Systems Programming
Final Review

Tongping Liu
tongpingliu@cs.utsa.edu

Types of Questions

Selection(24 Points)

Answering Questions (30 Points)

Program Understanding (24 Points)

Programming (42 Points)

What we have covered in 1st half semester?

Utility

Shell

Awk

Sed

Perl

C Programming

What we have covered in 2nd half semester?

Linking and Loading

Calling Convention

Virtual Memory

IO/FS

Testing, Debugging, and Profiling

Processes

Threads

Network, Cloud Computing and Big Data

Linking and Loading

ELF format

Linking

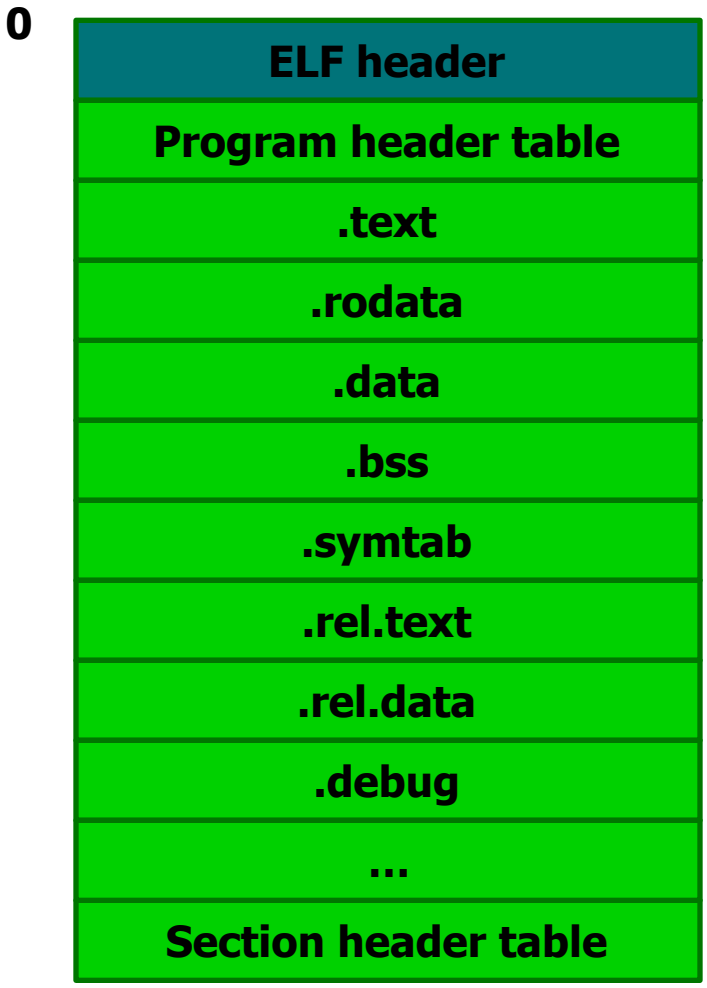
Loading

Running a program

ELF Header

Basic description of file contents:

- File format identifier
- Endianness
- Alignment for other sections
- Location of other sections
- Code's starting address
- …

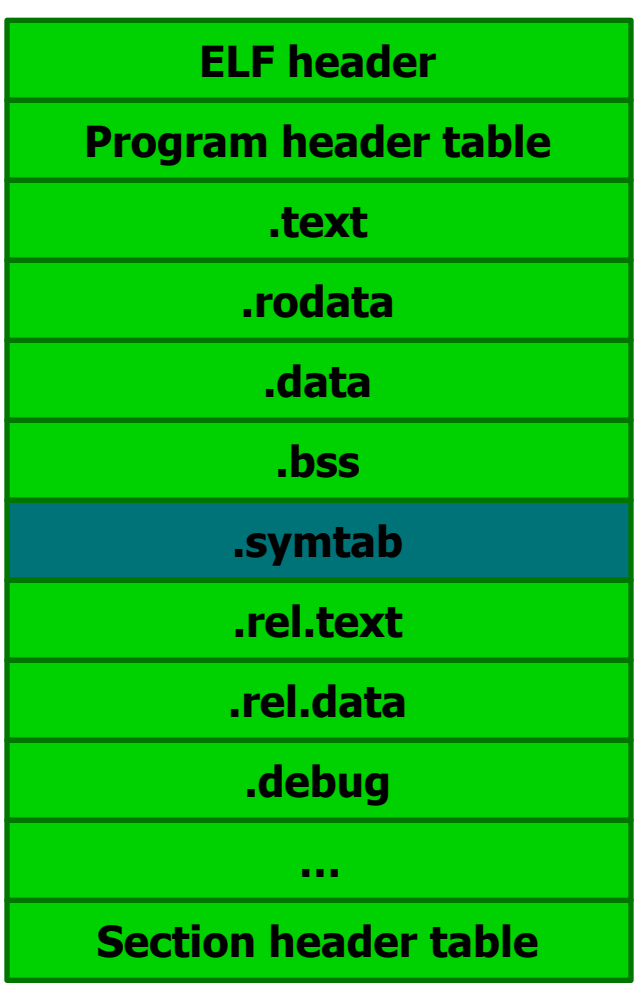


Symbol Table

Describes where global variables and functions are defined

- Present in all relocatable ELF files

```
/* main.c */
void swap(void);
int buf[2] = {1, 2};

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



Linker Symbol Classification

Global symbols

- Symbols defined by module ***m*** that can be referenced by other modules
- C: non-`static` functions & global variables

External symbols

- Symbols referenced by module ***m*** but defined by some other module
- C: `extern` functions & variables

Local symbols

- Symbols that are defined and referenced exclusively by module ***m***
- C: `static` functions & variables

Local linker symbols ≠ local function variables!

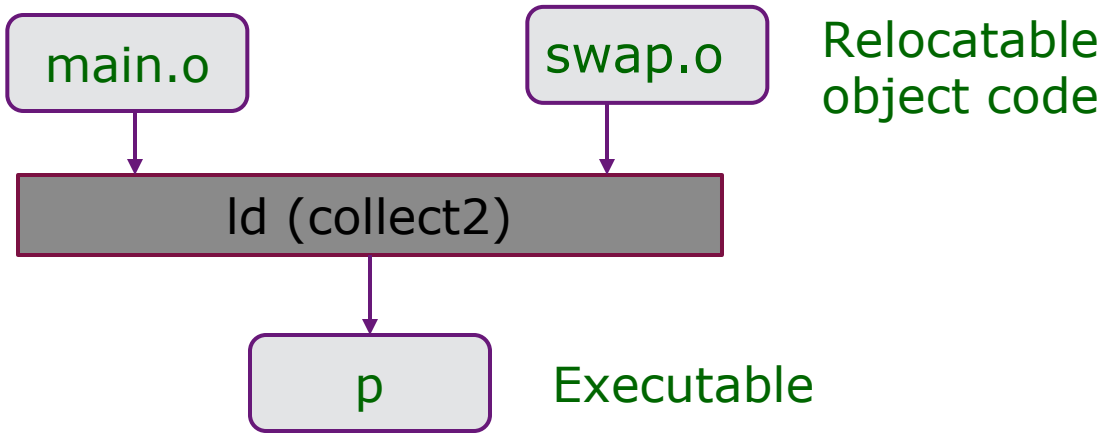
Linking Steps

Symbol Resolution

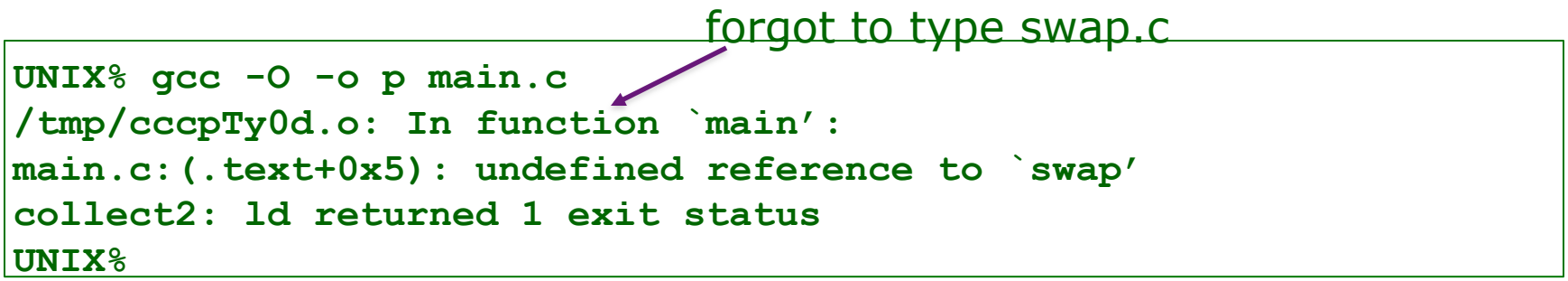
- Determine where symbols are located and what size data/ code they refer to

Relocation

- Combine modules, relocate code/data, and fix symbol references based on new locations



Problem: Undefined Symbols



Missing symbols are not compiler errors

- May be defined in another file
- Compiler just inserts an undefined entry in the symbol table

During linking, any undefined symbols that cannot be resolved cause an error

Shared libraries

Static libraries have the following disadvantages:

- Duplicating lots of common code in the executable files on a filesystem.
  - e.g., every C program needs the standard C library
- Duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink

Solution:

- ***shared libraries*** (dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
  - dynamic linking can occur when executable is first loaded and run.
    - common case for Linux, handled automatically by ld-linux.so.
  - dynamic linking can also occur after program has begun.
    - in Linux, this is done explicitly by user with dlopen().
  - shared library routines can be shared by multiple processes.

LD_PRELOAD mechanism

Build shared libraries memlib.so

- `gcc -fpic -shared -o memlib.so memlib.c`

Set LD_PRELOAD

- `LD_PRELOAD=/path/to/memlib.so ./memtest`

Demo: memtest.c
/proc/PID/maps

Static & Dynamic Libraries

Static

- **Library code added to executable file**
- **Larger executables**
- **Must recompile to use newer libraries**
- **Executable is self-contained**
- **Some time to load libraries at compile-time**
- **Library code shared only among copies of same program**

Dynamic

- Library code not added to executable file
- Smaller executables
- Uses newest (or smallest, fastest, …) library without recompiling
- Depends on libraries at run-time
- Some time to load libraries at run-time
- Library code shared among all uses of library

What we have covered in 2nd half semester?

Linking and Loading

Calling Convention

Virtual Memory

Testing, Debugging, and Profiling

Processes

Threads

Network, Cloud Computing and Big Data

Calling Convention and Library

Function Call Convention

Library Call

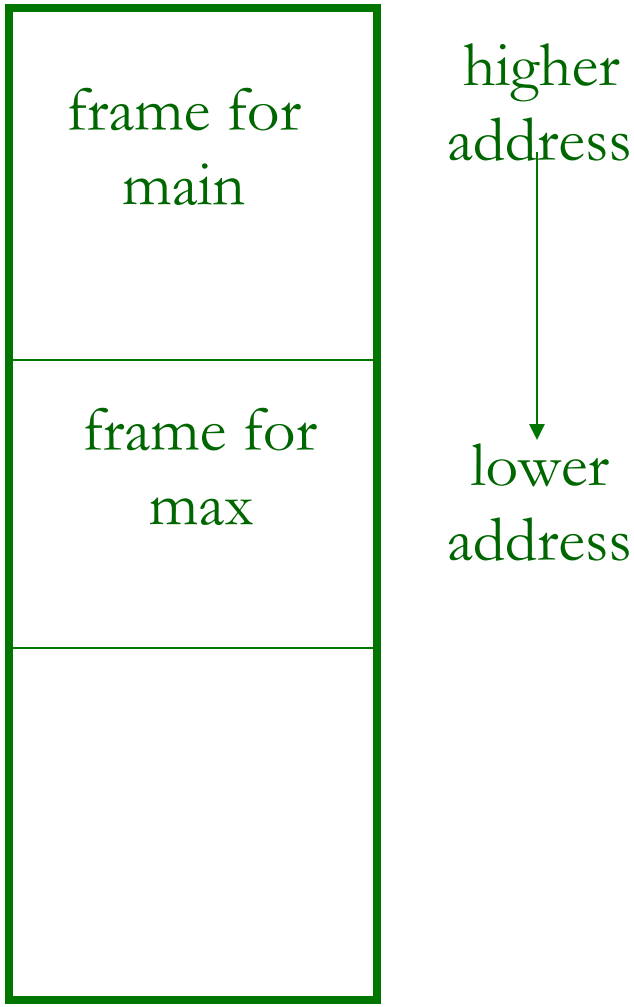
Exploiting Buffer Overflow Problems

Stack Frames:

The trick is to associate a frame with each *invocation* of a procedure.

We store data belonging to the invocation (e.g., the return address) in the frame.

```c
int max(int a, int b) {
  if(a > b)
    return a;
  else
    return b;
}

int main()
{
  int maxValue = max(3, 4);
  printf("maxvalue is %d\n", maxValue);
}
```



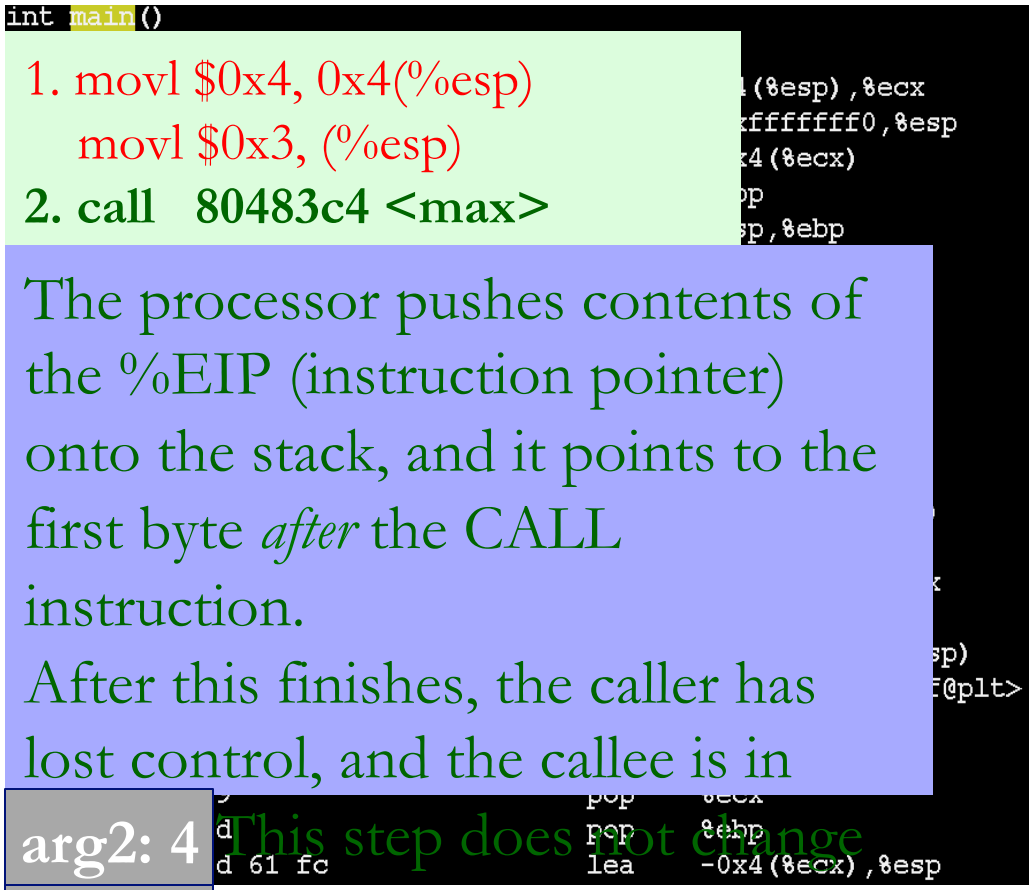
C Calling Convention

1. Push parameters onto the stack, from right to left
2. Call the function
3. Save and update the %ebp
4. Save CPU registers used for temporaries
5. Allocate local variables
6. Perform the function's purpose
7. Release local storage
8. Restore saved registers
9. Restore the old base pointer
10. Return from the function
11. Clean up pushed parameters

```
int main()
{
 80483e5:   8d 4c 24 04             lea    0x4(%esp),%ecx
 80483e9:   83 e4 f0                and    $0xfffffff0,%esp
 80483ec:   ff 71 fc                pushl  -0x4(%ecx)
 80483ef:   55                      push   %ebp
 80483f0:   89 e5                   mov    %esp,%ebp
 80483f2:   51                      push   %ecx
 80483f3:   83 ec 24                sub    $0x24,%esp
  int maxValue = max(3, 4);
 80483f6:   c7 44 24 04 04 00 00    movl   $0x4,0x4(%esp)
 80483fd:   00
 80483fe:   c7 04 24 03 00 00 00    movl   $0x3,(%esp)
 8048405:   e8 ba ff ff ff          call   80483c4 <max>
 804840a:   89 45 f8                mov    %eax,-0x8(%ebp)
  printf("maxvalue is %d\n", maxValue);
 804840d:   8b 45 f8                mov    -0x8(%ebp),%eax
 8048410:   89 44 24 04             mov    %eax,0x4(%esp)
 8048414:   c7 04 24 f0 84 04 08    movl   $0x80484f0,(%esp)
 804841b:   e8 d8 fe ff ff          call   80482f8 <printf@plt>
}
 8048420:   83 c4 24                add    $0x24,%esp
 8048423:   59                      pop    %ecx
 8048424:   5d                      pop    %ebp
 8048425:   8d 61 fc                lea    -0x4(%ecx),%esp
```

C Calling Convention

1. Push parameters onto the stack, from right to left
2. Call the function
3. Save and update the %ebp
4. Save CPU registers used for temporaries
5. Allocate local variables
6. Perform the function's purpose
7. Release local storage
8. Restore saved registers
9. Restore the old base pointer
10. Return from the function
11. Clean up pushed parameters



I386 Registers

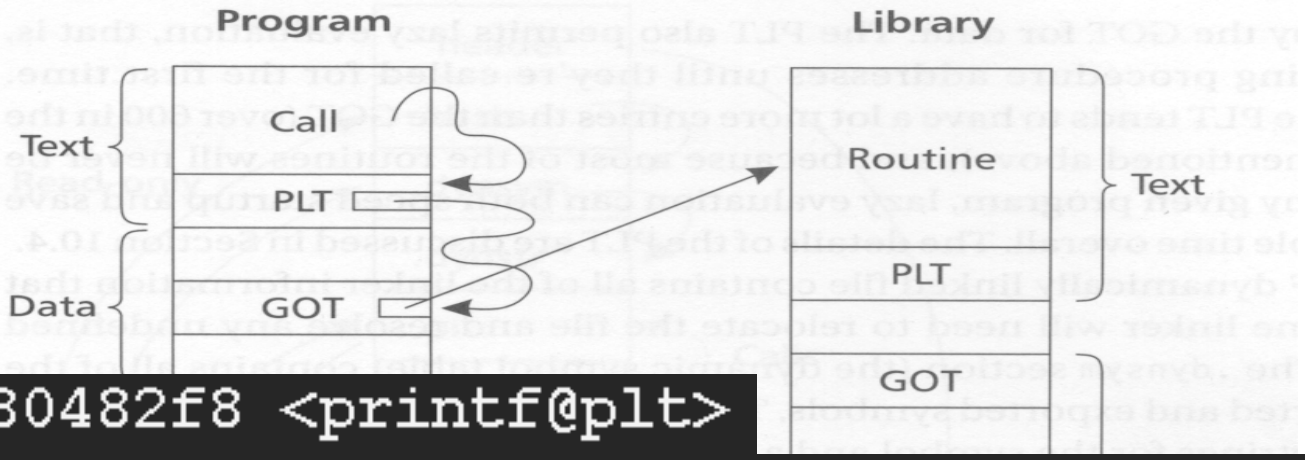
Arguments passed to functions via registers

- If more than 6 integral parameters, then pass rest on stack
- These registers can be used as caller-saved as well

All references to stack frame via stack pointer

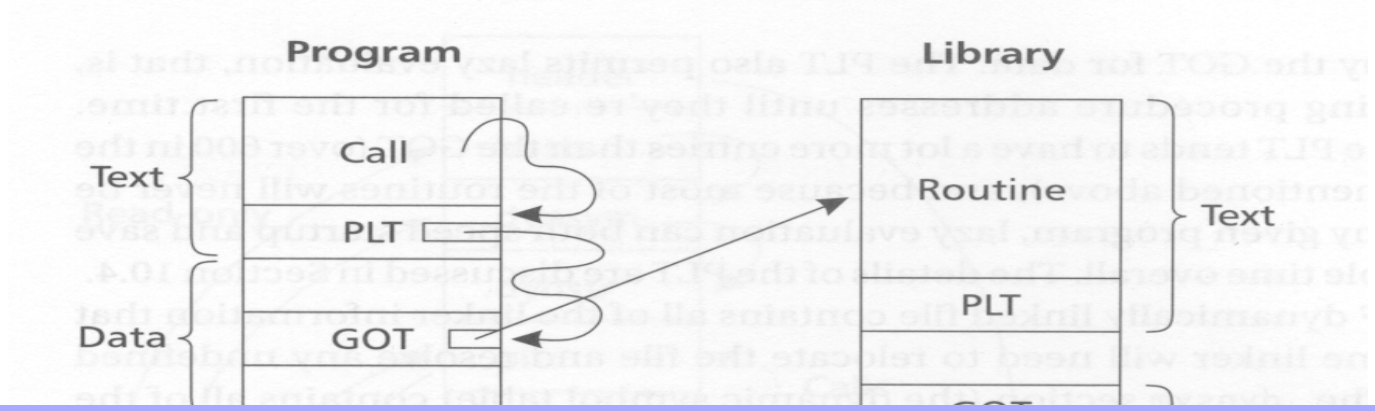
Other Registers

Library Function Call




