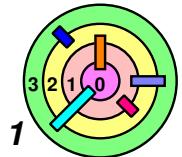


# Ch 1: Introduction

Bill Cheng

*<http://merlot.usc.edu/william/uscl/>*

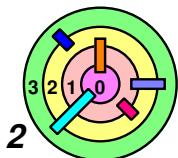


# What are Operating Systems?



## Possible definitions:

- the code that {Microsoft, Apple, Linus, Google} provides
- the code that you didn't write
- the code that runs in privileged mode
- the code that makes things work
- the code that makes things crash
- etc.



# Operating Systems

## → Abstraction

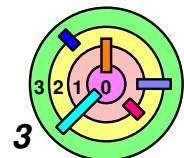
- providing an "appropriate" interface for applications
- but abstraction to what exactly? (next slide)

## → What's an "abstraction" anyway?

- think about "abstract data types" in a data structures class
  - it's data structures and associated functions to make something look like it has some behavior

## → A list object has a `sort()` function

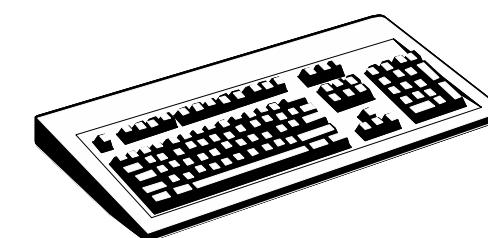
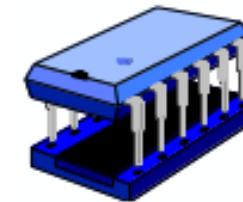
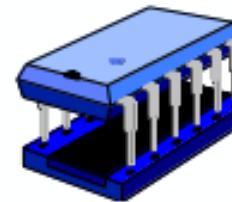
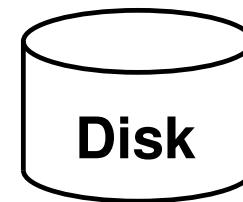
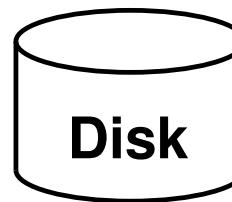
- "**objects**" is the word we use to mean **any** data types (primitive, data structures, pointers)
- can a list really sort itself?
  - of course not
  - we need to put the list under some sort of an "**execution context**" in order to execute the `sort()` function
    - ◊ umm... what's a "context"?
    - ◊ well, it's hard to say exactly what it is at this time



# Hardware

## Hardware

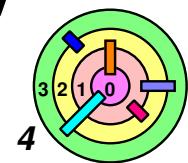
- disks
  - hard drives
  - optical drives
- memory
- processors
- network
  - ethernet
  - modem
- monitor
- keyboard
- mouse



Network

## Application programs are not allowed to use hardware directly

- that's why we have to provide abstractions



# OS Abstractions

## → Hardware

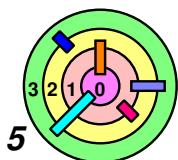
- disks
- memory
- processors
- network
- monitor
- keyboard
- mouse

## → Operating system

- *files (file system)*
- *programs (processes)*
- *threads of control*
- communication
- windows, graphics
- *input*
- locator

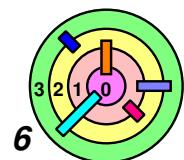
→ For those who knows about "processes", we use the word "program" to mean "process" in the introductory material

→ Application programs are not allowed to use hardware directly  
— that's why we have to provide abstractions

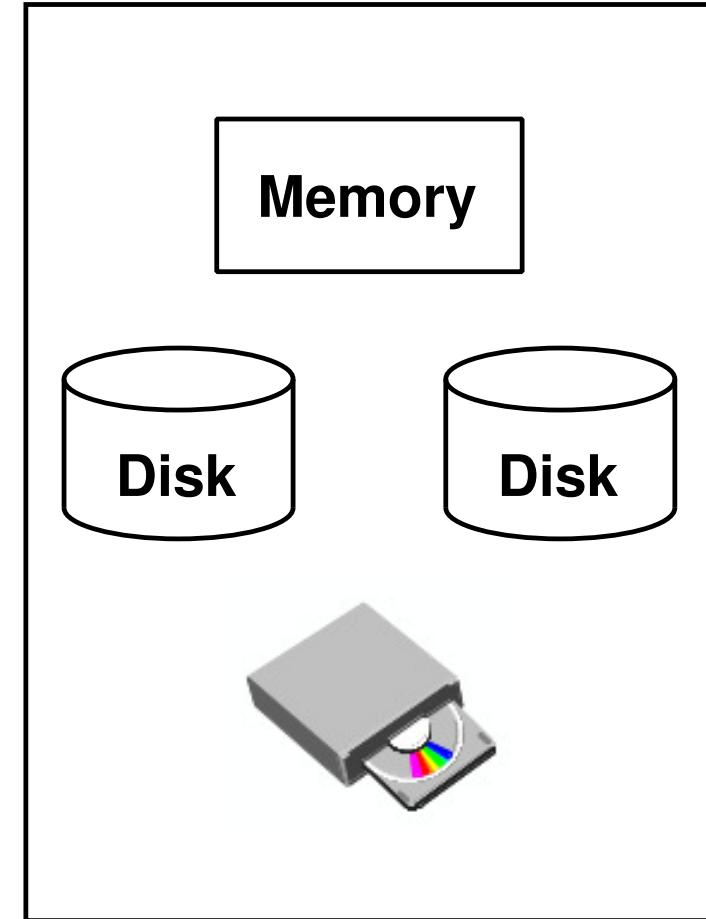
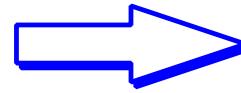
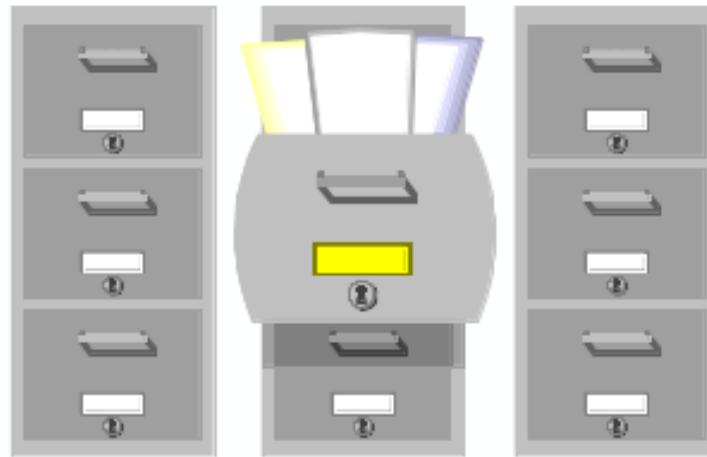


# OS Abstractions

- ➔ The main focus of this class is about how to provide these abstractions
  - don't just "hack" it until it works
  - we will talk about OS *design principles* and show how these abstractions can be *implemented*
    - this class is more about the *fundamentals* and is not a "tech" class
- ➔ Concerns
  - performance
    - time, space, energy
  - sharing and resource management
  - failure tolerance
  - security
  - marketability



# Abstraction Example: Files

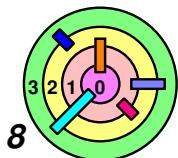


- ▶ It's nice to have a simple abstraction
- ▶ Abstraction did not come for free
  - it introduces problems that need to be solved and issues to be addressed



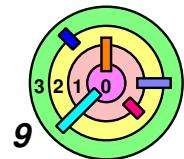
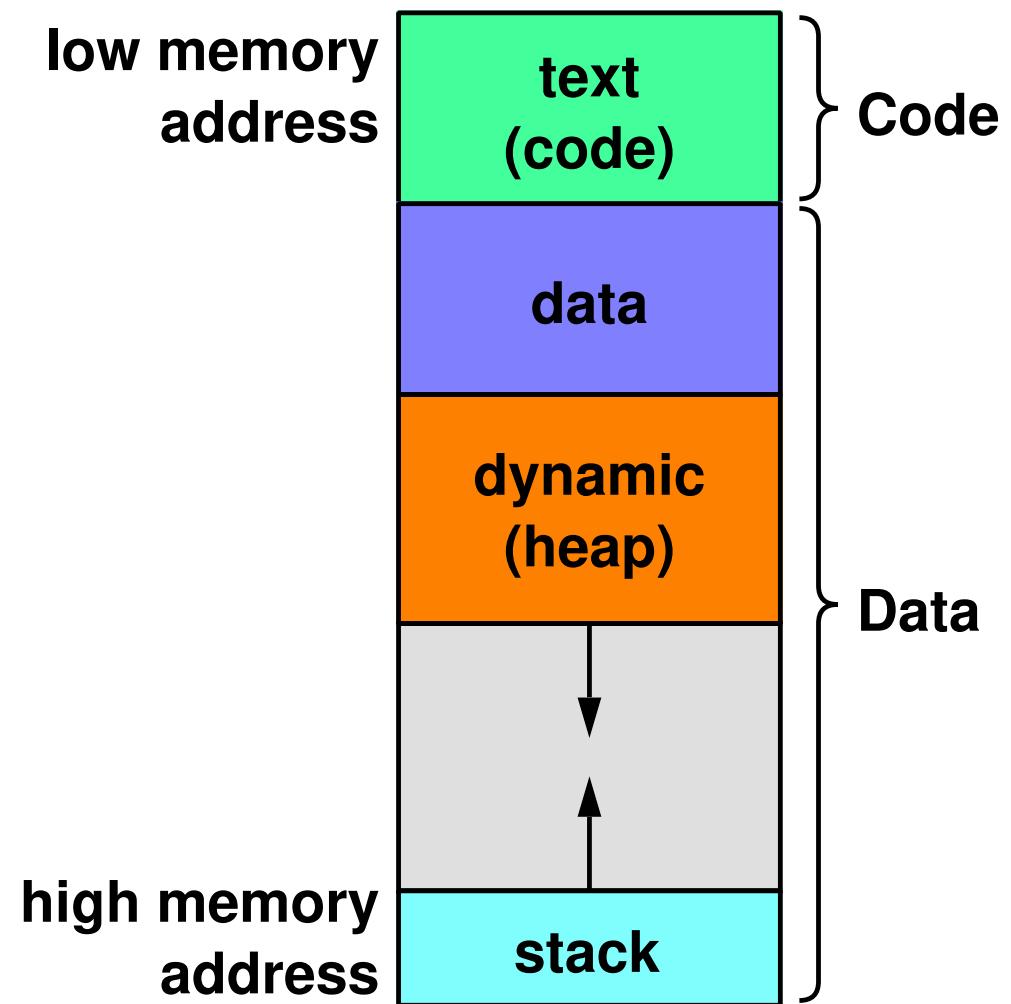
# Issues With The Files Abstraction

- ➔ **Naming**
  - device-independence
- ➔ **Allocating space on disk (permanent storage)**
  - organized for fast access
  - minimize waste
- ➔ **Shuffling data between disk and memory (high-speed temporary storage)**
- ➔ **Coping with crashes**



# Abstraction Example: Programs

Application programmers use the *Address Space* abstraction:

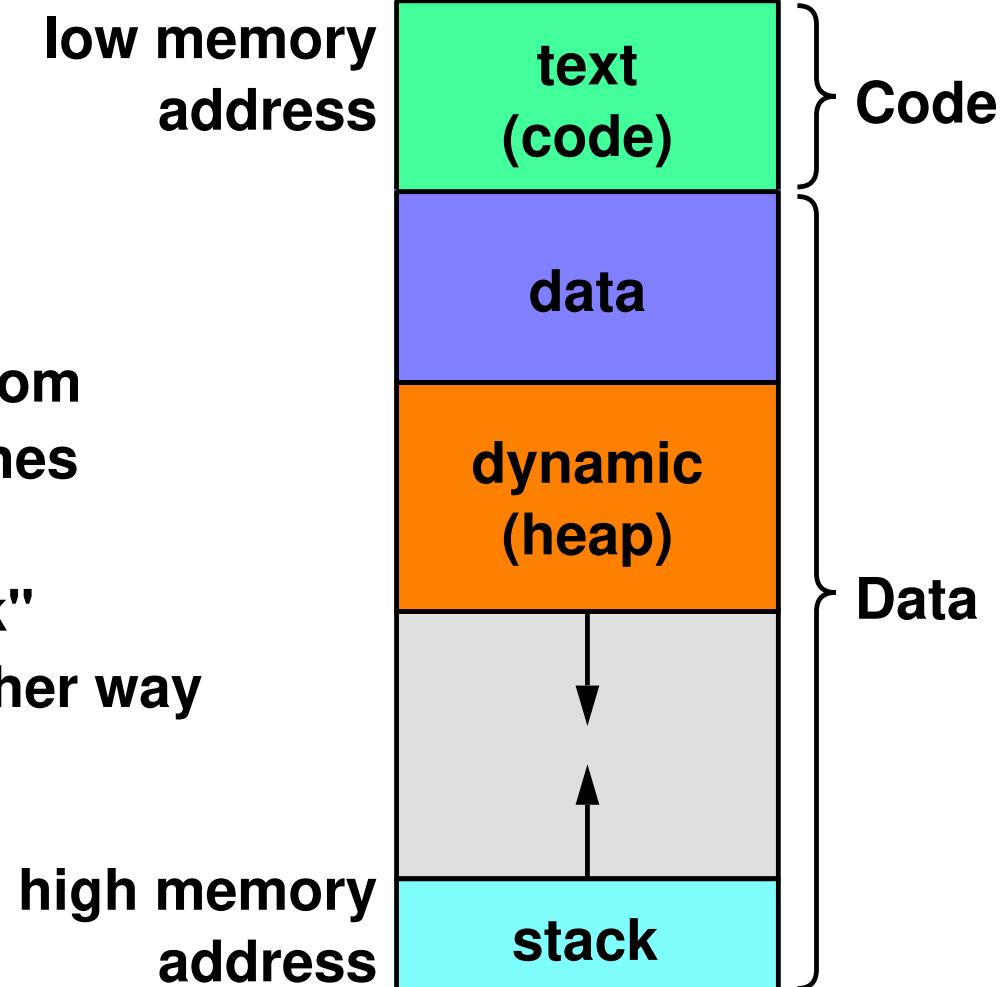


# Abstraction Example: Programs

→ Application programmers use the *Address Space* abstraction:

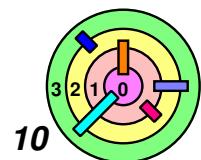
→ ***Very important:***

- our address space is *up-side-down* (compared with the textbook)
  - low address at the top
  - high address at the bottom
    - ◊ memory layout matches an array
  - stack looks like a "stack"
- our textbook does it the other way

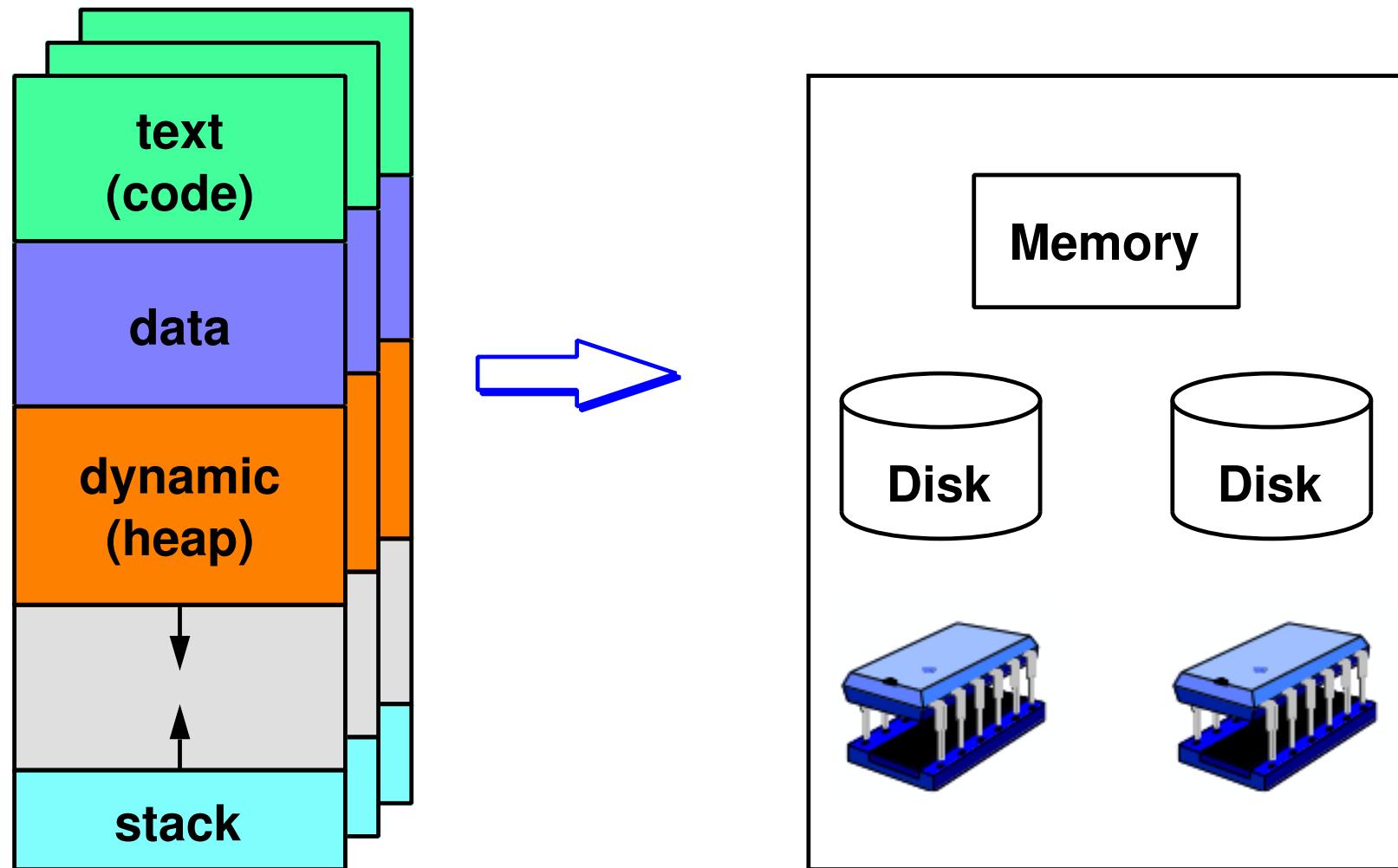


→ This is not the only possible memory layout

- compiler decides!

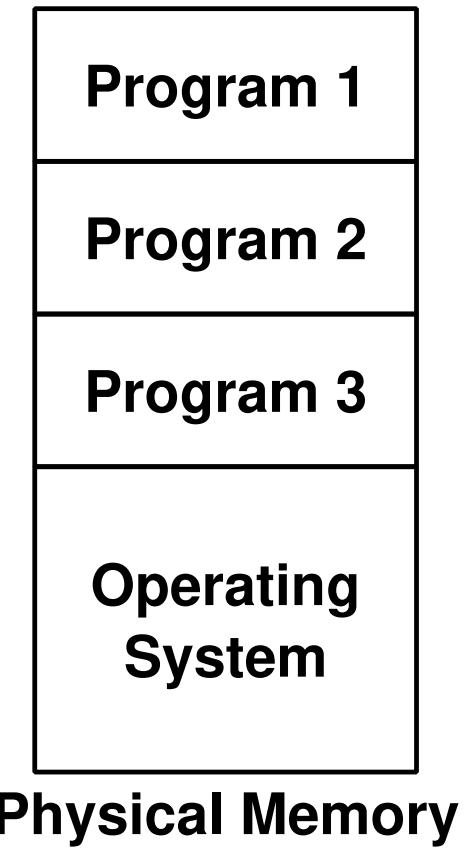


# Abstraction Example: Programs

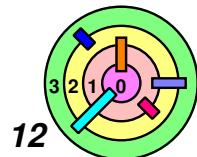


→ Application programmers do not have to worry about any *sharing* that's going on

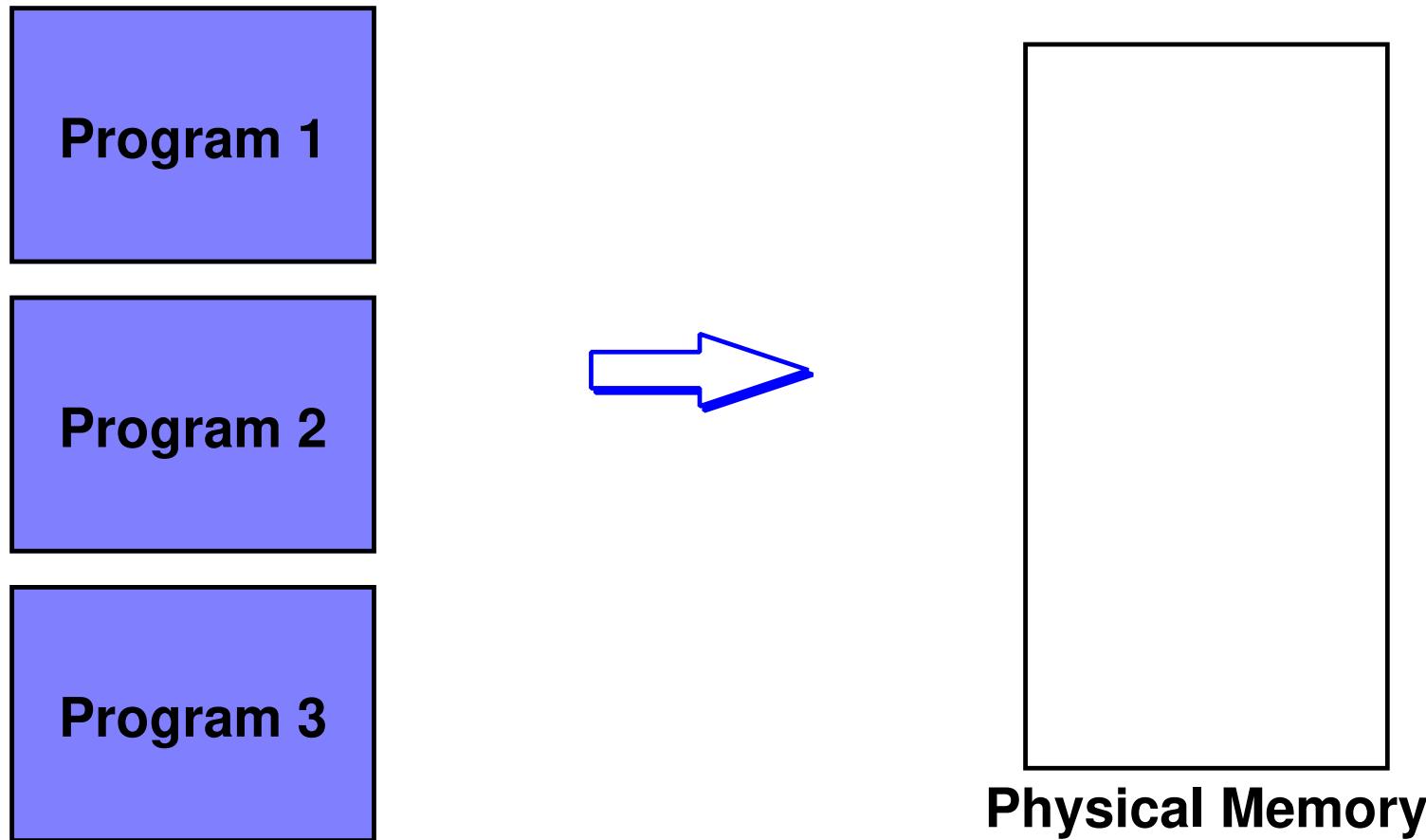
# Memory Sharing Option 1



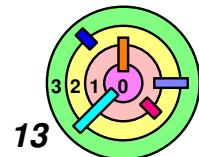
Does not appear to be very flexible



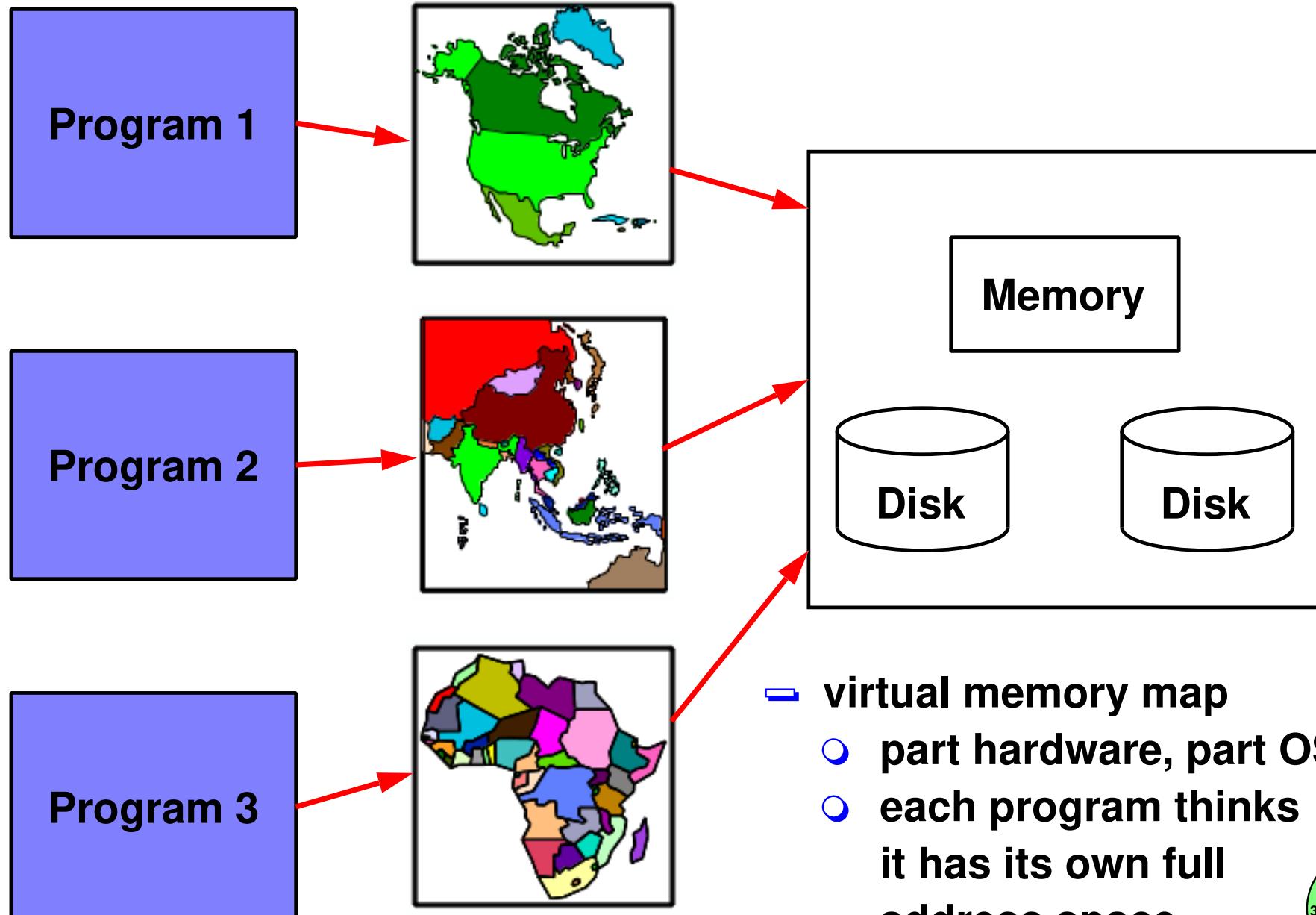
# Memory Sharing Option 2



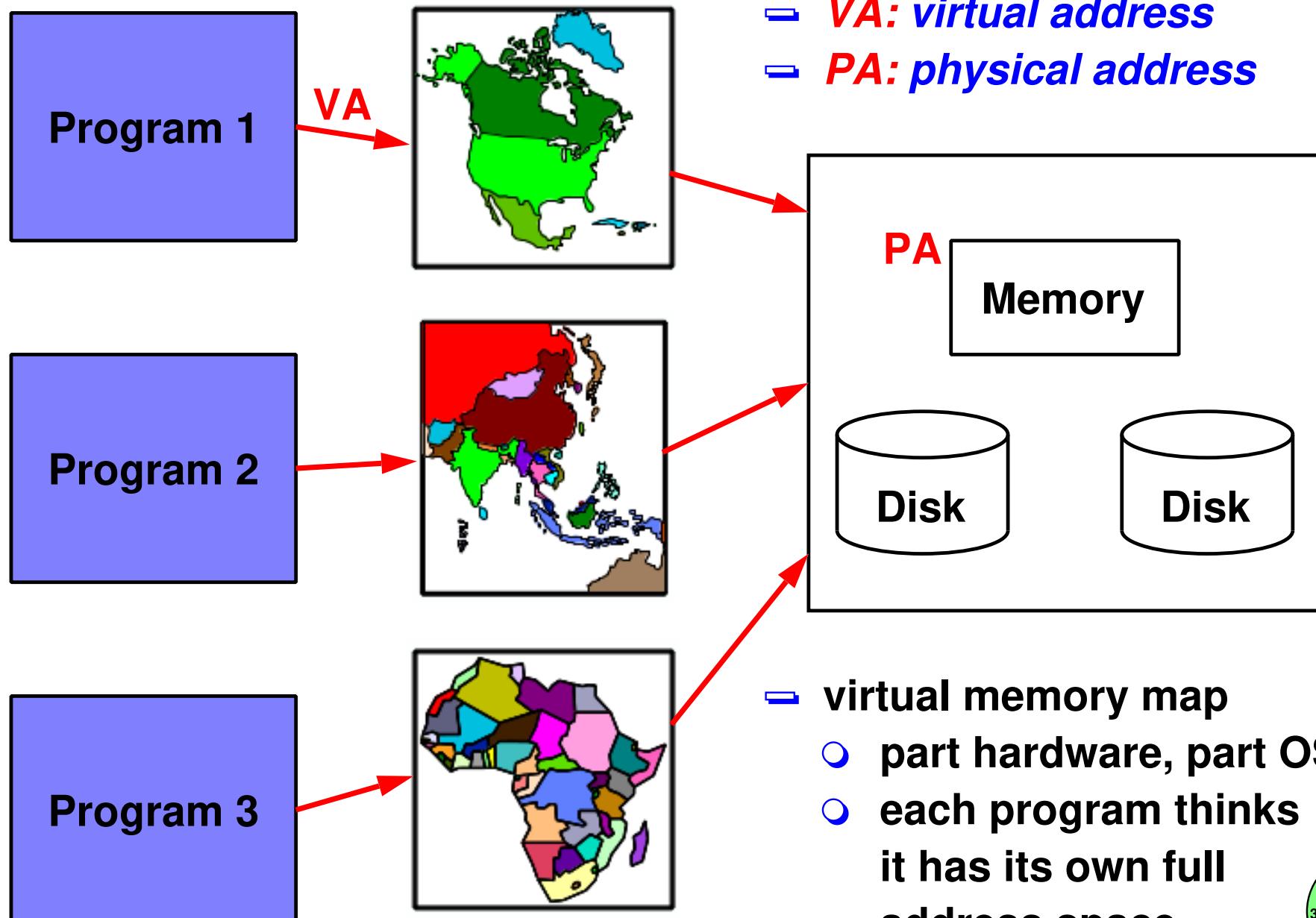
→ What if programs take up too much space (more than physical memory)?



# Virtual Memory

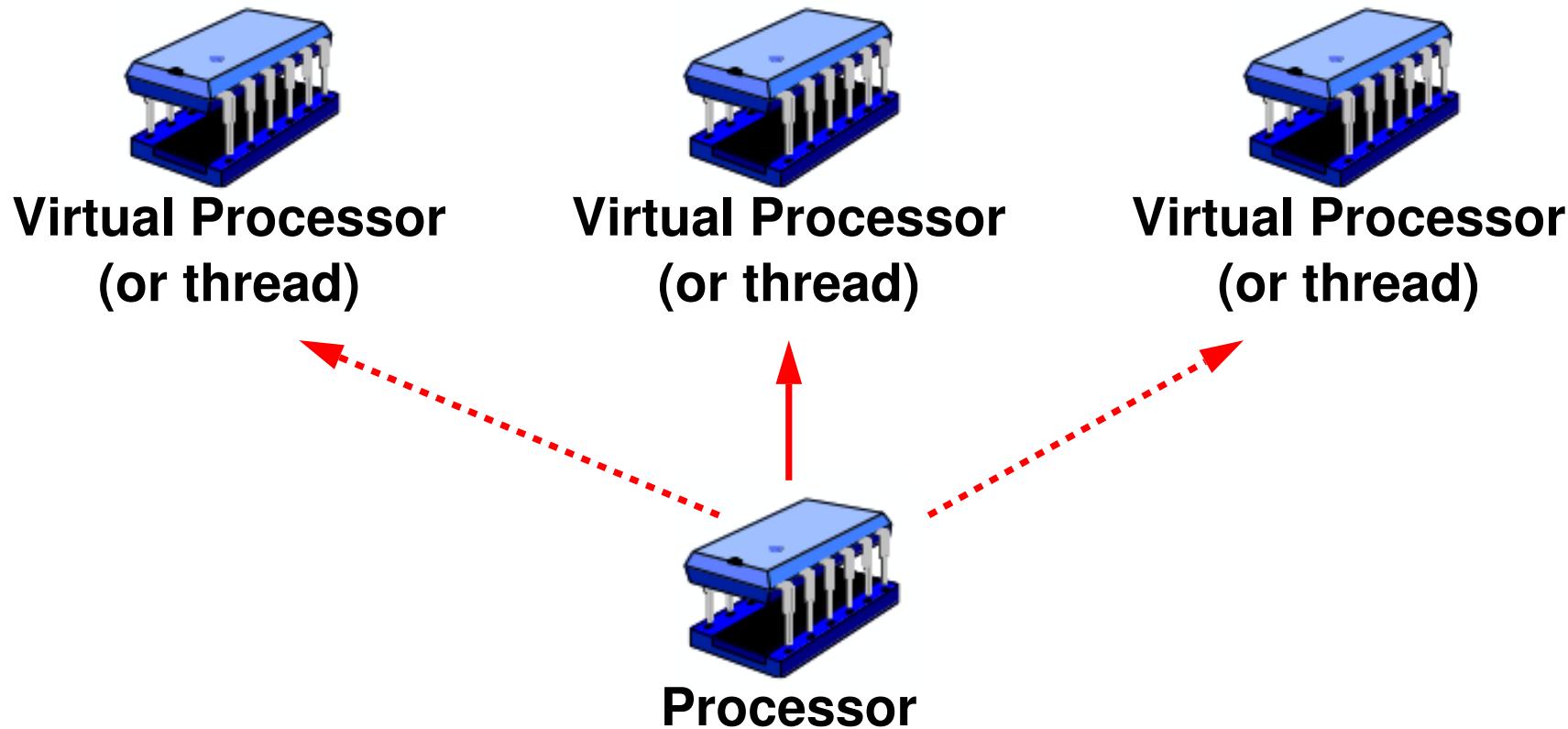


# Virtual Memory

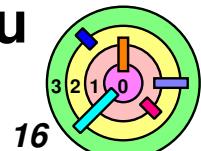


# Sharing of Processor: Concurrency

- If you only have one processor, how do you run multiple "programs" and every program thinks it owns the processor?
  - abstraction: threads (or "threads of execution")



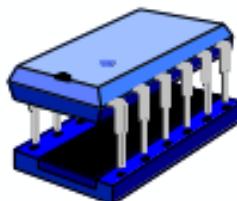
- How do you **suspend** a thread (**save execution context**) so you can **resume** its execution later (**restore execution context**)?



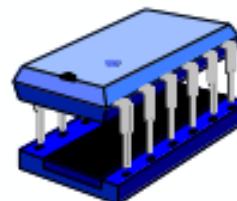
# Sharing of Processors: Parallelism



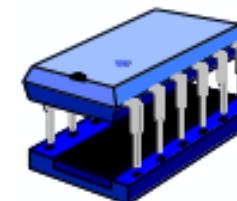
- What if you have a multicore processor or multiple processors?
  - we don't distinguish concurrency and parallelism in this class
  - can still use threads
    - but we need to worry about how well we do resource (processor) management/allocation



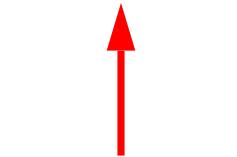
**Virtual Processor  
(or thread)**



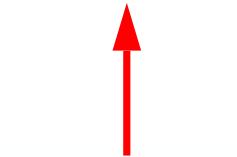
**Virtual Processor  
(or thread)**



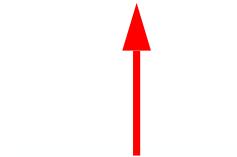
**Virtual Processor  
(or thread)**



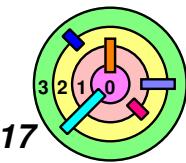
**Processor**



**Processor**

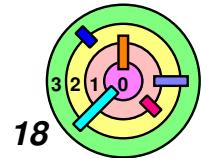


**Processor**



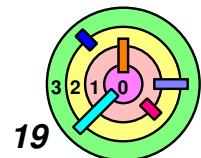
# 1960s OS Issues

- ➔ Multiprogramming (i.e., running things "in parallel" with one CPU)
- ➔ Time sharing (i.e., support interactive users)
- ➔ Software complexity
- ➔ Security



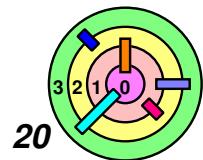
# 2010s OS Issues

- ➔ Multiprogramming (i.e., running things "in parallel" with one CPU)
  - not just one computer, but server farms
- ➔ Time sharing (i.e., support interactive users)
  - voice, video, sound, etc.
- ➔ Software complexity
  - a bigger problem than could be imagined in the 1960s
- ➔ Security
  - ditto



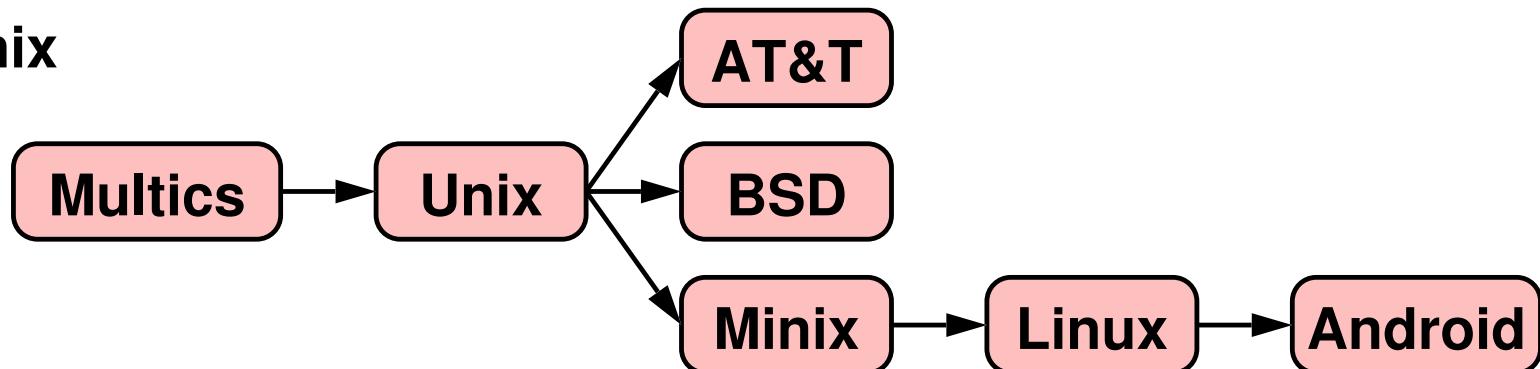
# 1.2 A Brief History of Operating Systems

- ➡ The 1950's: The Birth of the Concept
- ➡ The 1980's: The Modern OS Takes Form
- ➡ Minicomputers & Unix
- ➡ The Personal Computer

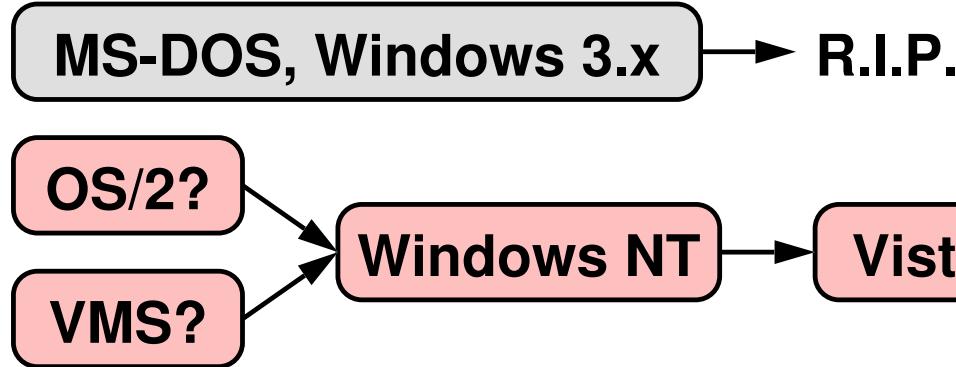


# Where Do Things Evolve From?

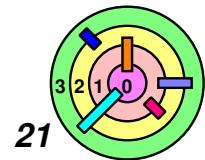
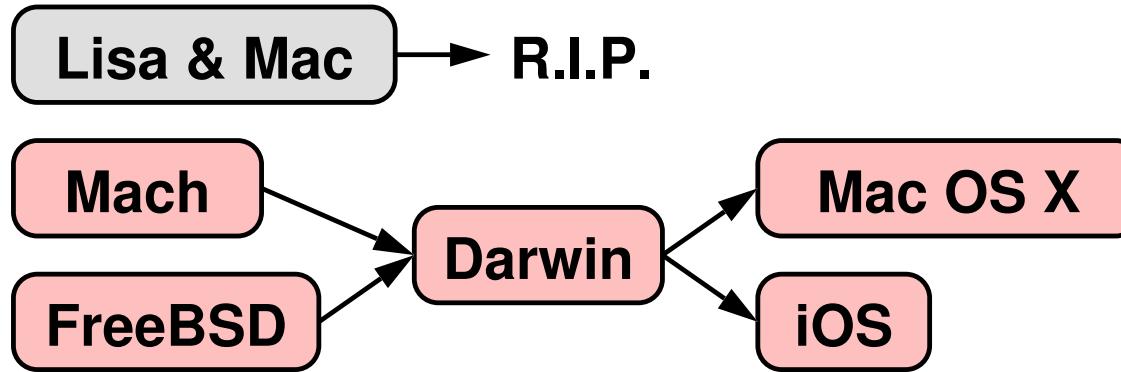
→ Linux & Unix



→ Microsoft

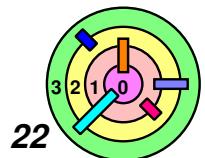


→ Apple

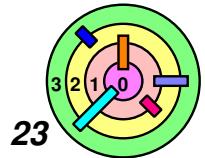


# History of C

- ➔ **Early 1960s: CPL (Combined Programming Language)**
  - developed at Cambridge University and University of London
- ➔ **1966: BCPL (Basic CPL): simplified CPL**
  - intended for systems programming
- ➔ **1969: B: simplified BCPL (stripped down so its compiler would run on minicomputer)**
  - used to implement earliest Unix
- ➔ **Early 1970s: C: expanded from B**
  - motivation: they wanted to play "Space Travel" on minicomputer
  - used to implement all subsequent Unix OSes
- ➔ **Unix has been written in C ever since**



# Extra Slides



23

# In the Beginning ...



**There was hardware**

- processor
- storage
- card reader
- tape drive
- drum



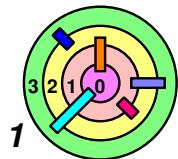
**And not much else**

- no operating system
- no libraries
- no compilers
- very little software in the beginning



# 1.3 A Simple OS

- ▶ ***OS Structure***
- ▶ **Processes, Address Spaces, & Threads**
- ▶ **Managing Processes**
- ▶ **Loading Program Into Processes**
- ▶ **Files**



# A Simple OS

→ The main focus of this class is on how to *build an OS*

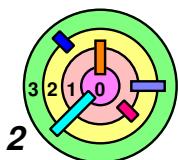
- since this is an intro class, we will focus on the fundamentals
  - occasionally, we will talk about the more advanced topics
- this is not a "tech" class

→ Sixth-Edition Unix

- source license available to universities in 1975 from Bell Labs
- had major influence on modern OSes
  - Solaris
  - Linux
  - MacOS X
  - Windows

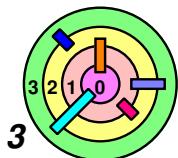
→ Fits into 64KB of memory

- single executable, completely stored in a *single file*
- *loaded* into memory as the OS boots
- *monolithic OS*



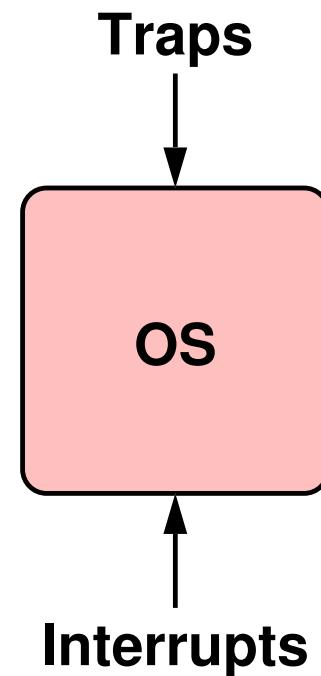
# Hardware Support - User vs. Privileged Modes

- ➔ **Processor modes:** part of the processor state (recall from your computer organization/architecture class regarding "processor")
  - most computers have at least two modes of execution
    - **user mode:** fewest privileges      user mode 不能使用hardware
    - **privileged mode:** most privileges
      - ◊ the only code that runs in this mode is part of the OS
- ➔ For Sixth-Edition Unix      6th Unix都在privilege mode里
  - the whole OS run in the privileged mode
  - everything else is an application and run in the user mode
- ➔ For other systems
  - major subsystems providing OS functionality may run in the user mode
- ➔ We use the word "***kernel***" to mean the portion of the OS that runs in privileged mode
  - sometimes, a subset of this



# A Simple OS Structure

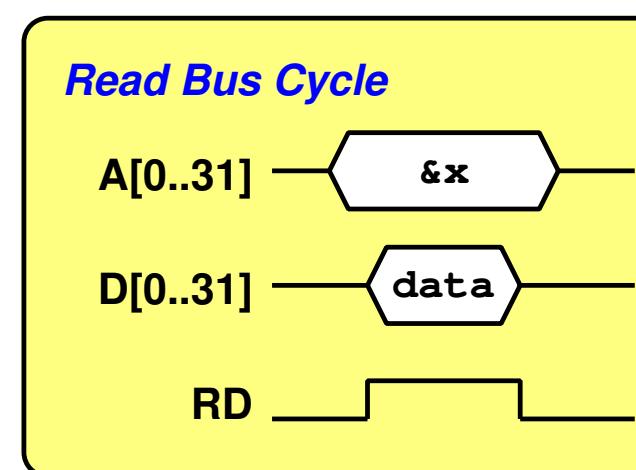
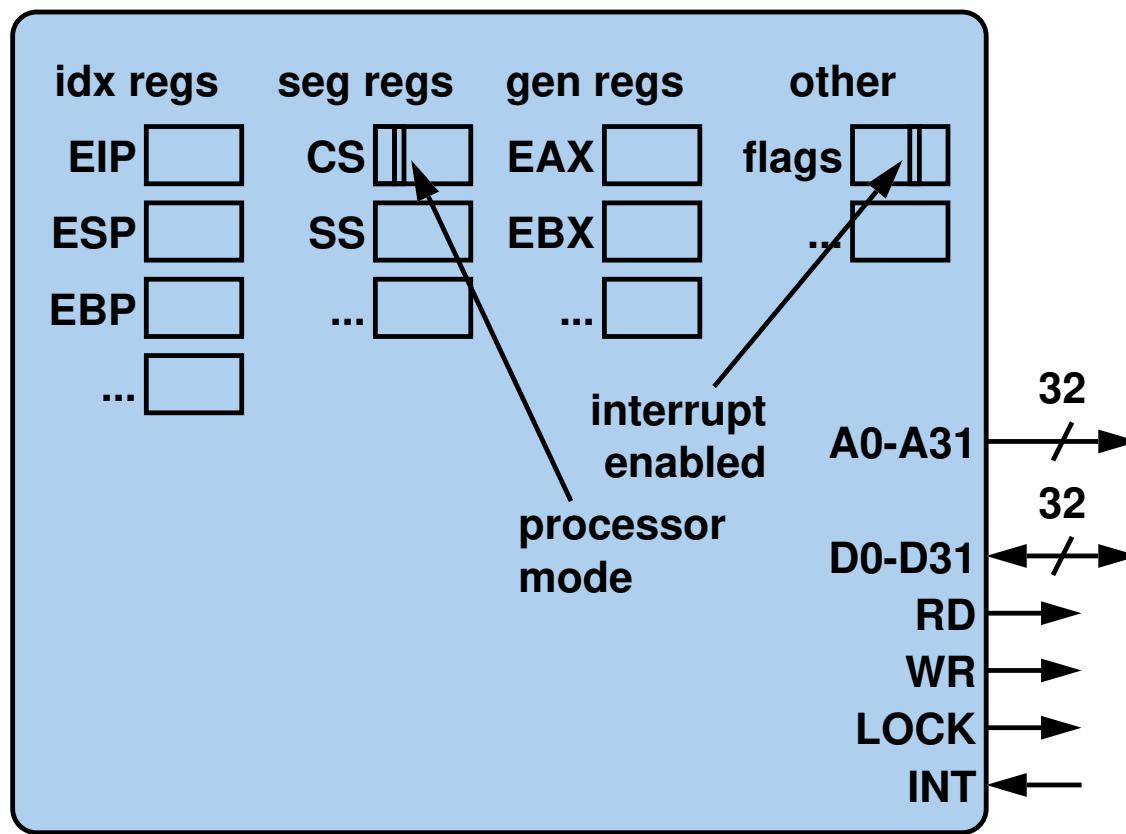
- ➡ Application programs call upon the OS via *traps*
- ➡ External devices call upon the OS via *interrupts*
  - I/O completion interrupt
    - executes *interrupt service routine*



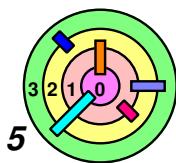
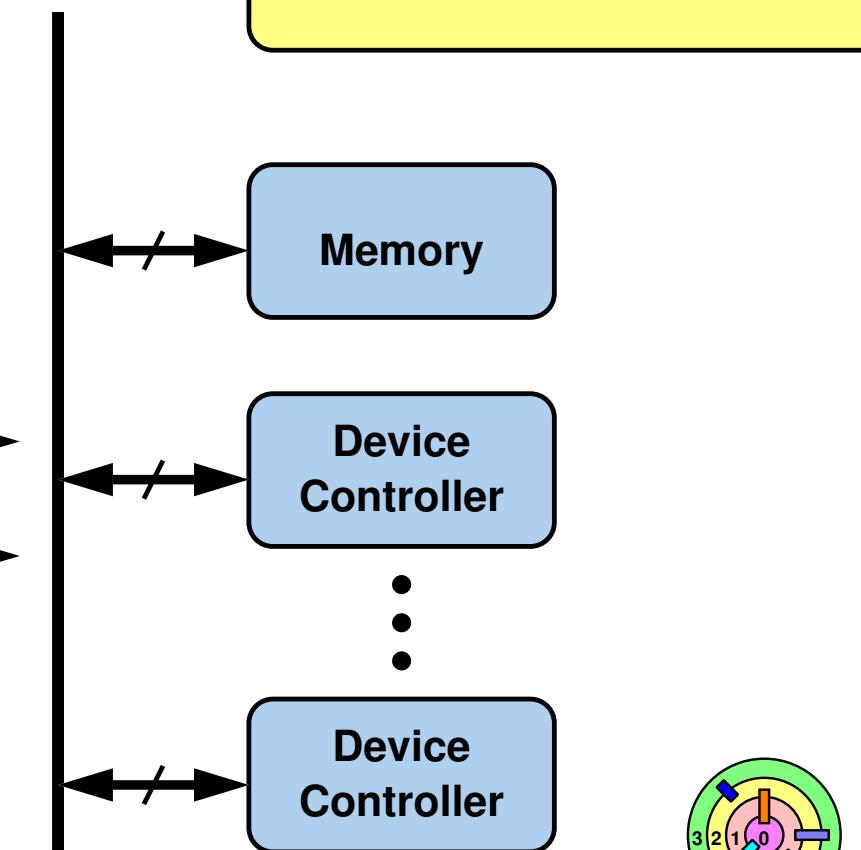
# A Simple OS Structure

Review of "Computer Organization"  
— bus architecture

x86 Processor



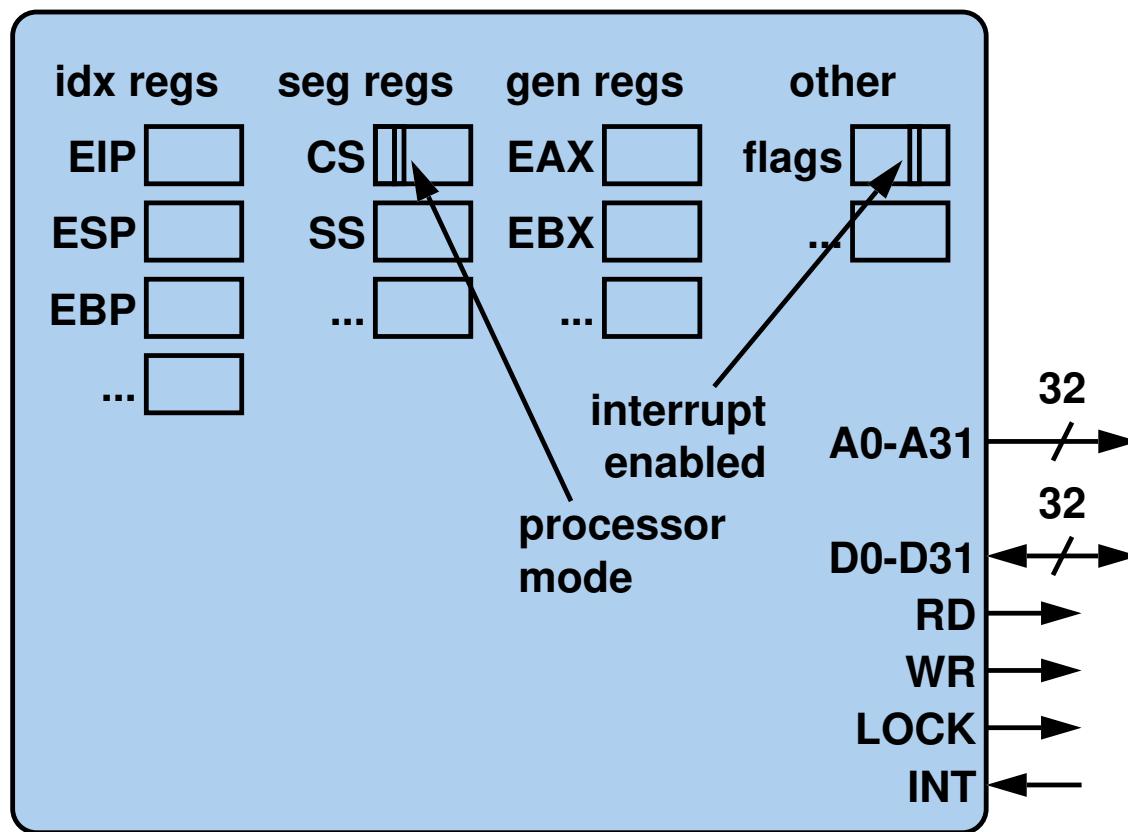
Bus



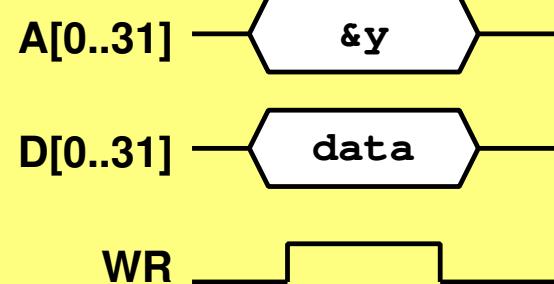
# A Simple OS Structure

Review of "Computer Organization"  
— bus architecture

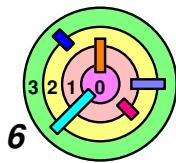
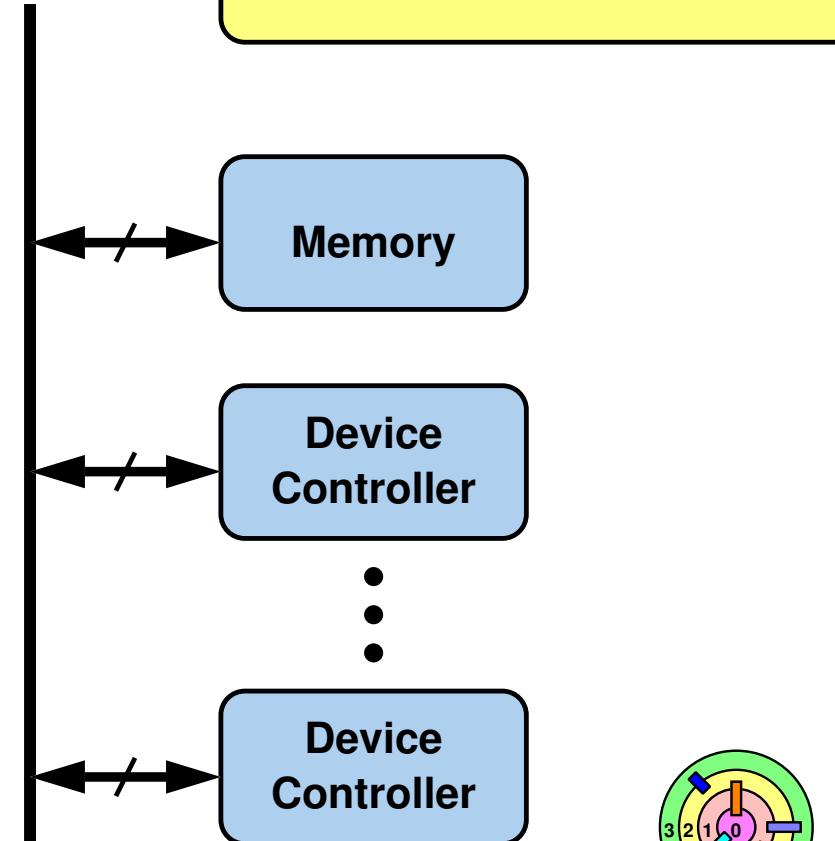
x86 Processor



## Write Bus Cycle



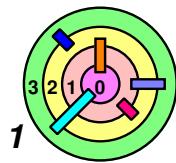
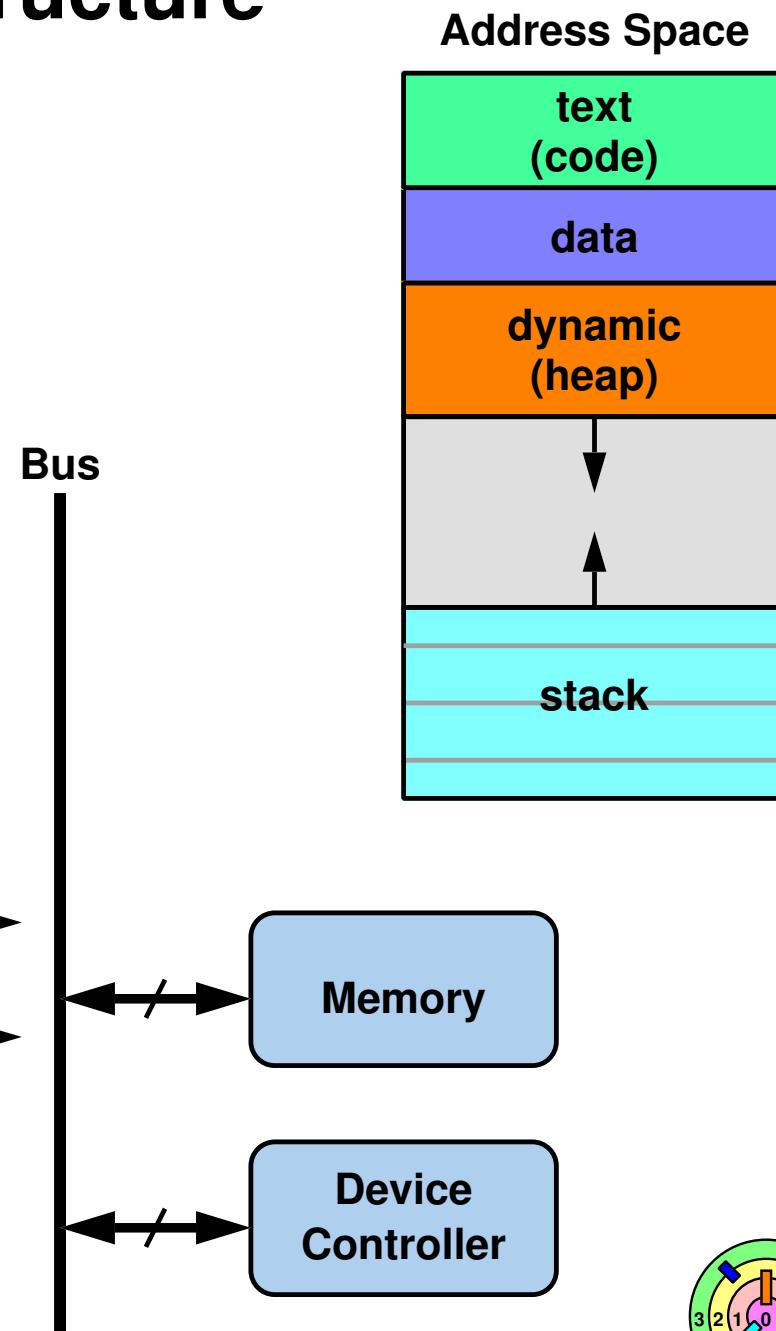
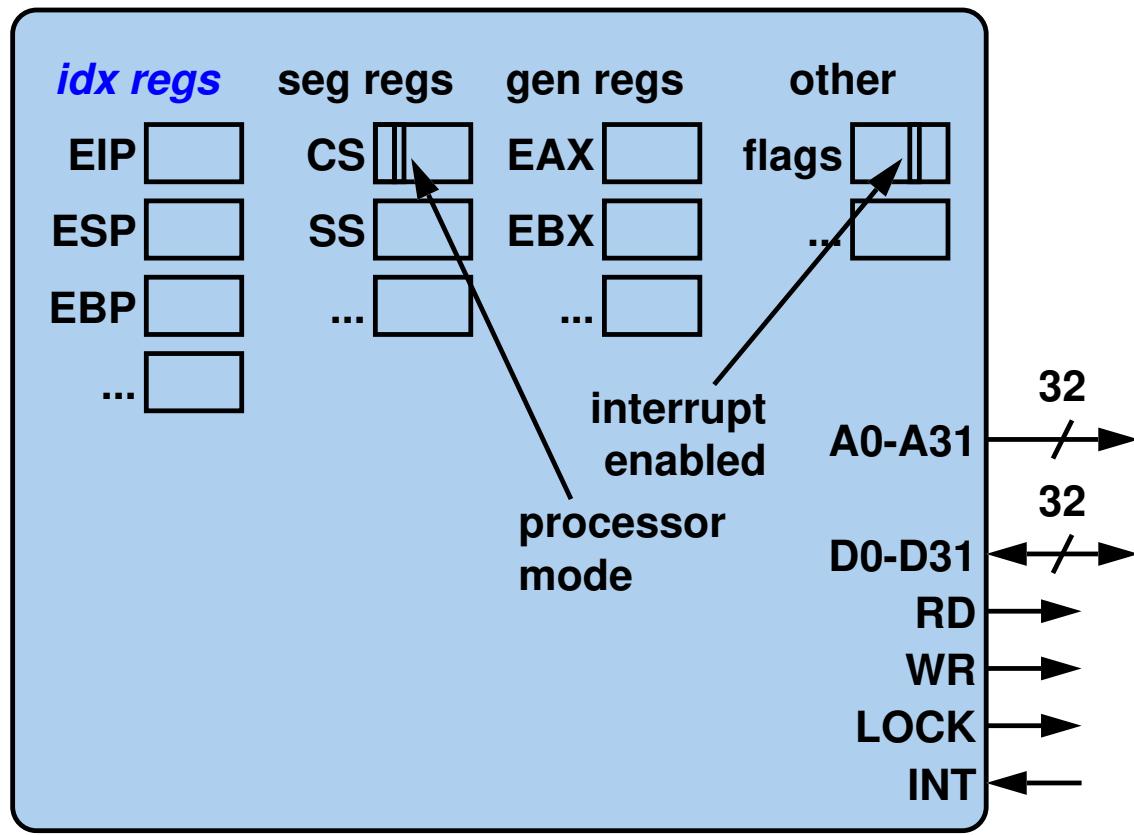
Bus



# A Simple OS Structure

Review of "Computer Organization"

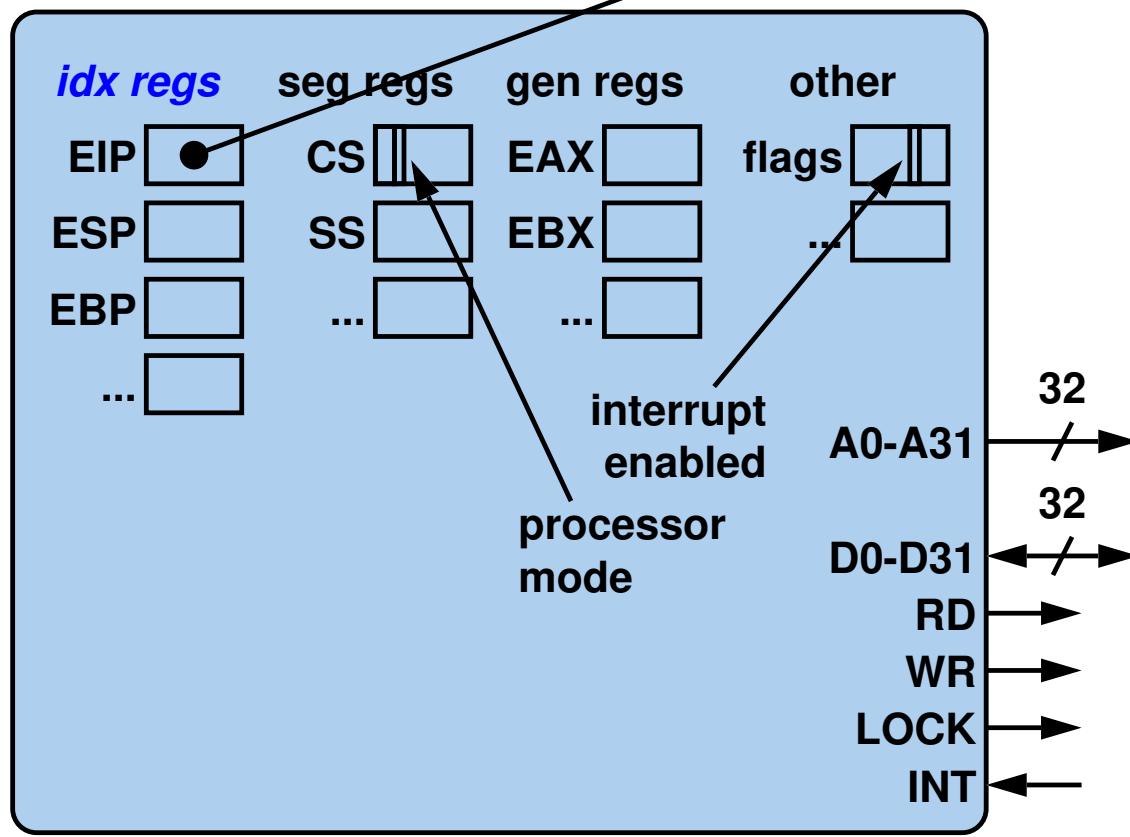
x86 Processor



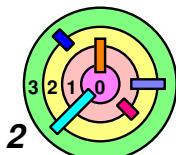
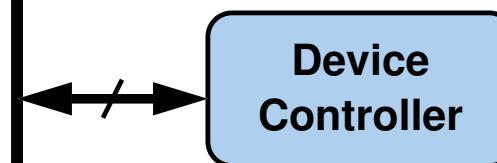
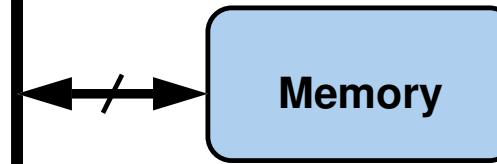
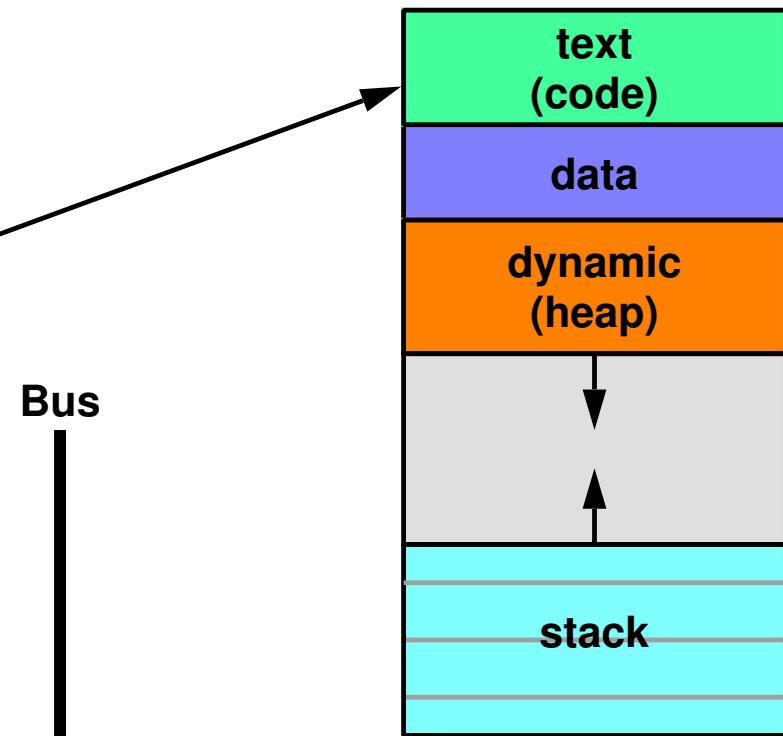
# A Simple OS Structure

Review of "Computer Organization"

x86 Processor



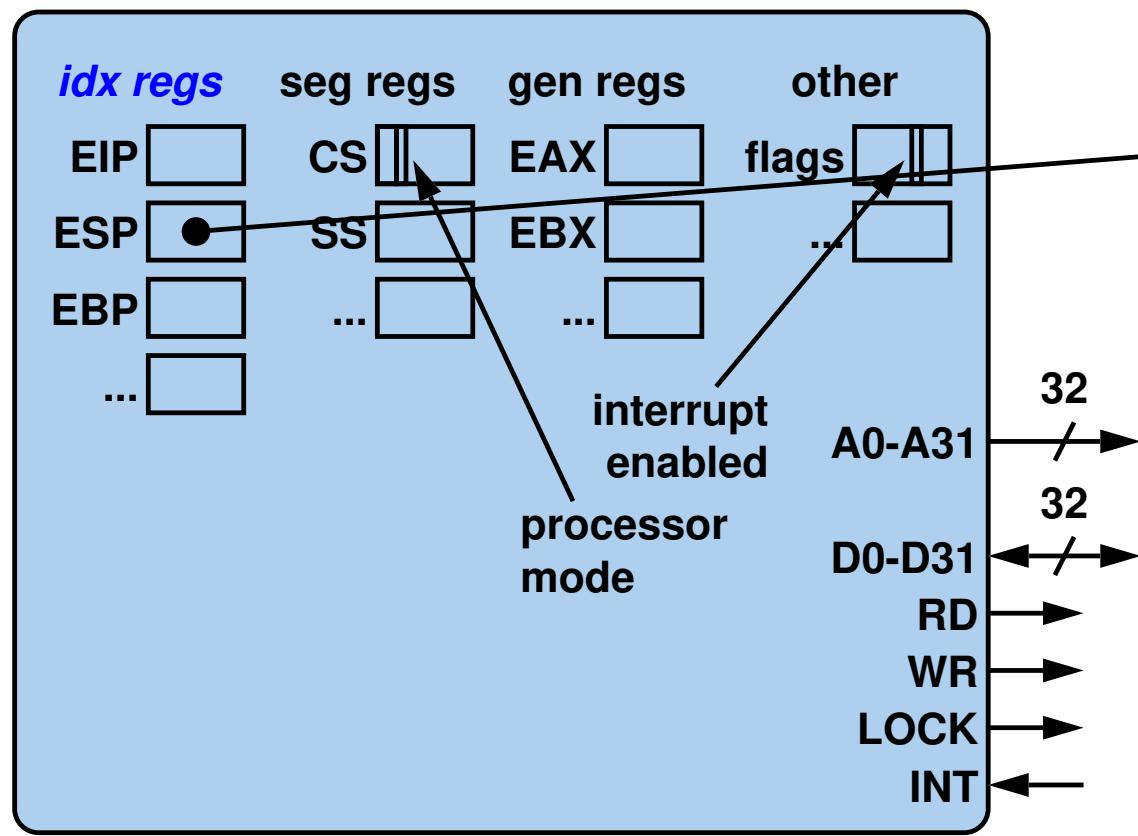
Bus



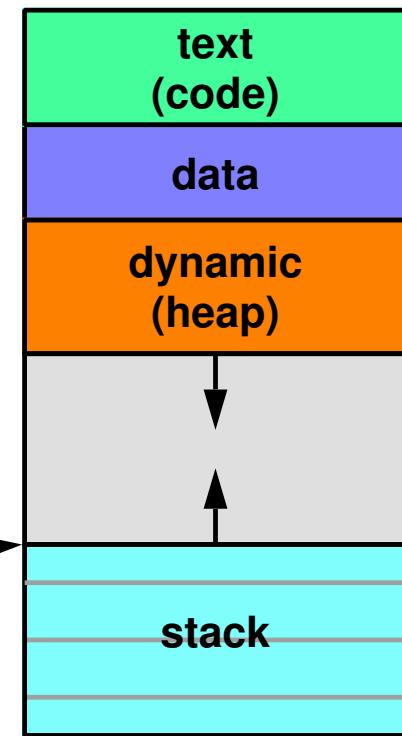
# A Simple OS Structure

Review of "Computer Organization"

x86 Processor



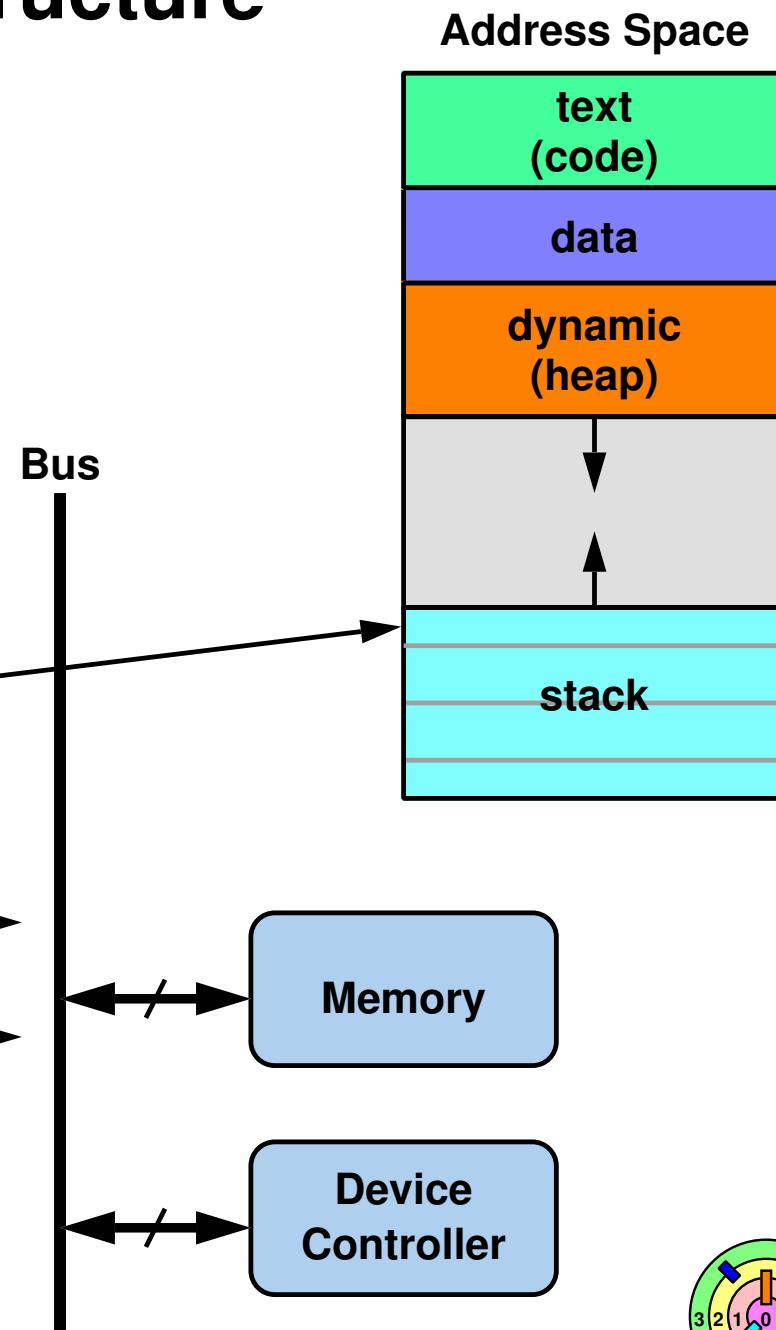
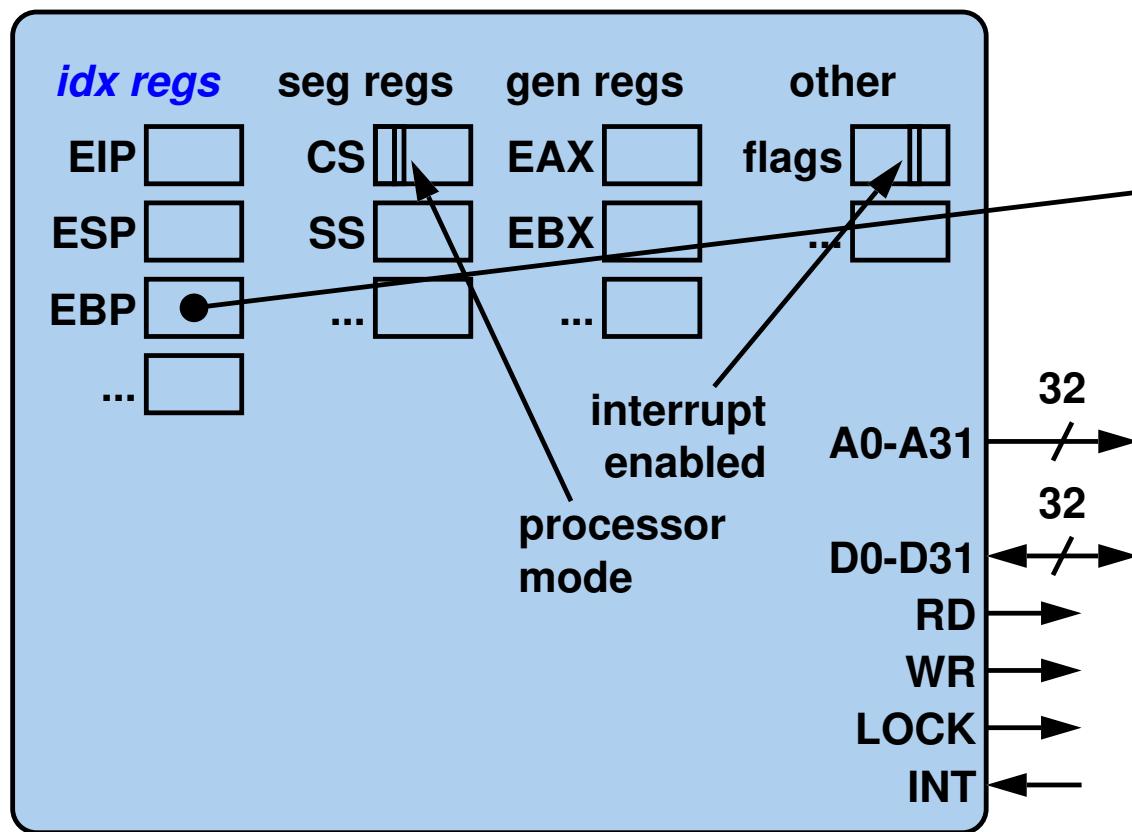
Address Space



# A Simple OS Structure

Review of "Computer Organization"

x86 Processor



# A Simple OS Structure

## Review of "Computer Organization"

$z = x + y$

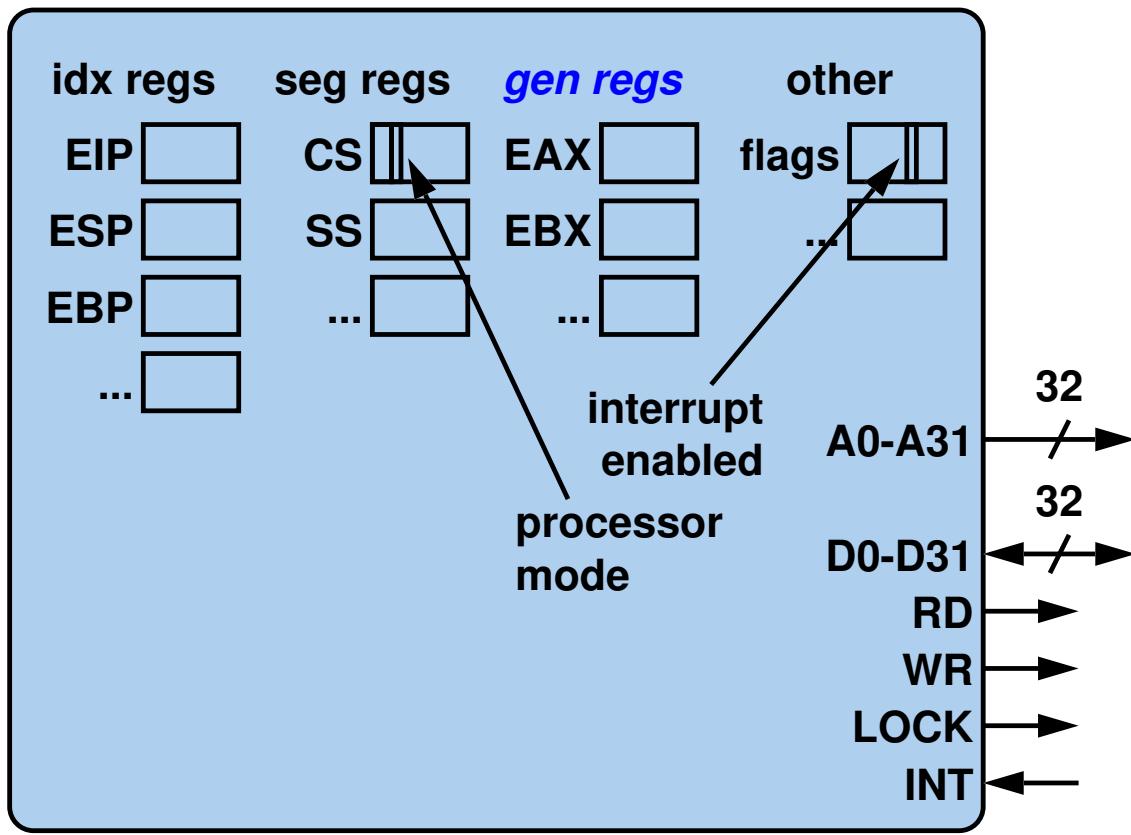


```

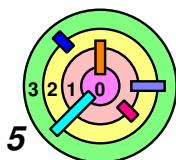
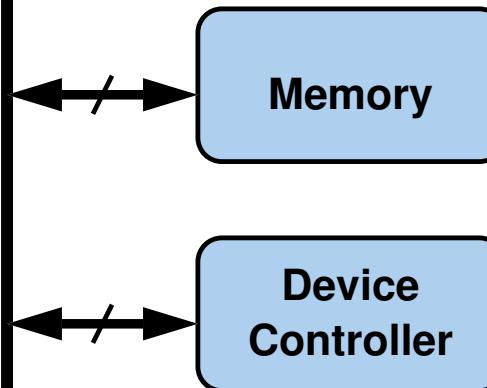
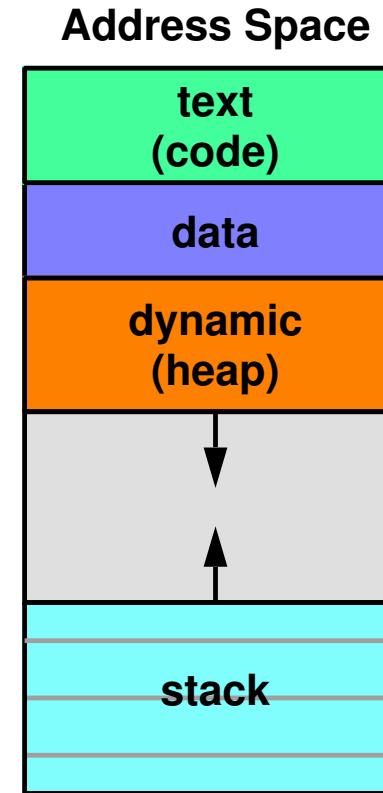
mov &x → eax
mov &y → ebx
add(eax, ebx)
mov eax → &z

