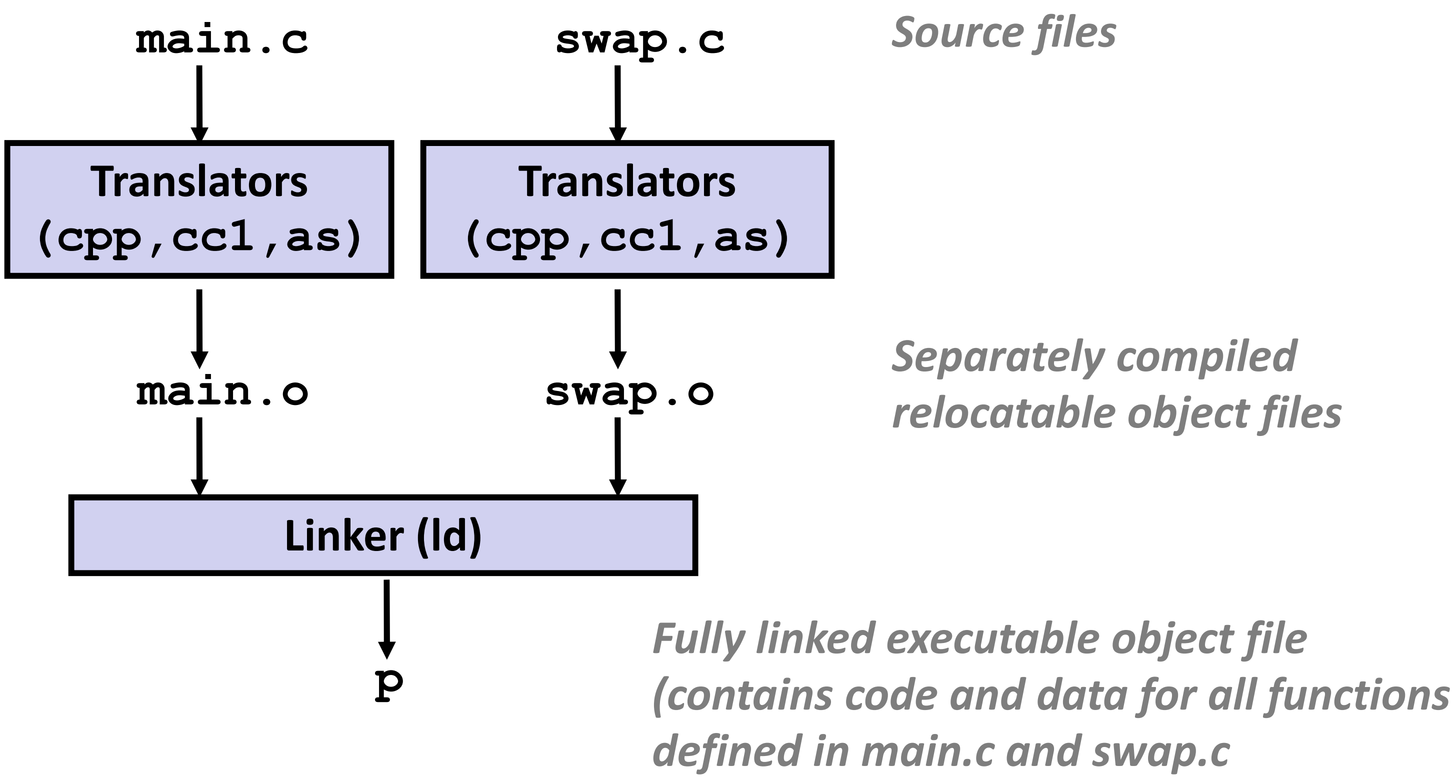
```c
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```c
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Static Linking

- Programs are translated and linked using a compiler driver:
  - unix> gcc -O2 -g -o p main.c swap.c
  - unix> ./p



Why Linkers? Modularity!

- Program can be written as a collection of smaller source files, rather than one monolithic mass.

- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library

Why Linkers? Efficiency!

- Time: Separate Compilation
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.

- Space: Libraries
  - Common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

What Do Linkers Do?

- Step 1: Symbol resolution
  - Programs define and reference symbols (variables and functions):
    - void swap() {…}   /* define symbol swap */
    - swap();   /* reference symbol swap */
    - int *xp = &x;   /* define xp, reference x */
  - Symbol definitions are stored (by compiler) in symbol table.
    - Symbol table is an array of structs
    - Each entry includes name, type, size, and location of symbol.