```
call    80482f8 <printf@plt>
080482f8 <printf@plt>:
 80482f8:    ff 25 08 a0 04 08          jmp     *0x804a008
 80482fe:    68 10 00 00 00             push    $0x10
 8048303:    e9 c0 ff ff ff             jmp     80482c8 <_init+0x30>
(gdb) x/16xw 0x804a008
0x804a008 <_GLOBAL_OFFSET_TABLE_+20>:    0x080482fe
080482c8 <__gmon_start__@plt-0x10>:
 80482c8:    ff 35 f8 9f 04 08          pushl   0x8049ff8
 80482ce:    ff 25 fc 9f 04 08          jmp     *0x8049ffc
 80482d4:    00 00                      add     %al,(%eax)
```

Library Function Call



1. Initially, the .PLT table is not initialize. Thus, it simply got to .gmonstart, which is going to fill in this entry after it is loaded.
2. After executing printf statement, PLT table will be pointing to actual

```
(gdb) x/16xw 0x804a008                                    OLD
0x804a008 <_GLOBAL_OFFSET_TABLE_+20>:    0x080482fe
(gdb) x/16xw 0x804a008                                    NEW
0x804a008 <_GLOBAL_OFFSET_TABLE_+20>:    0xb7ec5b30
```

```
b7e7c000-b7fd8000 r-xp 00000000 08:05 32899      /lib/tls/i686/cmov/libc-2.9.so
b7fd8000-b7fd9000 ---p 0015c000 08:05 32899      /lib/tls/i686/cmov/libc-2.9.so
```

Lazy Binding

Programs that use shared libraries generally contain calls to a lot of functions. In a single run of the program, many of the functions are never called.

To speed program startup, dynamically linked ELF programs use lazy binding of procedure addresses.

Each dynamically bound program has a PLT, with the PLT containing an entry for each nonlocal routine called from the program.

PLT and Lazy Binding

All calls within the program to a particular routine are adjusted to be calls to the actual routine's entry in the PLT.

The first time the program calls a routine, the PLT entry calls the run-time linker to resolve the actually address of the routine.

After that, the PLT entry jumps directly to the actual address.

So, after the first call, the cost of using the PLT is a single indirect jump at a procedure call and nothing at return.

PLT Details

The first entry in the PLT, which is called PLT0, is special code to call the dynamic linker.

At load time, the dynamically linker automatically places two values in the GOT.

- At GOT+4, it puts a code that identifies the particular library.
- At GOT+8, it puts the address of the dynamic linker's symbol resolution routine.

The rest of PLT entries, which we call PLTn, each starts with an indirect jump through a GOT entry that is initially set to point to the push instructions in the PLT entry that follows the jmp.

What we have covered in 2nd half semester?

Linking and Loading

Calling Convention

Virtual Memory

IO/FS

Testing, Debugging, and Profiling

Processes

Threads

Network, Cloud Computing and Big Data

What Is Virtual Memory?

If you think it's there, and it's there…it's *real*.

If you think it's there, and it's not there…it's *imaginary*.

If you think it's not there, and it's not there…it's *nonexistent*.

Virtual memory is imaginary memory: it gives you the illusion of a memory arrangement that's not physically there.

In computing, an address space defines a range of discrete addresses.

Motivations for Virtual Memory

Use physical DRAM as cache for the disk

- Address space of a process can exceed physical memory size
- Sum of address spaces of multiple processes can exceed physical memory (more common modern case)

Simplify memory management

- Multiple processes resident in main memory
  - Each with its own address space
- Only "active" code and data is actually in memory
  - Allocate more memory to process as needed

Provide protection

- One process can't interfere with another
  - Because they operate in different address spaces
- User process cannot access privileged information
  - Different sections of address spaces have different permissions

Motivation #1: DRAM as "Cache" for Disk

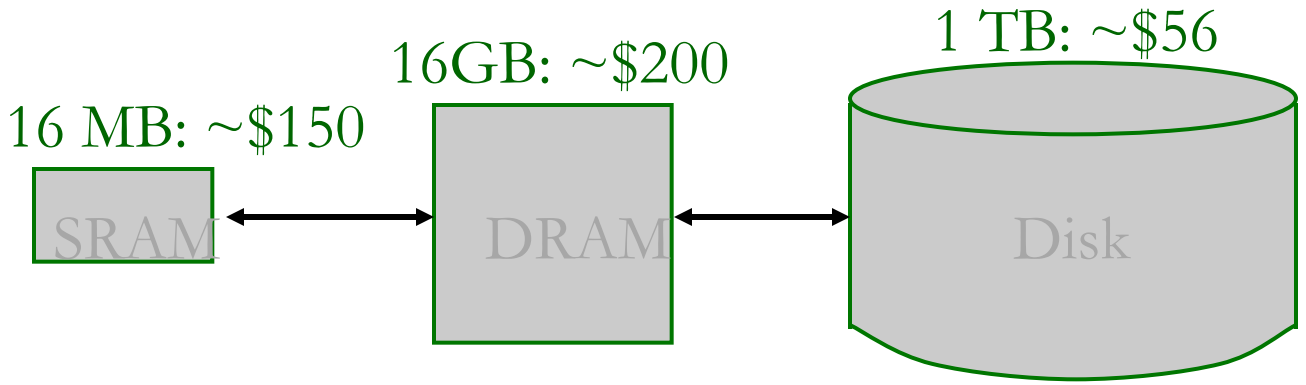
Full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

Disk storage is ~500X cheaper than DRAM storage

- 1 TB of DRAM: ~ $128*200
- 1 TB of disk: ~ $56

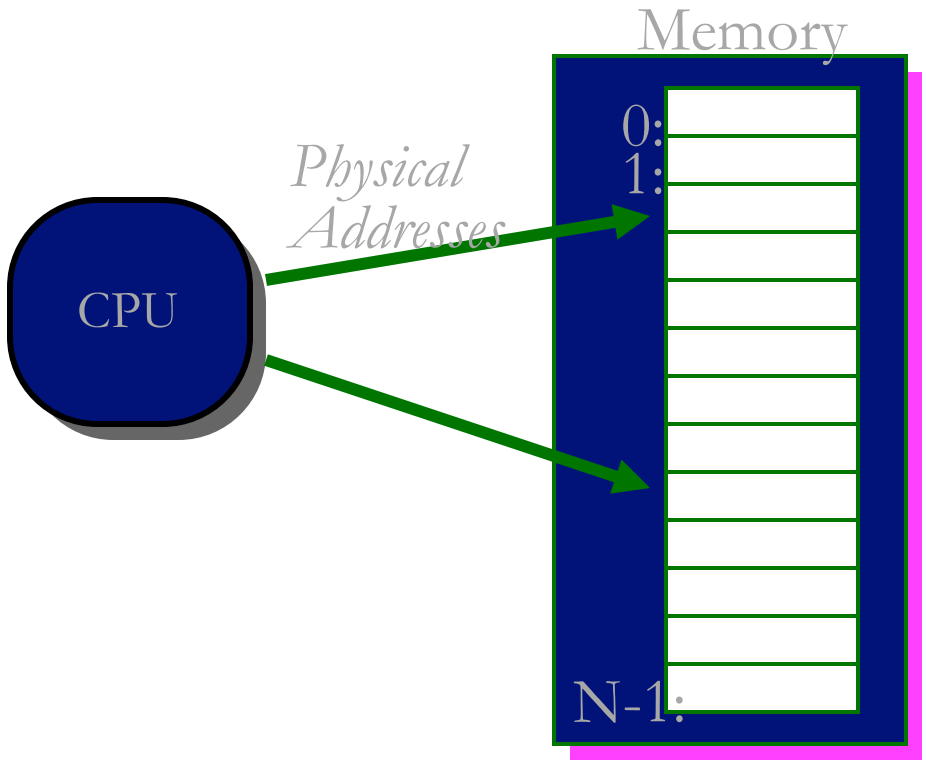
To get cost-effective access to large amounts of data, bulk of data must be stored on disk



A System with Physical Memory Only

Examples:

- **Most Cray machines, early PCs, nearly all embedded systems, etc.**

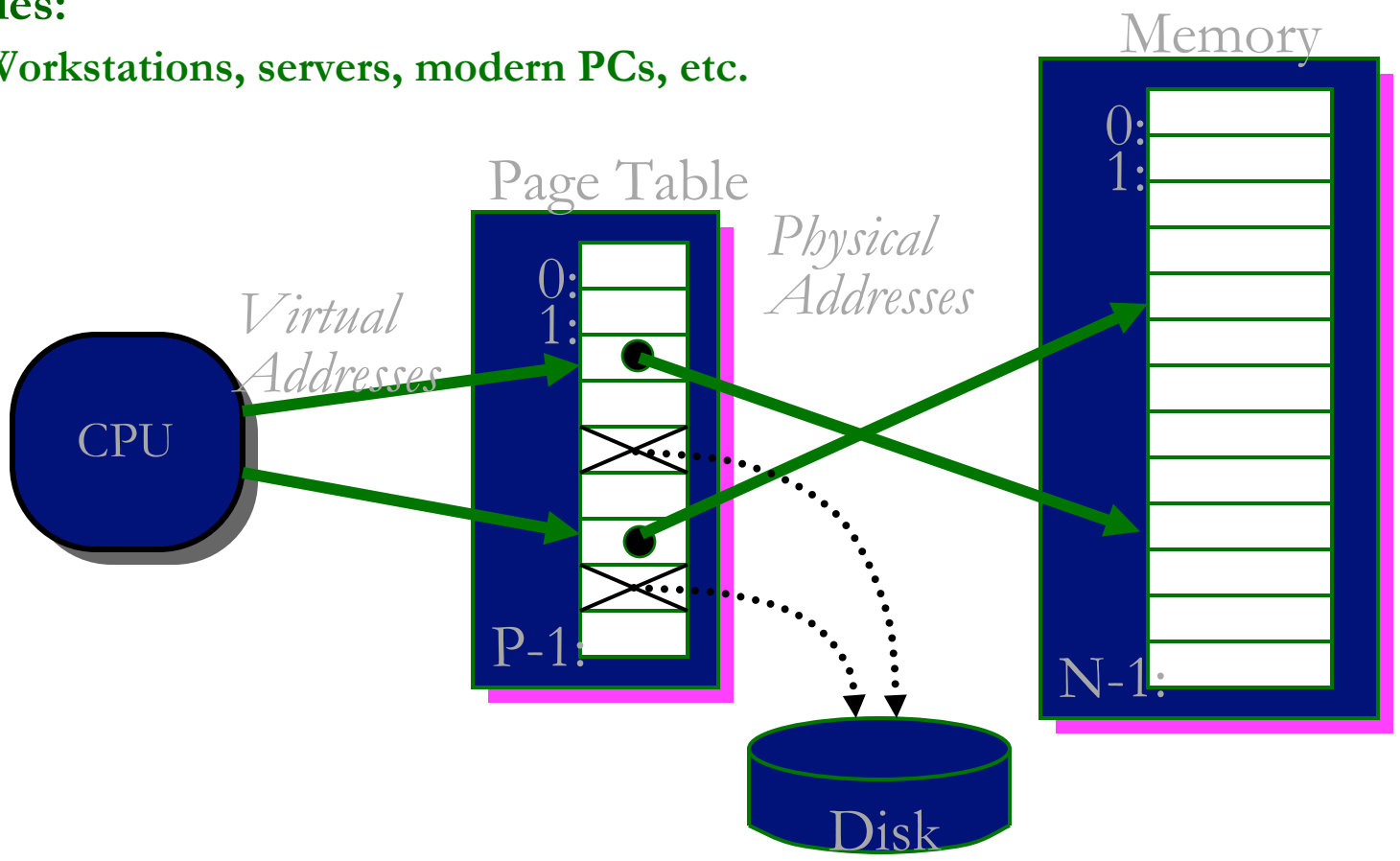


- Addresses generated by the CPU correspond directly to bytes in physical memory

A System with Virtual Memory

Examples:

- **Workstations, servers, modern PCs, etc.**



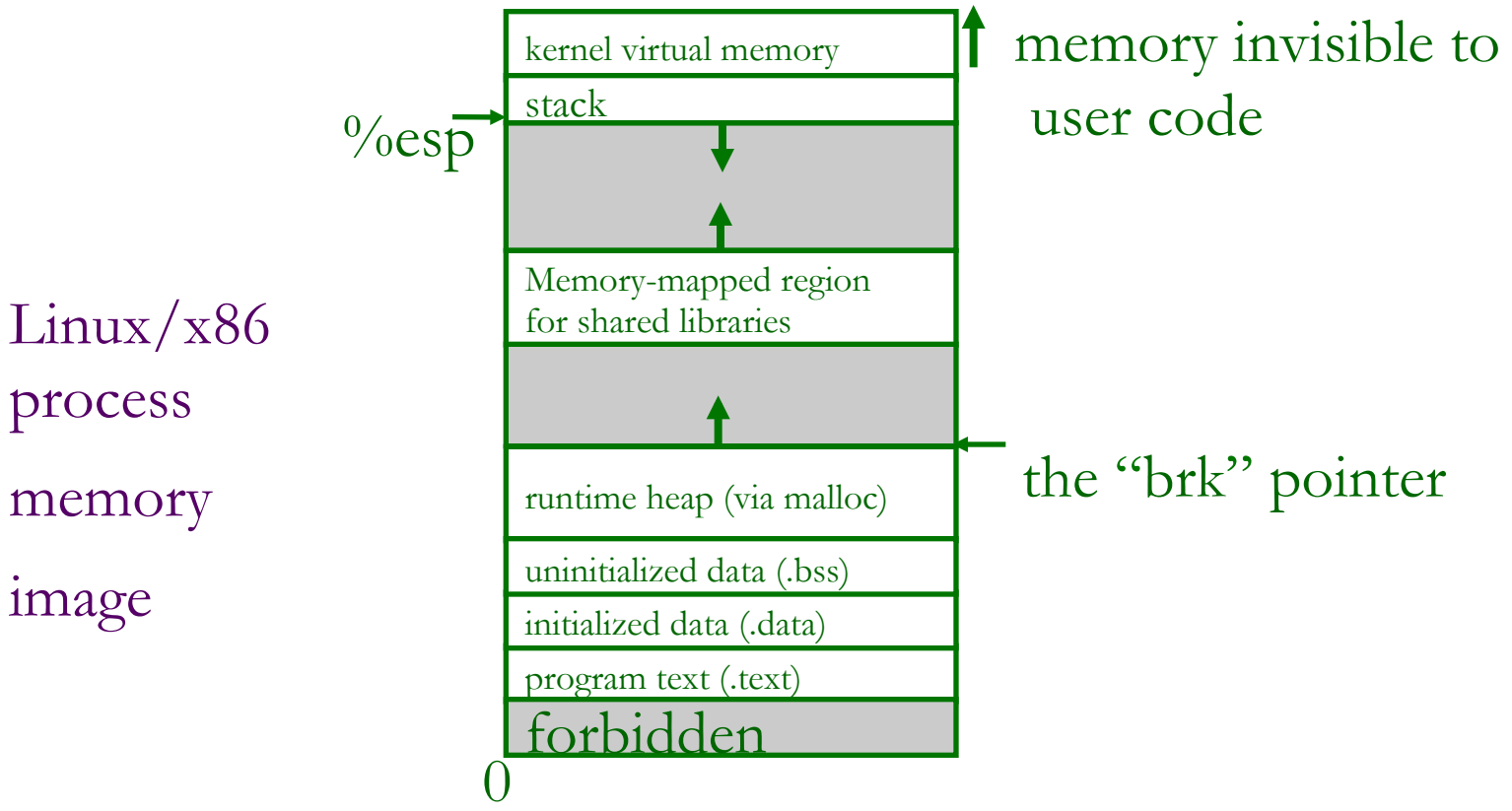
- Address Translation: Hardware converts virtual addresses to physical ones via OS-managed lookup table (page table)

Motivation #2: Memory Mgmt

Multiple processes can reside in physical memory.

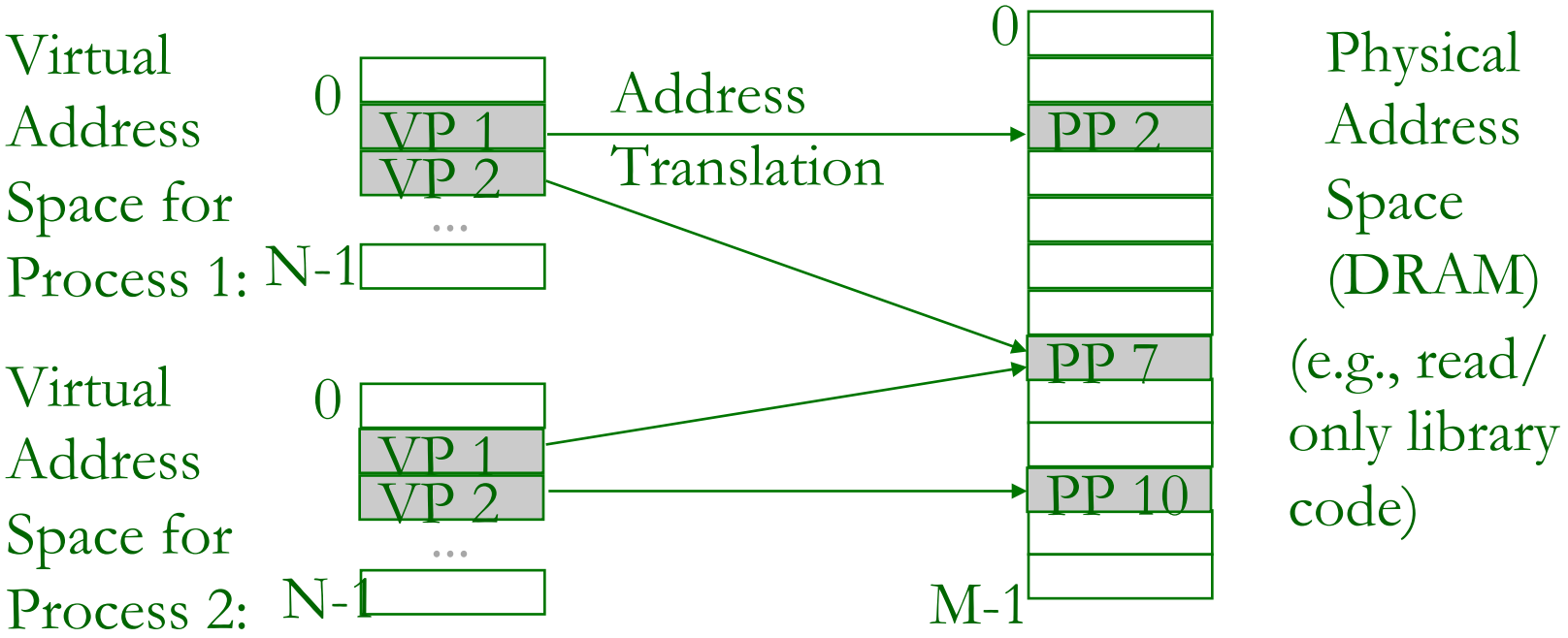
How do we resolve address conflicts?

- **What if two processes access something at same address?**



Solution: Separate Virtual Address Spaces

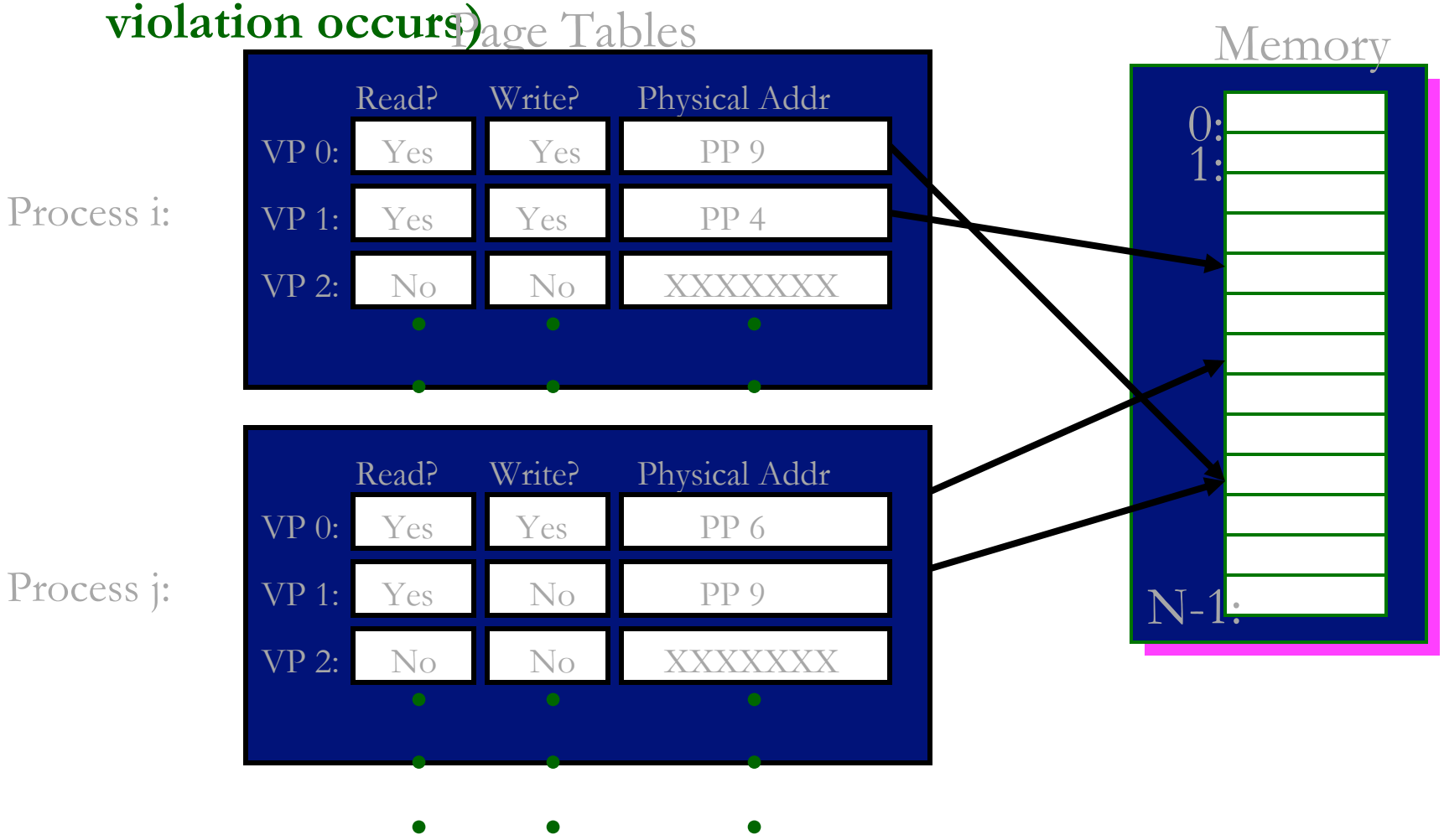
- Virtual and physical address spaces divided into equal-sized blocks
  - Blocks are called "pages" (both virtual and physical, 4096Bytes)
- Each process has its own virtual address space
  - Operating system controls how virtual pages are assigned to physical memory



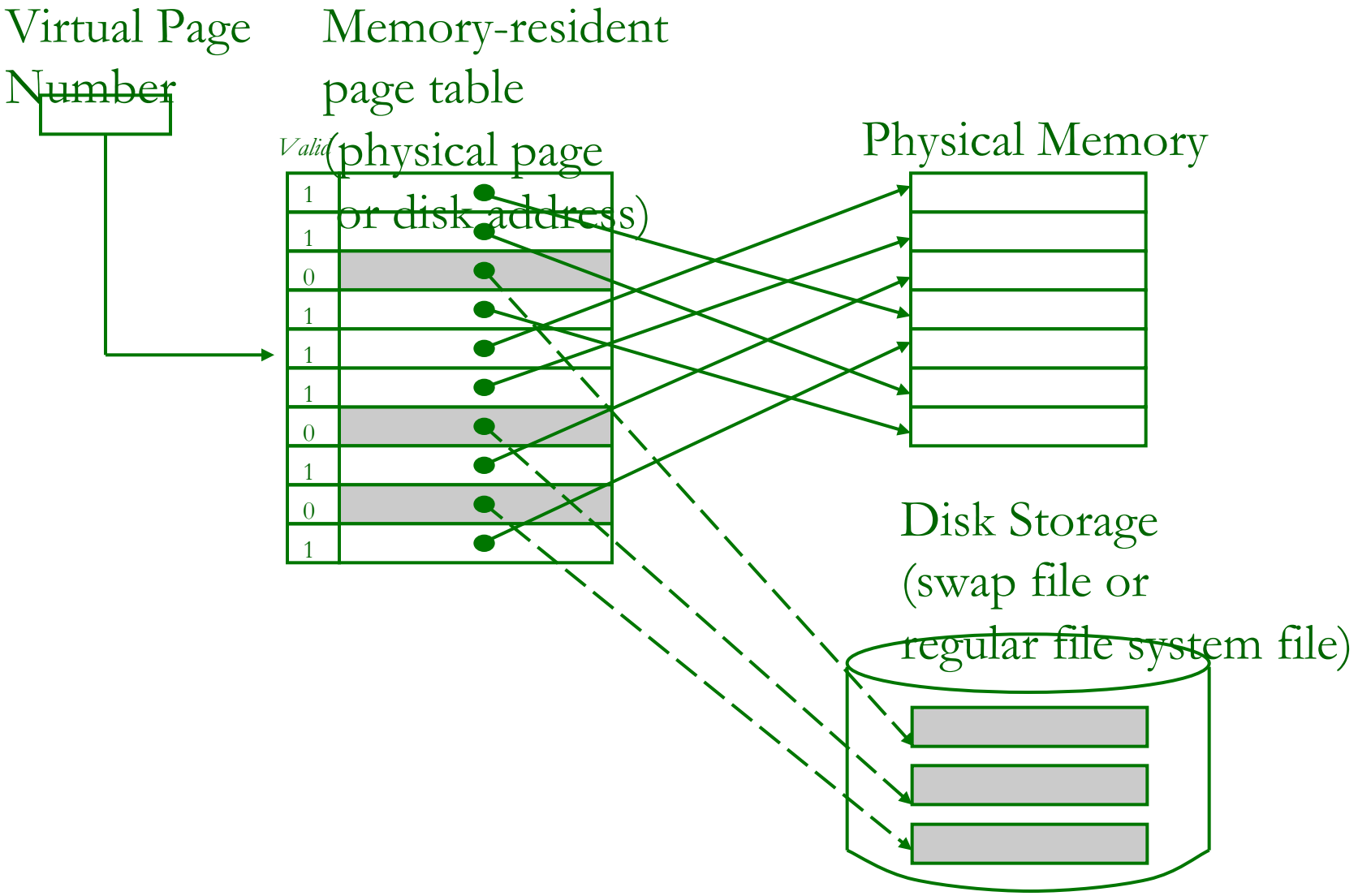
Motivation #3: Protection

Page table entry contains access-rights information

- **Hardware enforces this protection (trap into OS if violation occurs)**



Page Table Is Used for Address Translation



Memory allocation (using mmap/brk)