```

x86 Processor



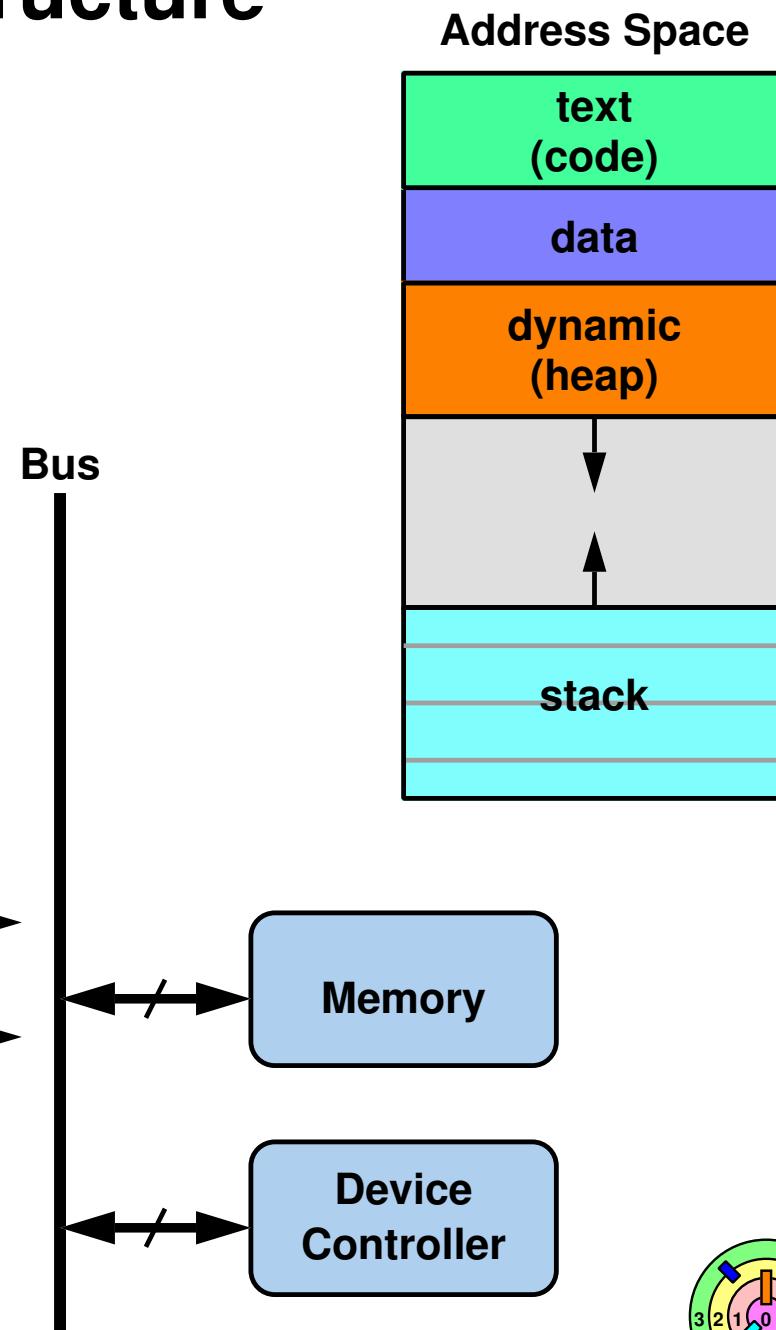
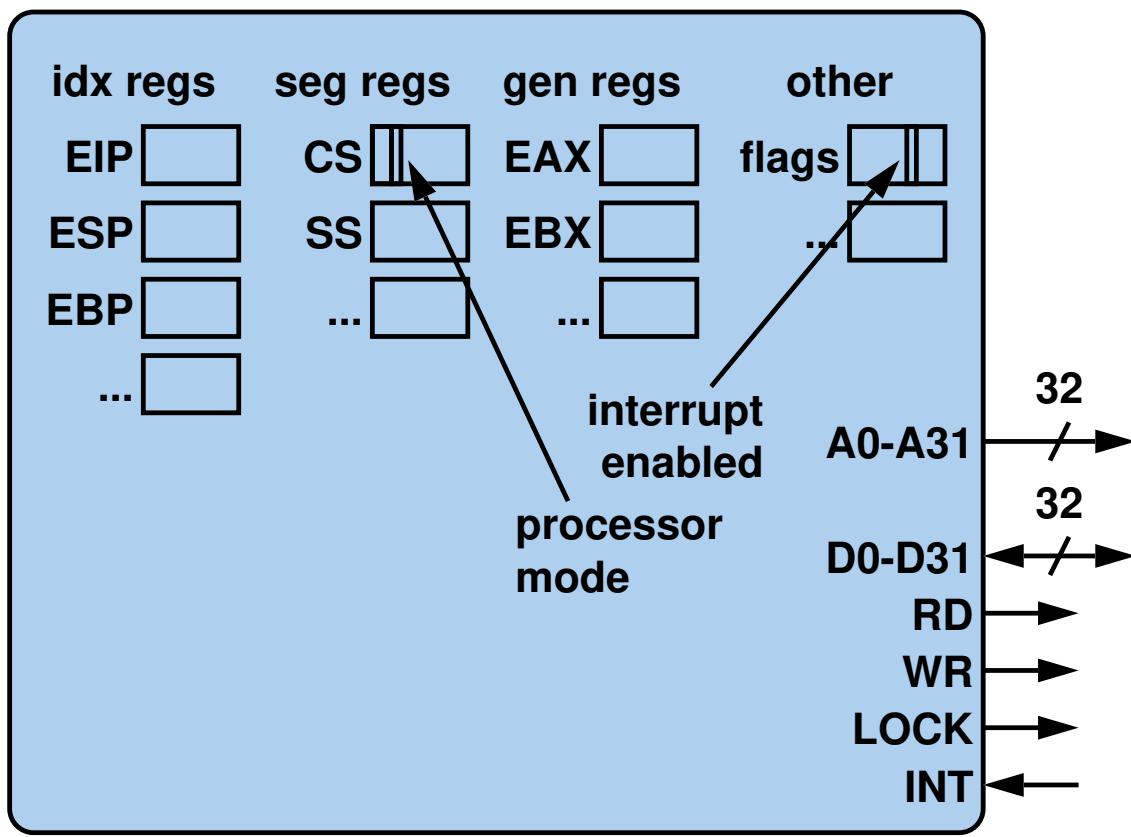
Bus



# A Simple OS Structure

Review of "Computer Organization"

x86 Processor



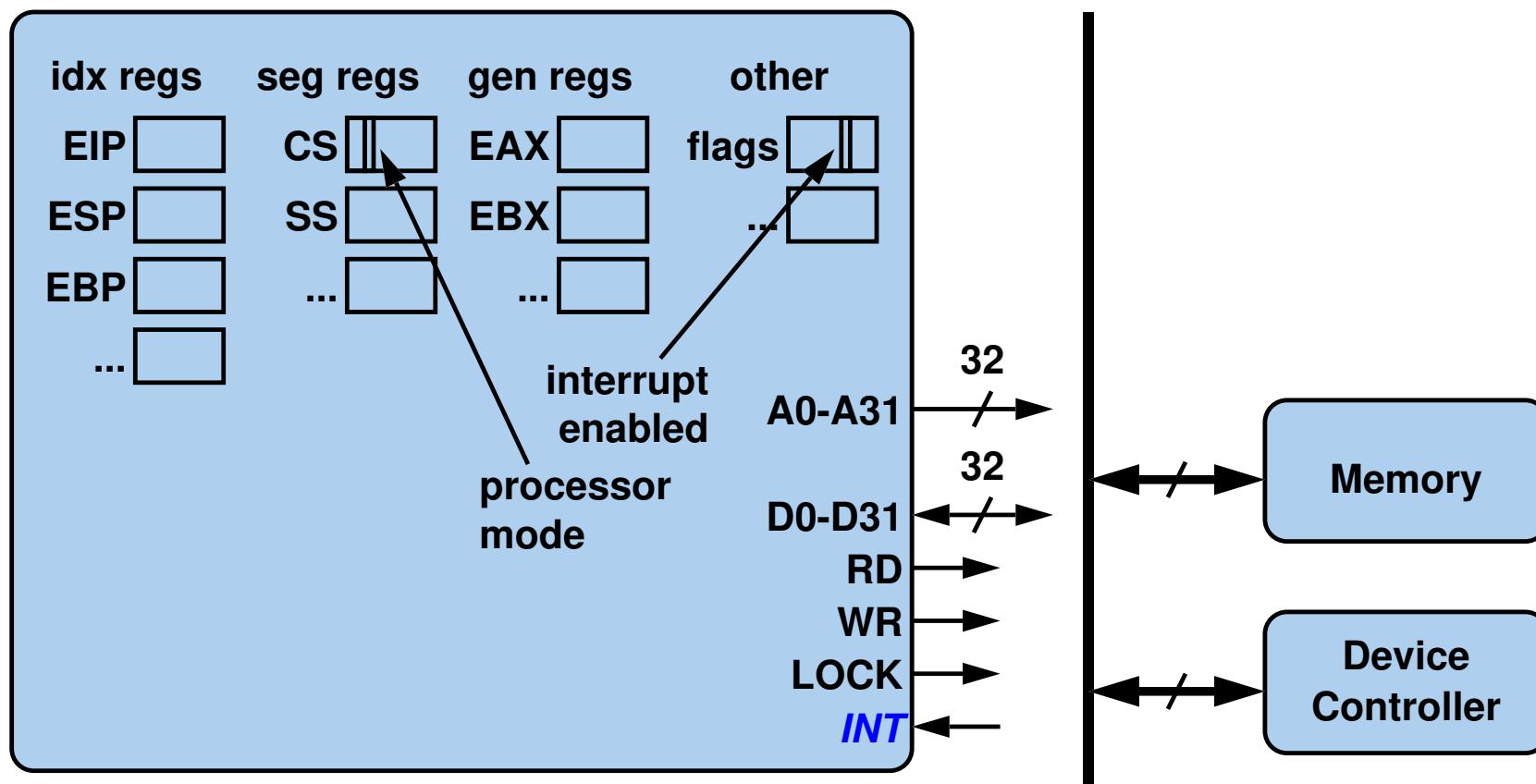
# A Simple OS Structure



Some important terms:

- = *interrupt pending*
- = *interrupt context*
- = *interrupt delivery*
- = *thread context*

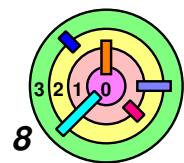
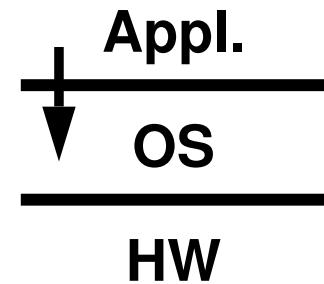
x86 Processor



su21-den-Q8

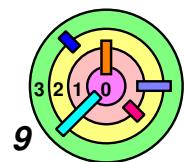
# Traps

- ▶ **Traps** are the general means for invoking the kernel from user code
  - although we usually think of traps as **errors**
    - divide by zero, segmentation fault, bus error, etc.
  - but they don't have to be
    - **system calls, page fault**, etc.
- ▶ Traps always elicit some sort of response
  - for programming errors, the default action is to **terminate** the user program
  - for system calls, the OS is asked to perform some service
  - for page faults, the OS need to fix the virtual memory map



# A Special Kind Of Trap - **System Calls**

- Invoking OS functionality in the kernel is more complex
  - but we want to make it look simple to applications
  - must be done carefully and correctly
    - really cannot trust the application programmers to do the right thing every time
- Provide **system calls** through which user code can access the kernel *in a controlled manner*
  - any necessary checking on whether the request should be permitted can be done in the system call
    - all done in user mode
  - if all goes well
    - sets things up
    - **traps** into the kernel by executing a special machine instruction, i.e., the "trap" machine instruction
    - the kernel figures out why it was invoked and handles the trap
  - more in Ch 3



# Interrupts

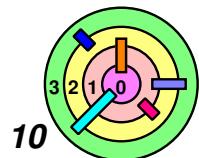
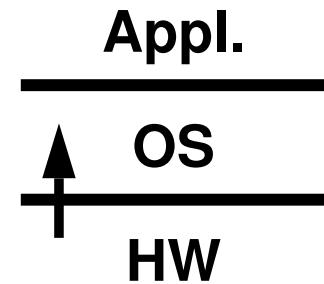


An ***interrupt*** is a request from an ***external device*** for a response from the ***processor***

- most ***hardware interrupts*** are ***I/O completion interrupts***

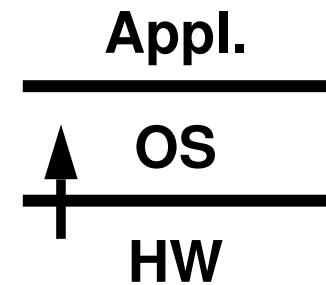
- an I/O device is telling the CPU, "I am done" (and "what do you want me to do next?")
- I/O devices are also hardware, they can run in parallel with the CPU, don't keep them idle unless you have nothing for them to work on

- ***interrupts*** are handled ***independently of any user program***
- unlike a ***trap***, which is handled ***as part of the program*** that caused the trap where response to a trap directly affects that program
- response to an interrupt may or may not indirectly affect the currently running program
- often has ***no direct effect*** on the currently running program



# Interrupts

- An **interrupt** is an **asynchronous** event
  - it's asynchronous with respect to the executing entity (threads or OS)
- A **trap** occurs **synchronously** with respect to the executing entity
  - when your thread executes a divide-by-zero instruction, we know exactly where it happens and we know when it will happen

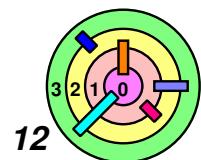
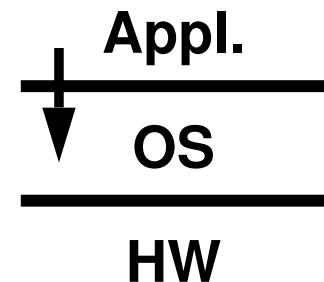


# Software Interrupt



There's also something called ***software interrupt***

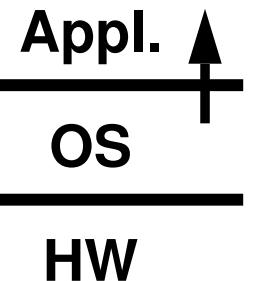
- generated programmatically (i.e., not by a device) when executing a machine instruction
  - e.g., executing an "interrupt" machine instruction
  - x86 CPU uses a software interrupt (i.e., "int 0x2e") to implement the "trap" machine instruction
    - ◊ other CPUs may have a separate "trap" machine instruction
- this is very different from a **hardware interrupt**
  - although the mechanisms of handling interrupts are all very similar as we will see in Ch 3



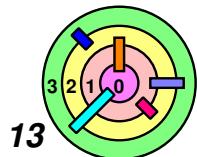
F20-Q10 S21-Q11

# Upcall

su21-a-Q15

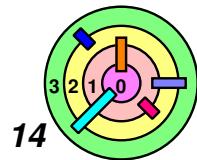


- A program may establish a handler (i.e., a **signal handler**) to be invoked in response to the error
  - the handler might clean up after the error and then terminate the program, or it might perform corrective action and continue with normal execution
  - more in Ch 2
  
- The **upcall** mechanism
  - **signals** allow the kernel to invoke code that's part of user program
    - for example, you can set a timer to expire at a certain time, when it expires, the OS can use the upcall mechanism to call a specified user function on behalf of the user program



# 1.3 A Simple OS

- ➔ OS Structure
- ➔ *Processes, Address Spaces, & Threads*
- ➔ Managing Processes
- ➔ Loading Program Into Processes
- ➔ Files



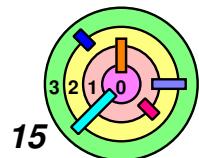
# Program Execution

## → Fundamental *abstraction* of *program execution*

- memory
  - address space
    - ◊ things that are addressable by the program are kept together here
  - in Sixth-Edition Unix, processes do not share address space
  - recall that *process* is an abstraction of *memory*
- processor(s)
  - recall that *thread* is an abstraction of *processor*
- "execution context"
  - which represents the *state* of a process and its threads
  - represents exactly "where you are" in the program
  - a thread needs some sort of a context to execute

## → Note: multiple meanings of the word "context" in this class

- *save (execution) context* and *restore (execution) context*
- *thread context* vs. *interrupt context*



# A Program

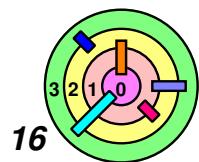
```

const int nprimes = 100;
int prime[nprimes];
int main() {
    int i;
    int current = 2;
    prime[0] = current;
    for (i=1; i<nprimes; i++) {
        int j;
        NewCandidate:
        current++;
        for (j=0; prime[j]*prime[j] <= current; j++) {
            if (current % prime[j] == 0)
                goto NewCandidate;
        }
        prime[i] = current;
    }
    return(0);
}

```



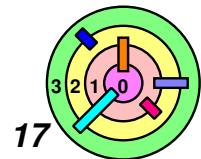
- My color codes for code**
- reserved words at in **blue**
  - numeric and string constants are in **red**
  - comments in **green**
  - black otherwise



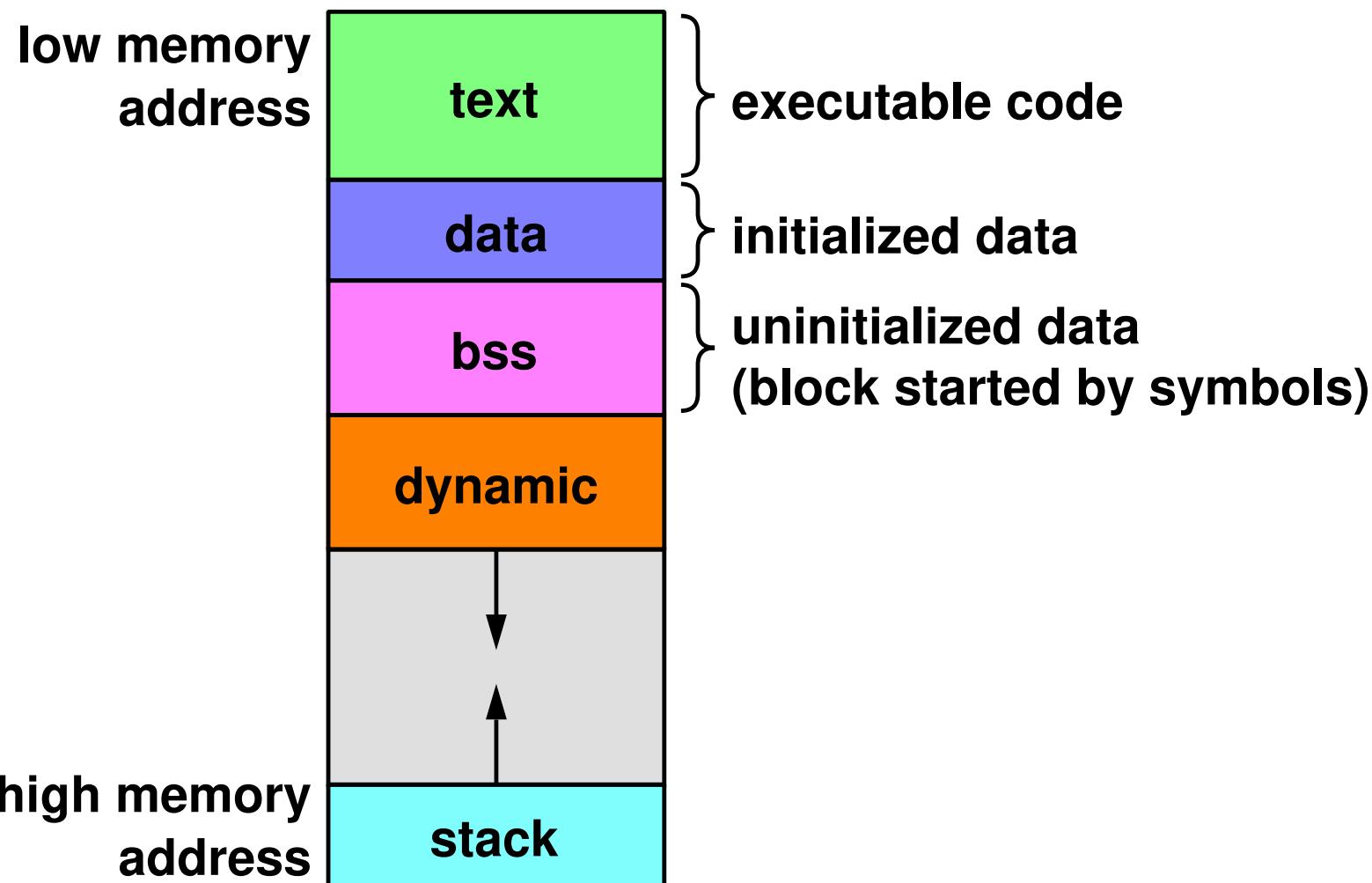
# Turing Machine Model of Computation



- A **Turing Machine** consists of
  - an infinite tape which is divided into cells, one next to the other (*i.e.*, **infinite storage**)
    - one symbol in each cell (or can be a blank symbol)
  - a head that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time
  - a state register that stores the state of the Turing machine, one of finitely many (*i.e.*, **finite state**)
  - a **finite** table of instructions that, given the state the machine is currently in and the symbol it is reading on the tape tells the machine to do the following in sequence
    - either erase or write a symbol
    - move the head
    - assume the same or a new state as prescribed



# The Unix Address Space



- This is part of the *tape* of the Turing Machine
  - the rest of the *tape* of the Turing Machine can be reached by using the "*extended address space*"

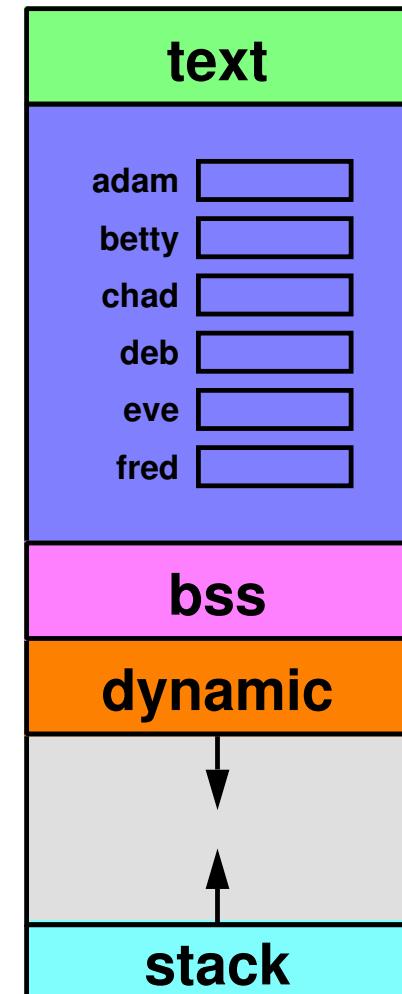
# Note About Naming Objects

→ How do you name objects in an address space

- = “**objects**” is the word we use to mean **any** data types (primitive, data structures, pointers)

→ Variables

- = name each object
- = a **variable** refers to a ***memory location***



# Note About Naming Objects

→ How do you name objects in an address space

- “**objects**” is the word we use to mean **any** data types (primitive, data structures, pointers)

→ Variables

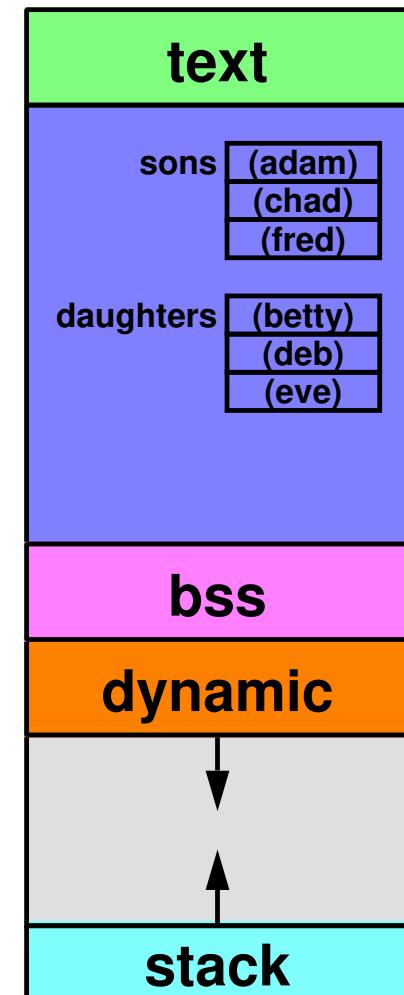
- name each object
- a **variable** refers to a **memory location**

→ Arrays

- name an object with a **base** and an **index**

→ Dynamically create objects do not have names

- no variable can have a “heap address”
- need **pointers**



# Note About Naming Objects

→ How do you name objects in an address space

- “**objects**” is the word we use to mean **any** data types (primitive, data structures, pointers)

→ Variables

- name each object
- a **variable** refers to a ***memory location***

→ Arrays

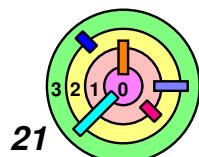
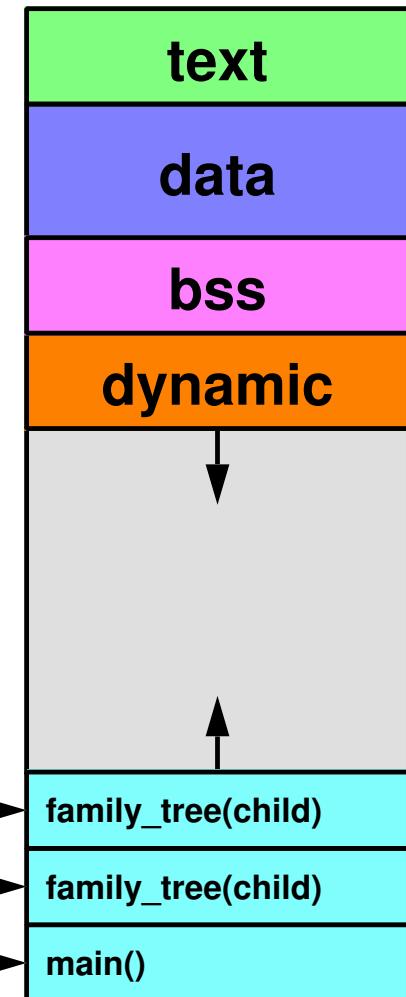
- name an object with a **base** and an **index**

→ Dynamically create objects do not have names

- no variable can have a “heap address”
- need **pointers**

→ For objects that lives in the stack, same name is used for different object instances

- **function arguments** and **local variables**



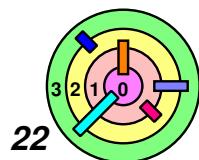
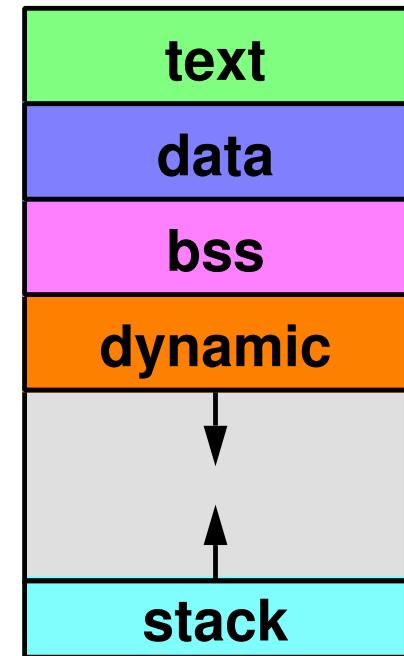
# Modified Program

```

int nprimes;                      // in bss region
int *prime;                        // in bss region
int main(int argc, char *argv[]) { // in stack
    int i;                          // in stack
    int current = 2;                // in stack
    nprimes = atoi(argv[1]);
    prime = (int*)malloc(nprimes*sizeof(int));
    prime[0] = current;
    for (i=1; i<nprimes; i++) {
        ...
    }
    return(0);
}

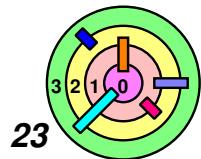
```

- where do all the variables reside?
- what is argv[1] and why atoi () ?
- what is sizeof () ?
- what does malloc () do?



# 1.3 A Simple OS

- ▶ OS Structure
- ▶ Processes, Address Spaces, & Threads
- ▶ *Managing Processes*
- ▶ Loading Program Into Processes
- ▶ Files



S21-Q13

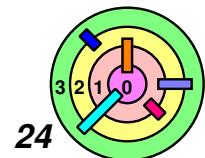
# Program Execution



With abstraction, comes an interface / API

— for processes

- `fork()`, `exit()`, `wait()`, `exec()`
- ◊ it's very important to understand *what they do exactly* because you will implement them in kernel assignments



# Creating a Process



Creating a process is deceptively simple

- make a copy of a process (the parent process)
  - `pid_t fork(void)`
  - the process where `fork()` is called is the **parent** process
  - the copy is the **child** process
  - in a way, `fork()` returns twice
    - ◊ once in the parent, the returned value is the **process ID (PID)** of the child process
    - ◊ once in the child, the returned value is 0
    - ◊ a PID is 16-bit long
- this is the **only** way to **create a process**

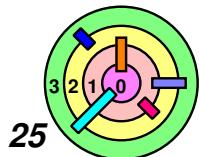


Making a copy of the entire address space can be expensive

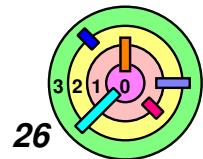
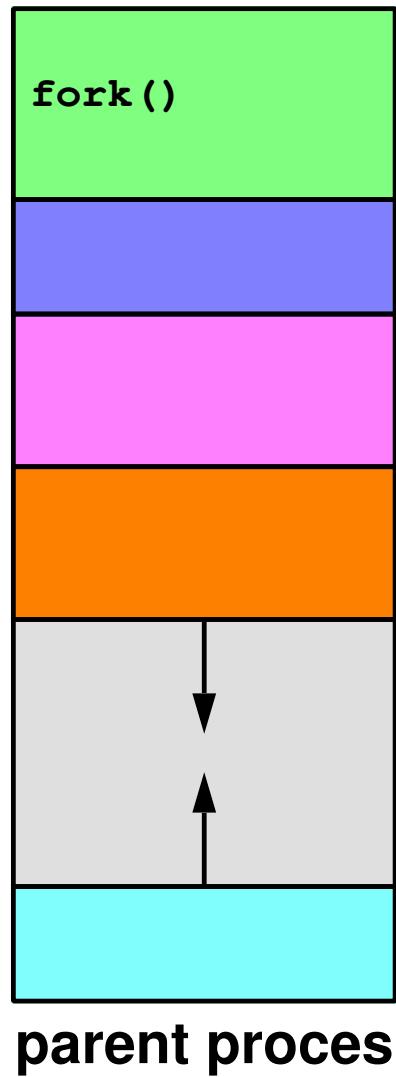
- Ch 7 shows speed up tricks
- e.g., text segment is read-only so parent and child can share it



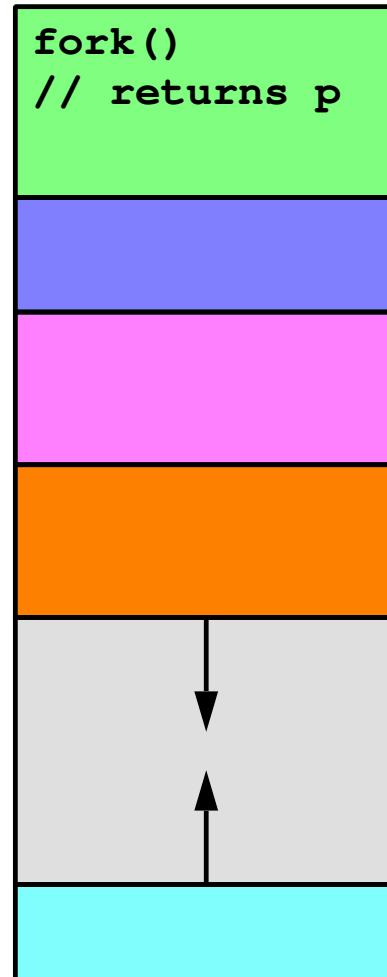
Example: relationship between a shell (i.e., a command interpreter, such as `/bin/tcsh`) and `/bin/ls`



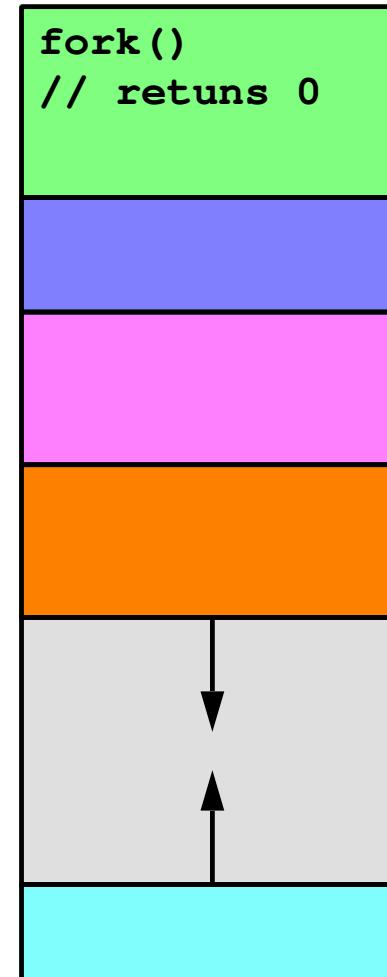
# Creating a Process: Before



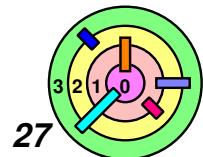
# Creating a Process: After



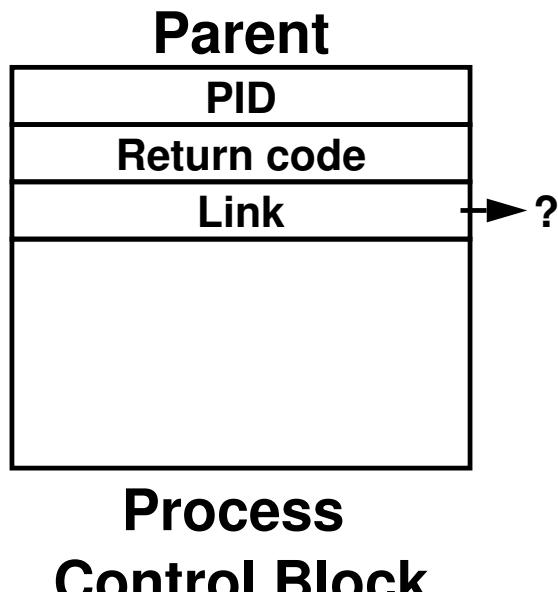
parent process



child process  
(pid = p)

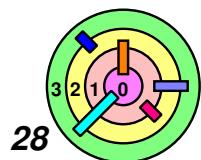


# Process Control Blocks

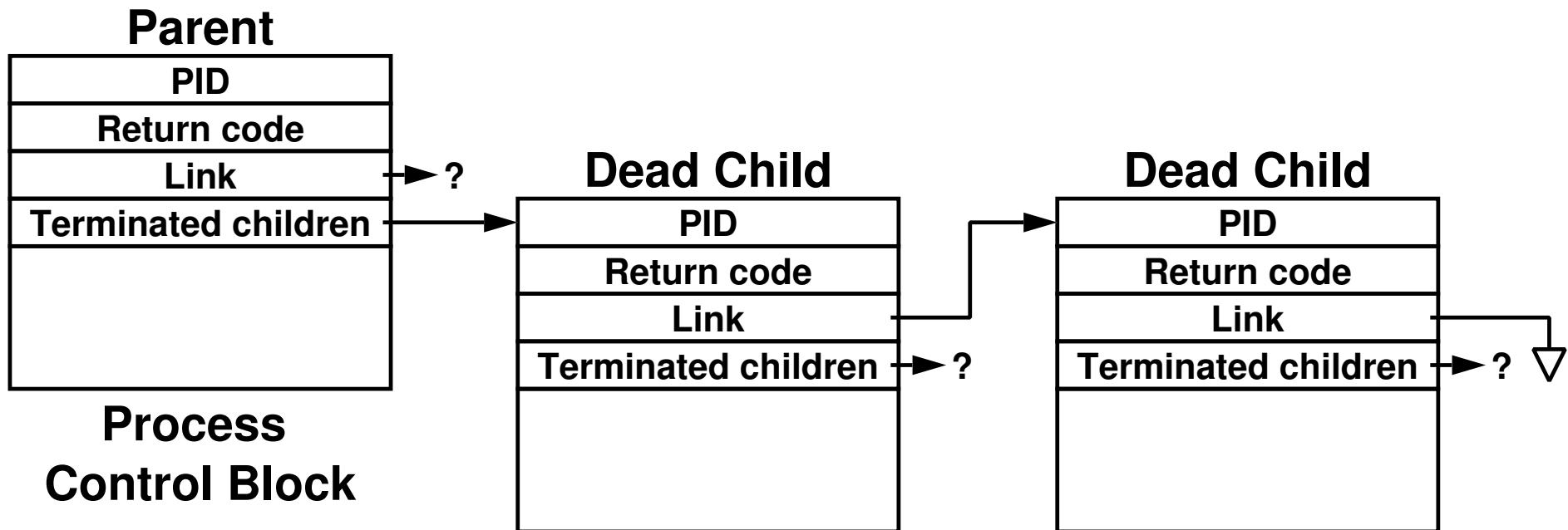


→ **Process Control Block (PCB)** is a kernel data structure

- pretty much every field is unsigned
- return code (when a process dies) is 8-bit long
  - so that the parent process can know what happened to child
- the "Link" field points to the next PCB
  - but, the next PCB in what list?



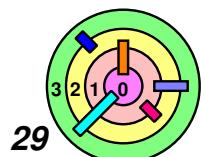
# Process Control Blocks



→ **Process Control Block (PCB)** is a kernel data structure

- pretty much every field is unsigned
- return code (when a process dies) is 8-bit long
  - so that the parent process can know what happened to child
- the "Link" field points to the next PCB
  - but, the next PCB in what list?

→ Above is **not** a real implementation (just an example)

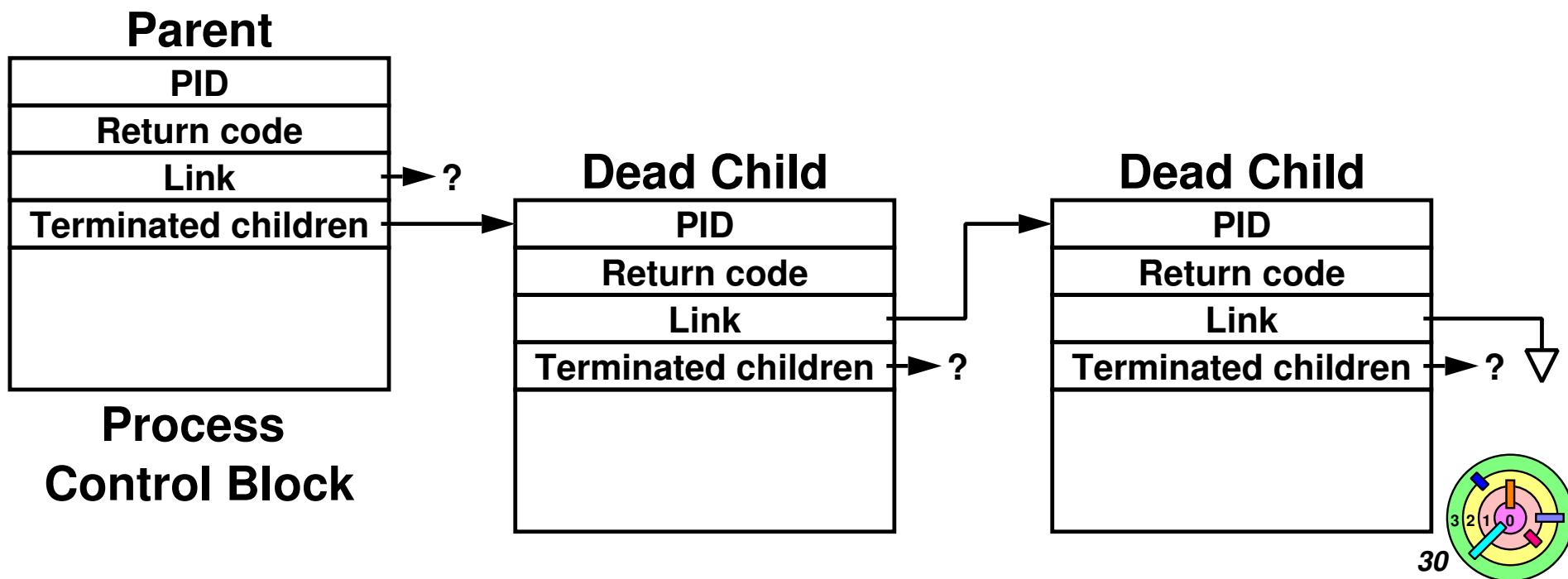


# The `exit()` System Calls

## → The `exit()` system call

```
void exit(int status)
```

- your process can call `exit(n)` to **self-terminate**
  - set `n` to be the "exit/return code" of this process
  - this system call does not return (your process will die inside the kernel)



# The `exit()` System Calls

## → The `exit()` system call

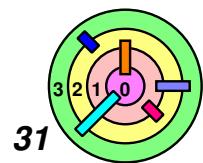
```
void exit(int status)
```

- your process can call `exit(n)` to self-terminate
  - set `n` to be the "exit/return code" of this process
  - this system call does not return (your process will die inside the kernel)

## → Where does the "primes" program go after it executes the "`return(0)`"?

- it returns to a "startup" function
- the code of the "startup" function is simply:

```
exit(main());
```

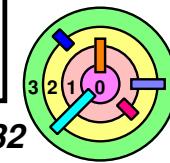
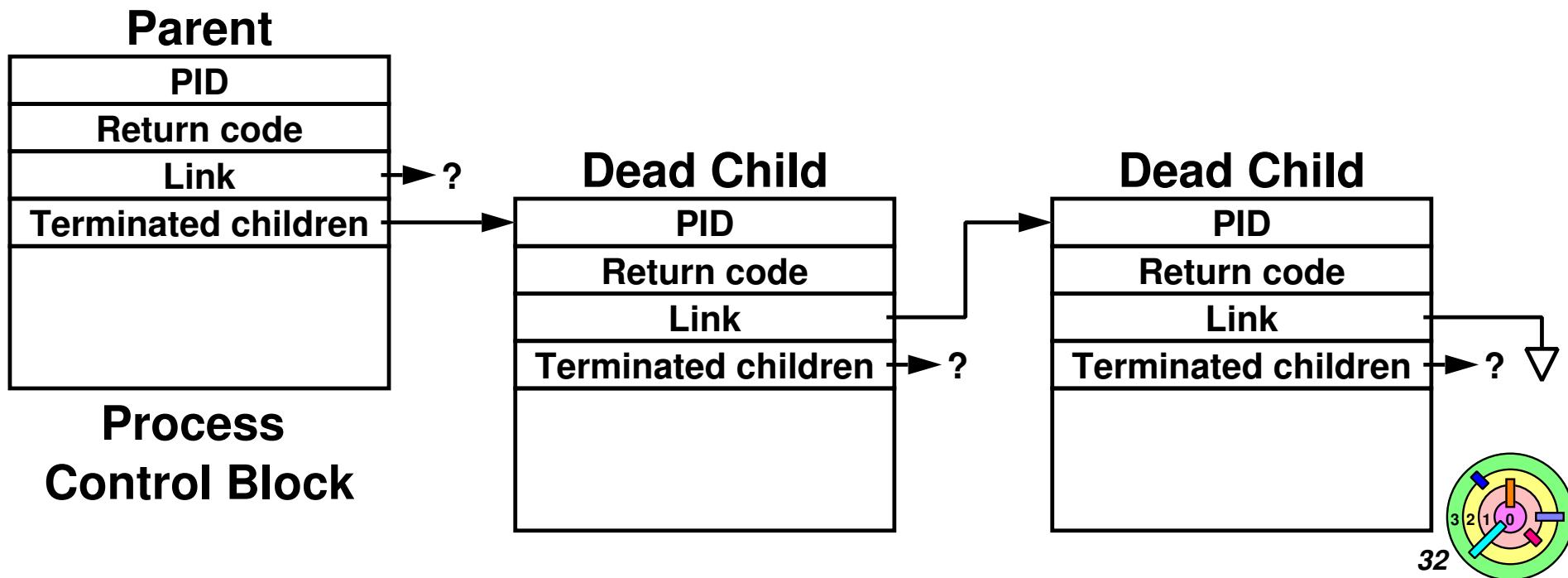


# The wait () System Calls

## → The wait () system call

```
pid_t wait(int *status)
```

- your process can call `wait()` to **wait for *any* child process to die**
  - returns the PID of a dead child process where `(*status)` is the exit/return code of the corresponding child process
    - ◊ if there are more than one dead child processes, one of them will be *chosen at random*

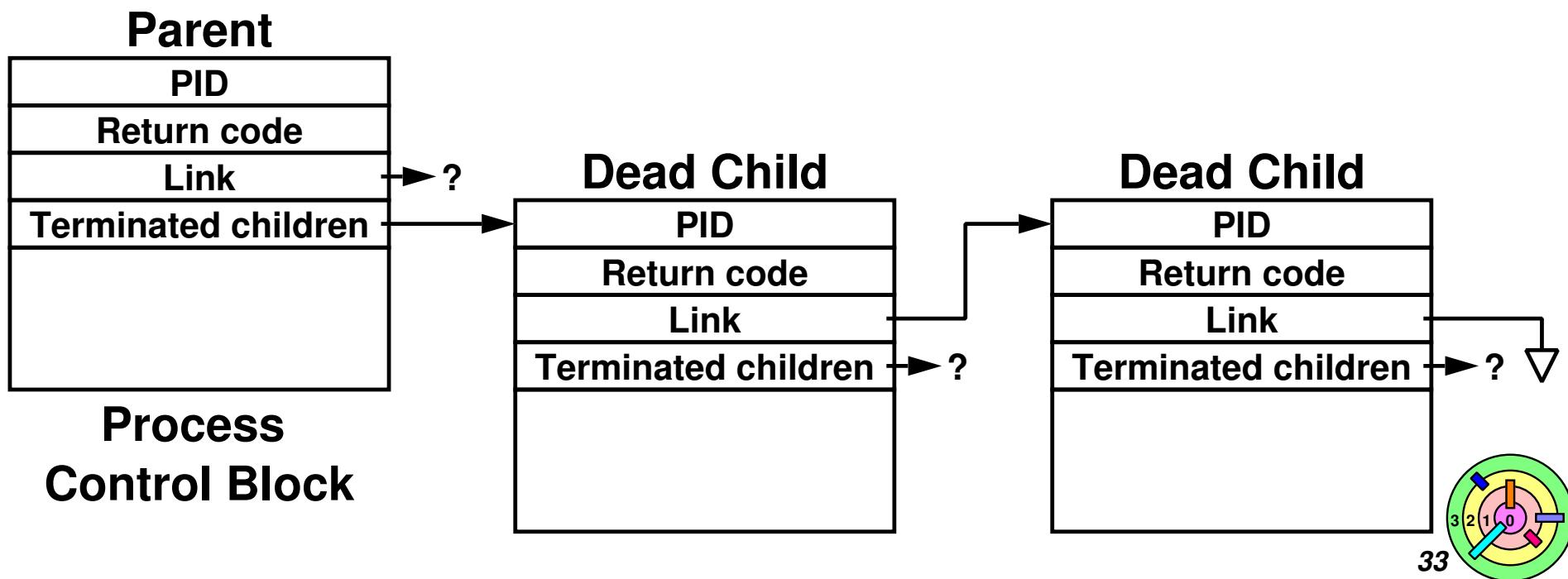


# The wait() System Calls

## → The wait() system call

```
pid_t wait(int *status)
```

- your process can call `wait()` to wait for *any* child process to die
  - it's a *blocking call*, i.e., the calling process gets suspended inside the kernel if this call cannot return yet



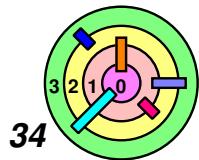
# Fork and Wait

```

short pid;
if ((pid = fork()) == 0) {
    /* some code is here for the child to execute */
    exit(n);
} else {
    int ReturnCode;
    while(pid != wait(&ReturnCode))
        ;
    /* the child has terminated with ReturnCode as
       its return code */
}

```

- e.g., this is the first step when /bin/tcsh forks /bin/ls
- what does `exit (n)` do other than copying `n` into PCB?
  - least significant 8-bits of `n`
- what happens when `main ()` calls `return (n)` ?
  - eventually, `exit (n)` will be invoked
- `pid_t wait(int *status)` is a ***blocking call***
  - it reaps dead child processes ***one at a time***
- parent and child are the same "program" here!



# Fork and Wait

```

short pid;
if ((pid = fork()) == 0) {
    /* some code is here for the child to execute */
    exit(n);
} else {
    int ReturnCode;
    while(pid != wait(&ReturnCode))
        ;
    /* the child has terminated with ReturnCode as
       its return code */
}

```

**Parent**

PID
Return code
Link
Terminated children

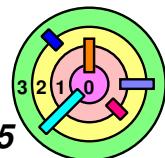
**Process  
Control Block**

**Dead Child**

PID
Return code
Link
Terminated children

**Dead Child**

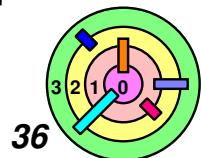
PID
Return code
Link
Terminated children



# Fork and Wait

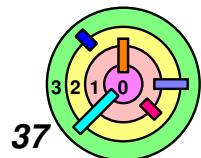
```
short pid;
if ((pid = fork()) == 0) {
    /* some code is here for the child to execute */
    exit(n);
} else {
    int ReturnCode;
    while(pid != wait(&ReturnCode))
        ;
    /* the child has terminated with ReturnCode as
       its return code */
}
```

- What if you don't want to write your code this way?
- you can write any code you want, you just shouldn't expect your code to work if you write weird code
  - you need to understand exactly what these system calls do and use them appropriately
    - if you do something weird, the OS will try to satisfy your request, but may end up with results you don't expect



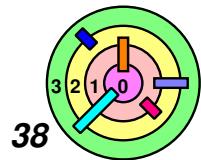
# Process Termination Issues

- ➡ PID is only 16-bits long
  - OS must not reuse PID too quickly or there may be ambiguity
- ➡ When `exit()` is called, the OS must not free up PCB too quickly
  - parent needs to get the return code
  - it's okay to free up everything else (such as address space)
- ➡ Solutions for both is for the terminated child process to go into a **zombie state** S21-Q5 su21-den-Q18
  - only after `wait()` returned with the child's PID can the PID be reused and the PCB can be freed up
  - but what if the parent calls `exit()` while the child is in the zombie state?
    - process 1 (the process with PID=1) inherits all the children of this parent process
      - ◊ this is known as "*reparenting*"
    - process 1 keeps calling `wait()` to reap the zombies



# 1.3 A Simple OS

- ▶ OS Structure
- ▶ Processes, Address Spaces, & Threads
- ▶ Managing Processes
- ▶ *Loading Program Into Processes*
- ▶ Files

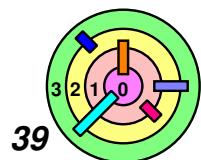


# Loading Programs Into Processes



How do you run a program?

- make a copy of a process
  - any process
- replace the child process with a new one
  - wipe out the child process
    - ◆ not everything, some stuff survives this (i.e., won't get destroyed)
    - ◆ definitely need a *new address space* since we will be running a different program
  - using a family of system calls known as *exec*
- kind of a waste to make a copy in the first place
  - but it's the only way
  - also, the OS does not know if the reason the parent process calls `fork()` is to run a new program or not



# Exec

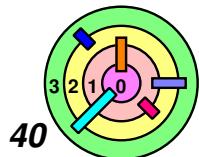
```

int pid;
if ((pid = fork()) == 0) {
    /* we'll discuss what might take place before
       exec is called */
    execl("/home/bc/bin/primes", "primes", "300", 0);
    exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
;

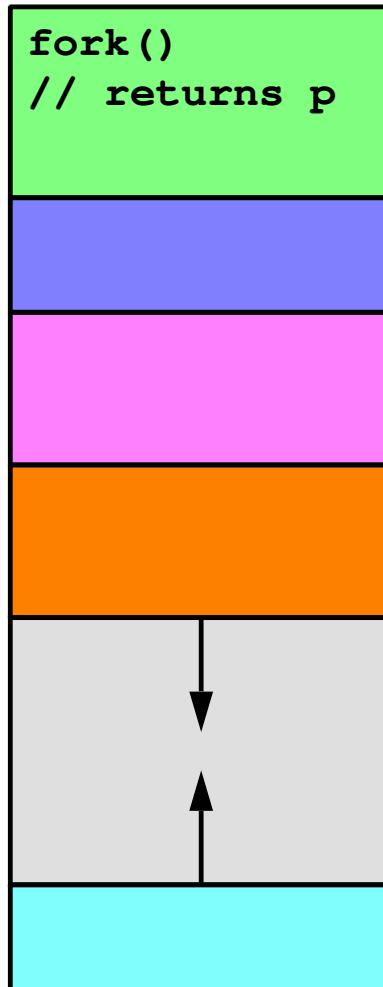
```

- what does `execl()` do?
  - "man execl" says:
 

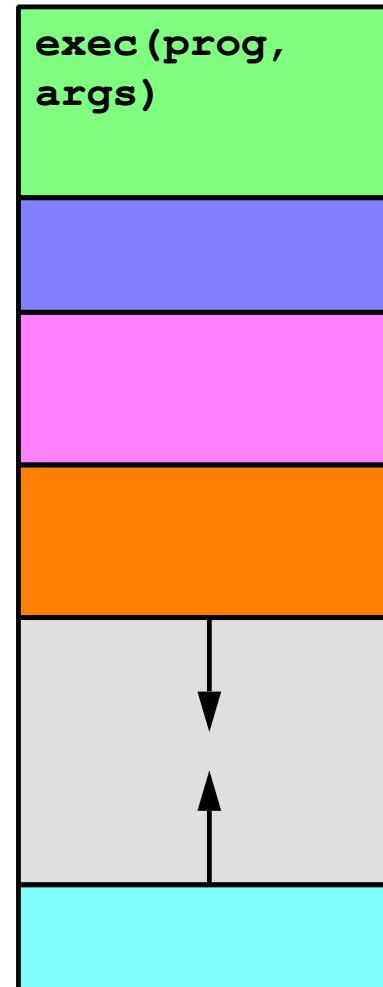
```
int execl(const char *path,
                    const char *arg, ...);
```
  - isn't "primes" in the 2nd argument kind of redundant?
  - what's up with "...)?
    - ◆ this is called "**"varargs"** (similar to `printf()`)



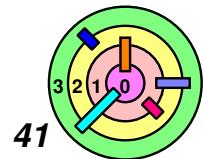
# Loading a New Image



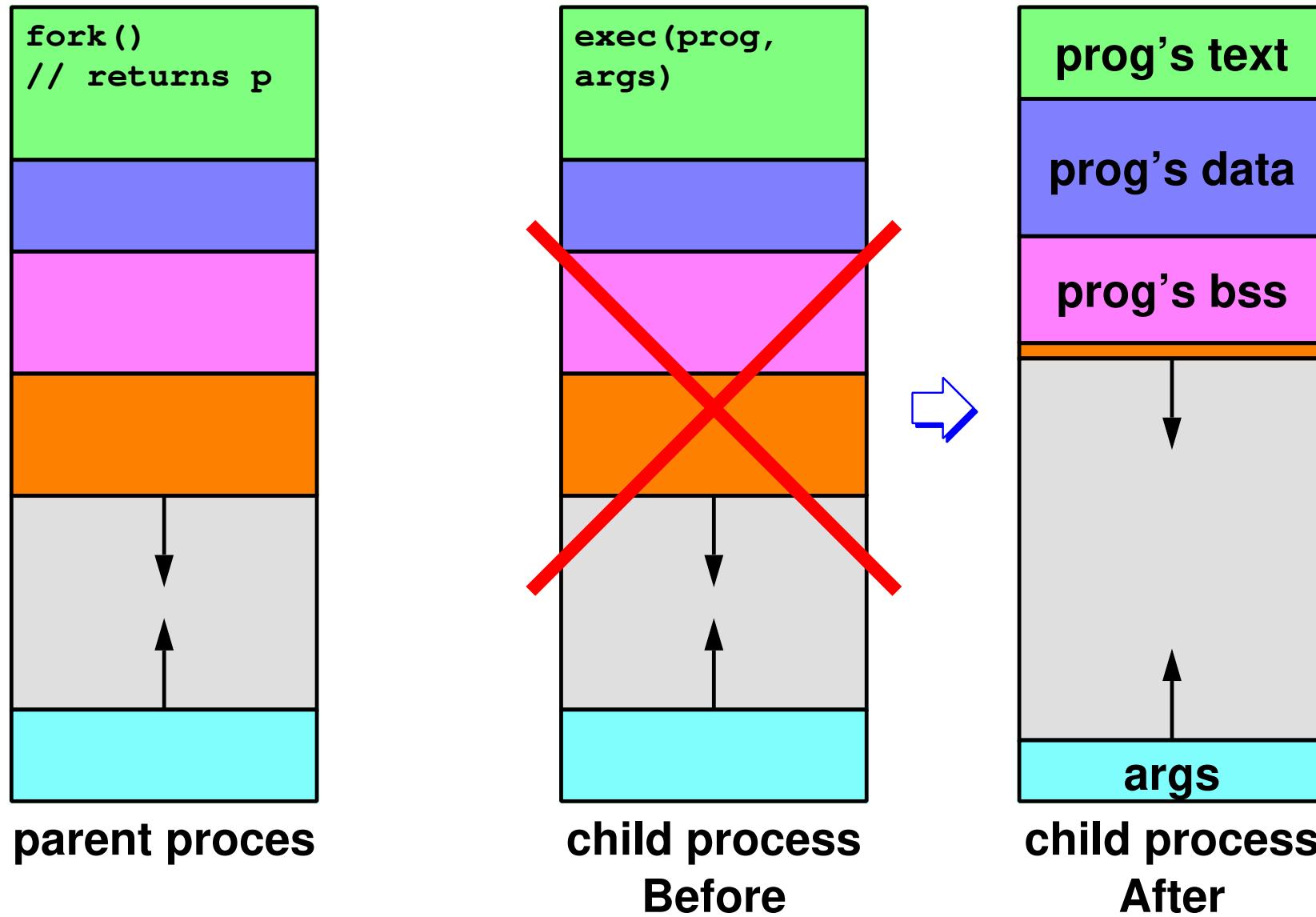
parent process



child process  
Before



# Loading a New Image

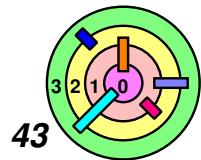


# Exec

```
int pid;
if ((pid = fork()) == 0) {
    execl("/home/bc/bin/primes", "primes", "300", 0);
    exit(1);
}
while(pid != wait(0)) /* ignore the return code */
;
```

```
% primes 300
```

- Your login shell forks off a child process, load the primes program on top of it, wait for the child to terminate
  - the same code as before
  - `exit(1)` would get called if somehow `execl()` returned
    - if `execl()` is successful, it cannot return since the code is **gone** (i.e., the code segment has been replaced by the code segment of "primes")



# Put It All Together

Parent  
(shell)

**fork()**

```
int pid;  
→ if ((pid = fork()) == 0) {  
    execl("/home/bc/bin/primes",  
          "primes", "300", 0);  
    exit(1);  
}  
while(pid != wait(0))  
;
```

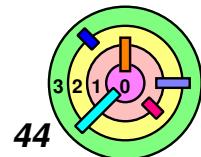
Applications

OS

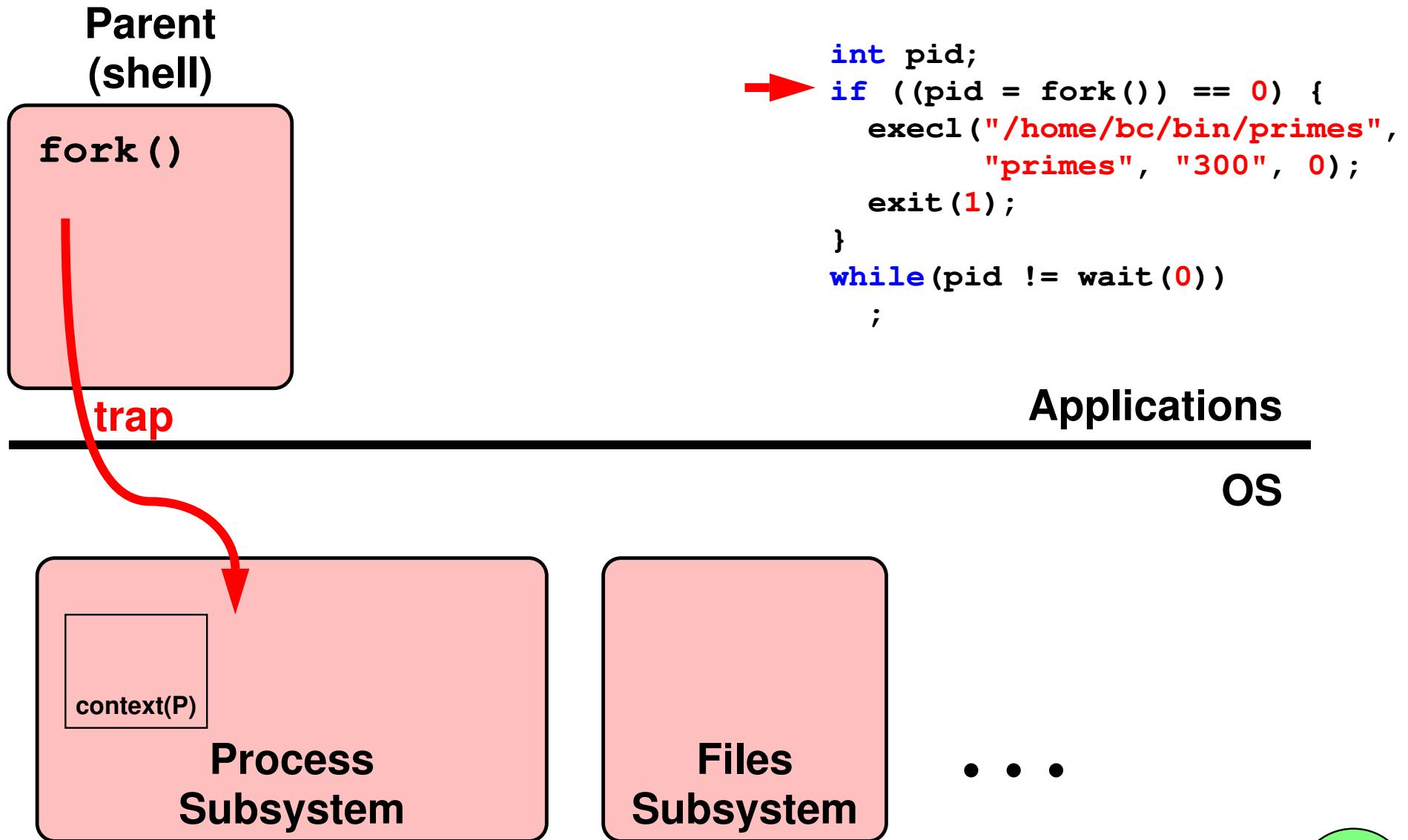
Process  
Subsystem

Files  
Subsystem

• • •

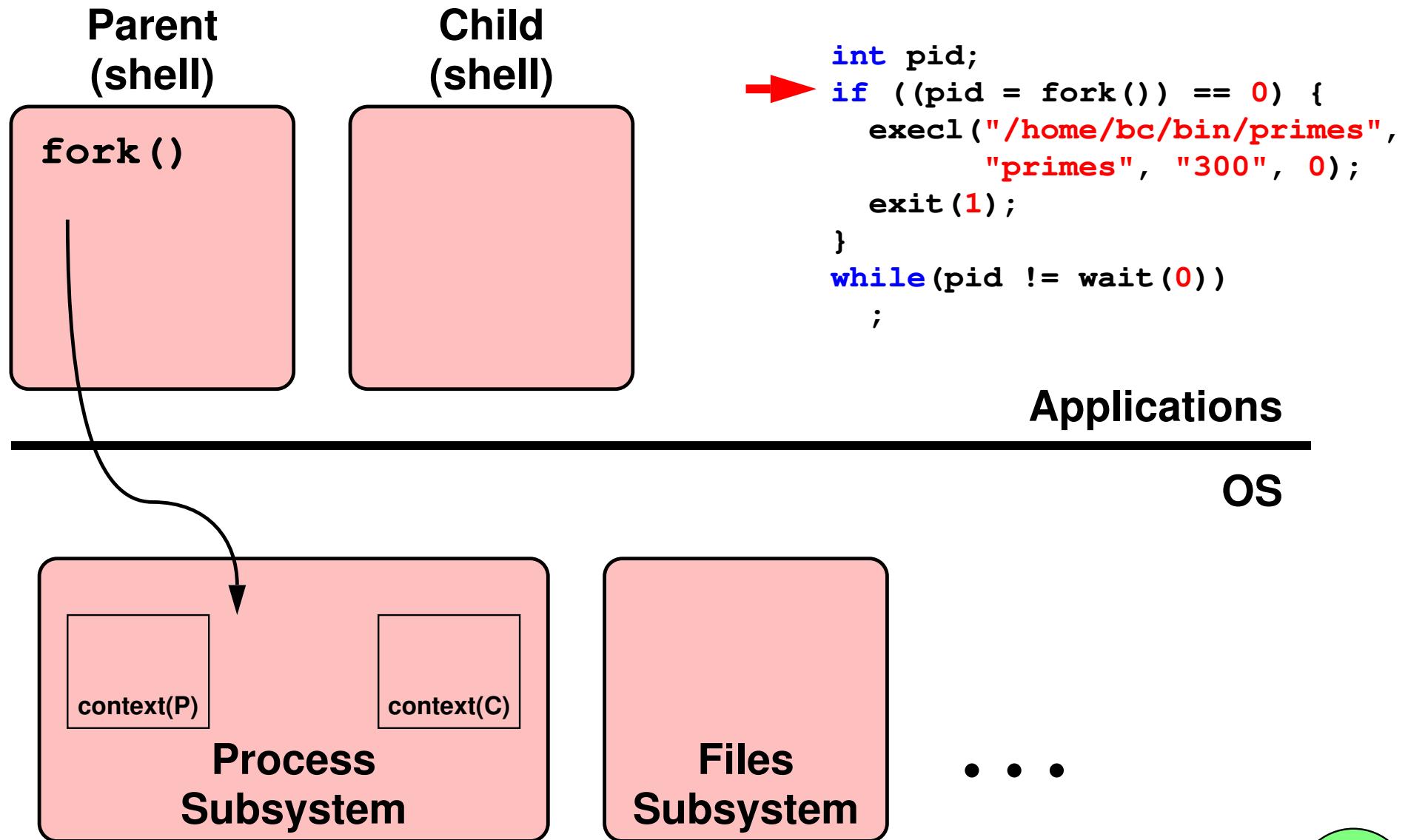


# Put It All Together

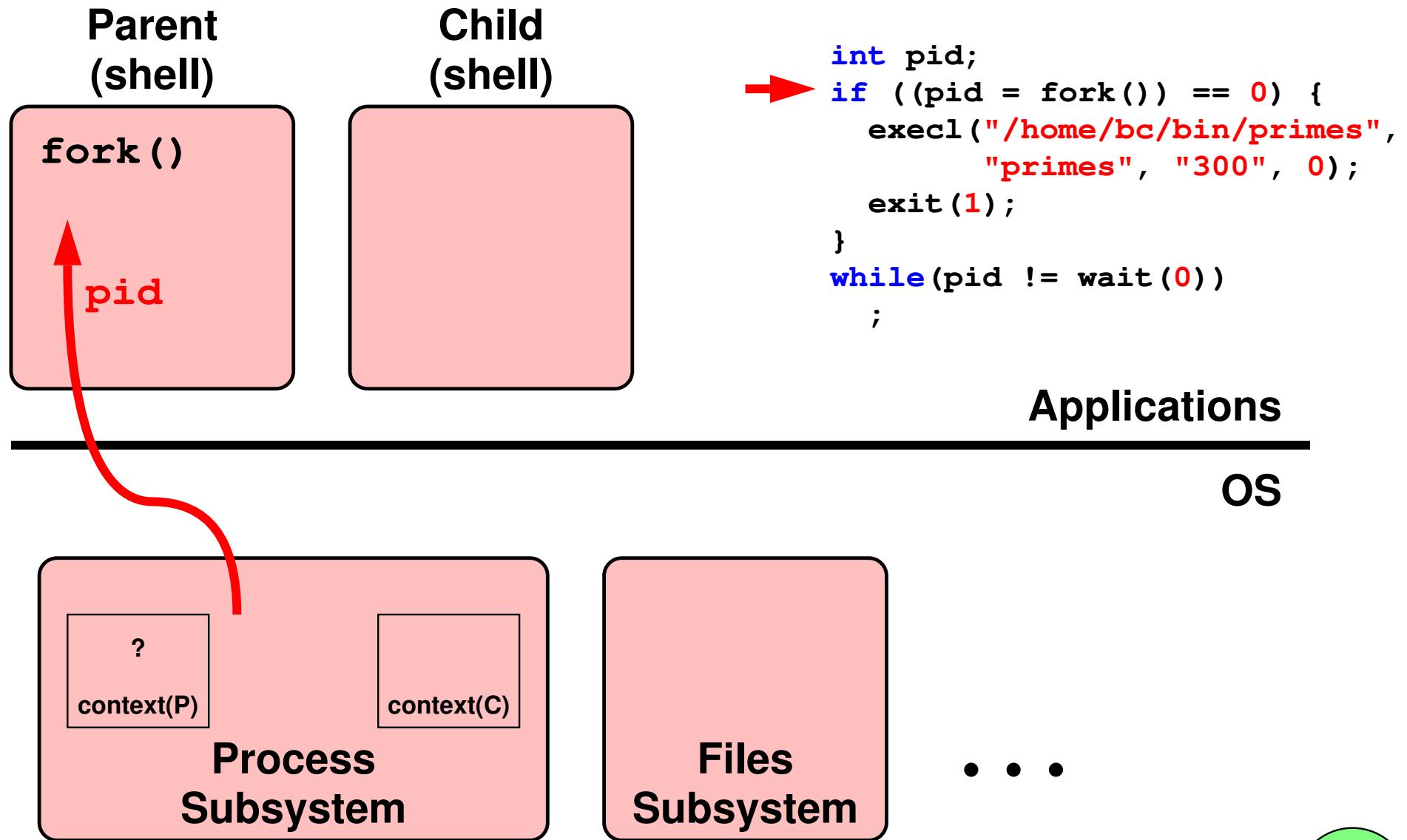


→ Where do you keep "context"?

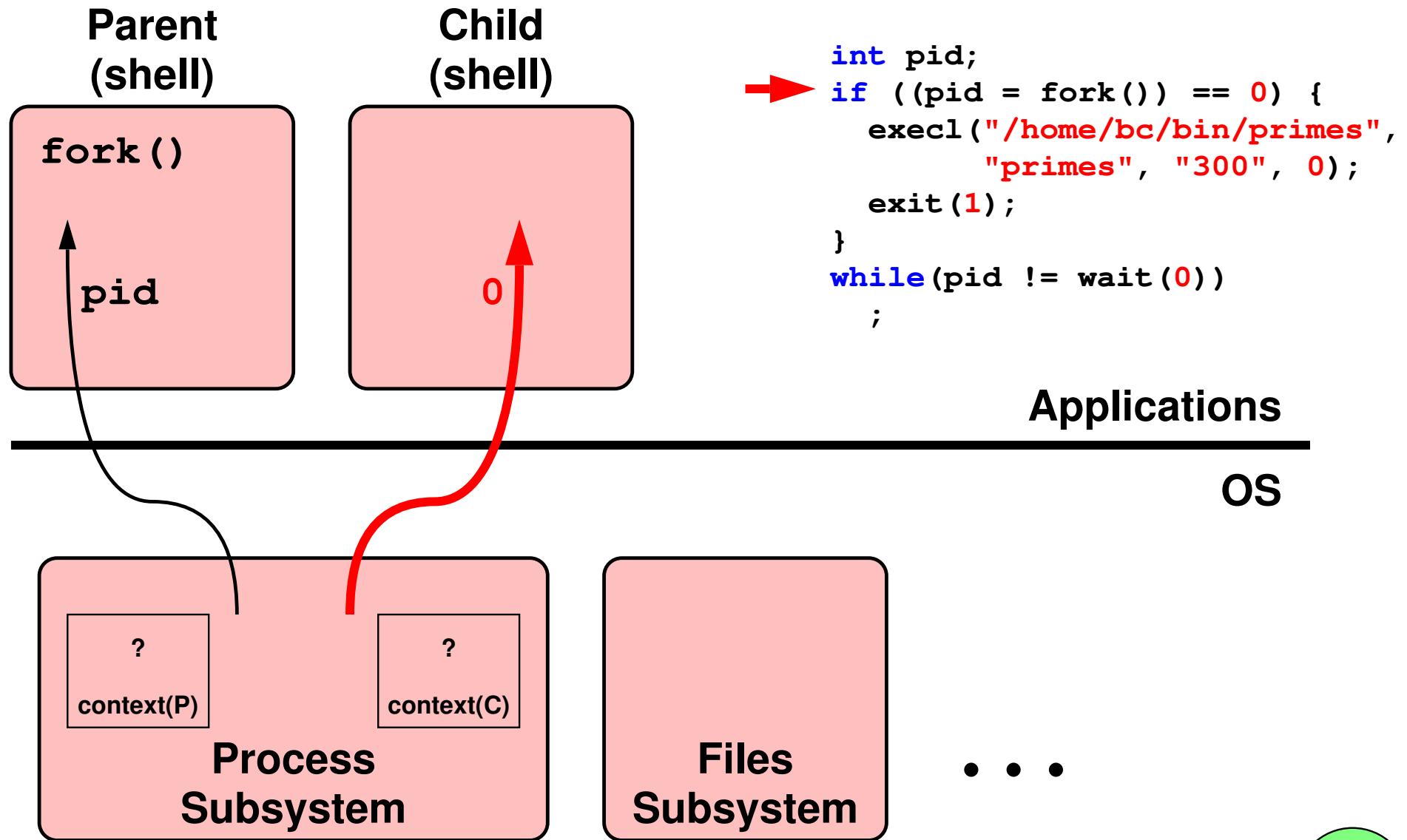
# Put It All Together



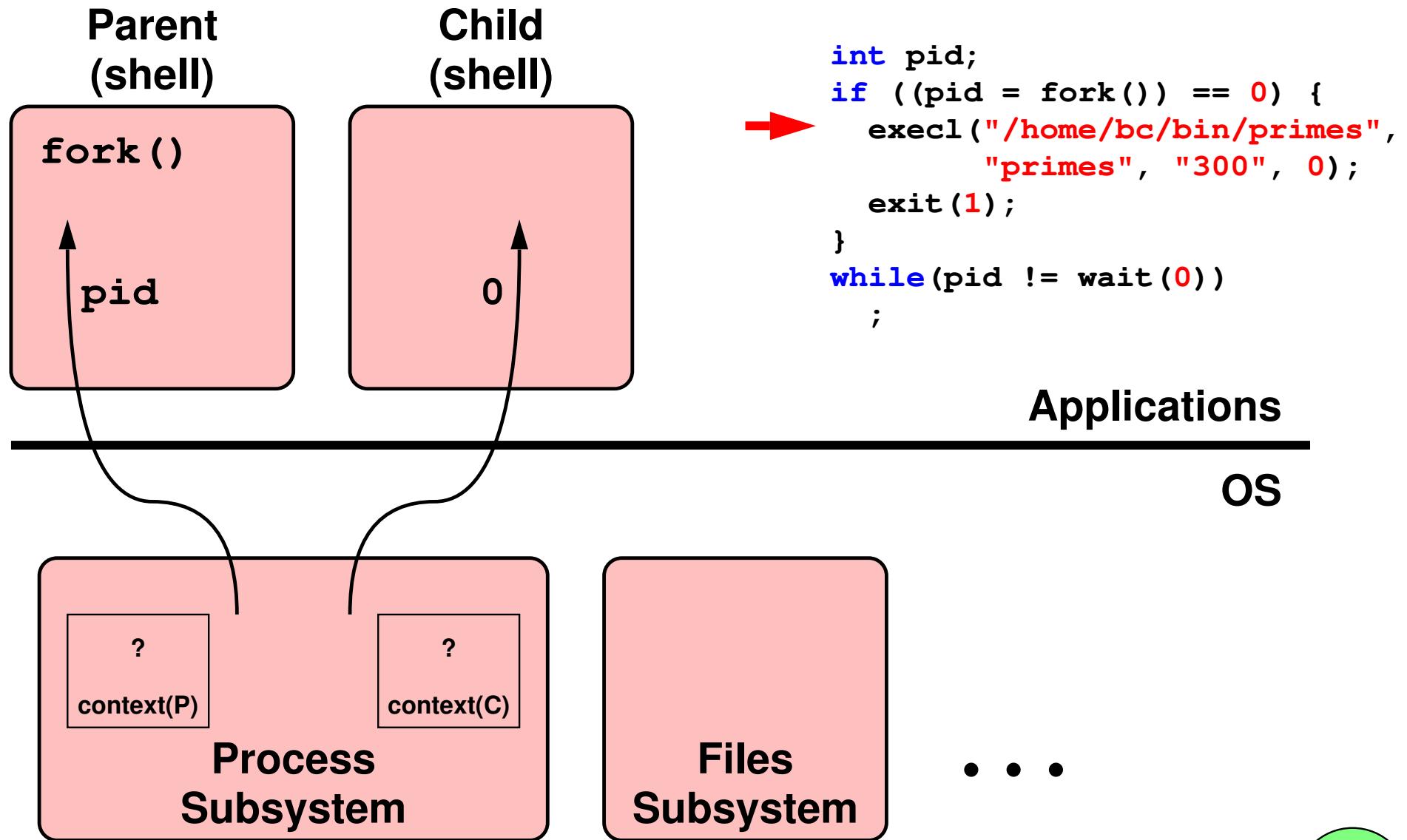
# Put It All Together



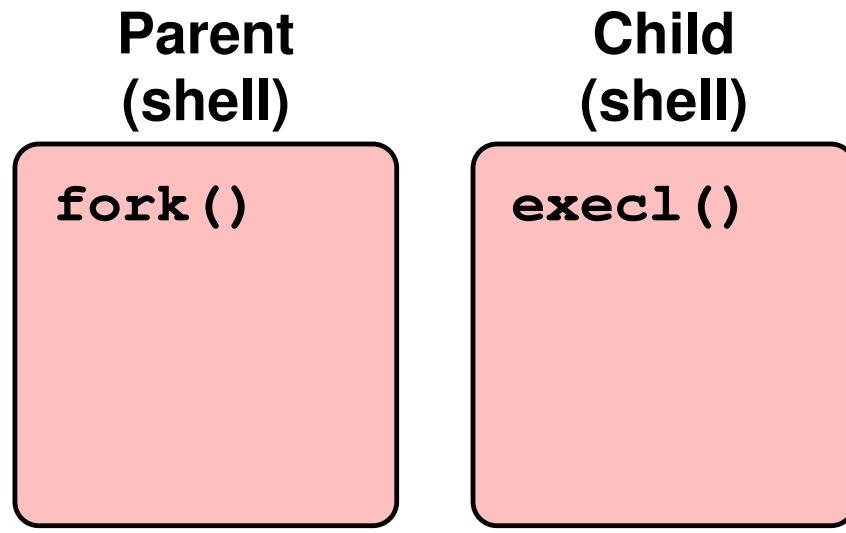
# Put It All Together



# Put It All Together



# Put It All Together



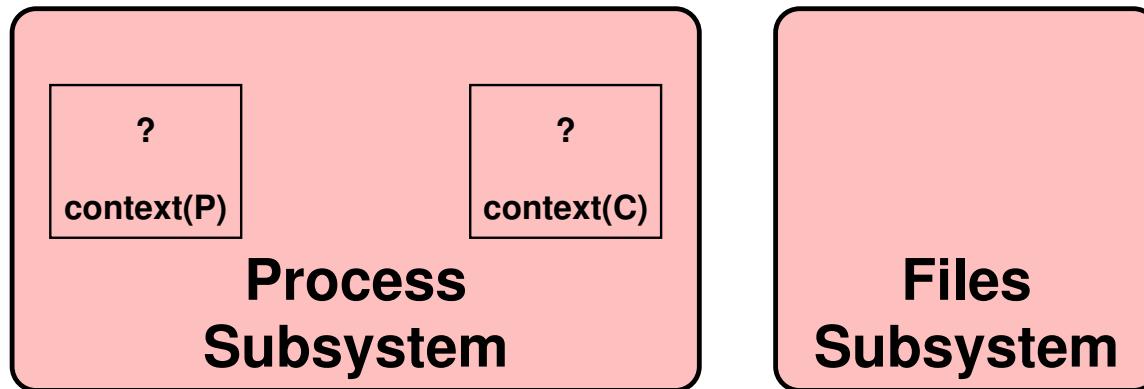
```

int pid;
if ((pid = fork()) == 0) {
    execl("/home/bc/bin/primes",
          "primes", "300", 0);
    exit(1);
}
while (pid != wait(0))
;

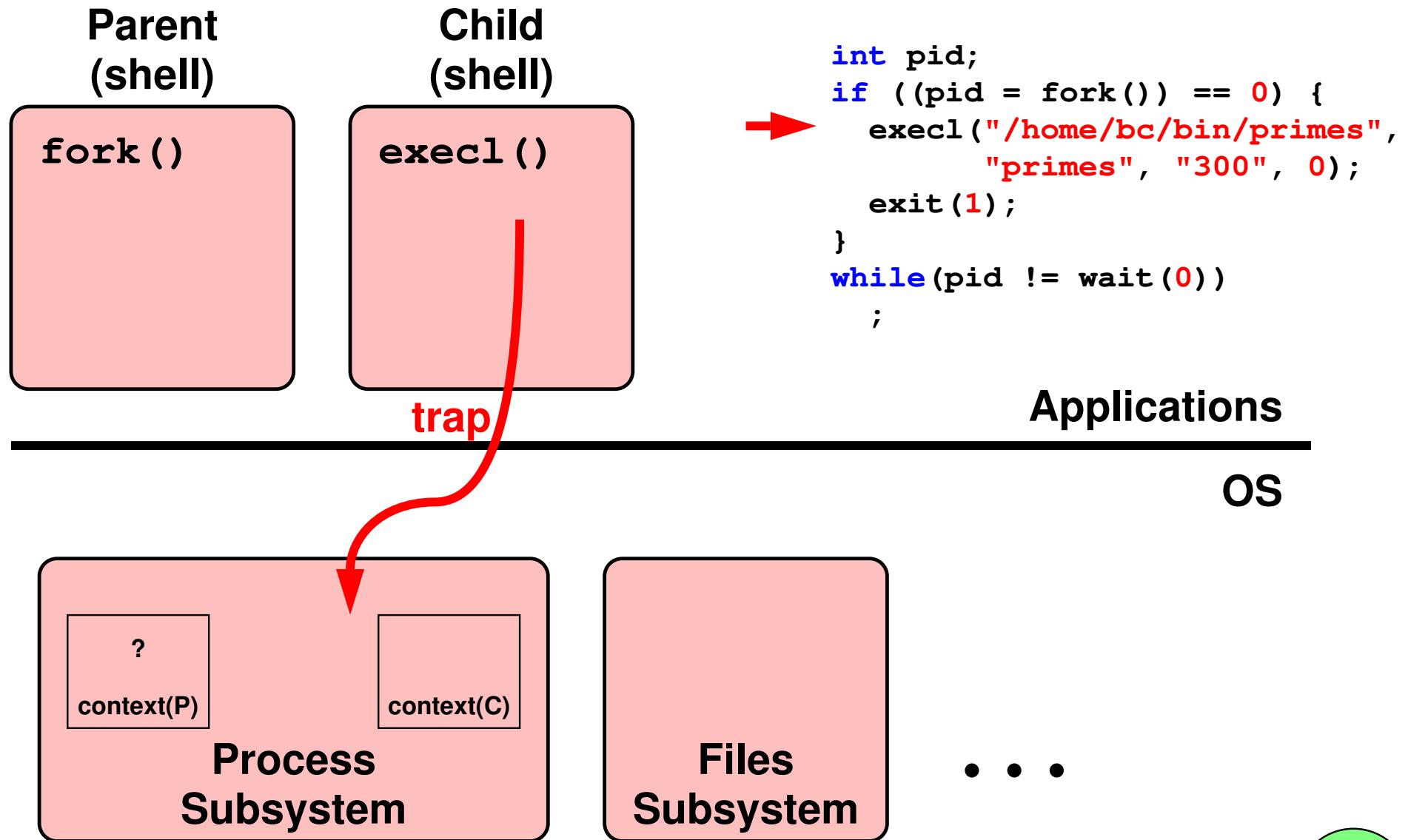
```

**Applications**

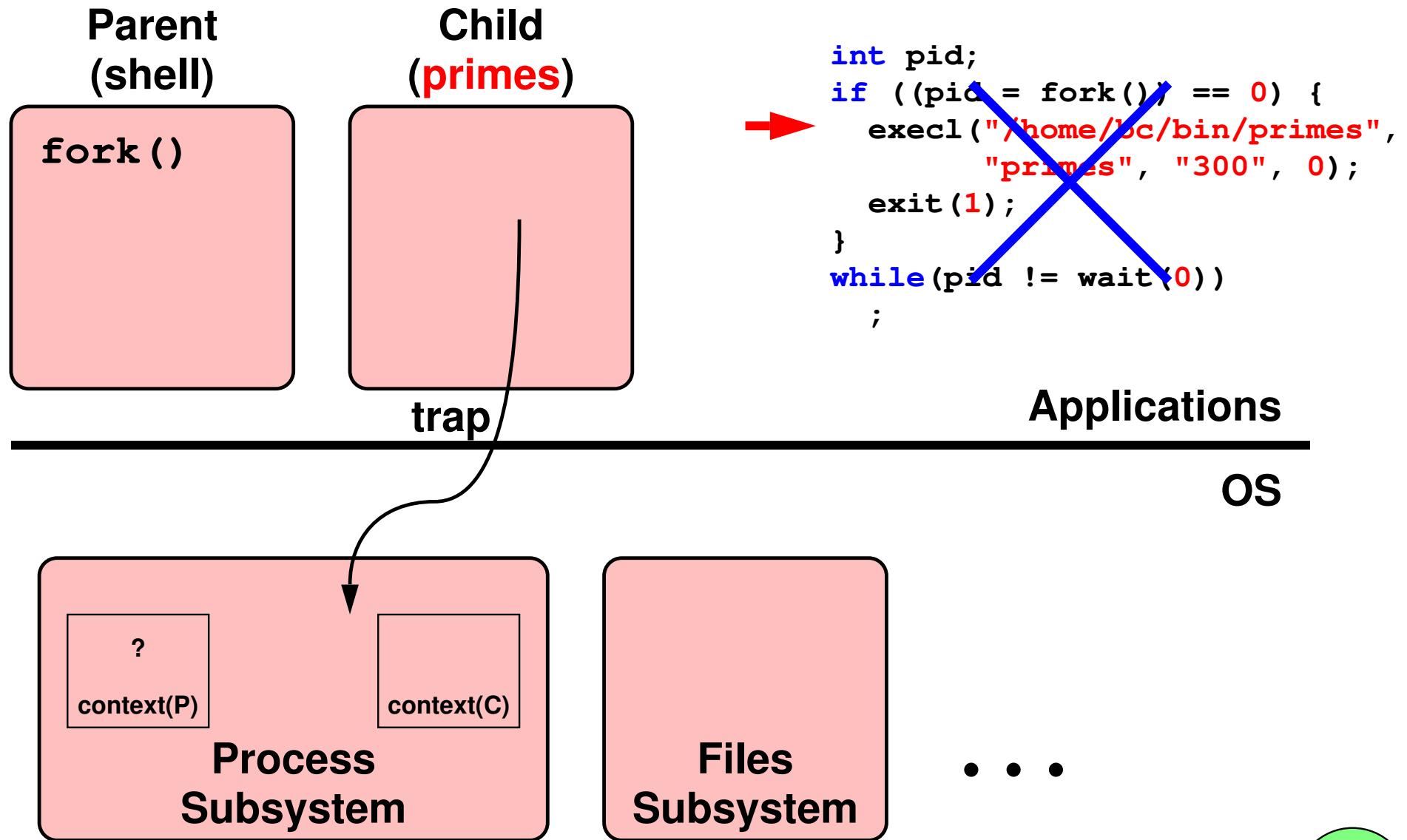
**OS**



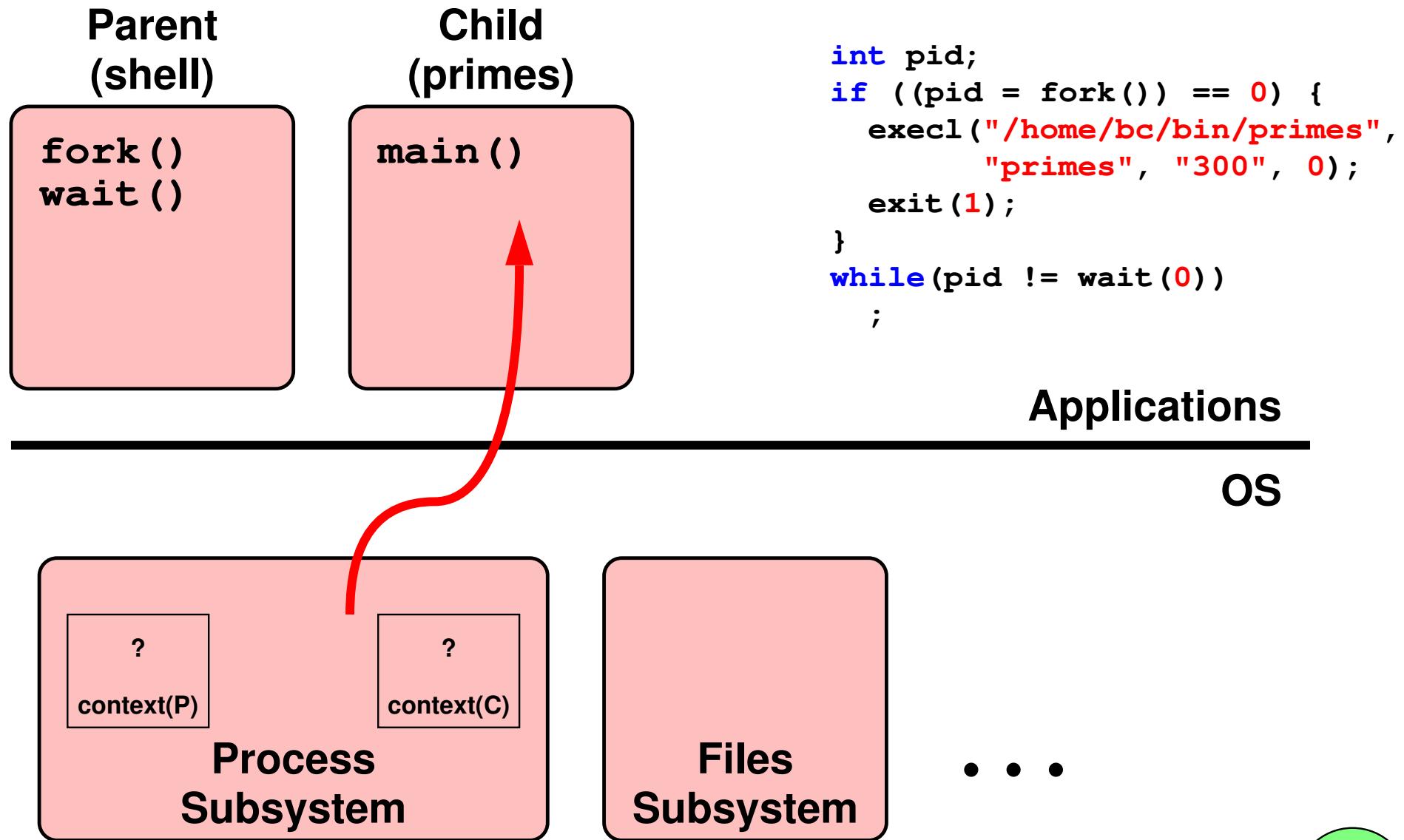
# Put It All Together



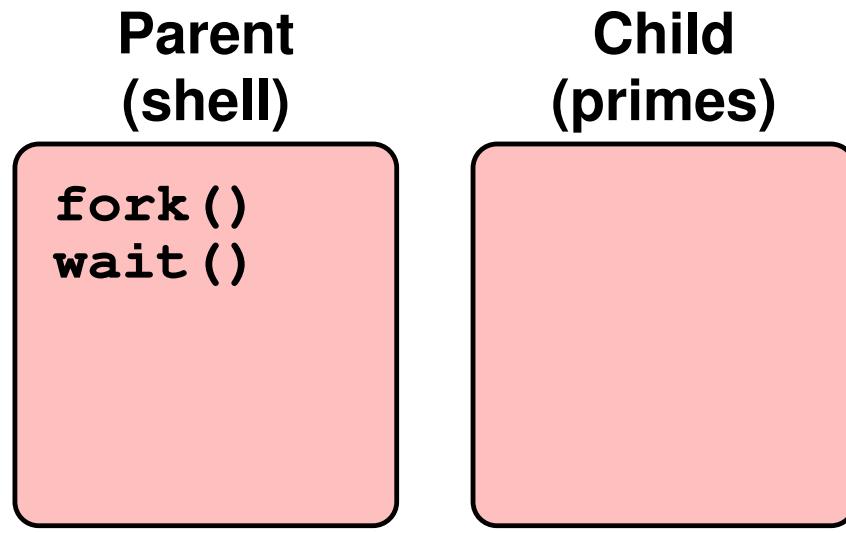
# Put It All Together



# Put It All Together



# Put It All Together



```

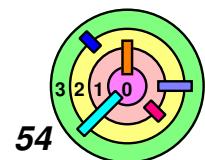
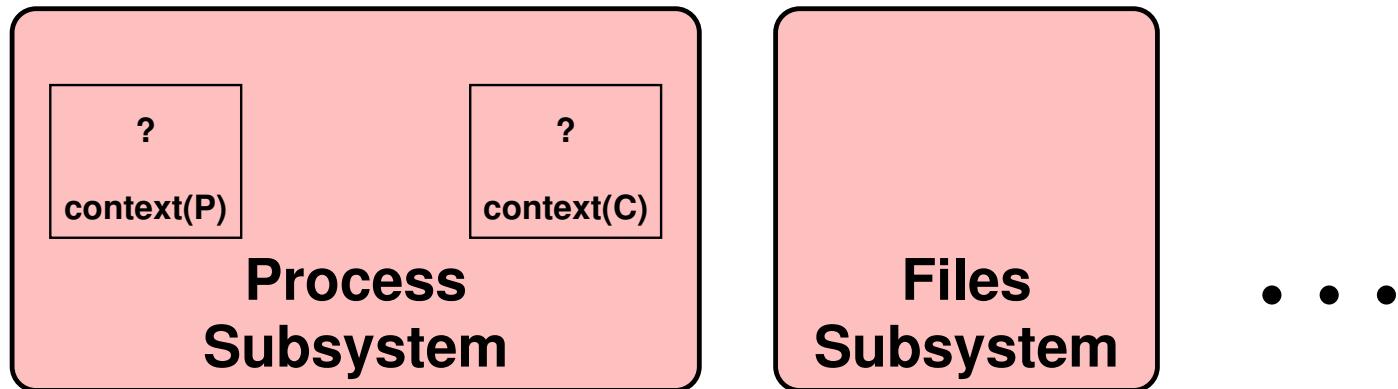
int pid;
if ((pid = fork()) == 0) {
    execl("/home/bc/bin/primes",
          "primes", "300", 0);
    exit(1);
}
while(pid != wait(0))
;
  
```

A red arrow points to the `while` loop in the child process code.