  - Linker associates each symbol reference with exactly one symbol definition.

What Do Linkers Do? (cont.)

- Step 2: Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Three Kinds of Object Files (Modules)

- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source (.c) file
- Executable object file
  - Contains code and data in a form that can be copied directly into memory and then executed.
- Shared object file (.so file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLLs) by Windows

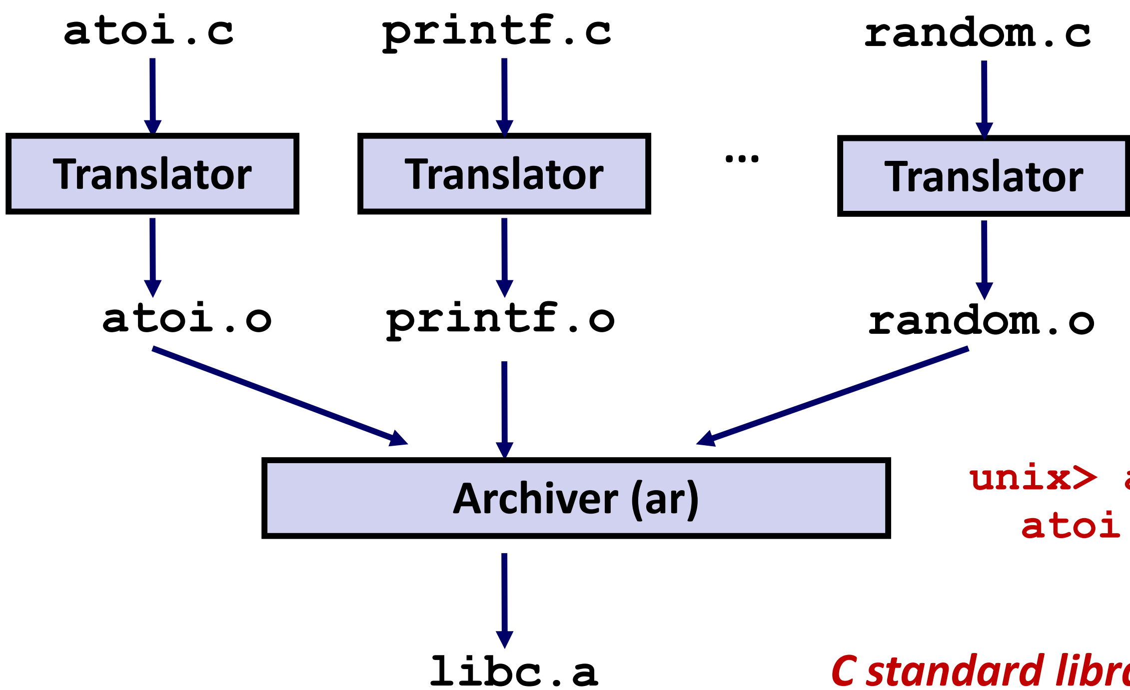
Packaging Commonly Used Functions`

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.
- Awkward, given the linker framework so far:
  - Option 1: Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - Option 2: Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Solution: Static Libraries

- Static libraries (.a archive files)
  - Concatenate related relocatable object files into a single file with an index (called an archive).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link into executable.

Creating Static Libraries