```
#include <stdio.h>
#include <stdlib.h>

int main() {

  int * ptr = malloc(4);

  *ptr = 1;

  free(ptr);
}
```

```
08048000-08049000 r-xp   test
08049000-0804a000 r—p  test
0804a000-0804b000 rw-p  test
b7e7b000-b7e7c000 rw-p  0
b7e7c000-b7fd8000 r-xp    libc-2.9.so
b7fd8000-b7fd9000 ---p    libc-2.9.so
b7fd9000-b7fdb000 r--p    libc-2.9.so
b7fdb000-b7fdc000 rw-p    libc-2.9.so
b7fdc000-b7fe1000 rw-p    0
b7fe1000-b7fe2000 r-xp    0      [vdso]
b7fe2000-b7ffe000 r-xp     ld-2.9.so
b7ffe000-b7fff000 r—p      ld-2.9.so
b7fff000-b8000000 rw-p     ld-2.9.so
bffeb000-c0000000 rw-p    [stack]
```

Currently, no heap space at all because we didn't use any heap

Memory allocation

```
#include <stdio.h>
#include <stdlib.h>

int main() {

  int * ptr = malloc(4);

  *ptr = 1;

  free(ptr);
}
```

```
08048000-08049000 r-xp   test
08049000-0804a000 r—p  test
0804a000-0804b000 rw-p  test
0804b000-0806c000 rw-p  [heap]
b7e7b000-b7e7c000 rw-p  0
b7e7c000-b7fd8000 r-xp   libc-2.9.so
b7fd8000-b7fd9000 ---p    libc-2.9.so
b7fd9000-b7fdb000 r--p    libc-2.9.so
b7fdb000-b7fdc000 rw-p   libc-2.9.so
b7fdc000-b7fe1000 rw-p   0
b7fe1000-b7fe2000 r-xp    0      [vdso]
b7fe2000-b7ffe000 r-xp     ld-2.9.so
b7ffe000-b7fff000 r—p      ld-2.9.so
b7fff000-b8000000 rw-p    ld-2.9.so
bffeb000-c0000000 rw-p   [stack]
```

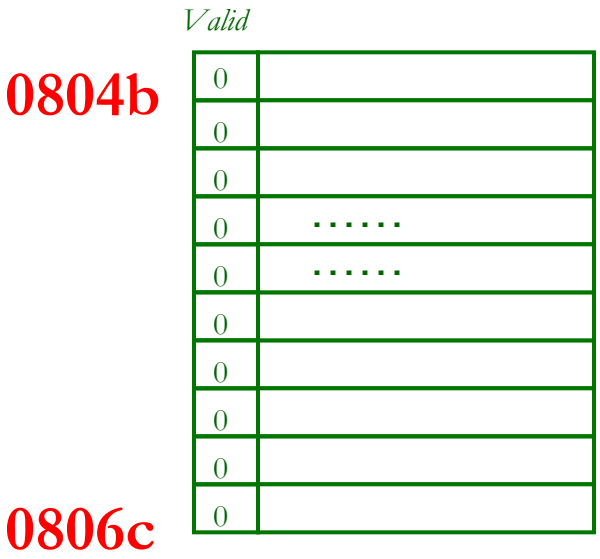
Now, the heap is allocated from the kernel, which means the virtual address from 0x0804b000 to 0x0806c000 (total 33K) are usable.

ptr is actually 0x804b008.

Memory Mapping (mmap or brk)

```
#include <stdio.h>
#include <stdlib.h>

int main() {

  int * ptr = malloc(4);

  *ptr = 1;

  free(ptr);
}
```

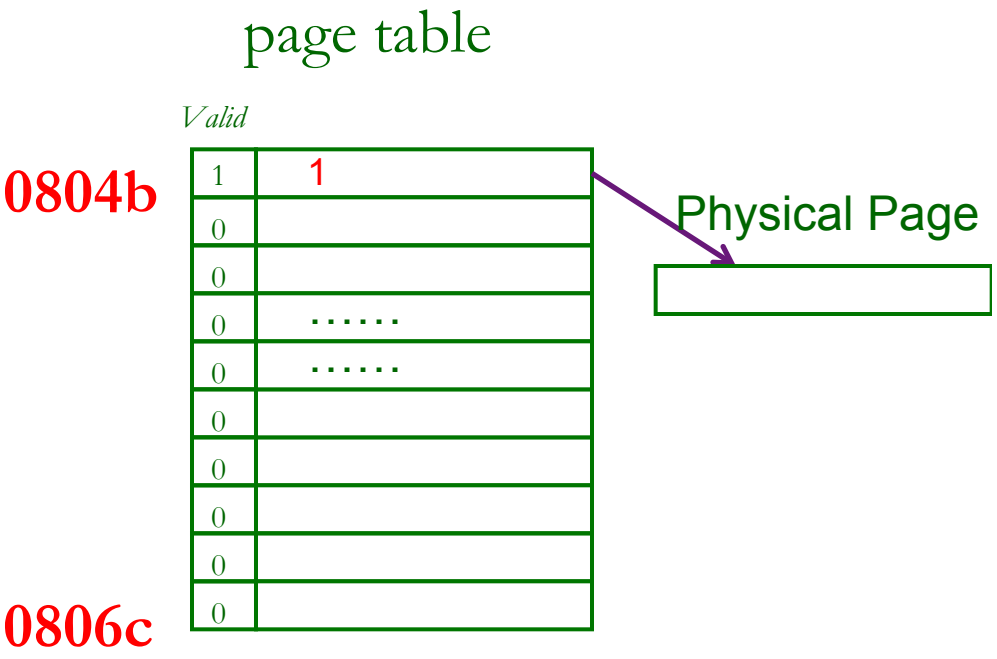
page table



0804b000-0806c000 rw-p [heap]

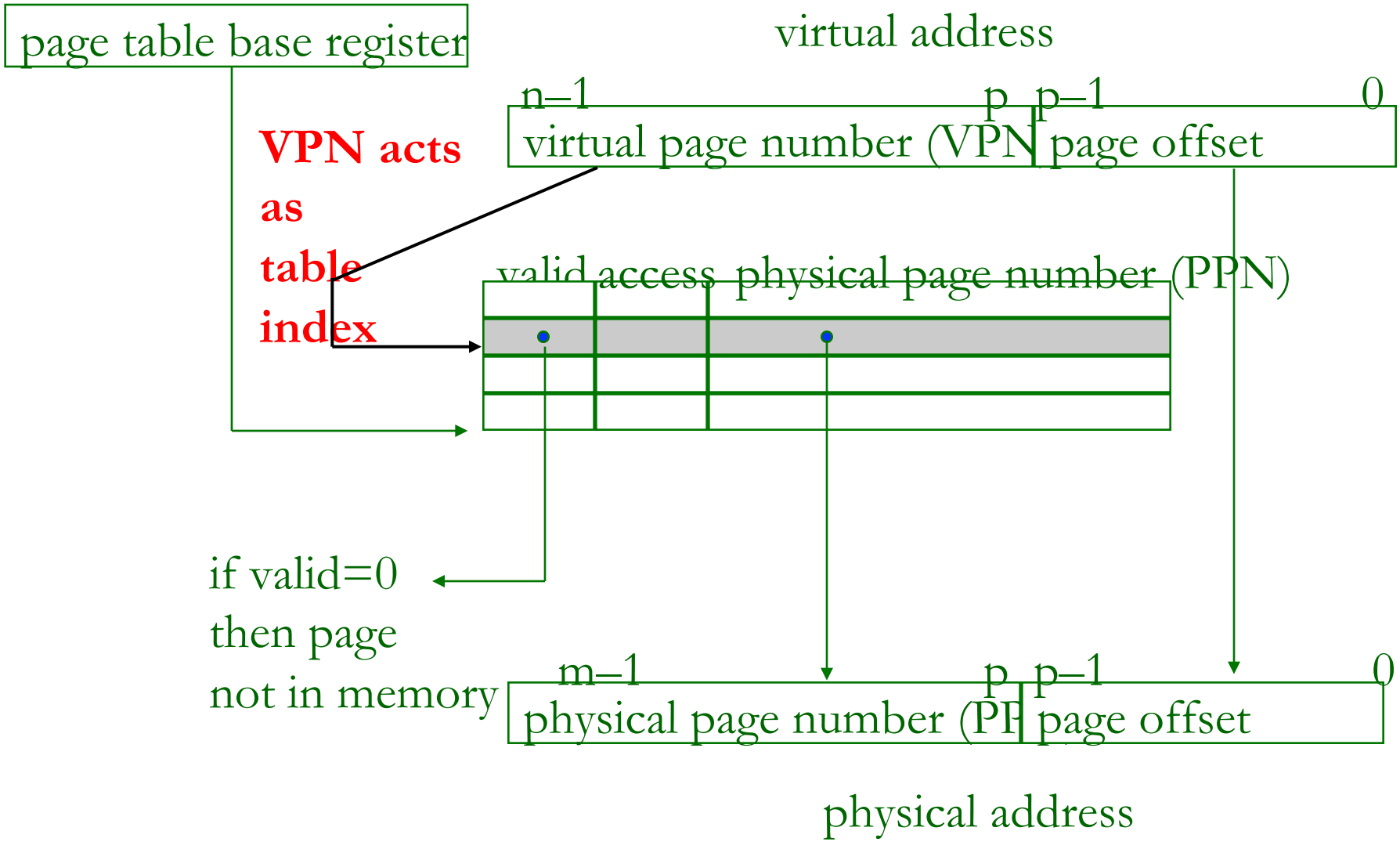
Memory Mapping (mmap or brk)