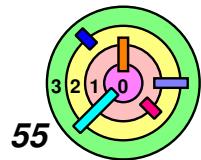
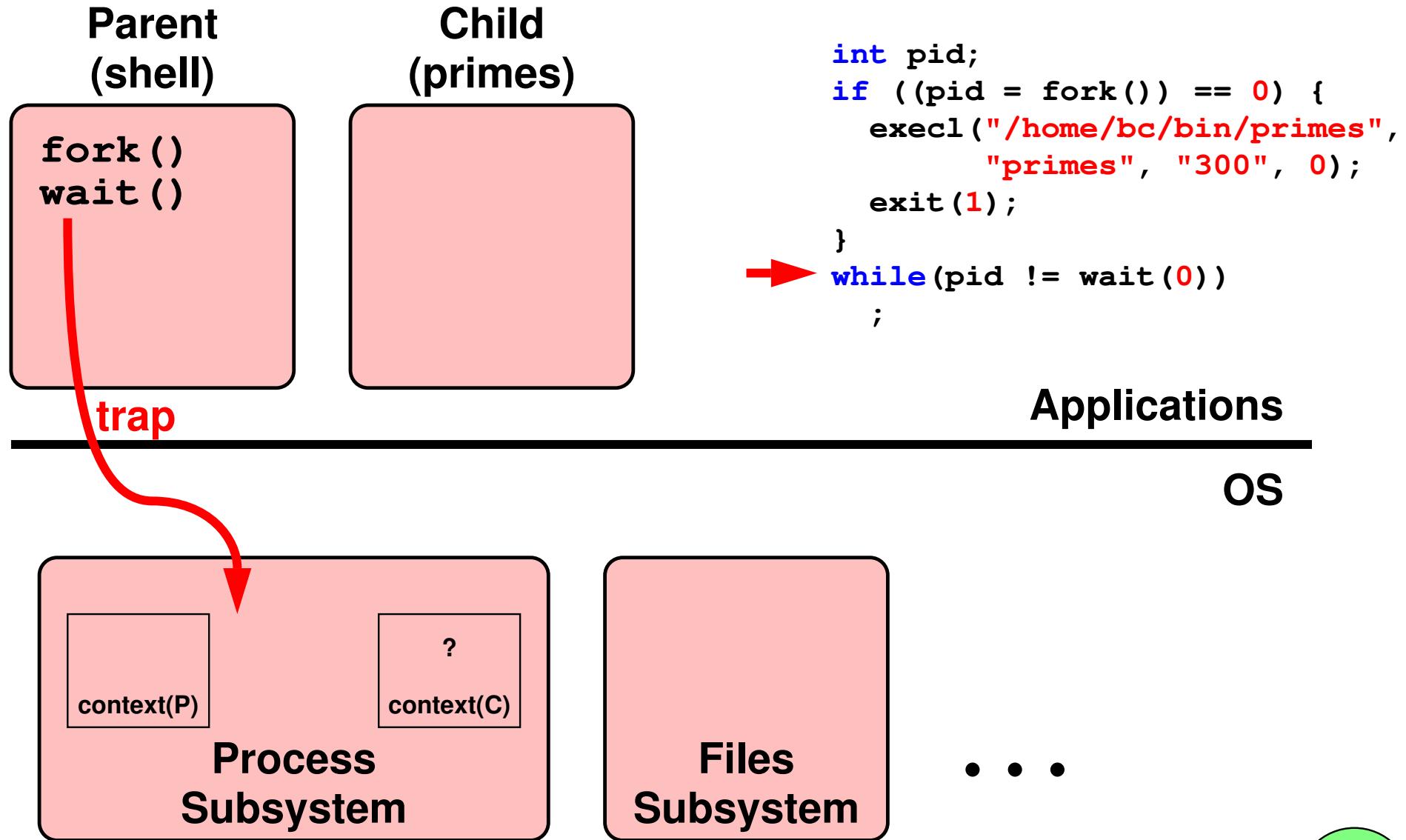
## Applications

---

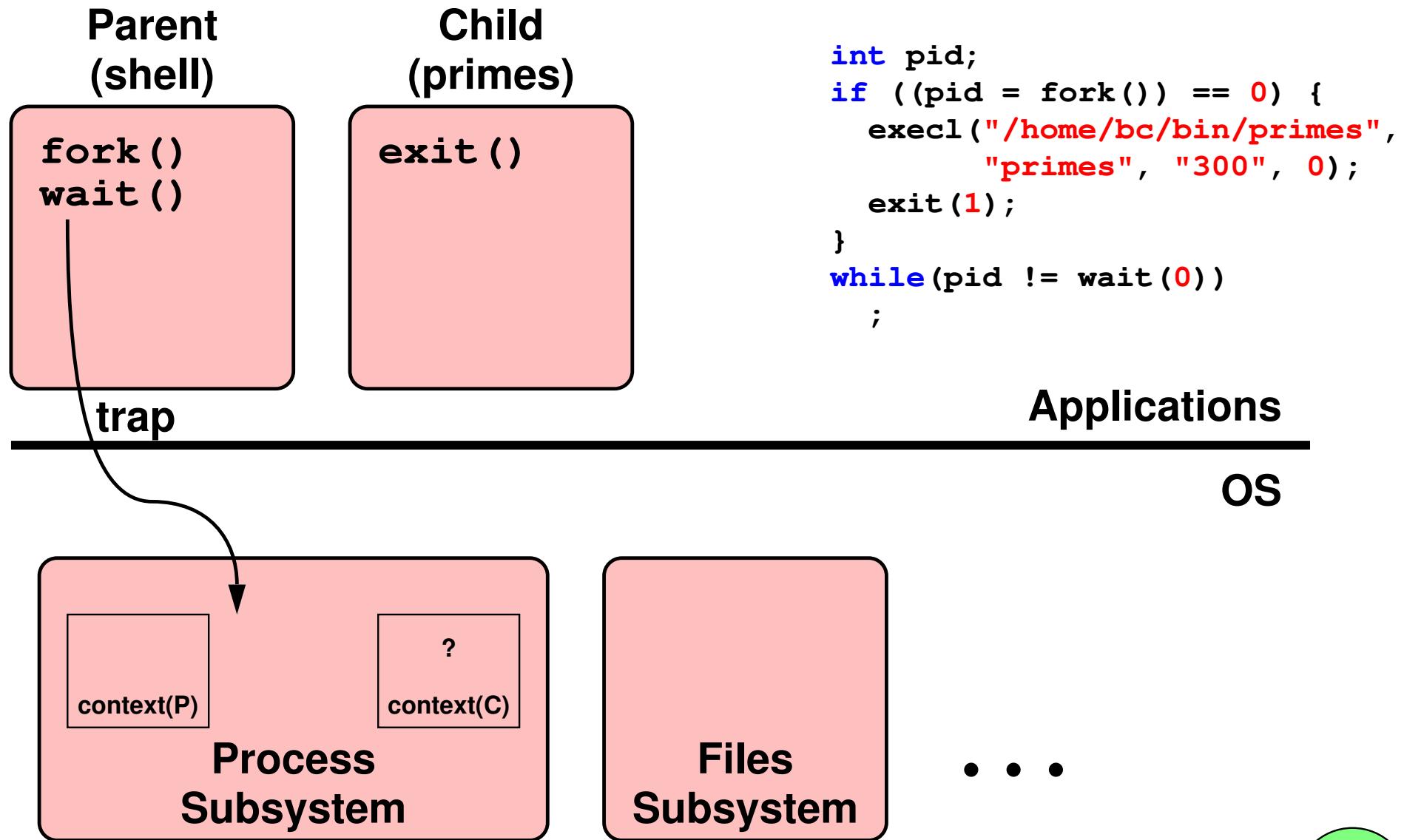
OS



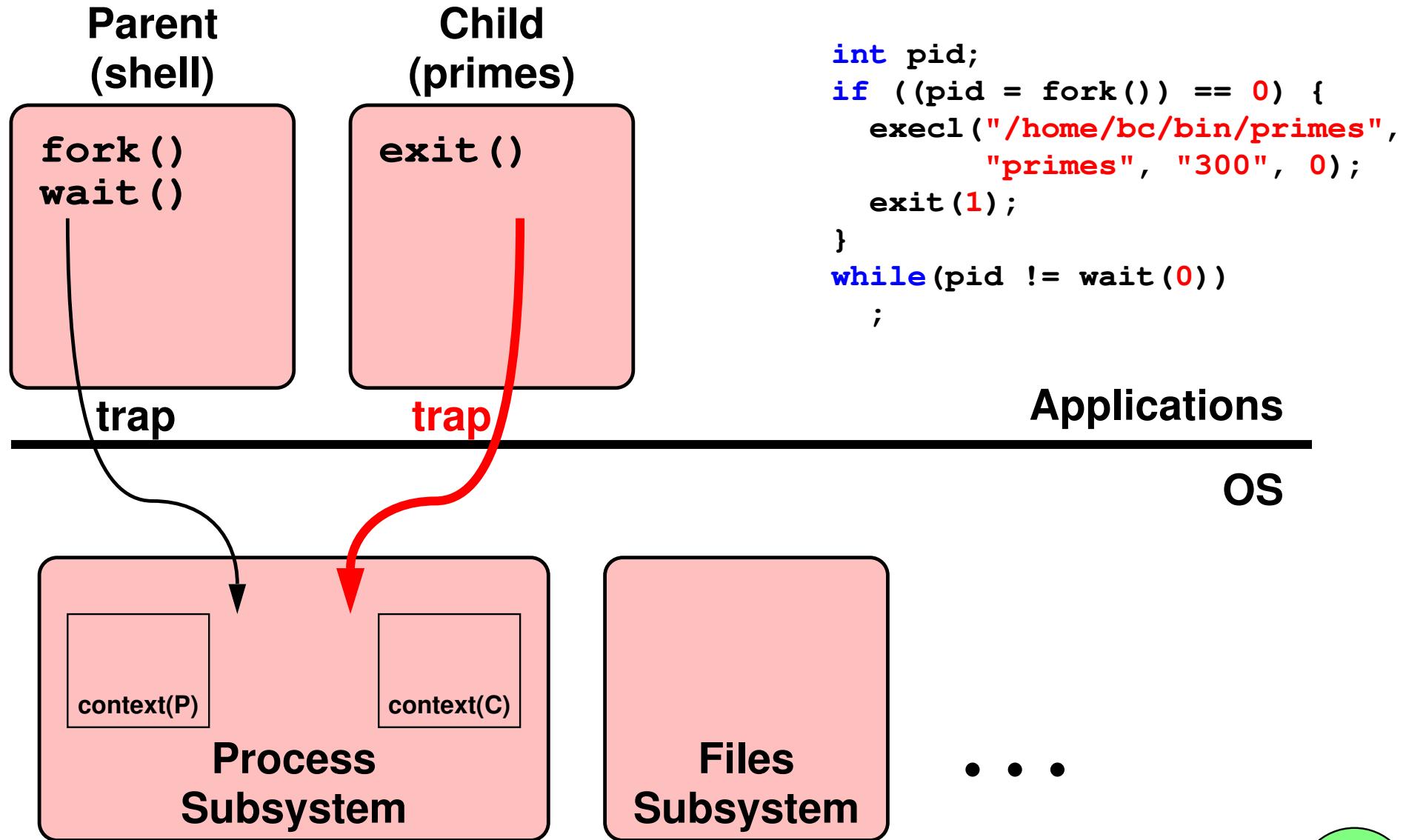
# Put It All Together



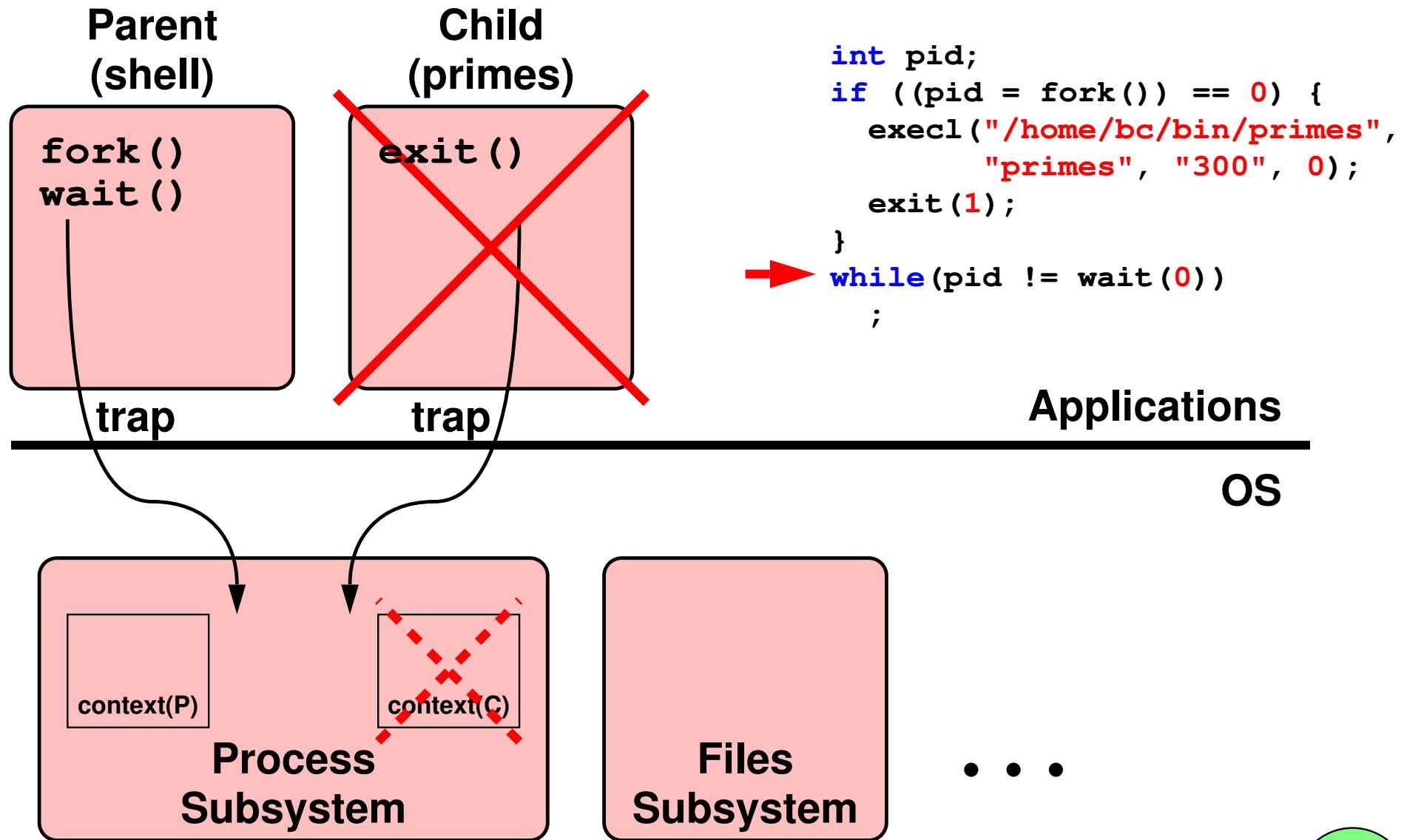
# Put It All Together



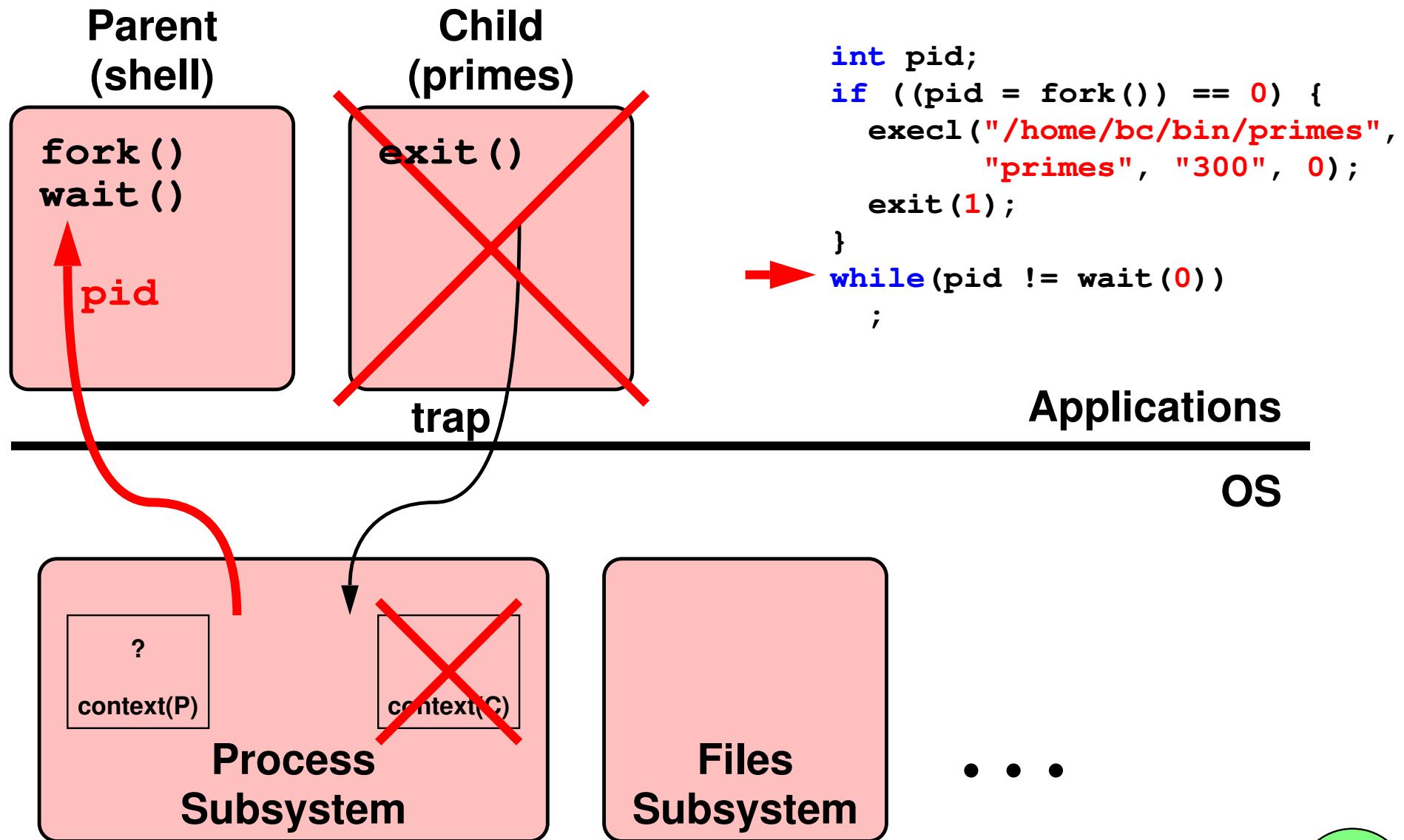
# Put It All Together



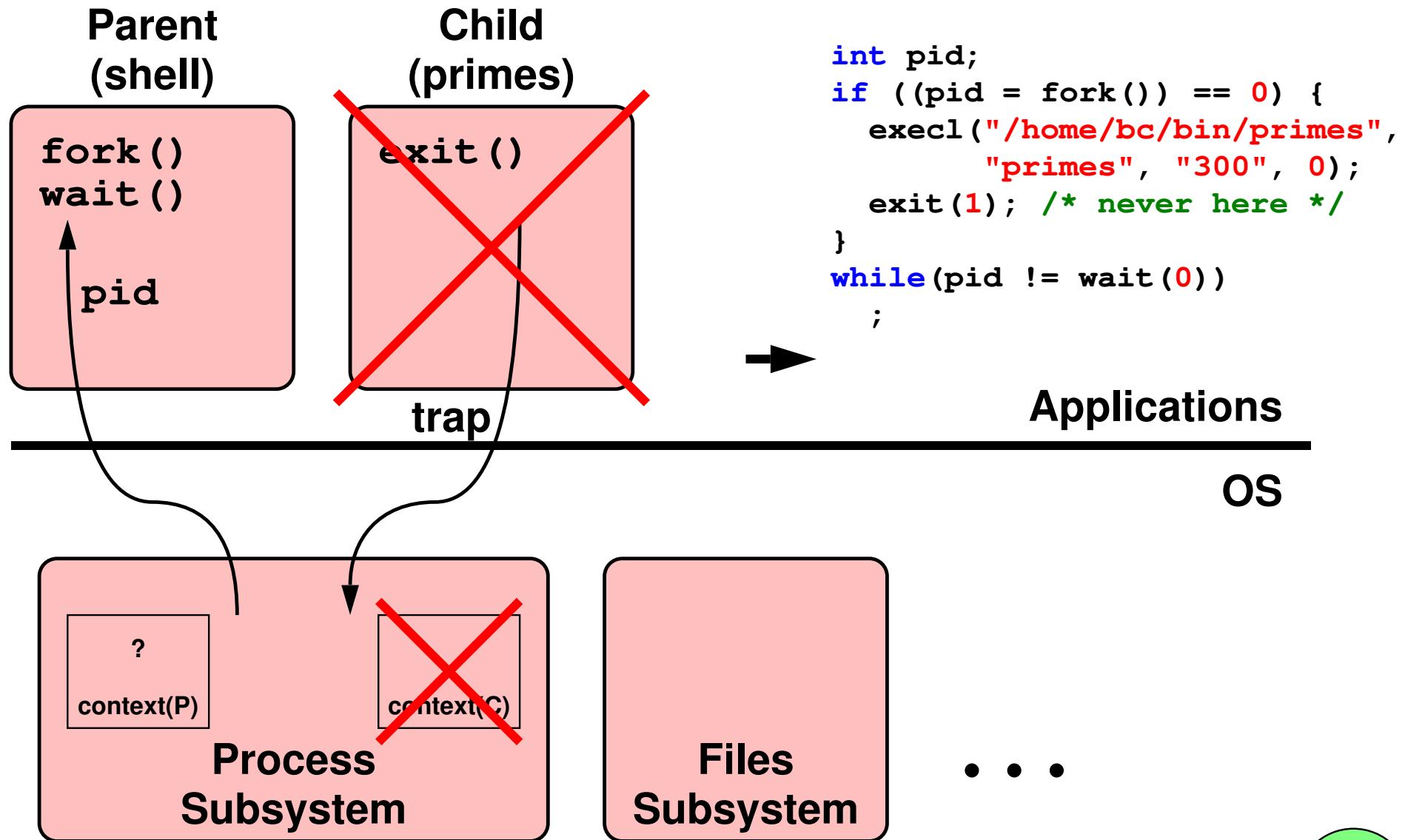
# Put It All Together



# Put It All Together



# Put It All Together



# More On System Calls

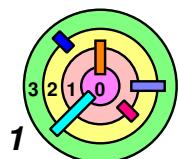
- ➔ **Sole interface** between user and kernel
  - this interface (definition of system calls) is what distinguishes one OS from another
    - in this class, we focus on Sixth-Edition Unix

- ➔ Implemented as library routines that execute "*trap machine instructions*" to enter kernel

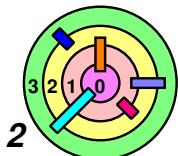
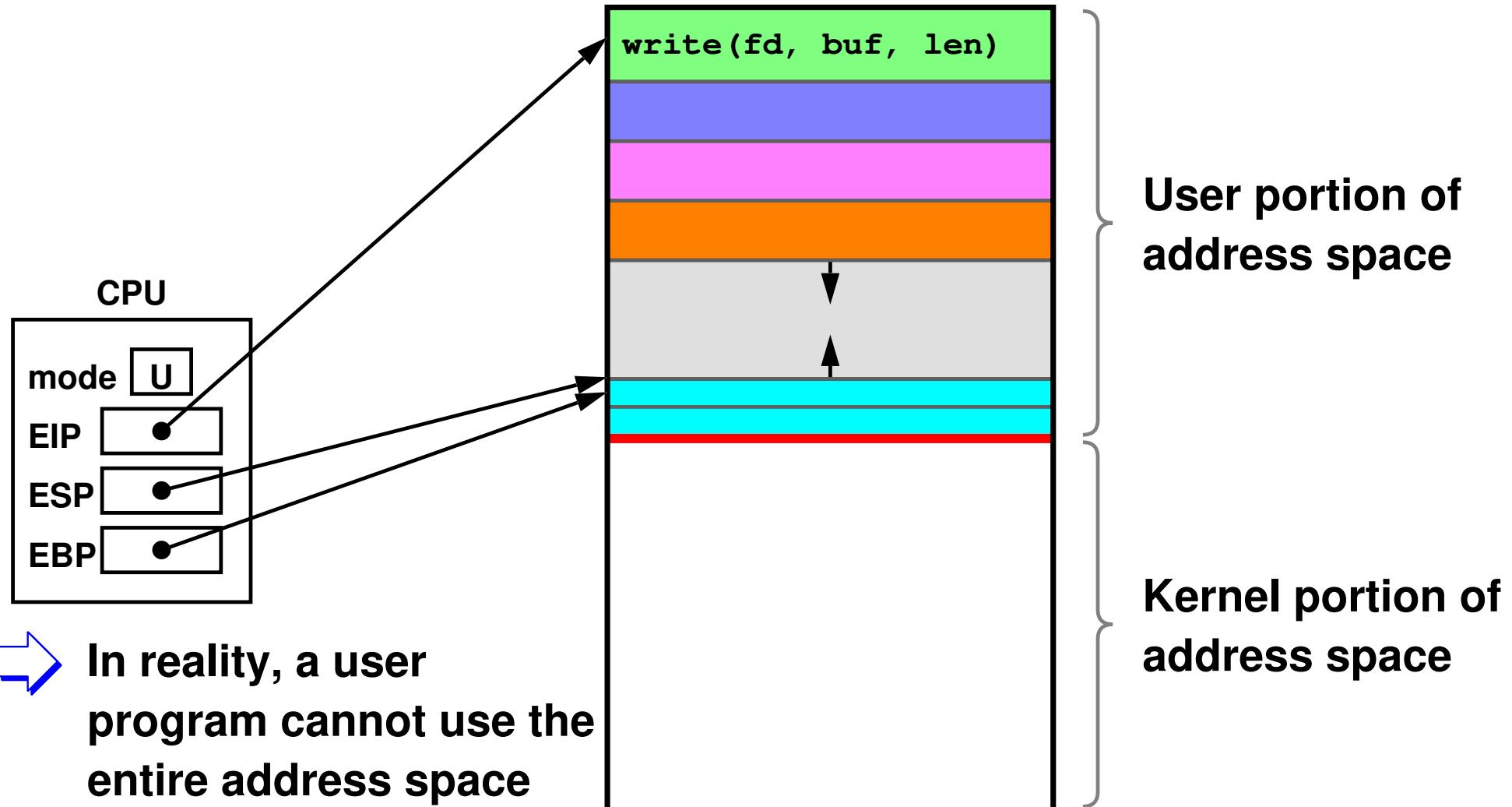
- ➔ Errors indicated by returning an invalid value
  - error code is in a global variable named **errno**

```
if (write(fd, buffer, bufsize) == -1) {  
    // error!  
    printf("error %d\n", errno);  
    // see perror  
}
```

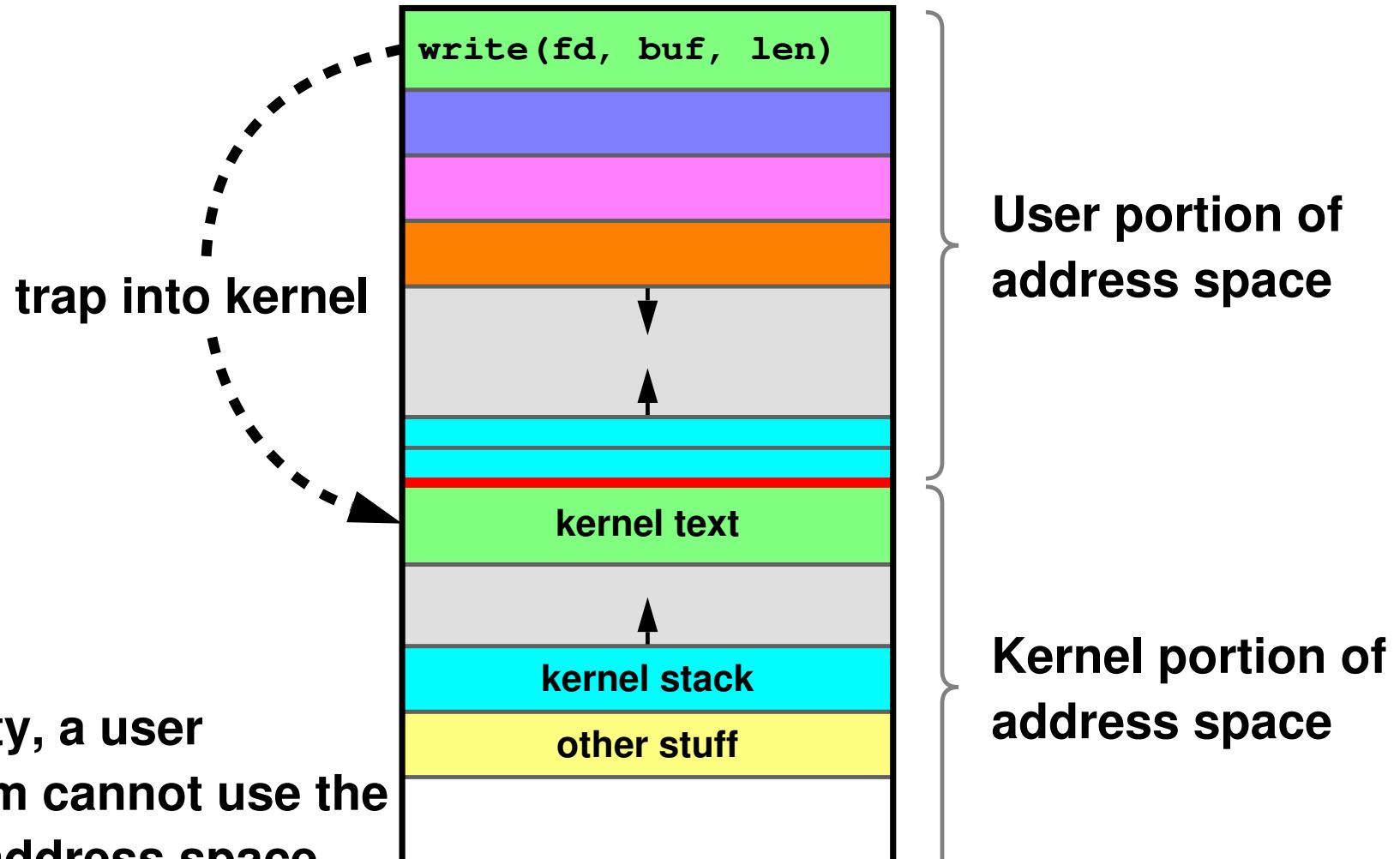
- on Ubuntu: "man 2 write" or "man -s 2 write"
- search man pages in all sections: "man -k ..."



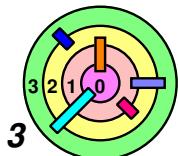
# System Calls



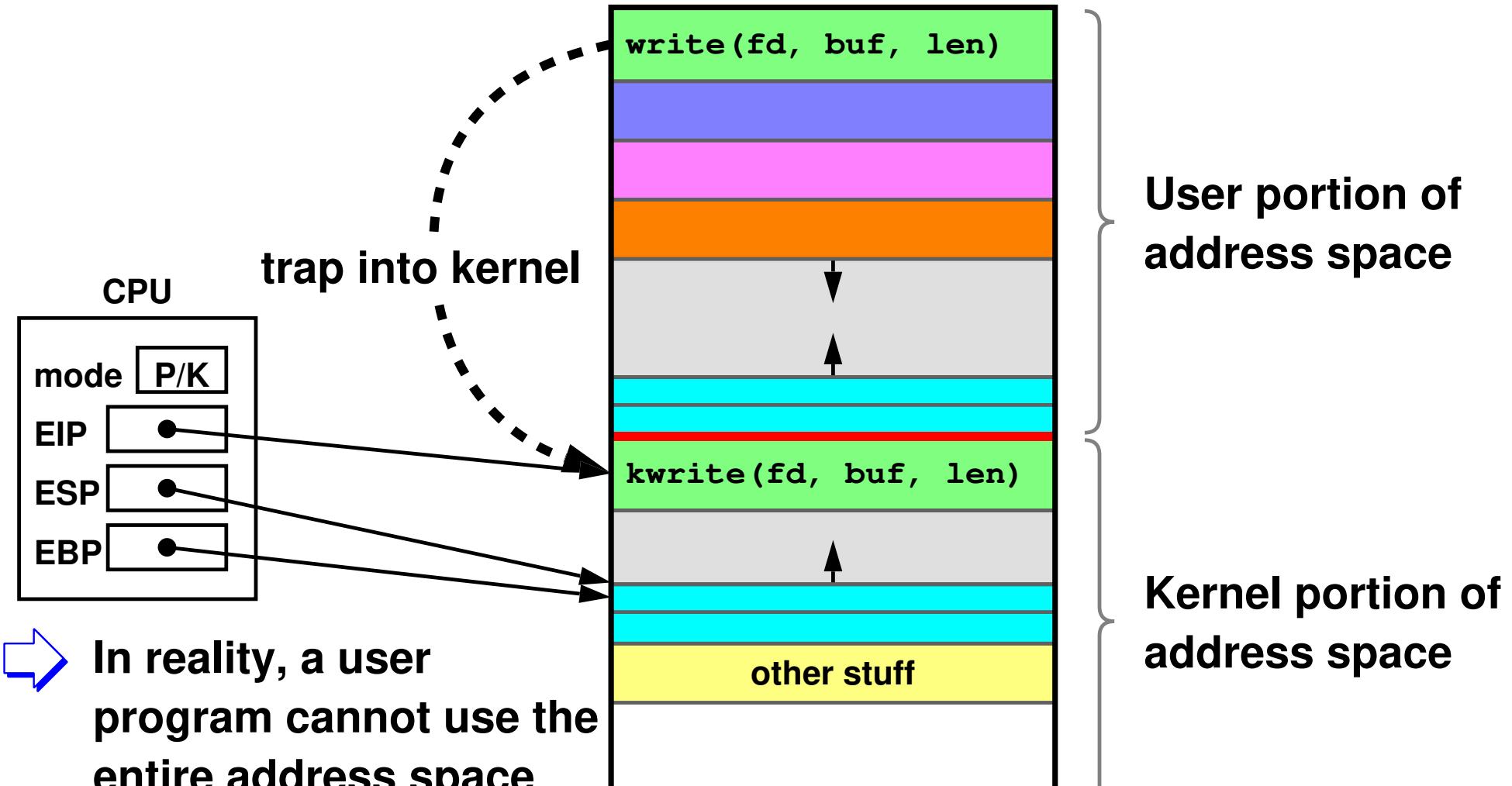
# System Calls



- In reality, a user program cannot use the entire address space
- Is this the same "*thread of execution*"?
  - is this the same process?

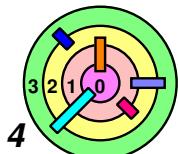


# System Calls

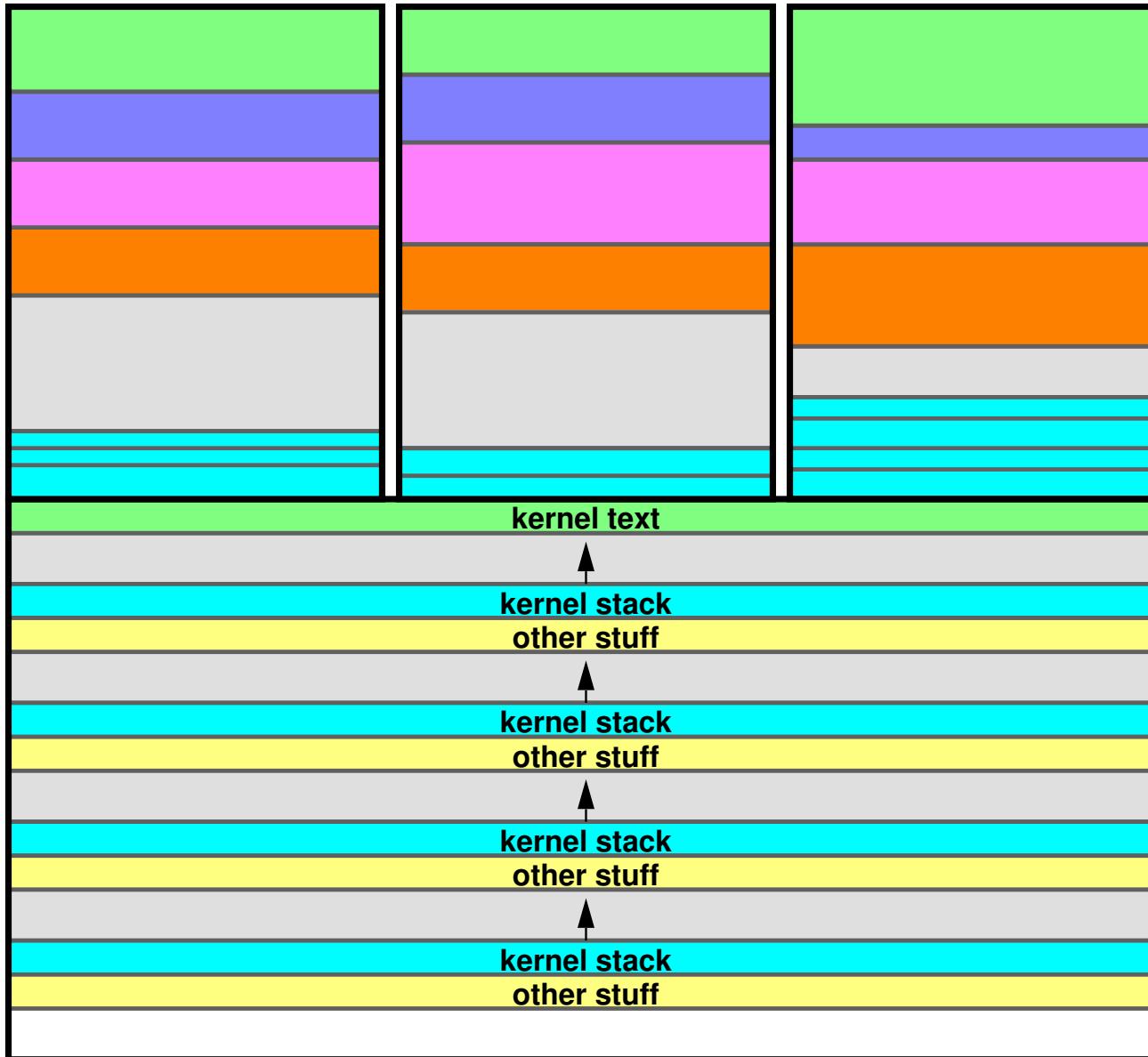


→ In reality, a user program cannot use the entire address space

→ Is this the same *"thread of execution"*?  
— is this the same process?



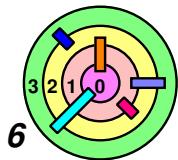
# Multiple Processes



- the same kernel spans across all user processes
  - although there are kernel-only processes as well (and they don't make system calls)
- process is just an abstraction
  - the kernel is very powerful

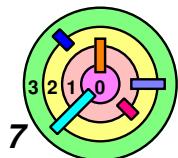
# 1.3 A Simple OS

- ▶ **OS Structure**
- ▶ **Processes, Address Spaces, & Threads**
- ▶ **Managing Processes**
- ▶ **Loading Program Into Processes**
- ▶ ***Files***



# Files

- Our "primes" program wasn't too interesting
  - it has no output!
  - cannot even verify that it's doing the right thing
  - other program cannot use its result
  - how does a process write to someplace *outside the process?*
- Files
  - abstraction of persistent data storage
  - means for fetching and storing data outside a process
    - including disks, another process, keyboard, display, etc.
    - need to *name* these different places
      - ◊ hierarchical naming structure
    - part of a process's *extended address space*
    - ◊ file "cursor position" is part of "execution context"
- The notion of a *file* is our Unix system's *sole abstraction* for this concept of "someplace outside the process"
  - modern Unix systems have additional abstractions



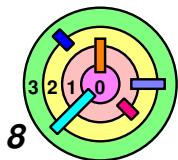
# Naming Files

## → Directory system

- shared by all processes running on a computer
  - although each process can have a different view
  - Unix provides a means to restrict a process to a subtree
    - ◆ by redefining what "root" means for the process
- name space is outside the processes
  - a user process provides the name of a file to the OS
  - the OS returns a **handle** to be used to access the file
    - ◆ after it has verified that the process is allowed **access** along the **entire path**, starting from root
  - user process uses the handle to read/write the file
    - ◆ avoid subsequent access checks

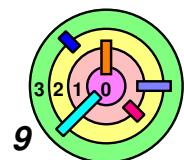
## → Using a **handle** (which can be an index into a kernel array) to **refer to an object managed by the kernel** is an important concept

- **handles** are essentially an **extension** to the process's **address space**
  - can even **survive execs!**



# The File Abstraction

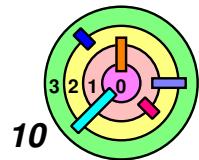
- A file is a simple array of bytes
- Files are made larger by writing beyond their current end
  - although you cannot *read* past the current end
- Files are named by paths in a naming tree
- File API
  - `open()`, `read()`, `write()`, `close()`
  - e.g., `cat`
- System calls on files are *synchronous* (unfortunately, Computer Science likes to use the same word to mean different things)
  - here, it means that the system call will not return until the operation is considered completed



# File Handles (File Descriptors)

```
int fd;
char buffer[1024];
int count;
if ((fd = open("/home/bc/file", O_RDWR) == -1) {
    // the file couldn't be opened
    perror("/home/bc/file");
    exit(1);
}
if ((count = read(fd, buffer, 1024)) == -1) {
    // the read failed
    perror("read");
    exit(1);
}
// buffer now contains count bytes read from the file
```

- what is O\_RDWR?
- what does perror() do?
- *cursor* position in an opened file depends on what functions/system calls you use
  - what about C++?



# Standard File Descriptors

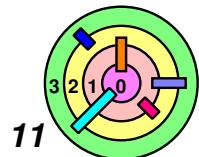


## Standard File Descriptors

- 0 is `stdin` (by default, "map/connect" to the keyboard)
- 1 is `stdout` (by default, "map/connect" to the display)
- 2 is `stderr` (by default, "map/connect" to the display)

```
main() {
    char buf[BUFSIZE];
    int n;
    const char *note = "Write failed\n";

    while ((n = read(0, buf, sizeof(buf))) > 0)
        if (write(1, buf, n) != n) {
            (void)write(2, note, strlen(note));
            exit(EXIT_FAILURE);
        }
    return(EXIT_SUCCESS);
}
```

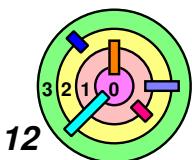


# Back to Primes

- Have our primes program write out the solution, i.e., the `primes[]` array

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    ...
    for (i=1; i<nprimes; i++) {
        ...
    }
    if (write(1, prime, nprimes*sizeof(int)) == -1) {
        perror("primes output");
        exit(1);
    }
    return(0);
}
```

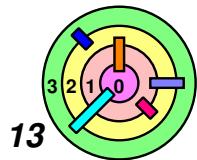
- the output is not readable by human



# Human-Readable Output

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    ...
    for (i=1; i<nprimes; i++) {
        ...
    }
    for (i=0; i<nprimes; i++) {
        fprintf(stdout, "%d\n", prime[i]);
    }
    return(0);
}
```

- **fprintf(stdout, ...)** is the same as **printf(...)**
  - **stdout** is a pre-defined **file pointer**
  - please see the **Programming FAQ** regarding the difference between a **file descriptor** and a **file pointer**



# Allocation of File Descriptors

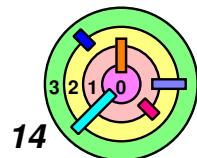
- For **each process**, the kernel maintains a **file descriptor table**, which is an array of pointers to "**file objects**"
  - a file object represents an **opened** file
  - a **file descriptor** is simply an **index** to this array

- Whenever a process requests a new file descriptor, the **lowest** numbered file descriptor not already associated with an open file is selected; thus

```
#include <fcntl.h>
#include <unistd.h>
...
close(0);
fd = open("file", O_RDONLY);
```

- the above will always associate "file" with file descriptor 0 (assuming that `open()` succeeds)

- You will need to implement the above rule in the kernel 2 assignment



# Running It

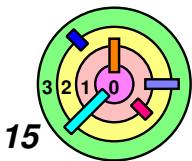
```

if (fork() == 0) {
    /* set up file descriptor 1 in the child process */
    close(1);
    if (open("/home/bc/Output", O_WRONLY) == -1) {
        perror("/home/bc/Output");
        exit(1);
    }
    execl("/home/bc/bin/primes", "primes", "300", 0);
    exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
;

```

- `close(1)` removes file descriptor 1 from ***extended address space***
- file descriptors are allocated ***lowest first*** on `open()`
- ***extended address space*** survives `execs`
- new code is same as running

% `primes 300 > /home/bc/Output`



# I/O Redirection

```
% primes 300 > /home/bc/Output
```

- The ">" parameter in a shell command that instructs the command shell to *redirect* the output to the given file
  - If ">" weren't there, the output would go to the display

- Can also redirect input

- ```
% cat < /home/bc/Output
```

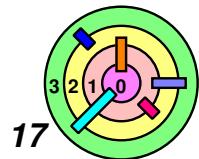
    - when the "cat" program reads from file descriptor 0, it would get the data bytes from the file "/home/bc/Output"

su21-den-Q11

# File Descriptor Table

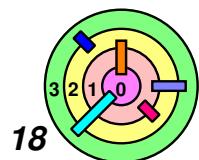


- A file descriptor refers not just to a file
  - it also refers to the *process's* current *context* for that file
    - includes how the file is to be accessed (how `open()` was invoked)
    - *cursor* position / file position
      - ◊ next location (zero-based array index) to read/write
      - ◊ initialized to 0 when a file is opened

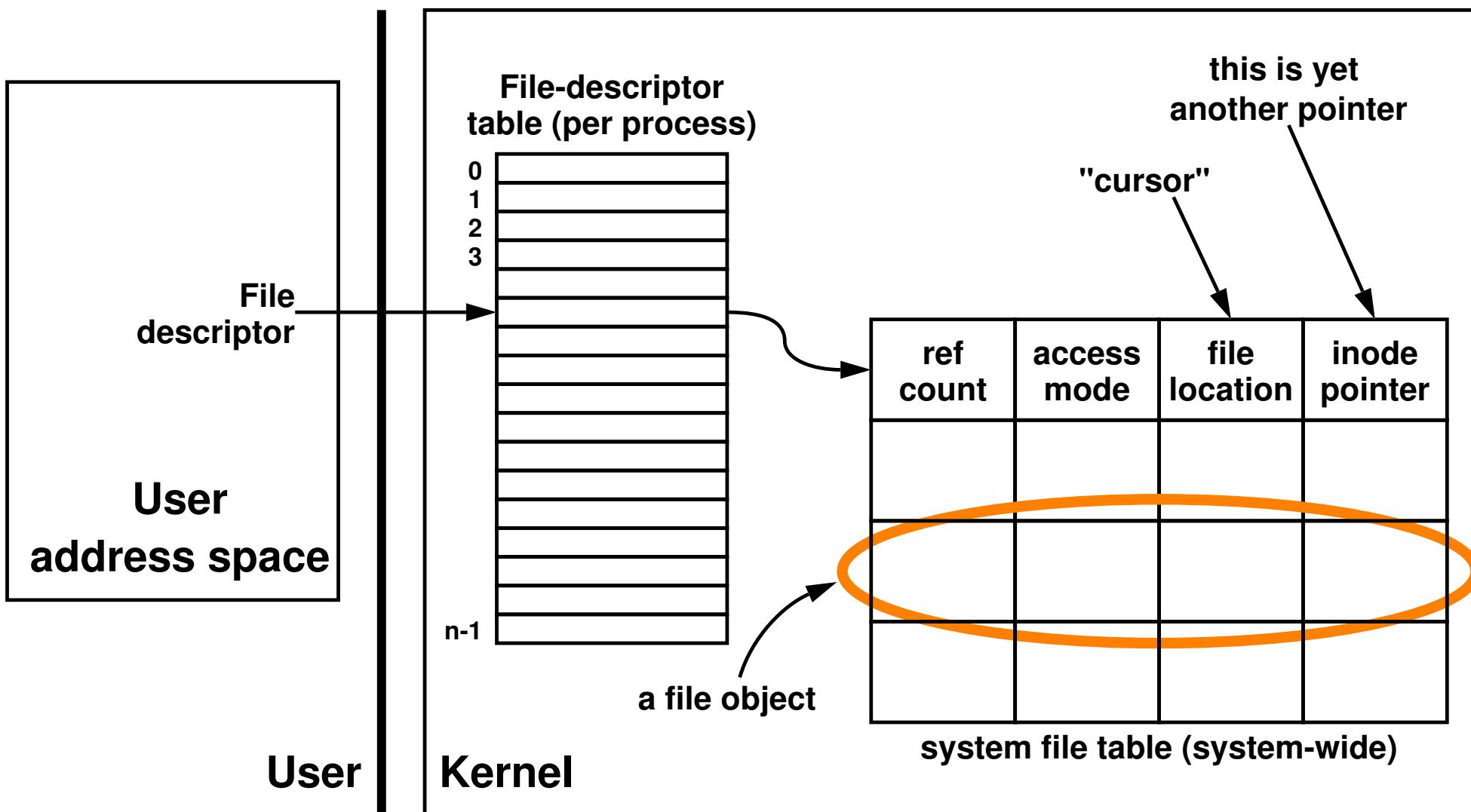


# File Object

- ➡ **Context** (or "execution context") information must be maintained by the OS and not directly by the user program
  - in this class, we will say that a **file object** is used to maintain the context information about an **opened file**
  - in addition to cursor position, a file object must also remember how a file was opened
- ➡ Let's say a user program opened a file with **O\_RDONLY**
  - later on it calls `write()` using the opened file descriptor
  - how does the OS knows that it doesn't have write access?
    - stores **O\_RDONLY** in context
  - if the user program can manipulate the context, it can change **O\_RDONLY** to **O\_RDWR**
  - therefore, user program must not have access to context!
    - all it can see is the handle
    - the file handle is an **index** into an array maintained for the process in kernel's address space



# File-Descriptor Table

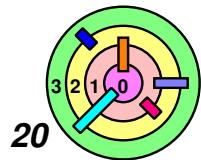


- = ***context*** is not stored directly into the file-descriptor table
  - one-level of ***indirection***

# Ch 2: Multithreaded Programming

Bill Cheng

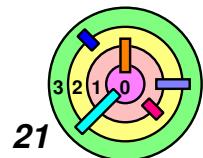
*<http://merlot.usc.edu/william/uscl/>*



20

# Overview

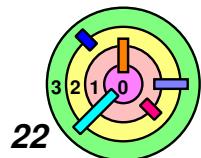
- ➡ **Why threads?**
- ➡ **How to program with threads?**
  - what is the API?
- ➡ **Synchronization**
  - mutual exclusion
  - semaphores
  - condition variables
- ➡ **Pitfall of thread programmings**



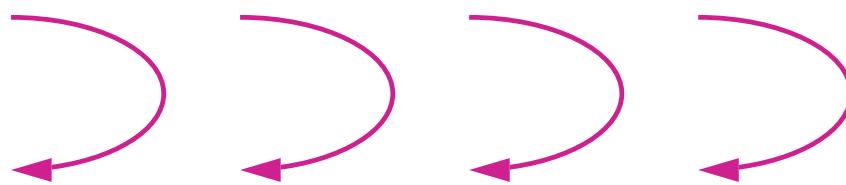
21

# Concurrency

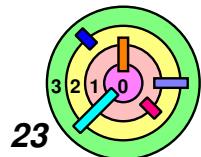
- ➔ Many things occur simultaneously in the OS
  - e.g., data coming from a disk, data coming from the network, data coming from the keyboard, mouse got clicked, jobs need to get executed
- ➔ If you have multiple processors, you may be able to handle things in parallel
  - that's real concurrency/parallelism
- ➔ If you only have one processor, you may want to make it look like things are running in parallel
  - do multiplexing to create the illusion
  - as it turns out, it's a *good idea* to do this even if you have only have one processor
- ➔ The down side is that if you want concurrency, you have to have *concurrency control* or bad things can happen



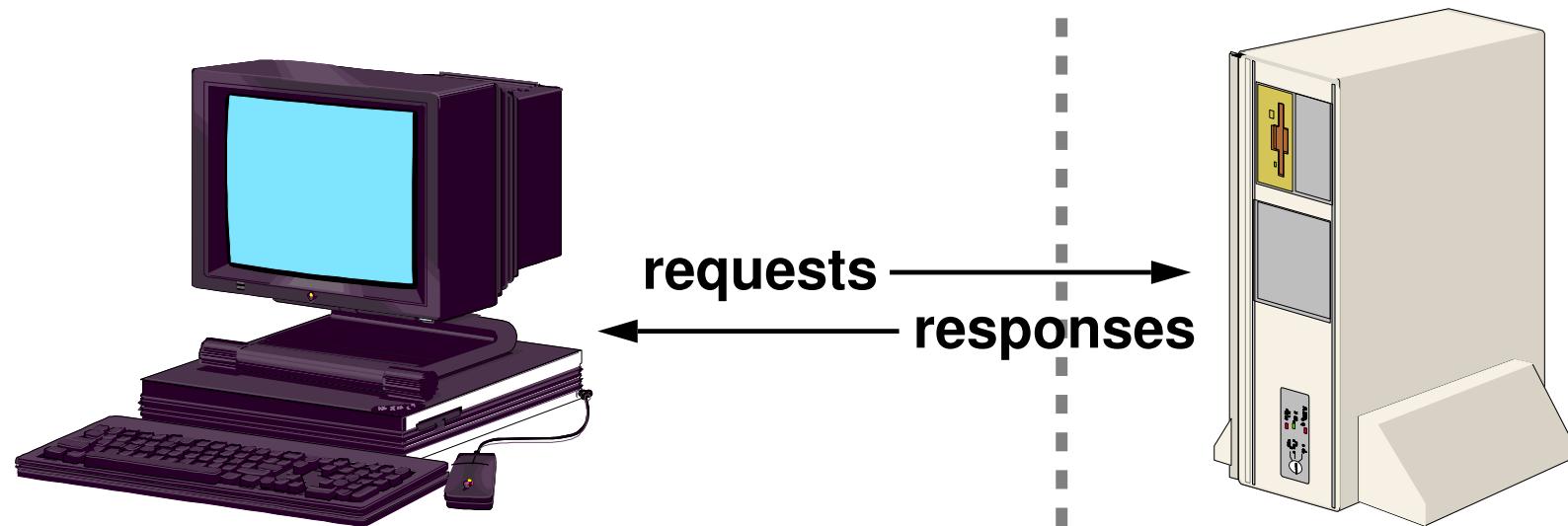
# Why Threads?



- ▶ Many things are easier to do with threads
  - *multithreading* is a *powerful paradigm*
  - makes your design *cleaner*, and therefore, less buggy
- ▶ Many things run faster with threads
  - if you are just waiting, don't waste CPU cycles, give the CPU to someone else, *without explicitly* giving up the CPU
- ▶ *Kernel threads* vs. *user threads*
  - basic concepts are the same
  - can easily do programming assignments for user-level threads
    - that's why we start here (to get you warmed up)!
    - for kernel programming assignments, you need to fill out missing parts of various kernel threads

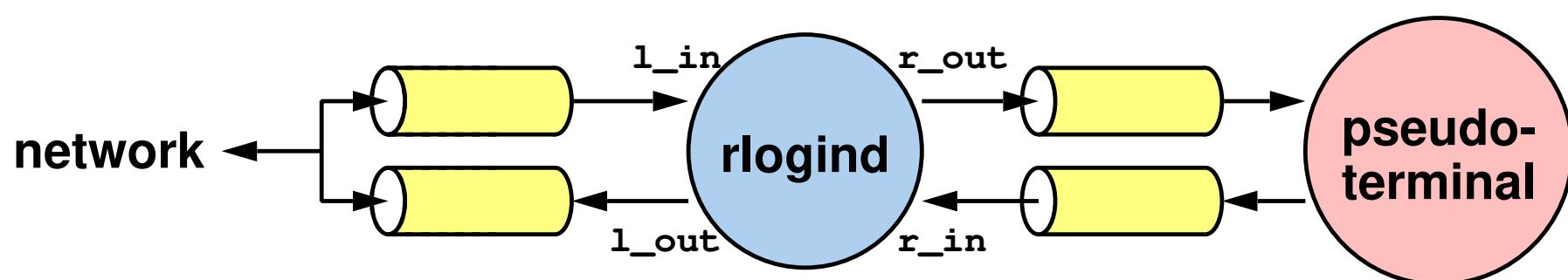


# A Simple Example: rlogind

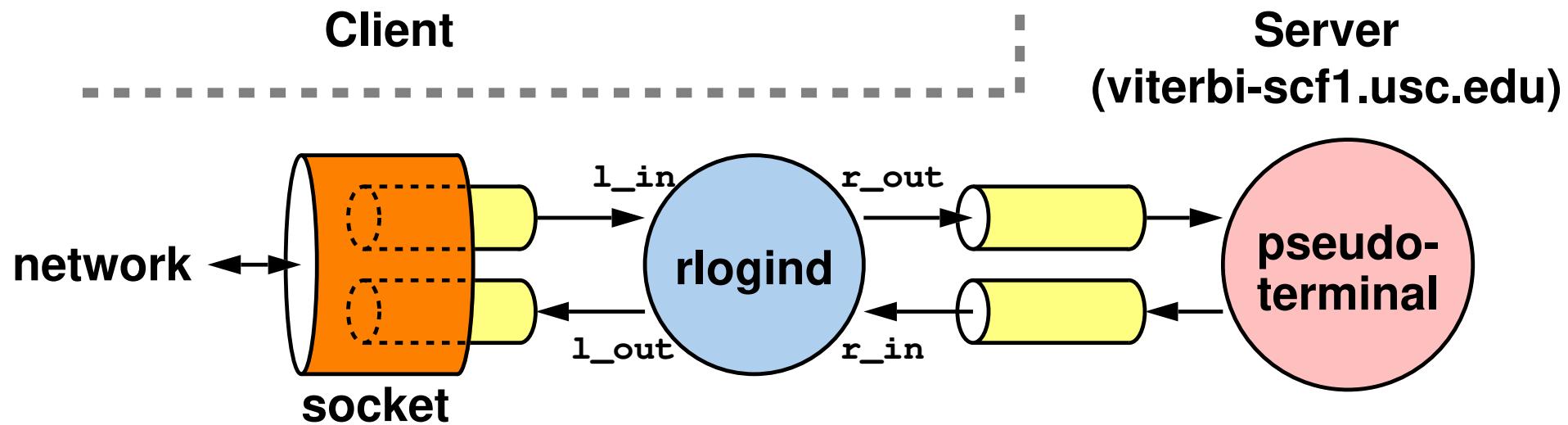
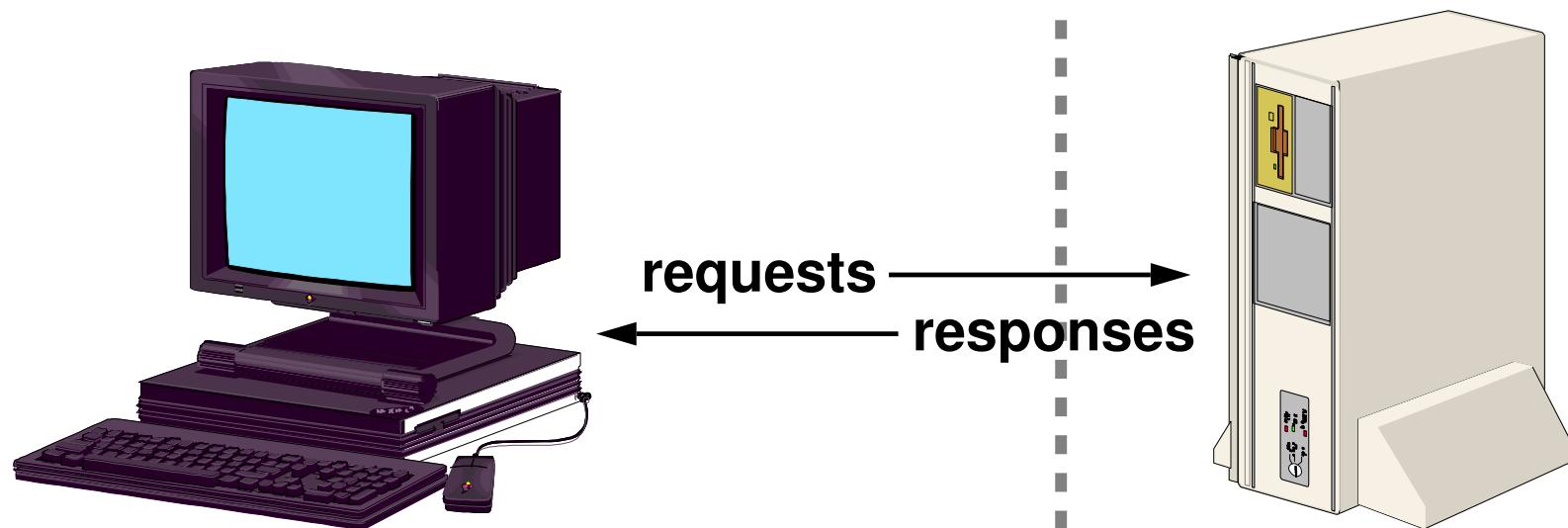


Client

Server  
(viterbi-scf1.usc.edu)



# A Simple Example: rlogind



- for a socket, `l_in` = `l_out`, i.e., you read and write using the same file descriptor



# Life Without Threads

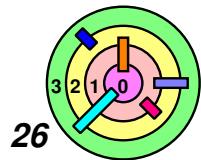
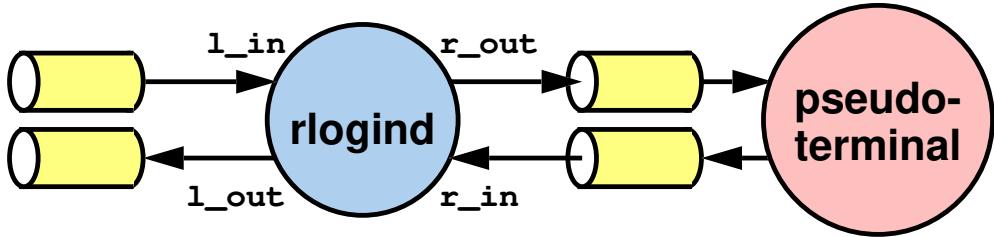
```

logind(int r_in, int r_out, int l_in, int l_out) {
    fd_set in = 0, out;
    int want_l_write = 0, want_r_write = 0;
    int want_l_read = 1, want_r_read = 1;
    int eof = 0, tsize, fsize, wret;
    char fbuf[BSIZE], tbuf[BSIZE];

    fcntl(r_in, F_SETFL, O_NONBLOCK);
    fcntl(r_out, F_SETFL, O_NONBLOCK);
    fcntl(l_in, F_SETFL, O_NONBLOCK);
    fcntl(l_out, F_SETFL, O_NONBLOCK);

    while(!eof) {
        FD_ZERO(&in);
        FD_ZERO(&out);
        if (want_l_read) FD_SET(l_in, &in);
        if (want_r_read) FD_SET(r_in, &in);
        if (want_l_write) FD_SET(l_out, &out);
        if (want_r_write) FD_SET(r_out, &out);
        select(MAXFD, &in, &out, 0, 0);
        if (FD_ISSET(l_in, &in)) {
            if ((tsize = read(l_in, tbuf, BSIZE)) > 0) {
                want_l_read = 0;
                want_r_write = 1;
            } else { eof = 1; }
        }
    }
}

```

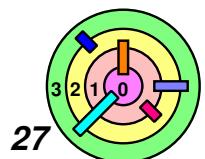
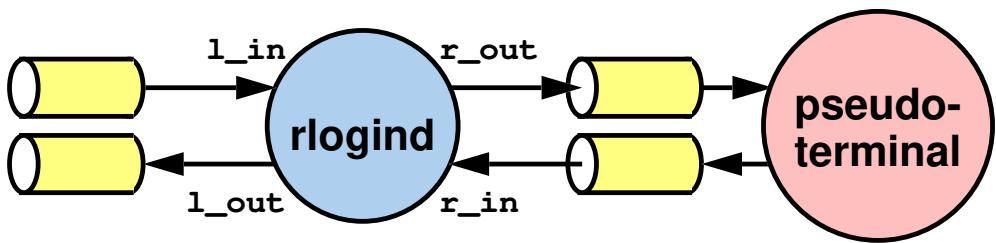


# Life Without Threads

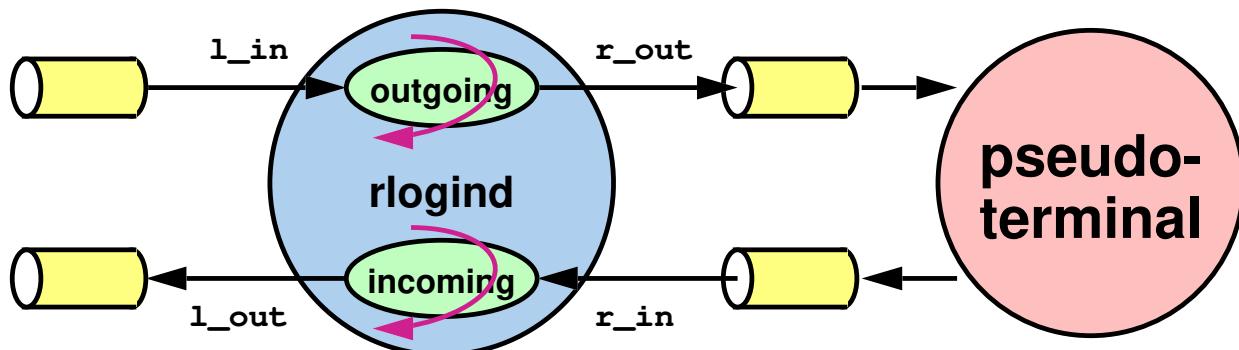
```

if (FD_ISSET(r_in, &in)) {
    if ((fsize = read(r_in, fbuf, BSIZE)) > 0) {
        want_r_read = 0;
        want_l_write = 1;
    } else { eof = 1; }
}
if (FD_ISSET(l_out, &out)) {
    if ((wret = write(l_out, fbuf, fszie)) == fszie) {
        want_r_read = 1;
        want_l_write = 0;
    } else if (wret >= 0) {
        tszie -= wret;
    } else { eof = 1; }
}
if (FD_ISSET(r_out, &out)) {
    if ((wret = write(r_out, tbuf, tszie)) == tszie) {
        want_l_read = 1;
        want_r_write = 0;
    } else if (wret >= 0) {
        tszie -= wret;
    } else { eof = 1; }
}
}
}

```



# Life With Threads



```

incoming(int r_in, int l_out) {
    int eof = 0;
    char buf[BSIZE];
    int size;

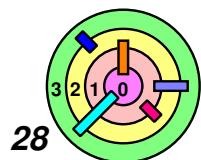
    while (!eof) {
        size = read(r_in, buf, BSIZE);
        if (size <= 0)
            eof = 1;
        if (write(l_out, buf, size) <= 0)
            eof = 1;
    }
}

outgoing(int l_in, int r_out) {
    int eof = 0;
    char buf[BSIZE];
    int size;

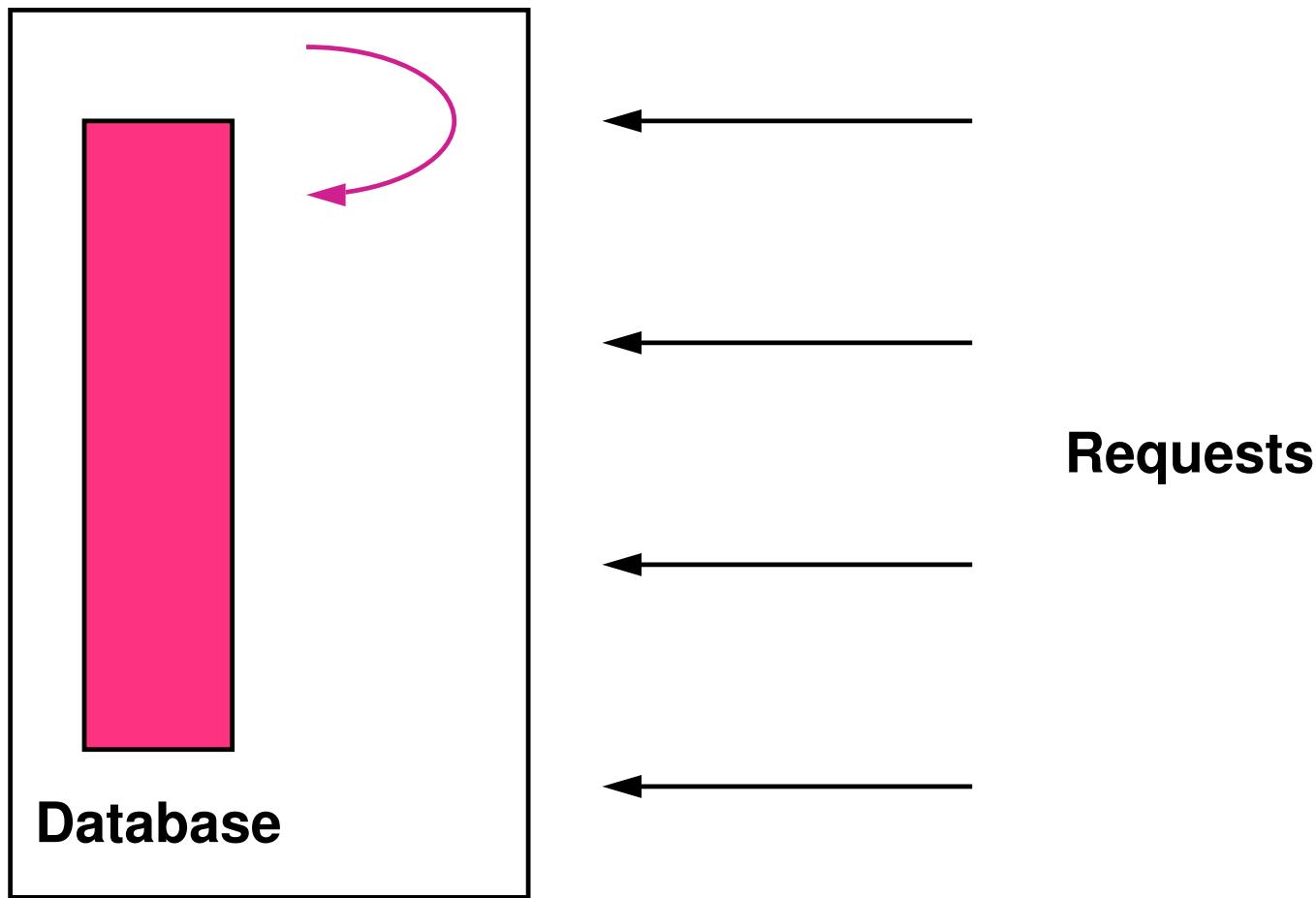
    while (!eof) {
        size = read(l_in, buf, BSIZE);
        if (size <= 0)
            eof = 1;
        if (write(r_out, buf, size) <= 0)
            eof = 1;
    }
}

```

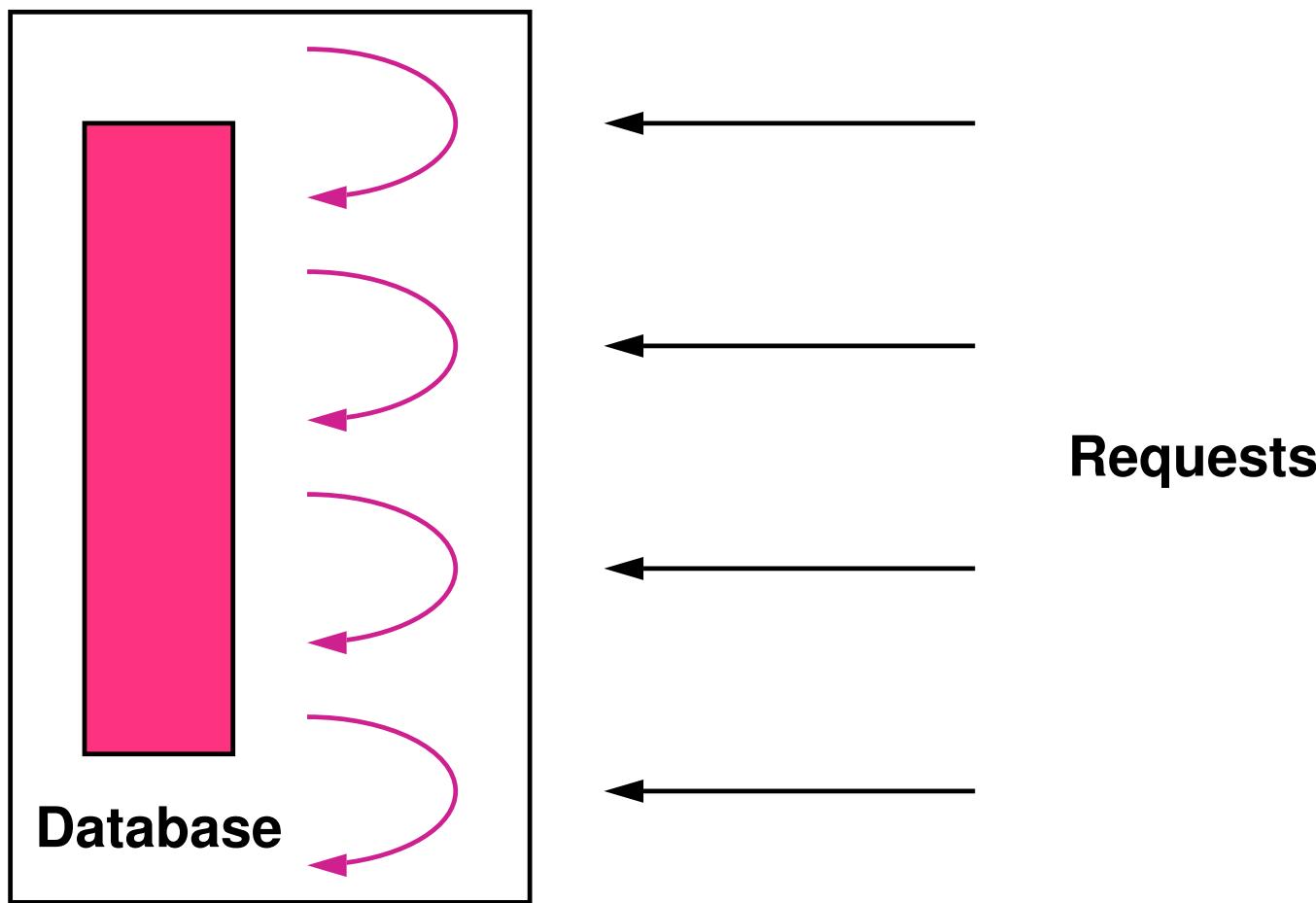
- don't have to call `select()`



# Single-Threaded Database Server



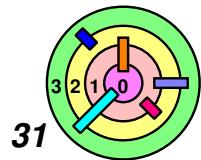
# Multithreaded Database Server



- will be very difficult to implement this without using threads if you want to handle a large number of requests simultaneously

# 2.2 Programming With Threads

- ▶ *Threads Creation & Termination*
- ▶ Threads & C++ su21-a-Q10
- ▶ Synchronization
- ▶ Thread Safety
- ▶ Deviations



# Creating a POSIX Thread



`man pthread_create`

su21-den-Q1

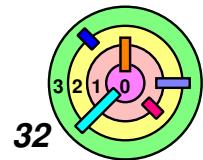
## SYNOPSIS

```
#include <pthread.h>

int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg);
```

Compile and link with `-pthread`.