```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

C standard library

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

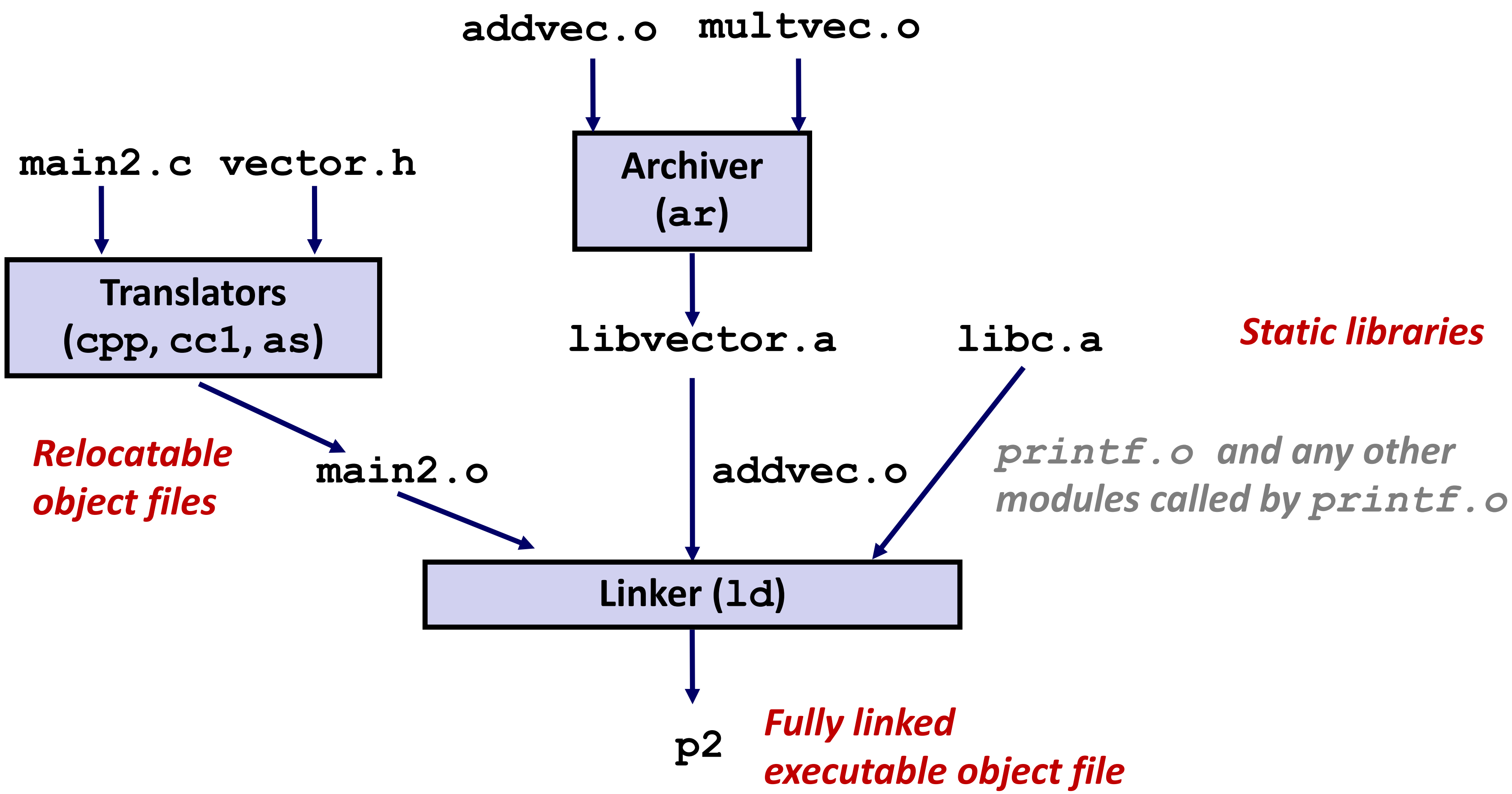
Commonly Used Libraries

- libc.a (the C standard library)
  - 8 MB archive of 900 object files.
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- libm.a (the C math library)
  - 1 MB archive of 226 object files.
  - floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries



Using Static Libraries

- Linker's algorithm for resolving external references:
  - Scan .o files and .a files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj.
  - If any entries in the unresolved list at end of scan, then error.
- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