```
#include <stdio.h>
#include <stdlib.h>

int main() {

  int * ptr = malloc(4);

  *ptr = 1;

  free(ptr);
}
```

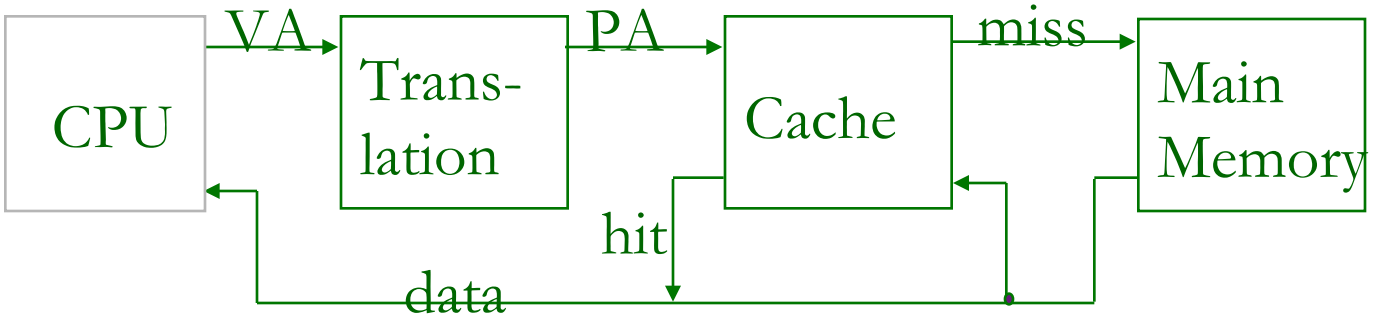


0804b000-0806c000 rw-p [heap]

Address Translation via Page Table



Integrating VM and Cache



Most caches "physically addressed"

- Accessed by physical addresses
- Allows multiple processes to have blocks in cache at same time else context switch == cache flush
- Allows multiple processes to share pages
- Cache doesn't need to be concerned with protection issues
  - Access rights checked as part of address translation

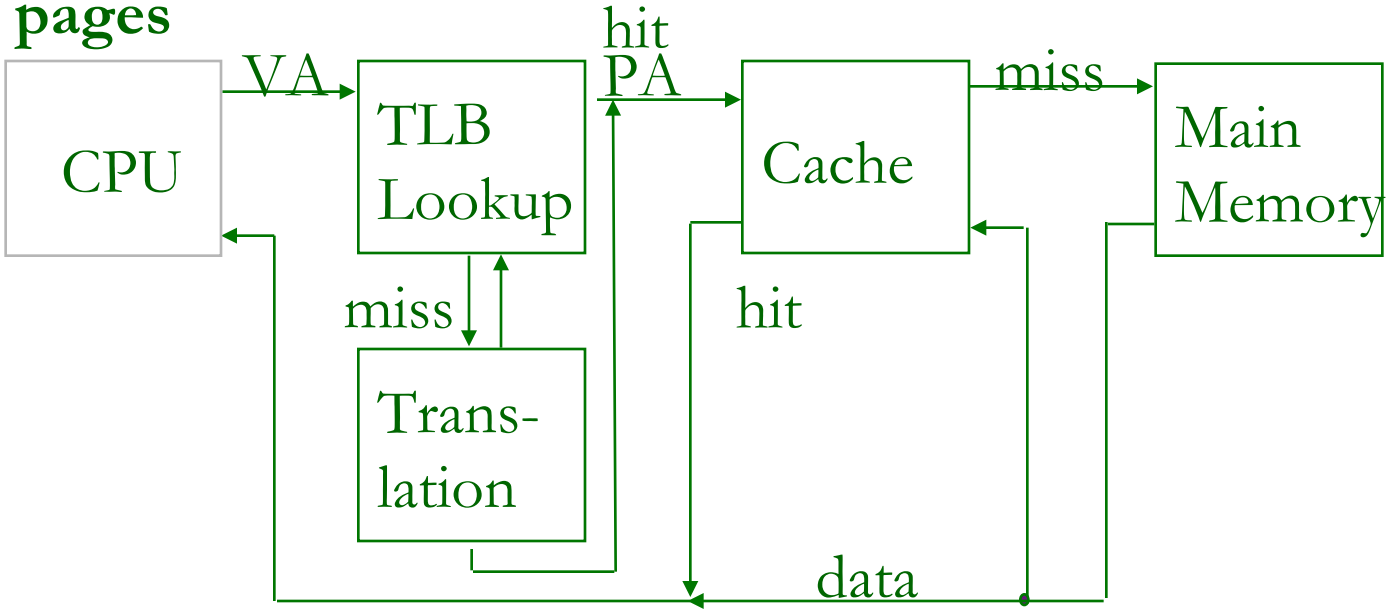
Perform address translation before cache lookup

- Could involve memory access itself (to get PTE)
- So page table entries can also be cached

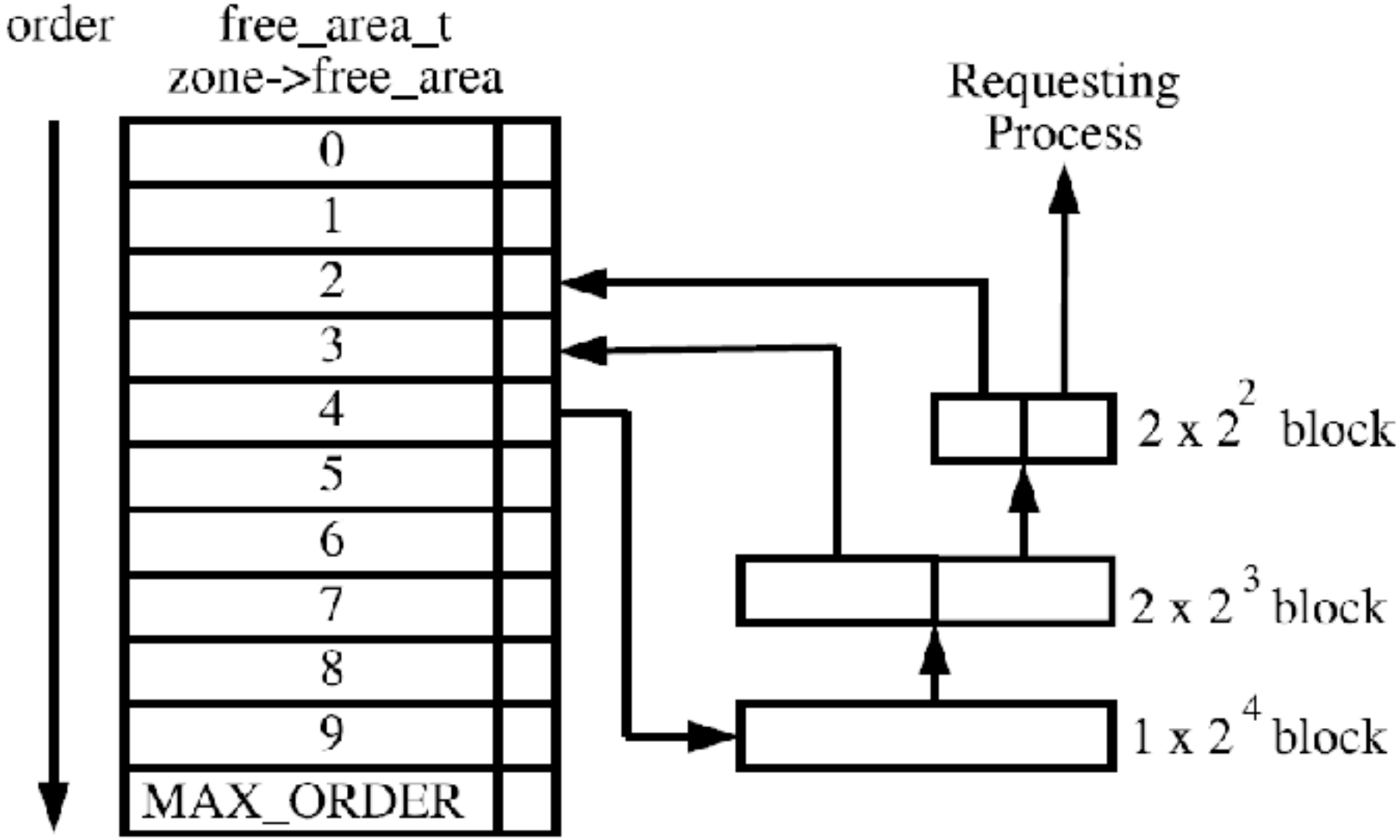
Speeding up Translation With a TLB

"Translation Lookaside Buffer" (TLB)

- **Small hardware cache in MMU**
- **Maps virtual page numbers to physical page numbers**
- **Contains complete page table entries for small number of pages**



Page Allocation of Buddy Allocator



Buddy Allocation

Example: Need to allocate 65 contiguous page frames.

- Look in list of free 128-page-frame blocks.
- If free block exists, allocate it, else look in next highest order list (here, 256-page-frame blocks).
- If first free block is in 256-page-frame list, allocate a 128-page-frame block and put remaining 128-page-frame block in lower order list.
- If first free block is in 512-page-frame list, allocate a 128-page-frame block and split remaining 384 page frames into 2 blocks of 256 and 128 page frames. These blocks are allocated to the corresponding free lists.

Question: What is the worst-case *internal* fragmentation?

What we have covered in 2nd half semester?

Linking and Loading

Calling Convention

Virtual Memory

IO/FS

Testing, Debugging, and Profiling

Processes

Threads

Network, Cloud Computing and Big Data

IO/FS

Unix I/O

Metadata, sharing, and redirection

Stream I/O

Special Files of Linux

Conclusions and examples

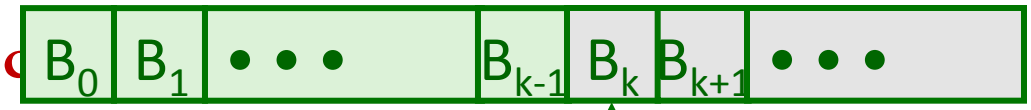
Unix I/O

Key Features

- Elegant mapping of files to devices allows kernel to export simple interface called Unix I/O
- Important idea:
  All input and output is handled in a consistent and uniform way

Basic Unix I/O operations (system calls):

- Opening and closing files
  - `open()` and `close()`
- Reading and writing a file
  - `read()` and `write()`
- Changing the c



  - indicates next offset into file to read or write
  - `lseek()`

Dealing with Short Counts

Short counts can occur in these situations:

- Encountering (end-of-file) EOF on reads
- Reading text lines from a terminal
- Reading and writing network sockets or Unix pipes

Short counts never occur in these situations:

- Reading from disk files (except for EOF)
- Writing to disk files

One way to deal with short counts in your code:

- Use the RIO (Robust I/O) package from your textbook's `csapp.c` file (Appendix B)

File Metadata

Metadata **is data about data, in this case file data**

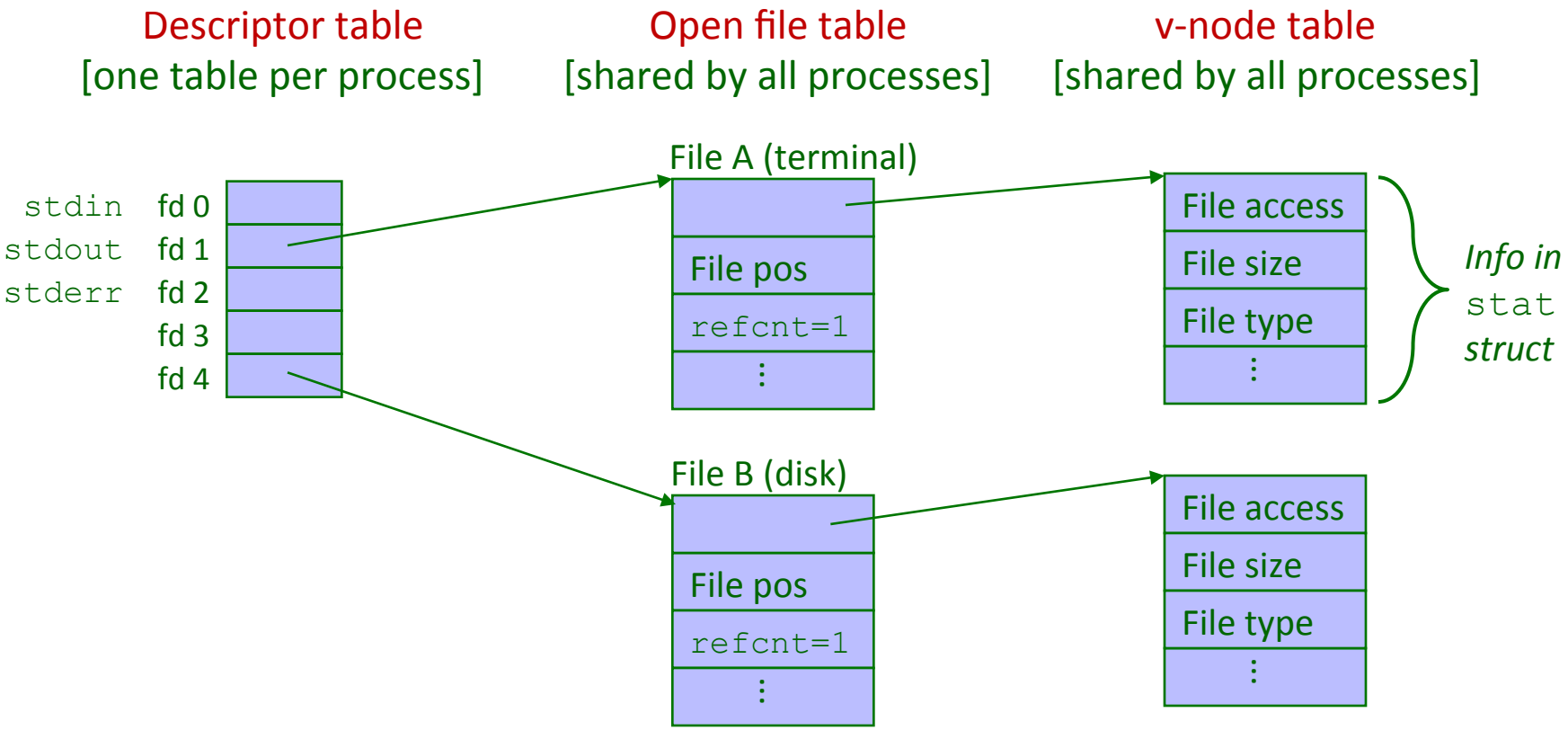
Per-file metadata maintained by kernel

- **accessed by users with the `stat` and `fstat` functions**

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

How the Unix Kernel Represents Open Files

Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



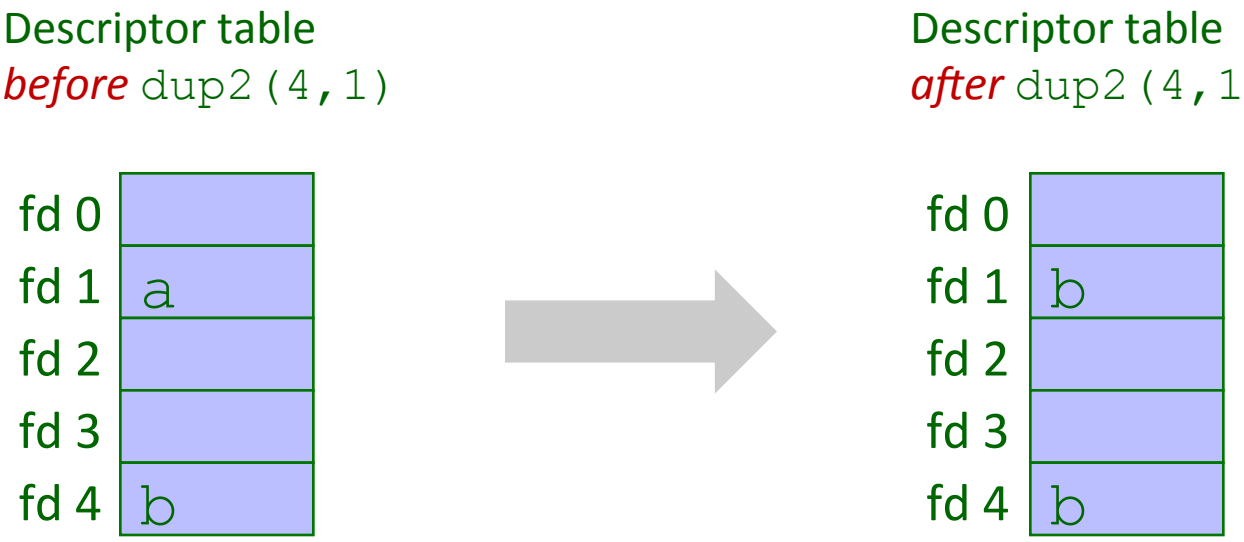
I/O Redirection

Question: How does a shell implement I/O redirection?