- the `start_routine` is also known as the "*first procedure*" or "*thread function*" of the child thread
  - it's like `main()` for the child thread
- the "thread ID" of the newly created thread will be *returned* in the first argument of `pthread_create()`
  - may *not* be a *Thread Control Block*



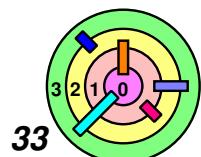
# Creating a POSIX Thread

```
start_servers( ) {
    pthread_t thread;
    int i;
    for (i = 0; i < 100; i++)
        pthread_create(&thread,           // thread ID
                      0,                 // default attributes
                      server,             // first procedure
                      argument);         // argument of first
                               // procedure
}
```

```
void *server(void *arg) {
    // perform service
    return(0);
}
```

← child thread ***starts*** executing here  
 ← arg = argument (from caller)  
 ← child thread ***ends*** when ***return***  
 from its ***start routine / first procedure***

- `pthread_create()` returns 0 if successful
- POSIX 1003.1c standard
  - `pthread` is a ***user-space*** library package
- ***threads*** in a process ***shares the address space***



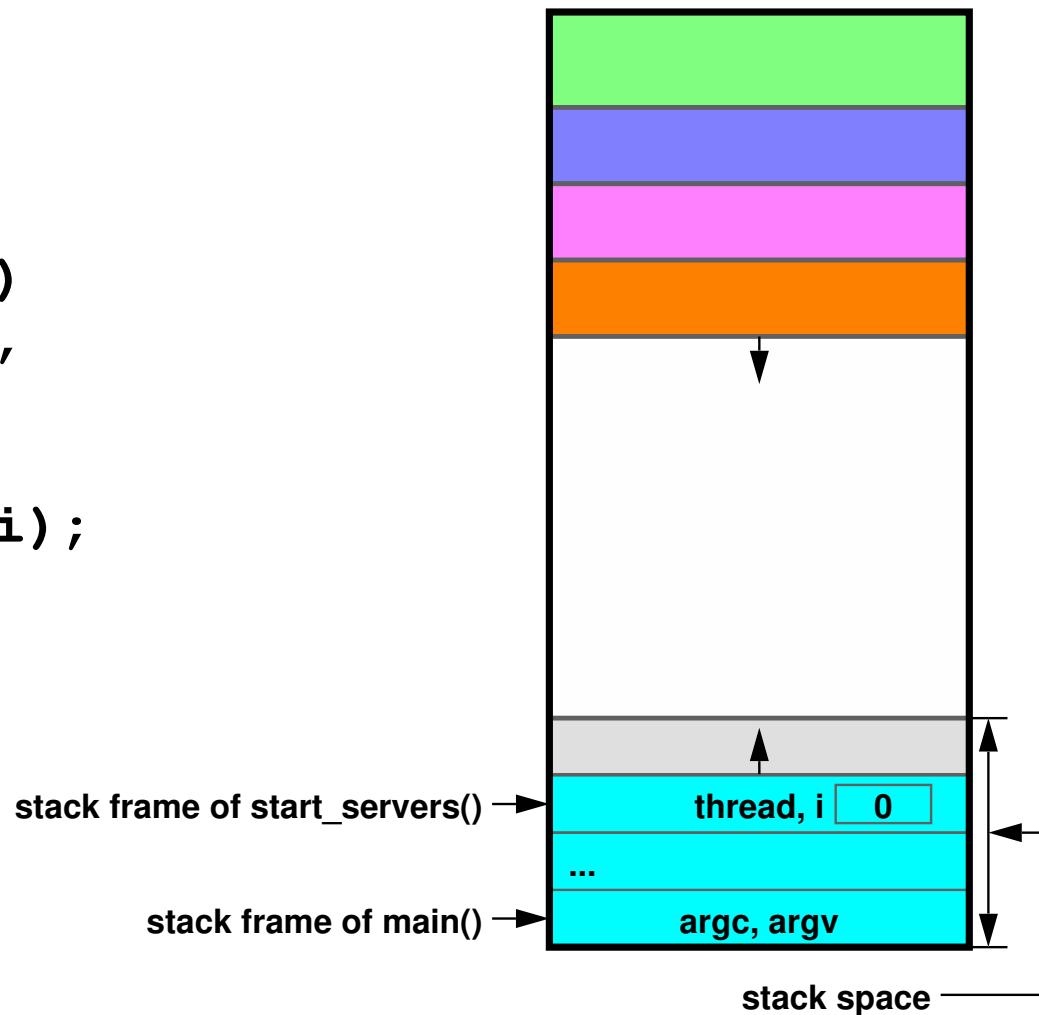
# Creating a POSIX Thread

```

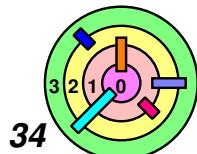
start_servers( ) {
    pthread_t thread;
    int i;
    for (i = 0; i < 100; i++)
        → pthread_create(&thread,
                         0,
                         server,
                         (void*)i);
}

void *server(void *arg) {
    int k=(int)arg;
    // perform service
    return(0);
}

```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - ◊ one arg in each of these stack frames



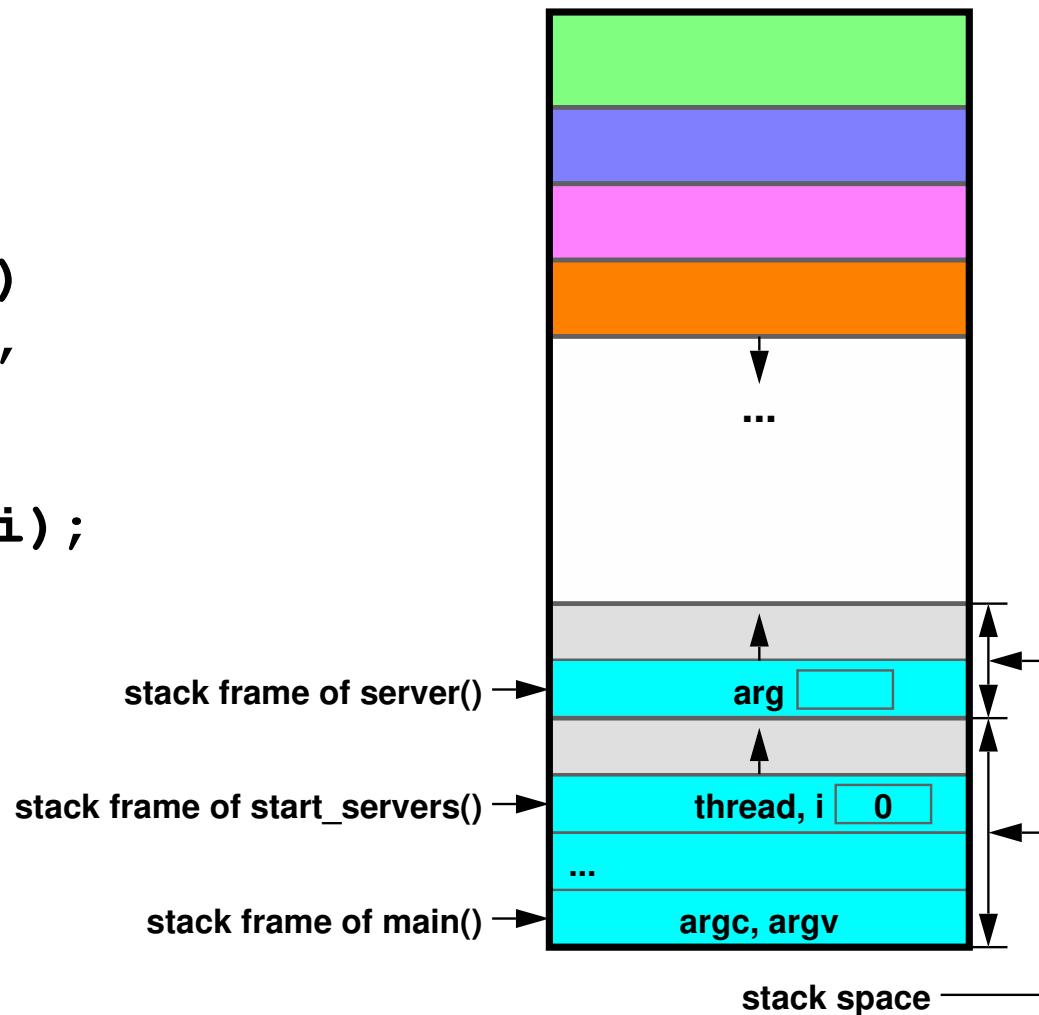
# Creating a POSIX Thread

```

start_servers( ) {
    pthread_t thread;
    int i;
    for (i = 0; i < 100; i++)
        → pthread_create(&thread,
                         0,
                         server,
                         (void*)i);
}

void *server(void *arg) {
    → int k=(int)arg;
    // perform service
    return(0);
}

```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - ◊ one arg in each of these stack frames

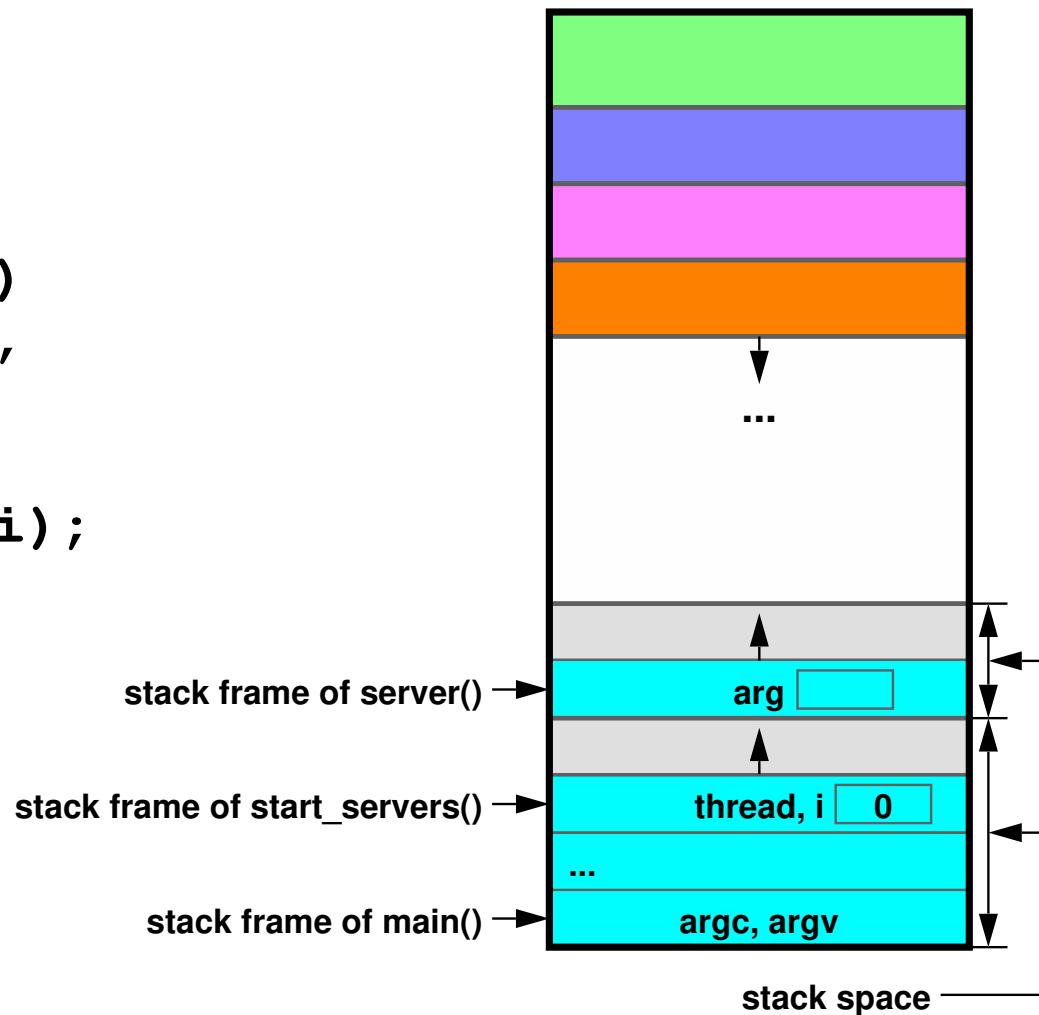
# Creating a POSIX Thread

```

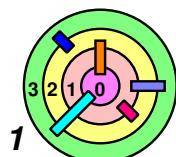
start_servers( ) {
    pthread_t thread;
    int i;
    for (i = 0; i < 100; i++)
        → pthread_create(&thread,
                         0,
                         server,
                         (void*)i);
}

void *server(void *arg) {
    → int k=(int)arg;
    // perform service
    return(0);
}

```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - ◊ one arg in each of these stack frames



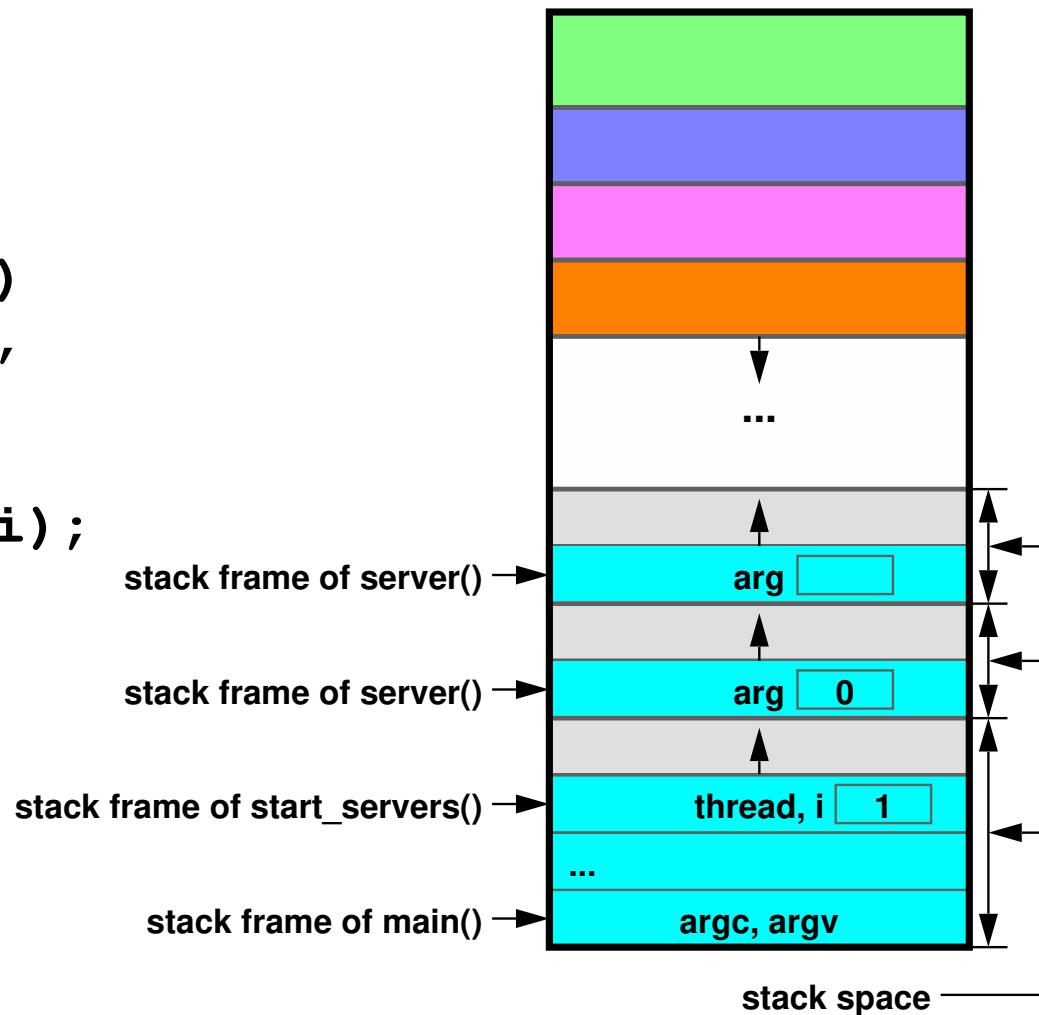
# Creating a POSIX Thread

```

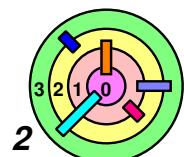
start_servers( ) {
    pthread_t thread;
    int i;
    for (i = 0; i < 100; i++)
        → pthread_create(&thread,
                         0,
                         server,
                         (void*)i);
}

void *server(void *arg) {
    → int k=(int)arg;
    // perform service
    return(0);
}

```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - ◊ one arg in each of these stack frames



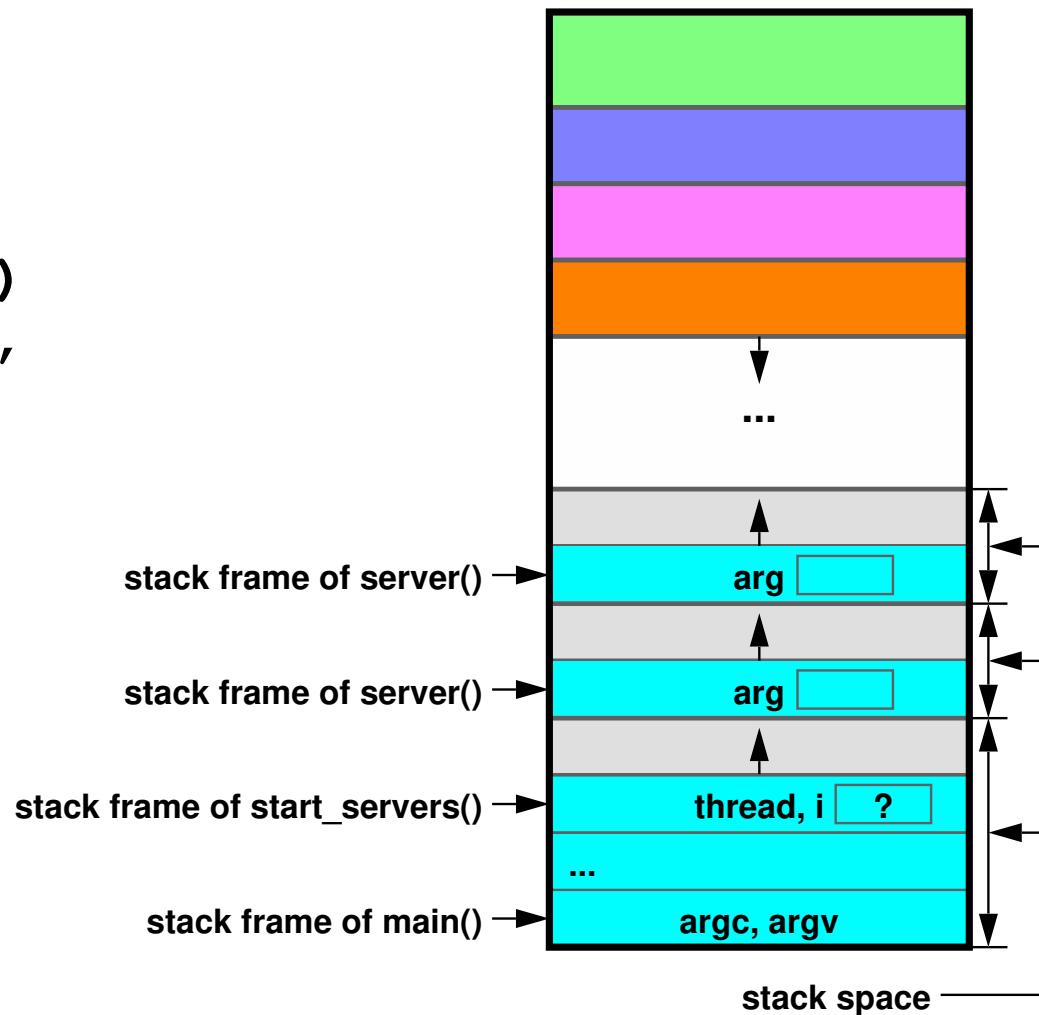
# Creating a POSIX Thread

```

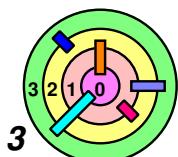
start_servers( ) {
    pthread_t thread;
    int i;
    for (i = 0; i < 100; i++)
        → pthread_create(&thread,
                          0,
                          server,
                          &i);
}

void *server(void *arg) {
    → int *iptr=(int*)arg;
    // perform service
    return(0);
}

```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - ◊ one arg in each of these stack frames



# Creating a POSIX Thread



These are the same:

- keep thread handle in the stack

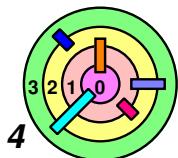
```
pthread_t thread;  
pthread_create(&thread, ...);
```

- keep thread handle in the heap

```
pthread_t *thread_ptr =  
    (pthread_t*)malloc(sizeof(pthread_t));  
pthread_create(thread_ptr, ...);
```

- need to make sure that eventually you will call the following to not leak memory

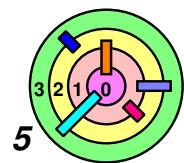
```
free(thread_ptr);
```



# Creating a Win32 Thread

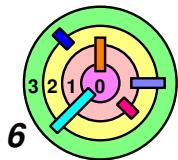
```
start_servers( ) {  
    HANDLE thread;  
    DWORD id;  
    int i;  
    for (i = 0; i < 100; i++)  
        thread = CreateThread(  
            0,          // security attributes  
            0,          // default # of stack pages allocated  
            server,    // first procedure  
            arg,       // argument  
            0,          // default attributes  
            0,          // creation flags  
            &id);      // thread ID  
}  
  
DWORD WINAPI server(void *arg) {  
    // perform service  
    return(0);  
}
```

- We won't talk about Win32 much



# Complications

```
rlogin(int r_in, int r_out, int l_in, int l_out) {  
    pthread_t in_thread, out_thread;  
  
    pthread_create(&in_thread,  
                  0,  
                  incoming,  
                  r_in, l_out); // Cannot do this ...  
    pthread_create(&out_thread,  
                  0,  
                  outgoing,  
                  l_in, r_out); // Cannot do this ...  
    /* How do we wait till they are done? */  
}
```



# Multiple Arguments

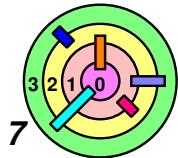
```
typedef struct {
    int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;

    two_ints_t in={r_in, l_out}, out={l_in, r_out};
    pthread_create(&in_thread,
                  0,
                  incoming,
                  &in);

    ...
    /* How do we wait till they are done? */
}

void *incoming(void *arg) {
    two_ints_t *p=(two_ints_t*)arg;
    ... p->first ...
    return NULL;
}
```



# Multiple Arguments

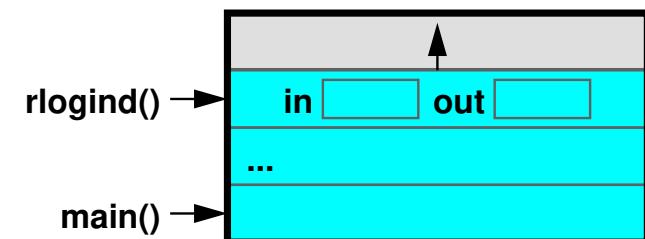
```
typedef struct {
    int first, second;
} two_ints_t;
```

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;

    two_ints_t in={r_in, l_out}, out={l_in, r_out};
    pthread_create(&in_thread,
                  0,
                  incoming,
                  &in);

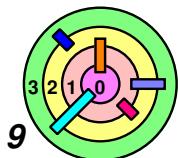
    ...
    /* How do we wait till they are done? */
}
```

```
void *incoming(void *arg) {
    two_ints_t *p=(two_ints_t*)arg;
    ... p->first ...
    return NULL;
}
```



# Multiple Arguments

- Need to be careful how to pass argument to new thread when you call `pthread_create()`
  - passing address of a *local* variable (like the previous example) only works if we are certain the this storage doesn't go out of scope until the thread is done with it
  - passing address of a *static* or a *global* variable only works if we are certain that only one thread at a time is using the storage
  - passing address of a *dynamically* allocated storage only works if we can free the storage when, and only when, the thread is finished with it
    - this would not be a problem if the language supports garbage collection
- Memory corruption happens when memory is re-used unexpectedly
  - ask yourself, "How can I be sure?"
    - if the answer is, "I hope it works", then you need a different solution



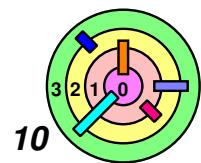
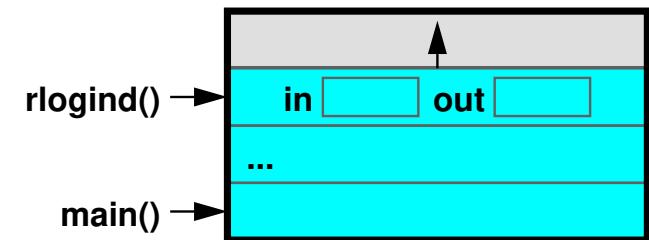
# When Is The Child Thread Done?

- In our example, what would happen if we return from `rlogind()` right after the child threads are created?

```
typedef struct {
    int first, second;
} two_ints_t;
```

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};
    pthread_create(&in_thread,
                  0,
                  incoming,
                  &in);
}

void *incoming(void *arg) {
    two_ints_t *p=(two_ints_t*)arg;
    ... p->first ...
    return NULL;
}
```



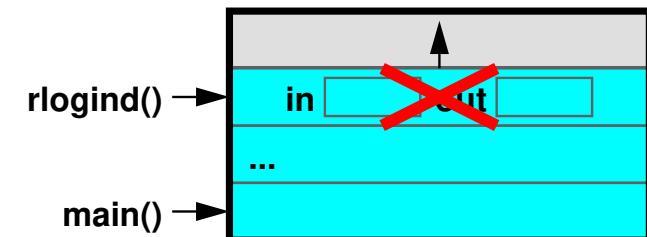
# When Is The Child Thread Done?

- In our example, what would happen if we return from `rlogind()` right after the child threads are created?

```
typedef struct {
    int first, second;
} two_ints_t;
```

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};
    pthread_create(&in_thread,
                  0,
                  incoming,
                  &in);
}
```

```
void *incoming(void *arg) {
    two_ints_t *p=(two_ints_t*)arg;
    ... p->first ...
    return NULL;
}
```



# When Is The Child Thread Done?

→ To wait for a child thread to die, use `pthread_join()`

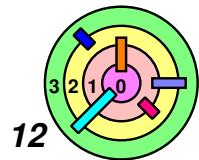
```
int pthread_join(pthread_t thread,
                 (void **) ret_value);
```

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};

    pthread_create(&in_thread, 0, incoming, &in);
    pthread_create(&out_thread, 0, outgoing, &out);

    /* if not interested in thread return values */
    pthread_join(in_thread, 0);
    pthread_join(out_thread, 0);
}
```

- `(void**)` is the address of a variable of type `(void*)`
- `pthread_join()` is a *blocking call*
  - can only return if the specified thread has terminated



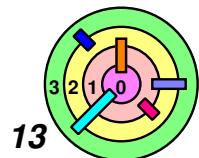
# Thread Termination



## Thread return values

- which threads receive these values
- how do they do it?
  - clearly, receiving thread must wait until the producer thread produced it, i.e., producer thread has terminated
  - so we must have a way for one thread to wait for another thread to terminate
- must have a way to say which thread you are waiting for
  - need a unique identifier
  - tricky if it can be reused

```
int pthread_join(pthread_t thread,
                 (void **) ret_value);
```



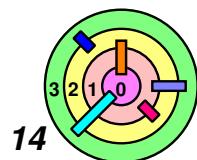
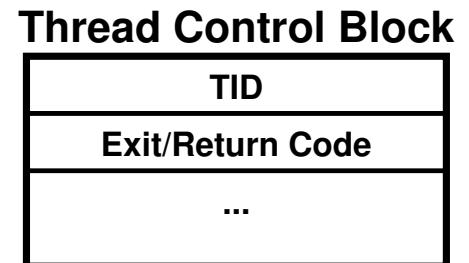
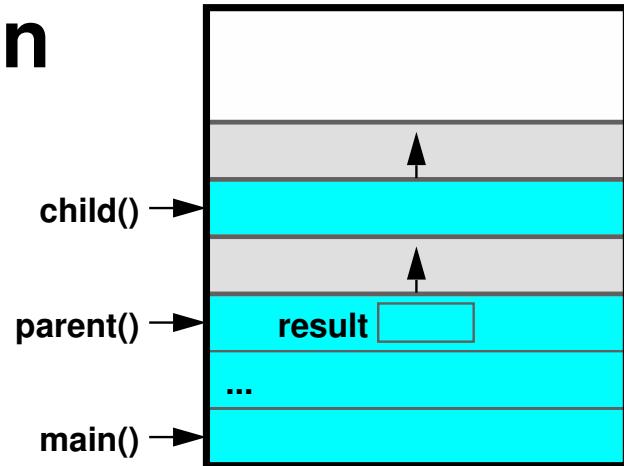
# Thread Termination

→ How does a thread **self-terminate**?

- 1) return from its "first procedure"
  - ◊ return a value of type `(void*)`
- 2) call `pthread_exit(ret_value)`
  - ◊ `ret_value` is of type `(void*)`

```
parent() {
    pthread_t thread;
    void *result=(void*) 0;
    pthread_create(&thread,
                  0, child, 0);
    pthread_join(thread,
                 (void**)&result);
    switch ((int)result) {
        case 1: ...
        case 2: ...
    }
    ...
}
```

```
void *child(void *arg) {
    ...
    if (terminate_now) {
        pthread_exit((void*) 1);
    }
    return ((void*) 2);
}
```



# Thread Termination

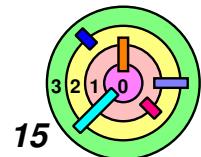
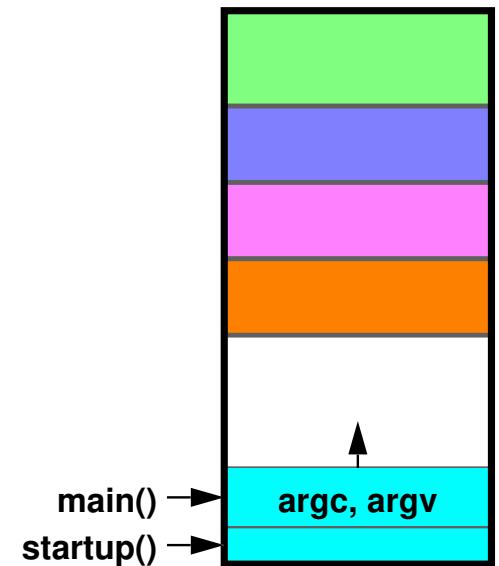
- Difference between `pthread_exit()` and `exit()`
- `pthread_exit()` terminates only the calling thread
  - `exit()` terminates the process, including all threads running in it
    - it will not wait for any thread to terminate
    - what will this code do?

```
int main(int argc, char *argv[]) {
    // create all the threads
    return(0);
}
```

- when `main()` returns, `exit()` will be called
  - ◊ as a result, none of the created child threads may get a chance to run

- `main()` is called by a "*startup routine*":

```
exit(main(argc, argv))
```



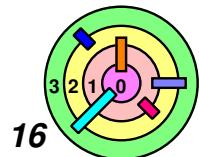
# Thread Termination

- Difference between `pthread_exit()` and `exit()`
- `pthread_exit()` terminates only the calling thread
  - `exit()` terminates the process, including all threads running in it
    - it will not wait for any thread to terminate
    - what about this code?

```
int main(int argc, char *argv[]) {  
    // create all the threads  
    pthread_exit(0); // exit the main thread  
    return(0);  
}
```

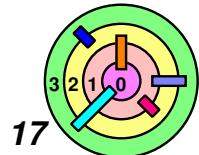
- here, `pthread_exit()` will terminate the main thread, so `exit()` is never called
  - ◊ as it turns out, this special case is taken care of in the `pthread` library implementation

- You should use `pthread_join()` unless you are absolutely sure what you are doing



# Thread Termination

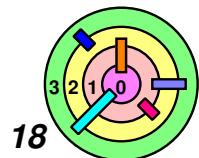
- Calling `pthread_exit()` is the only way a thread can ***self-terminate*** (without affecting other threads)
- Any thread can join with any other thread
  - there's ***no parent/child relationships*** among threads
    - unlike process termination and `wait()`
- What happens if a thread terminates and no other thread wants to join with this thread?
  - it also goes into a ***zombie state***
    - all the thread related information is freed up, except for the thread ID, return code, and stack space
      - ◆ a running thread cannot delete its own stack!
- What if two threads want to join with the same thread?
  - after the first thread joins, the thread ID and return code are freed up and the thread ID may get reused
  - so don't do this!



# Detached Threads

- What if you have a thread that you don't want any thread to join with it?
- you can detach the thread after you have created it

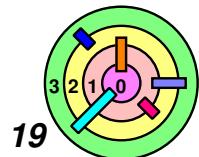
```
start_servers( ) {  
    pthread_t thread;  
    int i;  
    for (i = 0; i < 100; i++) {  
        pthread_create(&thread, 0, server, 0);  
        pthread_detach(thread);  
    }  
    ...  
}  
  
server( ) {  
    ...  
}
```



# Types

```
pthread_create(&tid,  
              0,  
              (void *(*)(void *))func,  
              (void *)1);  
  
int func = 4; // func definition 1  
  
void func(int i) { // func definition 2  
    ...  
}  
  
void *func(void *arg) { // func definition 3  
    int i = (int)arg;  
    ...  
    return(0);  
}
```

- a function is just an address (of something in the text/code segment)



# Thread Attributes

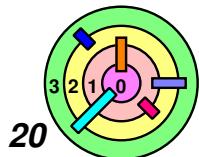
```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
/* establish some attributes */

...
pthread_create(&thread, &thr_attr, startroutine, arg);
pthread_attr_destroy(&thr_attr);

...
```

- **thread attribute only needs to be valid when a thread is created**
  - **therefore, it can be destroyed as soon as the thread is created**

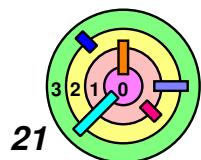


# Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);
...
pthread_create(&thread, &thr_attr, startroutine, arg);
pthread_attr_destroy(&thr_attr);
```

- the above code set the stack size to 20MB
- the default stack size is not small
  - default stack size is probably around 1MB in Solaris and 8MB in some Linux implementations
  - if you need to create a lot of threads, you may want to have smaller stack size
  - if you have very deep recursion in your code, you may want a bigger stack size



# Example

```

#include <stdio.h>
#include <pthread.h>
#include <string.h>

#define M 3
#define N 4
#define P 5

int A[M][N];
int B[N][P];
int C[M][P];

void *matmult(void *arg) {
    int row = (int)arg, col;
    int i, t;

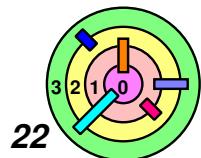
    for (col=0; col < P; col++) {
        t = 0;
        for (i=0; i<N; i++)
            t += A[row][i] * B[i][col];
        C[row][col] = t;
    }
    return(0);
}

main( ) {
    int i;
    pthread_t thr[M];
    int error;

    /* initialize the matrices ... */

    ...
    // create the worker threads
    for (i=0; i<M; i++) {
        if (error = pthread_create(
            &thr[i],
            0,
            matmult,
            (void *)i)) {
            fprintf(stderr,
                    "pthread_create: %s",
                    strerror(error));
            exit(1);
        }
    }
    // wait for workers to finish
    for (i=0; i<M; i++)
        pthread_join(thr[i], 0)
    /* print the results ... */
}

```



# Compiling It

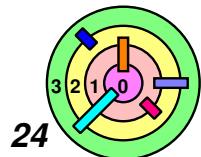
```
% gcc -o mat mat.c -pthread
```

- technically speaking, "**-pthread**" is for *linking* with the pthread library
- **compiling** is to compile "mat.c" into "mat.o"
  - ◆ "mat.o" is deleted after the "mat" executable file is created
- another syntax is (for Unix systems in general):

```
% gcc -o mat mat.c -lpthread
```

## 2.2.3 Synchronization

- ➔ In real life, "synchronization" means that you want to do things at the same time
- ➔ In computer science, "synchronization" could mean the above, **OR**, it means that you want to *prevent* do things at the same time



# Mutual Exclusion



Also see <https://en.wikipedia.org/wiki/Therac-25>

# Threads and Mutual Exclusion

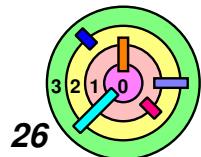
Thread 1:

$x = x+1;$

Thread 2:

$x = x+1;$

- looks like it doesn't matter how you execute, x will be incremented by 2 in the end
  - choices are
    - ◊ thread 1 executes  $x = x+1$  then thread 2 executes  $x = x+1$
    - ◊ thread 2 executes  $x = x+1$  then thread 1 executes  $x = x+1$
  - are there other choices?



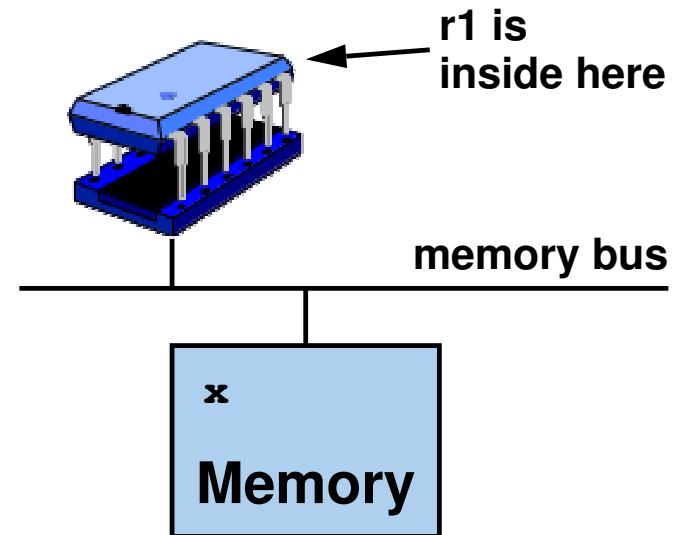
# Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
/*
ld  r1,x
add r1,1
st  r1,x
*/
```

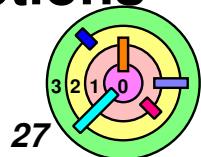
Thread 2:

```
x = x+1;
/*
ld  r1,x
add r1,1
st  r1,x
*/
```



→ Unfortunately, machines do not execute high-level language statements

- they execute machine instructions
- now if thread 1 executes the first (or two) machine instructions
- context **switch** can happen (to run a different thread)
  - this **can** happen if you have a *preemptive scheduler*
- then thread 2 executes all 3 machine instructions
- then later thread 1 executes the remaining machine instructions
- x would have only increased by 1



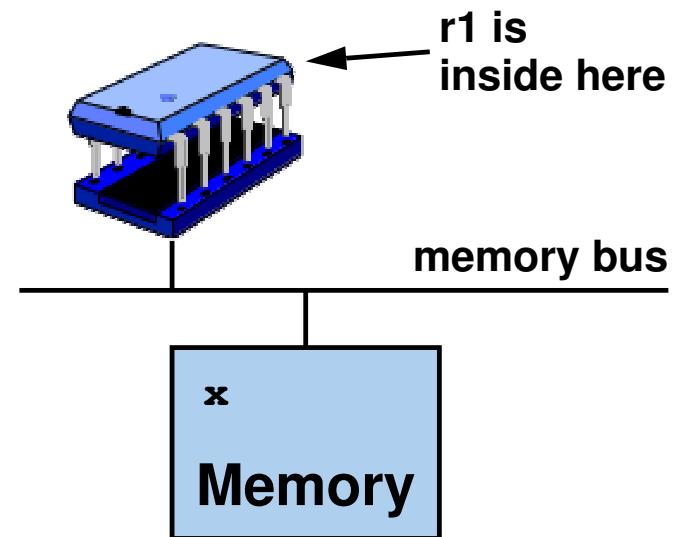
# Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
/*
ld  r1,x
add r1,1
st  r1,x
*/
```

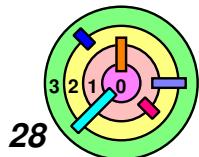
Thread 2:

```
x = x+1;
/*
ld  r1,x
add r1,1
st  r1,x
*/
```



We want  $x=x+1$  to be executed ***atomically***

- = ***atomically*** means that the 3 machine instructions are ***locked*** together
  - if you execute the first machine instruction, you must execute all 3 ***without interruption***
- = ***atomicity*** is an ***abstraction***
  - it's important to understand ***exactly*** what it means to be ***atomic***



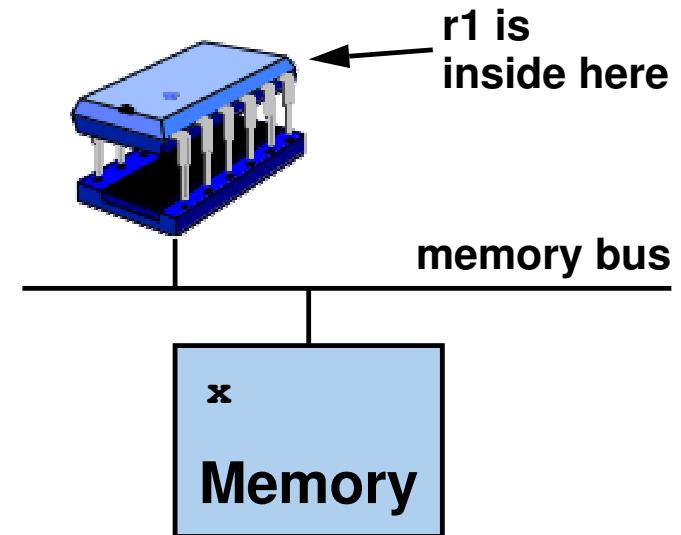
# Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
/*
ld  r1,x
add r1,1
st  r1,x
*/
```

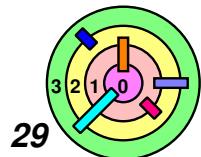
Thread 2:

```
x = x+1;
/*
ld  r1,x
add r1,1
st  r1,x
*/
```



## Atomic operation

- if you execute the first machine instruction, can the CPU go do something else (e.g., handle a hardware interrupt)?
  - yes!
- what does atomic really mean if you can go do something else?
  - it means *atomic, with respect to the variables involved*
    - ◊ in this example, it's just x
    - ◊ you *can* do something else that does not involve x
  - more involved in general as we will see soon



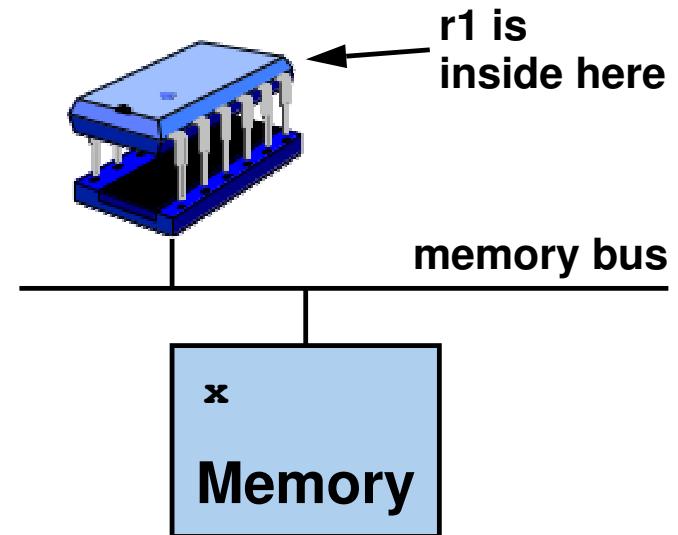
# Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
/*
ld  r1,x
add r1,1
st  r1,x
*/
```

Thread 2:

```
x = x+1;
/*
ld  r1,x
add r1,1
st  r1,x
*/
```

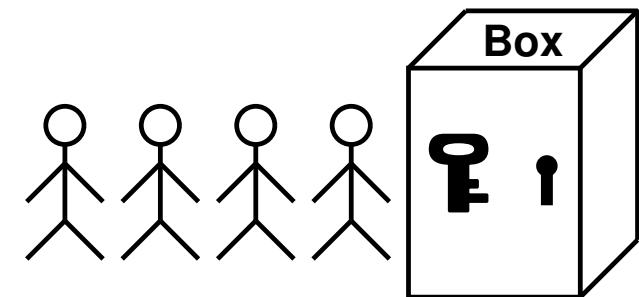


## → **Atomic** operation

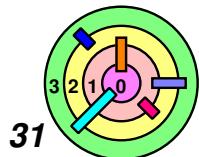
- every time you talk about an ***atomic operation***, you need to be very clear about exactly what it is ***with respect to***
- we will use a visual aid

# Threads and Synchronization

- Solution: put  $x$  in an (abstract) safe-deposit box under lock and key
  - whoever has the key gets to use  $x$ 
    - what if you fall asleep while inside the box?
      - ◊ no problem, others will just have to *wait*
    - isn't that inefficient?
      - ◊ correctness is more important
      - ◊ if you know you will fall asleep inside the box, you should be nice to others (and be more efficient) by getting out of the box and get in line later



- Rule for accessing  $x$  from now on:
  - you can only access  $x$  using an atomic operation
- What if you have more than one variable (e.g.,  $x$ ,  $y$ ,  $z$ )?
  - put all of them inside one safe-deposit box
  - you can only access  $x$ ,  $y$ ,  $z$  *atomically, with respect to the operation of the "box"*

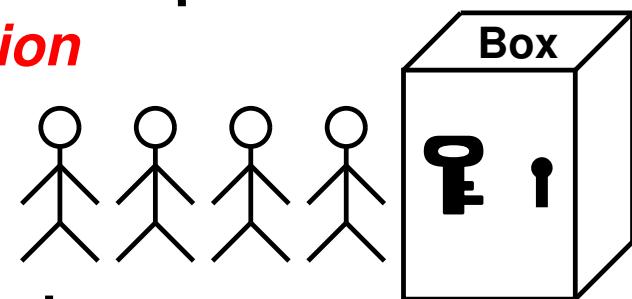


# Threads and Synchronization

```
// shared by both threads
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int x;
...
pthread_mutex_lock(&m);
    x = x+1;
}
pthread_mutex_unlock(&m);
```

➡ Locking a mutex is like getting the key to a safe-deposit box

*critical section*



- code between `pthread_mutex_lock()` and `pthread_mutex_unlock()` for a particular **mutex** is called a ***critical section with respect to that mutex***
- **all** the critical sections ***with respect to a particular mutex*** are "***mutually exclusive***"
  - ◊ the system (not necessarily the OS) guarantees that only **one** critical section can be executing at any point in time ***with respect to a particular mutex***
- **how it's really done will be covered in Ch 5**

# Set Up

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

## → Mutex initialization

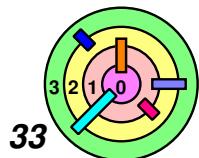
- mutex is unlocked
- initialize data structure (initially empty) used to keep track of *waiting threads*

## → If a mutex cannot be initialized statically, do:

```
int pthread_mutex_init(
    pthread_mutex_t *mutexp,
    pthread_mutexattr_t *attrp)
```

```
int pthread_mutex_destroy(
    pthread_mutex_t *mutexp)
```

## → Usually, mutex attributes are not used



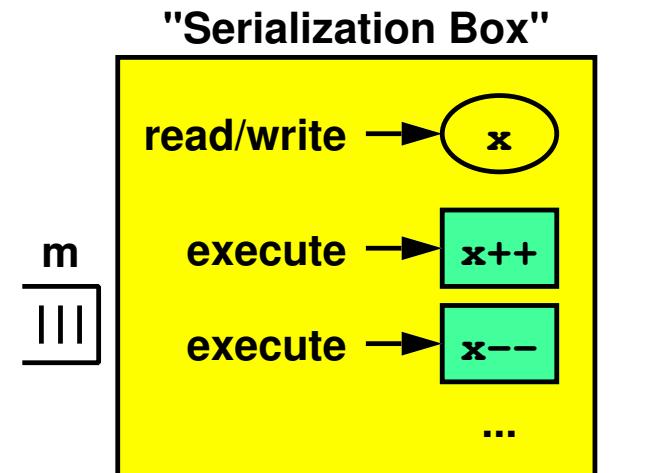
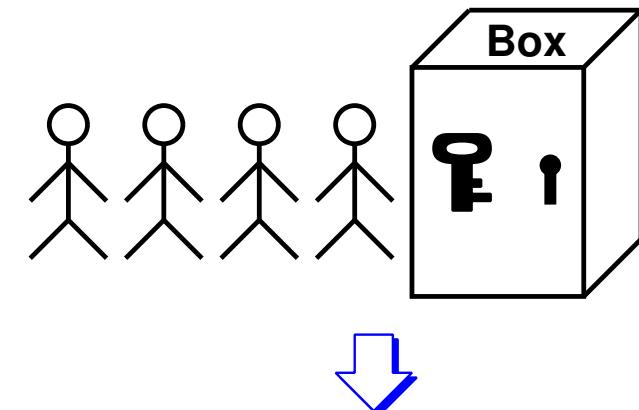
# One Mutex, Multiple Critical Sections

```

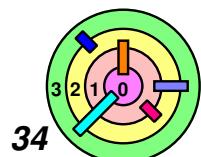
f1 () {
    pthread_mutex_lock (&m) ;
    x++; } critical section
    pthread_mutex_unlock (&m) ;
}

f2 () {
    pthread_mutex_lock (&m) ;
    x--; } critical section
    pthread_mutex_unlock (&m) ;
}

```



- only one thread can be running inside the "Serialization Box" at a time (access to the "box" is "serialized" or "synchronized")
  - you should **only** access shared variables using ***critical section code***
- the "Serialization Box" is not a real box, it's conceptual



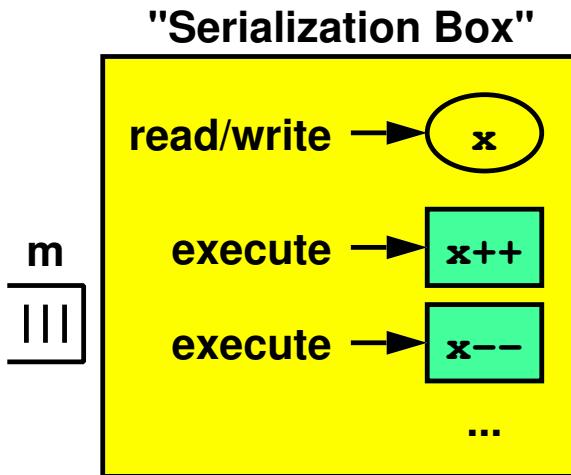
# One Mutex, Multiple Critical Sections

```

f1 () {
    pthread_mutex_lock (&m) ;
    x++; } critical section
    pthread_mutex_unlock (&m) ;
}

f2 () {
    pthread_mutex_lock (&m) ;
    x--; } critical section
    pthread_mutex_unlock (&m) ;
}

```



- By calling `pthread_mutex_lock (&m)`, a thread can be placed into a **queue** and **wait** there **indefinitely** for mutex `m` to become available
  - multiple threads would join this queue
  - queue is served one at a time, like a supermarket checkout
  - when it's your thread's turn, `pthread_mutex_lock ()` returns with the mutex locked, your thread can execute critical section code, and then release the mutex

# Taking Multiple Locks



**Mutex is not a cure-all**

- when you have more than one locks, you may get into trouble

```
proc1( ) {  
    pthread_mutex_lock(&m1) ;  
    /* use object 1 */  
    pthread_mutex_lock(&m2) ;  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m2) ;  
    pthread_mutex_unlock(&m1) ;  
}
```

```
proc2( ) {  
    pthread_mutex_lock(&m2) ;  
    /* use object 2 */  
    pthread_mutex_lock(&m1) ;  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m1) ;  
    pthread_mutex_unlock(&m2) ;  
}
```

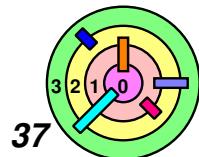
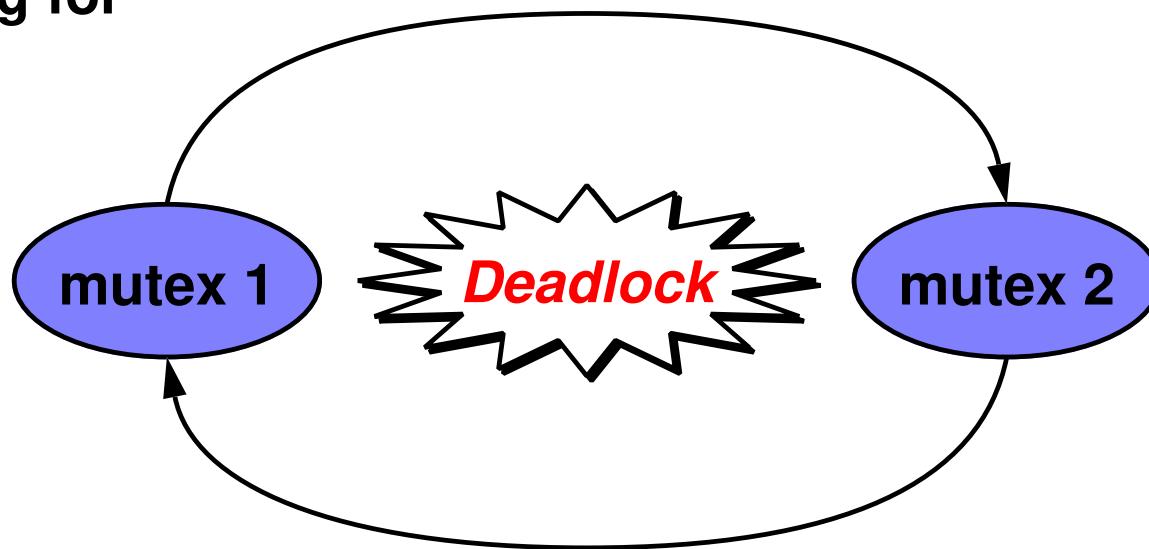
# Taking Multiple Locks

```
proc1( ) {
    pthread_mutex_lock(&m1);
    /* use object 1 */
    pthread_mutex_lock(&m2);
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
    pthread_mutex_lock(&m2);
    /* use object 2 */
    pthread_mutex_lock(&m1);
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
}
```

→ Graph representation ("wait-for" graph) for the entire process

- draw an arrow from a mutex you are holding to another mutex you are waiting for



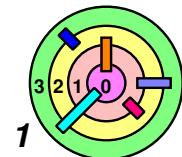
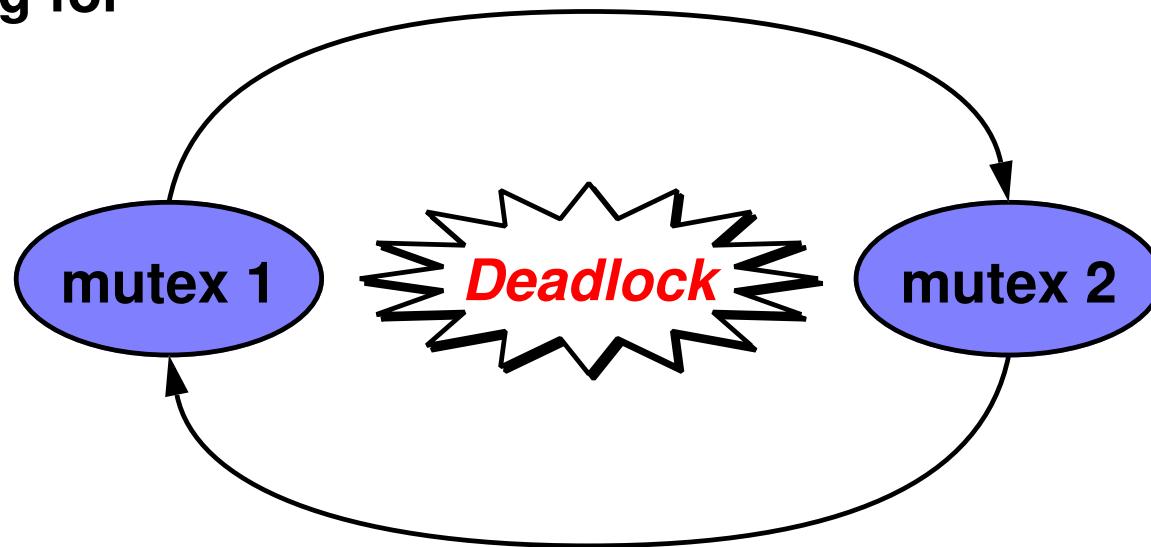
# Taking Multiple Locks

```
proc1( ) {
    pthread_mutex_lock(&m1);
    /* use object 1 */
    pthread_mutex_lock(&m2);
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
    pthread_mutex_lock(&m2);
    /* use object 2 */
    pthread_mutex_lock(&m1);
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
}
```

→ Graph representation ("wait-for" graph) for the entire process

- draw an arrow from a mutex you are holding to another mutex you are waiting for



# Necessary Conditions For Deadlocks

→ All 4 conditions below must be met in order for a deadlock to be possible (no guarantee that a deadlock may occur)

## 1) Bounded resources

- only a finite number of threads can have concurrent access to a resource

## 2) Wait for resources

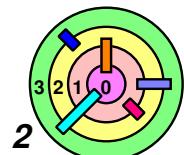
- threads wait for resources to be freed up, *without releasing* resources that they hold

## 3) No preemption

- resources cannot be *revoked* from a thread

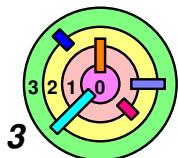
## 4) Circular wait

- there exists a set of waiting threads, such that each thread is waiting for a resource held by another

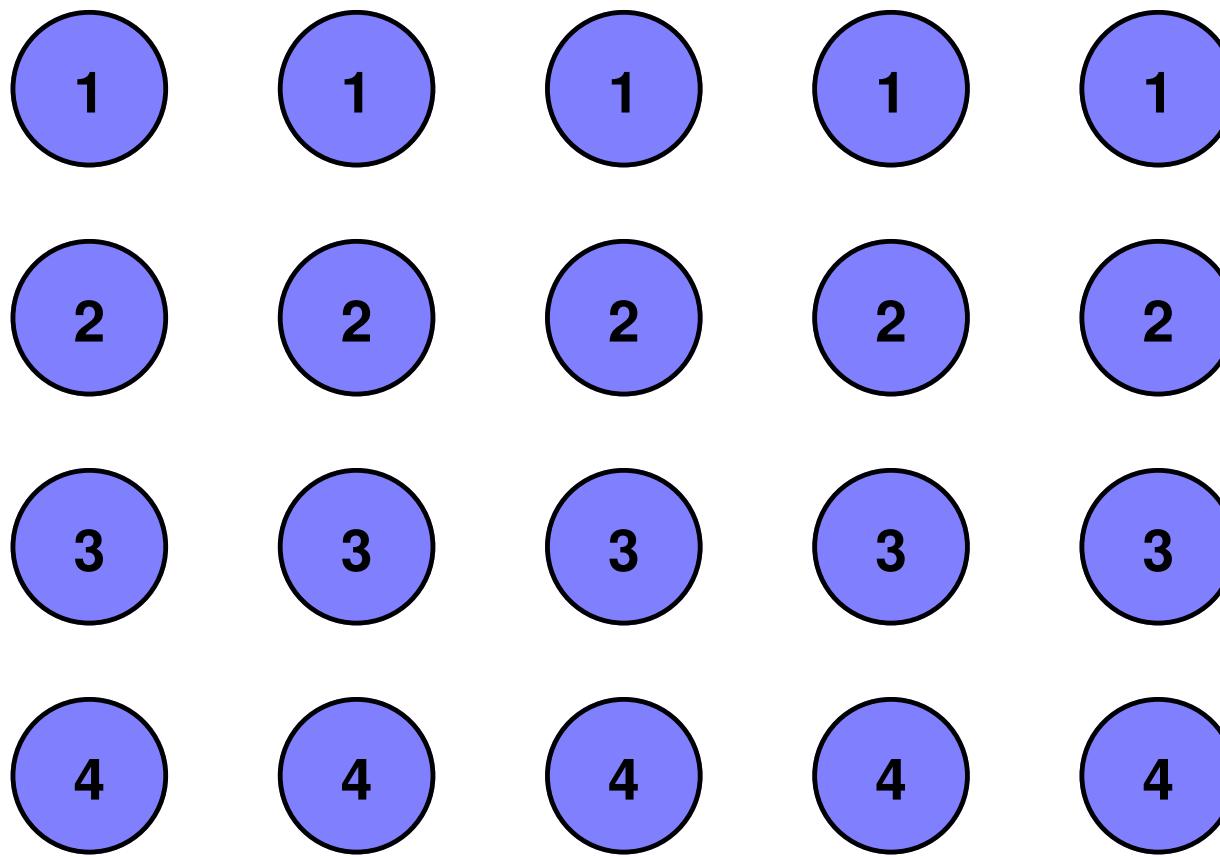


# Dealing with Deadlock

- Deadlock is a programming bug
  - one of the oldest bug
  - it's a tricky one because it only deadlocks *sometimes*
- Hard
  - is the system deadlocked?
  - will this move lead to deadlock?
  - this is *detection*
    - if you can detect deadlocks, what do you do after you have detected them?
- Easy
  - restrict use of mutexes so that deadlock cannot happen
  - this is *prevention*
- Deadlock is a complicated subject
  - some textbooks spend an entire chapter on deadlocks
  - we will only look at a couple of cases

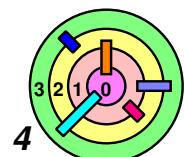


# Deadlock Prevention: Lock Hierarchies

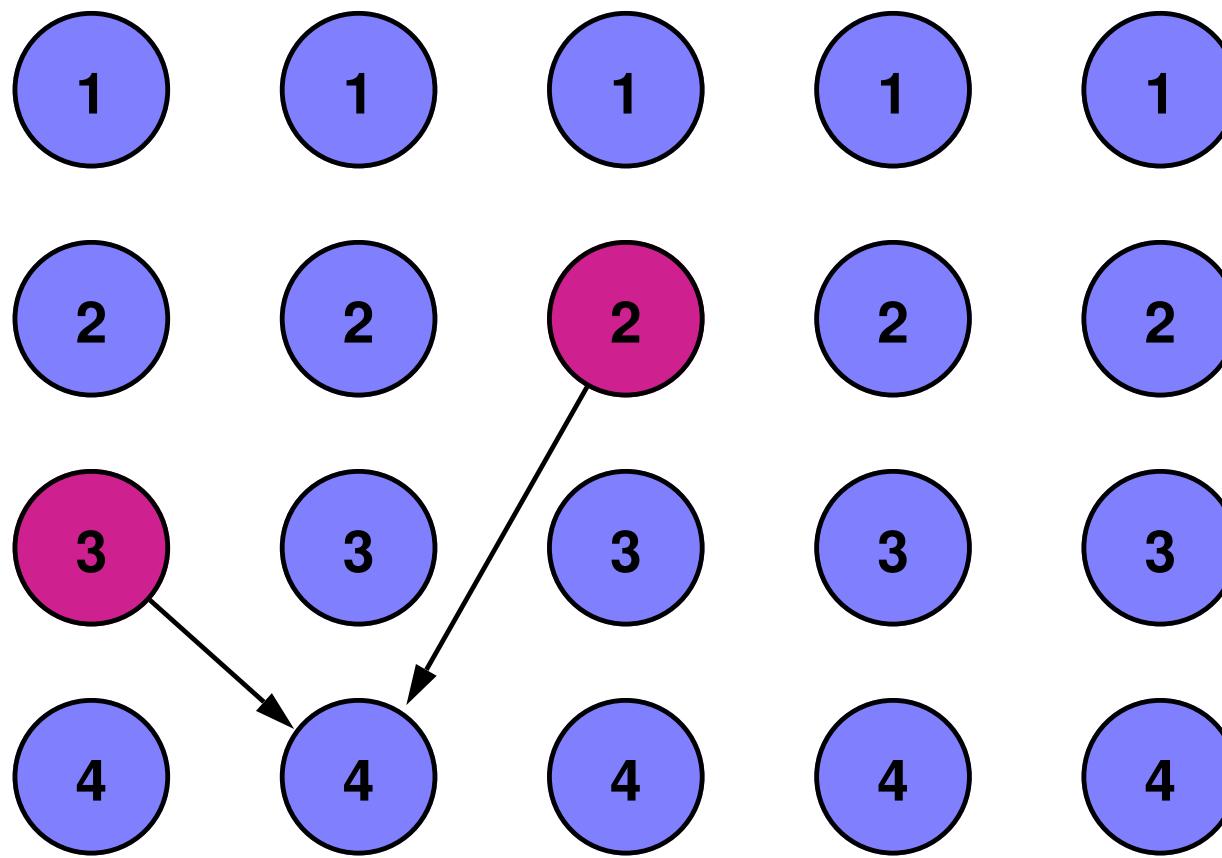


→ **If** you can organize mutexes into levels and satisfies:

- must not try locking a mutex at level *i* if already holding a mutex at equal or higher level, otherwise it's okay



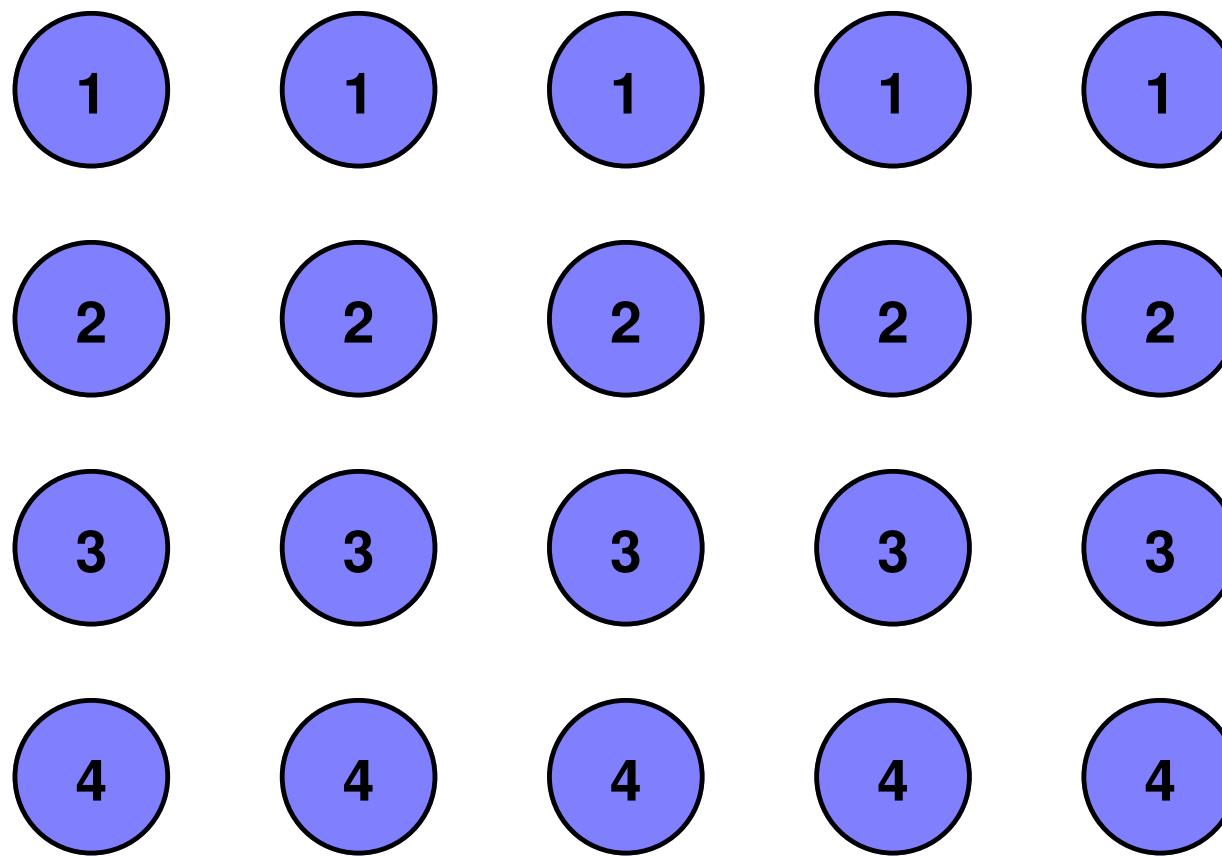
# Deadlock Prevention: Lock Hierarchies



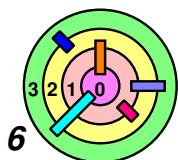
→ If you can organize mutexes into levels and satisfies:

- must not try locking a mutex at level *i* if already holding a mutex at equal or higher level, otherwise it's okay
  - e.g., if holding mutexes at levels 2 and 3, can only wait for a mutex at levels 4 or higher

# Deadlock Prevention: Lock Hierarchies



- What if you cannot organize your mutexes in such strict order for deadlock prevention?
- can we avoid "necessary conditions" for deadlocks (1), (2), or (3)?

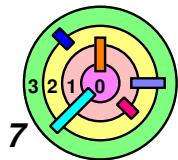


# Deadlock Prevention: Conditional Locking

```
proc1( ) {
    pthread_mutex_lock(&m1);
    /* use object 1 */
    pthread_mutex_lock(&m2);
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);
}

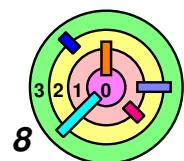
proc2( ) {
    while (1) {
        pthread_mutex_lock(&m2);

        if (!pthread_mutex_trylock(&m1))
            break;
        pthread_mutex_unlock(&m2);
    }
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
}
```



# Mutex Summary

- How do threads interact with each other?
  - they interact with each other by calling pthread functions
    - e.g., `pthread_mutex_lock()`, `pthread_mutex_unlock()`
  - they interact with each other using ***shared variables***
- If multiple threads want to ***share read-only data*** (i.e., no thread will ever modify the shared data)
  - you don't need to use mutex
- If multiple threads want to share data for ***reading and writing*** (or just for writing, which is unusual)
  - all these threads must share a mutex and only access the shared data using ***critical section code with respect to this mutex***
  - in general, critical section code may be ***nested*** (as in the case of locking hierarchy)
    - also, a thread may be using different mutexes to interact with different threads



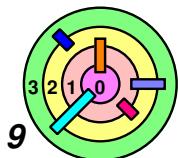
F20-Q13

su21-den-Q6

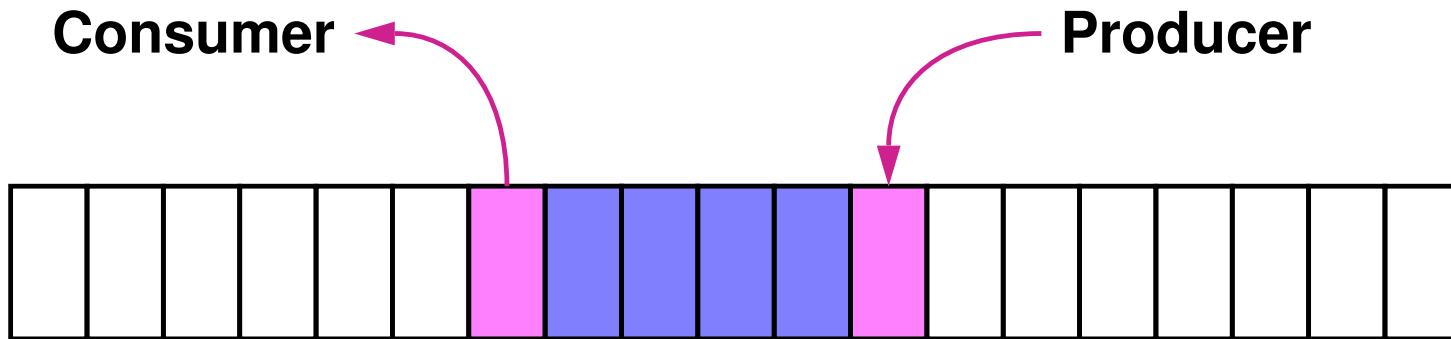
# Beyond Mutexes



- Mutex is necessary when shared data is being modified
  - although there are cases where using a mutex is an **overkill** (i.e., too restrictive and inefficient and would lock threads out when it's not necessary)
    - can always use one mutex to lock up all shared data
      - ◆ no parallelism (when *some parallelism can be* permitted)
    - we would like to have better concurrency (i.e., "**fine-grained parallelism**") when *complete mutual exclusion is not required*
  - two major categories to illustrate this
    - 1) what if threads don't interfere one another most of the time and synchronization is only required occasionally?
      - ◆ e.g., **Producer-Consumer problem** (a.k.a., bounded-buffer problem)
    - 2) what if some threads just want to *look at* (i.e., *read*) a piece of data?
      - ◆ e.g., **Readers-Writers problem**
  - will also look at **Barrier Synchronization**

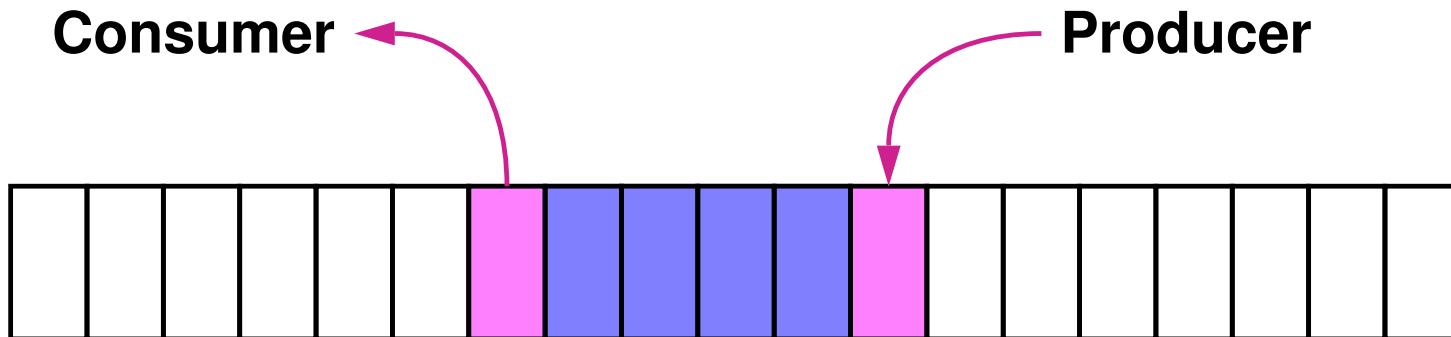


# Producer-Consumer Problem

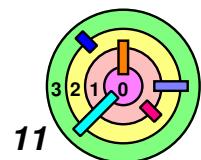


- **Conveyor belt (hardware)**
- perfect parallelism between producer and consumer

# Producer-Consumer Problem



- ➔ **Conveyor belt (hardware)**
  - perfect parallelism between producer and consumer
- ➔ **A circular buffer is used in software implementation**
- ➔ **Most of the time, no interference**
  - if you use a **single mutex** to lock the entire array of buffers, it's an **overkill** (i.e., too **inefficient**)
- ➔ **When does it require synchronization?**
  - producer needs to be blocked when all slots are full
  - consumer needs to be blocked when all slots are empty



# Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
  */
  ...
]
```

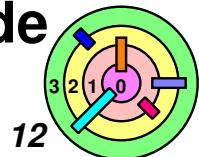
}

*command sequence*

- this means that the command sequence is executed (*atomically*) **when the guard is evaluated** to be **true**
  - a **guard** is a **boolean expression** (evaluates to true or false)
  - **atomically** mean that it's executed "without interruption"
    - ◊ but with respect to what?
    - ◊ **evaluating the guard** and **executing the command sequence** altogether is an **atomic operation if** the **guard is true**
    - ◊ you cannot evaluate the guard if your thread is not running



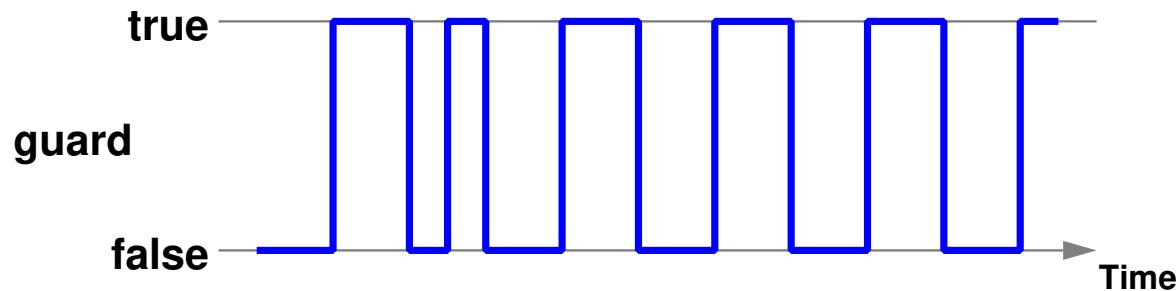
For exams, you need to know how to write simple pesudo-code in the language of **Guarded Commands**



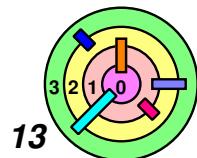
# Guarded Commands

```
when (guard) [
/*  
once the guard is true,  
execute this code atomically  
*/  
...  
]
```

A curly brace on the right side of the code block groups the entire block under the label **command sequence**.

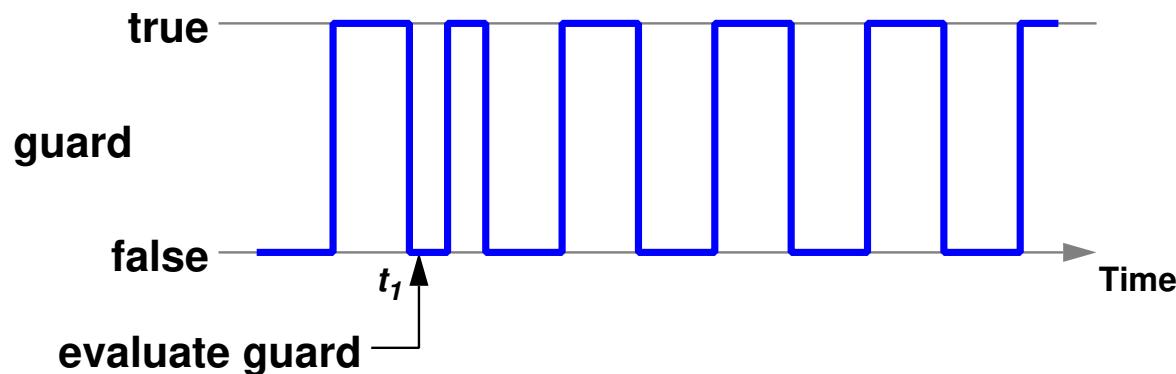


- you can only evaluate the guard if your thread is running

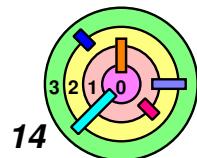


# Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
  */
  ...
]
```

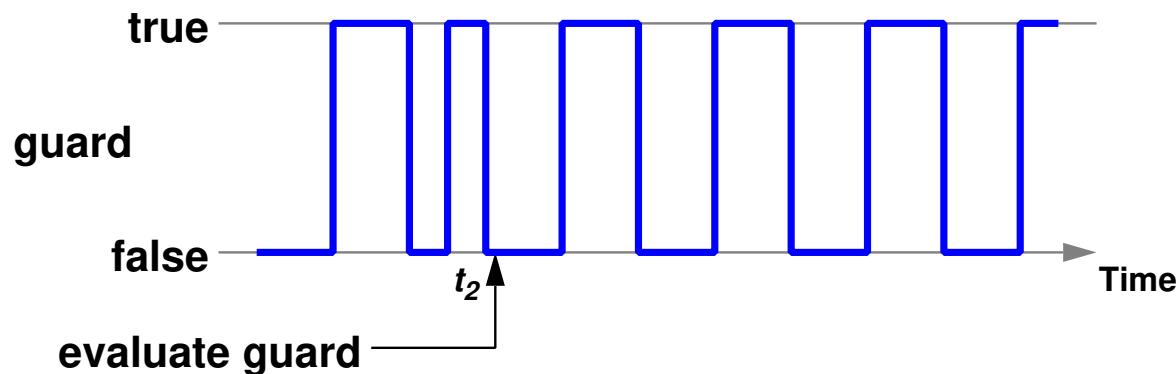


- you can only evaluate the guard if your thread is running

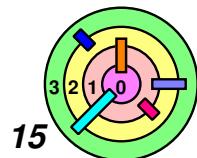


# Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
  */
  ...
]
```



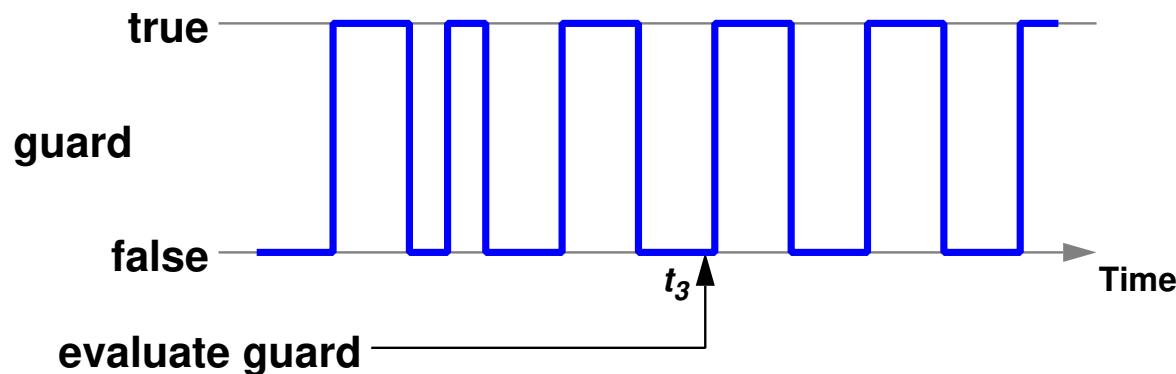
- you can only evaluate the guard if your thread is running



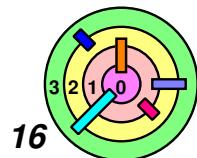
# Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
  */
  ...
]
```

} *command sequence*



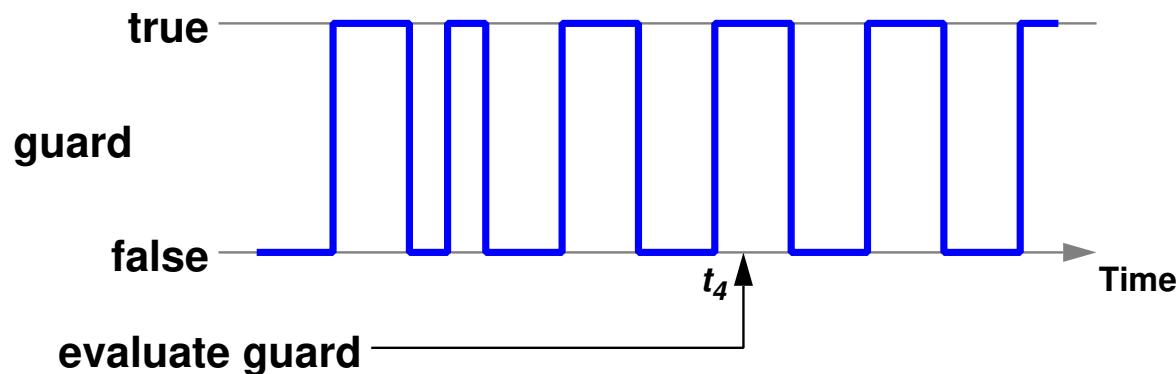
- you can only evaluate the guard if your thread is running



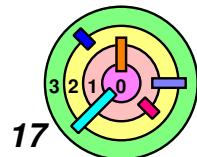
# Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
  */
  ...
]
```

} *command sequence*



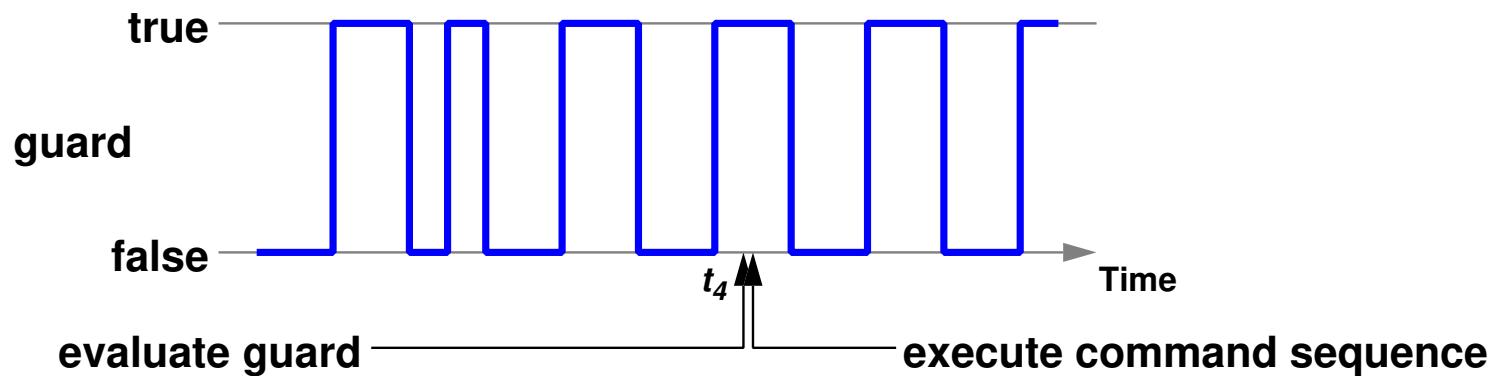
- you can only evaluate the guard if your thread is running



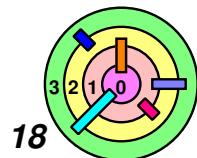
# Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
  */
  ...
]
```

A curly brace on the right side of the code block groups the code block under the label **command sequence**.

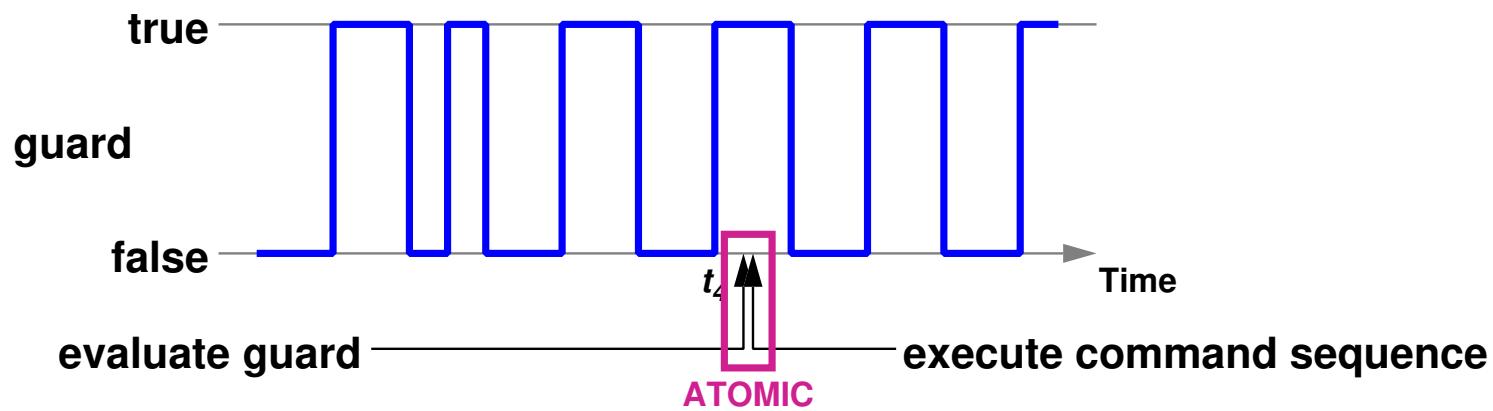


- you can only evaluate the guard if your thread is running



# Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
  */
  ...
]
```



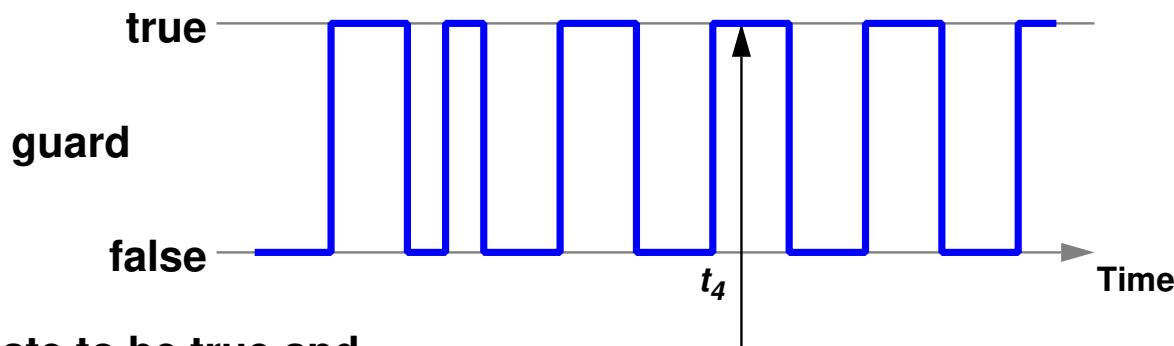
- please understand that ***command sequence* ≠ *critical section***
  - evaluate the guard to be true ***and*** execute command sequence ***together*** is done inside ***one critical section***

# Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
  */
  ...
]
```

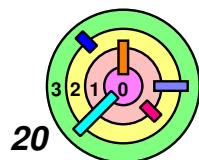
}

***command sequence***

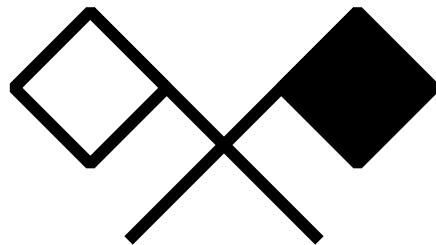


guard evaluate to be true and  
execute command sequence

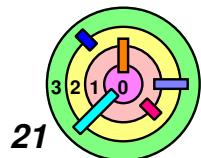
- ***atomic:*** as if it's executed in an instance of time (duration = 0)
  - this is okay because it's just pseudo-code



# Semaphores



- ➡ A **semaphore**,  $s$ , is a **nonnegative integer** on which there are exactly two operations defined by two garded commands
  - **P ( $s$ ) operation (implemented as a guarded command):**
    - **when** ( $s > 0$ ) [  
           $s = s - 1;$   
        ]
  - **v ( $s$ ) operation (implemented as a guarded command):**
    - [ $s = s + 1;$  ]
  - there are no other means for manipulating the value of  $s$
  - other than initializing it



# Mutexes with Semaphores

```

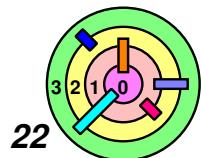
semaphore S = 1;

void OneAtATime( ) {
    P(S);
    ...
    /* code executed mutually
       exclusively */
    ...
    V(S);
}

```

- **P (S) operation:**
  - **when (S > 0) [**
  - S = S - 1;**
  - ]**
  
- **V(S) operation:**
  - **[S = S + 1; ]**

- this is known as a *binary semaphore*



# Implement A Mutex With A Binary Semaphore

→ Instead of doing

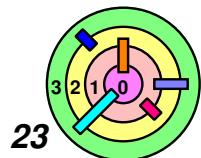
```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock(&m);  
x = x+1;  
pthread_mutex_unlock(&m);
```

— do:

```
semaphore s = 1;  
P(s);  
x = x+1;  
V(s);
```

→ So, you can lock a data structure using a binary semaphore

- this looks just like mutex, what have we really gained?
  - if you use it this way, nothing



# Mutexes with Semaphores

```

semaphore S = N;

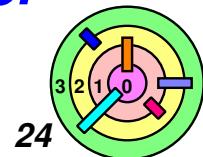
void NATATime( ) {
    P(S);
    ...
    /* no more than N threads
       here at once */
    ...
    V(S);
}

```

- P (S) operation:
  - when  $(S > 0)$  [
  $S = S - 1;$ 
 ]
- V(S) operation:
  - $[S = S + 1;]$

- this is known as a *counting semaphore*
- can be used to solve the producer-consumer problem

- Main difference between a semaphore and a mutex
- if a thread locks a mutex, it's holding the lock
    - therefore, it must be that thread that unlocks that mutex
  - one thread performs a P operation on a semaphore, *another thread* performs a v operation on the same semaphore
    - this is often why you would use a semaphore



# Producer/Consumer with Semaphores