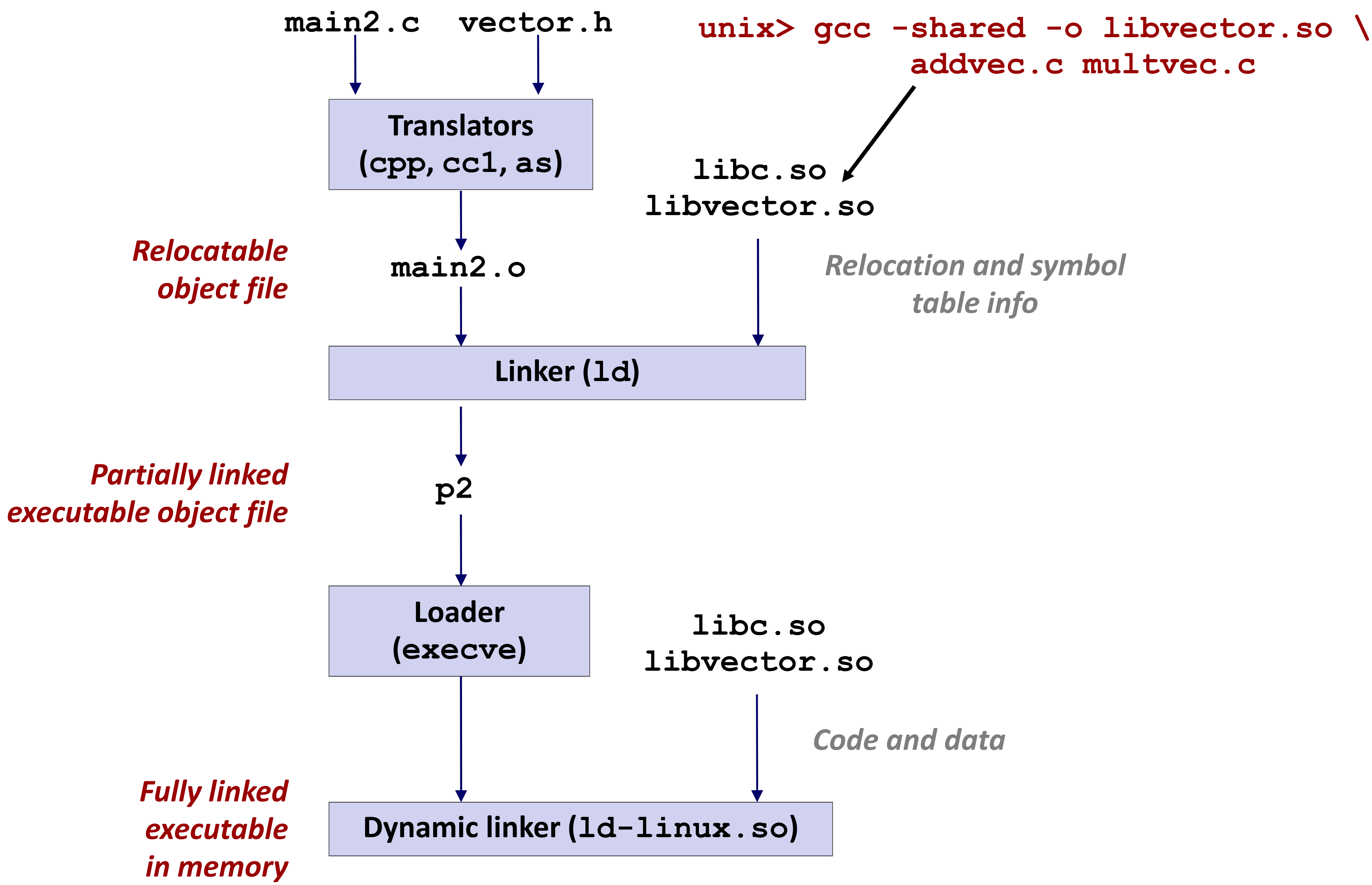
Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function need std libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
- Modern Solution: Shared Libraries
  - Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
  - Also called: dynamic link libraries, DLLs, .so files