```
unix> ls > foo.txt
```

Answer: By calling the `dup2(oldfd, newfd)` function

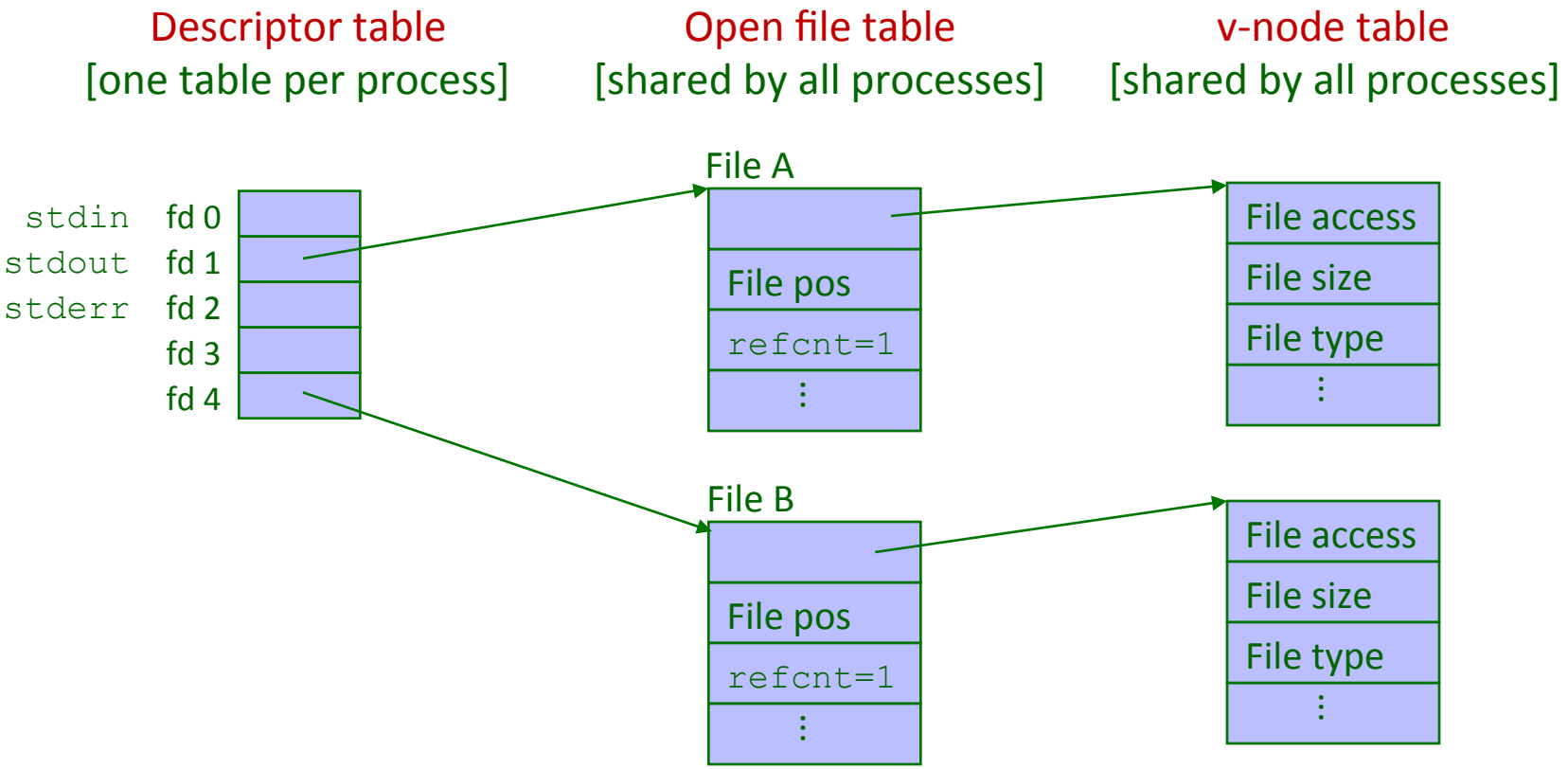
- **Copies (per-process) descriptor table entry `oldfd` to entry `newfd`**



I/O Redirection Example

Step #1: open file to which stdout should be redirected

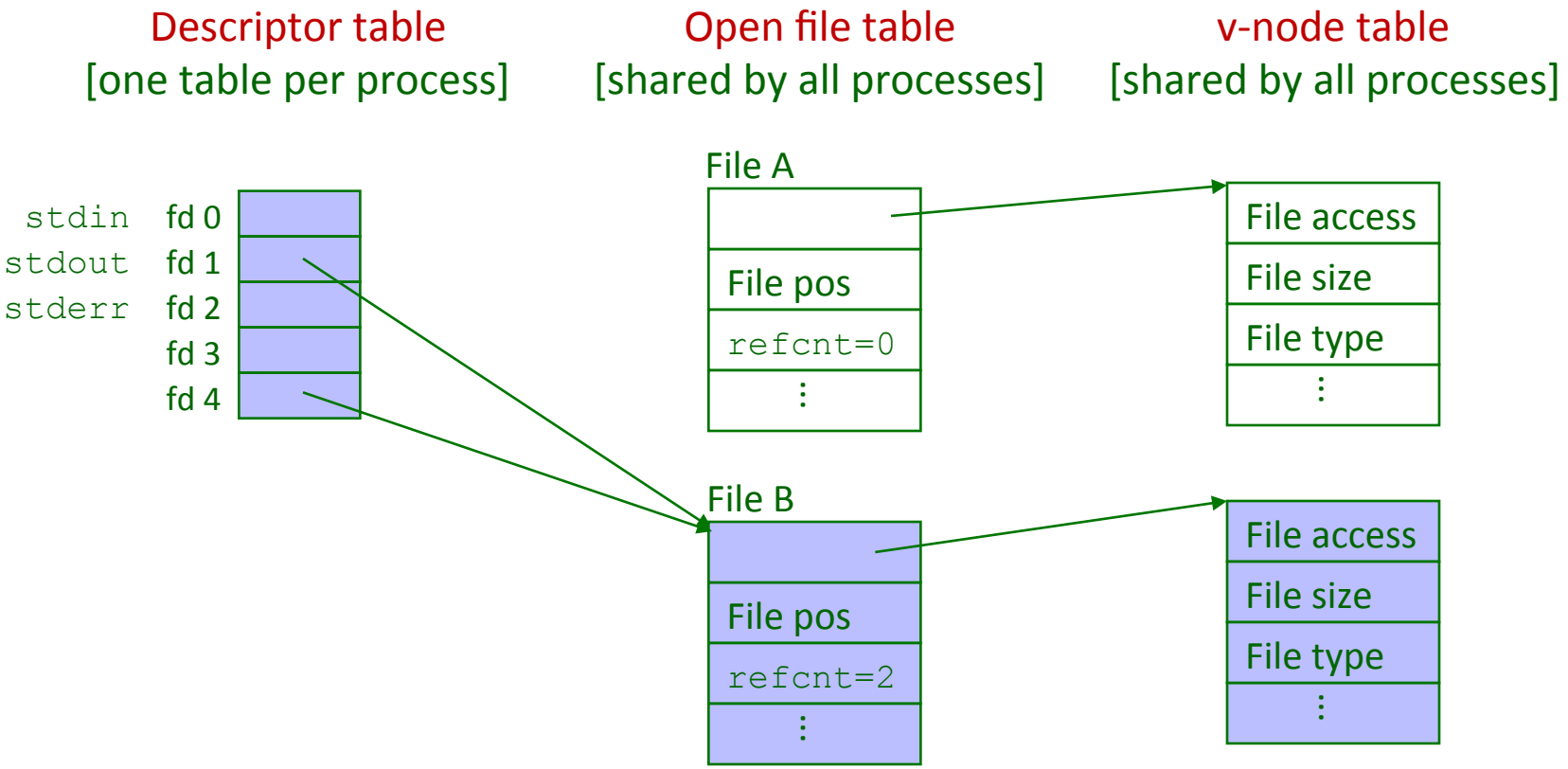
- **Happens in child executing shell code, before `exec`**



I/O Redirection Example (cont.)

Step #2: call `dup2(3,1)`

- cause fd=1 (stdout) to refer to disk file pointed at by fd=3

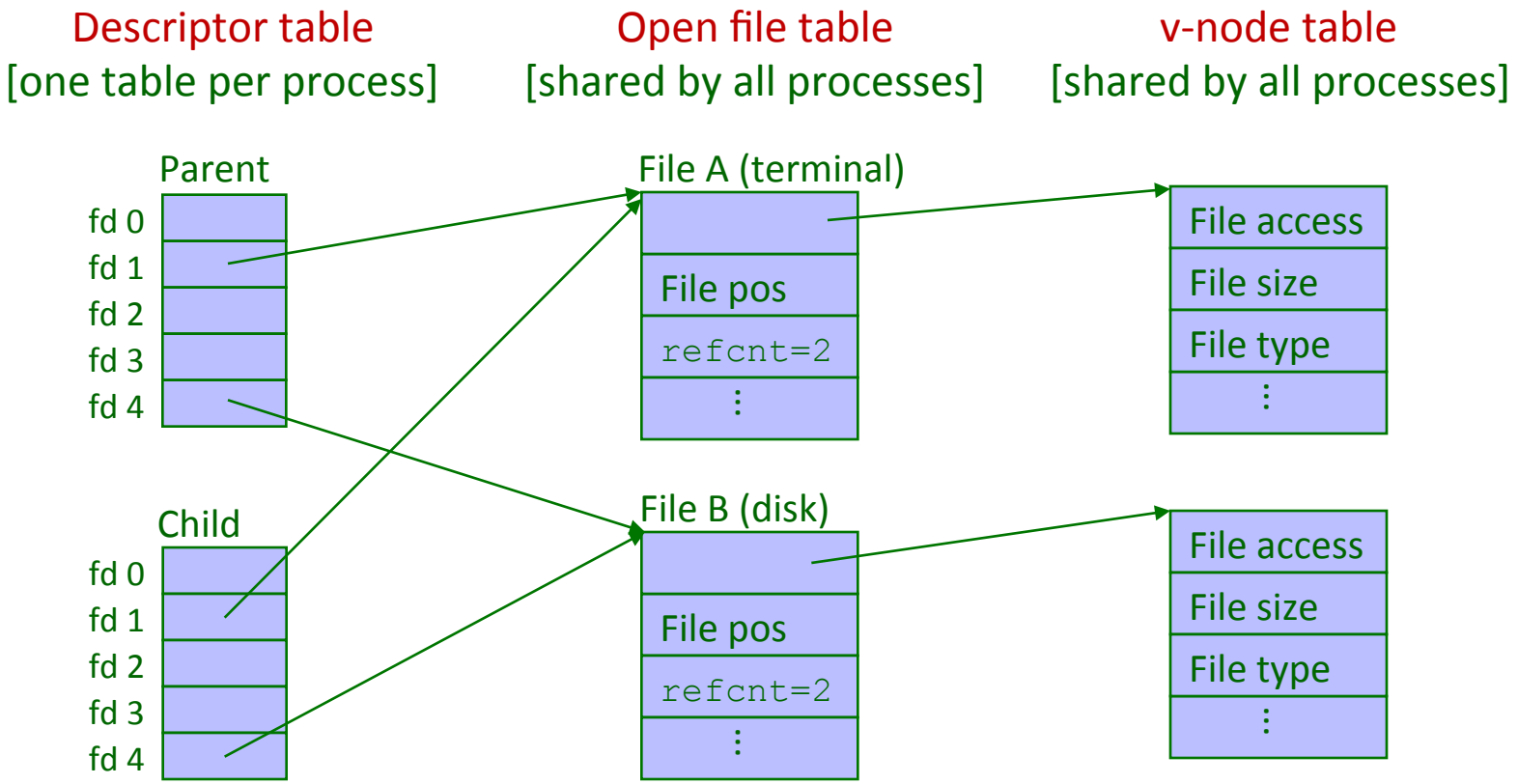


How Processes Share Files: Fork()

A child process inherits its parent's open files

After **fork():**

- Child's table same as parent's, and +1 to each refcnt



Stream I/O Functions

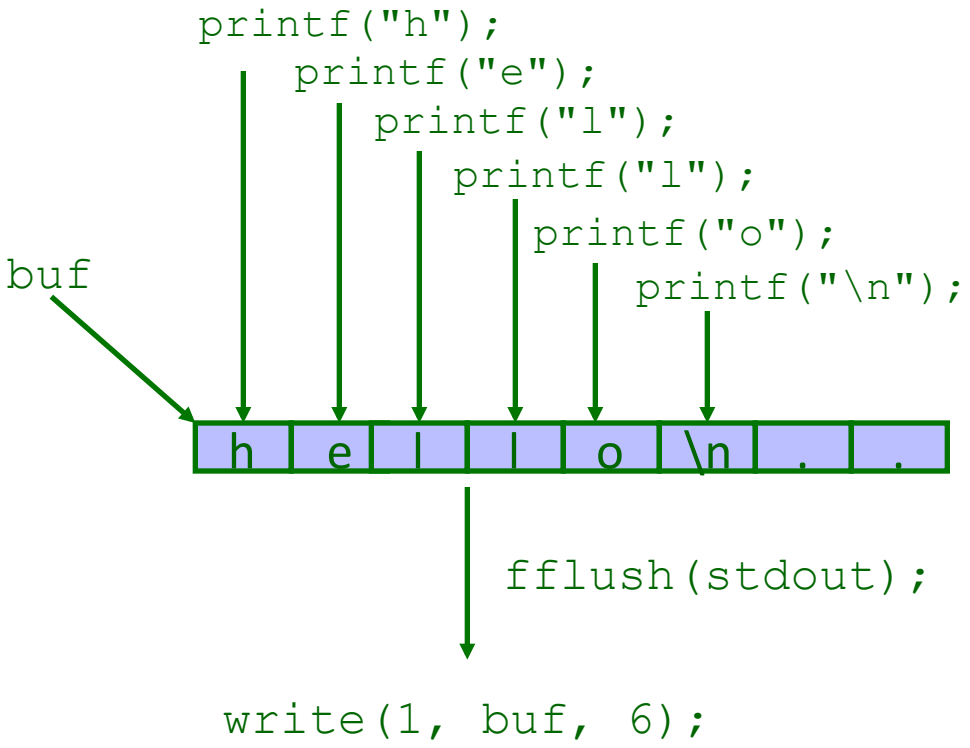
The C standard library (`libc.so`) contains a collection of higher-level *stream I/O* functions

Examples of standard I/O functions:

- Opening and closing files (`fopen` and `fclose`)
- Reading and writing bytes (`fread` and `fwrite`)
- Reading and writing text lines (`fgets` and `fputs`)
- Formatted reading and writing (`fscanf` and `fprintf`)

Buffering in Standard I/O

Standard I/O functions use buffered I/O



Buffer flushed to output fd on "\n" or `fflush()` call

Pros and Cons of Unix I/O

Pros

- **Unix I/O is the most general and lowest overhead form of I/O.**
  - *All other I/O packages are implemented using Unix I/O functions.*
- **Unix I/O provides functions for accessing file metadata.**
- **Unix I/O functions are async-signal-safe and can be used safely in signal handlers.**

Cons

- **Dealing with short counts is tricky and error prone.**
- **Efficient reading of text lines requires some form of buffering, also tricky and error prone.**
- **Both of these issues are addressed by the standard I/O and RIO packages.**

Pros and Cons of Standard I/O

Pros:

- Buffering increases efficiency by decreasing the number of `read` and `write` system calls
- Short counts are handled automatically

Cons:

- Provides no function for accessing file metadata
- Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers.
- Standard I/O is not appropriate for input and output on network sockets

What we have covered in 2nd half semester?

Linking and Loading

Calling Convention

Virtual Memory

IO/FS

Testing, Debugging, and Profiling

Processes

Threads

Network, Cloud Computing and Big Data

Testing, debugging and Profiling

Testing: "running a program with the intent of finding bugs"

Debugging: once errors are found, "finding the exact nature of a suspected programming error and fixing it"

Profiling: measures the space (memory) or time complexity, the usage of particular instructions, or the frequency and duration of function calls

Compile for debugging

When compiling your program, add the –g flag to the command line:

- `gcc -g -o prog prog.c`

This adds extra symbol information, so the debugger knows how you called the variables in your source, can show you the source code and which line will be executed next

Compile for debugging

When compiling your program, add the –g flag to the command line:

- `gcc -g -o prog prog.c`

This adds extra symbol information, so the debugger knows how you called the variables in your source, can show you the source code and which line will be executed next

Inspecting a corefile

You can look at any program that has crashed (and produced a corefile) to see any of its state at the time of the crash

Load executable and corefile into the debugger

Use GDB's `backtrace` (`bt`) command to see the call stack

Breakpoints & Watchpoints

A place where execution pauses, waits for a user command

Can break at a function, a line number, or on a certain condition

- **`break`** or **`b`** (set a breakpoint)
  - **`break main`**
  - **`break 10`**

`watch` ***`expr:`***

- stops whenever the value of the expression changes