```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

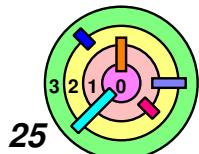
void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return item;
}

```



- Note:** this is not the first procedures of producer and consumer threads
  - this is **synchronization code** (producer and consumer threads calls them to **synchronize** with each other)



# Producer/Consumer with Semaphores

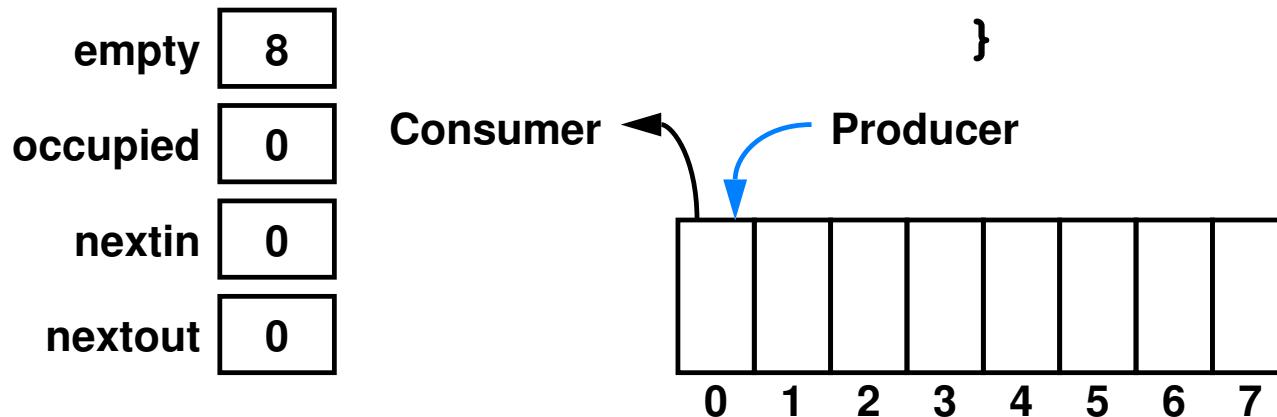
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



# Producer/Consumer with Semaphores

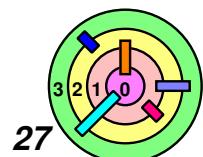
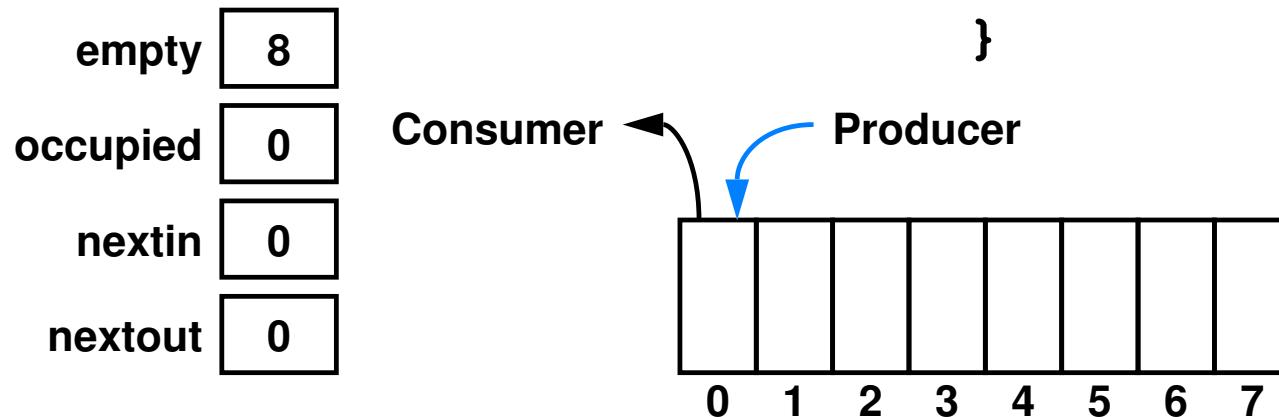
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



# Producer/Consumer with Semaphores

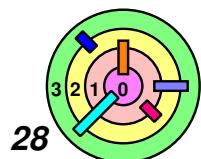
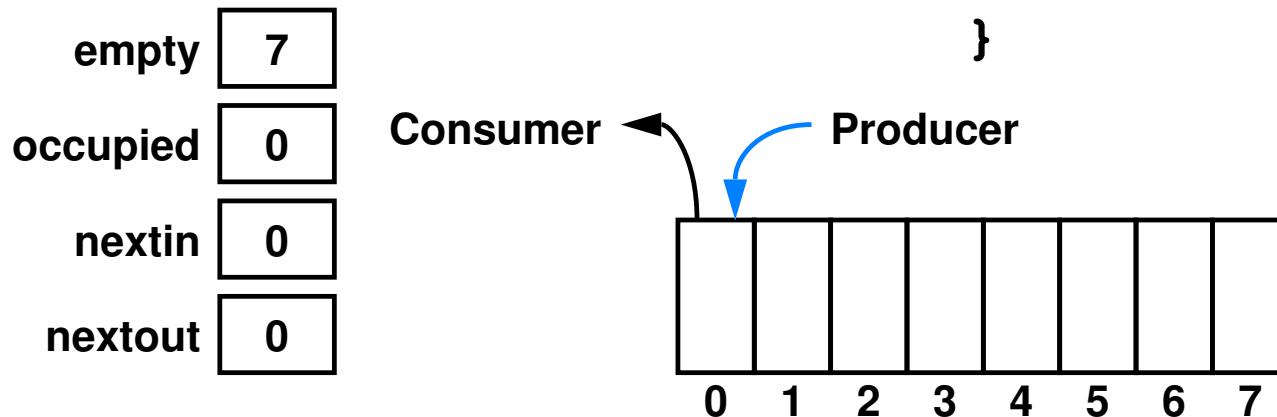
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



# Producer/Consumer with Semaphores

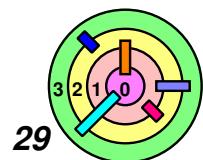
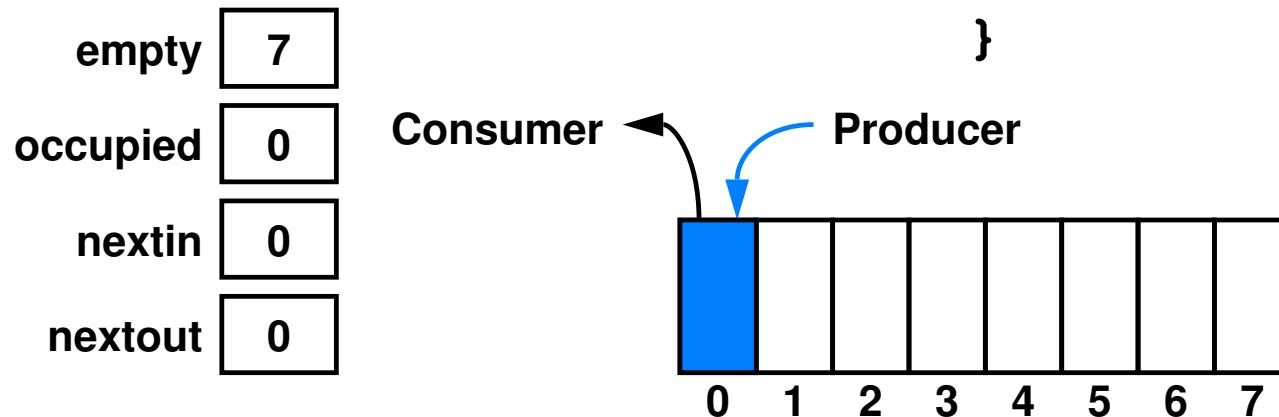
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



# Producer/Consumer with Semaphores

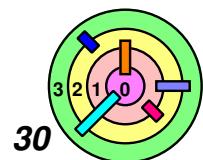
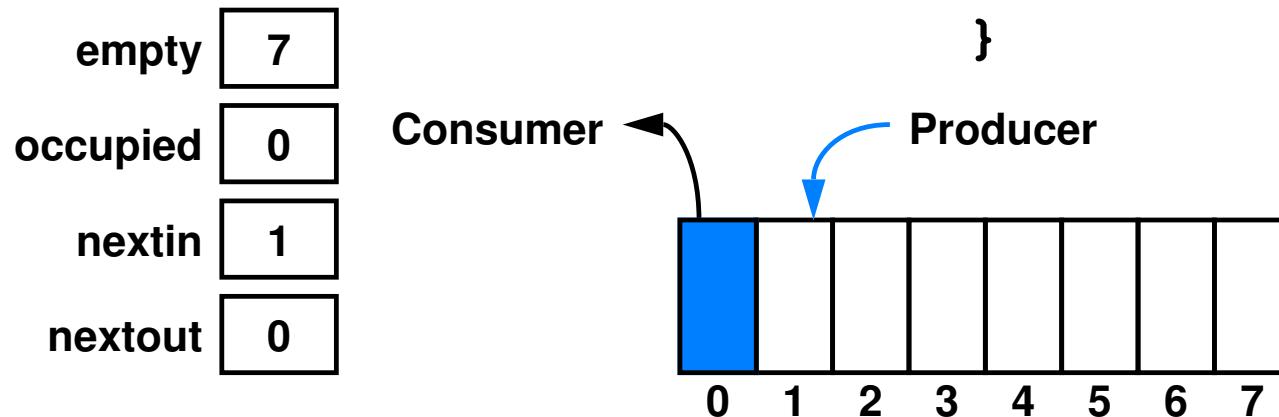
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



# Producer/Consumer with Semaphores

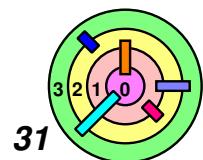
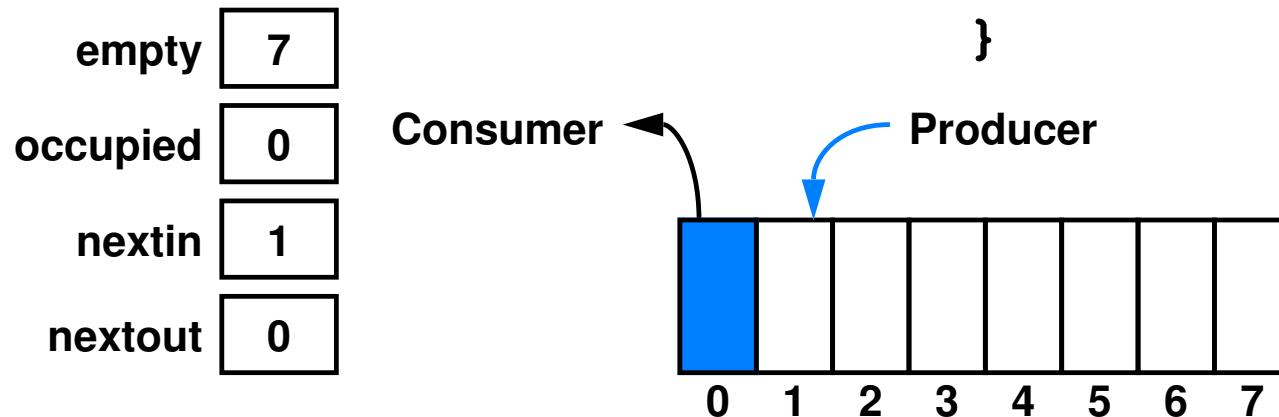
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



# Producer/Consumer with Semaphores

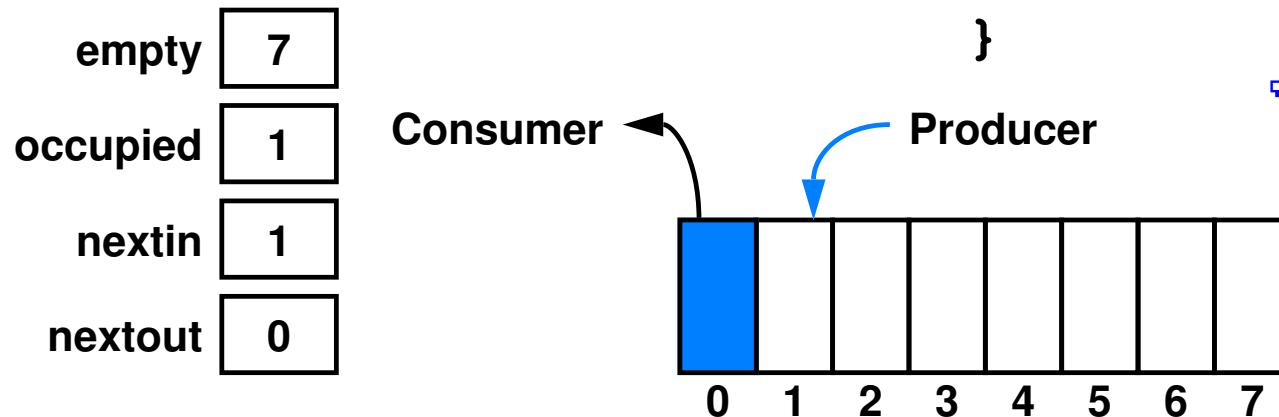
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

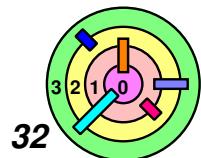
void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



= note: producer  
continue to produce



# Producer/Consumer with Semaphores

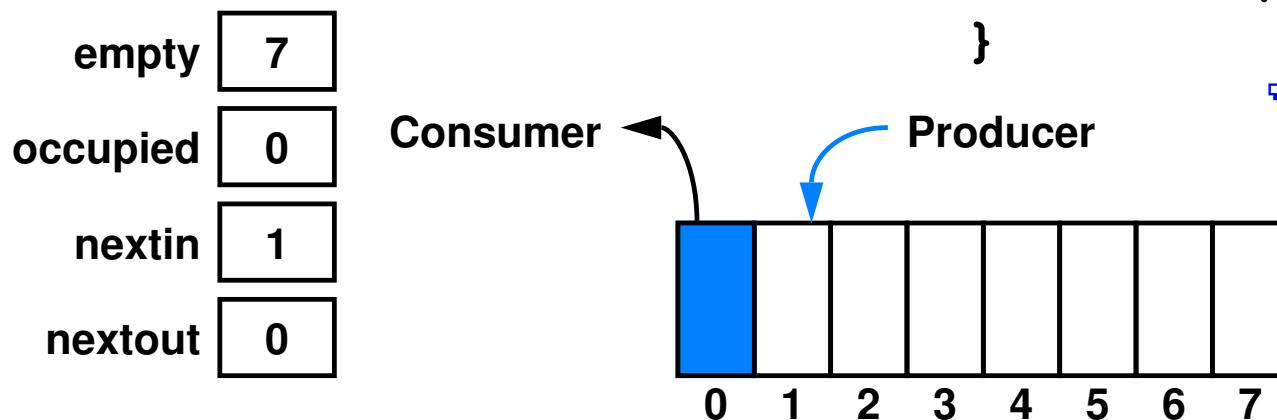
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

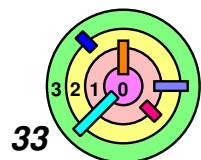
void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



= note: producer  
continue to produce



# Producer/Consumer with Semaphores

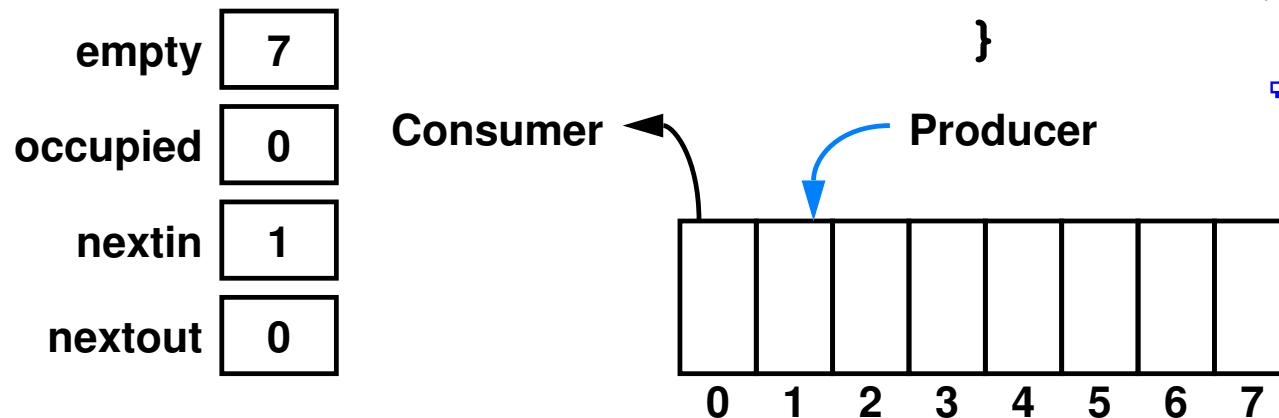
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

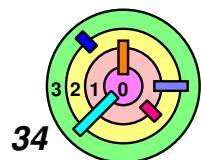
void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



= note: producer  
continue to produce



# Producer/Consumer with Semaphores

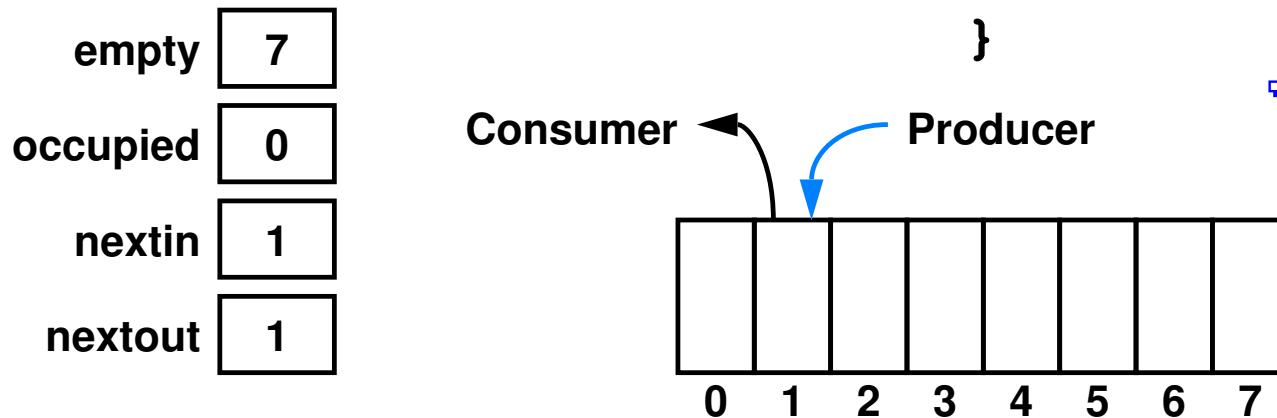
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

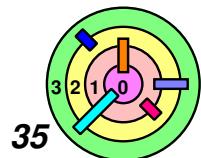
void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



— note: producer  
continue to produce



# Producer/Consumer with Semaphores

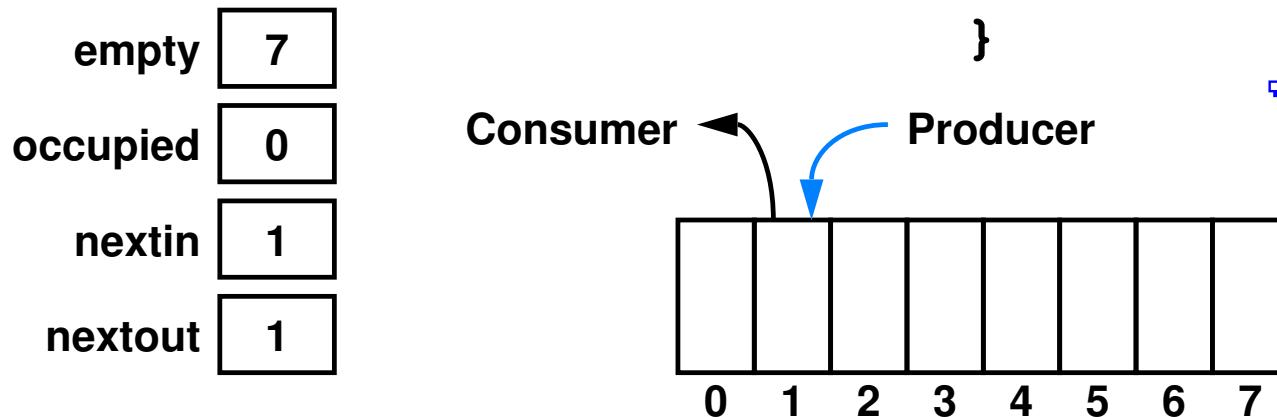
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

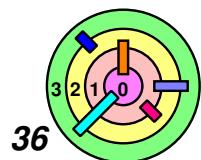
void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return item;
}

```



— note: producer  
continue to produce



# Producer/Consumer with Semaphores

```

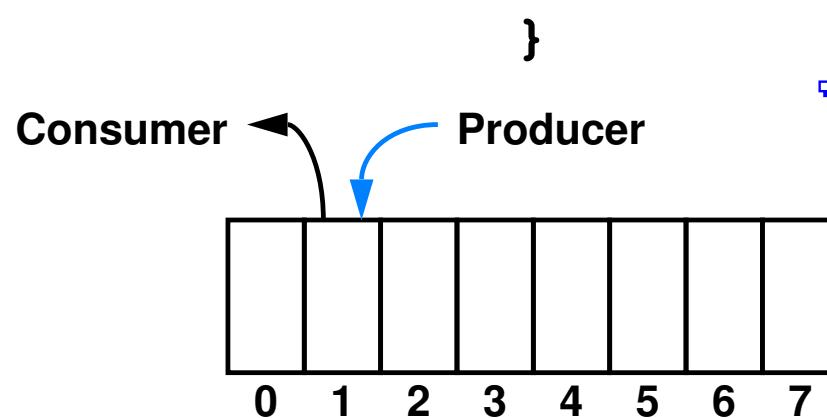
Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

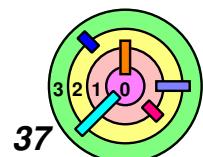
char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return item;
}

```

|          |   |
|----------|---|
| empty    | 8 |
| occupied | 0 |
| nextin   | 1 |
| nextout  | 1 |



— note: producer  
continue to produce



# Producer/Consumer with Semaphores

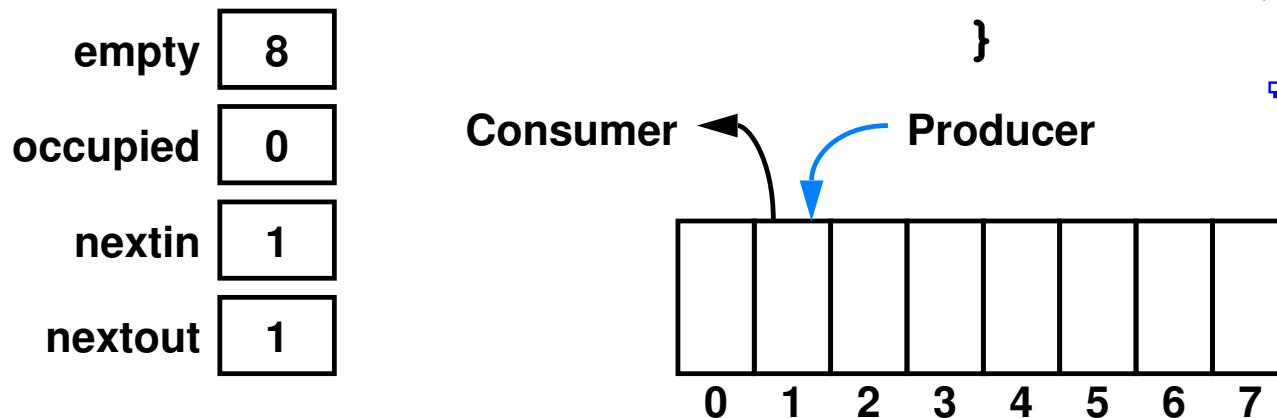
```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

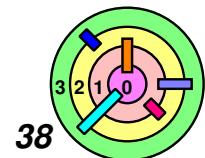
void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```



— note: producer  
continue to produce



# Producer/Consumer with Semaphores

```

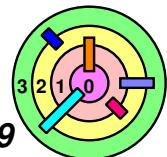
Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}

```

- if produce and consume at same rate, no one may ever wait
  - parallelism of 2
- if producer is fast and consumer slow, producer may wait
- if consumer is fast and producer slow, consumer may wait



# Producer/Consumer with Semaphores

```

Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return item;
}

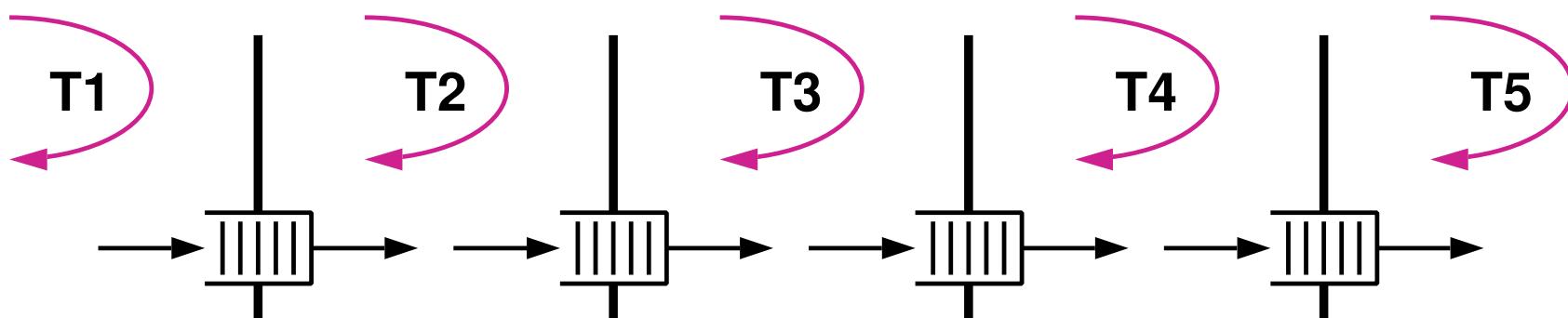
```

- **Mutex** by itself is more "*coarse grain*"
  - you may use one mutex to control access to the number of empty and occupied cells, `nextin`, and `nextout`
- **Semaphore** gives more "*fine grain parallelism*"
  - but *not general* enough

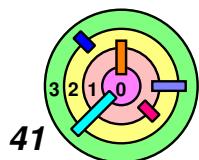
# Semaphore

→ **Semaphore** has limited use

- pretty much the only place it's really good for is producer-consumer
  - although it's a very important application of semaphores



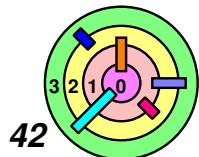
- the queues above are *queues with bounded buffer space*
  - ◊ recall that the "producer-consumer problem" is also known as the "bounded-buffer problem"
- "*pipelined parallelism*"



# POSIX Semaphores

```
#include <semaphore.h>
sem_t semaphore;
int err;
int pshared = 0;      // not shared among processes
int init_value = B;  // initial value

err = sem_init(&semaphore, pshared, init_value);
err = sem_destroy(&semaphore);
err = sem_wait(&semaphore);    /* P operation */
err = sem_trywait(&semaphore); /* conditional P
                                operation
                                */
err = sem_post(&semaphore);   /* V operation */
```



# Producer-Consumer with POSIX Semaphores

```

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume() {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return (item);
}

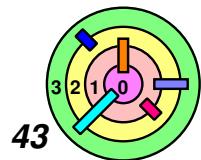
```

```

void Produce(char item) {
    sem_wait(&empty);
    buf[nextin++] = item;
    if (nextin >= B)
        nextin = 0;
    sem_post(&occupied);
}

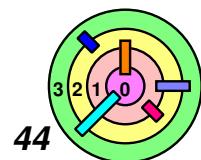
char Consume() {
    char item;
    sem_wait(&occupied);
    item = buf[nextout++];
    if (nextout >= B)
        nextout = 0;
    sem_post(&empty);
    return (item);
}

```



# Implementing Semaphore With Mutex

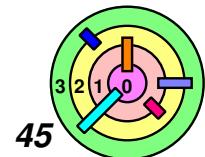
| <i>Semaphore operation</i>                             | <i>POSIX implementation</i>                                                                                                                                                                          |
|--------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> <b>when</b> (S &gt; 0) [     S = S - 1; ] </pre> | <pre> <b>while</b>(1) {     pthread_mutex_lock(&amp;m);     if (S &gt; 0) {         S = S - 1;         pthread_mutex_unlock(&amp;m);         break;     }     pthread_mutex_unlock(&amp;m); } </pre> |
| <pre>[S = S + 1;]</pre>                                | <pre> pthread_mutex_lock(&amp;m); S = S + 1; pthread_mutex_unlock(&amp;m); </pre>                                                                                                                    |



# Implementing Semaphore With Mutex

| <i>Semaphore operation</i>                             | <i>POSIX implementation</i>                                                                                                                                                                                                             |
|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> <b>when</b> (S &gt; 0) [     S = S - 1; ] </pre> | <pre> <b>while</b>(1) {     <b>pthread_mutex_lock</b>(&amp;m);     <b>if</b> (S &gt; 0) {         S = S - 1;         <b>pthread_mutex_unlock</b>(&amp;m);         <b>break</b>;     }     <b>pthread_mutex_unlock</b>(&amp;m); } </pre> |

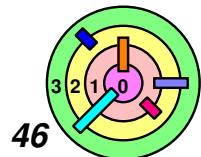
- Implementation of P (S) above works but not good - **inefficient**
- what if guard is false (in this case,  $S = 0$ ) and no other thread is waiting for the mutex?
    - **busy waiting:** your thread will not give up CPU while waiting for the guard to become true
      - ◊ it does not do anything "useful"
  - right way to wait is to **give up the CPU** when waiting



# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

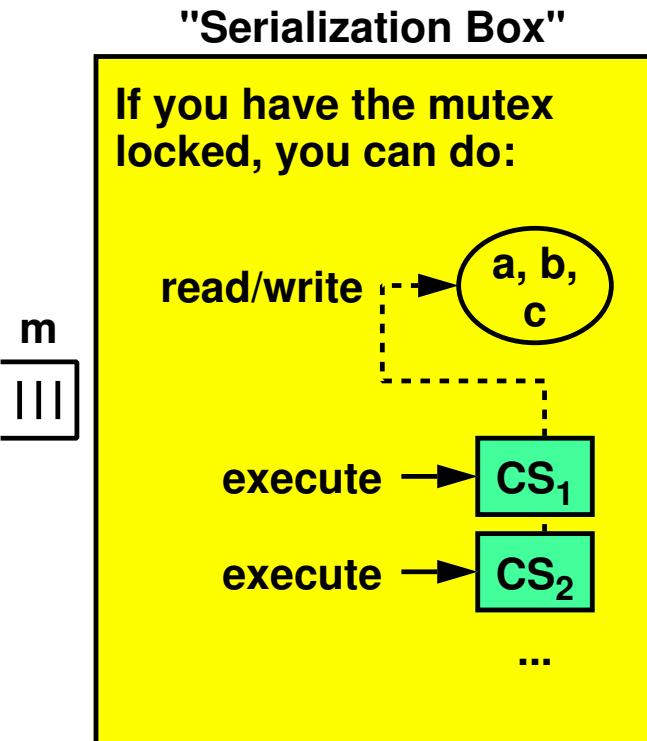
- In general, the *guard* can be *complicated* and involving the evaluation of several variables (e.g.,  $a > 3 \&& f(b) \leq c$ )
- the guard (which evaluates to either true or false) *keeps changing its value, continuously* and by multiple threads in multiple CPUs *simultaneously*
  - how can we "capture" the instance of time when it evaluates to true so we can execute the command sequence atomically?
    - we have to "sample" it, i.e., take snap shot of all the variables that are involved and then evaluate it
    - a mutex is involved, but how?



# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

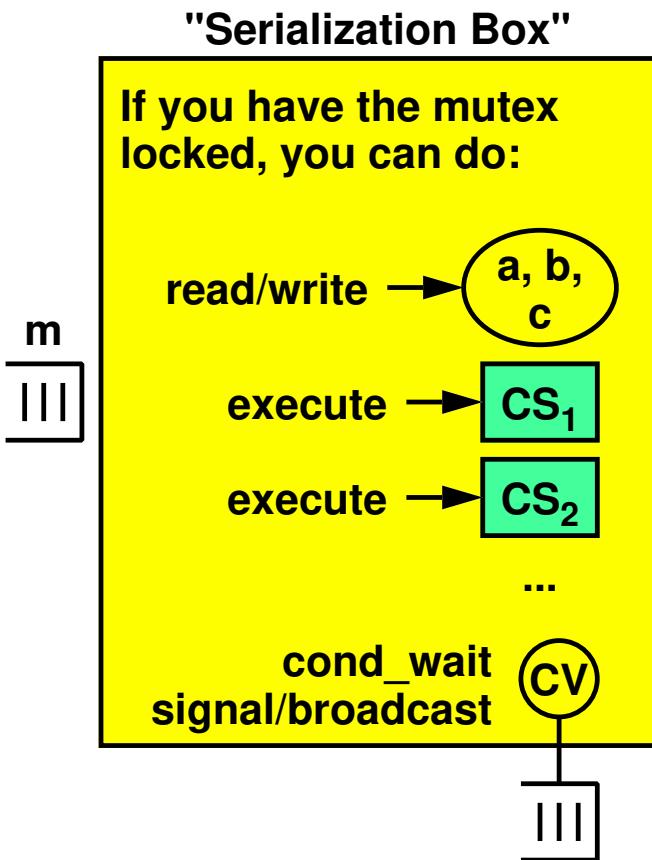
- In general, the *guard* can be *complicated* and involving the evaluation of several variables (e.g.,  $a > 3 \&& f(b) \leq c$ )
  - the guard (which evaluates to either true or false) *keeps changing its value, continuously* and by multiple threads in multiple CPUs *simultaneously*
  - how can we "capture" the instance of time when it evaluates to true so we can execute the command sequence atomically?
    - we have to "sample" it, i.e., take snap shot of all the variables that are involved and then evaluate it
    - a mutex is involved, but how?
    - this would work, but can lead to *busy-waiting*



# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

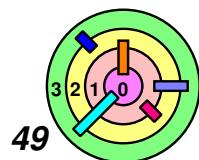
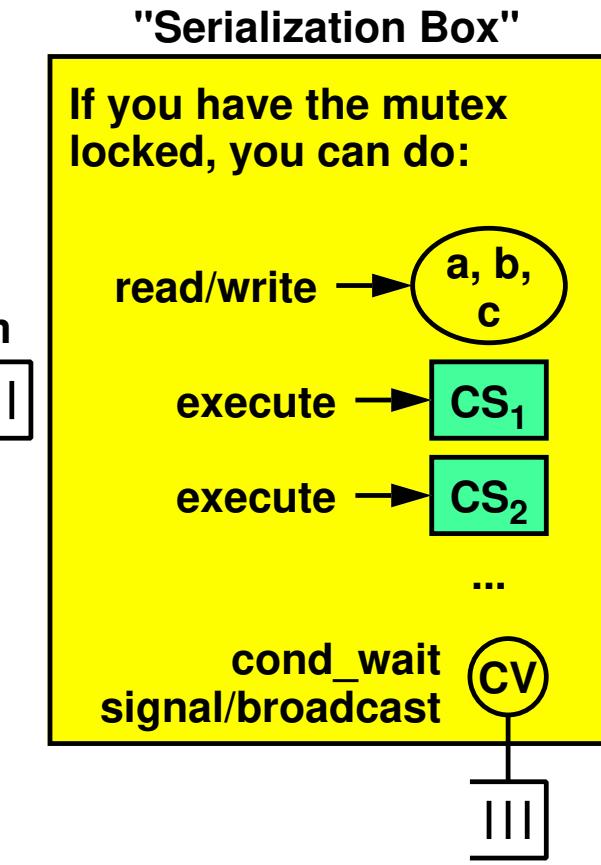
- In general, the *guard* can be *complicated* and involving the evaluation of several variables (e.g.,  $a > 3 \&& f(b) \leq c$ )
  - the guard (which evaluates to either true or false) *keeps changing its value, continuously* and by multiple threads in multiple CPUs *simultaneously*
- Need something else (*known as condition variables*)
  - and a bunch of *rules* to follow



# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

- ▶ POSIX provides **condition variables (CV)** for programmers to implement guarded commands
  - a **condition variable** is a **queue of threads** waiting for some sort of notification (**an "event" or "condition"**)
    - threads, waiting for a guard to become true, go to **sleep in such a queue**
      - ◊ they wait for a specific **condition** to be **signaled**
      - ◊ they wait for **the right time** to **re-evaluate the guard**

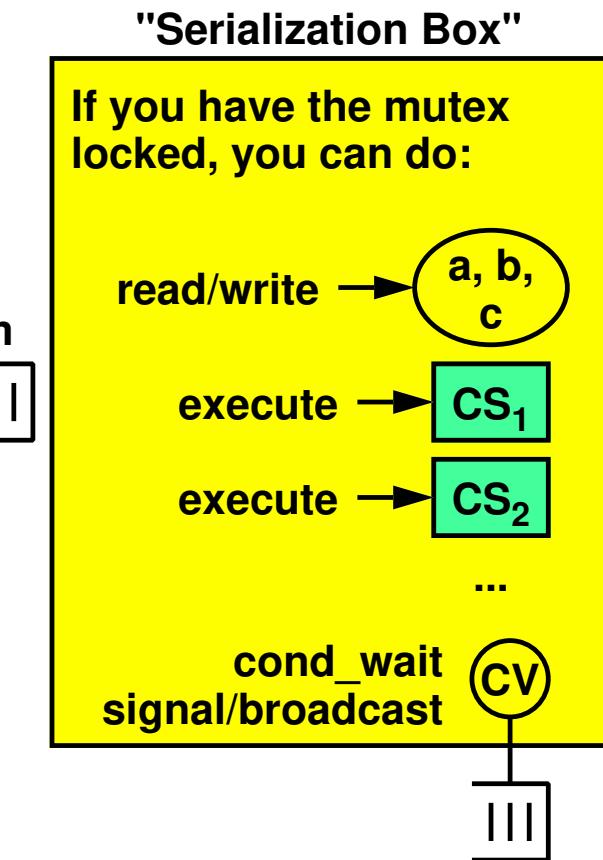


# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

- POSIX provides **condition variables (CV)** for programmers to implement guarded commands

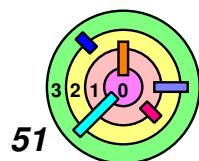
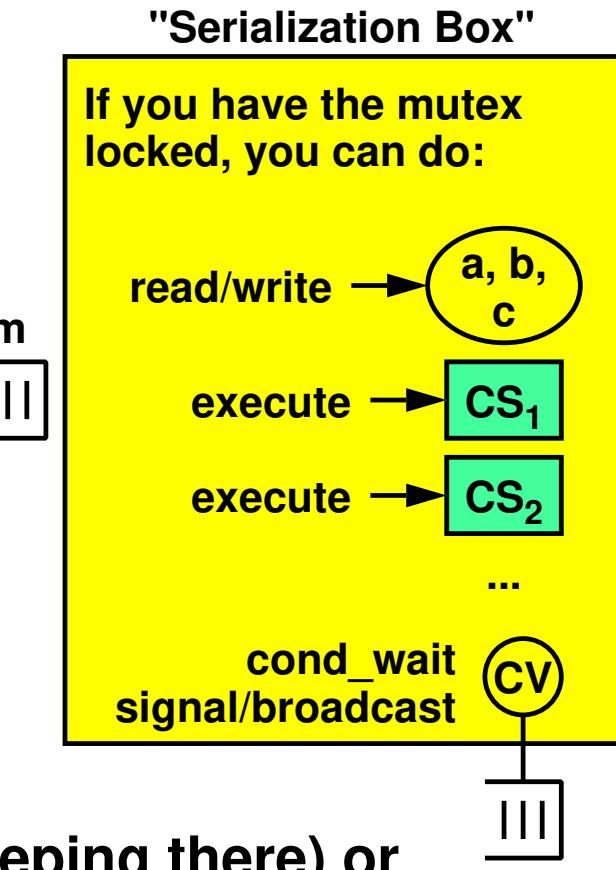
- unlike notifications on your cellphone, an "event" (signaling/broadcasting of a condition) happens in an instance of time (duration of this "event" is zero)
  - if you are not waiting for it, you'll miss it
  - how do you make sure you won't miss an event?
    - ◊ you have to **follow the rules/protocol** (for multiple interacting threads to follow) described here



# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

- POSIX provides **condition variables (CV)** for programmers to implement guarded commands
- threads that do something to **potentially** change the truth value of the guard can then wake up the threads that were sleeping in the queue
    - they can **signal** (wake up **one** thread sleeping there) or **broadcast** (wake up **all** threads sleeping there) the **condition**
      - ◊ **wake up thread(s)** by **moving it into the mutex queue**
    - **no guarantee** that the guard will be true when it's time for **another thread** to evaluate the guard again



# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

- POSIX provides **condition variables (CV)** for programmers to implement guarded commands

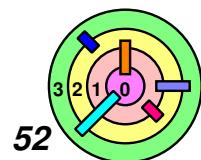
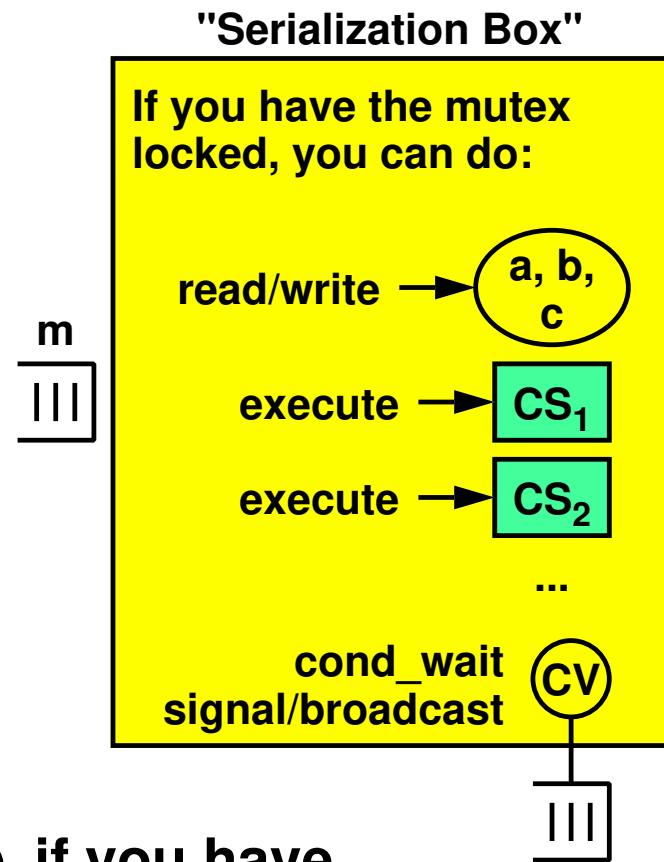
F20-Q8

S21-Q2

su21-den-Q9

1) **pthread\_cond\_wait**(  
 pthread\_cond\_t \*cv,  
 pthread\_mutex\_t \*mutex)

- must only call **pthread\_cond\_wait ()** if you have the **mutex locked**
- **atomically unlocks mutex** and wait for the "event"
- when the event is signaled/broadcasted, **pthread\_cond\_wait () returns with the mutex locked**

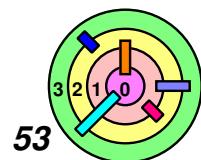
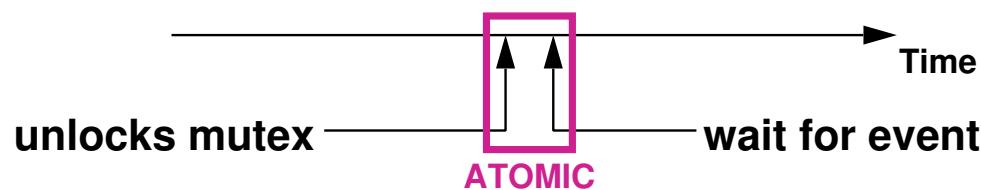
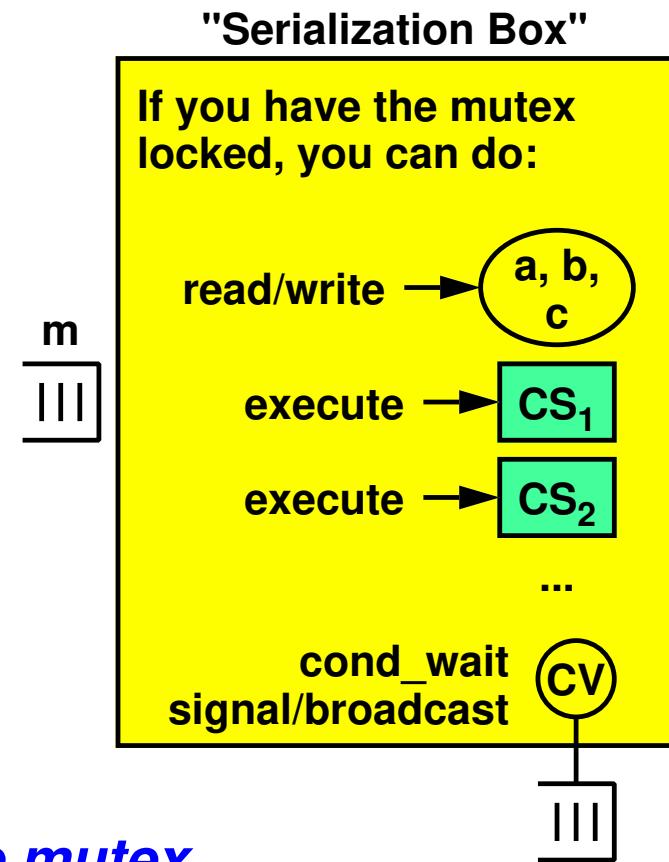


# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

- POSIX provides **condition variables (CV)** for programmers to implement guarded commands

- 1) `pthread_cond_wait(cv, mutex)`
  - **atomically** unlocks `mutex` and wait for the "event"
  - ◊ with respect to the **operation of the mutex**

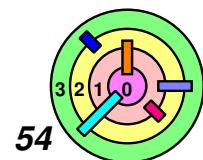
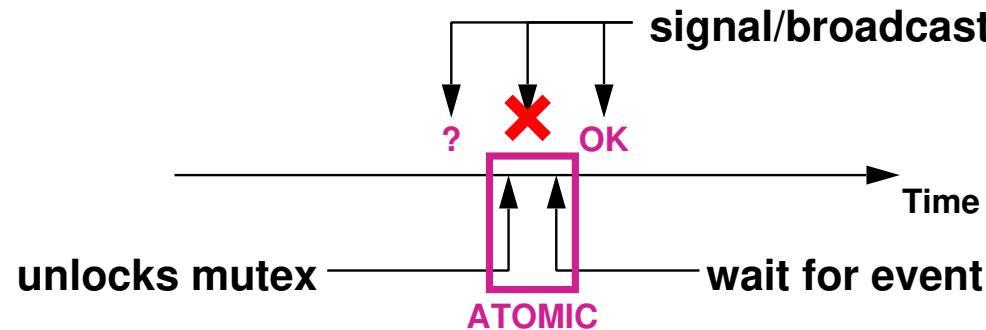
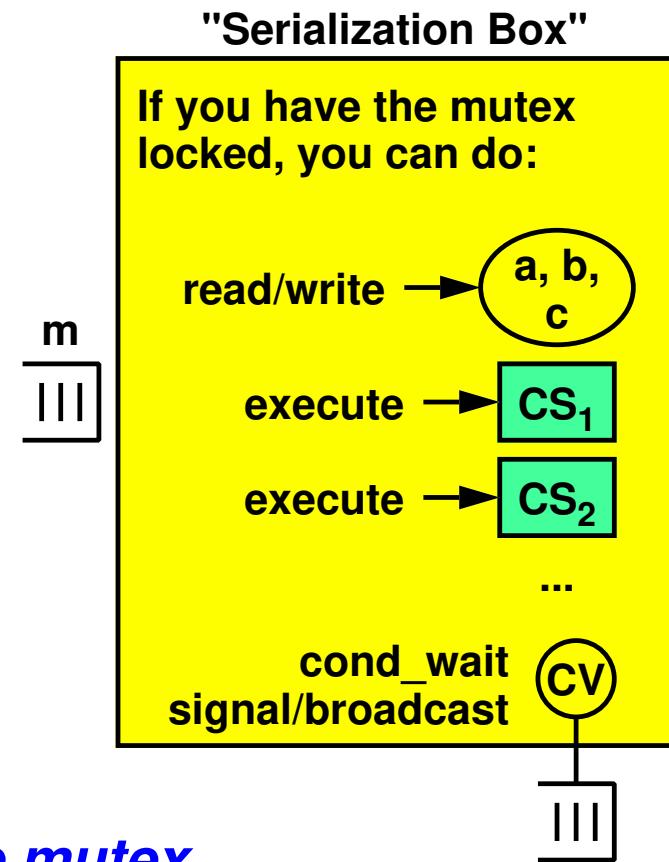


# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

- POSIX provides **condition variables (CV)** for programmers to implement guarded commands

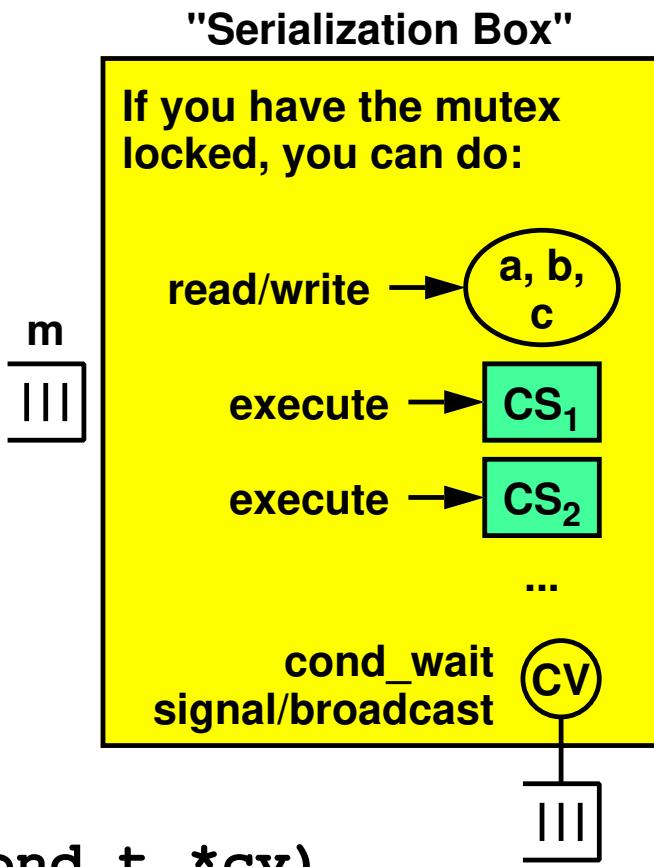
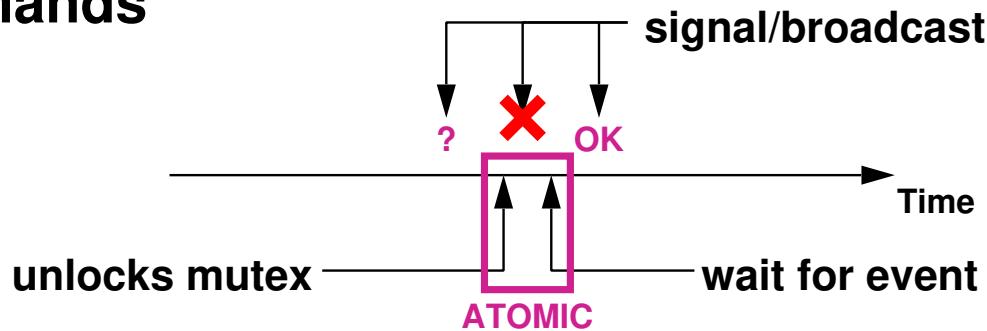
- 1) `pthread_cond_wait(cv, mutex)`
  - **atomically** unlocks `mutex` and wait for the "event"
  - ◊ with respect to the **operation of the mutex**



# Implementation Of General Guarded Commands

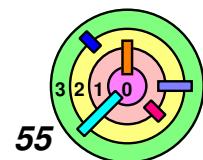
```
when (guard) [
    /* command sequence */
    ...
]
```

- POSIX provides **condition variables (CV)** for programmers to implement guarded commands



- 2) `pthread_cond_broadcast(pthread_cond_t *cv)`  
`pthread_cond_signal(pthread_cond_t *cv)`

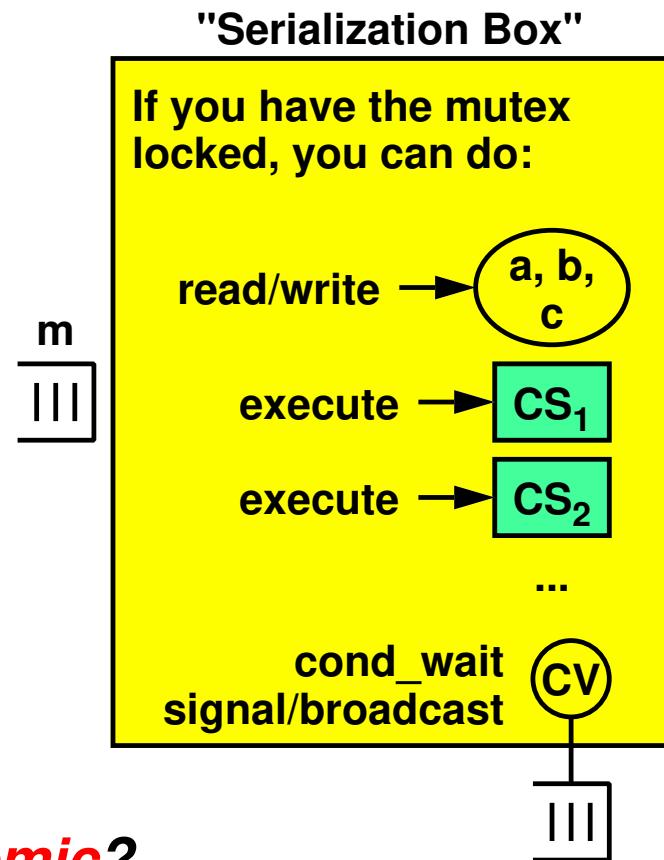
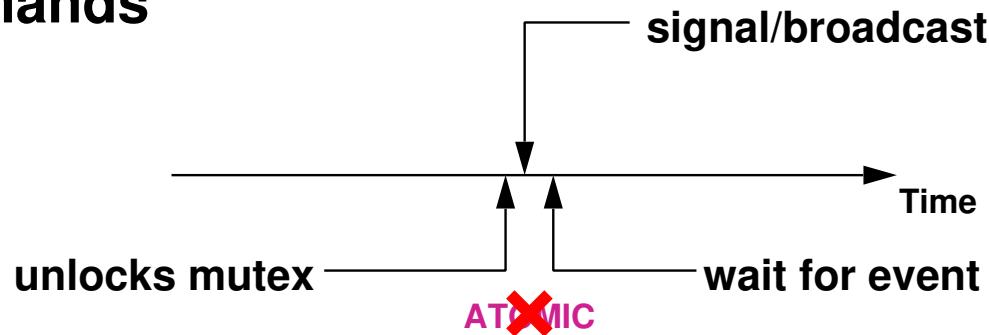
- must only call `pthread_cond_wait()`,  
`pthread_cond_broadcast()` or `pthread_cond_signal()`  
if you have the corresponding mutex **locked**



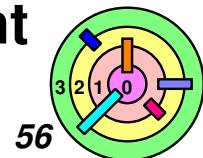
# Implementation Of General Guarded Commands

```
when (guard) [
    /* command sequence */
    ...
]
```

- POSIX provides **condition variables (CV)** for programmers to implement guarded commands



- what if `pthread_cond_wait()` is **not atomic**?
  - your thread may miss the event and sleep forever in the CV queue
  - is this a deadlock?
    - ◊ no, this is a **race condition** (i.e., bad timing-dependent behavior)



# Set Up

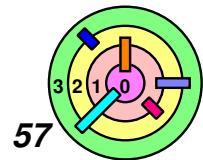
```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
```

→ If a condition variable cannot be initialized statically, do:

```
int pthread_cond_init(
    pthread_cond_t *cvp,
    pthread_condattr_t *attrp)
```

```
int pthread_cond_destroy(
    pthread_cond_t *cvp)
```

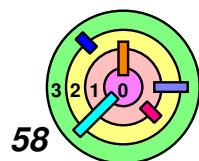
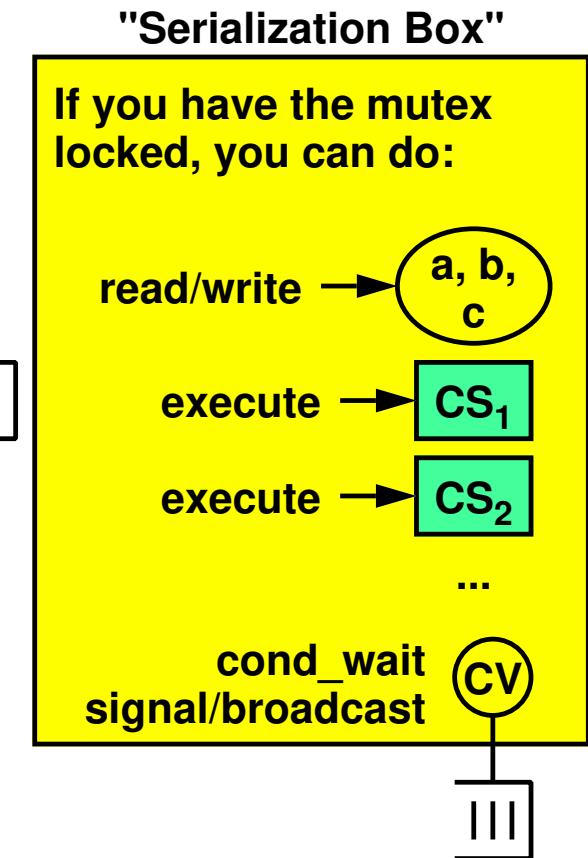
→ Usually, condition variable attributes are not used



# Implementation Of General Guarded Commands

→ **Synchronization:** mutex, condition variables, guards, critical sections

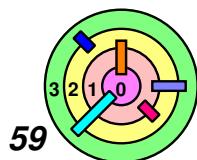
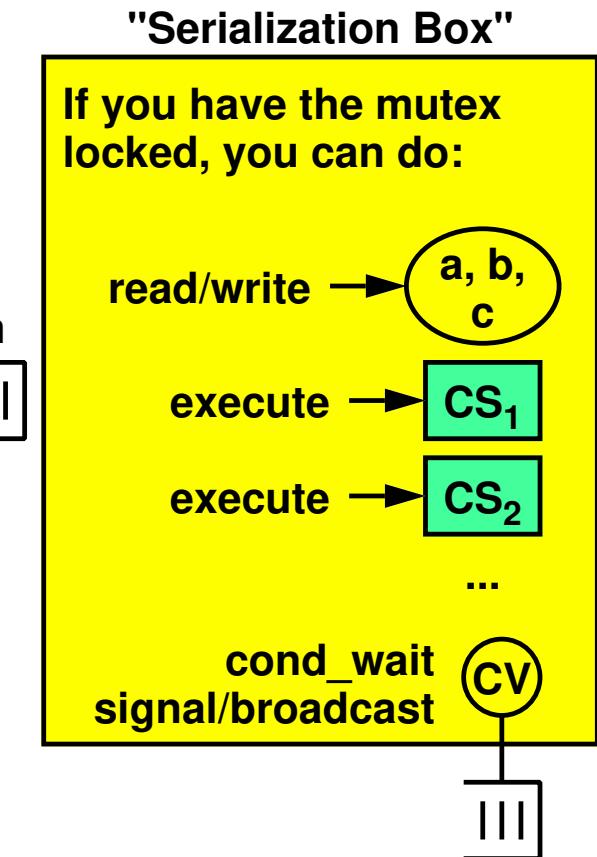
- with respect to a mutex, a thread can be
  - waiting in the mutex queue
  - got the lock and inside the "serialization box"
    - ◆ only one thread can be inside the "serialization box"
  - waiting in the CV queue
  - or outside
- with respect to a mutex, a, b, c are variables that can affect the value of the guard
  - can only access (i.e., read/write) them if a thread is inside the "serialization box" (i.e., has the mutex locked)



# Implementation Of General Guarded Commands

→ **Synchronization:** mutex, condition variables, guards, critical sections

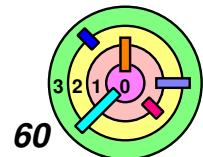
- when you **signal** CV
  - one thread in the CV queue gets moved to the mutex queue
- when you **broadcast** CV
  - all threads in the CV queue get moved to the mutex queue
- you can only get **added** to the CV queue if you have the mutex locked
- you can only **modify** the variables in the guard if you have the mutex locked
- you can only **read** the variables in the guard (i.e., evaluate the guard) if you have the mutex locked
- you can only **execute** critical section code if you have the mutex locked



# POSIX Condition Variables

| <i>Guarded command</i>                                                       | <i>POSIX implementation</i>                                                                                                                                                                   |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre><b>when</b> (guard) [     statement 1;     ...     statement n; ]</pre> | <pre>pthread_mutex_lock(&amp;mutex); <b>while</b> (!guard)     pthread_cond_wait(         &amp;cv,         &amp;mutex); statement 1; ... statement n; pthread_mutex_unlock(&amp;mutex);</pre> |
| <pre>[ /* code  * modifying  * the guard  */ ]</pre>                         | <pre>pthread_mutex_lock(&amp;mutex); /*code modifying the guard*/ ... pthread_cond_broadcast(&amp;cv); pthread_mutex_unlock(&amp;mutex);</pre>                                                |

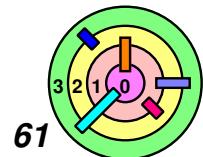
- if you don't follow these rules, your code will have ***race conditions*** (i.e., bad timing-dependent behavior)



# POSIX Condition Variables

| <i>Guarded command</i>                                                       | <i>POSIX implementation</i>                                                                                                                                                                   |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre><b>when</b> (guard) [     statement 1;     ...     statement n; ]</pre> | <pre>pthread_mutex_lock(&amp;mutex); <b>while</b> (!guard)     pthread_cond_wait(         &amp;cv,         &amp;mutex); statement 1; ... statement n; pthread_mutex_unlock(&amp;mutex);</pre> |
| <pre>[ /* code  * modifying  * the guard  */ ]</pre>                         | <pre>pthread_mutex_lock(&amp;mutex); /*code modifying the guard*/ ... pthread_cond_broadcast(&amp;cv); pthread_mutex_unlock(&amp;mutex);</pre>                                                |

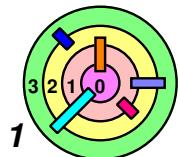
- don't believe that `pthread_cond_signal/broadcast()` can be called without locking the mutex



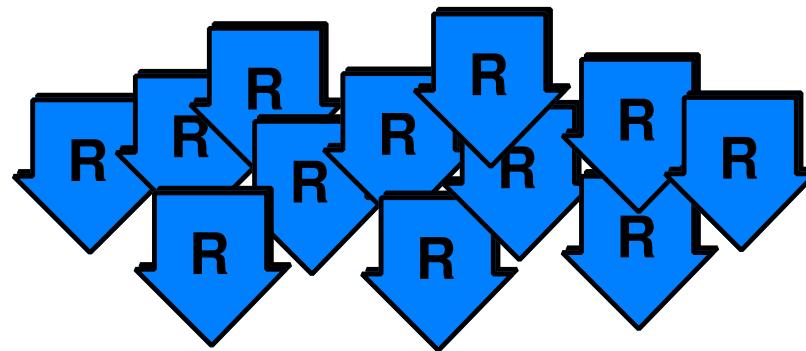
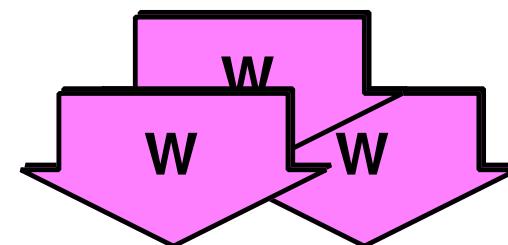
# POSIX Condition Variables

| <i>Guarded command</i>                                                       | <i>POSIX implementation</i>                                                                                                                                                                   |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre><b>when</b> (guard) [     statement 1;     ...     statement n; ]</pre> | <pre>pthread_mutex_lock(&amp;mutex); <b>while</b> (!guard)     pthread_cond_wait(         &amp;cv,         &amp;mutex); statement 1; ... statement n; pthread_mutex_unlock(&amp;mutex);</pre> |
| <pre>[ /* code  * modifying  * the guard  */ ]</pre>                         | <pre>pthread_mutex_lock(&amp;mutex); /*code modifying the guard*/ ... pthread_cond_broadcast(&amp;cv); pthread_mutex_unlock(&amp;mutex);</pre>                                                |

- don't believe that `pthread_cond_signal/broadcast()` can be called without locking the mutex



# Readers-Writers Problem



# Readers-Writers Pseudocode

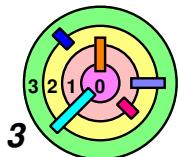
```

reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--;]
}

writer( ) {
    when ((writers == 0) &&
            (readers == 0)) [
        writers++;
    ]
    /* write */
    [writers--;]
}

```

— this is synchronization code



# Pseudocode with Assertions

```

reader( ) {
    when (writers == 0) [
        readers++;
    ]
    // sanity check
    assert((writers == 0) &&
           (readers > 0));
    /* read */
    [readers--];
}

```

```

writer( ) {
    when ((writers == 0) &&
           (readers == 0)) [
        writers++;
    ]
    // sanity check
    assert((readers == 0) &&
           (writers == 1));
    /* write */
    [writers--];
}

```

- the sanity checks are really not necessary

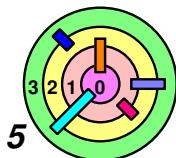


# Readers-Writers Pseudocode

```
reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--;]
}
```

```
writer( ) {
    when ((writers == 0) &&
           (readers == 0)) [
        writers++;
    ]
    /* write */
    [writers--;]
}
```

- since **readers** is part of the guard in the implementation of [**readers--;**], you may need to signal/broadcast the corresponding condition used to implement that guard
  - in this case, only have to signal if **readers** becomes 0 (*if the guard may become true*)

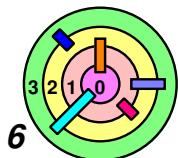


# Readers-Writers Pseudocode

```
reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--;]
}
```

```
writer( ) {
    when !(writers == 0) &&
        (readers == 0)) [
        writers++;
    ]
    /* write */
    [writers--;]
}
```

- also, since `writers` is part of the guards (and these two guards are not identical), in the implementation of `[writers--; ]`, you may need to signal/broadcast the corresponding conditions used to implement these guards
  - in this case, signal/broadcast if `writers` becomes 0 (*if the guard may become true*)

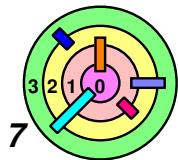


# Readers-Writers Pseudocode

```
reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--;]
}
```

```
writer( ) {
    when ((writers == 0) &&
           (readers == 0)) [
        writers++;
    ]
    /* write */
    [writers--;]
}
```

- don't have to worry about this **readers**
- don't have to worry about this **writers**
  - you need to look at your program logic and figure when signal/broadcast conditions won't be useful
    - ◊ it's *not wrong* to signal/broadcast here, it's just **wasteful/inefficient**



# Readers-Writers Pseudocode

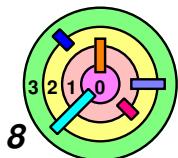
```

reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--];
}

writer( ) {
    when ((writers == 0) &&
            (readers == 0)) [
        writers++;
    ]
    /* write */
    [writers--];
}

```

- **writers** behaves like a binary semaphore
- **readers** behaves like a counting semaphore
- but they are ***not semaphores***
  - due to the definition of a semaphore



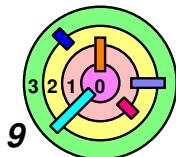
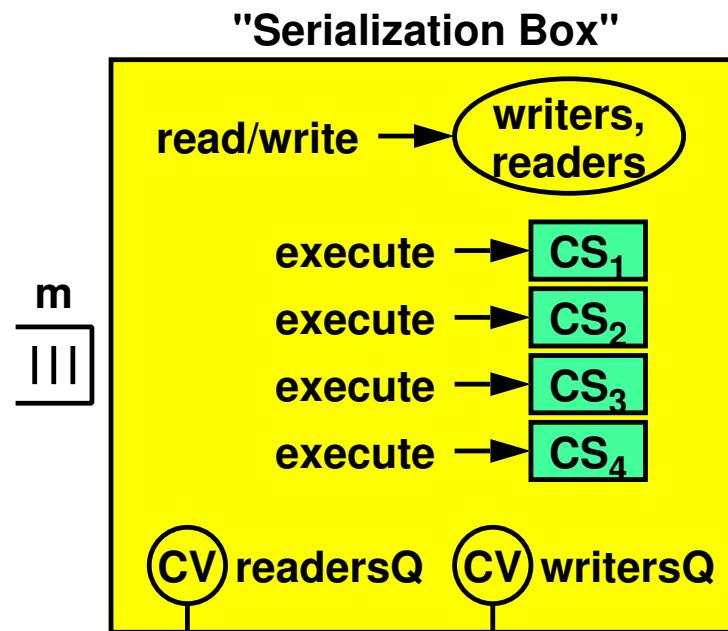
# Solution with POSIX Threads

```

reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--];
}
writer( ) {
    when ((writers == 0) &&
            (readers == 0)) [
        writers++;
    ]
    /* write */
    [writers--];
}

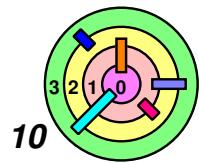
```

- to be even more "efficient", can use multiple CVs
- so you don't have to wake up a thread unnecessarily
  - here we use one CV for reader's guard and one CV for writer's guard (since we want to wake them up separately)



# Recall POSIX Guarded Command Implementation

| <i>Guarded command</i>                                                       | <i>POSIX implementation</i>                                                                                                                                                                   |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre><b>when</b> (guard) [     statement 1;     ...     statement n; ]</pre> | <pre>pthread_mutex_lock(&amp;mutex); <b>while</b> (!guard)     pthread_cond_wait(         &amp;cv,         &amp;mutex); statement 1; ... statement n; pthread_mutex_unlock(&amp;mutex);</pre> |
| <pre>[ /* code  * modifying  * the guard  */ ]</pre>                         | <pre>pthread_mutex_lock(&amp;mutex); /*code modifying the guard*/ ... pthread_cond_broadcast(&amp;cv); pthread_mutex_unlock(&amp;mutex);</pre>                                                |



# Solution with POSIX Threads

```

reader( ) {
    pthread_mutex_lock(&m);
    while (! (writers == 0))
        pthread_cond_wait(
            &readersQ, &m);
    readers++;
    pthread_mutex_unlock(&m);
    /* read */
    pthread_mutex_lock(&m);
    if (--readers == 0)
        pthread_cond_signal(
            &writersQ);
    pthread_mutex_unlock(&m);
}

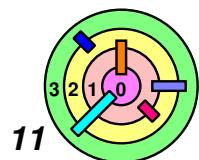
```

```

writer( ) {
    pthread_mutex_lock(&m);
    while (! ((readers == 0) &&
              (writers == 0)))
        pthread_cond_wait(
            &writersQ, &m);
    writers++;
    pthread_mutex_unlock(&m);
    /* write */
    pthread_mutex_lock(&m);
    writers--;
    pthread_cond_signal(
        &writersQ);
    pthread_cond_broadcast(
        &readersQ);
    pthread_mutex_unlock(&m);
}

```

- one mutex (`m`) and two condition variables (`readersQ` and `writersQ`)



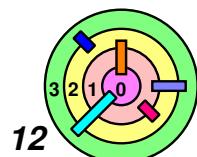
# Solution with POSIX Threads

```
reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--];
}
```



```
reader( ) {
    pthread_mutex_lock(&m);
    while (! (writers == 0))
        pthread_cond_wait(
            &readersQ, &m);
    readers++;
    pthread_mutex_unlock(&m);
    /* read */
    pthread_mutex_lock(&m);
    if (--readers == 0)
        pthread_cond_signal(
            &writersQ);
    pthread_mutex_unlock(&m);
}
```

- one mutex (**m**) and two condition variables (**readersQ** and **writersQ**)



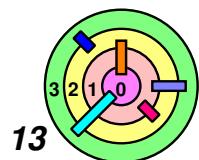
# Solution with POSIX Threads

```
writer( ) {
    when ((writers == 0) &&
          (readers == 0)) [
        writers++;
    ]
    /* write */
    [writers--];
}
```



```
writer( ) {
    pthread_mutex_lock(&m);
    while(!((readers == 0) &&
             (writers == 0)))
        pthread_cond_wait(
            &writersQ, &m);
    writers++;
    pthread_mutex_unlock(&m);
    /* write */
    pthread_mutex_lock(&m);
    writers--;
    pthread_cond_signal(
        &writersQ);
    pthread_cond_broadcast(
        &readersQ);
    pthread_mutex_unlock(&m);
}
```

- one mutex (**m**) and two condition variables (**readersQ** and **writersQ**)



# The Starvation Problem

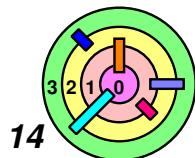
```

reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--];
}

writer( ) {
    when ((writers == 0) &&
            (readers == 0)) [
        writers++;
    ]
    /* write */
    [writers--];
}

```

- Can the writer never get a chance to write?
  - yes, if there are always readers
  - so, this implementation can be unfair to writers
  
- Solution
  - once a writer arrives, shut the door on new readers
    - writers now means the number of writers *wanting* to write
    - use active\_writers to make sure that only one writer can do the actual writing at a time



# Solving The Starvation Problem

```

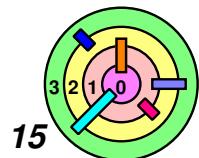
reader( ) {
    when (writers == 0) [
        readers++;
    ]
    /* read */
    [readers--;]
}

writer( ) {
    [writers++]
    when ((readers == 0) &&
           (active_writers == 0))
    [
        active_writers++;
    ]
    /* write */
    [ writers--;
      active_writers--;
    ]
}

```

- now it's unfair to the readers
- isn't writing more important than reading anyway?

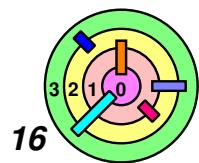
→ This is an example of how to give threads priority *without* assigning priorities to threads!



# Improved Reader

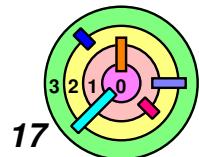
```
reader( ) {  
    pthread_mutex_lock(&m);  
    while (! (writers == 0))  
        pthread_cond_wait(  
            &readersQ, &m);  
    readers++;  
    pthread_mutex_unlock(&m);  
    /* read */  
    pthread_mutex_lock(&m);  
    if (--readers == 0)  
        pthread_cond_signal(  
            &writersQ);  
    pthread_mutex_unlock(&m);  
}
```

- exactly the same as before!



# Improved Writer

```
writer( ) {
    pthread_mutex_lock(&m);
    writers++;
    while (!((readers == 0) &&
              (active_writers == 0))) {
        pthread_cond_wait(&writersQ, &m);
    }
    active_writers++;
    pthread_mutex_unlock(&m);
    /* write */
    pthread_mutex_lock(&m);
    writers--;
    active_writers--;
    if (writers > 0)
        pthread_cond_signal(&writersQ);
    else
        pthread_cond_broadcast(&readersQ);
    pthread_mutex_unlock(&m);
}
```

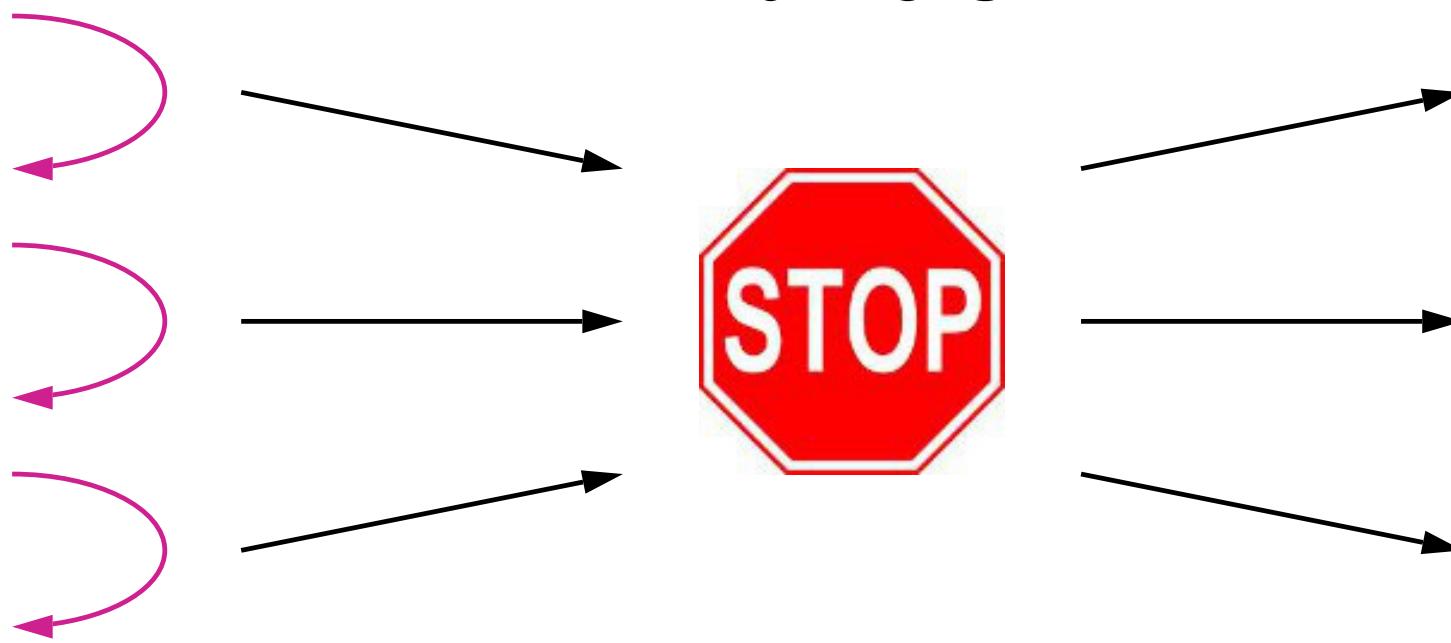


# New, From POSIX!

```
int pthread_rwlock_init(
    pthread_rwlock_t *lock,
    pthread_rwlockattr_t *att);
int pthread_rwlock_destroy(
    pthread_rwlock_t *lock);
int pthread_rwlock_rdlock(
    pthread_rwlock_t *lock);
int pthread_rwlock_wrlock(
    pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(
    pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(
    pthread_rwlock_t *lock);
int pthread_timedrwlock_rdlock(
    pthread_rwlock_t *lock, struct timespec *ts);
int pthread_timedrwlock_wrlock(
    pthread_rwlock_t *lock, struct timespec *ts);
int pthread_rwlock_unlock(
    pthread_rwlock_t *lock);
```



# Barriers



- When a thread reaches a barrier, it must stop (do nothing) and simply wait for other threads to arrive at the same barrier
  - when all the threads that were suppose to arrive at the barrier have all arrived at the barrier, they are all given the signal to proceed forward
    - the barrier is then reset
- Ex: fork/join (fork to create parallel execution)

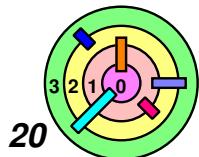
# A Solution?

```

int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
    pthread_mutex_lock(&m);
    if (++count < n) {
        pthread_cond_wait(&BarrierQueue, &m);
    } else {
        count = 0;
        pthread_cond_broadcast(&BarrierQueue);
    }
    pthread_mutex_unlock(&m);
}

```

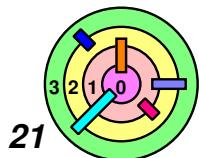
- the idea here is to have the last thread broadcast the condition while all the other threads are blocked at waiting for the condition to be signaled
- as it turns out, `pthread_cond_wait()` might return *spontaneously*, so this won't work
  - [http://pubs.opengroup.org/onlinepubs/009604599/functions/pthread\\_cond\\_signal.html](http://pubs.opengroup.org/onlinepubs/009604599/functions/pthread_cond_signal.html)



# A Solution?

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
    pthread_mutex_lock(&m);
    if (++count < n) {
        while (count < n)
            pthread_cond_wait(&BarrierQueue, &m);
    } else {
        pthread_cond_broadcast(&BarrierQueue);
        count = 0;
    }
    pthread_mutex_unlock(&m);
}
```

- if the  $n^{\text{th}}$  thread wakes up all the other blocked threads, most likely, *none* of these threads will see `count == n`



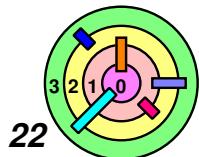
# A Solution?

```

int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
    pthread_mutex_lock(&m);
    if (++count < n) {
        while (count < n)
            pthread_cond_wait(&BarrierQueue, &m);
    } else {
        pthread_cond_broadcast(&BarrierQueue);
    }
    pthread_mutex_unlock(&m);
    count = 0;
}

```

- if the  $n^{\text{th}}$  thread wakes up all the other blocked threads, most likely, *none* of these threads will see `count == n`
- moving `count = 0` around won't help
  - cannot guarantee all  $n$  threads will exit the barrier



su21-a-Q18

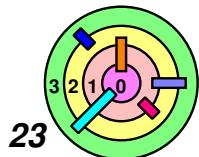
# Barrier in POSIX Threads

```

int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
    pthread_mutex_lock(&m);
    if (++count < number) {
        int my_generation = generation;
        while (my_generation == generation)
            pthread_cond_wait(&BarrierQueue, &m);
    } else {
        count = 0;
        generation++;
        pthread_cond_broadcast(&BarrierQueue);
    }
    pthread_mutex_unlock(&m);
}

```

- don't use count in the guard since its problematic!
- introduce a new guard (with a new variable)



# More From POSIX!

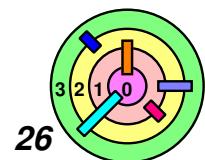
```
int pthread_barrier_init(
    pthread_barrier_t *barrier,
    pthread_barrierattr_t *attr,
    unsigned int count);
int pthread_barrier_destroy(
    pthread_barrier_t *barrier);
int pthread_barrier_wait(
    pthread_barrier_t *barrier);
```



## 2.2.4 Thread Safety

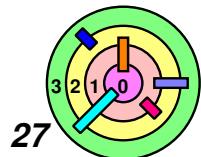
# Thread Safety

- ➔ Unix was developed way before threads were commonly used
  - Unix libraries were built without threads in mind
  - running code using these library functions with threads became unsafe
  - to make these library functions safe to run under *multithreading* is known as *Thread Safety*
- ➔ General problems with the old Unix API
  - global variables, e.g., `errno`
  - shared data, e.g., `printf()`



# Thread Safety vs. Reentrancy

- ➡ Strictly speaking, making a function ***thread-safe*** is not the same as making it ***reentrant***
  - ***thread-safe***: multiple threads can call the function in parallel or concurrently
  - ***reentrant***: enter twice (even if you have only one thread)
    - how do you enter a function twice if you are not running multiple threads?
      - ◊ for a ***kernel function***, you can get interrupted and call the same function inside an interrupt service routine
      - ◊ for a ***user space function***, you can get interrupted and the kernel makes an ***upcall*** that calls the same function
  - most of the time, making a function thread-safe and making it reentrant ends up to be the same thing
    - but you need to be careful
    - you can google "reentrancy" to see the difference between reentrant code and thread-safe code and see examples
  - we focus on ***multi-threading*** (and "no signal handles")



# Global Variables

```

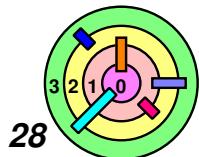
int IOfunc(int fd) {
    extern int errno;
    ...
    if (write(fd, buffer, size) == -1) {
        if (errno == EIO)
            fprintf(stderr, "IO problems ...\\n");
        ...
    }
    return(0);
}
...
}

```

su21-a-Q13

F20-Q16

- if 2 threads call this function and both failed, how do you guarantee that a thread would get the right **errno**?
  - the code is *not "thread safe"*
- **errno** is a system-call level *global variable*
  - Unix system-call library was implemented before multi-threading was a common practice



# Coping

→ Fix Unix's C/system-call interface

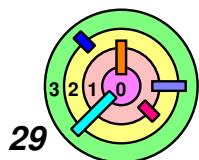
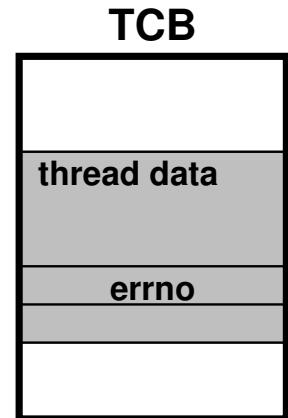
- want backwards compatibility

→ Make `errno` refer to a different location in each thread

- e.g.,

```
#define errno __errno(thread_ID)
```

- `__errno(thread_ID)` will return the *thread-specific* `errno`
  - need a place to store this thread-specific `errno`
  - POSIX threads provides a general mechanism to store *thread-specific data*
    - ◊ Win32 has something similar called thread-local storage
    - POSIX does not specify how this private storage is allocated and organized
      - ◊ done with an array of `(void*)`
      - ◊ then `errno` would be at a fixed index into this array
  - don't need to change application, just recompile



# Add "Reentrant" Version Of System Call

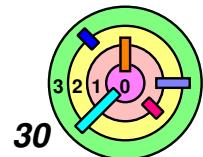
→ **gethostbyname () system call is not "thread safe"**

```
struct hostent *gethostbyname(const char *name)
```

- it returns a pointer to a global variable
  - (what a terrible idea!)
- POSIX's fix for this problem is to add a function to the system library

```
int gethostbyname_r(const char *name,
                     struct hostent *ret,
                     char *buf,
                     size_t buflen,
                     struct hostent **result,
                     int *h_errnop)
```

- caller of this function must provide the buffer to hold the return data
  - ◊ (a good idea in general)
- caller is aware of thread-safety
  - ◊ (a more educated programmer is desirable)



# Shared Data

→ Thread 1:

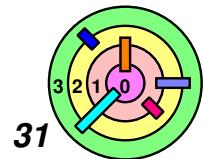
```
printf("goto statement reached");
```

→ Thread 2:

```
printf("Hello World\n");
```

→ Printed on display:

```
goto Hello Wostatement reachedrld
```

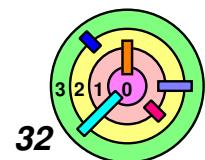


# Coping

- Wrap library calls with synchronization constructs
- Fix the libraries
- Application can use a mutex
- If application is using the `(FILE*)` object in `<stdio.h>`, can wrap functions like `printf()` around these functions

```
void flockfile(FILE *filehandle)
int ftrylockfile(FILE *filehandle)
void funlockfile(FILE *filehandle)
```

- basically, `flockfile()` would block until lockcount is 0
  - then it increments the lockcount
- `funlockfile()` decrements the lockcount



# Killing Time ...

- To suspend your thread for a certain duration

```
struct timespec timeout, remaining_time;

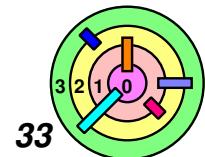
timeout.tv_sec = 3;           // seconds
timeout.tv_nsec = 1000; // nanoseconds
nanosleep(&timeout, &remaining_time);
```

- Unix/Linux is "best-effort"
- okay to do this in `warmup2` since it only has to run on Ubuntu

- The right way to sleep is to say when you want to wake up, e.g.,

```
int pthread_cond_timedwait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    struct timespec *abstime)
```

- after `abstime`, give up waiting for an event and return with an error code
- you need to calculate `abstime` carefully and correctly



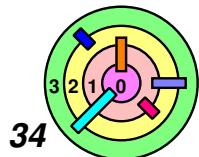
# Timeouts

```

struct timespec relative_timeout, absolute_timeout;
struct timeval now;
relative_timeout.tv_sec = 3;           // seconds
relative_timeout.tv_nsec = 1000;        // nanoseconds
gettimeofday(&now, 0);
absolute_timeout.tv_sec = now.tv_sec +
    relative_timeout.tv_sec;
absolute_timeout.tv_nsec = 1000*now.tv_usec +
    relative_timeout.tv_nsec;
if (absolute_timeout.tv_nsec >= 1000000000) {
    // deal with the carry
    absolute_timeout.tv_nsec -= 1000000000;
    absolute_timeout.tv_sec++;
}
pthread_mutex_lock(&m);
while (!may_continue)
    pthread_cond_timedwait(&cv, &m, &absolute_timeout);
pthread_mutex_unlock(&m);

```

- must check return code of `pthread_cond_timedwait()`



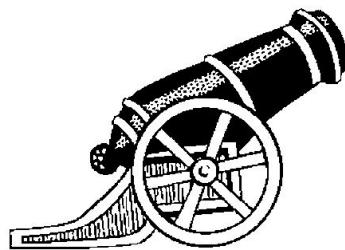
## 2.2.5 Deviations

# Deviations

- ➔ How do you ask another thread to deviate from its normal execution path?
  - Unix's *signal* mechanism
- ➔ How do you force another thread to terminate cleanly
  - POSIX *cancellation* mechanism

# Signals

```
int x, y;  
  
x = 0;  
...  
y = 16/x;
```



```
for (;;) keep_on_trying( );
```



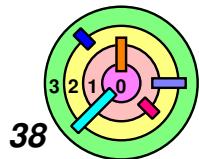
- the original intent of Unix signals was to force the *graceful termination* of a process
  - e.g., <Ctrl+C>

# The OS to the Rescue



## Signals

- some would call a *signal* a *software interrupt*
  - but it's really not
    - ◊ it's a "*callback mechanism*"
    - ◊ implemented in the OS by performing an *upcall*
- generated (by OS) in response to
  - exceptions (e.g., arithmetic errors, addressing problems)
  - external events (e.g., timer expiration, certain keystrokes, actions of other processes such as to terminate or pause the process)
  - user defined events
- effect on process (*i.e., when the signal is "delivered"*):
  - termination (possibly after producing a core dump)
  - invocation of a procedure that has been set up to be a signal handler (*requires an upcall*)
  - suspension of execution
  - resumption of execution



# Terminology

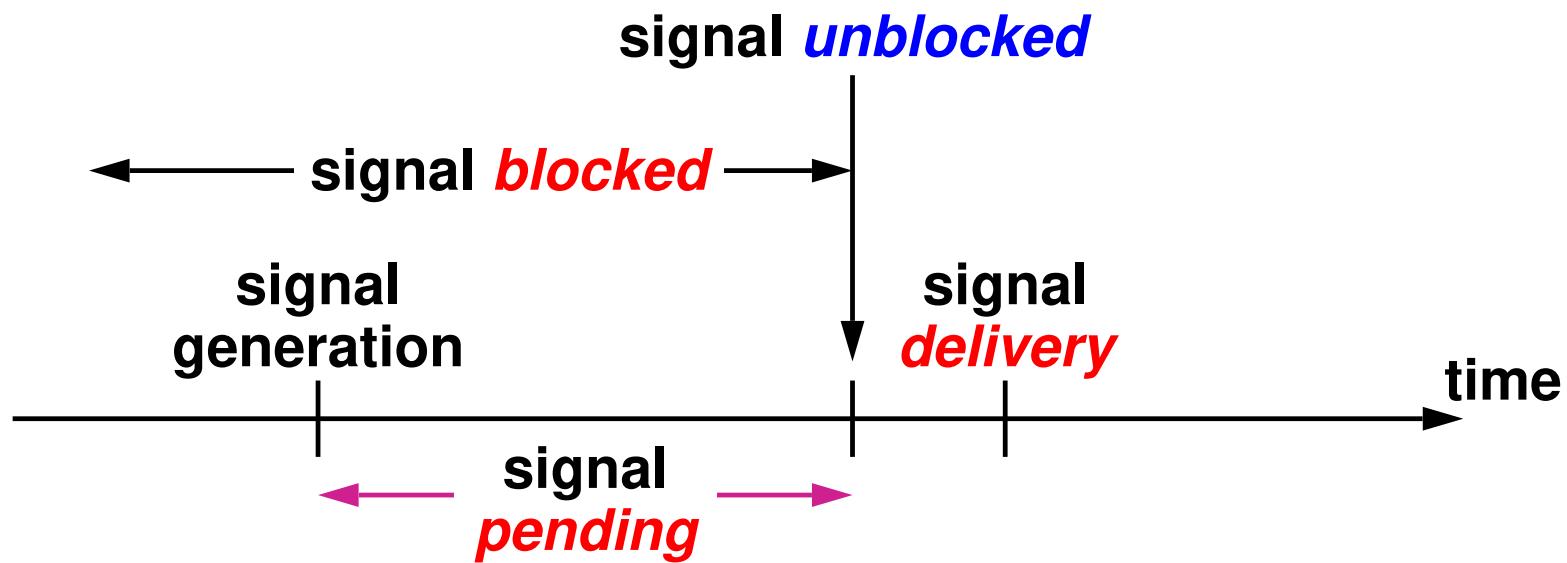
← signal not blocked →



→ Ex: <Ctrl+C>

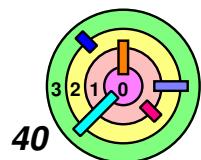
→ When a signal is generated, it is delivered as soon as possible **if** the signal is not "**blocked**"

# Terminology



- ➔ Ex: <Ctrl+C>
- ➔ A signal is *pending* if it's generated but *blocked*
  - when the signal becomes unblocked, it will be *delivered*
- ➔ If you replaced the word "signal" with "interrupt" and "blocked/unblocked" with "disabled/enabled", everything would be correct for a hardware interrupt

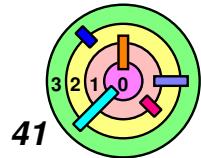
s21-den-Q1



40

# Signal Types

| Name    | Description                       | Default Action |
|---------|-----------------------------------|----------------|
| SIGABRT | abort called                      | term, core     |
| SIGALRM | alarm clock                       | term           |
| SIGCHLD | death of a child                  | ignore         |
| SIGCONT | continue after stop               | cont           |
| SIGFPE  | erroneous arithmetic operation    | term, core     |
| SIGHUP  | hangup on controlling terminal    | term           |
| SIGILL  | illegal instruction               | term, core     |
| SIGINT  | interrupt from keyboard           | term           |
| SIGKILL | kill                              | forced term    |
| SIGPIPE | write on pipe with no one to read | term           |
| SIGQUIT | quit                              | term, core     |
| SIGSEGV | invalid memory reference          | term, core     |
| SIGSTOP | stop process                      | forced stop    |
| SIGTERM | software termination signal       | term           |
| SIGTSTP | stop signal from keyboard         | stop           |
| SIGTTIN | background read attempted         | stop           |
| SIGTTOU | background write attempted        | stop           |
| SIGUSR1 | application-defined signal 1      | stop           |
| SIGUSR2 | application-defined signal 2      | stop           |



# Sending a Signal

→ `int kill(pid_t pid, int sig)`

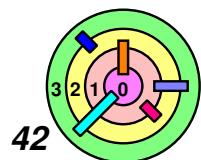
- send signal `sig` to process `pid`
- (not always) terminate with extreme prejudice

→ Also

- type Ctrl-c (or <Ctrl+C>)
  - sends signal 2 (SIGINT) to current process
- `kill` shell command
  - send SIGINT to process with pid=12345: "kill -2 12345"
- do something illegal
  - bad address, bad arithmetic, etc.