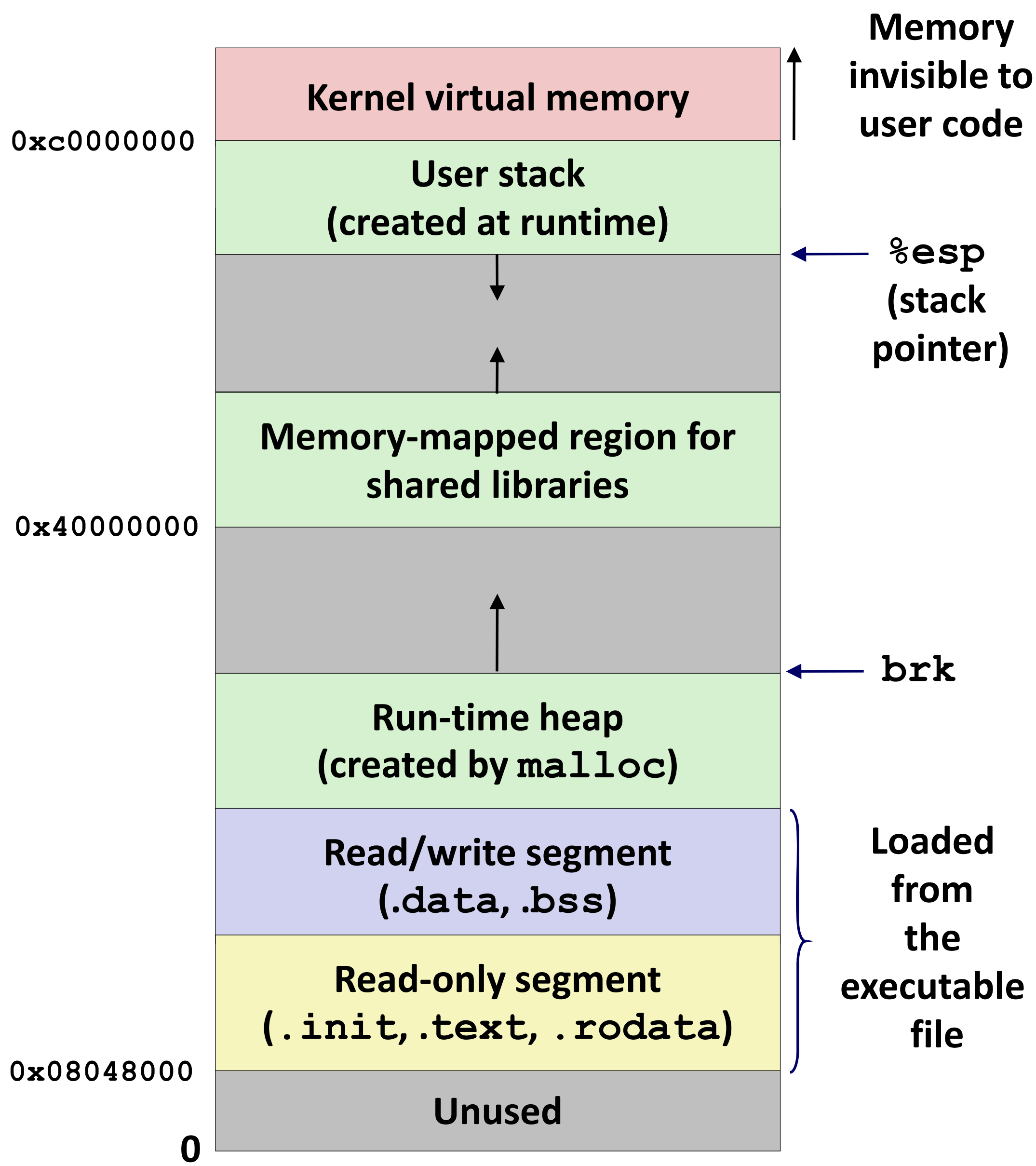
Shared Libraries (cont.)

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker (ld-linux.so).
  - Standard C library (libc.so) usually dynamically linked.
- Dynamic linking can also occur after program has begun (run-time linking).
  - In Unix, this is done by calls to the dlopen() interface.
    - High-performance web servers.
    - Runtime library interpositioning
- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory

Dynamic Linking at Load-time



Refined View of Memory



Case Study: Library Interpositioning

- Library interpositioning is a powerful linking technique that allows programmers to intercept calls to arbitrary functions
- Interpositioning can occur at:
  - compile time
    - When the source code is compiled
  - link time
    - When the relocatable object files are linked to form an executable object file
  - load/run time
    - When an executable object file is loaded into memory, dynamically linked, and then executed.

Some Interpositioning Applications

- Security
  - Confinement (sandboxing)
    - Interpose calls to libc functions.
  - Behind the scenes encryption
    - Automatically encrypt otherwise unencrypted network connections.
- Monitoring and Profiling
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - Malloc tracing
    - Detecting memory leaks
    - Generating malloc traces