```
(gdb) watch foo
(gdb) watch foo mask 0xffff00ff
(gdb) watch *0xdeadbeef mask 0xffffff00
```

More can be seen at
https://sourceware.org/gdb/onlinedocs/gdb/Set-Watchpoints.html

The mask specifies that some bits of an address (the bits which are reset in the mask) should be ignored

Debugging an already-running process

From inside GDB:

attach process-id

/ To get the process ID use the UNIX command ps /

From outside GDB:

gdb my_prog process-id

- The first thing GDB does after arranging to debug the specified process is to stop it.
- detach – detaches the currently attached process from the GDB control. A detached process continues its own execution.

Today

Overview

gprof

Oprofiler

Valgrind

Strace

Performance Tools Overview

Timing mechanisms

- **Stopwatch : UNIX time utility**

Optimizing compiler (easy way)

System load monitors

- **vmstat , iostat , Vtune Counter**

Software profiler

- **Gprof, VTune, Visual C++ Profiler** Predator

Memory debugger/profiler

- **Valgrind , IBM Purify, Parasoft Insure++** DoubleTake

Profilers

Profiler may show time elapsed in each function and its descendants

- number of calls , call-graph (some)

Profilers use either instrumentation or sampling to identify performance issues

What we have covered in 2nd half semester?

Linking and Loading

Calling Convention

Virtual Memory

IO/FS

Testing, Debugging, and Profiling

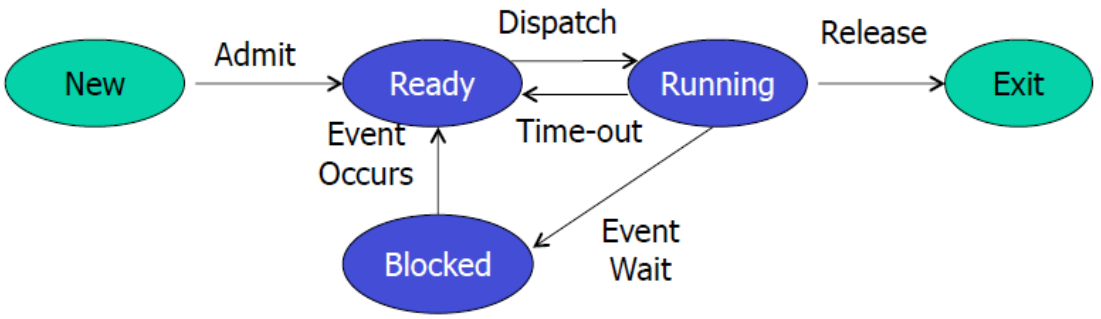
Processes

Threads

Network, Cloud Computing and Big Data

Process Concept

- Process – a program in execution
- Textbook uses the terms job and process almost interchangeably
- A process includes:
  - program counter
  - stack
  - data section
- Process life-cycle



The Process

Multiple parts

- The program code, also called text section
- Current activity including program counter, processor registers
- Stack containing temporary data
  - Function parameters, return addresses, local variables
- Data section containing global variables
- Heap containing memory dynamically allocated during run time

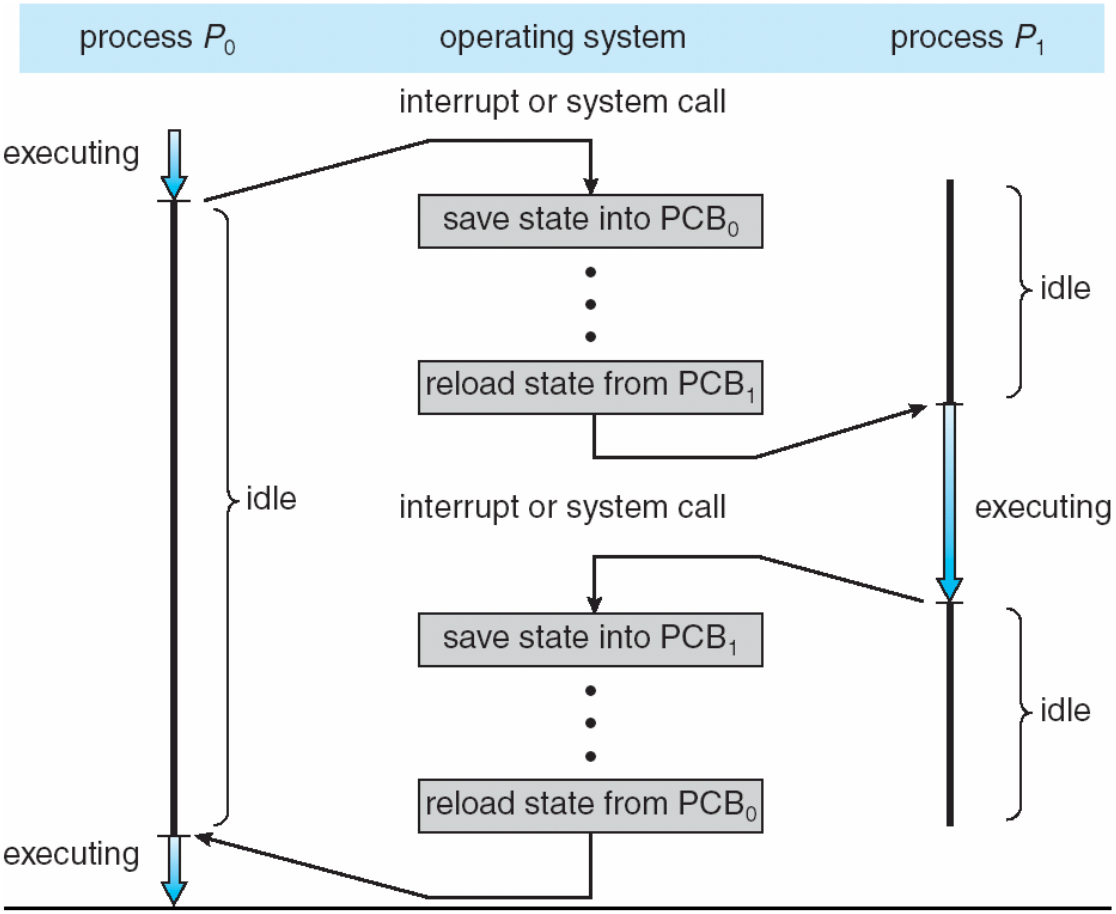
Program is passive entity, process is active

- Program becomes process when executable file loaded into memory

One program can be several processes

- Consider multiple users executing the same program

CPU Switch From Process to Process



Context Switch

When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.

Context of a process represented in the PCB

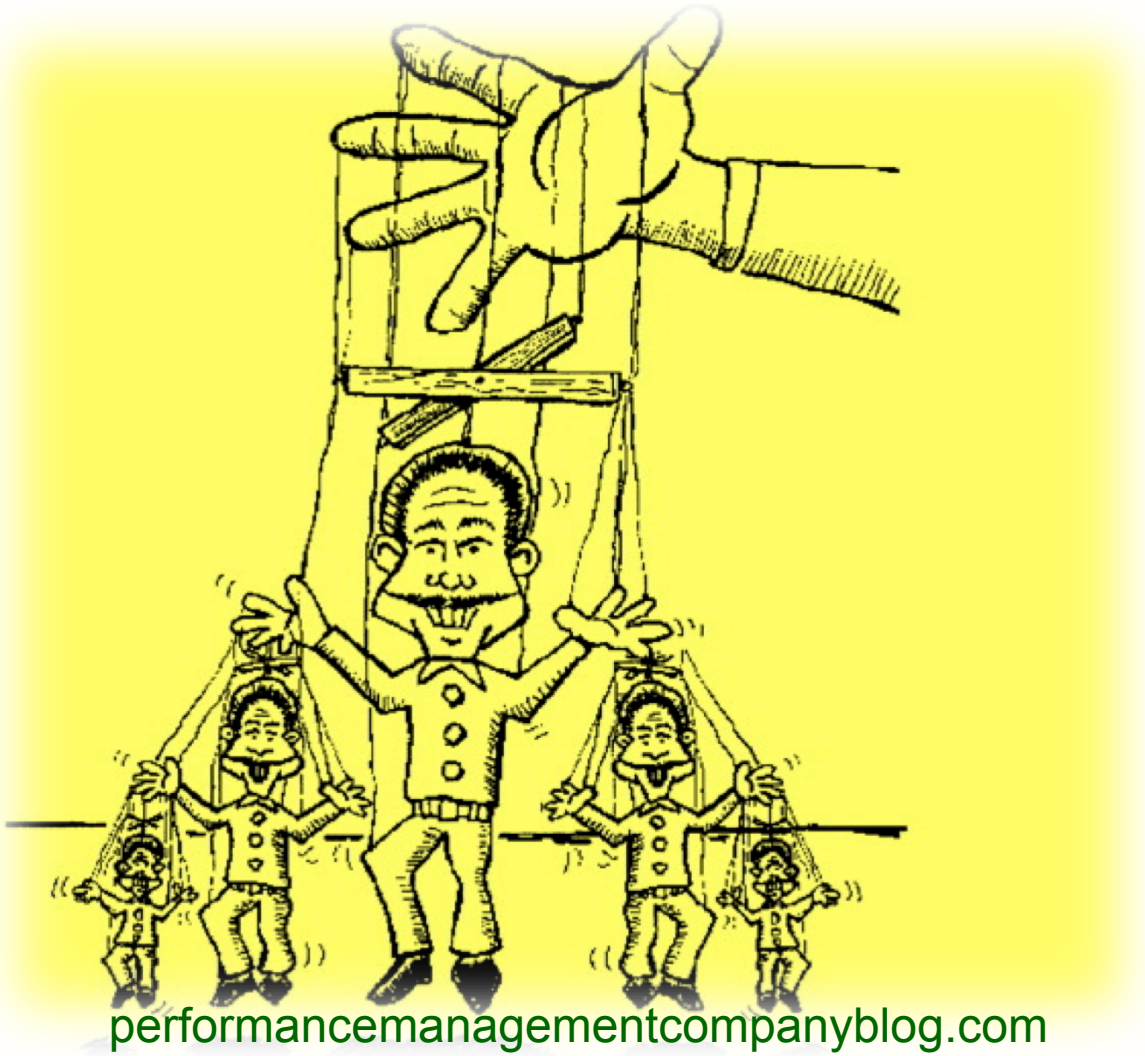
Context-switch time is overhead; the system does no useful work while switching

- The more complex the OS and the PCB -> longer the context switch

Time dependent on hardware support

- Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

How do we control processes?



performancemanagementcompanyblog.com

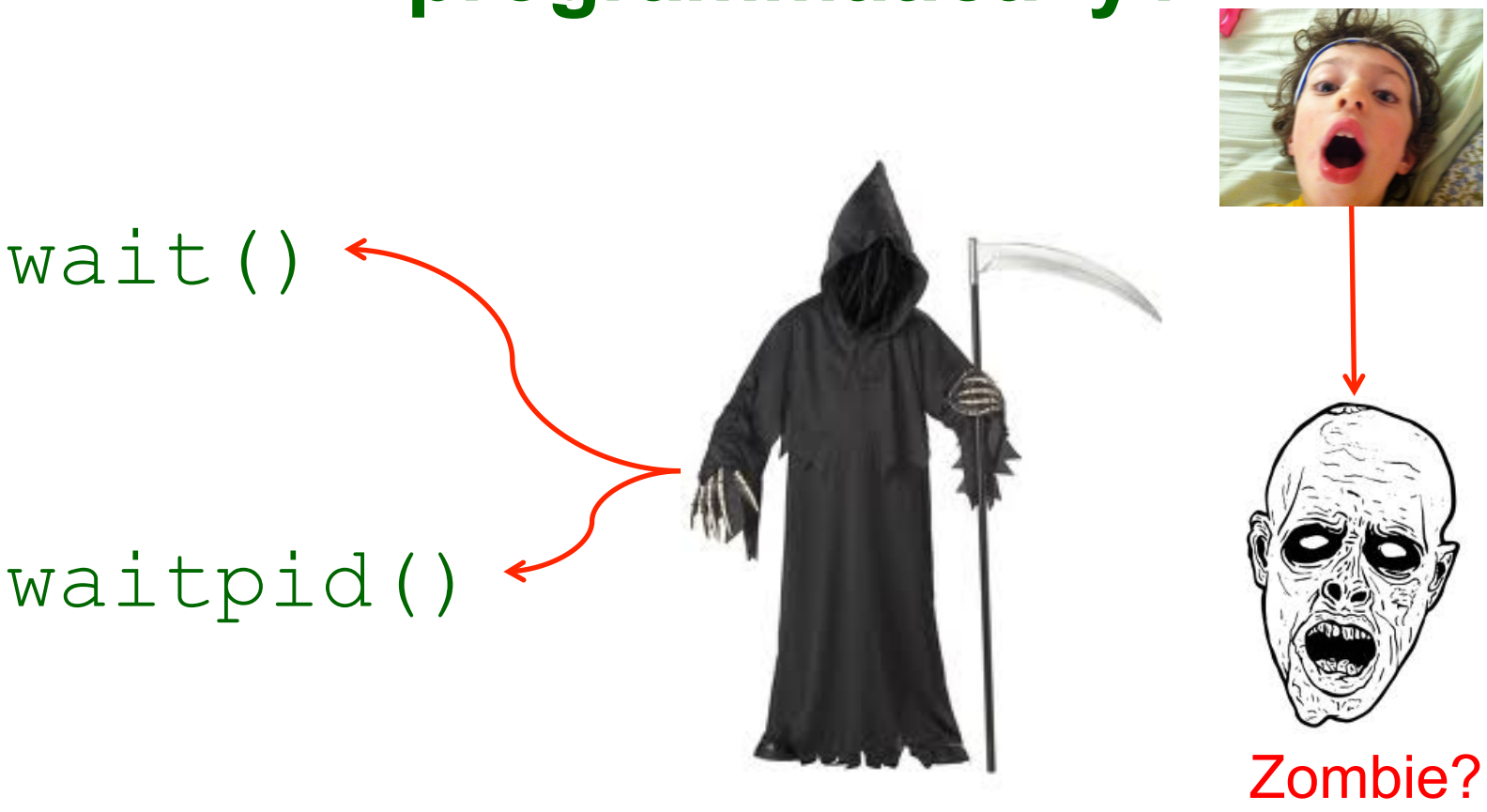
Create & Terminate Processes

We use the fork() function to create a new process.

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Create & Terminate Processes

So, how do we "reap" a child process programmatically?



InterProcesses Communitions

Pipes and FIFOs (named pipes)

- **Better for producer-consumer mode**

Shared Memory Regions (System V IPC)

Semaphores (System V IPC)

Messages

- **System V IPC messages**
- **POSIX message queues**

Sockets:

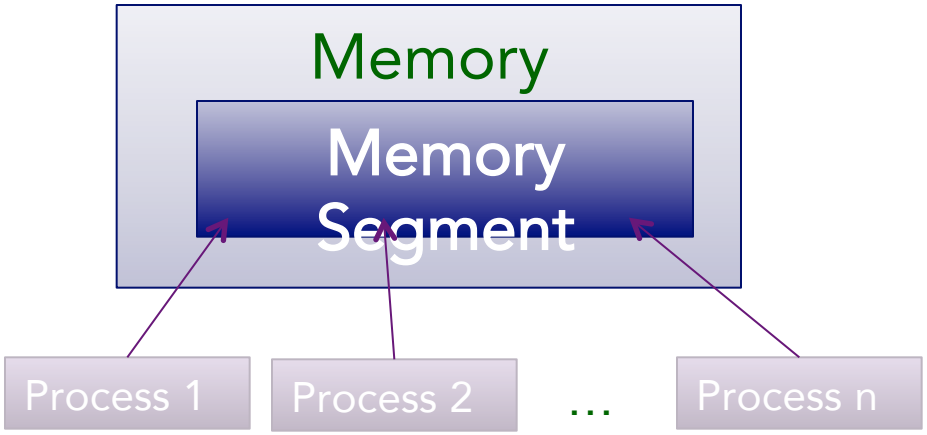
- **Processes or different computer to exchange data**

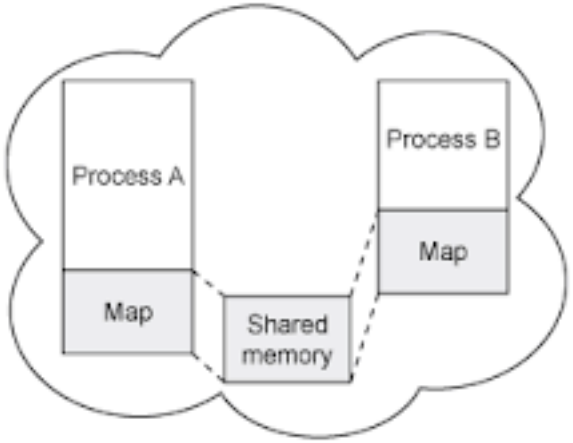
System V IPC(SysV IPC) first appeared in a development Unix variant called "Columbus Unix" and later was adopted by AT&T's

Shared Memory Segment

What is shared memory?

- Shared memory (SHM) is one method of inter-process communication (IPC) whereby 2 or more processes share a single chunk of memory to communicate





Internally

Steps of Shared Memory IPC

1. Creating the segment and connecting
2. Getting a pointer to the segment
3. Reading and Writing
4. Detaching from and deleting segments

Shared Memory, Pros and Cons

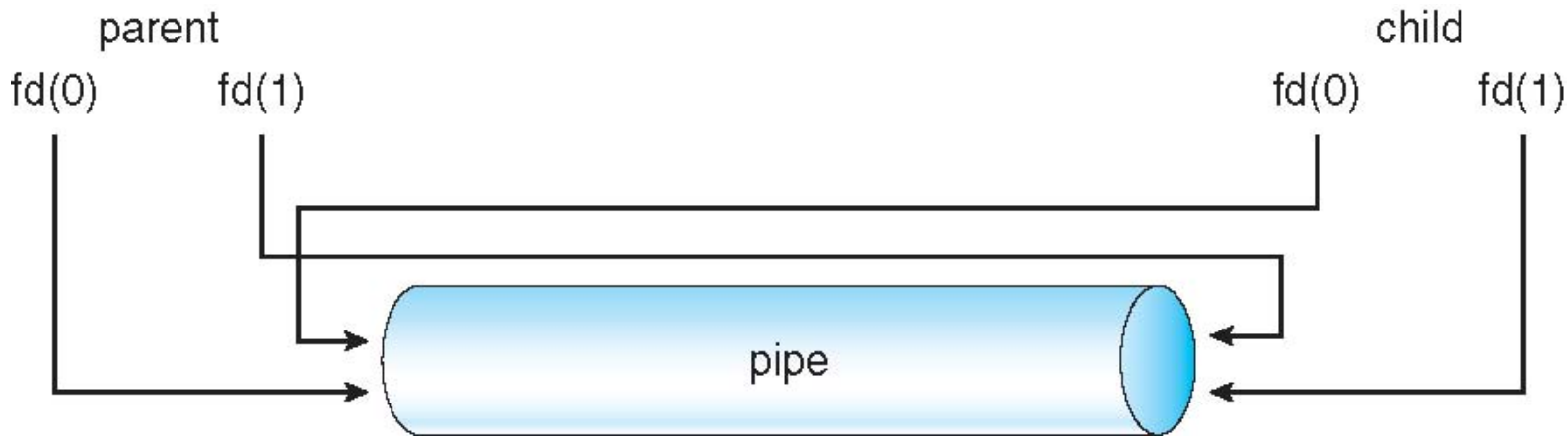
Pros

- Fast bidirectional communication among any number of processes
- Saves Resources

Cons

- Needs concurrency control (leads to data inconsistencies like 'Lost update')
- Lack of data protection from Operating System (OS)

Ordinary Pipes



Ordinary Pipe (ls | more)

1. Invokes the pipe( ) system call, which returns fd_READ(3) and fd_WRITE(4)
2. Invokes the fork( ) system call twice
3. Invokes the close( ) system call twice to fd_READ and fd_WRITE

First child that executes the "ls" performs the following:

1. Invokes dup2(4,1) so that fd 1 refers to the write channel
2. Invokes close() twice to close fd_READ(3) and fd_WRITE(4)
3. Invokes execuve("ls"), which writes output to fd 1 (the pipe)

Second Child that executes the "more" performs:

1. Invokes dup2(3,0) so that fd 0 refers to the read channel
2. Invokes close() twice to close fd_READ(3) and fd_WRITE(4)
3. Invokes execve("more") . By default, this program reads its input from the file that has file descriptor 0 (the pipe)

Ordinary Pipes

Ordinary Pipes allow communication in standard producer-consumer style

Producer writes to one end (the *write-end* of the pipe)

Consumer reads from the other end (the *read-end* of the pipe)

Ordinary pipes are therefore unidirectional

Require parent-child/sibling relationship between communicating processes

Example of using Pipe

```
main() {
  int pid, pfd[2];
  char line[100];
  pipe(pfd);
  pid = fork();
  if (pid == 0) {
    ChildFunc();
  } else {
    ParentFunc();
    waitpid(NULL);
 }
}
```

```
Void ParentFunc() {
   close(pfd[1]);
   read (pfd[0], line, sizeof(line));
   printf("date from child is: %s\n", line);
   close(pfd[0]);
}