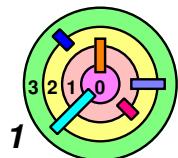
→ `int pthread_kill(pthread_t thr, int sig)`

- send signal `sig` to thread `thr` (in the same process as the calling thread)
- avoid using this and use *pthread cancellation* mechanism instead if you want to "kill a thread"



# Handling Signals

- Two ways to handle signals
  - handling it *asynchronously*
    - using *signal handlers*
  - handling it *synchronously*
    - using `sigwait()` in a signal-catching thread



# Handling Signals Asynchronously



## Signal handler

- each signal in a *process* can have *at most one handler*
- to specify a signal handler of a process, use:
  - `sigset/signal()`
    - ◊ returns the current handler (which could be the "default handler")
  - `sigaction()`
    - ◊ more functionality

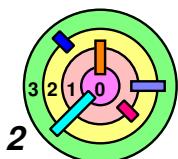
```
#include <signal.h>
```

```
typedef void (*sighandler_t) (int);
```

```
sighandler_t sigset(int signo, sighandler_t handler);
sighandler_t signal(int signo, sighandler_t handler);
```

```
sighandler_t OldHandler = sigset(SIGINT, NewHandler);
```

- signal handler is part of the *context* of a process



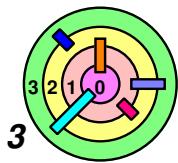
# Special Handlers

## → SIG\_DFL

- use the default handler
- usually terminates the process
- `sigset/signal(SIGINT, SIG_DFL);`

## → SIG\_IGN

- ignore the signal
- `sigset/signal(SIGINT, SIG_IGN);`



# Example

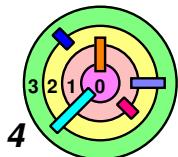
```
#include <signal.h>

int main() {
    void handler(int);

    sigset(SIGINT, handler);
    while(1)
        ;
    return 1;
}

void handler(int signo) {
    printf("I received signal %d. Whoopee!!\n", signo);
}
```

- **SIGINT is blocked** inside `handler()`
- but how do you kill this program from your console?
  - can use the "kill" shell command, e.g., "kill -15 <pid>"
- instead of using `sigset()`, you can also use `sigaction()`



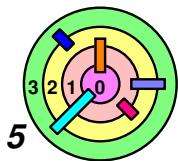
# Example

```
#include <signal.h>

int main() {
    void handler(int);

    sigset(SIGINT, handler);
    while(1)
        ;
    return 1;
}

void handler(int signo) {
    printf("I received signal %d. Whoopee!!\n", signo);
    sigset(SIGINT, handler); ← in some systems, you may have to
}   re-establish the signal handler inside
  the signal handler if you want to receive
  the same signal more than once
```



# sigaction

```

int sigaction(int sig,
const struct sigaction *new,
struct sigaction *old);

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
};

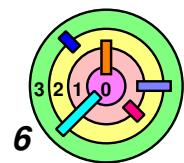
```

- sigaction() allows for more complex behavior
- e.g., block *additional* signals (specified by `sa_mask`) when handler is called

```

int main() {
    struct sigaction act;
    void sighandler(int);
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    act.sa_handler = sighandler;
    sigaction(SIGINT, &act, NULL);
    ...
}

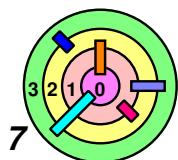
```



S21-Q15 su21-a-Q2

# Async-Signal Safety

- The problem with *asynchronous signal* is that you have to worry about *async-signal safety*
  - if you don't take care of it just right, bad things can happen
- **Async-Signal Safety:** Make your code safe when working with *asynchronous signals*
- The general rule to provide async-signal safety:
  - any data structure the signal handler accesses must be async-signal safe
    - i.e., an async signal must not corrupt data structures
- An alternative is to make async-signal synchronous
  - use a *signal-catching thread* to receive a particular signal



su21-a-Q5

# Example 1: Waiting for a Signal

```

sigset(SIGALRM, DoSomethingInteresting);

...
struct timeval waitperiod = {0, 1000};
    /* seconds, microseconds */
struct timeval interval = {0, 0};
struct itimerval timerval;

timerval.it_value = waitperiod;
timerval.it_interval = interval;

setitimer(ITIMER_REAL, &timerval, 0);
    /* SIGALRM sent in ~one millisecond */

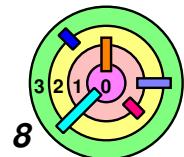
pause(); /* wait for it */

```

- can SIGALRM occur before pause() is called?



- Note:** strictly speaking, this is *not* a deadlock
- it has a *race condition*



## Example 2: Status Update

```
#include <signal.h>
computation_state_t state;

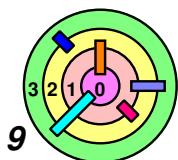
int main() {
    void handler(int);
    sigset(SIGINT, handler);
    long_running_proc();
    return 0;
}

void long_running_proc() {
    while (a_long_time) {
        update_state(&state);
        compute_more();
    }
}

void handler(int signo) {
    display(&state);
}
```

- long-running job that can take days to complete
  - the `handler()` can be used to print a progress report
  - need to make sure that `state` is in a consistent state
  - this is a synchronization issue
  - our `handler()` is not *async-signal safe*

- Note: this is *not* a deadlock and really not a race condition
- this is the case where the code is not async signal safe



## Example 2: Status Update

```

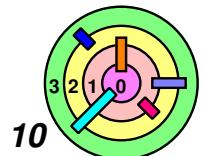
void long_running_proc() {
    while (a_long_time) {
        pthread_mutex_lock(&m);
        update_state(&state);
        pthread_mutex_unlock(&m);
        compute_more();
    }
}

void handler(int signo) {
    pthread_mutex_lock(&m);
    display(&state);
    pthread_mutex_unlock(&m);
}

```

- Does this work?
- no (*this code is not reentrant*)
  - it may get stuck in `handler()`
  - signal handler gets executed till ***completion***
    - in general, keep it simple and brief

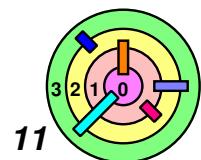
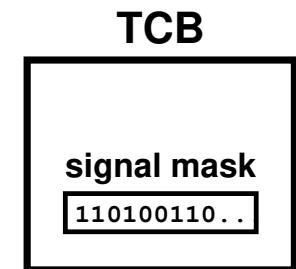
— yes, you can deadlock with yourself even if you only have one thread



# Masking (Blocking) Signals

→ Solution: control signal delivery by *masking/blocking the signal*

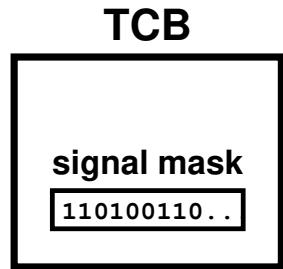
- don't mask/block all signals, just the ones you want
- a set of signals is represented as a set of bits called `sigset_t`
  - which is just an `unsigned int`
  - if a mask bit is 1, the corresponding signal is *blocked*; otherwise, the corresponding signal is *unblocked*
- when a child thread is created, it *inherits* signal mask from the parent thread



# Masking (Blocking) Signals

→ To examine or change the signal mask of the calling process

```
#include <signal.h>
int sigprocmask(
    int how,
    const sigset_t *set,
    sigset_t *old);
```



→ how is one of three commands:

- **SIG\_BLOCK**: the new signal mask is the union of the current signal mask and (\*set)
- **SIG\_UNBLOCK**: the new signal mask is the intersection of the current signal mask and the complement of (\*set)
- **SIG\_SETMASK**: the new signal mask is (\*set)

# sigset\_t

- There are bunch of functions to manipulate `sigset_t`
  - be careful, with *some APIs*, 1 means to *allowed/unblock* a signal, and with other APIs, 1 means to *blocked* a signal

- To clear a set:

```
int sigemptyset(sigset_t *set);
```

- To add or remove a signal from the set:

```
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

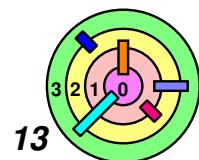
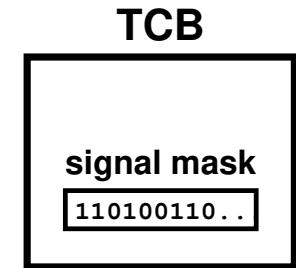
- Example: to refer to both SIGHUP and SIGINT:

```
sigset_t set;
```

```
sigset_t set;
```

```
sigemptyset(&set);
sigaddset(&set, SIGHUP);
sigaddset(&set, SIGINT);
```

```
sigfillset(&set);
sigdelset(&set, SIGHUP);
sigdelset(&set, SIGINT);
```



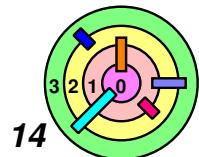
# Example 1: Correct Way of Waiting for a Signal

```

sigset_t set, oldset;
sigemptyset(&set);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, &oldset);
    /* SIGALRM now masked */
setitimer(ITIMER_REAL, &timerval, 0);
    /* SIGALRM sent in ~one millisecond */
sigfillset(&set);
sigdelset(&set, SIGALRM);
sigsuspend(&set); /* wait for it safely */
    /* SIGALRM masked again */
...
sigprocmask(SIG_SETMASK, &oldset, (sigset_t *) 0);
    /* SIGALRM unmasked */

```

- `sigsuspend()` **replaces** the caller's signal mask with the set of signals pointed to by the argument
  - in the above, all signals are blocked/masked except for SIGALRM
  - **atomically unblocks** the signal and **waits** for the signal



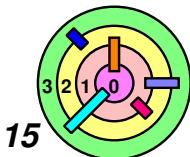
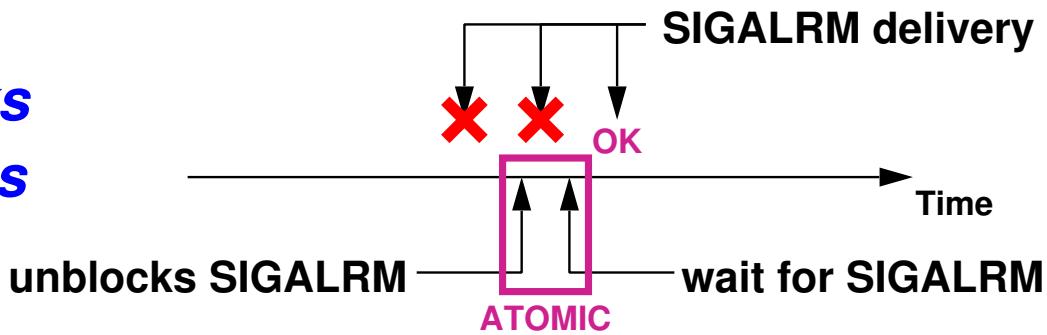
# Example 1: Correct Way of Waiting for a Signal

```

sigset_t set, oldset;
sigemptyset (&set);
sigaddset (&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, &oldset);
    /* SIGALRM now masked */
setitimer(ITIMER_REAL, &timerval, 0);
    /* SIGALRM sent in ~one millisecond */
sigfillset (&set);
sigdelset (&set, SIGALRM);
sigsuspend(&set); /* wait for it safely */
    /* SIGALRM masked again */
...
sigprocmask(SIG_SETMASK, &oldset, (sigset_t *) 0);
    /* SIGALRM unmasked */

```

- `sigsuspend()`
  - **atomically unblocks** the signal and **waits** for the signal



# Async-Signal Safety

- There is only *one* correct way to wait for an asynchronous event *you generated*
  - e.g., an alarm that *you* wind up and wait for is an asynchronous event you generate
  
- Step 1) **block** the asynchronous event
- Step 2) do something that will cause the asynchronous event to get generated
- Step 3) **unblock** the event and **wait** for the event in *one atomic operation*
  
- There is only *one* correct way to wait for an asynchronous event *someone else generated*
  - e.g., wait for a guard to become true in a guarded command
  
- Step 1) **block** the asynchronous event
- Step 2) check if the event has been generated, if not, **unblock** the event and **wait** for the event in *one atomic operation*



## Example 2: Correct Way to Handle Status Update

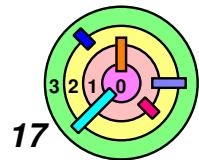
```
#include <signal.h>
computation_state_t state;
sigset_t set;

int main() {
    void handler(int);
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigset(SIGINT, handler);
    long_running_proc();
    return 0;
}
```

- now SIGINT cannot be delivered in update\_state()

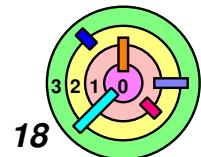
```
void long_running_proc() {
    while (a_long_time) {
        sigset_t old_set;
        sigprocmask(
            SIG_BLOCK,
            &set,
            &old_set);
        update_state(&state);
        sigprocmask(
            SIG_SETMASK,
            &old_set,
            0);
        compute_more();
    }
}

void handler(int signo) {
    display(&state);
}
```



# Signals and Threads

- In Unix, signals are sent to *processes*, not threads!
  - in a single-threaded process, it's obvious which thread would handle the signal
  - in a multi-threaded process, it's not so clear
    - in POSIX, the signal is delivered to a thread chosen *at random*
- What about the signal mask (i.e., blocked/enabled signals)?
  - should one set of sigmask affect all threads in a process?
  - or should each thread get its own sigmask?
    - this certainly makes more sense
- POSIX rules for a multithreaded process:
  - the thread that is to receive the signal is chosen *randomly* from the set of threads that do not have the signal blocked
    - if all threads have the signal blocked, then the signal remains pending until some thread unblocks it
      - ◆ at which point the signal is delivered to that thread
  - child thread *inherits* signal mask from parent thread



# Synchronizing Asynchrony

```

some_state_t state;
sigset_t set;

main() {
    pthread_t thread;
    sigemptyset(&set);
    sigaddset(&set,
              SIGINT);
    sigprocmask(
        SIG_BLOCK,
        &set, 0);
    // main thread           S21-Q14
    //     blocks SIGINT
    pthread_create(
        &thread, 0,
        monitor, 0);
    long_running_proc();
}

```

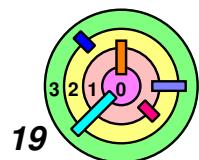
```

void long_running_proc() {
    while (a_long_time) {
        pthread_mutex_lock(&m);
        update_state(&state);
        pthread_mutex_unlock(&m);
        compute_more();
    }
}

void *monitor() {
    int sig;
    while (1) {
        sigwait(&set, &sig);
        pthread_mutex_lock(&m);
        display(&state);
        pthread_mutex_unlock(&m);
    }
    return (0);
}

```

- = this is **PREFERRED**, no need for signal handler!



F20-Q12

**sigwait**

```
int sigwait(sigset_t *set, int *sig)
```

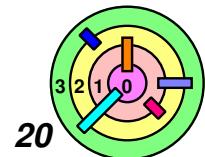
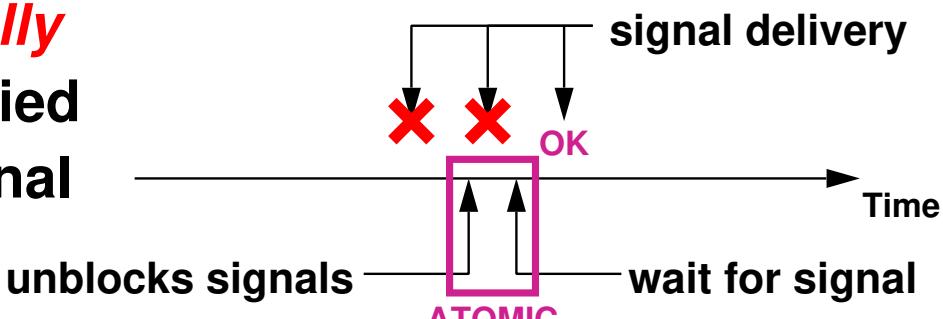
→ **sigwait () blocks until a signal specified in set is received**

- return which signal caused it to return in **sig**
- if you have a signal handler specified for **sig**, it will **not** get invoked when the signal is delivered
  - instead, **sigwait ()** will return

→ **You should make sure that all the threads in your process have these signals blocked!**

- this way, when **sigwait ()** is called, the calling thread temporarily becomes the **only** thread in the process who can receive the signal

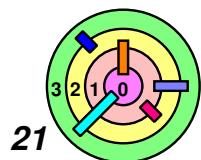
→ **sigwait (set) atomically unblocks signals specified in set and waits for signal delivery**



# Signals and Blocking System Calls

→ What if a signal is generated while a process is blocked in a system call?

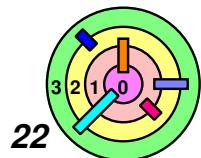
- 1) deal with it when the system call completes
  - 2) interrupt the system call, deal with signal, resume system call
  - 3) interrupt system call, deal with signal, return from system call with indication that something happened
- most systems choose (3)
- errno sets to EINTR to mean that the system call was not completed because it was interrupted by a signal
    - ◊ this is the errno for the thread that was "deviated" to execute the signal handler
  - this may be the reason why `pthread_cond_wait()` may return "spontaneously" even when the CV has not been signaled/broadcasted
    - ◊ what if this thread was borrowed to deliver a signal?



# Interrupted System Calls

```
while(read(fd, buffer, buf_size) == -1) {  
    if (errno == EINTR) {  
        /* interrupted system call; try again */  
        continue;  
    }  
    /* the error is more serious */  
    perror("big trouble");  
    exit(1);  
}
```

- need to check the return value of `read()` because `read()` can return when less than `buf_size` bytes have been read
- can use similar code for `write()`
  - same consideration as `read()`
- please note that the above code is incomplete, it needs to handle the case where `read()` return 0 to mean end-of-input

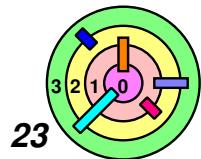


# Interrupted While Underway

```

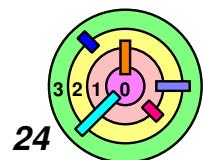
remaining = total_count; /* write this many bytes */
bptr = buf;             /* starting from here */
for ( ; ; ) {
    num_xfrd = write(fd, bptr, remaining);
    if (num_xfrd == -1) {
        if (errno == EINTR) {
            /* interrupted early */
            continue;
        }
        perror("big trouble");
        exit(1);
    }
    if (num_xfrd < remaining) {
        /* interrupted in the middle of write() */
        remaining -= num_xfrd;
        bptr += num_xfrd;
        continue;
    }
    /* success! */
    break;
}

```



# Interrupted System Calls

- If a thread can "see" signal delivery (i.e., "deviated to execute a signal handler"), every `read()` and `write()` call in that thread needs to look like the previous slides
- much easier if you use a *signal-catching thread*
    - and *block all appropriate signals* in all "regular" threads
    - for warmup2, it is strongly encouraged that you do it this way to catch <Ctrl+C> and avoid using a signal handler
      - ◊ when `sigwait()` returns, lock mutex, set global flag, cancel packet arrival and token depositing threads, broadcast CV, unlock mutex, and self-terminate
      - ◊ according to spec, you must not cancel server threads
      - ◊ will talk about cancellation shortly

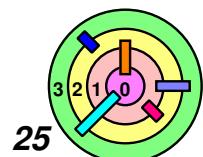


# Inside A Signal Handler

→ Which library routines are safe to use *within* signal handlers?

|               |            |           |             |             |                  |
|---------------|------------|-----------|-------------|-------------|------------------|
| access        | dup2       | getgroups | rename      | sigprocmask | time             |
| aio_error     | dup        | getpgrp   | rmdir       | sigqueue    | timer_getoverrun |
| aio_suspend   | execle     | getpid    | sem_post    | sigsuspend  | timer_gettime    |
| alarm         | execve     | getppid   | setgid      | sleep       | timer_settime    |
| cfgetispeed   | _exit      | getuid    | setpgid     | stat        | times            |
| cfgetospeed   | fcntl      | kill      | setsid      | sysconf     | umask            |
| cfsetispeed   | fdatasync  | link      | setuid      | tcdrain     | uname            |
| cfsetospeed   | fork       | lseek     | sigaction   | tcflow      | unlink           |
| chdir         | fstat      | mkdir     | sigaddset   | tcflush     | utime            |
| chmod         | fsync      | mkfifo    | sigdelset   | tcgetattr   | wait             |
| chown         | getegid    | open      | sigemptyset | tcgetpgrp   | waitpid          |
| clock_gettime | geteuid    | pathconf  | sigfillset  | tcsendbreak | write            |
| close         | getgid     | pause     | sigismember | tcsetattr   |                  |
| creat         | getoverrun | pipe      | sigpending  | tcsetpgrp   |                  |

→ Note: in general, you should only do what's absolutely necessary inside a signal handler (and figure out where to do the rest)



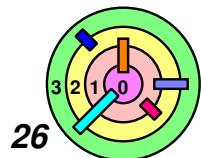
# Cancellation

→ The user pressed <Ctrl+C>

- or a request is generated to terminate the process
- the chores being performed by the remaining threads are no longer needed
- in general, we may just want to cancel a bunch of threads and not the entire process

→ Concerns

- getting cancelled at an inopportune moment
  - a mutex left locked
  - a data structure is left in an inconsistent state
    - ◊ e.g., you get a cancellation request when you are in the middle of a `insert()` operation into a doubly-linked list and `insert()` is protected by a mutex
- cleaning up (free up resources that only this thread can free up)
  - memory leaks
  - unlocking mutex if locked



# Cancellation State & Type

- Send cancellation request to a thread (*this is a non-blocking call*)

```
pthread_cancel(thread)
```

- Cancels enabled or disabled

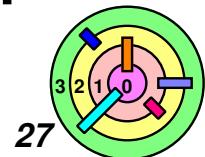
```
int pthread_setcancelstate(
    { PTHREAD_CANCEL_DISABLE,
      PTHREAD_CANCEL_ENABLE },
    &oldstate)
```

- Asynchronous vs. deferred cancels

```
int pthread_setcanceltype (
    { PTHREAD_CANCEL_ASYNCHRONOUS,
      PTHREAD_CANCEL_DEFERRED },
    &oldttype)
```

- By default, a thread has cancellation *enabled* and *deferred*

- it's for a good reason
  - if you are going to change it, you must ask yourself, "Why?" and "Are you sure this is really a good idea?"

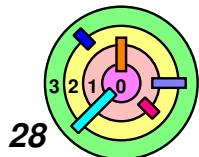


# POSIX Cancellation Rules



POSIX threads cancellation rules (part 1):

- when `pthread_cancel()` gets called, the target thread is marked as having a *pending cancel*
  - the thread that called `pthread_cancel()` does not wait for the cancel to take effect
- if the target thread has cancellation *disabled*, the target thread stays in the pending cancel state
- if the target thread has cancellation *enabled* ...
  - if the cancellation type is *asynchronous*, the target thread immediately *acts on cancel* (i.e., cancellation is "delivered" by "deviating" the thread to call `pthread_exit()`)
  - if the cancellation type is *deferred*, cancellation is *delayed* until it reaches a *cancellation point* in its execution
    - ◆ cancellation points correspond to points in the thread's execution at which it is safe to *act on cancel*

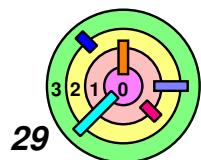


# Cancellation Points

**aio\_suspend**  
**close**  
**creat**  
**fcntl (when F\_SETLCKW  
is the command)**  
**fsync**  
**mq\_receive**  
**mq\_send**  
**msync**  
**nanosleep**  
**open**  
**pause**  
**pthread\_cond\_wait**  
**pthread\_cond\_timedwait**

**pthread\_join**  
**pthread\_testcancel**  
**read**  
**sem\_wait**  
**sigsuspend**  
**sigtimedwait**  
**sigwait**  
**sigwaitinfo**  
**sleep**  
**system**  
**tcdrain**  
**wait**  
**waitpid**  
**write**

- **pthread\_mutex\_lock () is not on the list!**
- **pthread\_testcancel () creates a cancellation point**
  - useful if a thread contains no other cancellation point



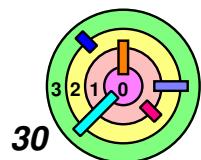
# POSIX Cancellation Rules



POSIX threads cancellation rules (part 2):

- when a thread *acts on cancel*
  - it calls `pthread_exit()`
  - in `pthread_exit()`, it first walks through a *stack of cleanup handlers*
    - ◊ when stack is empty, the thread goes into the **zombie state**
  - remember that the thread that called `pthread_cancel()` does not wait for the cancel to take effect
    - ◊ it may join and wait for the target thread to terminate

```
pthread_cleanup_push(  
    (void) (*routine) (void *),  
    void *arg)  
pthread_cleanup_pop(int execute)
```



# Example

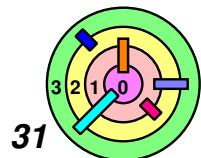
```
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));

    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);

    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```

- How can this thread control when it acts on cancel?  
— so it doesn't leak memory



# Example

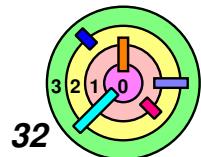
```

list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, 0);
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, 0);
    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}

```

- How can this thread control when it acts on cancel?
- so it doesn't leak memory
  - although this implementation is technically "correct", long delay may not be acceptable
    - need to *respond in a timely manner*



# Example

```

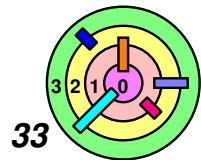
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item);
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);

    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}

```

- Can act on cancel inside GetDataItem()
- in this case, will invoke `free(item)`
  - in C library, `free()` is defined as: `void free(void *ptr);`
    - perfectly matches the argument types for `pthread_cleanup_push()`



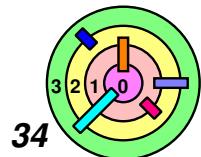
# Example

```
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item);
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);

    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```

- What if it acts on cancel inside printf()  
— will end up calling free(item) twice  
○ can cause segmentation fault later

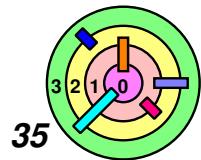


# Example

```
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item);
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);
    pthread_cleanup_pop(0);
    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```

- What if it acts on cancel inside printf()  
— will end up calling free(item) twice  
    ○ can cause segmentation fault later  
— pop free(item) off the cleanup stack



# Example

```

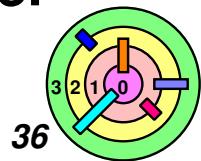
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item); // {
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);
    pthread_cleanup_pop(0);           // }
    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}

```

*// GetDataItem() contains many cancellation points*  
*GetDataItem(&item->value);*  
*pthread\_cleanup\_pop(0); // }*  
*insert(item); // add item to a global list*  
*must match up (like a pair of brackets)*

- **pthread\_cleanup\_push()** and the corresponding **pthread\_cleanup\_pop()** must match up (like a pair of brackets)
- must not call **pthread\_cleanup\_push()** in one function and call the corresponding **pthread\_cleanup\_pop()** in another
  - compile-time error

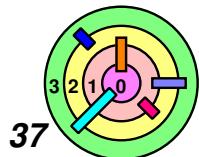


# Cancellation and Cleanup

```
void close_file(int fd) {
    close(fd);
}

fd = open(file, O_RDONLY);
pthread_cleanup_push(close_file, fd);
while(1) {
    read(fd, buffer, buf_size);
    // ...
}
pthread_cleanup_pop(0);
```

- should close any opened files when you clean up
- int is compatible with void\*
- well, sort of
- void\* can be a 64-bit quantity, so may need to be careful  
(best to be explicit)



# Cancellation and Conditions

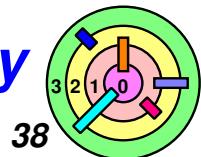
```

pthread_mutex_lock(&m);
pthread_cleanup_push(CleanupHandler, argument);

while(should_wait)
    pthread_cond_wait(&cv, &m);
// ... (code containing other cancellation points)
pthread_cleanup_pop(0);
pthread_mutex_unlock(&m);

```

- should `CleanupHandler()` call `pthread_mutex_unlock()`?
  - remember, if the thread is canceled between `push()` and `pop()`, we need to ensure that the mutex is *locked*
  - `pthread_cond_wait()` is a cancellation point
    - ◊ must not unlock the mutex twice!
- should `CleanupHandler()` call `pthread_mutex_lock()` then call `pthread_mutex_unlock()`?
  - what if the mutex is locked?
- application cannot solve this problem since there is *no way to check if a mutex is locked or not*

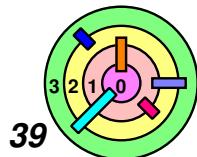


# Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(pthread_mutex_unlock, &m);

while(should_wait)
    pthread_cond_wait(&cv, &m);
// ... (code containing other cancellation points)
pthread_cleanup_pop(1);
```

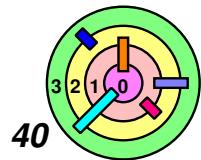
- pthreads library implementation ensures that a thread, when acting on a cancel inside `pthread_cond_wait()`, would first lock the mutex, before calling the cleanup routines
  - this way, the above code would work correctly



# Warmup2 Cancellation

→ Only packet arrival and token depositing threads are allowed to be canceled

- use a <Ctrl+C>-catching thread (i.e., use `sigwait()`)
- make cancellation requests only when mutex is *locked*
  - this makes it impossible for those threads to act on cancel when they have the mutex locked
- can make the following simplification for these two threads:
  - at the start of their first procedures, *disable cancellation*
  - right before calling `usleep()`, *enable cancellation*
  - when `usleep()` returns, *disable cancellation again*
  - this way, during the time the mutex is locked, cancellation is always disabled
    - ◊ therefore, don't have to worry about using cleanup routines to unlock mutex
  - but didn't we just say that it's not a good idea to disable cancellation?
  - also, need to take care of a *race condition*

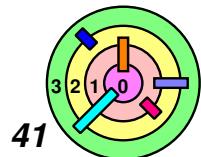


# Cancellation & C++

```
void tcode() {  
    A a1;  
    pthread_cleanup_push(handler, 0);  
    foo();  
    pthread_cleanup_pop(0);  
}  
  
void foo() {  
    A a2;  
    pthread_testcancel();  
}
```

- are the destructors of a1 and a2 getting called?
  - not sure
  - they should get called
  - some C++ implementation does not do this correctly!

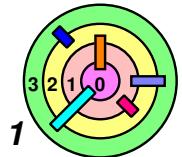
- Note: current C++ standard also does *not* support thread cancellation
- standard C++ threads must self-terminate!



# Ch 3: Basic Concepts

Bill Cheng

*<http://merlot.usc.edu/william/uscl/>*



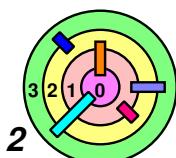
# What's Next?

- ➡ So far, we have talked about abstractions
  - processes, files, threads
    - stuff at the user level
- ➡ We are not ready to talk about the OS yet
- ➡ Next step is something in between

Abstractions  
(processes, files, threads)

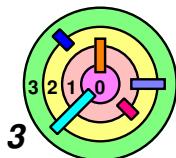
- context for execution
- I/O architecture
- dynamic storage allocation
- linking & loading
- booting

User  
OS



# 3.1 Context Switching

- ▶ Procedures
- ▶ Threads & Coroutines
- ▶ Systems Calls
- ▶ Interrupts



# Context Switching

## → The magic of OS

- to provide the *illusion* that applications run *concurrently* and each application thinks it's the only application running on the processor

## → The OS switches the processor from one application to another

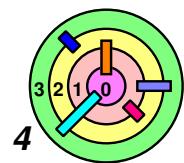
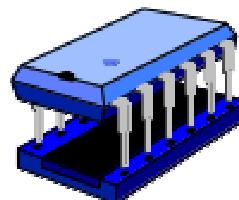
- switching happens *transparently* to the applications

## → What is the OS doing when an application is running?

Application1

Application2

Application3



# Context



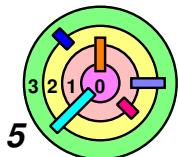
What's the execution context of a thread?

- if we are going to talk about context switching, we need to know what we are switching and how to get back



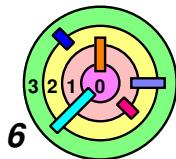
The **execution context** of a thread is the **current state** of our thread

- what does the execution context include?
  - CPU registers, including the **instruction pointer**, **stack pointer**, **base/frame pointer**, etc.
  - stack
  - open files
  - etc.
  - i.e., everything that may affect the execution of the thread
- turns out the stack is complicated
  - in reality, it's just the **current stack frame** of the current thread
  - what's **below** the current stack frame (and the rest of the address space) is also part of the thread's state/context



# 3.1 Context Switching

- ▶ ***Procedures***
- ▶ **Threads & Coroutines**
- ▶ **Systems Calls**
- ▶ **Interrupts**



# Subroutines

```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

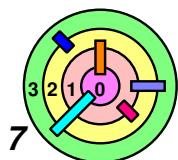
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}

```



You are in `main()` and are ready to call `sub()`

- how do you make sure that `sub()` has the right context to execute the code in `sub()`?
  - you need to *prepare/setup the context* for `sub()`
- how do you make sure that you can return from `sub()` and restore the `main()` context and continue to execute properly?
  - you need to first *save* the context of `main()`
  - after return from `sub()`, you need to *restore* the context of `main()` so `main()` can *resume* execution



# Subroutines

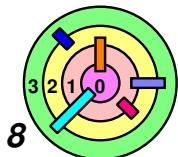
```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

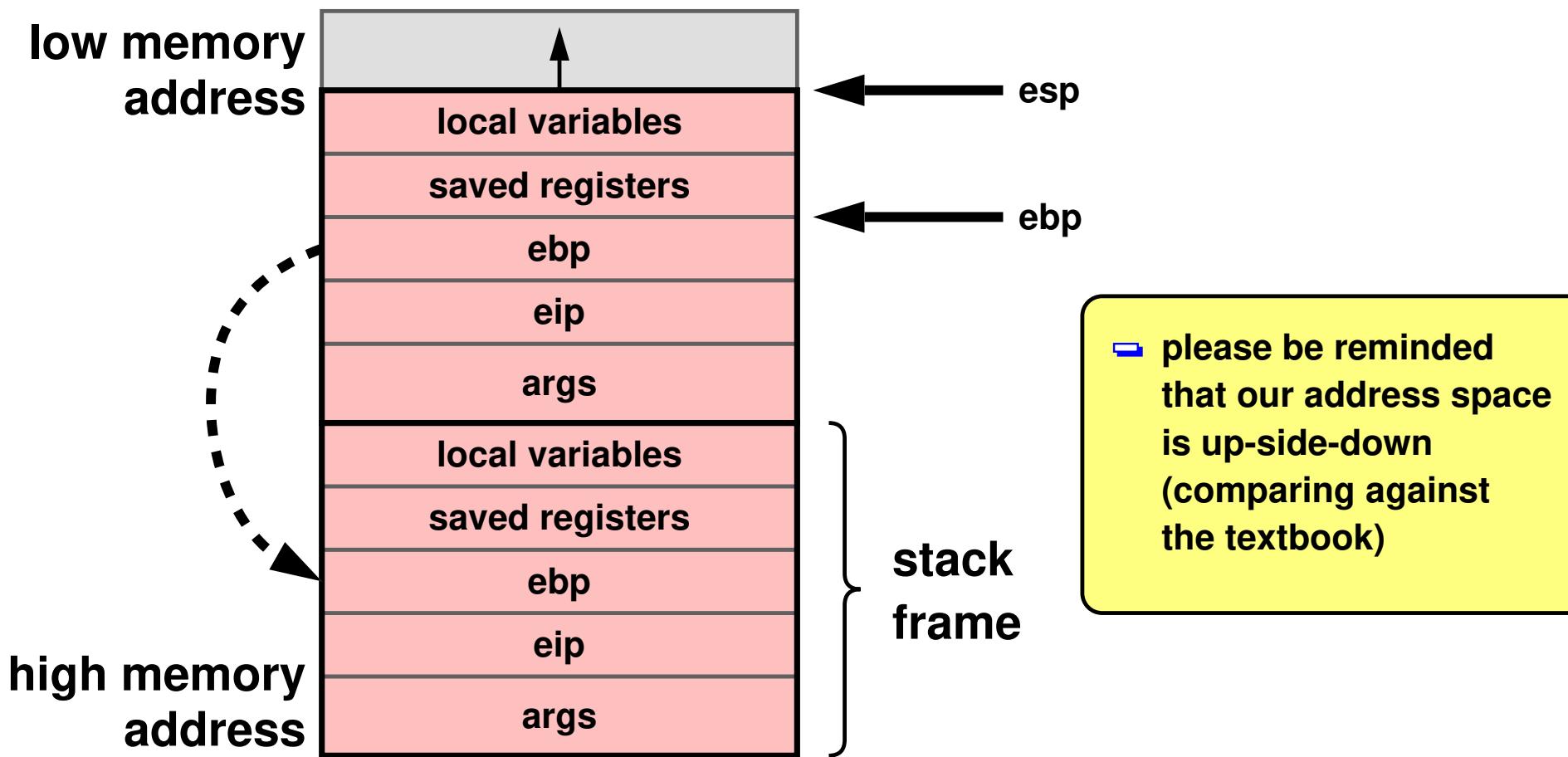
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}

```

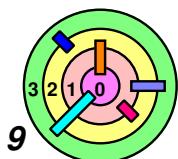
- ▶ The context of `main()` includes CPU registers, any global variables (none here) and its local variables, `i` and `a`
- ▶ The context of `sub()` includes
  - any global variables, none here
  - its local variables, `i` and `result`
  - its arguments, `x` and `y`
- ▶ Global variables are in fixed location in the address space
- ▶ ***Local variables*** and ***arguments*** are in ***current stack frame***



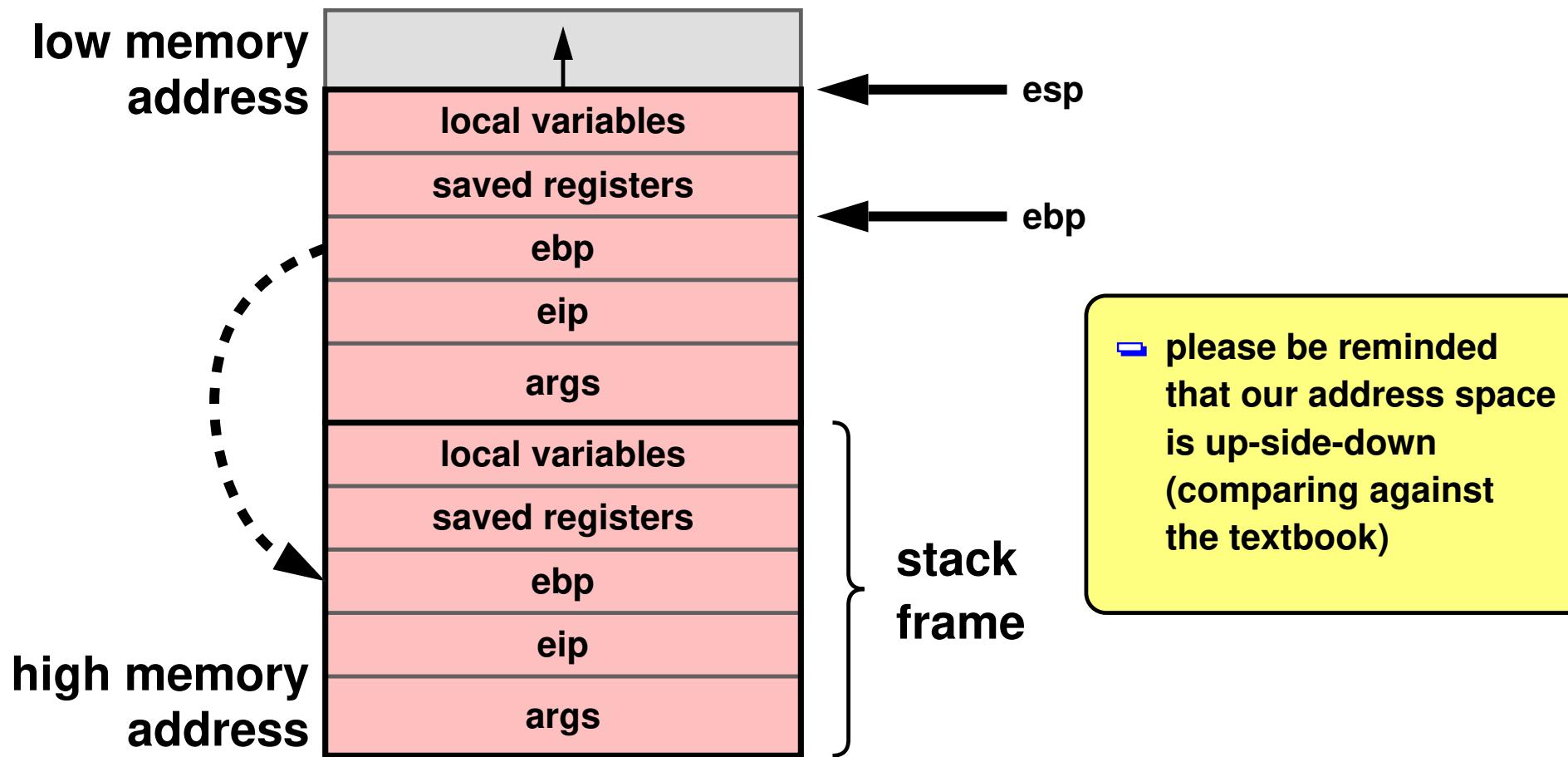
# Intel x86 (32-Bit): Subroutine Linkage



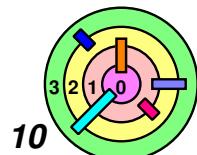
- **esp** points to the end/top of the current stack frame
  - it is used to prepare the next stack frame
- **eip** contains the caller's **instruction pointer** register
  - in the stack frame, this is the **return address!**



# Intel x86 (32-Bit): Subroutine Linkage



- **ebp** contains the caller's **base (frame) pointer** register
  - this is a link to the caller's **stack frame**
- **eax** contains the return value of a function
- some fields are not always present, compiler decides



# Intel x86 (32-Bit): Subroutine Linkage



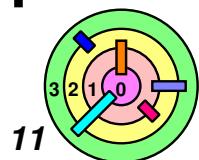
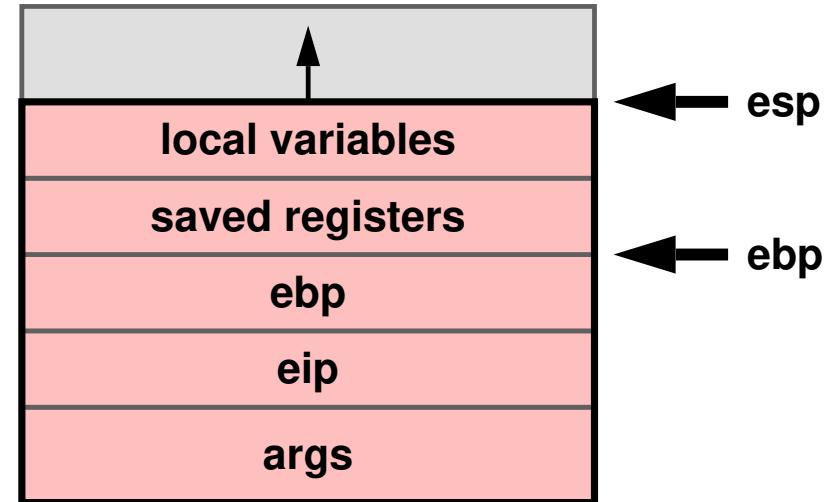
Who sets what?

- **args** is explicitly setup by the **caller**
- **eip** is copied into the stack frame by a "call" machine instruction in the **caller** function
- **ebp** is copied explicitly by the **callee**
- other registers are saved explicitly by the **callee** code in the "saved registers" region
  - as it turned out, for x86, some registers are designated to be saved by the callee code
- space for local variables is **created** explicitly by the **callee** code
  - local variables are **not** initialized automatically



What does the stack frame look like for the following function?

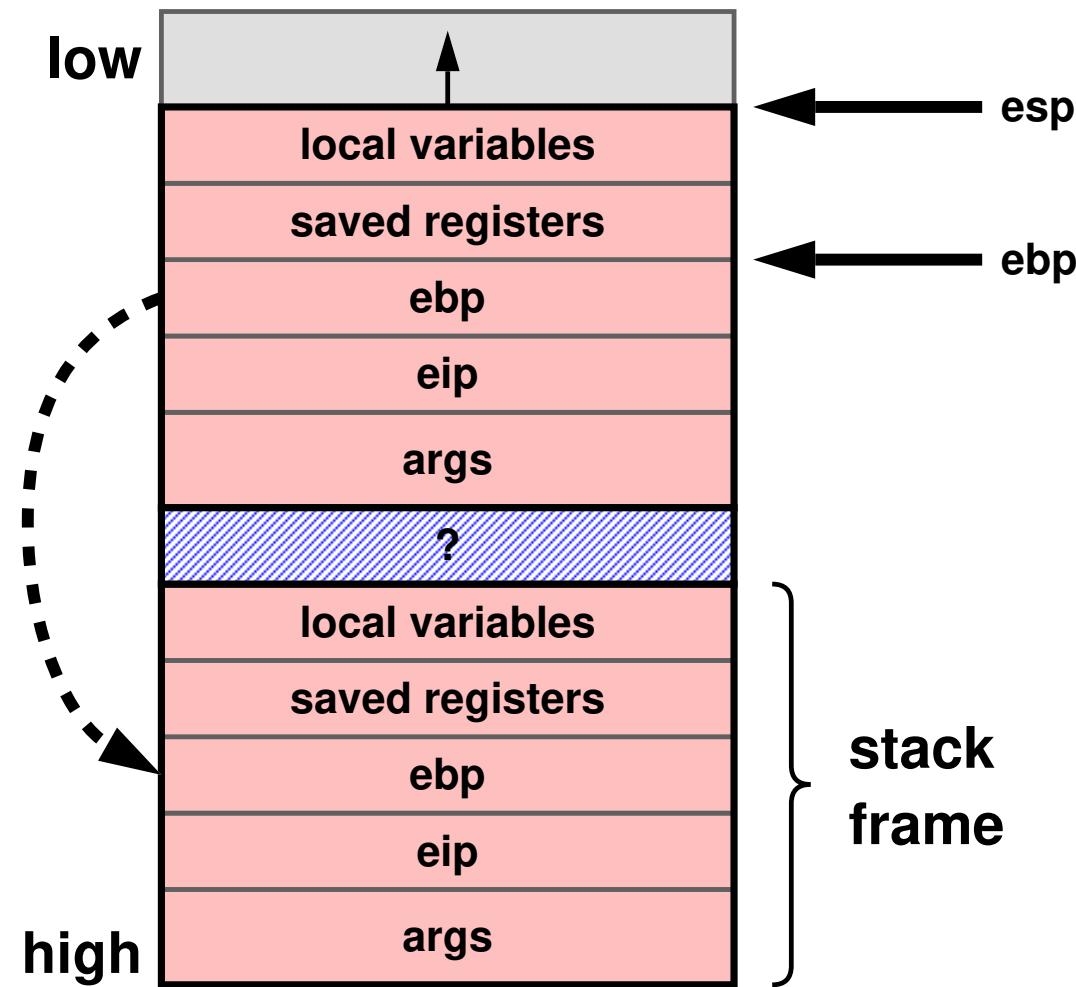
```
void func() { printf("I'm here.\n"); }
```



# Intel x86 (32-Bit): Subroutine Linkage

→ In reality, there can be stuff between stack frames

- e.g., by convention, specific registers are saved and restored by the caller (this can depend on the compiler)



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

```

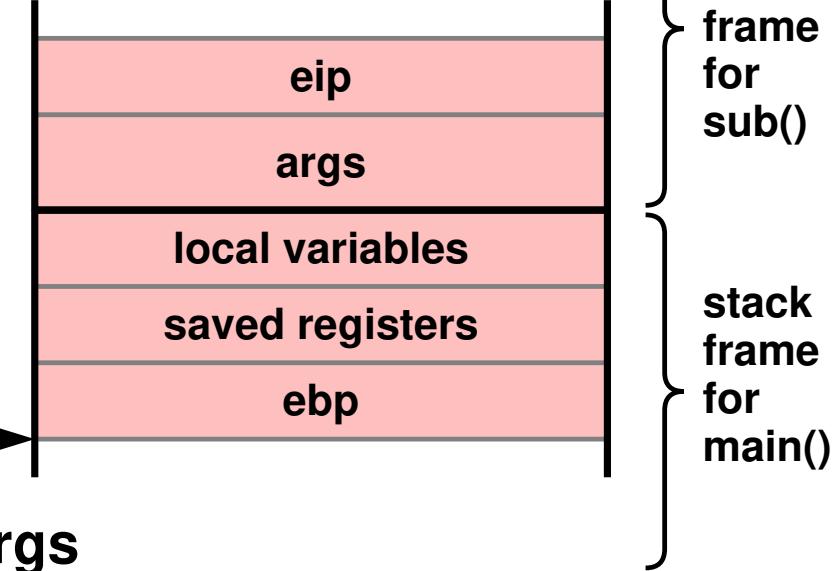
*textbook  
is wrong*

set up stack frame

push args

pop args;  
get result

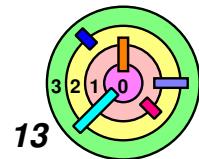
set return  
value and  
restore frame



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
  pushl %ebp
  movl %esp, %ebp
  pushl %esi
  pushl %edi
  subl $8, %esp
  ...
  pushl $1
  movl -12(%ebp), %eax
  pushl %eax
  call sub
  addl $8, %esp
  movl %eax, -16(%ebp)
  ...
  addl $8, %esp
  movl $0, %eax
  popl %edi
  popl %esi
  movl %ebp, %esp
  popl %ebp
  ret

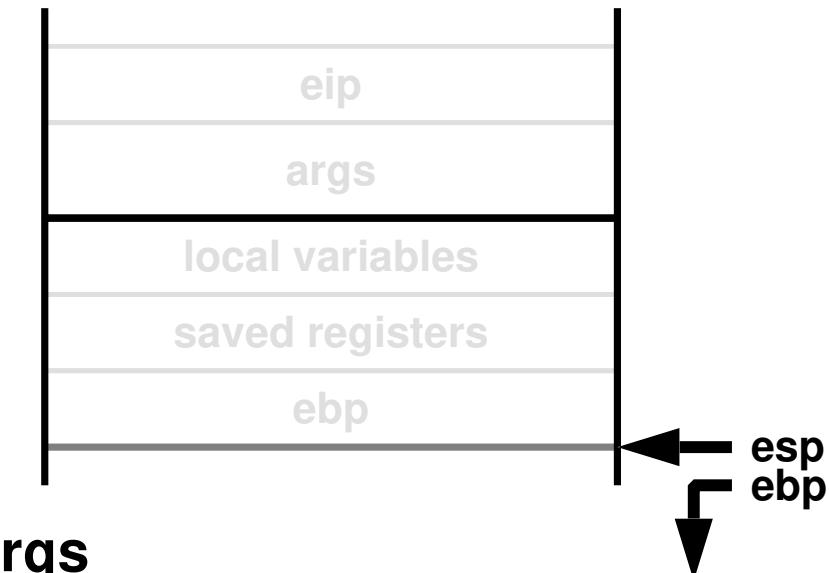
```

**set up stack frame**

**push args**

**pop args; get result**

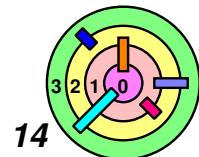
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

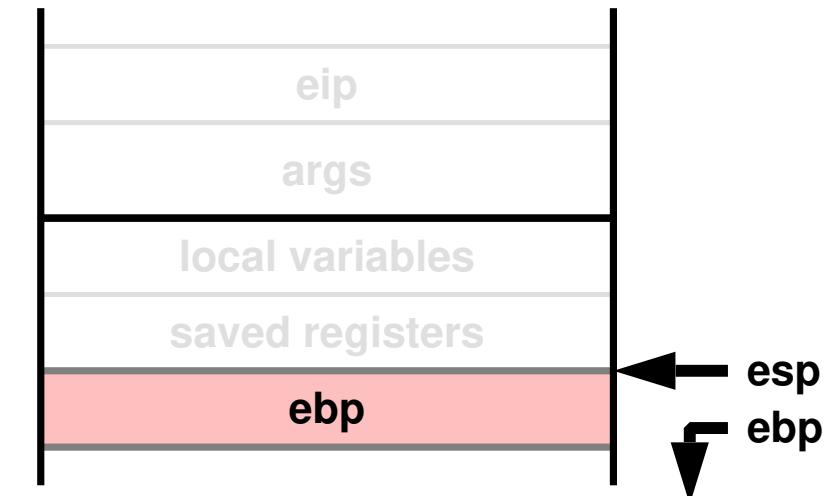
```

**set up stack frame**

**push args**

**pop args; get result**

**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

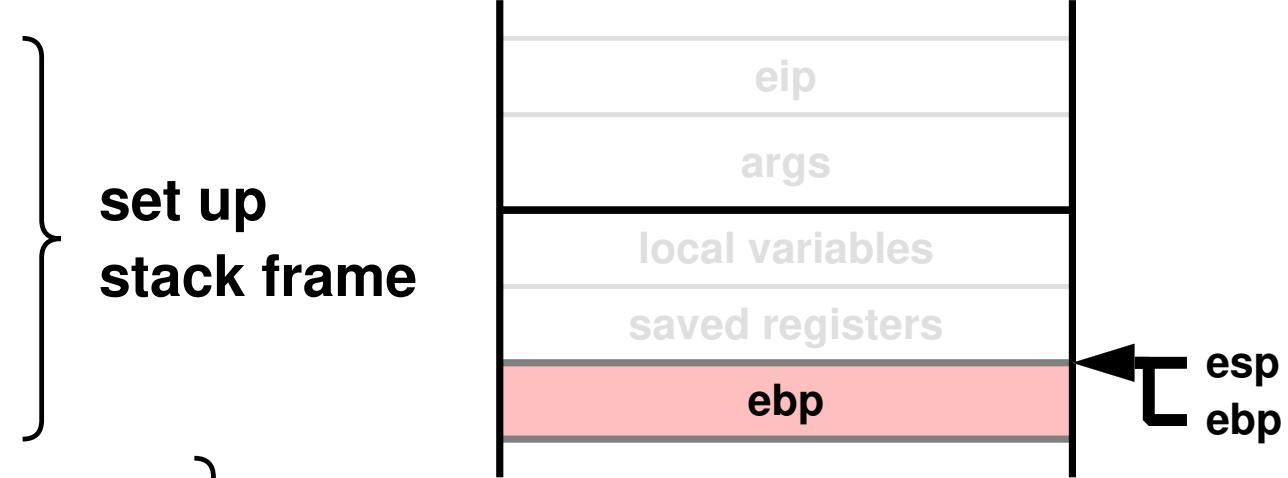
```

# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi           } set up stack frame
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax } push args
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp) } pop args; get result
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp } set return value and restore frame
    popl %ebp
    ret

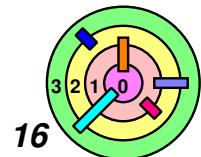
```



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi           } set up stack frame
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax             } push args
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)   } pop args; get result
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

```

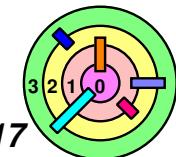
17



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



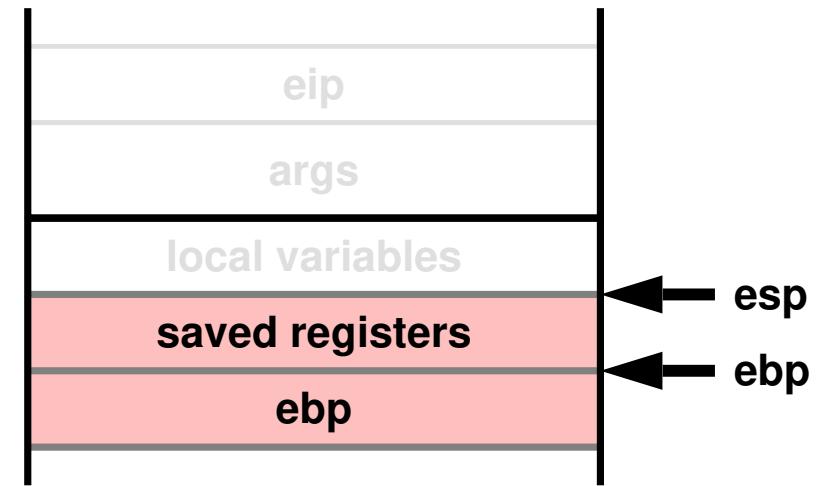
# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp           } set up stack frame
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

```

**set up stack frame**  
**push args**  
**pop args; get result**  
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```

# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

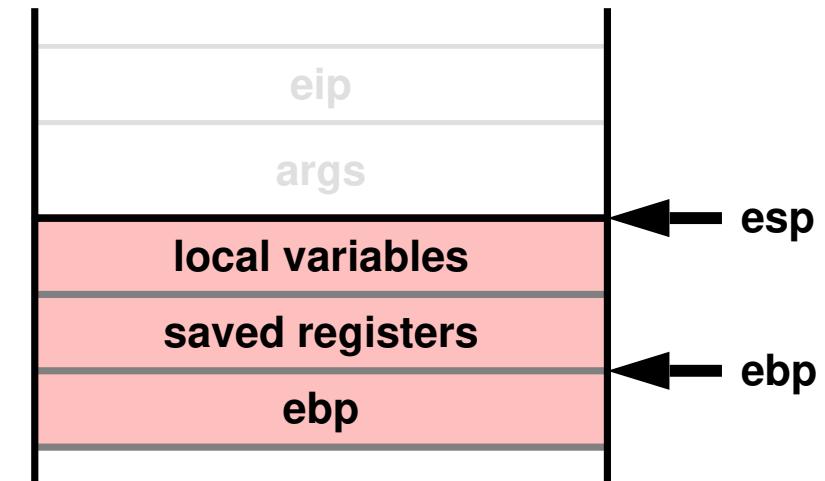
```

**set up stack frame**

**push args**

**pop args; get result**

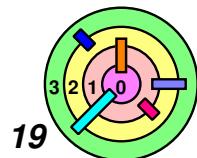
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl %eax
    movl -4(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

```

**set up stack frame**

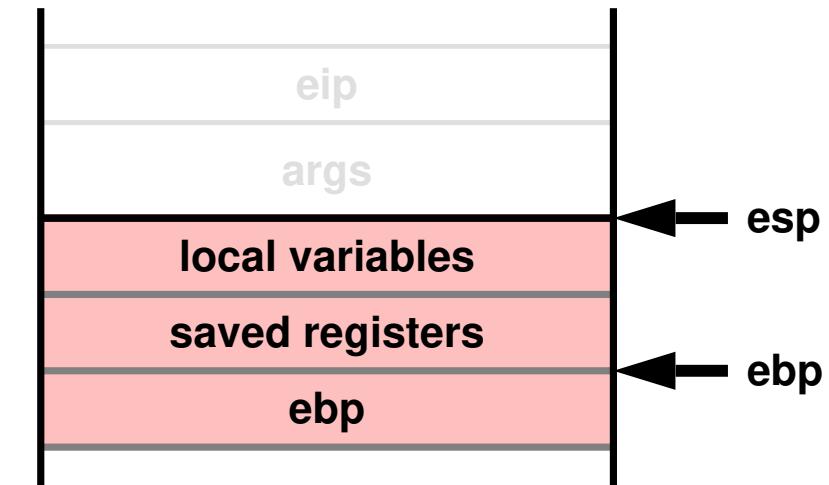
**push args**

**pop args; get result**

**set return value and restore frame**

**now you are ready to execute C code**

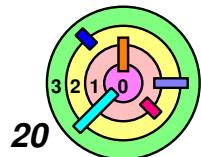
**in gdb, if you break in main(), the breakpoint is set here**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```
main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...

```

**set up stack frame**

```
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub

```

```
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...

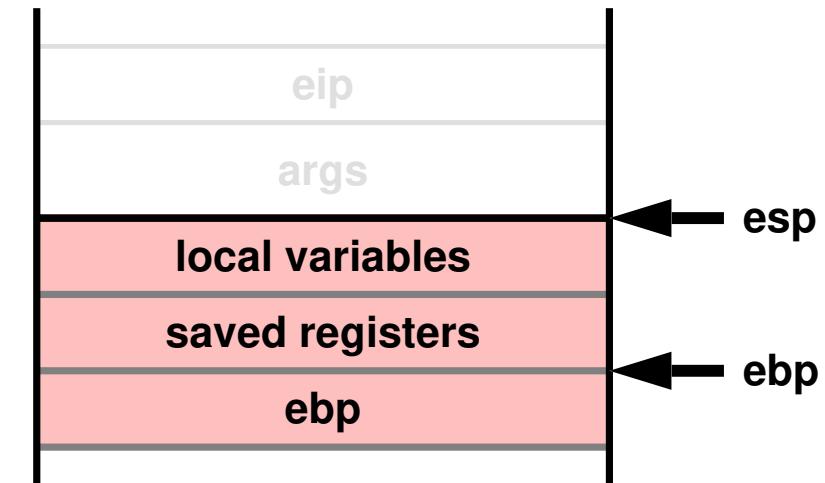
```

```
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret
```

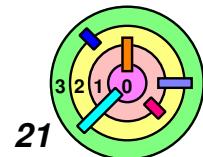
**push args**

**pop args;  
get result**

**set return  
value and  
restore frame**



```
int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}
```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    → movl -12(%ebp), %eax
    → pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

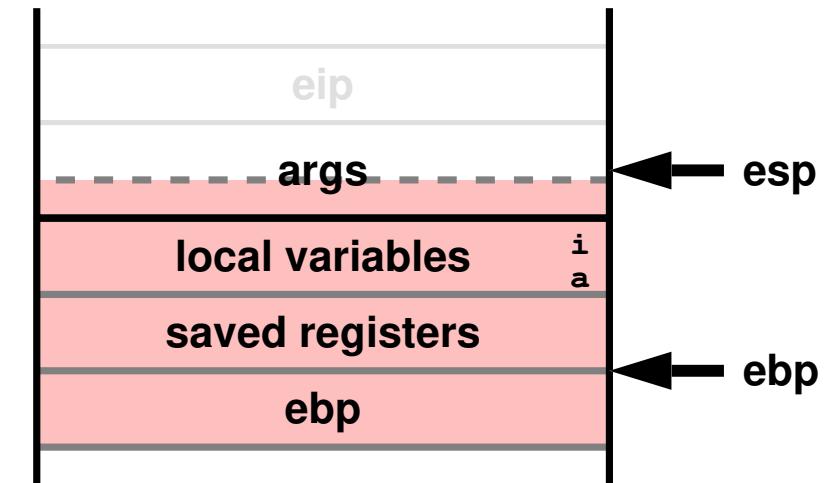
```

**set up stack frame**

**push args**

**pop args;  
get result**

**set return  
value and  
restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```

# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

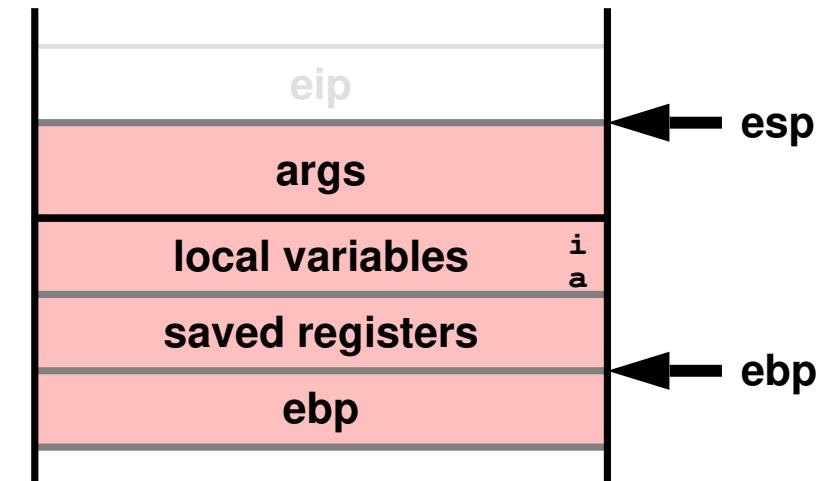
```

**set up stack frame**

**push args**

**pop args; get result**

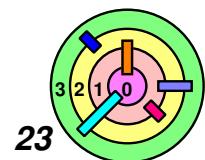
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    → addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

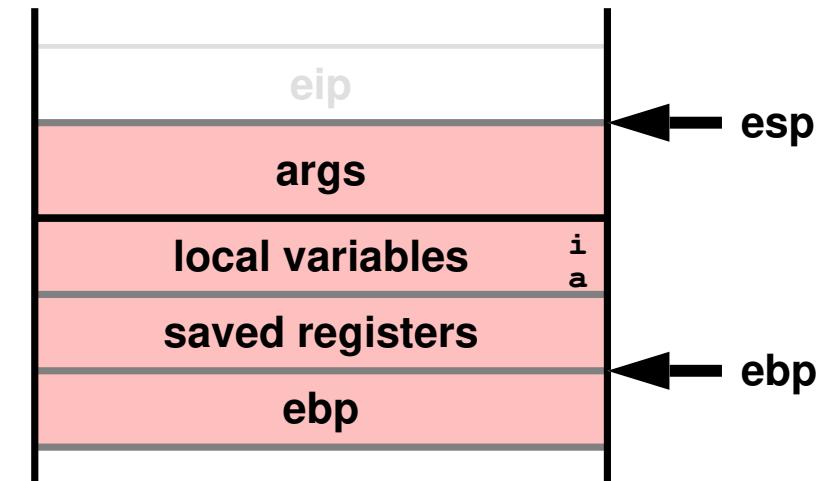
```

**set up stack frame**

**push args**

**pop args; get result**

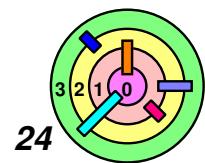
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

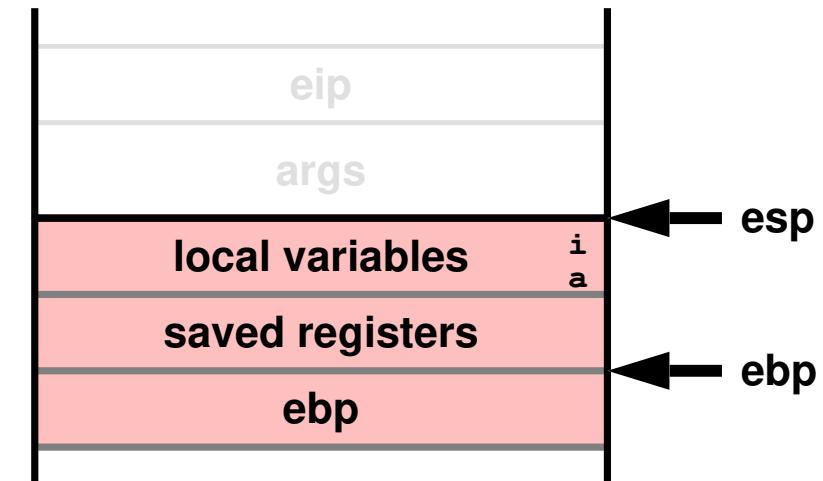
```

**set up stack frame**

**push args**

**pop args; get result**

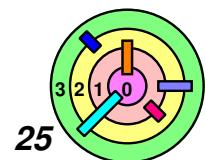
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

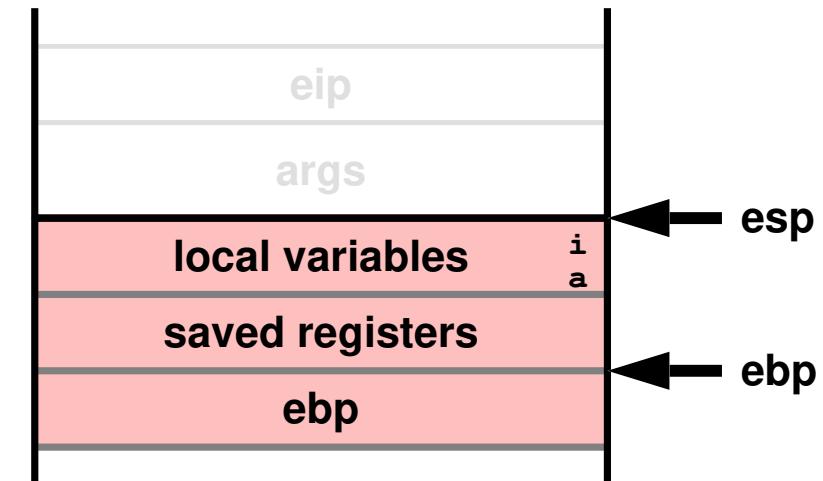
```

**set up stack frame**

**push args**

**pop args; get result**

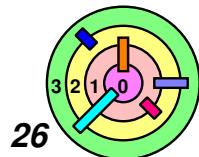
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl %eax, -16(%ebp)
    popl %ebp
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

```

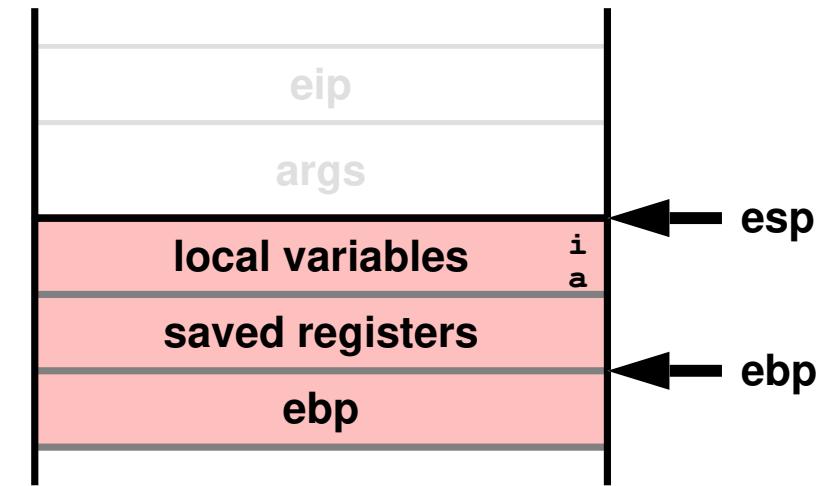
} set up stack frame

} push args

} pop args; result

} value and restore frame

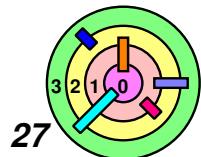
**what happens when you make a subroutine call now?**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    → addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

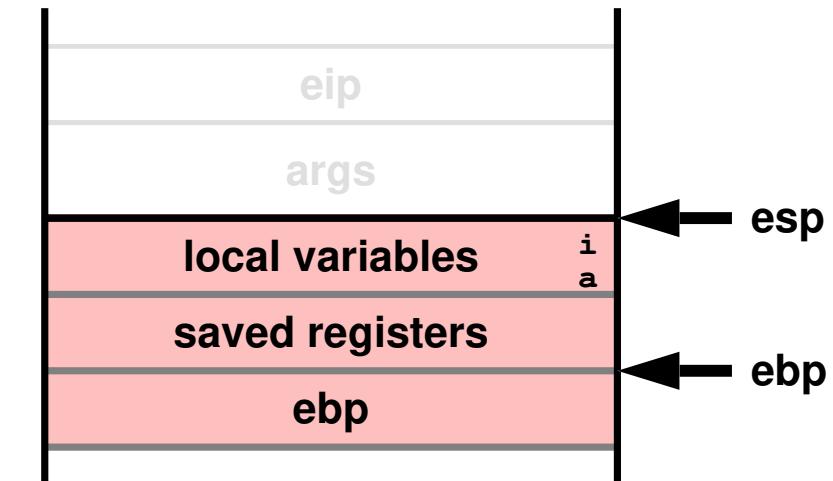
```

**set up stack frame**

**push args**

**pop args; get result**

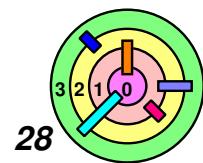
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

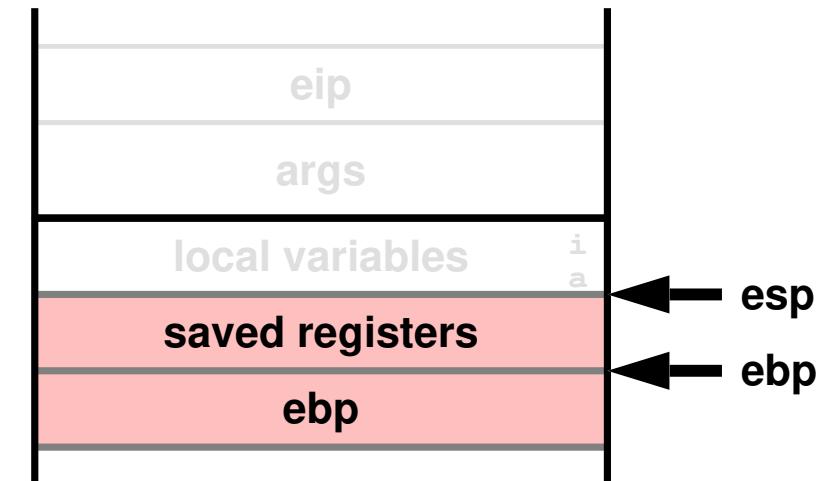
```

**set up stack frame**

**push args**

**pop args; get result**

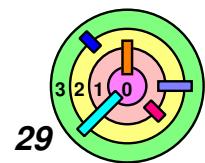
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    →popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

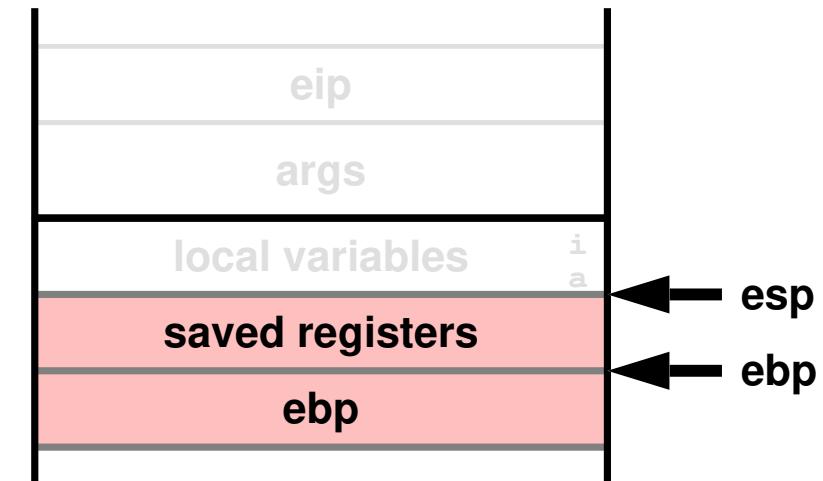
```

**set up stack frame**

**push args**

**pop args; get result**

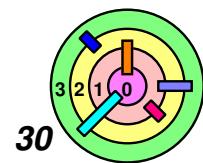
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    → popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

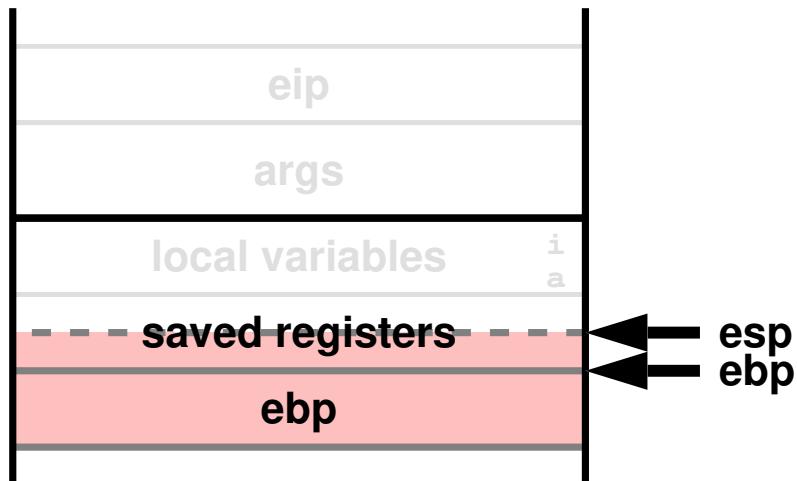
```

**set up stack frame**

**push args**

**pop args; get result**

**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```

# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

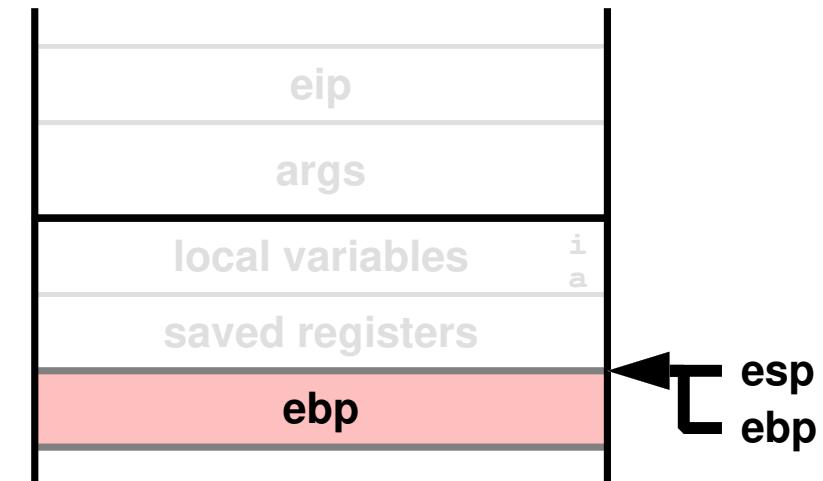
```

**set up stack frame**

**push args**

**pop args; get result**

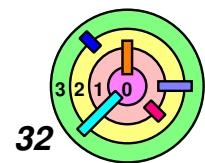
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    → popl %ebp
    ret

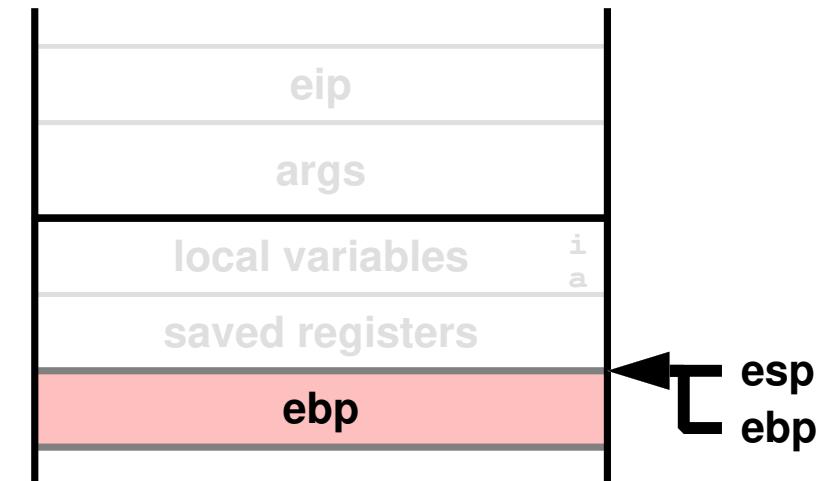
```

**set up stack frame**

**push args**

**pop args; get result**

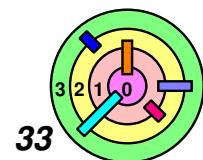
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



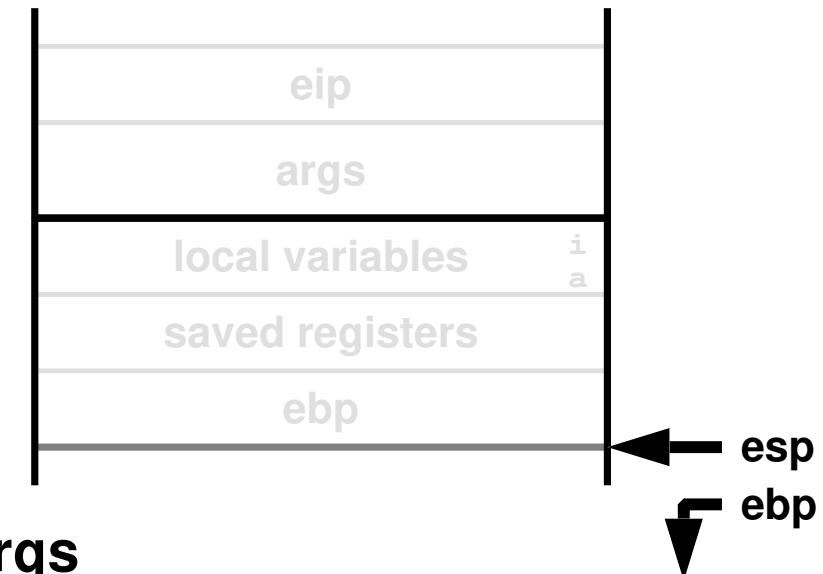
# Intel x86: Subroutine Code (1)

```

main:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

```

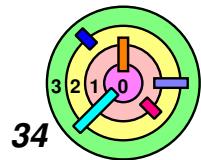
**set up stack frame**  
**push args**  
**pop args; get result**  
**set return value and restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

```



# Intel x86: Subroutine Code (2)

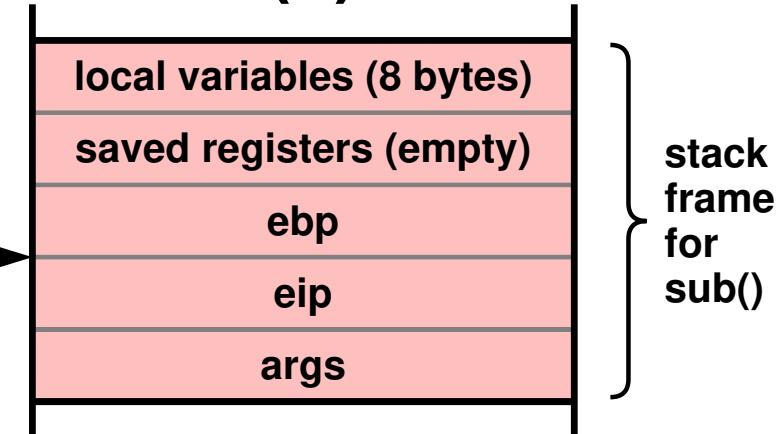
```

sub:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $1, -4(%ebp)
    movl $0, -8(%ebp)
    movl -4(%ebp), %ecx
    movl -8(%ebp), %eax

beginloop:
    cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop

endloop:
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
  
```

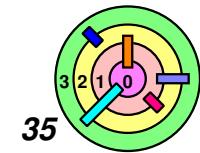
} set up esp →



```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
  
```

} restore



# Intel x86: Subroutine Code (2)

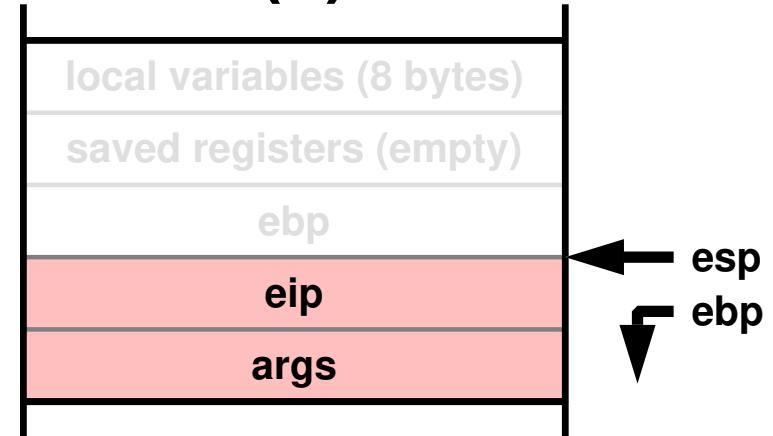
```

sub:
    → pushl %ebp } enter
    movl %esp, %ebp
    subl $8, %esp
    movl $1, -4(%ebp)
    movl $0, -8(%ebp)
    movl -4(%ebp), %ecx
    movl -8(%ebp), %eax

beginloop:
    cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop

endloop:
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp } leave
    popl %ebp
    ret

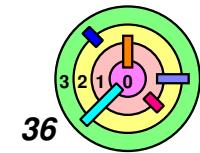
```



```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}

```



# Intel x86: Subroutine Code (2)

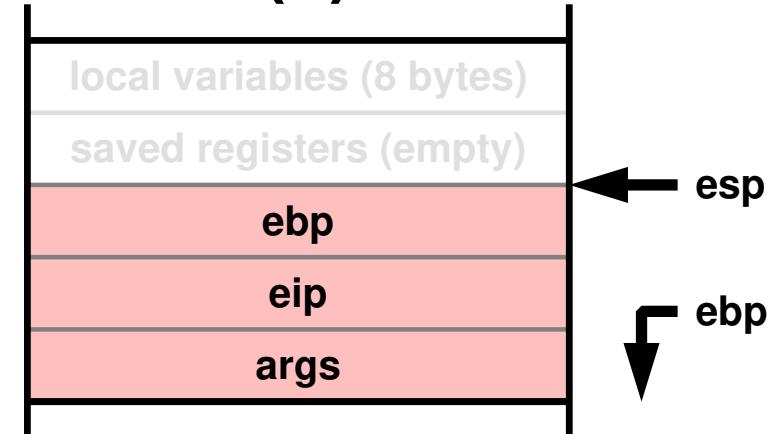
```

sub:
    pushl %ebp
    movl %esp, %ebp           } set up
    subl $8, %esp
    movl $1, -4(%ebp)
    movl $0, -8(%ebp)
    movl -4(%ebp), %ecx
    movl -8(%ebp), %eax

beginloop:
    cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop

endloop:
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp           } restore
    popl %ebp
    ret

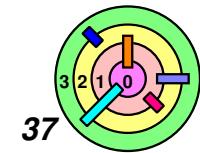
```



```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}

```



# Intel x86: Subroutine Code (2)

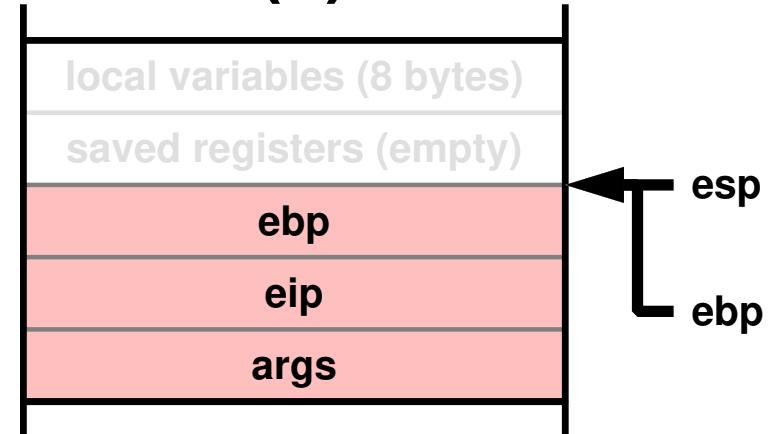
```

sub:
    pushl %ebp
    movl %esp, %ebp
    → subl $8, %esp
    movl $1, -4(%ebp)
    movl $0, -8(%ebp)
    movl -4(%ebp), %ecx
    movl -8(%ebp), %eax

beginloop:
    cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop

endloop:
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
  
```

} set up      } restore



```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
  
```

# Intel x86: Subroutine Code (2)

**sub:**

```

    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $1, -4(%ebp) } set up
    movl $0, -8(%ebp) } not local var
    movl -4(%ebp), %ecx
    movl -8(%ebp), %eax

```

**beginloop:**

```

    cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop

```

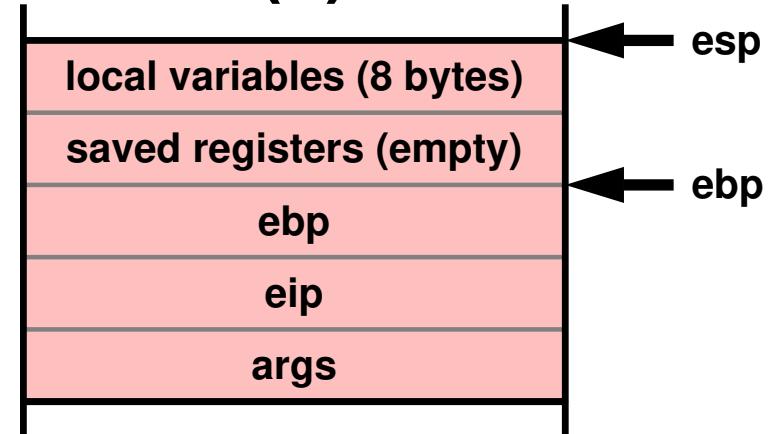
**endloop:**

```

    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret

```

} not local var  
initialization

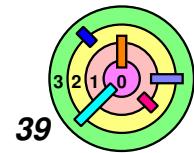


```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}

```

} restore



# Intel x86: Subroutine Code (2)

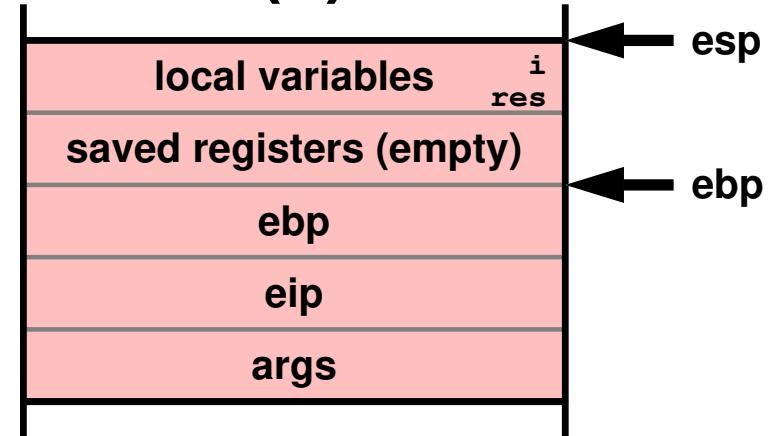
```

sub:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $1, -4(%ebp)
    → movl $0, -8(%ebp)
    movl -4(%ebp), %ecx
    movl -8(%ebp), %eax

beginloop:
    cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop

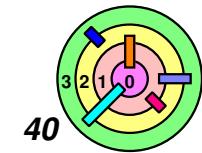
endloop:
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
  
```

} set up      } restore



```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
  
```

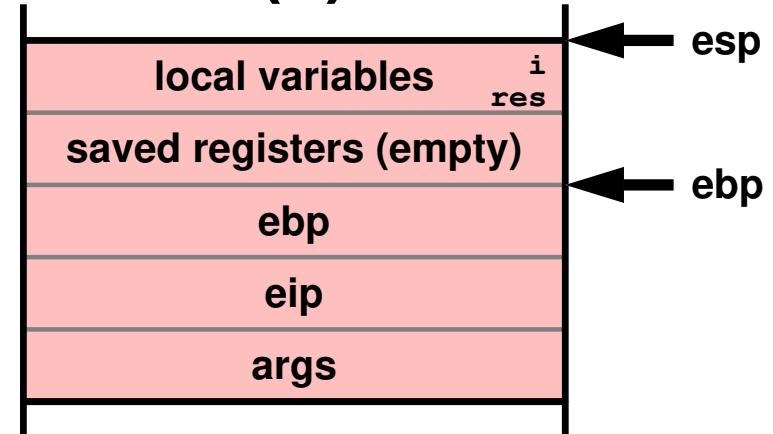


# Intel x86: Subroutine Code (2)

```

sub:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $1, -4(%ebp)
    movl $0, -8(%ebp)
    → movl -4(%ebp), %ecx
    movl -8(%ebp), %eax
beginloop:
    cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop
endloop:
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
  
```

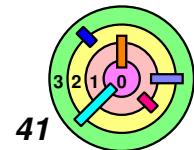
} set up



} restore

```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
  
```



# Intel x86: Subroutine Code (2)

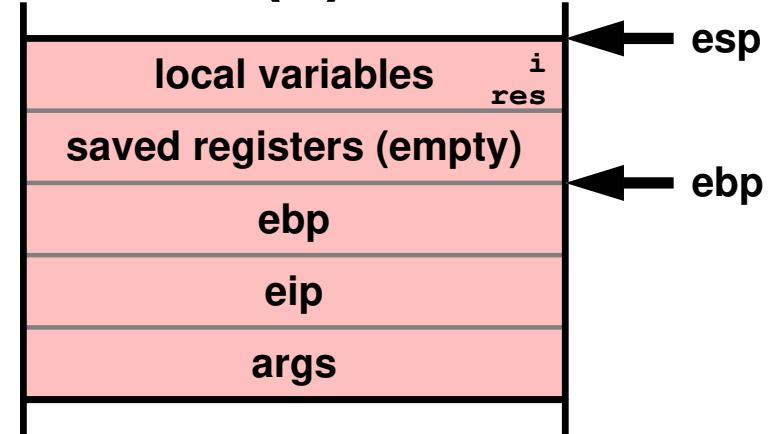
```

sub:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $1, -4(%ebp)
    movl $0, -8(%ebp)
    movl -4(%ebp), %ecx
    → movl -8(%ebp), %eax

beginloop:
    cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop

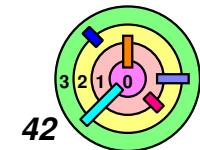
endloop:
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
  
```

} set up      } restore



```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
  
```



# Intel x86: Subroutine Code (2)

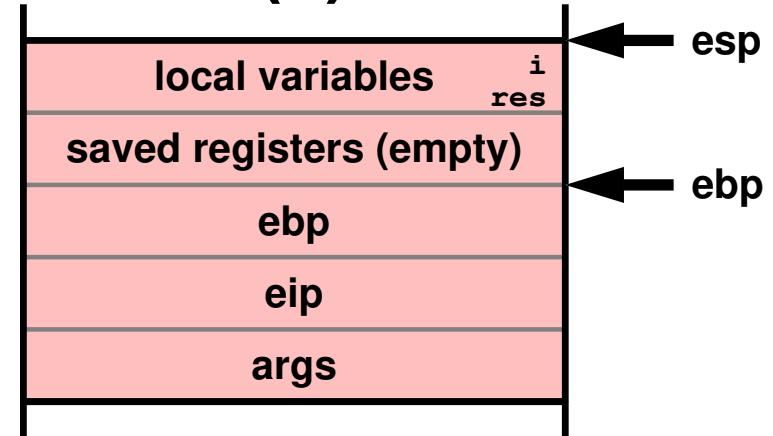
```

sub:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $1, -4(%ebp)
    movl $0, -8(%ebp)
    movl -4(%ebp), %ecx
    movl -8(%ebp), %eax

beginloop:
    → cmpl 12(%ebp), %eax
    jge endloop
    imull 8(%ebp), %ecx
    addl $1, %eax
    jmp beginloop

endloop:
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
  
```

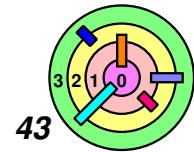
} set up



```

int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
  
```

} restore



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

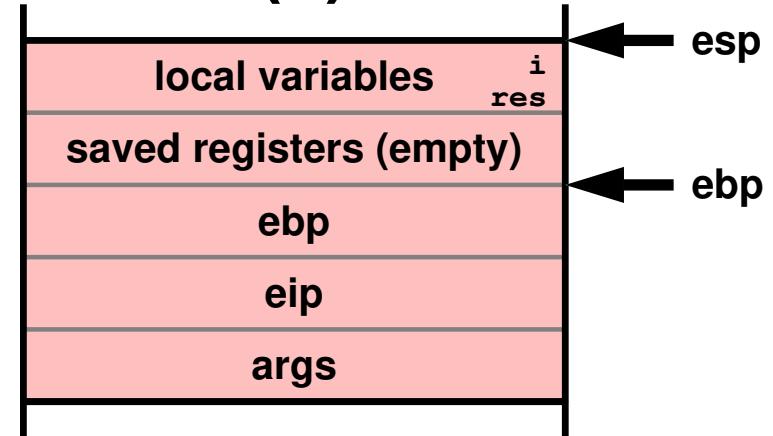
**beginloop:**

```
cmpl 12(%ebp), %eax
→ jge endloop
imull 8(%ebp), %ecx
addl $1, %eax
jmp beginloop
```

**endloop:**

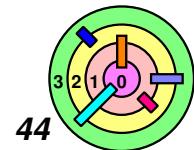
```
movl %ecx, -4(%ebp)
movl -4(%ebp), %eax
movl %ebp, %esp
popl %ebp
ret
```

} set up



```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```

} restore



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

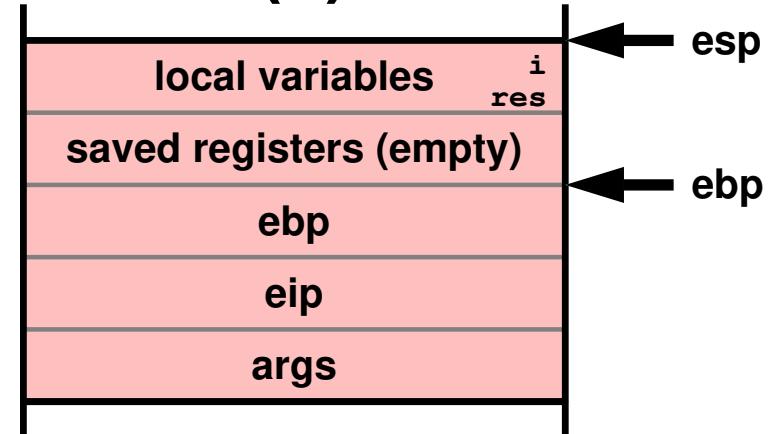
**beginloop:**

```
cmpl 12(%ebp), %eax
jge endloop
→ imull 8(%ebp), %ecx
addl $1, %eax
jmp beginloop
```

**endloop:**

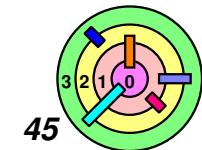
```
movl %ecx, -4(%ebp)
movl -4(%ebp), %eax
movl %ebp, %esp
popl %ebp
ret
```

} set up



} restore

```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

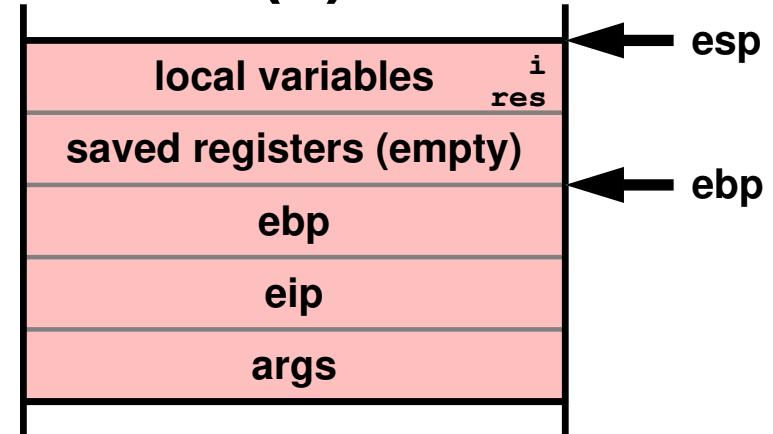
**beginloop:**

```
cmpl 12(%ebp), %eax
jge endloop
imull 8(%ebp), %ecx
→ addl $1, %eax
jmp beginloop
```

**endloop:**

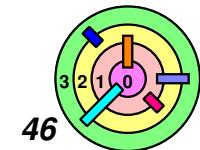
```
movl %ecx, -4(%ebp)
movl -4(%ebp), %eax
movl %ebp, %esp
popl %ebp
ret
```

} set up



} restore

```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

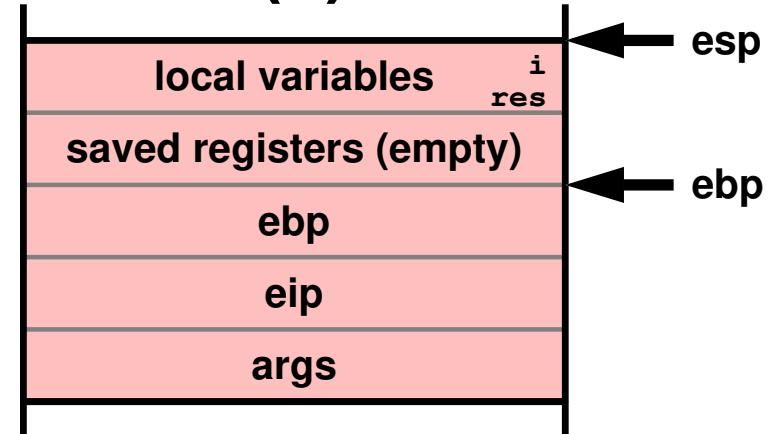
**beginloop:**

```
cmpl 12(%ebp), %eax
jge endloop
imull 8(%ebp), %ecx
addl $1, %eax
jmp beginloop
```

**endloop:**

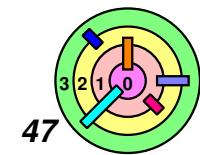
```
movl %ecx, -4(%ebp)
movl -4(%ebp), %eax
movl %ebp, %esp
popl %ebp
ret
```

} set up



```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```

} restore



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

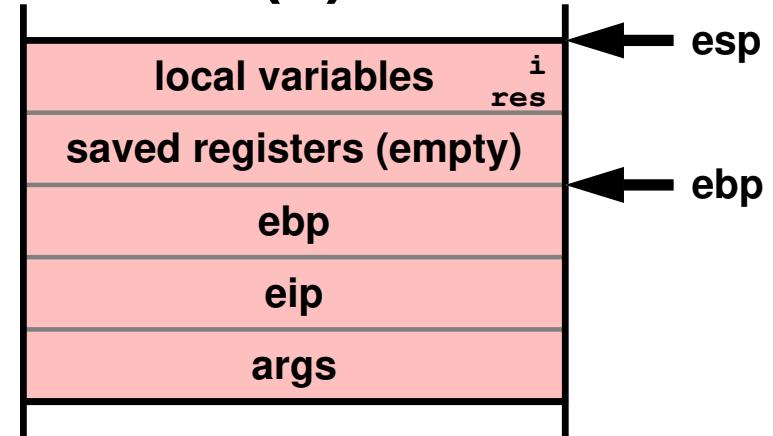
**beginloop:**

```
cmpl 12(%ebp), %eax
jge endloop
imull 8(%ebp), %ecx
addl $1, %eax
jmp beginloop
```

**endloop:**

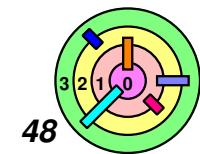
```
→ movl %ecx, -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
```

} set up



} restore

```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

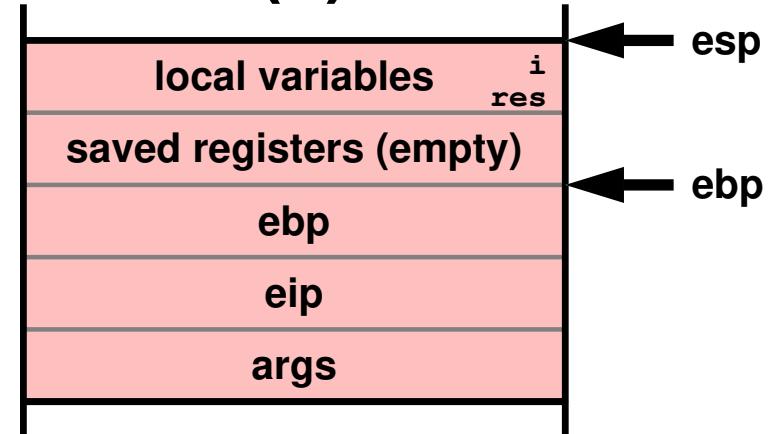
**beginloop:**

```
cmpl 12(%ebp), %eax
jge endloop
imull 8(%ebp), %ecx
addl $1, %eax
jmp beginloop
```

**endloop:**

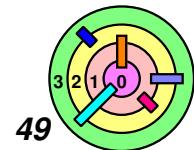
```
movl %ecx, -4(%ebp)
→ movl -4(%ebp), %eax
movl %ebp, %esp
popl %ebp
ret
```

} set up



} restore

```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

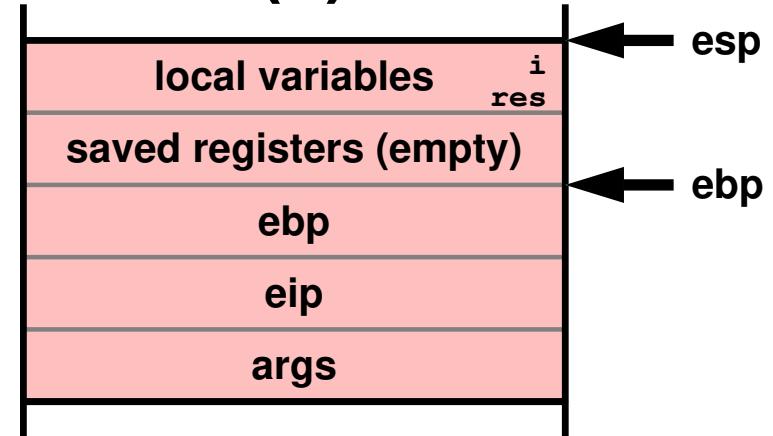
**beginloop:**

```
cmpl 12(%ebp), %eax
jge endloop
imull 8(%ebp), %ecx
addl $1, %eax
jmp beginloop
```

**endloop:**

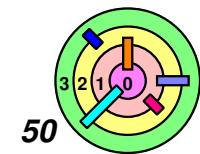
```
movl %ecx, -4(%ebp)
movl -4(%ebp), %eax
→ movl %ebp, %esp
popl %ebp
ret
```

} set up



```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```

} restore



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

} set up

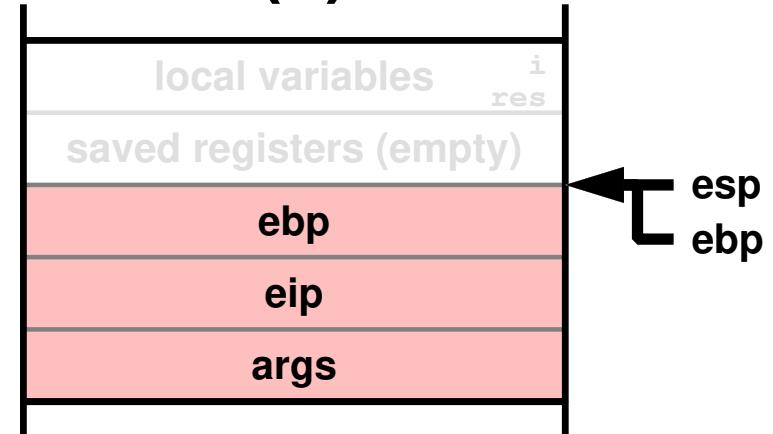
**beginloop:**

```
cmpl 12(%ebp), %eax
jge endloop
imull 8(%ebp), %ecx
addl $1, %eax
jmp beginloop
```

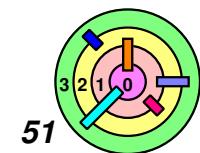
**endloop:**

```
movl %ecx, -4(%ebp)
movl -4(%ebp), %eax
movl %ebp, %esp
popl %ebp
ret
```

} restore



```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```



# Intel x86: Subroutine Code (2)

**sub:**

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $1, -4(%ebp)
movl $0, -8(%ebp)
movl -4(%ebp), %ecx
movl -8(%ebp), %eax
```

} set up

**beginloop:**

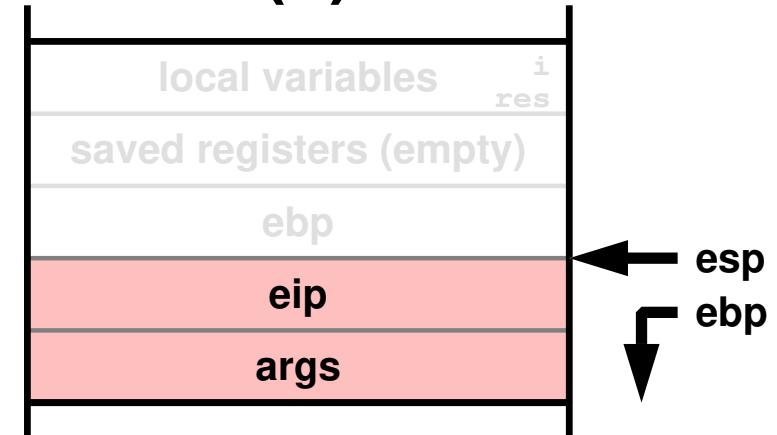
```
cmpl 12(%ebp), %eax
jge endloop
imull 8(%ebp), %ecx
addl $1, %eax
jmp beginloop
```

**endloop:**

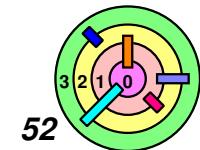
```
movl %ecx, -4(%ebp)
movl -4(%ebp), %eax
movl %ebp, %esp
popl %ebp
```

} restore

→ **ret**



```
int sub(int x, int y) {
    // computes x^y
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```

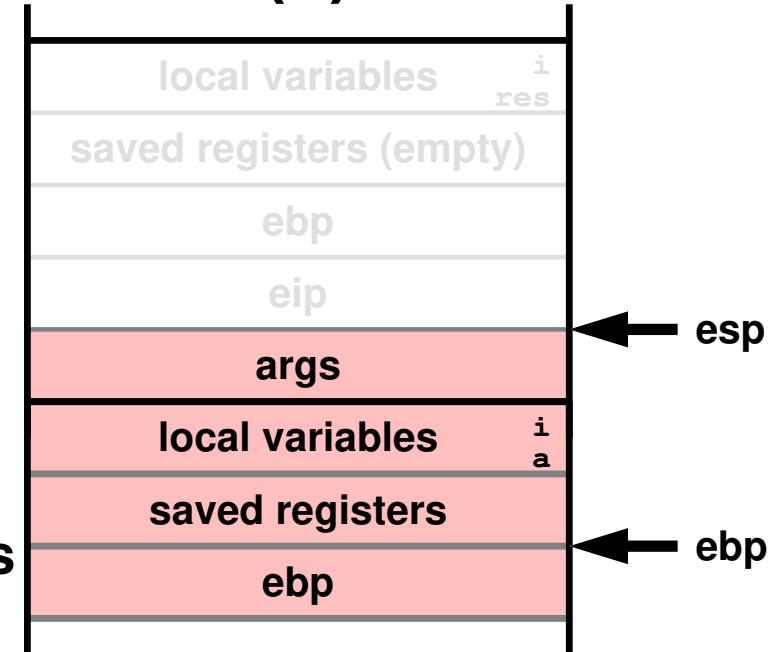


# Intel x86: Subroutine Code (1)

```

main:
  pushl %ebp
  movl %esp, %ebp
  pushl %esi
  pushl %edi
  subl $8, %esp
  ...
  pushl $1
  movl -12(%ebp), %eax
  pushl %eax
  call sub
  → addl $8, %esp
  movl %eax, -16(%ebp)
  ...
  addl $8, %esp
  movl $0, %eax
  popl %edi
  popl %esi
  movl %ebp, %esp
  popl %ebp
  ret
  }
```

set up stack frame      push args      pop args; get result      set return value and restore frame



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}
```



# SPARC Architecture

|                |
|----------------|
| return address |
| frame pointer  |
|                |
|                |
|                |
|                |
|                |
|                |

Input Registers

i7 r31  
 i6 r30  
 i5 r29  
 i4 r28  
 i3 r27  
 i2 r26  
 i1 r25  
 i0 r24

|               |
|---------------|
| stack pointer |
|               |
|               |
|               |
|               |
|               |
|               |
|               |

Output Registers

o7 r15  
 o6 r14  
 o5 r13  
 o4 r12  
 o3 r11  
 o2 r10  
 o1 r9  
 o0 r8

|  |
|--|
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

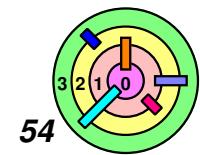
Local Registers

l7 r23  
 l6 r22  
 l5 r21  
 l4 r20  
 l3 r19  
 l2 r18  
 l1 r17  
 l0 r16

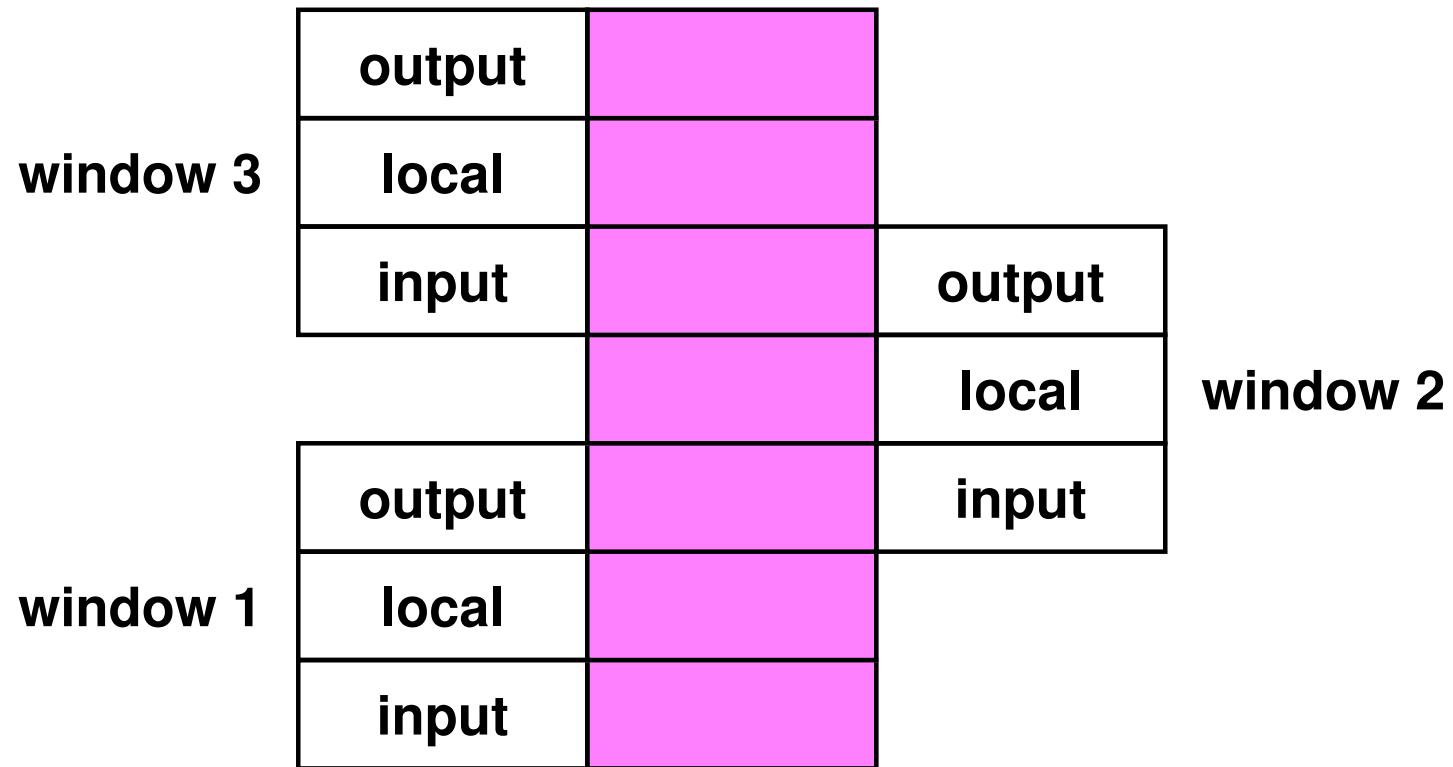
|   |
|---|
| 0 |
|   |
|   |
|   |
|   |
|   |
|   |
|   |

Global Registers

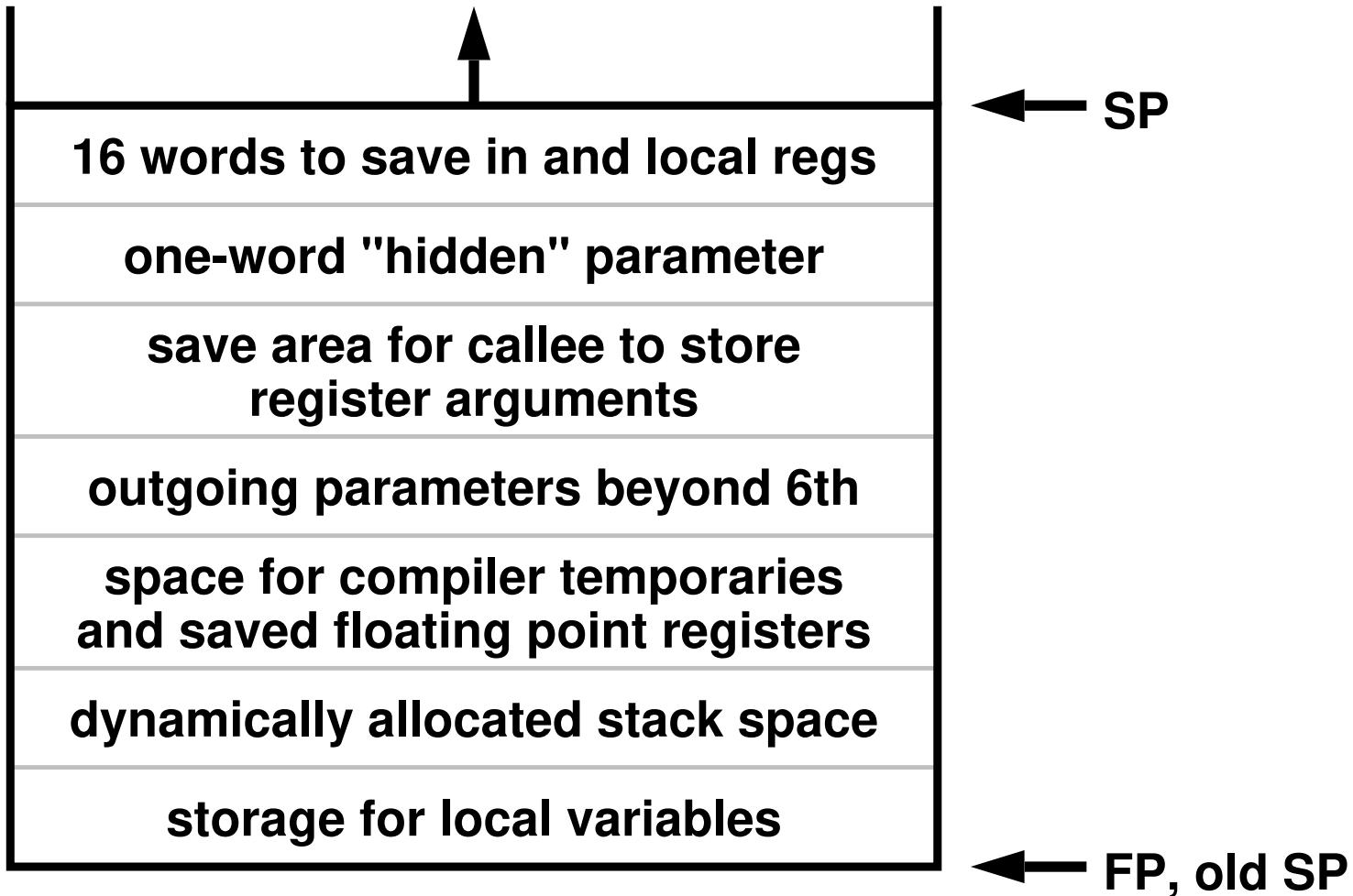
g7 r7  
 g6 r6  
 g5 r5  
 g4 r4  
 g3 r3  
 g2 r2  
 g1 r1  
 g0 r0



# SPARC Architecture: Register Windows



# SPARC Architecture: Stack



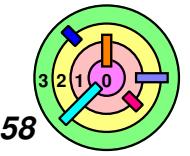
# SPARC Architecture: Subroutine Code

```
ld [%fp-8], %o0
! put local var (a) into out register
mov 1, %o1
! deal with 2nd parameter
call sub
nop
st %o0, [%fp-4]
! store result into local var (i)
...
sub:
save %sp, -64, %sp
! push a new stack frame
add %i0, %i1, %i0
! compute sum
ret
! return to caller
restore
! pop frame off stack (in delay slot)
```



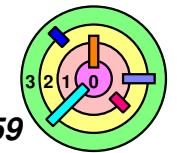
# 3.1 Context Switching

- ➔ Procedures
- ➔ *Threads & Coroutines*
- ➔ Systems Calls
- ➔ Interrupts



# Threads & Coroutines

- Normally, threads are independent of one another and don't directly control one another's execution
  - threads can be made aware of each other and be able to *transfer control* from one thread to another
    - this is known as *coroutine linkage*

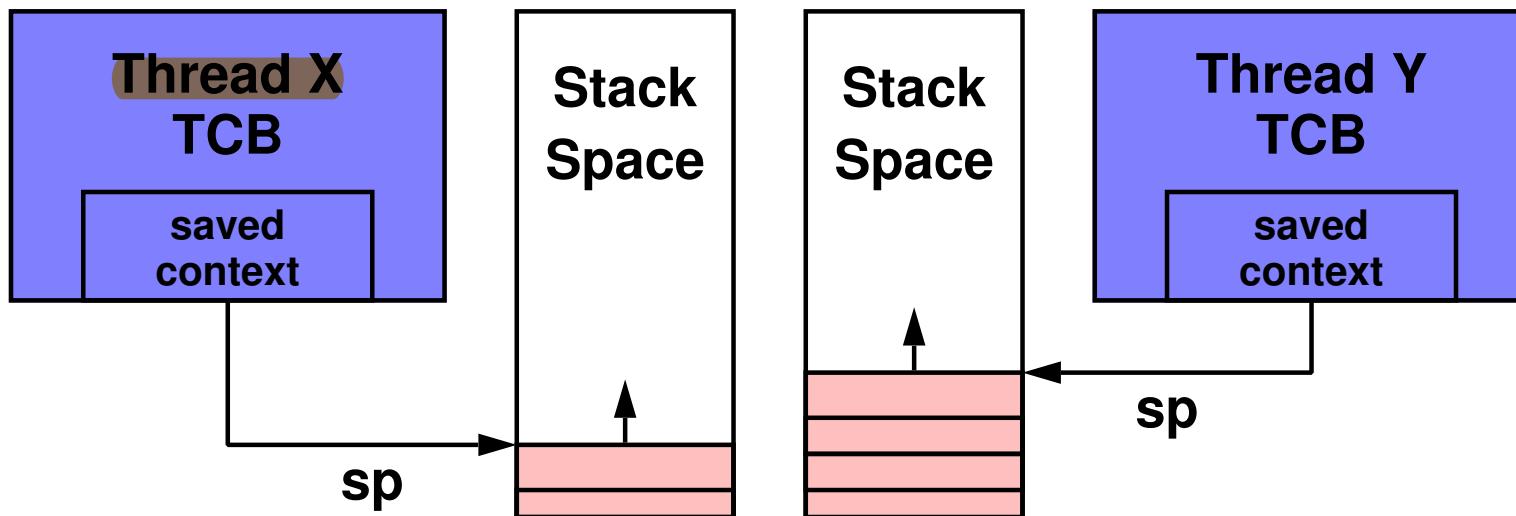


F20-Q14

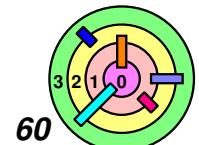
# Representing Threads

## → A thread's context

- its **stack**, its **register state**
  - can be stored in a **Thread Control Block (TCB)**
  - this is what **sleeping** threads look like

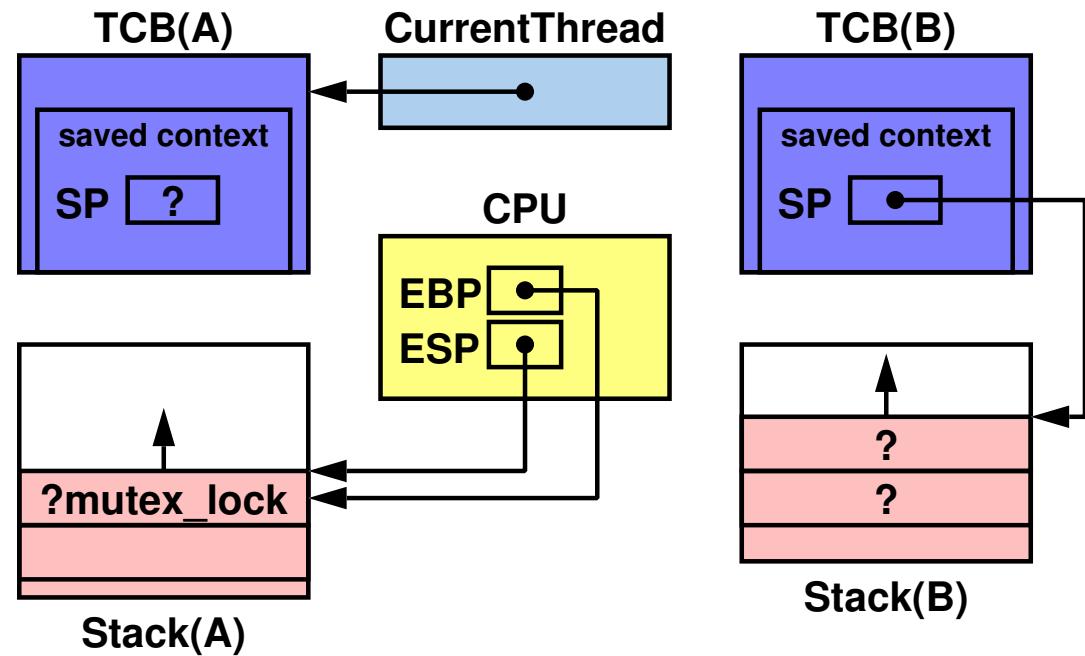


- to transfer control from one thread to another is equivalent to copying the thread control block of the target thread into the "**Current Thread context**"
- we will look at the single CPU case

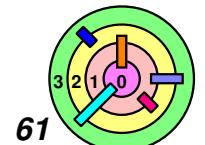


# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```



- thread A calls `switch()` to switch to thread B
- thread B's TCB has the *exact context* of thread B right before thread B was *last suspended*
- context information in thread A's TCB is *out-dated*

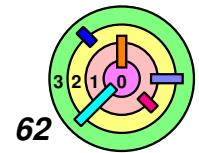


F20-Q1

# Switching Between Threads

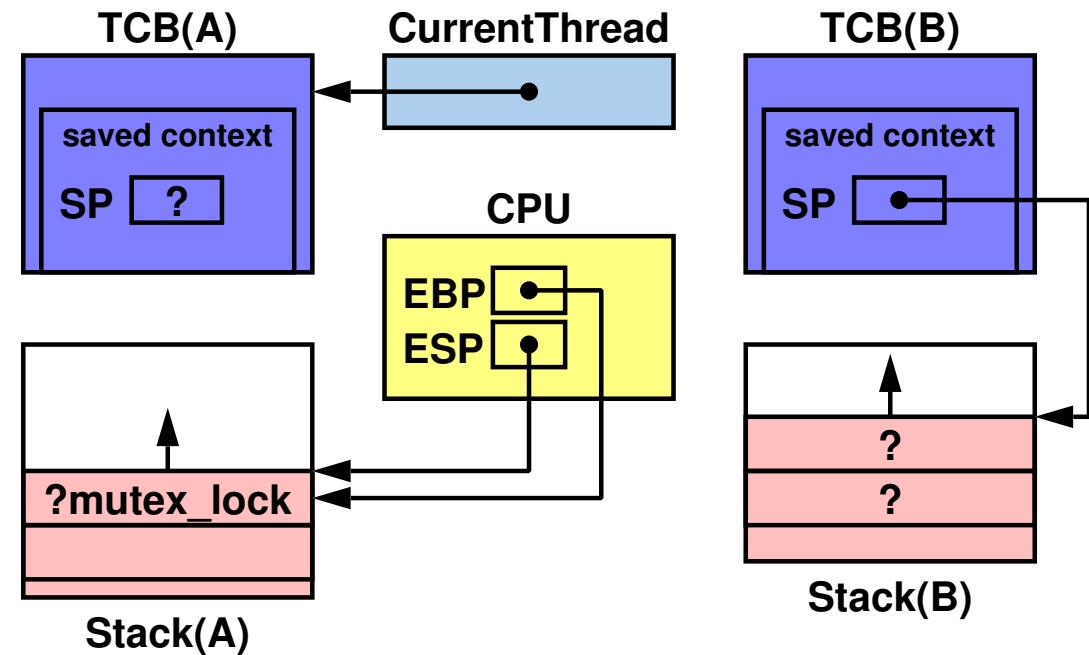
```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```

```
switch:
    ;enter switch, creating new stack frame
    pushl %ebp ;push FP
    movl %esp,%ebp ;set FP to point to new frame
    pushl %esi ;save esi register
    movl CurrentThread,%esi ;load address of caller's TCB
    movl %esp,SP(%esi) ;save SP in control block
    movl 8(%ebp),CurrentThread ;store target TCB address
                                ;into CurrentThread
    movl CurrentThread,%esi ;put new TCB address into esi
    movl SP(%esi),%esp ;restore target thread's SP
    ;we're now in the context of the target thread!
    popl %esi ;restore target thread's esi register
    popl %ebp ;pop target thread's FP
    ret ;return to caller within target thread
```

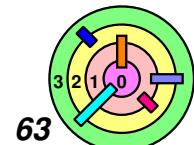


# Switching Between Threads

```
→ void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```



- why is `switch()` called?
  - may be because thread A called `pthread_mutex_lock()` and the mutex is not available

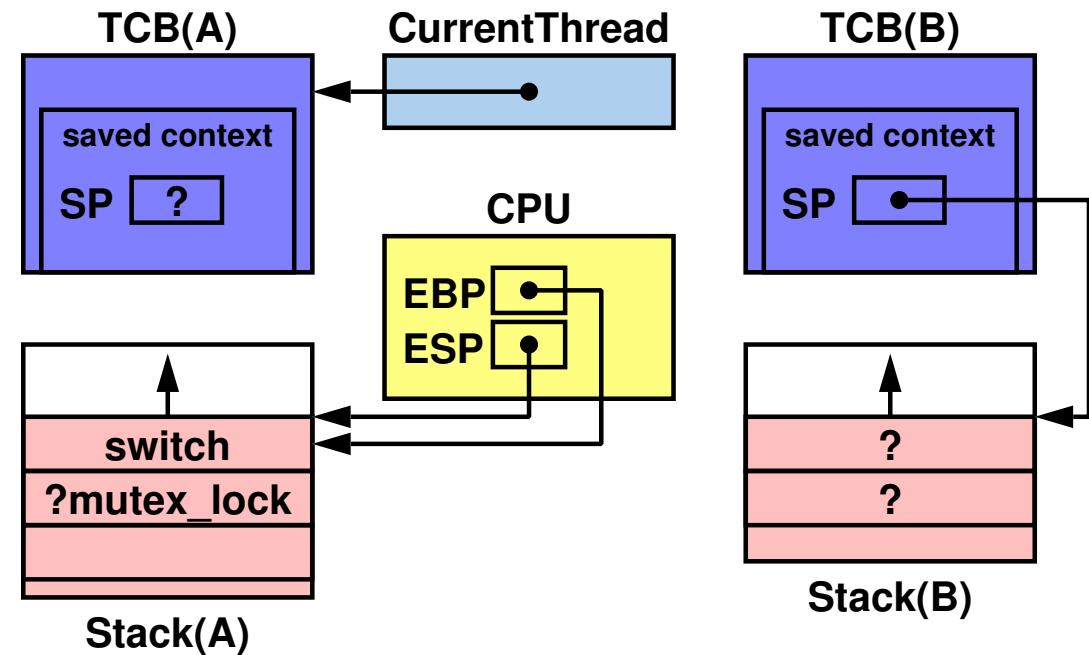


# Switching Between Threads

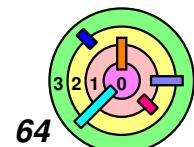
```

→ void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}

```

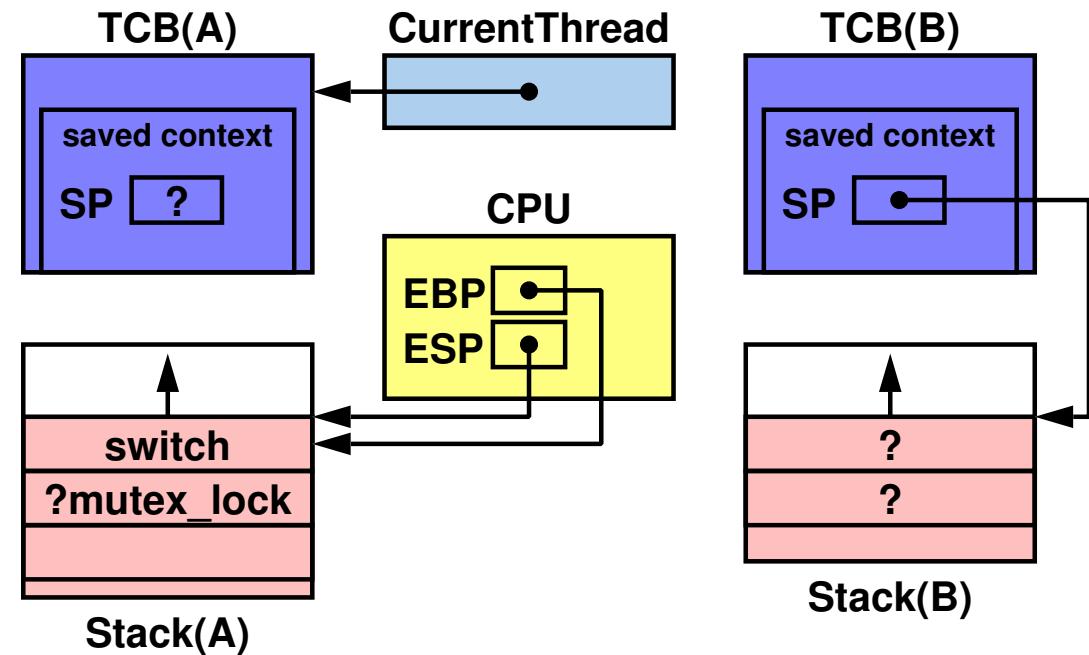


- on entry into **switch()**, the caller's registers are saved!

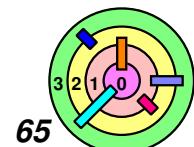


# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```

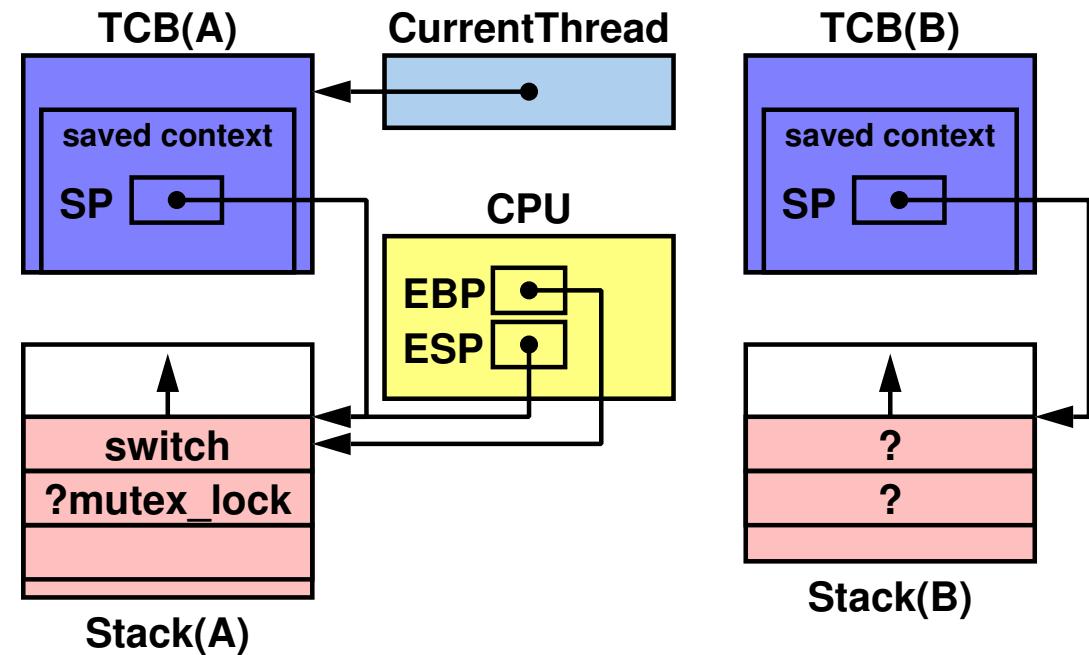


- then the current stack pointer is saved into current thread's thread control block



# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```



- then the current stack pointer is saved into current thread's thread control block

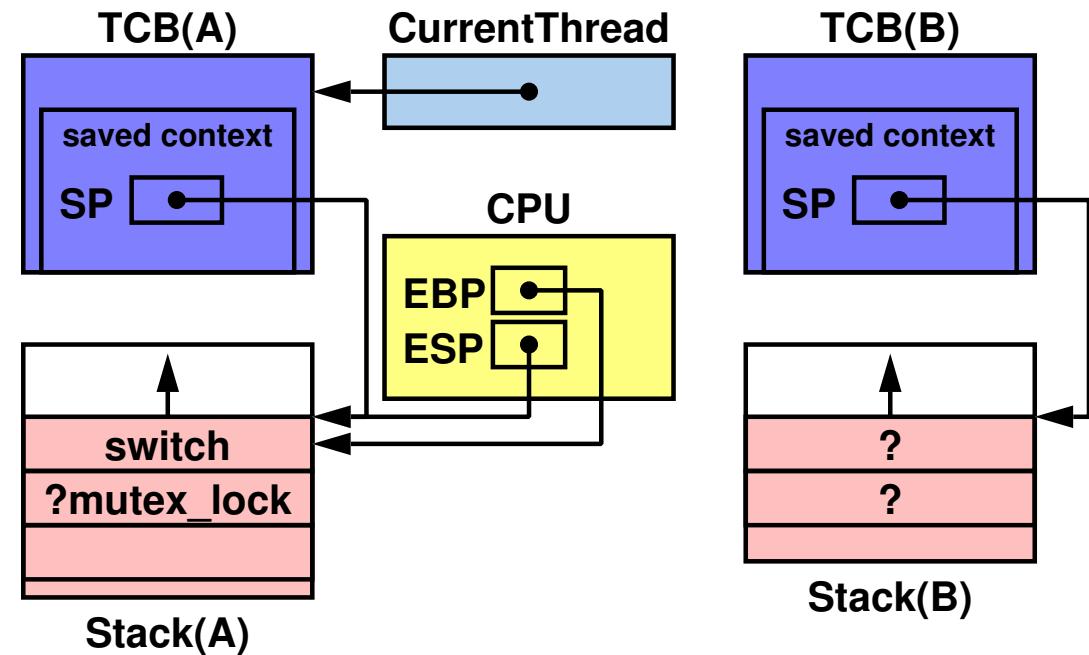


# Switching Between Threads

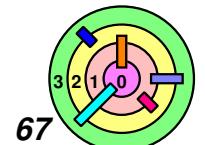
```

void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    → CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}

```

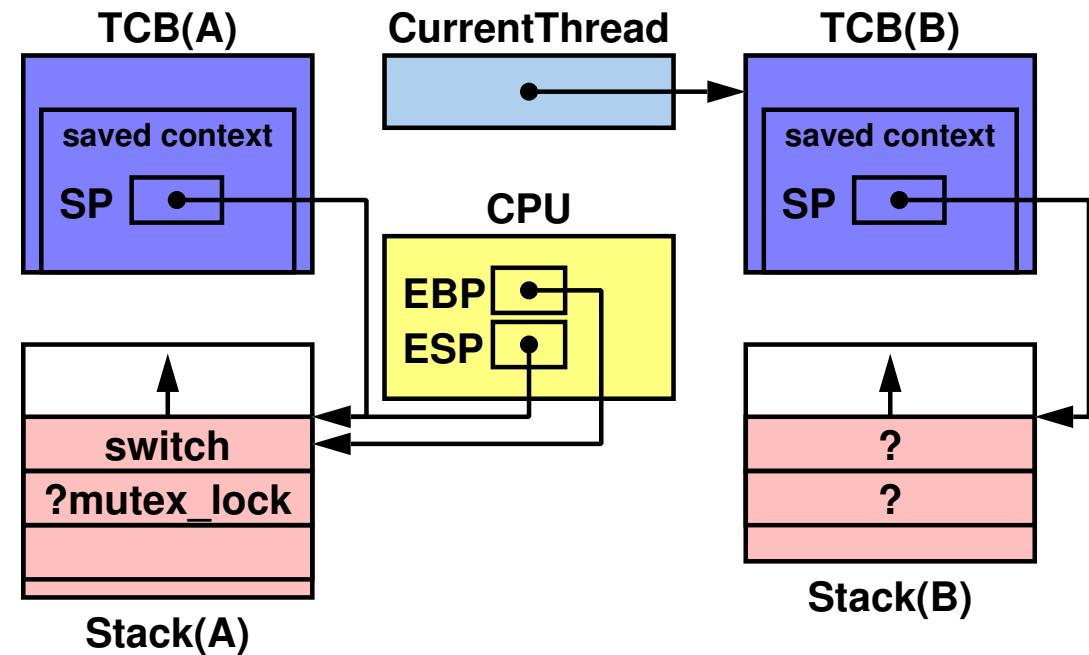


- the target thread becomes the current thread

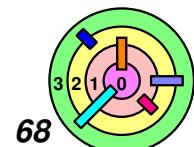


# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    → CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```



- the target thread becomes the current thread

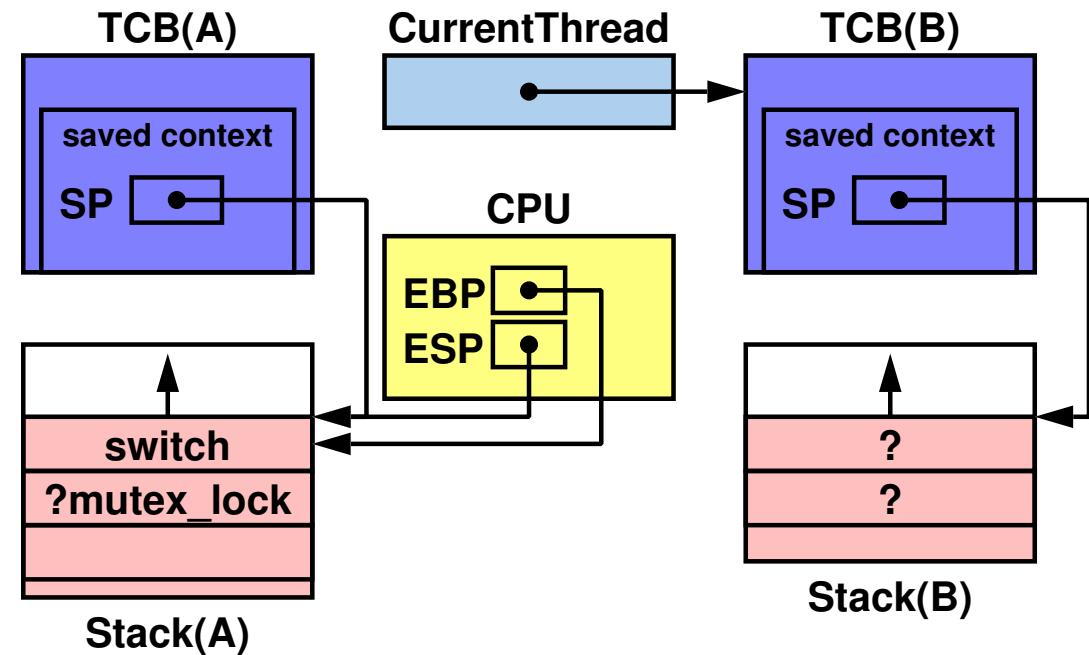


# Switching Between Threads

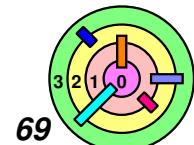
```

void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
→ SP = CurrentThread->SP;
return;
}

```

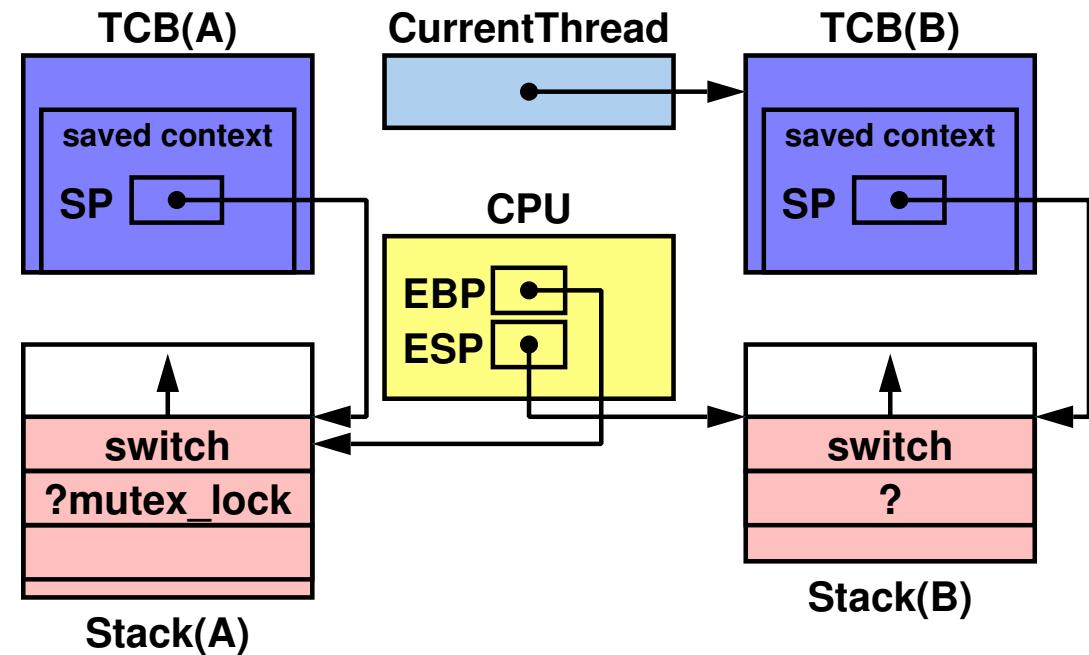


- fetch the target thread's stack pointer (esp for x86) from its thread control block and loads it into the actual stack pointer



# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    → SP = CurrentThread->SP;
    return;
}
```

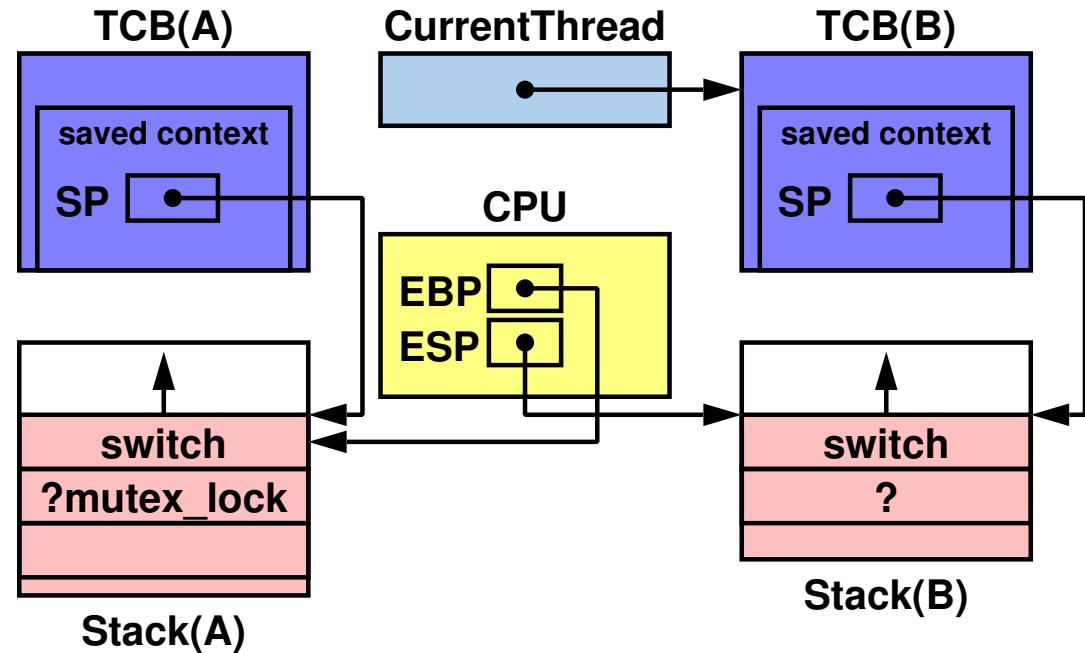


- fetch the target thread's stack pointer (esp for x86) from its thread control block and loads it into the actual stack pointer
  - hmm... which thread executes this?
    - ◊ does it matter? what about EIP?

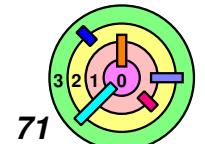


# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    → return;
}
```

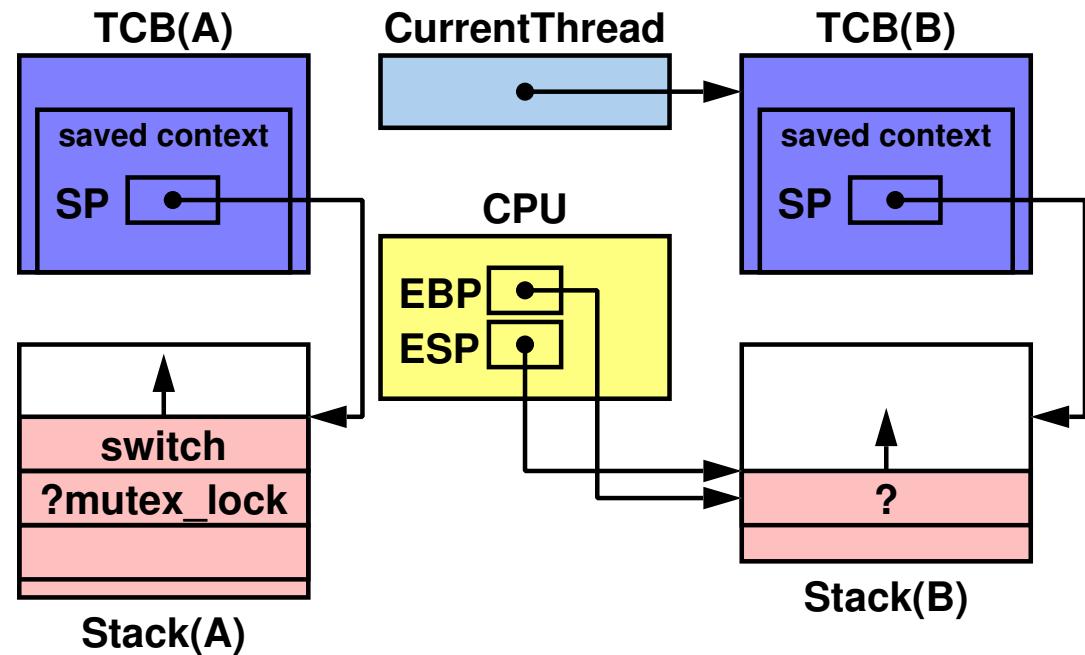


- on return from `switch()`, the registers (ebp and eip for x86) are restored into the current thread, which is the target thread!
  - which thread executes `return`?
  - ◊ does it matter? what about EIP?

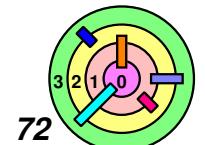


# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```

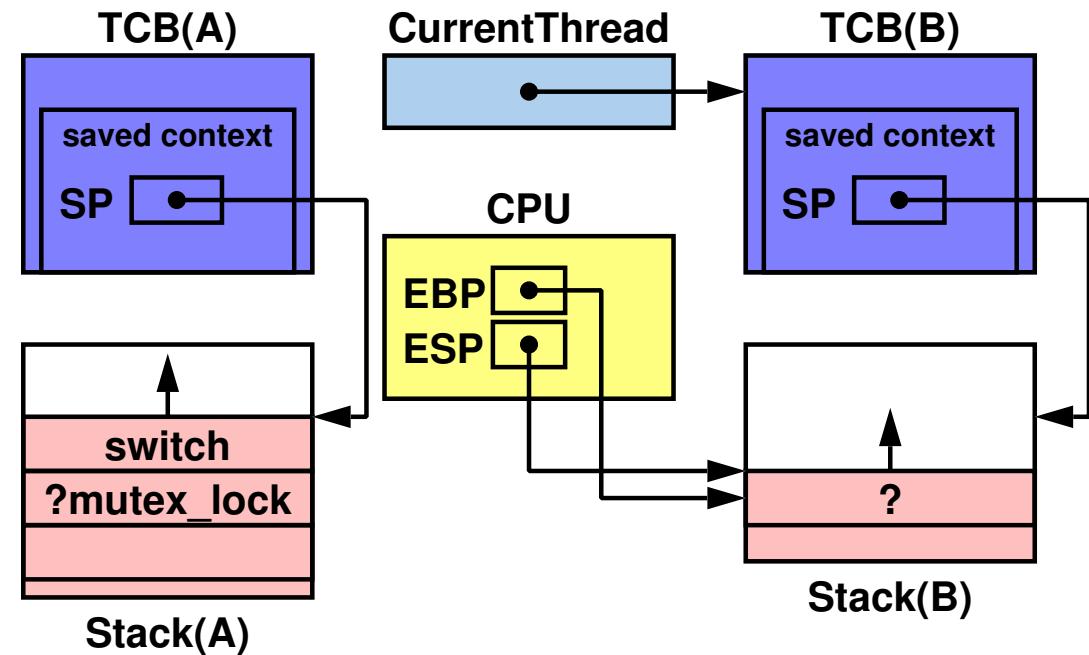


- on return from `switch()`, the registers (ebp and eip for x86) are restored into the current thread, which is the target thread!
  - which thread executes `return`?
  - ◊ does it matter? what about EIP?



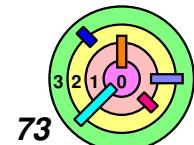
# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```



- if thread control blocks were user-space data structures, threads were switched *without* getting the kernel involved!

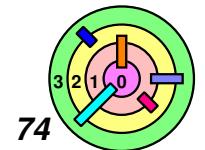
→ Note: SP field inside TCB(B) *no longer* tracks ESP in CPU



# Switching Between Threads

```
void switch(thread_t *next_thread) {  
    CurrentThread->SP = SP;  
    CurrentThread = next_thread;  
    SP = CurrentThread->SP;  
    return;  
}
```

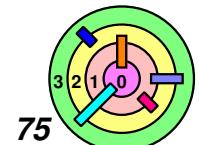
- Note: one very interesting thing happened in this call
- usually, a single thread executes the entire procedure call
  - with `switch()`, at the beginning of the procedure call, one thread is executing
    - half way through the procedure call, another thread starts to execute
    - so, one thread enters the `switch()` call, and *a different thread* leaves the `switch()` call!
    - but from whose perspective?!



# Switching Between Threads

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```

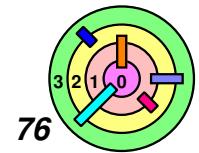
- With `switch()`, one thread enters the `switch()` call, and *a different thread* leaves the `switch()` call!
  - that's from the *system's perspective*
  - from the *calling thread's perspective*, it fell asleep in `switch()`
  - from the *next\_thread's perspective*, it woke up in `switch()`
    - it fell asleep a while back and now woke up somehow
  - from the *CPU's perspective*
    - "I did what?! What's a thread?"
    - *thread* is just an *abstraction*, CPU doesn't really know about threads



# Switching Between Threads

```
void switch(thread_t *next_thread) {  
    CurrentThread->SP = SP;  
    CurrentThread = next_thread;  
    SP = CurrentThread->SP;  
    return;  
}
```

- This is an elegant way of switching threads
  - all threads come here to switch to another thread
- This code is incomplete
  - what if `next_thread` is NULL?
  - it saves **some** of the CPU registers, but we need to save **all** of the CPU registers
  - will revisit in Ch 5



# ... in x86 Assembler

**switch:**

```
; enter switch, creating new thread
pushl %ebp ;push FP
movl %esp,%ebp ;set FP to stack
pushl %esi ;save esi register
movl CurrentThread,%esi ;
movl %esp,SP(%esi) ;save old stack pointer
movl 8(%ebp),CurrentThread ;store target TCB address
                           ;into CurrentThread
movl CurrentThread,%esi ;put new TCB address into esi
movl SP(%esi),%esp ;restore target thread's SP
;we're now in the context of the target thread!
popl %esi ;restore target thread's esi register
popl %ebp ;pop target thread's FP
ret ;return to caller within target thread
```

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```

- The above is *hand-written* and not compiler generated
  - can easily change it to save more registers



# ... in SPARC Assembler

**switch:**

```

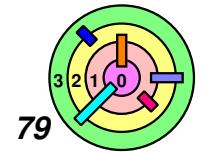
save %sp, -64, %sp      ! Push a new stack frame.
t    3                      ! Trap into the OS to force
                               ! window overflow.
st    %sp, [%g0+SP]       ! Save CurrentThread's SP in
                               ! control block.
mov   %i0, %g0            ! Set CurrentThread to be
                               ! target thread.
ld    [%g0+SP], %sp       ! Set SP to that of target thread
ret                         ! return to caller (in target
                               ! thread's context).
restore                   ! Pop frame off stack (in delay
                               ! slot).

```



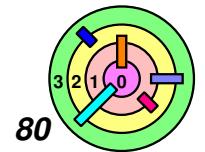
# 3.1 Context Switching

- ➔ Procedures
- ➔ Threads & Coroutines
- ➔ *Systems Calls*
- ➔ Interrupts



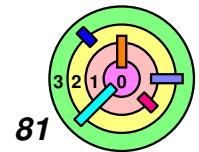
# System Calls

- ➡ A system call involves the transfer of control from user code to system/kernel code and back
  - there is *no thread switching!*
    - although there was a *context switch*
    - depending on the OS implementation (and as far as this class is concerned), this can view this as a user thread *change status* and becomes a kernel thread
      - and executes in *privileged mode*
      - and executing operating-system code
        - ◆ effectively, it's part of the OS
      - in reality, more complex than just changing status
    - it then changed back to a user thread eventually



# System Calls

- Most systems provide threads with *two stacks*
  - one for use in user mode
  - and one for use in kernel mode (as far as this class is concerned)
    - in some systems (not common these days), one kernel stack is shared by all threads in the same user process (Ch 5)
  - therefore, when a thread performs a system call and switches from user mode to kernel mode
    - it switches to use its kernel-mode stack
    - the kernel cannot use the user-space stack because it *cannot trust* the user process
      - ◊ what if the user process has almost used up the stack space?

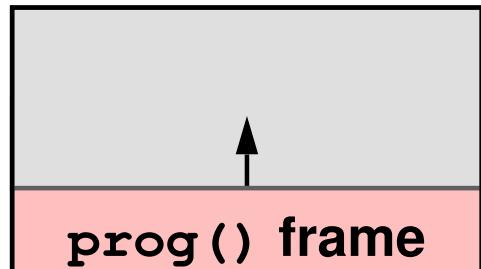


# System Calls

- A **trap** is a type of "**software interrupt**"  
 — interrupt handler will invoke trap handler

```
prog( ) {
  ...
  → write(fd, buffer, size);
  ...
}
```

```
write( ) {
  ...
  trap(write_code);
  ...
}
```



User Stack

User

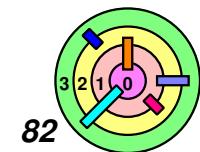
Kernel

```
intr_handler(intr_code) {
  ...
  if (intr_code == SYSCALL)
    syscall_handler( );
  ...
}

syscall_handler(trap_code) {
  ...
  if (trap_code == write_code)
    write_handler( );
  ...
}
```



Kernel Stack

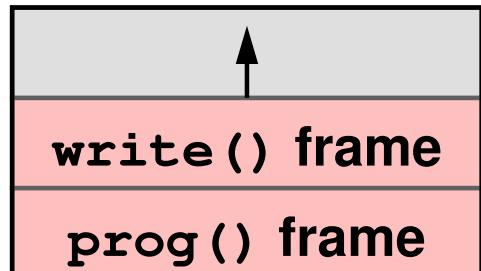


# System Calls

- A **trap** is a type of "**software interrupt**"  
 — interrupt handler will invoke trap handler

```
prog( ) {
    ...
    write(fd, buffer, size); → trap(write_code);
    ...
}
```

```
write( ) {
    ...
}
```



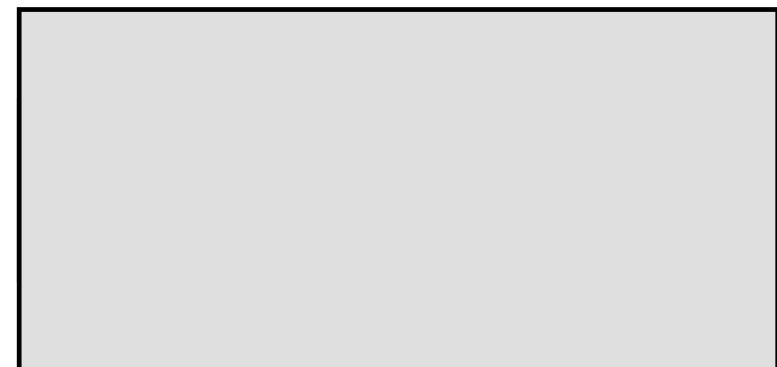
User Stack

User

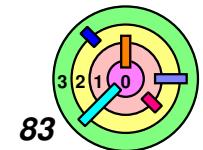
Kernel

```
intr_handler(intr_code) {
    ...
    if (intr_code == SYSCALL)
        syscall_handler( );
    ...
}

syscall_handler(trap_code) {
    ...
    if (trap_code == write_code)
        write_handler( );
    ...
}
```



Kernel Stack



# System Calls

- A **trap** is a type of "**software interrupt**"  
 — interrupt handler will invoke trap handler

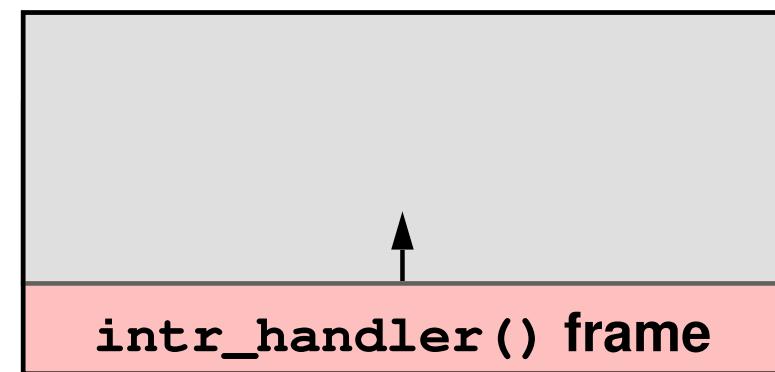
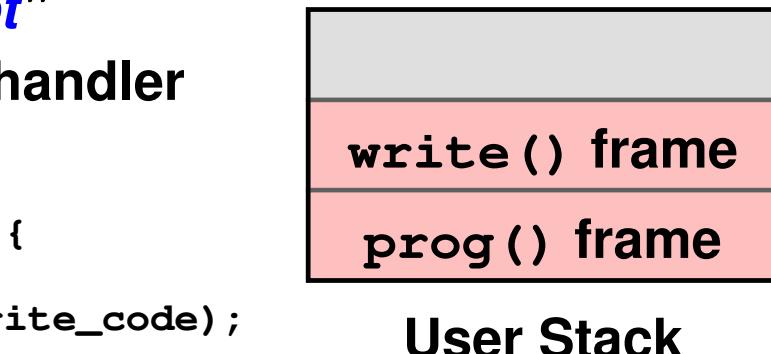
```
prog( ) {
    ...
    write(fd, buffer, size);
    ...
}
```

```
        write( ) {
            ...
            trap(write_code);
            ...
}
```

User  
Kernel

```
intr_handler(intr_code) {
    ...
    if (intr_code == SYSCALL)
        → syscall_handler( );
    ...
}

syscall_handler(trap_code) {
    ...
    if (trap_code == write_code)
        write_handler( );
    ...
}
```

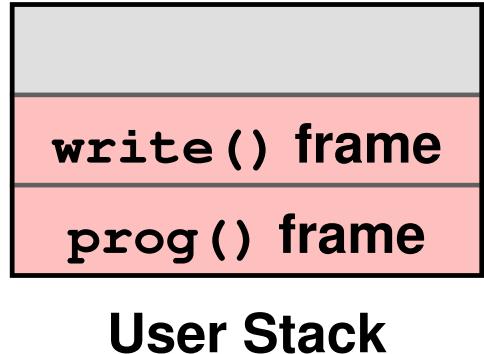


# System Calls

- A **trap** is a type of "**software interrupt**"  
 — interrupt handler will invoke trap handler

```
prog( ) {
    ...
    write(fd, buffer, size);
    ...
}
```

```
        write( ) {
            ...
            trap(write_code);
            ...
}
```

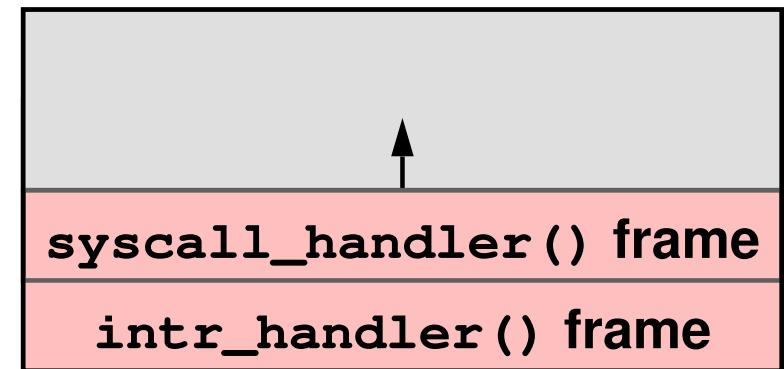


User

Kernel

```
intr_handler(intr_code) {
    ...
    if (intr_code == SYSCALL)
        syscall_handler( );
    ...
}

syscall_handler(trap_code) {
    ...
    if (trap_code == write_code)
        write_handler( );
    ...
}
```



Kernel Stack

# System Calls

- A **trap** is a type of "**software interrupt**"  
 — interrupt handler will invoke trap handler

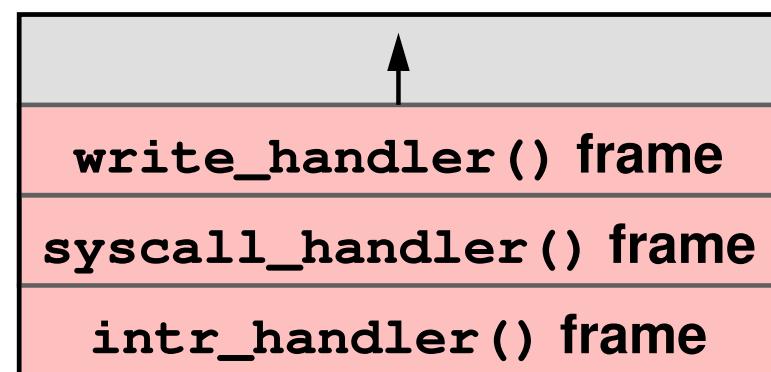
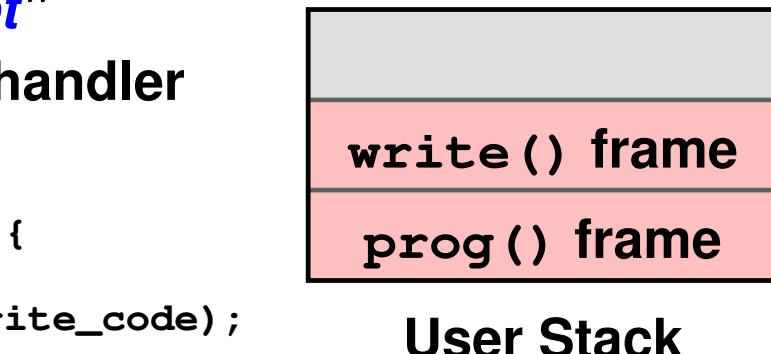
```
prog( ) {
    ...
    write(fd, buffer, size);
    ...
}
```

```
        write( ) {
            ...
            trap(write_code);
            ...
}
```

User  
Kernel

```
intr_handler(intr_code) {
    ...
    if (intr_code == SYSCALL)
        syscall_handler( );
    ...
}

syscall_handler(trap_code) {
    ...
    if (trap_code == write_code)
        write_handler( );
    ...
}
```



# System Calls

- A **trap** is a type of "**software interrupt**"  
 — interrupt handler will invoke trap handler

```
prog( ) {
    ...
    write(fd, buffer, size);
    ...
}
```

```
        write( ) {
            ...
            trap(write_code);
            ...
}
```



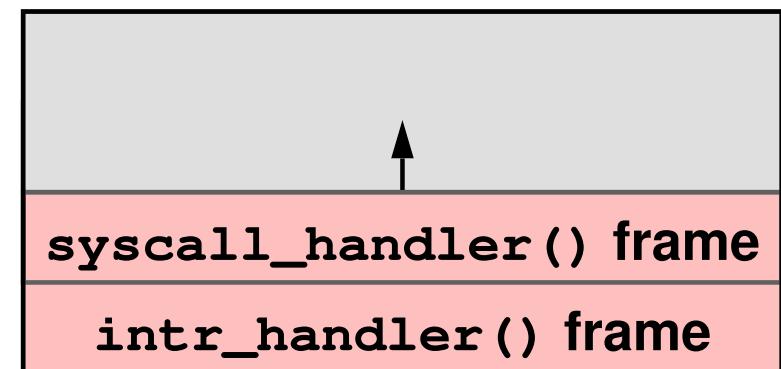
User Stack

User

Kernel

```
intr_handler(intr_code) {
    ...
    if (intr_code == SYSCALL)
        syscall_handler( );
    ...
}

syscall_handler(trap_code) {
    ...
    if (trap_code == write_code)
        write_handler( );
    ...
}
```



Kernel Stack

# System Calls

- A **trap** is a type of "**software interrupt**"
- interrupt handler will invoke trap handler

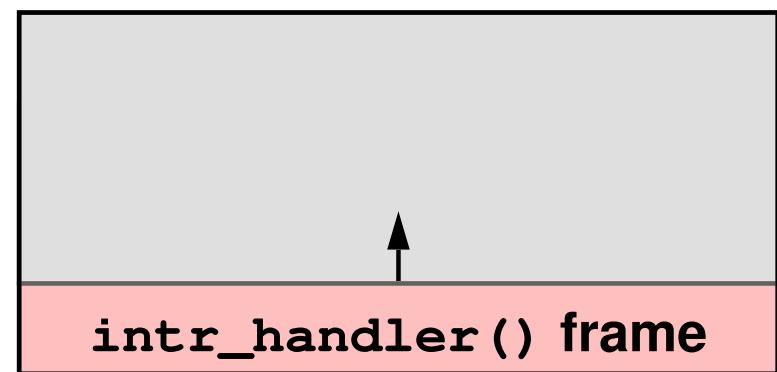
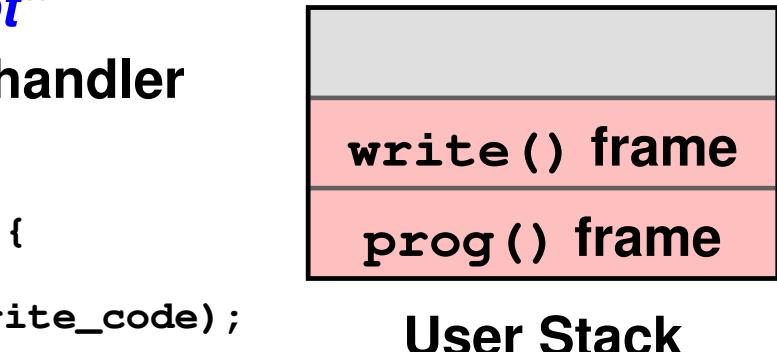
```
prog( ) {
    ...
    write(fd, buffer, size);
    ...
}
```

```
        write( ) {
            ...
            trap(write_code);
            ...
}
```

User  
Kernel

```
intr_handler(intr_code) {
    ...
    if (intr_code == SYSCALL)
        syscall_handler( );
    ...
}

syscall_handler(trap_code) {
    ...
    if (trap_code == write_code)
        write_handler( );
    ...
}
```



# System Calls

- A **trap** is a type of "**software interrupt**"  
 — interrupt handler will invoke trap handler

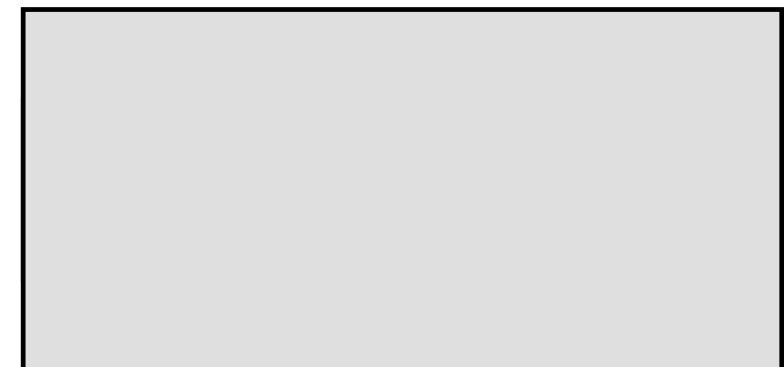
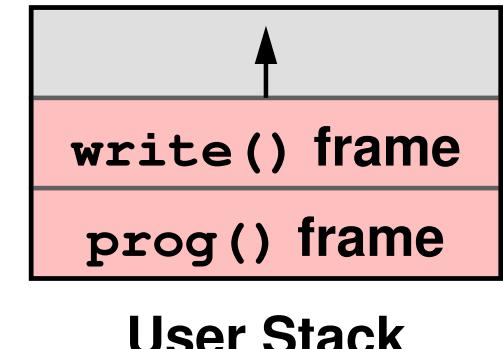
```
prog( ) {
    ...
    write(fd, buffer, size);
    ...
}
```

```
        write( ) {
            ...
            trap(write_code);
        }
```

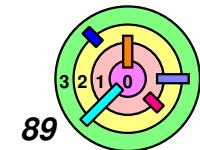
User  
Kernel