Void ChildFunc() {
   close(pfd[0]);
   dup2(pfd[1], 1);
   close(pfd[1]);
   execl("/bin/date", "date", 0);
}
```

Problems of Pipe

Pros

- simple
- flexible
- efficient communication

Cons:

- no way to open an already existing pipe. This makes **it impossible for two arbitrary processes to share the same pipe**, unless the pipe was created by a common ancestor process.

Named Pipes (FIFO)

Named Pipes are more powerful than ordinary pipes

Communication is bidirectional

No parent-child/sibling relationship is necessary between the communicating processes

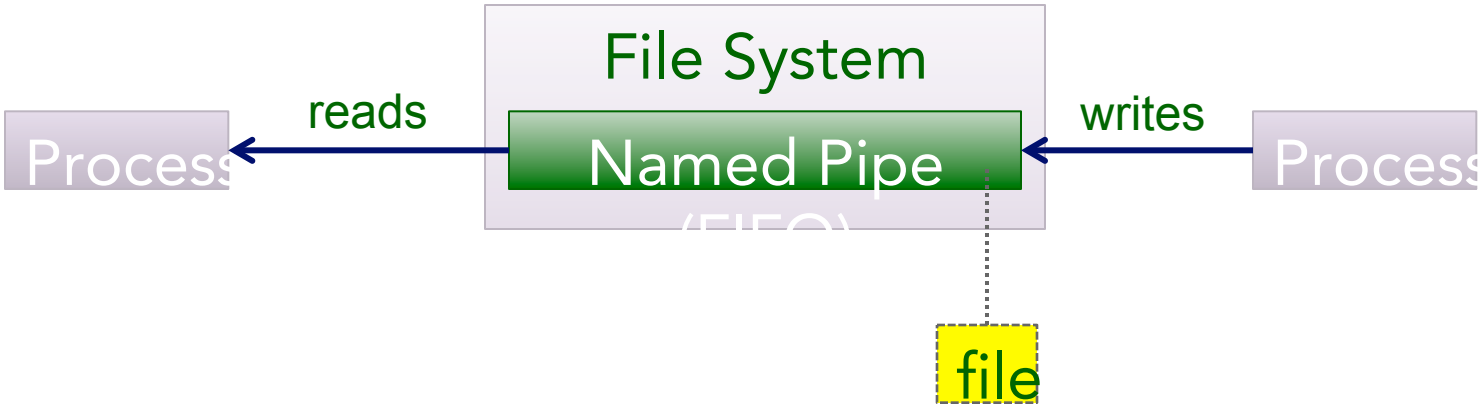
Several processes can use the named pipe for communication

Provided on both UNIX and Windows systems

Named Pipes Cont.

How does it work?

- Uses an access point (A file on the file system)
- Two unrelated processes opens this file to communicate
- One process writes and the other reads, thus it is a **half-duplex** communication



What we have covered in 2nd half semester?

Linking and Loading

Calling Convention

Virtual Memory

IO/FS

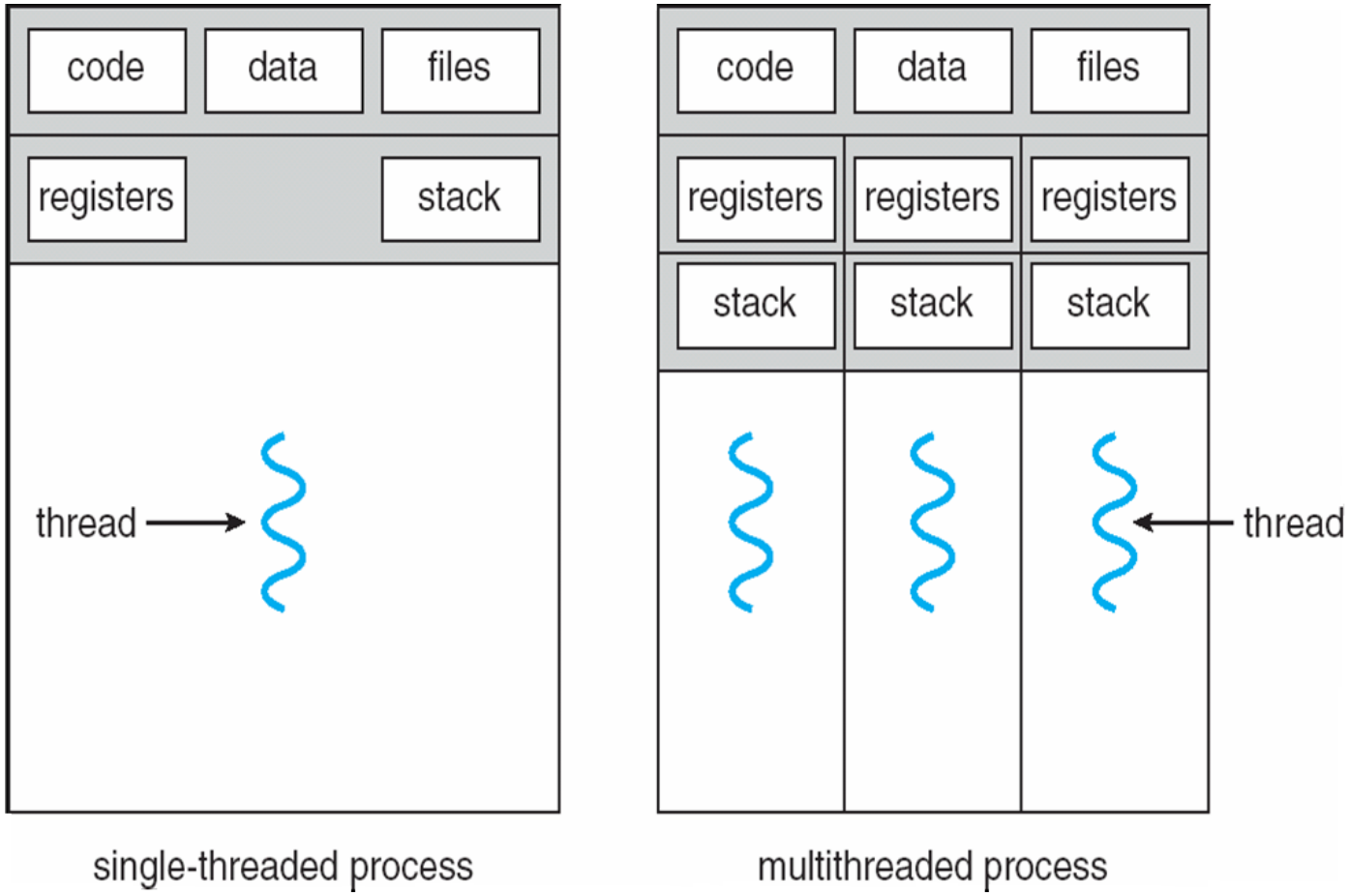
Testing, Debugging, and Profiling

Processes

Threads

Network, Cloud Computing and Big Data

Single and Multithreaded Processes



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 less expensive than processes
  - Process control (creation and exit) is more expensive as thread control
  - Context switches: processes are more expensive than for threads

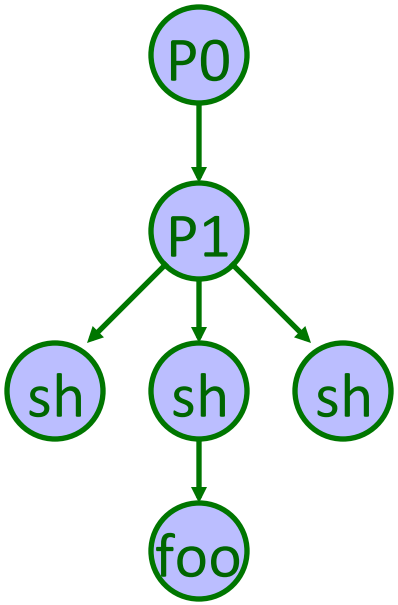
Threads vs. Processes (cont.)

- Processes form a tree hierarchy

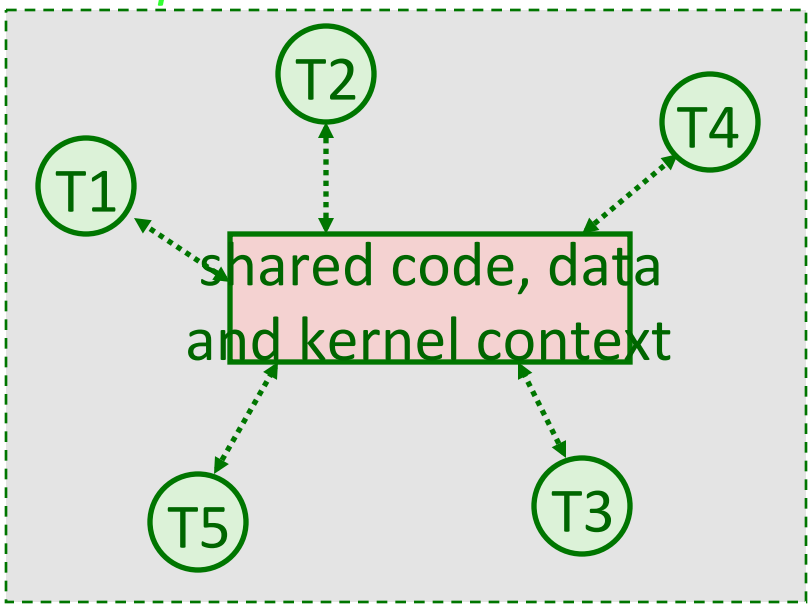
Threads form a pool of peers

- 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



Pros and Cons of Thread-Based Designs

+ Easy to share data structures between threads
  - e.g., logging information, file cache
+ Threads are more efficient than processes

– Unintentional sharing can introduce subtle and hard-to-reproduce errors!

Multithreading Models

Many-to-One

One-to-One

Many-to-Many

Threads Memory Model

Conceptual model:

- Multiple threads run in the same context of a process
- Each thread has its own separate thread context
  - Thread ID, stack, stack pointer, PC, and GP registers
- All threads share the remaining process context
  - Code, data, heap, and shared library segments
  - Open files and installed handlers

Operationally, this model is not strictly enforced:

- Register values are truly separate and protected, but…
- Any thread can read and write the stack of any other thread

Mapping Variable Instances to Memory

Global variables

- *Def:* **Variable declared outside of a function**
- **Virtual memory contains exactly one instance of any global variable**

Local variables

- *Def:* **Variable declared inside function without `static` attribute**
- **Each thread stack contains one instance of each local variable**

Local static variables

- *Def:* **Variable declared inside function with the `static`**

Mapping Variable Instances to Memory

Global var: 1 instance (`ptr` [data])

Local vars: 1 instance (`i.m, msgs.m`)

Local var: 2 instances (
`myid.p0` [peer thread 0's stack],
`myid.p1` [peer thread 1's stack]
)

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    …….
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (svar=%d)\n",
        myid, ptr[myid], ++cnt);
}
```

Local static var: 1 instance (`cnt` [data])

sharing.c

Mapping Variable Instances to Memory

Global var: 1 instance (`ptr` [data])

Local vars: 1 instance (`i.m`, `msgs.m`)

Local var: 2 instances (
`myid.p0` [peer thread 0's stack],
`myid.p1` [peer thread 1's stack]
)

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

                %s (svar=%d)\n",
            tr[myid], ++cnt);
```

1 instance (`cnt` [data])

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p0`	no	no	yes

Mutual Exclusion

Mutual exclusion

- Ensuring that no two or threads are in a critical section at the same time.

Critical section

- code only one thread can execute at a time

Lock

- mechanism for mutual exclusion
- Lock entering critical section, accessing shared data
- Unlock when complete
- Wait if locked

	Mutex	Spinlock
Mechanism	Test for lock. If available, use the resource If not, go to wait queue	Test for lock. If available, use the resource. If not, loop again and test the lock till you get the lock
When to use	Used when putting process is not harmful like user space programs. Use when there will be considerable time before process gets the lock.	Used when process should not be put to sleep like Interrupt service routines. Use when lock will be granted in reasonably short time.
Drawbacks	Incur process context switch and scheduling cost.	Processor is busy doing nothing till lock is granted, wasting CPU cycles.

Using Semaphores for Mutual Exclusion

Basic idea:

- **Associate a unique semaphore *mutex*, initially 1, with each shared variable.**
- **Surround corresponding critical sections with *P(mutex)* and *V(mutex)* operations.**

Terminology:

- ***Binary semaphore*: semaphore whose value is always 0 or 1**
- ***Mutex:* binary semaphore used for mutual exclusion**
  - **P- "locking" the mutex, V - "unlocking" the mutex**
- ***Counting semaphore*: used as a counter for set of available resources.**

Binary Semaphore and Mutex Lock?

Binary Semaphore:

- **No ownership**

Mutex lock

- **Only the owner of a lock can release a lock.**
- **Priority inversion safety: potentially promote a task**
- **Deletion safety: a task owning a lock can't be deleted.**

Synchronization Operations

Safety

- **Locks provide mutual exclusion**

  But, we need more than just mutual exclusion of critical regions…

Coordination

- *Condition variables* **provide ordering**

Condition Variables

Special *pthread* data structure

Make it possible/easy to go to sleep

- Atomically:
  - release lock
  - put thread on wait queue
  - go to sleep

Each CV has a queue of waiting threads

Do we worry about threads that have been put on the wait queue but have NOT gone to sleep yet?

- no, because those two actions are atomic

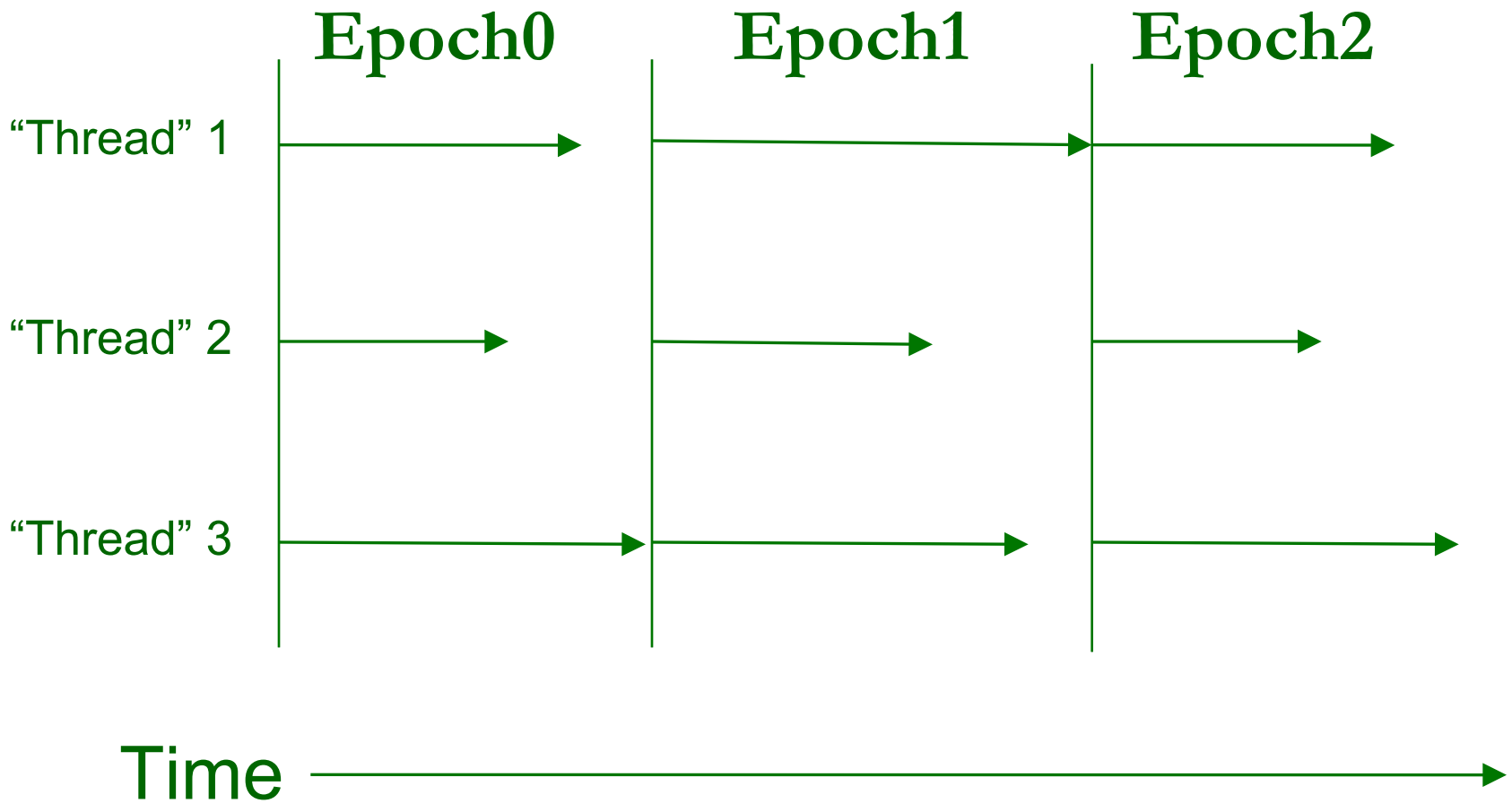
Each condition variable associated with one lock

Condition Variables

Wait for 1 event, atomically release lock

- wait(Lock& l, CV& c)
  - If queue is empty, wait
    - Atomically releases lock, goes to sleep
    - You must be holding lock!
    - May reacquire lock when awakened (pthreads do)
- signal(CV& c)
  - Insert item in queue
    - Wakes up one waiting thread, if any
- broadcast(CV& c)
  - Wakes up all waiting threads

Barrier



What we have covered in 2nd half semester?

Linking and Loading

Calling Convention

Virtual Memory

IO/FS

Testing, Debugging, and Profiling

Processes

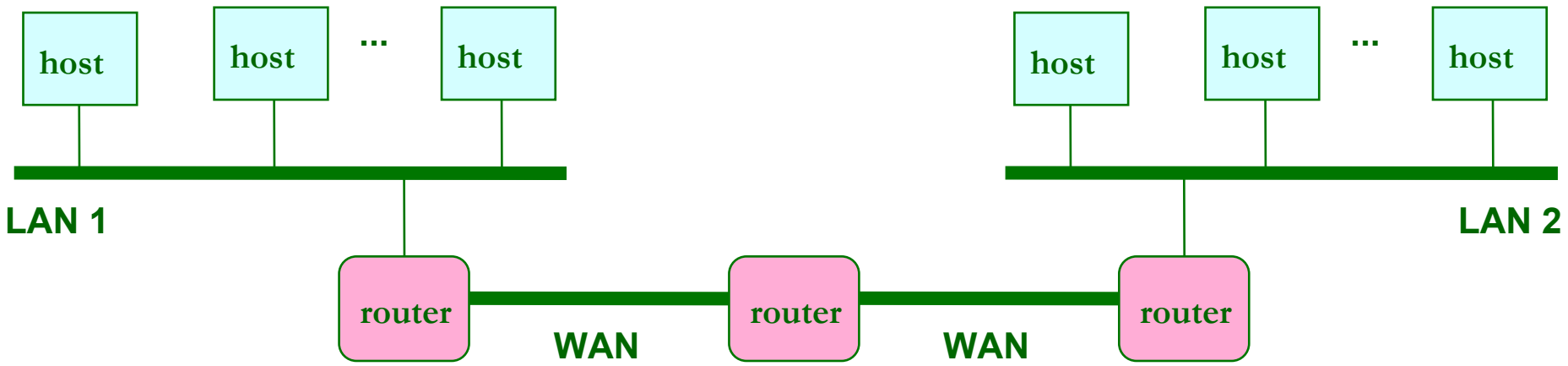
Threads

Network, Cloud Computing and Big Data

internets

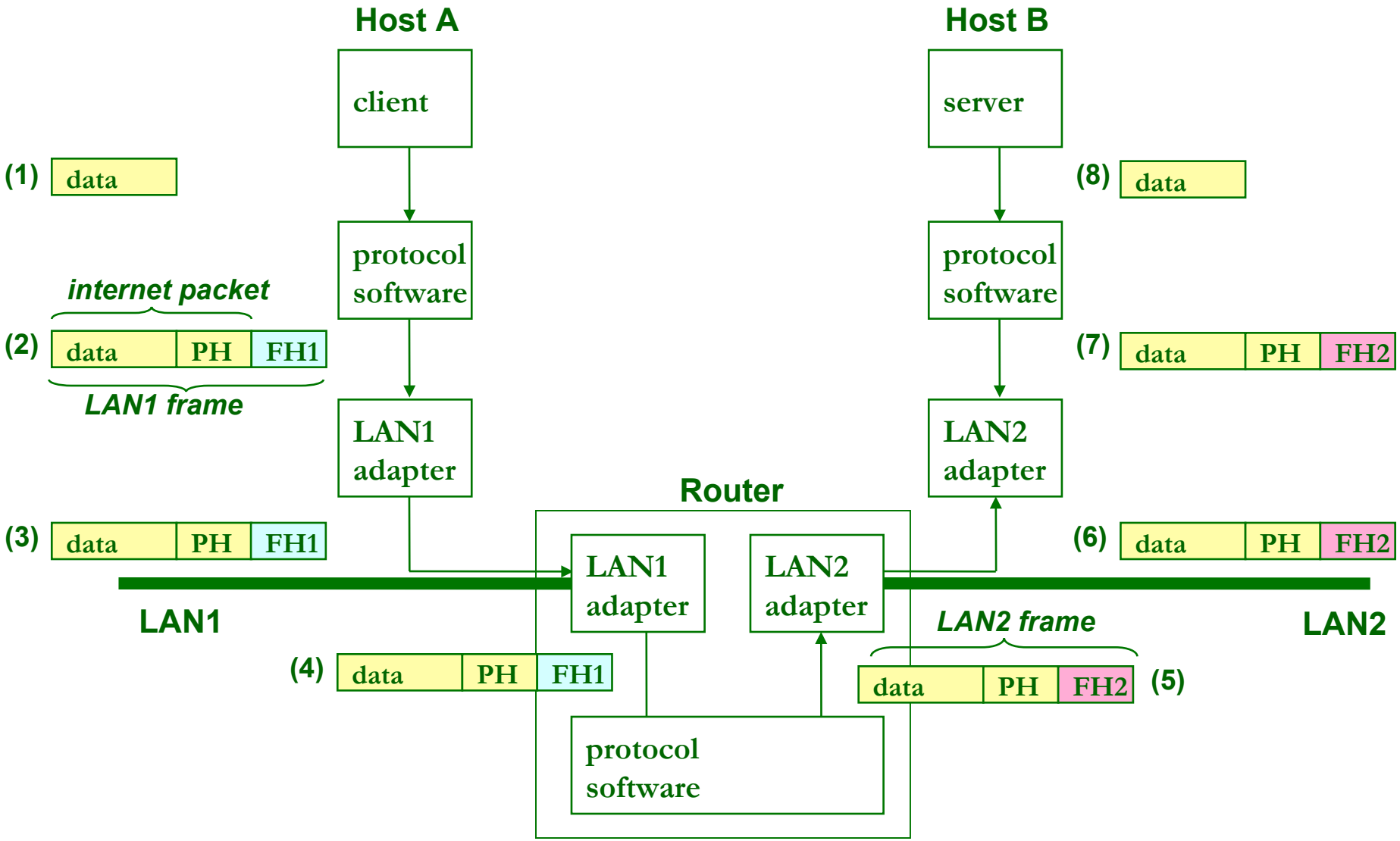
Multiple incompatible LANs can be physically connected by specialized computers called routers

The connected networks are called an internet



LAN 1 and LAN 2 might be completely different, totally incompatible LANs (e.g., Ethernet and WiFi, 802.11*, T1-links, DSL, …)

Transferring Data Over an internet



Sockets Interface

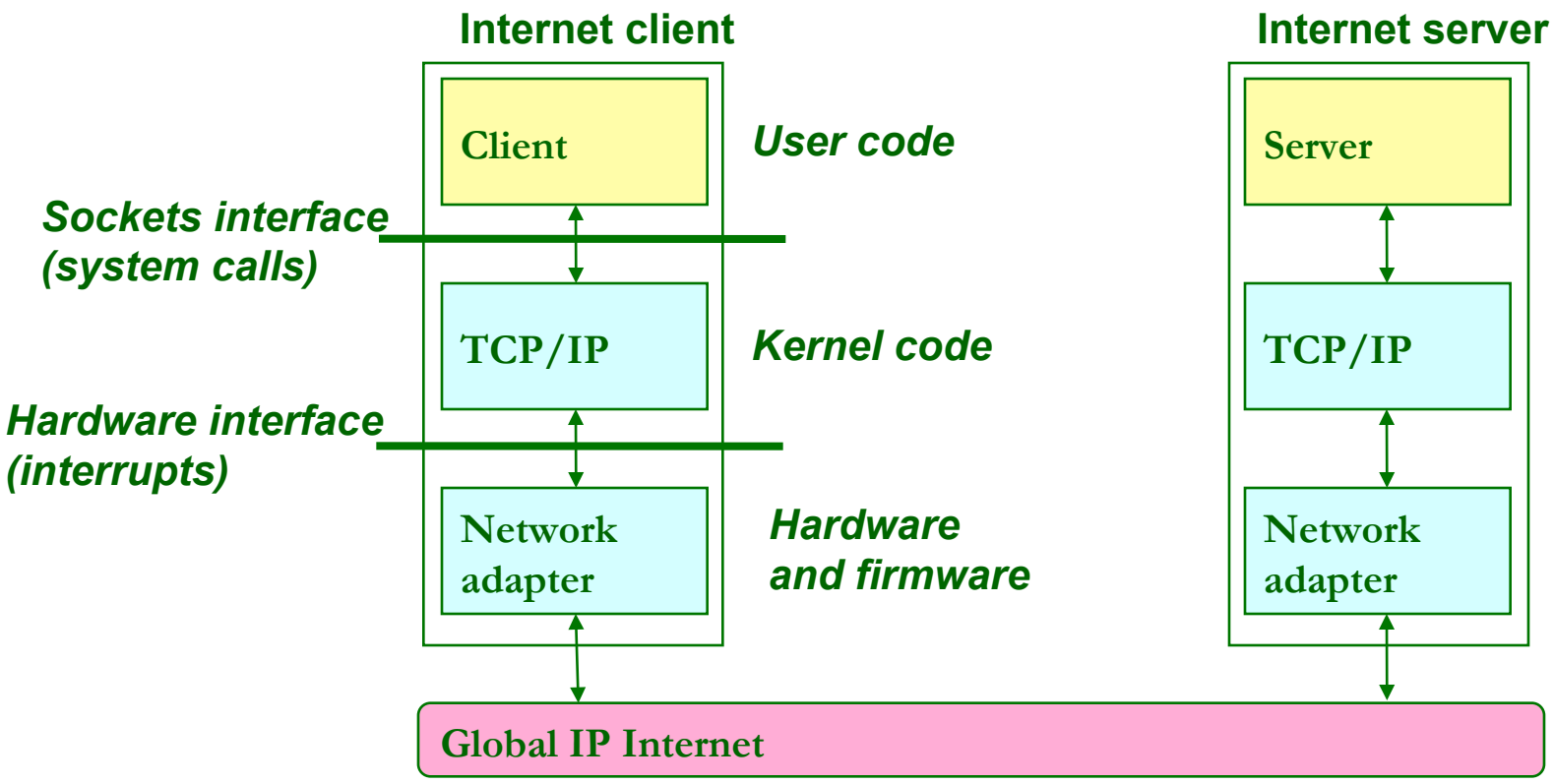
Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols

Provides a user-level interface to the network

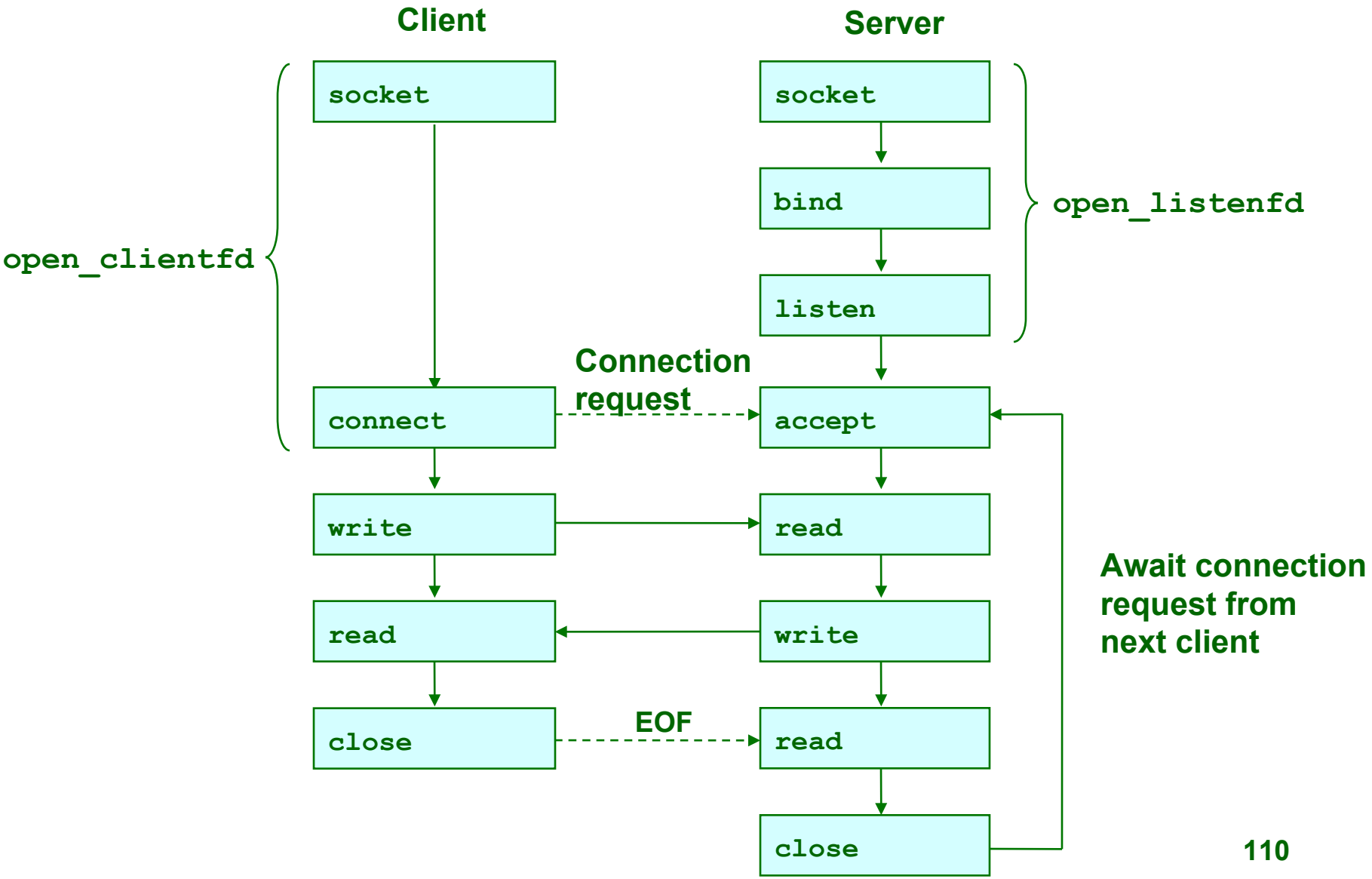
Underlying basis for all Internet applications

Based on client/server programming model

Organization of an Internet Application



Overview of the Sockets Interface



Types of Cloud Computing

Software as a Service (SaaS)

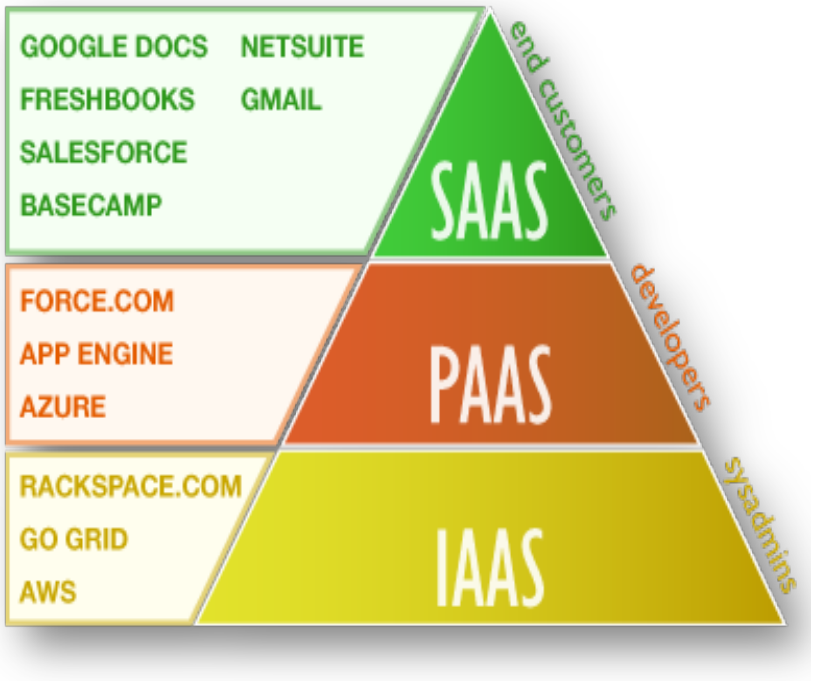
- Access to resources and applications

Platform as a Service (PaaS)

- Access to development and operational components

Infrastructure as a Service (IaaS)

- Completely outsources needed storage and resources



http://www.ladenterprizes.com/pdf/CloudComputing.pdf

Internal Engine of Cloud Computing: Virtualization

You don't need to own the hardware

Resources are rented as needed from a cloud

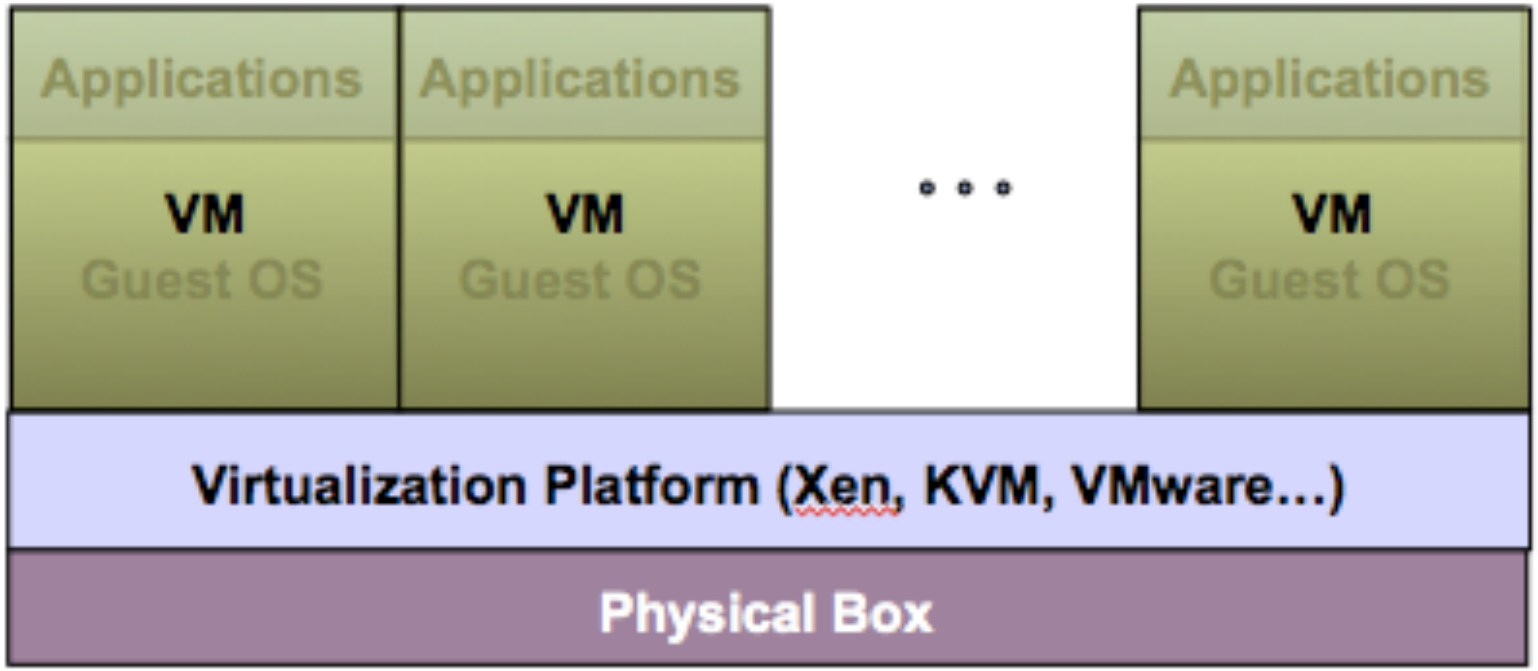
Various providers allow creating virtual servers:

- Choose the OS and software each instance will have
- The chosen OS will run on a large server farm
- Can instantiate more virtual servers or shut down existing ones within minutes

You get billed only for what you used

Virtualization Architecture

- A Virtual machine (VM) is an isolated runtime environment (guest OS and applications)
- Multiple virtual systems (VMs) can run on a single physical system



Map? Reduce?

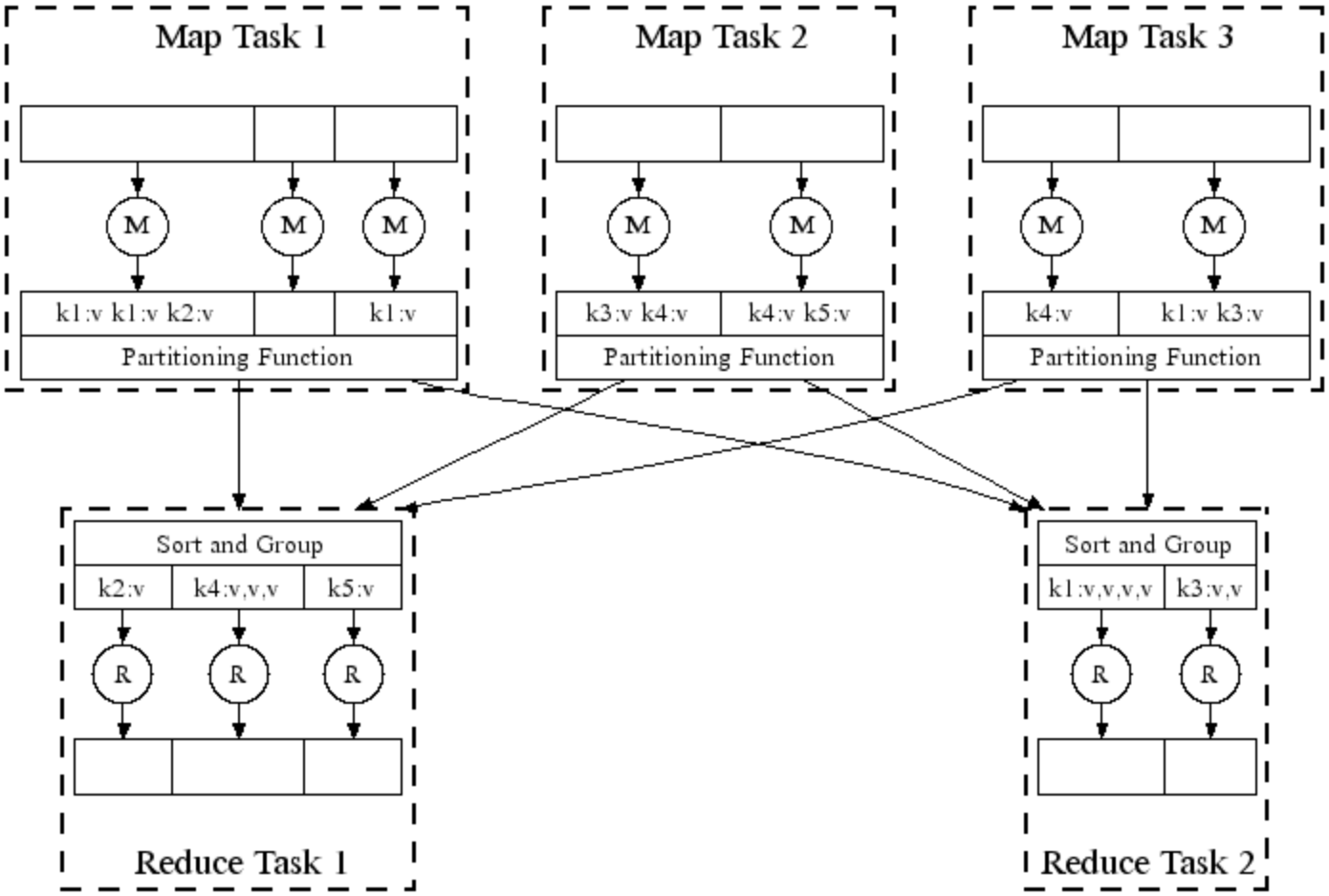
Mappers read in data from the filesystem, and output (typically) modified data

Reducers collect all of the mappers output on the keys, and output (typically) reduced data

The outputted data is written to disk

All data is in terms of key value pairs

Parallel Execution



Independent Study: Please contact me through email.