```
intr_handler(intr_code) {
    ...
    if (intr_code == SYSCALL)
        syscall_handler( );
    ...
}

syscall_handler(trap_code) {
    ...
    if (trap_code == write_code)
        write_handler( );
    ...
}
```



Kernel Stack

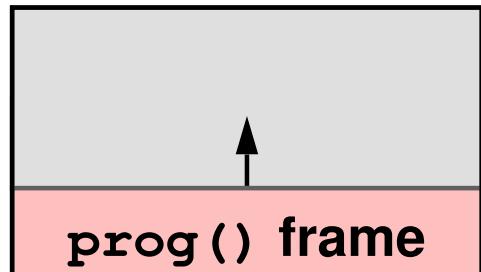


# System Calls

- A **trap** is a type of "*software interrupt*"
- interrupt handler will invoke trap handler

```
prog( ) {
    ...
    write(fd, buffer, size);
    ...
}
```

```
        write( ) {
            ...
            trap(write_code);
            ...
        }
```



User Stack

User

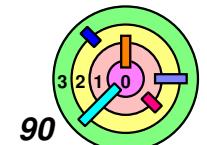
Kernel

```
intr_handler(intr_code) {
    ...
    if (intr_code == SYSCALL)
        syscall_handler( );
    ...
}

syscall_handler(trap_code) {
    ...
    if (trap_code == write_code)
        write_handler( );
    ...
}
```



Kernel Stack



# System Calls

- A **trap** is a type of "**software interrupt**"  
 — interrupt handler will invoke trap handler

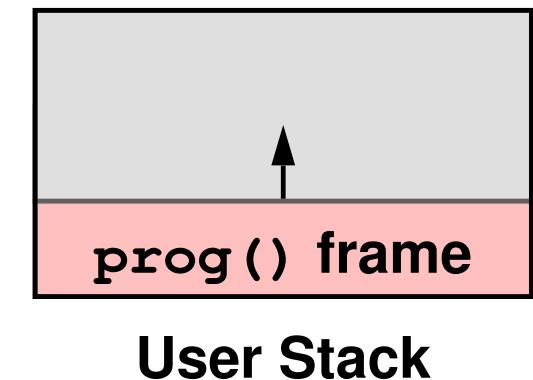
```
prog( ) {
  ...
  write(fd, buffer, size);
  ...
}
```

```
write( ) {
  ...
  trap(write_code);
  ...
}
```

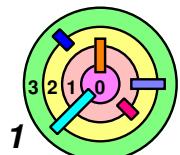
User  
Kernel

```
intr_handler(intr_code) {
  ...
  if (intr_code == SYSCALL)
    syscall_handler();
  ...
}

syscall_handler(trap_code) {
  ...
  if (trap_code == write_code)
    write_handler();
  ...
}
```



Kernel Stack



# System Calls

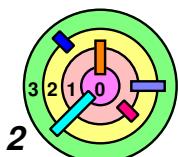


More details on the "*trap*" machine instruction

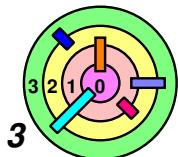
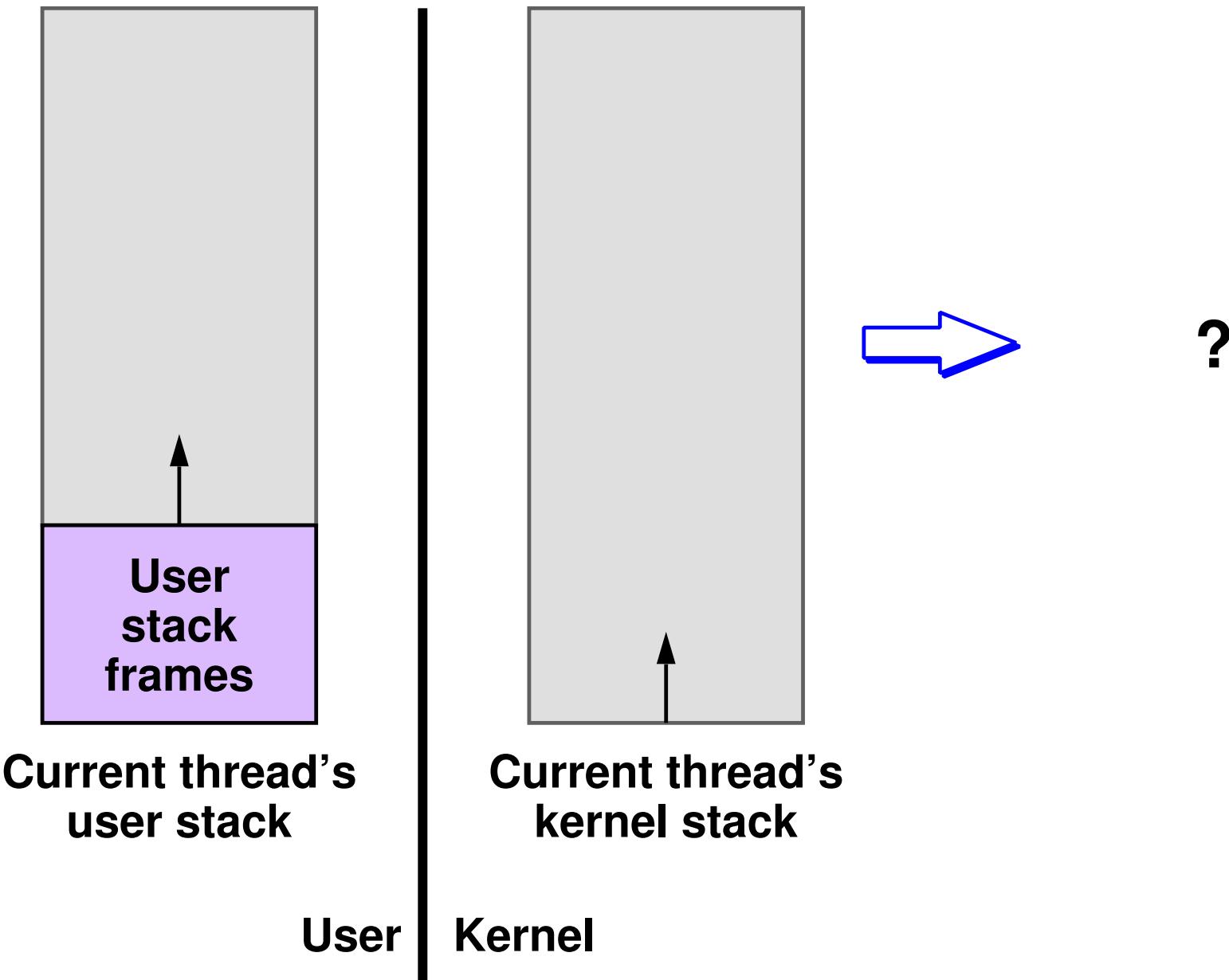
- 1) Trap into the kernel with all *interrupt disabled* and processor mode set to *privileged mode*
- 2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space (e.g., *the interrupt stack*)
  - additional registers may be saved
  - *HAL* is *hardware-dependent* (outside the scope of this class)
- 3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user thread (information from PCB)
- 4) HAL sets IP to *interrupt handler* (written in C)
  - copy user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt*
- 5) On return from the trap handler, disable interrupt and executes a special "return" instruction to *return to user process*
  - iret on x86



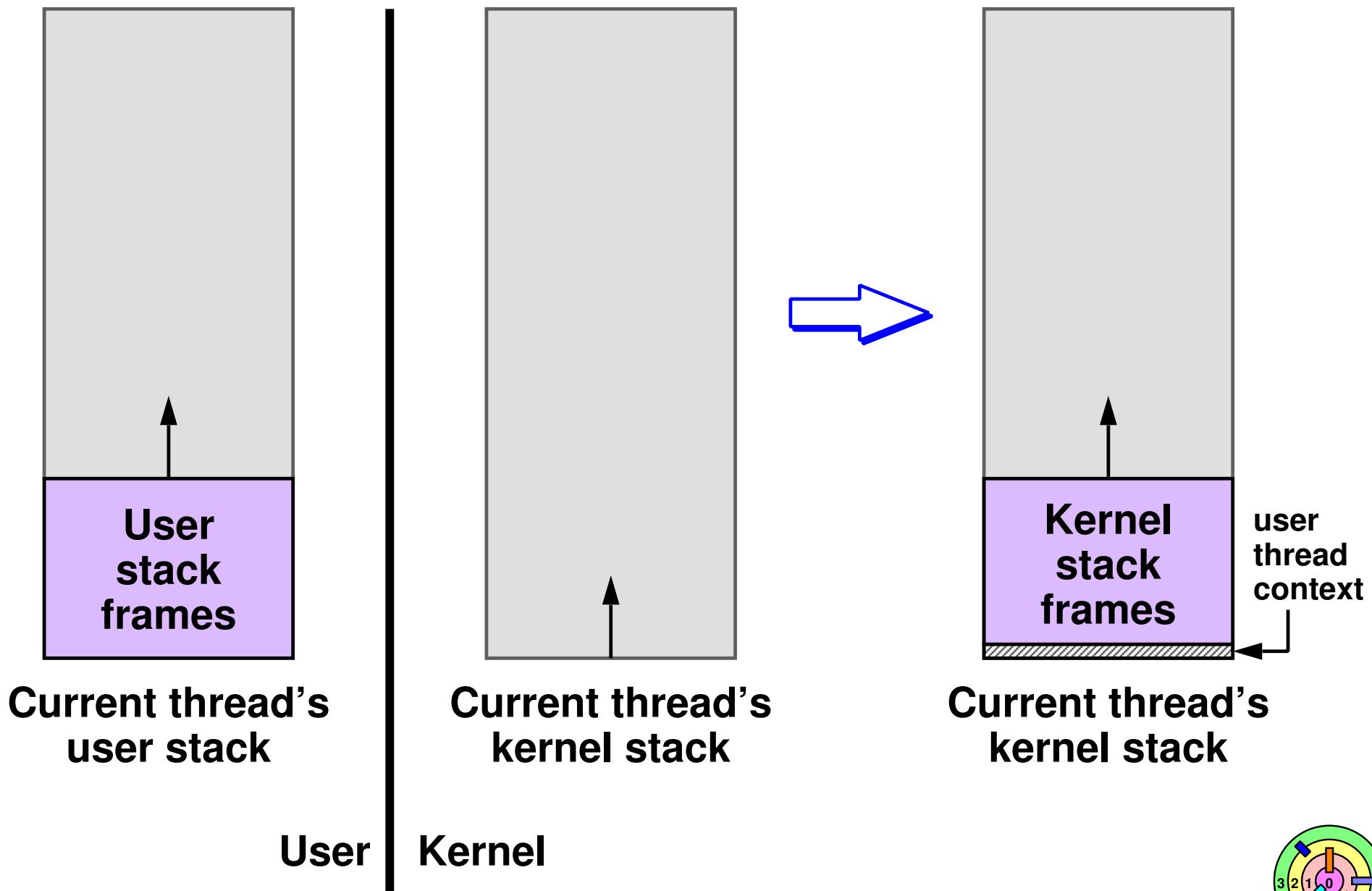
Similar sequence happens when you get *hardware interrupt*



# System Calls



# System Calls



# System Calls



More details on the "*trap*" machine instruction

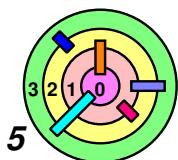
interrupt context

- 1) Trap into the kernel with all *interrupt disabled* and processor mode set to *privileged mode*
- 2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space (e.g., the *interrupt stack*)
  - additional registers may be saved
  - *HAL* is *hardware-dependent* (outside the scope of this class)
- 3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user process (information from PCB)
- 4) HAL sets IP to *interrupt handler* (written in C)
  - copy user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt*
- 5) On return from the trap handler, disable interrupt and executes a special "return" instruction to *return to user process*
  - iret on x86

interrupt context

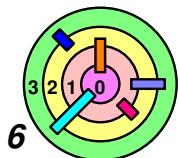


Similar sequence happens when you get *hardware interrupt*



# Context Switch

- The big idea here is that in order to perform a context switch, you must **save** your context, **build** new context, then **switch** to it
  - therefore, you must know what constitutes the context
  - then you save all of it
    - what's the **minimum** amount of context to save?
    - context can be stored in several places
      - ◊ stack
      - ◊ thread control block (e.g., in a system call, the TCB contains pointers to **both** the corresponding user stack frame and the kernel stack frame)
      - ◊ etc.
  - when switching back, you must restore the context
- In general, it's difficult to make a "clean" context switch
  - when you switch from context A to context B
    - there may be time you are in the context of **both** A and B
    - there may be time you are in **neither** contexts



# 3.1 Context Switching

- ▶ Procedures
- ▶ Threads & Coroutines
- ▶ Systems Calls
- ▶ *Interrupts*

S21-Q10

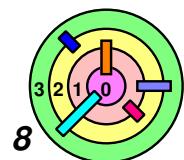
# Interrupts

→ Do not confuse *interrupts* with *signals* (even though the terminologies related to them are similar)

- *signals* are *generated by the kernel*
  - they are delivered to the *user process*
  - *signal* ≠ *software interrupt*
- *interrupts* are *generated by the hardware*
  - they are delivered to the *kernel*
    - ◊ they are delivered to HAL and then the kernel

→ When an *interrupt* occurs, the processor puts aside the current context and switch to an *interrupt context*

- the current context can be a *thread (user or kernel) context* or *another interrupt context*
  - need to be able to *mask/block individual interrupts* (similar to *signal* masking/blocking)
    - ◊ separate from enabling/disabling interrupt
- when the interrupt handler finishes, the processor generally resumes the context that was interrupted



# Interrupting A User Thread



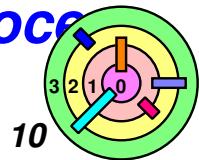
- If interrupt occurs when a *user thread* is executing in the CPU
  - 1) *Disable interrupt* and set processor mode to *privileged mode*
  - 2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space (e.g., *the interrupt stack*)
    - additional registers may be saved
    - *HAL* is *hardware-dependent* (outside the scope of this class)
  - 3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user thread (information from PCB)
    - umm... which kernel stack?
  - 4) HAL sets IP to *interrupt handler* (written partly in C)
    - copy user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt (with some interrupts blocked/masked)*
    - stay in *interrupt context*
  - 5) On return from the interrupt handler, disable interrupt and executes a special "return" instruction to *return to user process*
    - iret on x86



# Interrupting A User Thread

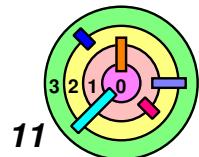


- If interrupt occurs when a *user thread* is executing in the CPU
  - 1) *Disable interrupt* and set processor mode to *privileged mode*
  - 2) The *Hardware Abstraction Layer (HAL)* save IP and SP in "temporary locations" in kernel space (e.g., *the interrupt stack*)
    - additional registers may be saved
    - *HAL* is *hardware-dependent* (outside the scope of this class)
  - 3) HAL sets the SP to point to the *kernel stack* designated for the corresponding user thread (information from PCB)
    - umm... which kernel stack?
  - 4) HAL sets IP to *interrupt handler* (written partly in C)
    - copy user IP and SP from "temporary location" and push them onto kernel stack, then *re-enable interrupt (with some interrupts blocked/masked)*
    - stay in *interrupt context*
  - 5) On return from the interrupt handler, disable interrupt and executes a special "return" instruction to *return to user process*
    - *iret* on x86

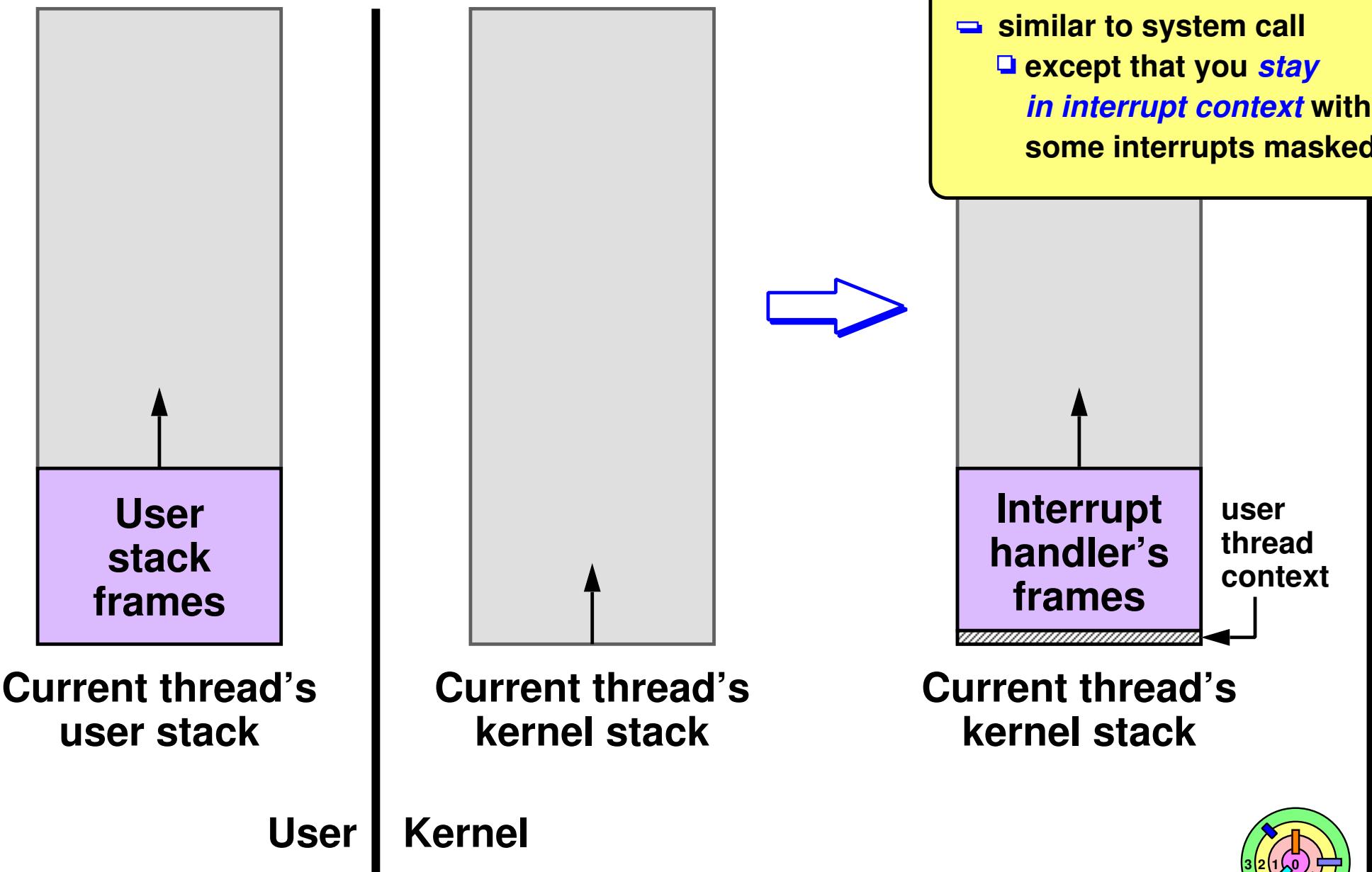


# Interrupts

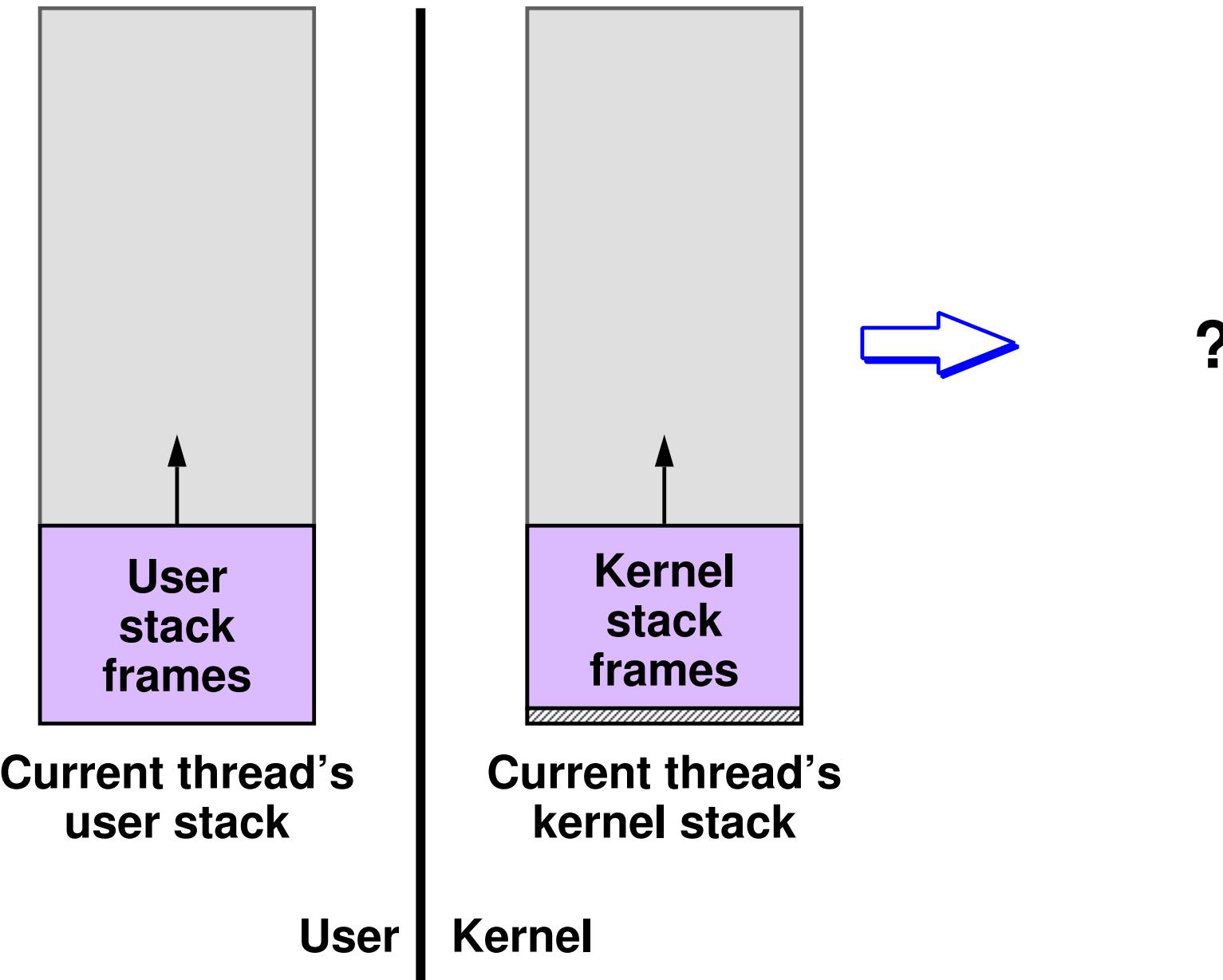
- **Interrupt service routine** is executed in an *interrupt context*
  - no thread context here
- Interrupt service routine is written in C (mixed with assembly code)
  - to execute C code, you need a stack
  - which stack should it use?
  - there are several possibilities
    - 1) allocate a new stack each time an interrupt occurs
      - ◆ too slow
    - 2) have one stack shared by all interrupt handlers
      - ◆ not often done
    - 3) interrupt handler could *borrow the kernel stack* from the thread it is interrupting
      - ◆ most common



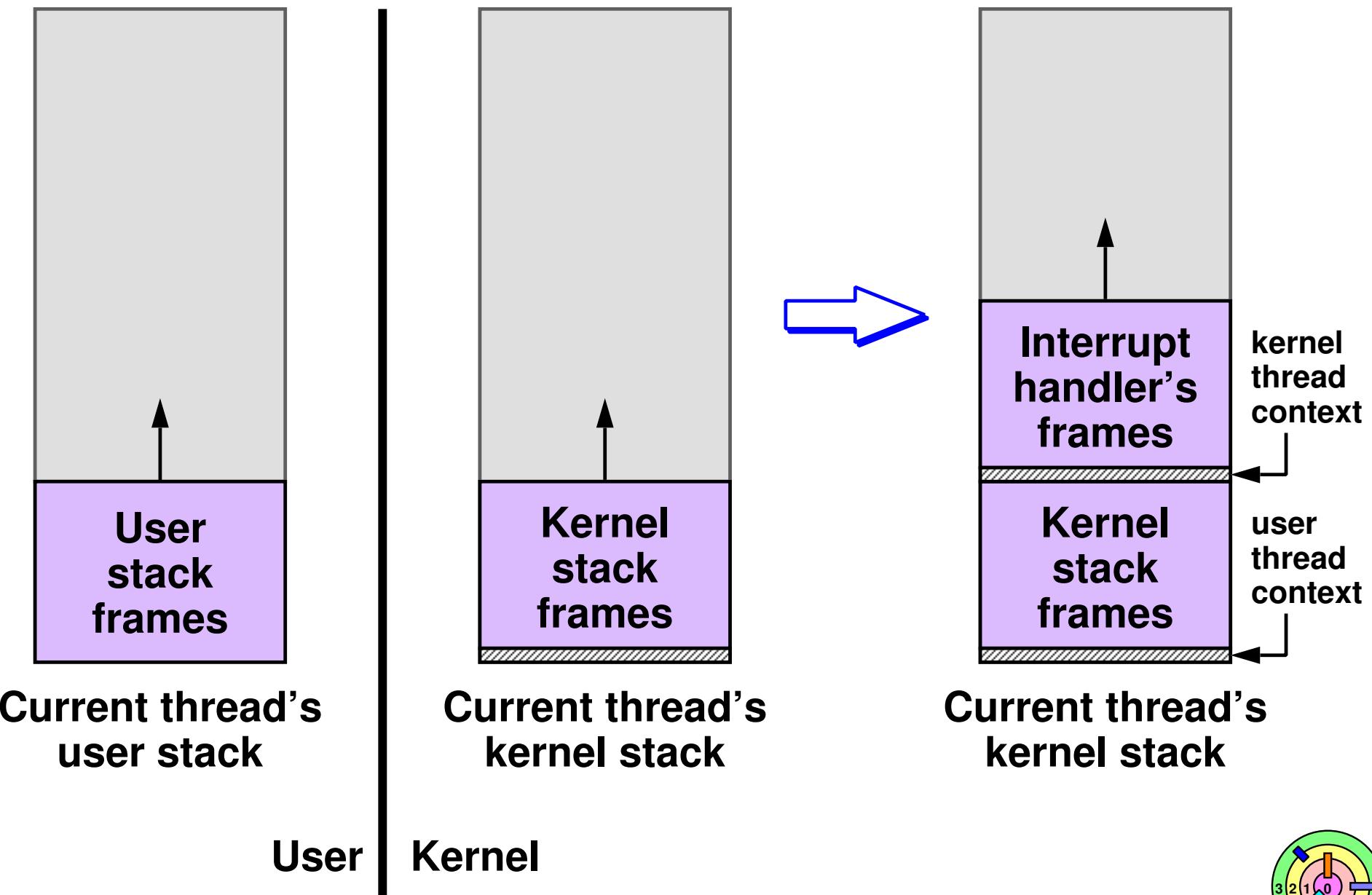
# Interrupting User Thread



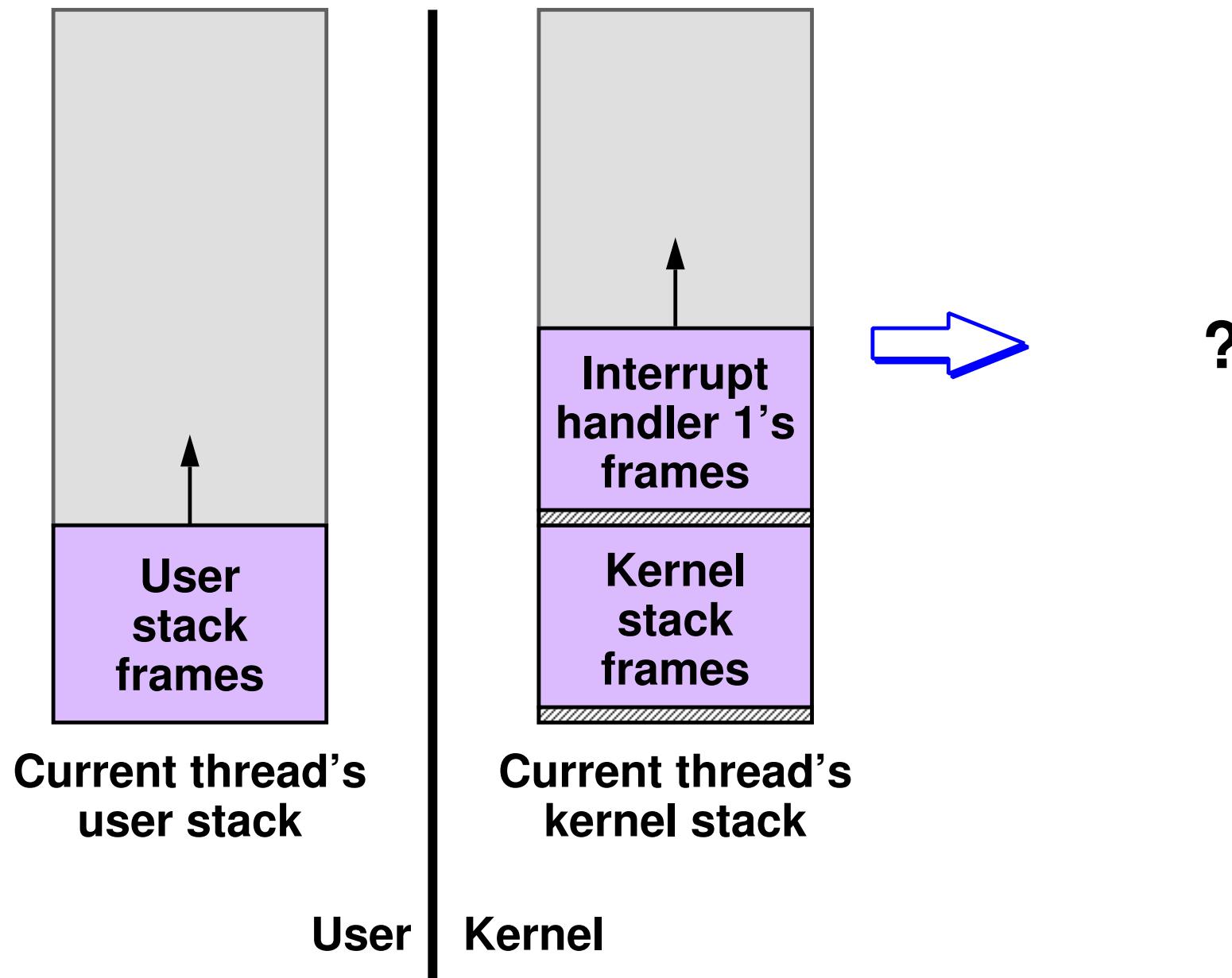
# Interrupting Kernel Thread



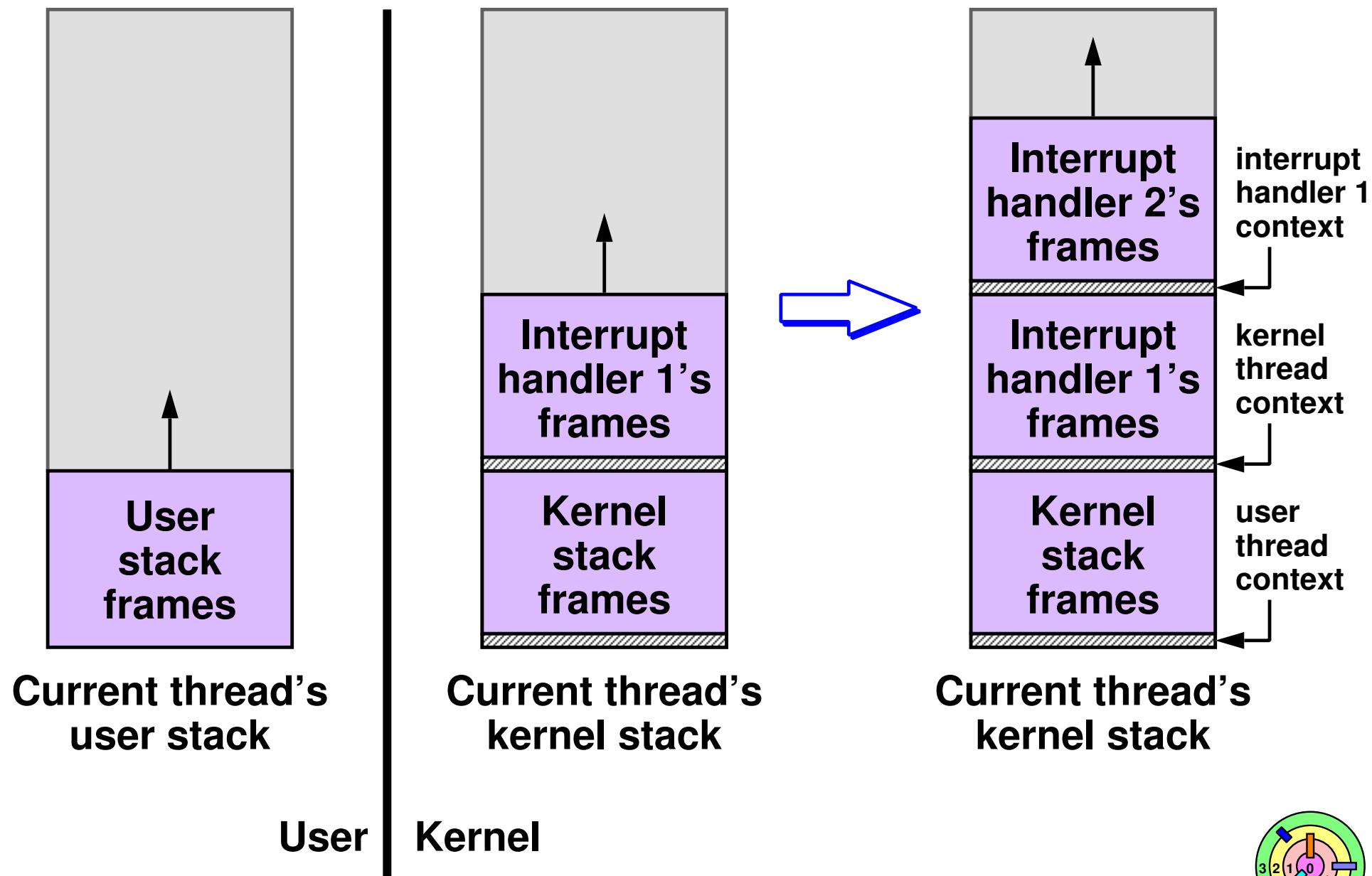
# Interrupting Kernel Thread



# Interrupting Another Interrupt Service Routine

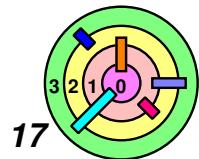


# Interrupting Another Interrupt Service Routine



# Interrupts

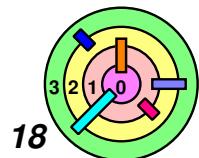
- ➡ For approaches (2) and (3), there is **no way** to suspend one interrupt handler and **resume** the execution of another
  - since there is only **one stack** for all the interrupt handlers
    - the global variable `CurrentThread` does not change!
  - therefore, the handler of the most recent interrupt must **run to completion**
    - when it's done, the stack frame is removed, and the next-most-recent interrupt now must run to completion
  - this is a **big deal!**
    - once you have interrupt handlers running, a normal thread (no matter how important it is) cannot run until **all** interrupt handlers complete
      - ◊ this is why an **interrupt service routine** should **do as little as possible** (and figure out a way to do the rest later)
    - if we have approach (1), then we won't have this problem
      - ◊ but it's so slow that it's unacceptable



# Interrupts

→ What if an interrupt service routine takes too long to run?

- interrupt handler places a description of the work that must be done on a queue of some sort, then arranges for it to be done in some other context at a later time
  - still need to do something in the interrupt handler
    - 1) *unblock a kernel thread* that's sleeping in the corresponding I/O queue
    - 2) *start the next I/O operation* on the same device
  - this approach is used in many systems, including Windows and Linux
    - will discuss further in Ch 5



# Interrupt Mask



The CPU can have interrupt *disabled*

- if any interrupt occurs while interrupt is disabled, the interrupt indication remains *pending*
- once interrupt is *enabled*, a pending interrupt is *delivered* and the CPU is interrupted to execute a corresponding *interrupt service routine*
- disable/enable *all* interrupts



When interrupt is *enabled*, *individual* interrupt can be *masked*, i.e., temporarily *blocked*

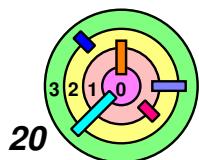
- similar to signal masking/blocking in user space programs
  - although you cannot "disable" all signals in user space
- if an interrupt occurs while it is masked, the interrupt indication remains *pending*
- once that interrupt is unmasked/unblocked, the interrupt is *delivered* and the CPU is interrupted to execute a corresponding *interrupt service routine (ISR)*



# Interrupt Mask

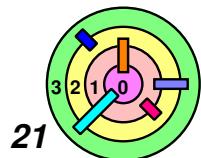
→ How interrupts are masked/blocked is architecture-dependent  
— common approaches

- 1) hardware register implements a *bit vector / mask*
  - ◊ if a particular bit is set, the corresponding interrupt class is enable (or disabled)
  - ◊ the kernel masks interrupts by setting bits in the register
  - ◊ when an interrupt does occur, the corresponding mask bit is set in the register (block other interrupts of the same class)
  - ◊ cleared when the handler returns
- 2) *hierarchical interrupt levels* (more common)

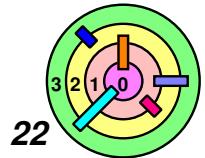


# Interrupt Mask

- ➡ How interrupts are masked/blocked is architecture-dependent
  - common approaches
    - 1) hardware register implements a *bit vector / mask*
    - 2) *hierarchical interrupt levels* (more common)
      - ◆ the processor masks interrupts by setting an **su21-den-Q5** *Interrupt Priority Level (IPL)* in a hardware register
      - ◆ all interrupts with the *current or lower levels* are masked
      - ◆ the kernel masks a class of interrupts by setting the IPL to a particular value
      - ◆ when an interrupt does occur, the current IPL is set to that of the level the interrupt belongs
      - ◆ restores to previous value on handler return
  - even if (1) is implemented in the hardware, abstraction (in HAL) can be built to make it look as if (2) is implemented
    - easier for kernel programmers to work with interrupts
    - for this class, we assume that (2) is used



# 3.2 Input/Output Architectures



# Input/Output

su21-a-Q3



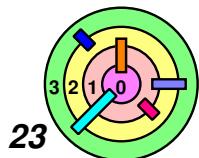
## Architectural concerns

- memory-mapped I/O
  - programmed I/O (**PIO**)
  - direct memory access (**DMA**)
- I/O processors (channels)



## Software concerns

- device drivers
- concurrency of I/O and computation



23

# What Does A Computer Look Like?

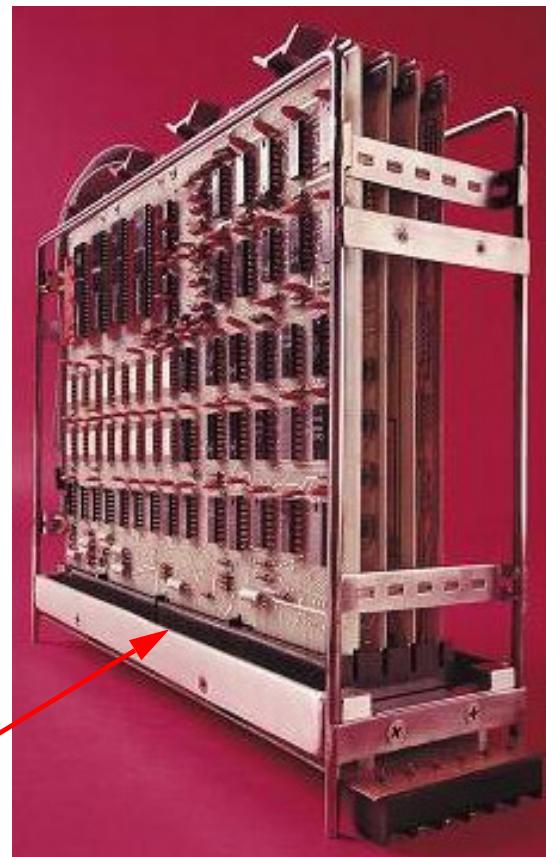
→ LSI-11

- processor for PDP-11

→ Boards are connected over  
a "bus"

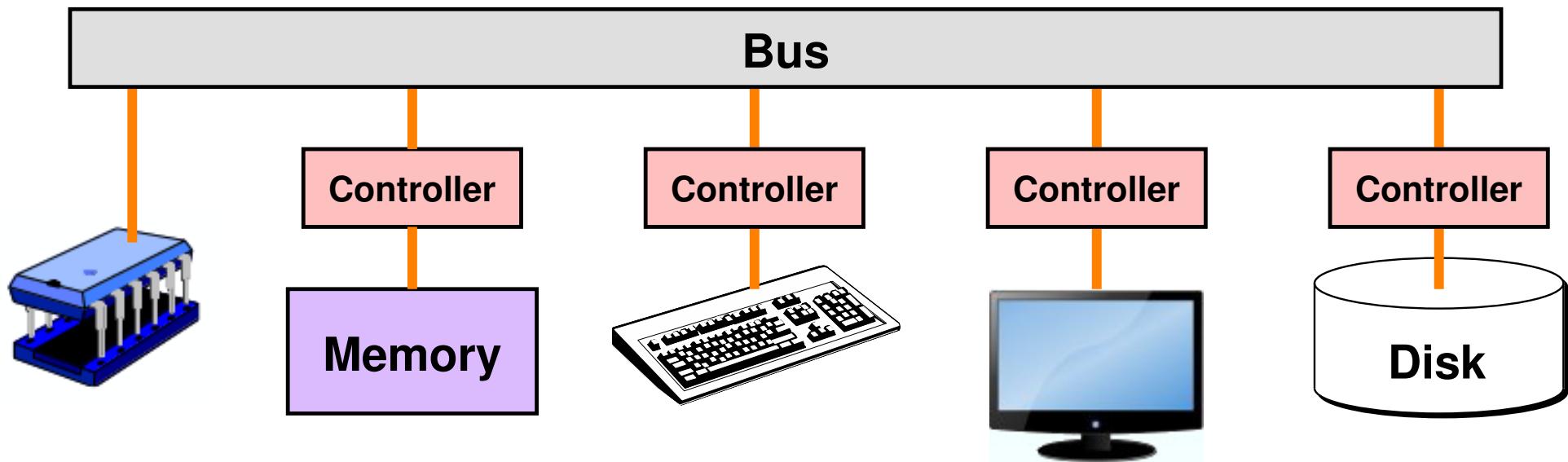
- on the "backplane"
- various standards for  
PDP-11
  - Unibus, Q-Bus, etc.

connect to backplane bus

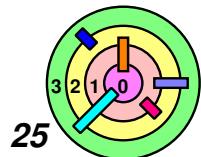


<http://hampage.hu/pdp-11/lsi11.html>

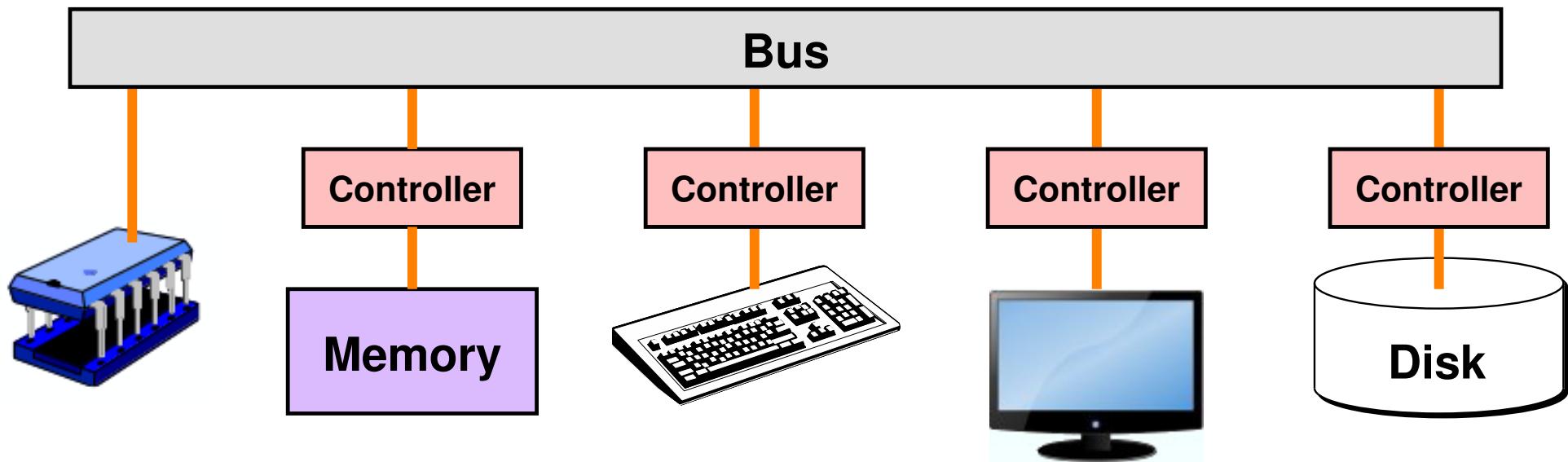
# Simple I/O Architecture



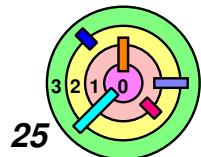
- **memory-mapped I/O**
  - all controllers listen on the bus to determine if a request is for itself or not
  - memory controller behaves differently from other controllers, i.e., it passes the bus request to primary memory
  - others "process" the bus request
    - ◊ and respond to relatively few addresses
    - ◊ if no one responds, you get a "bus error"
  - memory is not really a "device"



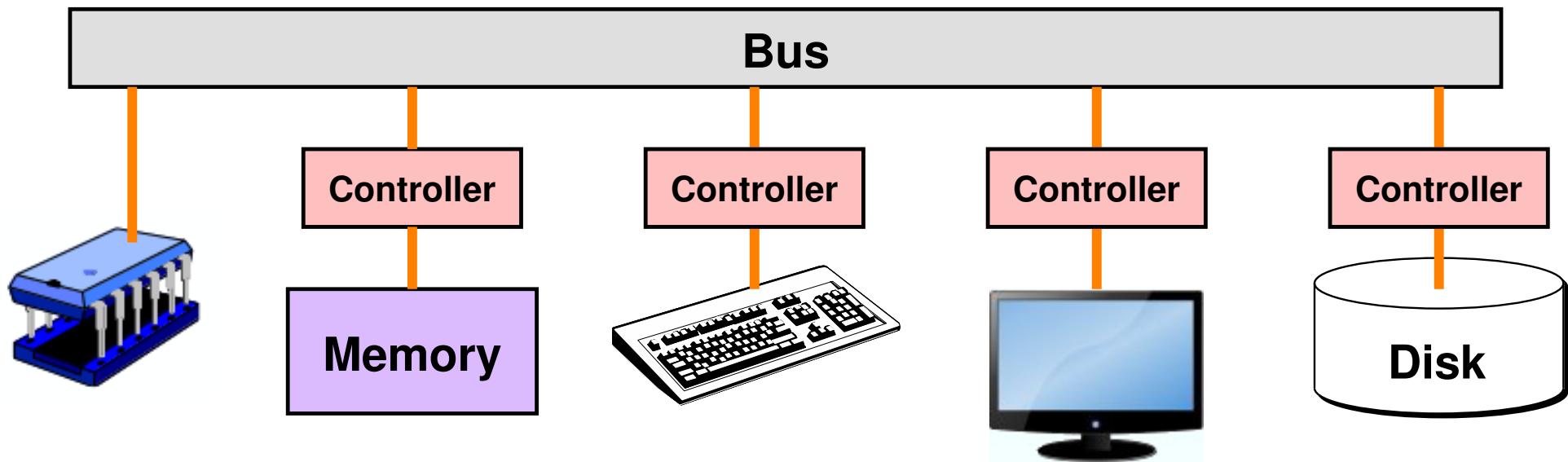
# Simple I/O Architecture



- **memory-mapped I/O**
  - all controllers listen on the bus to determine if a request is for itself or not
  - memory controller behaves differently from other controllers, i.e., it passes the bus request to primary memory
  - others "process" the bus request
    - ◊ and respond to relatively few addresses
    - ◊ if no one responds, you get a "bus error"
  - memory is not really a "device"

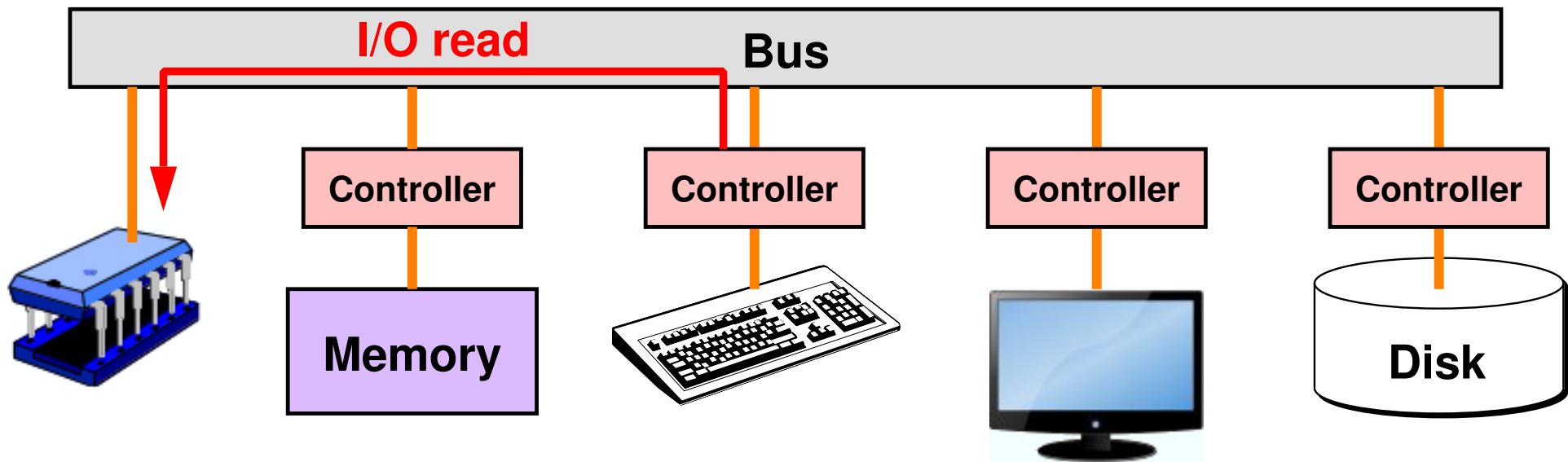


# Simple I/O Architecture



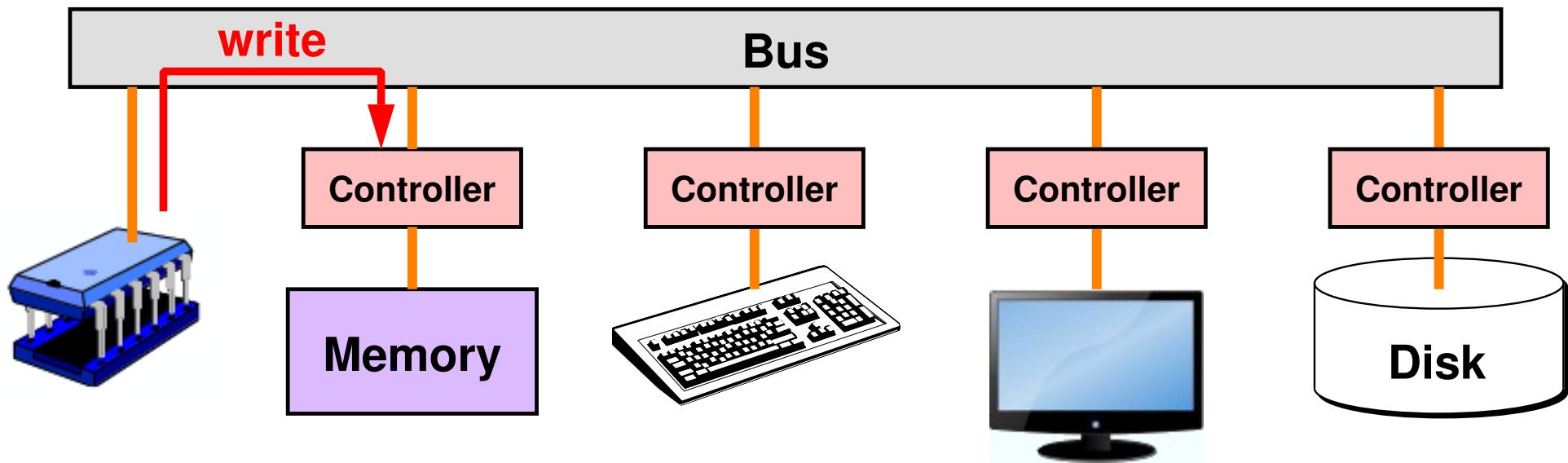
- **memory-mapped I/O**
- **two categories of devices**
  - **PIO (programmed I/O)**
    - ◆ perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

# Simple I/O Architecture



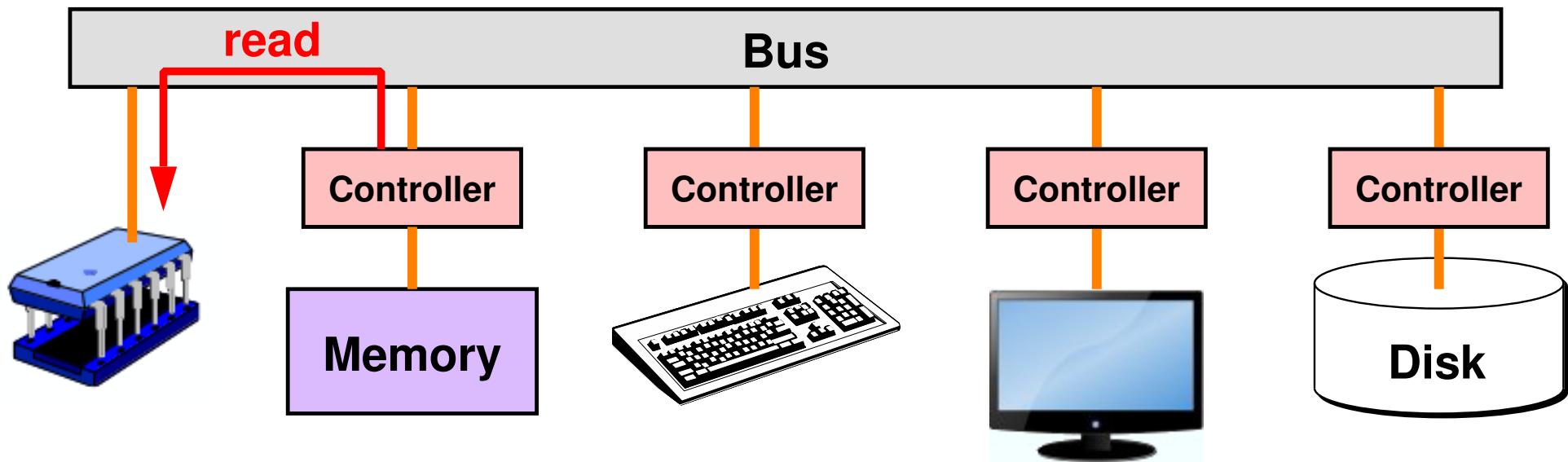
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - ◆ perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

# Simple I/O Architecture



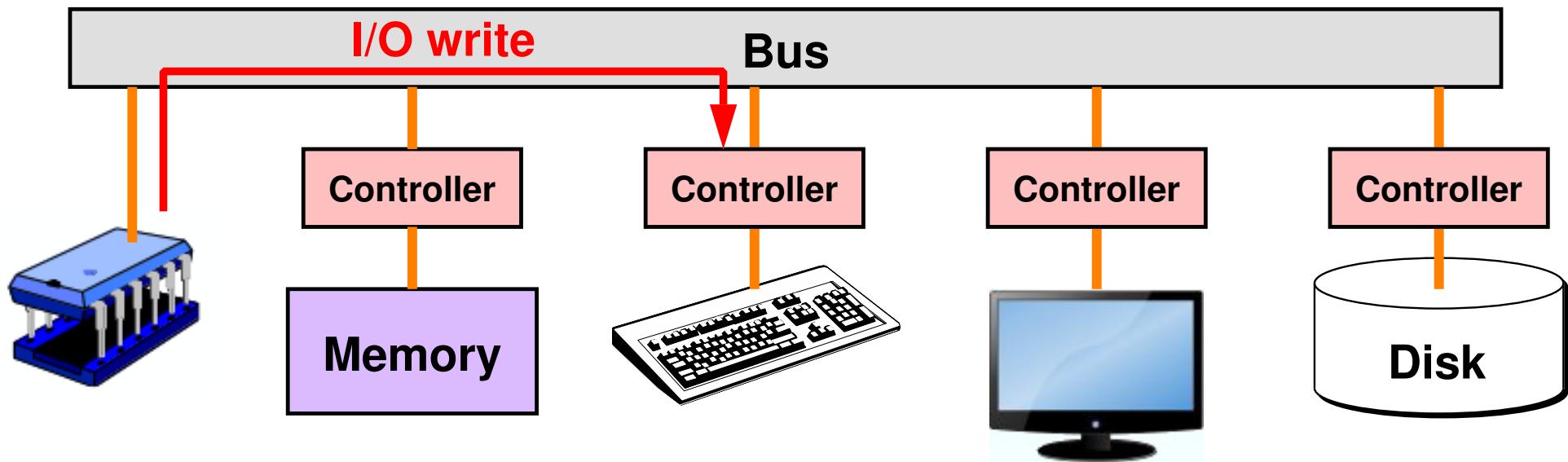
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - ◆ perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

# Simple I/O Architecture



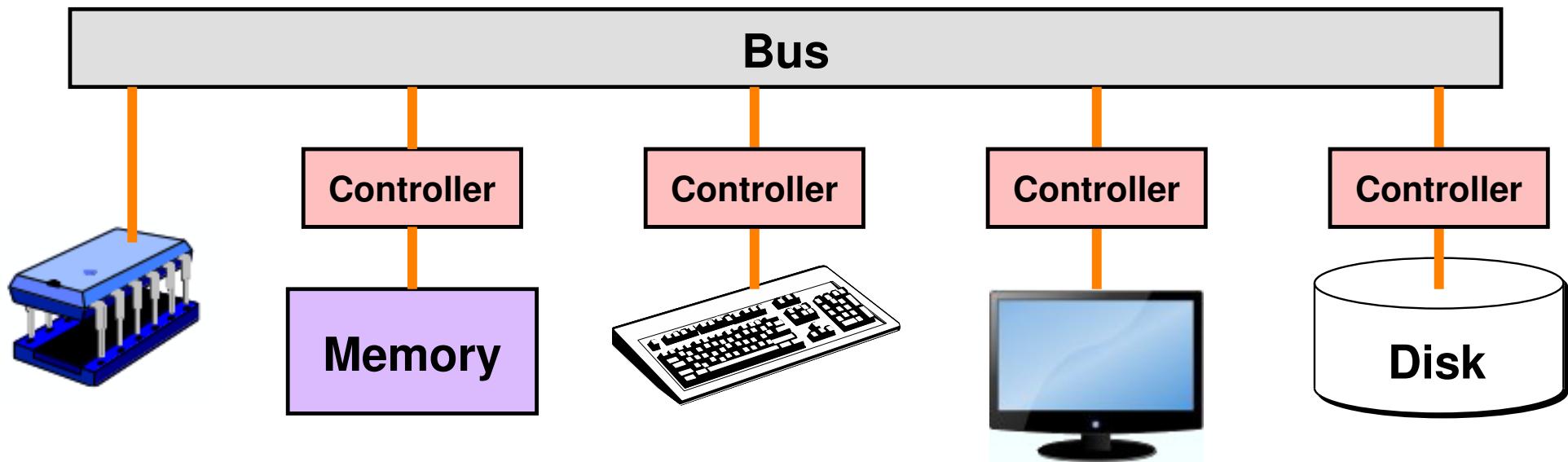
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - ◆ perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

# Simple I/O Architecture

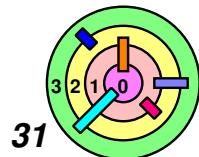


- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - ◆ perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

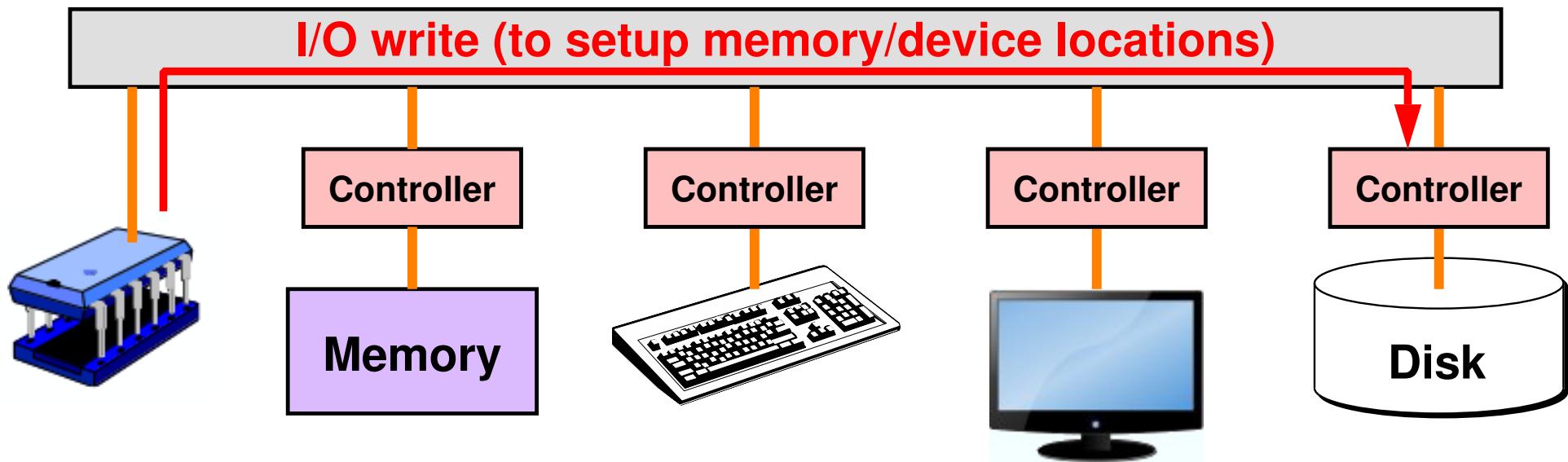
# Simple I/O Architecture



- **memory-mapped I/O**
- **two categories of devices**
  - **PIO (programmed I/O)**
  - **DMA (direct memory access)**
    - ◆ **the controller performs the I/O itself**
    - ◆ **the processor writes to the controller to tell it where to transfer the results to**
    - ◆ **the controller takes over and transfers data between itself and primary memory**

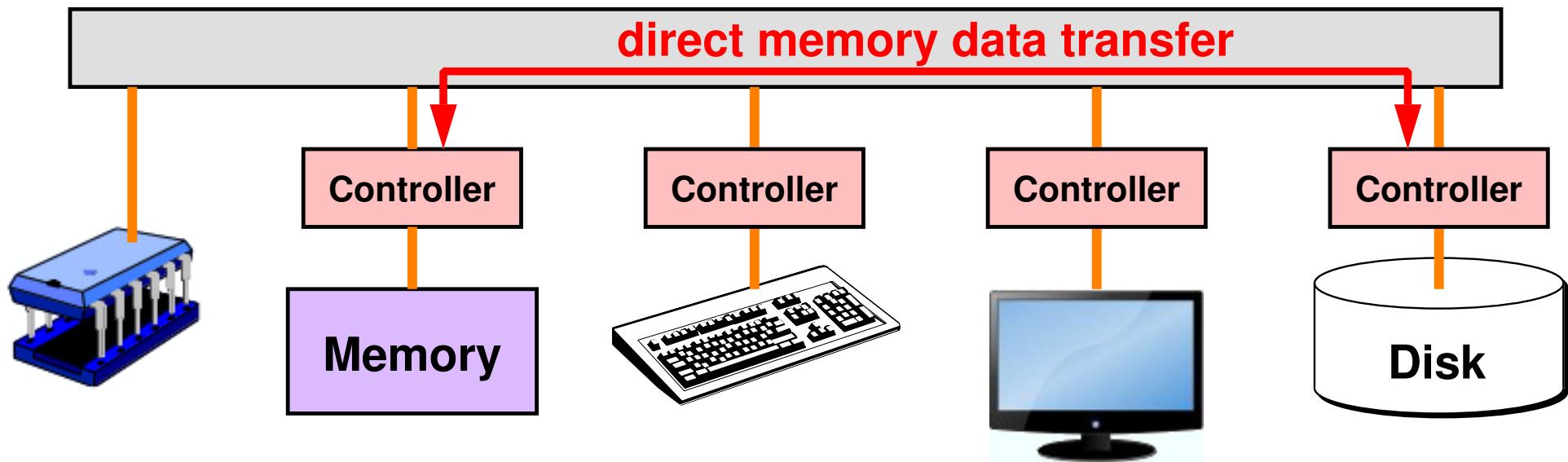


# Simple I/O Architecture

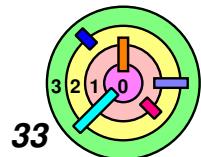


- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - ◊ the controller performs the I/O itself
    - ◊ the processor writes to the controller to tell it where to transfer the results to
    - ◊ the controller takes over and transfers data between itself and primary memory

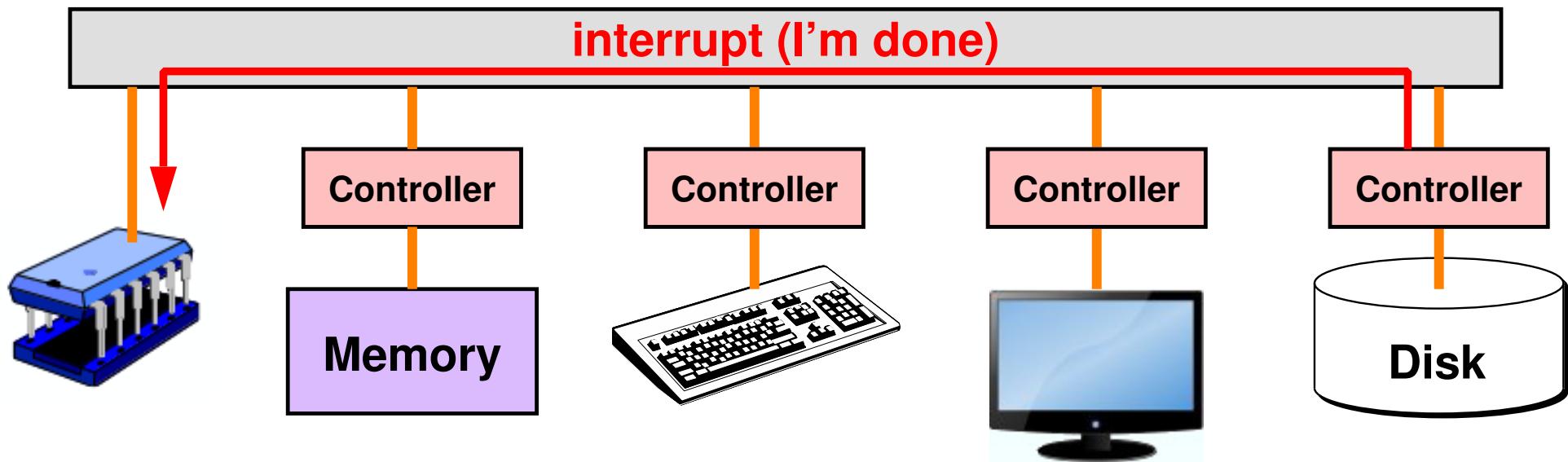
# Simple I/O Architecture



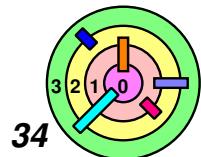
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - ◊ the controller performs the I/O itself
    - ◊ the processor writes to the controller to tell it where to transfer the results to
    - ◊ the controller takes over and transfers data between itself and primary memory



# Simple I/O Architecture



- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - ◊ the controller performs the I/O itself
    - ◊ the processor writes to the controller to tell it where to transfer the results to
    - ◊ the controller takes over and transfers data between itself and primary memory



# PIO Registers



This is the abstraction of a PIO device

- a "*register*" is just a *memory-mapped I/O address* on the bus

|      |      |     |     |  |  |  |  |
|------|------|-----|-----|--|--|--|--|
| GoR  | GoW  | IER | IEW |  |  |  |  |
| RdyR | RdyW |     |     |  |  |  |  |
|      |      |     |     |  |  |  |  |
|      |      |     |     |  |  |  |  |

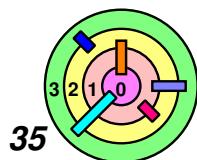
Control register (1 byte)

Status register (1 byte)

Read register (1 byte)

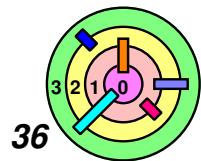
Write register (1 byte)

|         |      |                                    |
|---------|------|------------------------------------|
| Legend: | GoR  | Go read (start a read operation)   |
|         | GoW  | Go write (start a write operation) |
|         | IER  | Enable read-completion interrupts  |
|         | IEW  | Enable write-completion interrupts |
|         | RdyR | Ready to read                      |
|         | RdyW | Ready to write                     |



# Programmed I/O

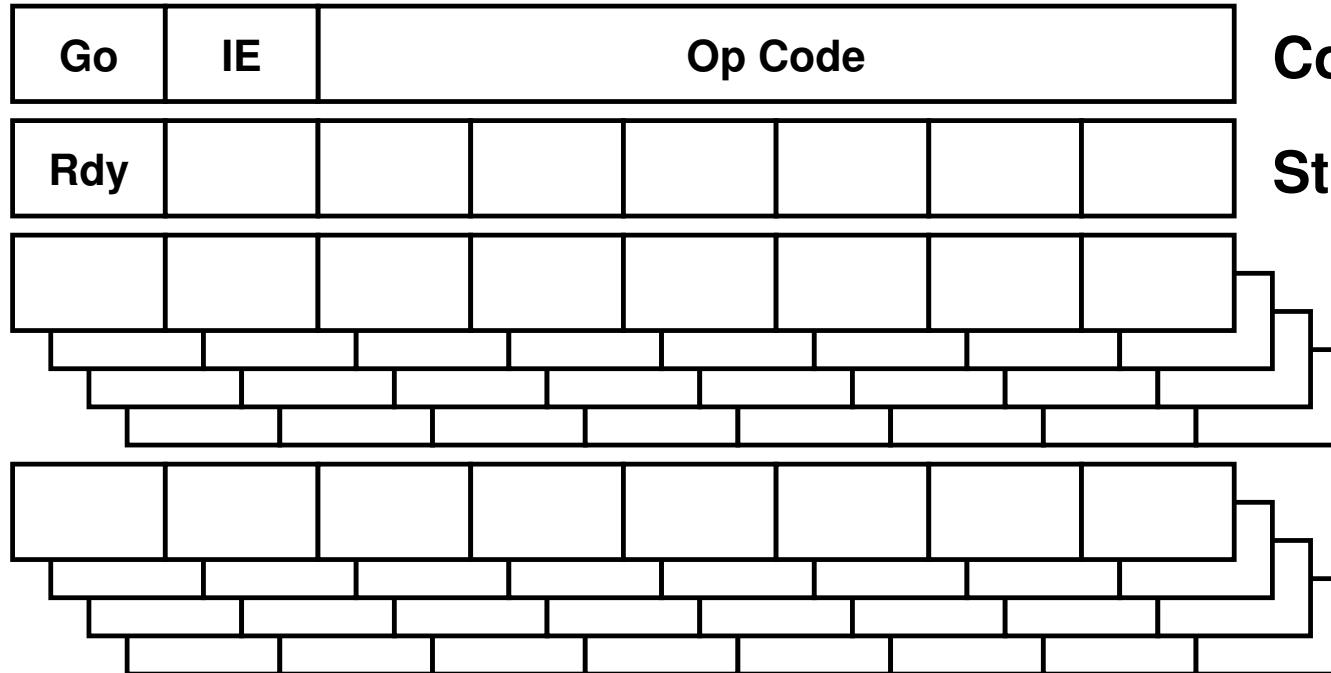
- ➡ E.g.: Terminal controller
- ➡ Procedure (write)
  - write a byte into the write register
  - set the GoW bit (and optionally the IEW bit if you'd like to be notified via an interrupt) in the control register
  - poll and wait for RdyW bit (in status register) to be set (if interrupts have been enabled, an interrupt occurs when this happens)



# DMA Registers

→ This is the abstraction of a DMA device

- a “*register*” is just a *memory-mapped I/O address* on the bus



Control register (1 byte)

Status register (1 byte)

Memory address  
register (4 bytes)

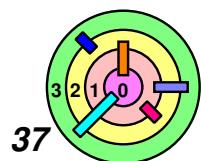
Device address  
register (4 bytes)

**Legend:** Go      Start an operation

Op Code      Operation code (identifies the operation)

IE      Enable interrupts

Rdy      Controller is ready

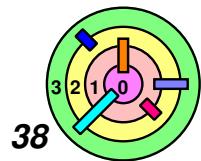


# Direct Memory Access

→ E.g.: Disk controller

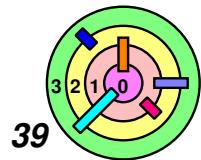
→ Procedure

- set the disk address in the device address register (only relevant for a seek request)
- set the buffer address in the memory address register
- set the op code (SEEK, READ or WRITE), the Go bit and, if desired, the IE bit in the control register
- wait for interrupt or for Rdy bit to be set



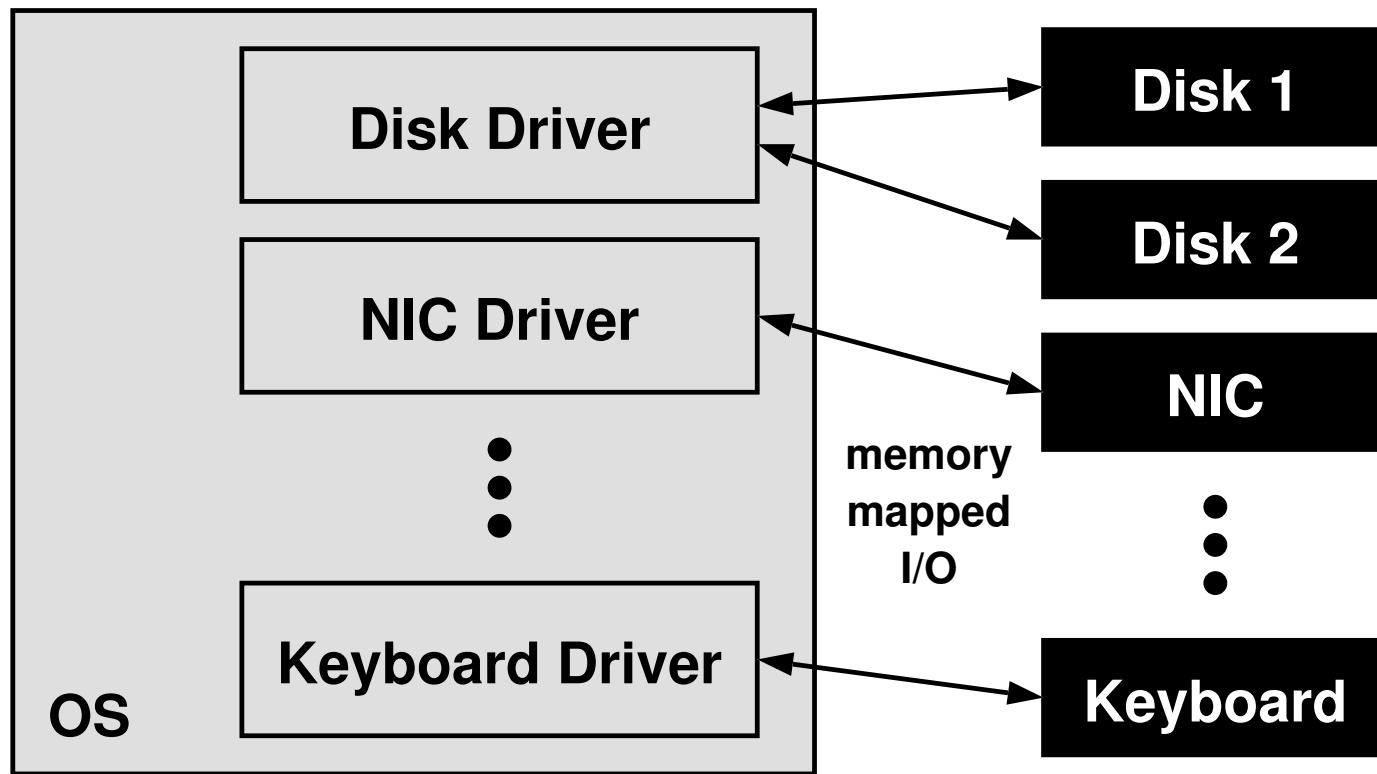
# Device Drivers

- Who knows how to use memory-mapped I/O to talk to devices?
- *not* the kernel developers
  - device manufacturers do
  - it's desirable if kernel is *device-independent*
    - but how?

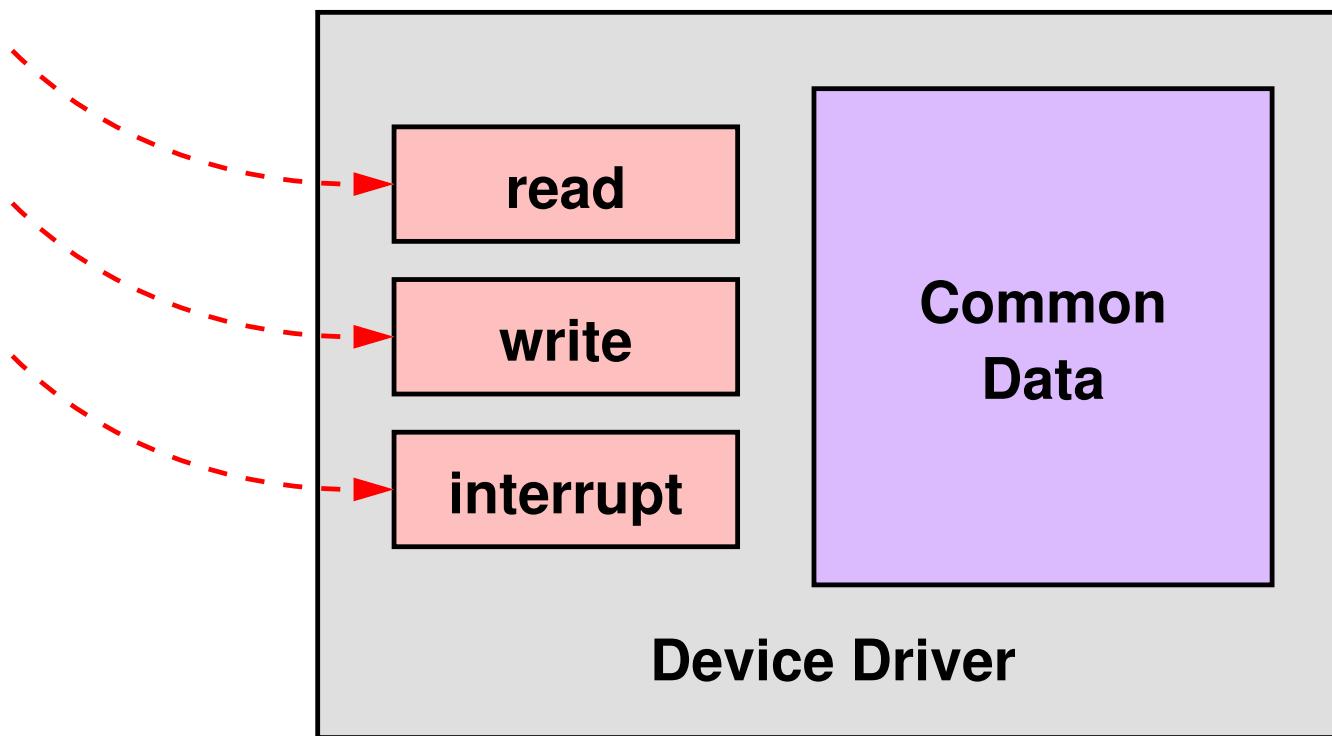


# Device Drivers

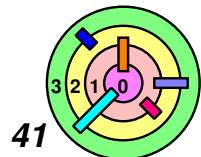
- Who knows how to use memory-mapped I/O to talk to devices?
  - *not* the kernel developers
  - device manufacturers do
  - it's desirable if kernel is *device-independent*
    - but how?
- Device manufacturers package their knowledge into *device drivers*



# Device Drivers



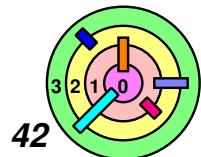
- ▶ **Device drivers** provide a **standard interface** to the rest of the OS
  - **code in device drivers** knows how to talk to devices (the rest of the OS really doesn't know the details)
  - OS can treat I/O in a **device-independent** manner by calling functions in the standard interface
    - "interface" = **array of function pointers**



# C++ Interface = Array of Function Pointers

```
class disk {  
public:  
    virtual status_t read(request_t) = 0;  
    virtual status_t write(request_t) = 0;  
    virtual status_t interrupt() = 0;  
};
```

- C++ **polymorphism** achieved using **virtual base class** or **interface**
  - each type of disk driver is a **subclass** of the disk class and has its own implementation of these functions
    - each disk driver looks like a generic disk to the OS
  - this gets compiled into an array of function pointers (which is what C++ code gets compiled into)
    - in reality, there are no object classes and no polymorphism
      - ◊ the CPU doesn't even know about data structures
      - ◊ the CPU only knows about memory addresses and how to execute machine instructions



# C Implementation of C++ Polymorphism

```

struct disk {
    void **disk_ops;
    ...
}

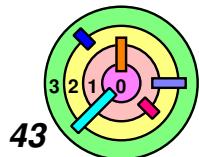
(void *)wd123_disk_ops[] = {
    read_handler_t wd123_read;
    write_handler_t wd123_write;
    intr_handler_t wd123_intr;
    ...
}

(void *)sg76_disk_ops[] = {
    read_handler_t sg76_read;
    write_handler_t sg76_write;
    intr_handler_t sg76_intr;
    ...
}

struct disk *d = ...;
d->disk_ops = wd123_disk_ops; /* known as "binding" */
/* or d->disk_ops = sg76_disk_ops; */

    — to read from any disk, call the first function indirectly
    (*((read_handler_t)d->disk_ops[0]))(...);

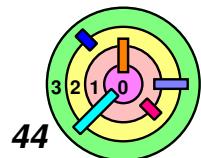
```



## ... in C++

```
class disk {  
public:  
    virtual status_t read(request_t) = 0;  
    virtual status_t write(request_t) = 0;  
    virtual status_t interrupt() = 0;  
};
```

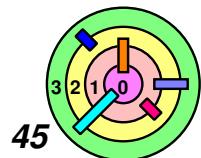
- This is a synchronous interface
  - a user thread would call `read/write()` system call
  - these functions are called in the kernel which starts the device
  - the device driver's interrupt method is called in the interrupt context
  - if I/O is completed, the thread is unblocked and return from the `read/write()` system call



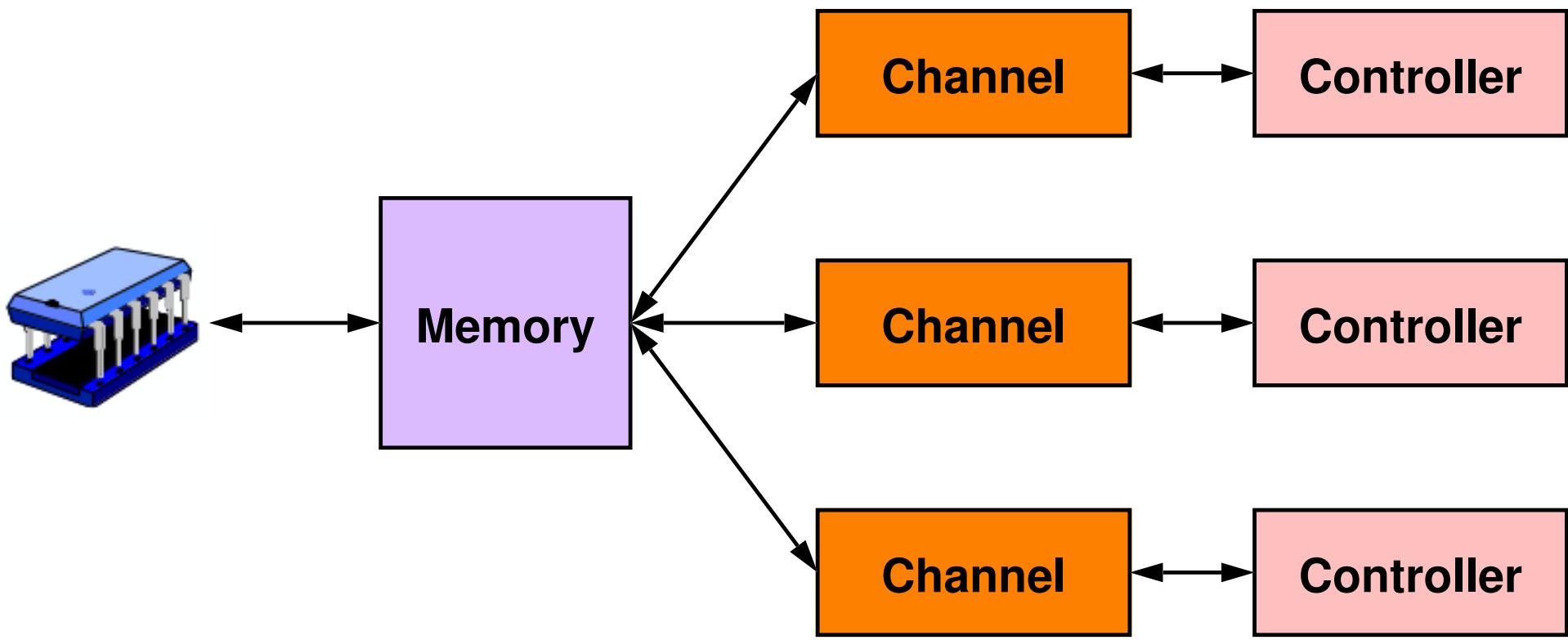
# A Bit More Realistic

```
class disk {  
public:  
    virtual handle_t start_read(request_t) = 0;  
    virtual handle_t start_write(request_t) = 0;  
    virtual status_t wait(handle_t) = 0;  
    virtual status_t interrupt() = 0;  
};
```

- Even in Sixth-Edition Unix, the internal driver interface is often **asynchronous**
- `start_read/start_write()` returns a handle identifying the operation that has started
  - a thread can call the `wait()` method to **synchronously** wait for *I/O completion*



# I/O Processors: Channels

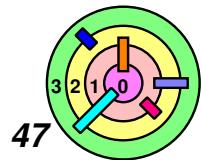


- when I/O costs dominate computation costs
  - use I/O processors (a.k.a. channels) to handle much of the I/O work
  - important in large data-processing applications
- can even download program into a channel processor



# 3.3 Dynamic Storage Allocation

- ➡ ***Best-fit & First-fit Algorithms***
- ➡ Buddy System
- ➡ Slab Allocation



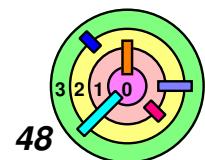
# Dynamic Storage Allocation

→ Where in the kernel do you need to do memory allocation?

- stack space
- `malloc()`
- `fork()`
- various OS data structures
  - process control block
  - thread control block
  - mutex (it's a queue)
- etc.

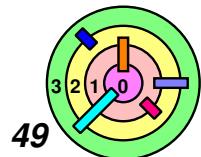
→ Memory allocator

- computer science people like to personify things
  - just like we say that a variable "lives" in an address space
- a "memory allocator" is simply a collection of functions
  - it's not a thread
  - it's not a process

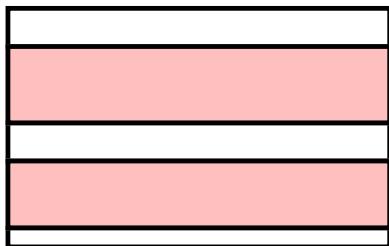


# Dynamic Storage Allocation

- Goal: allow dynamic creation and destruction of data structures
  - use "memory allocator" to manage a large block of memory (i.e., a collection of contiguous memory addresses)
    - **contract** with application:
      - ◊ application will not touch anything owned by the memory allocator (memory allocator can do anything with them)
      - ◊ application will only touch "allocated memory blocks" (memory allocator promises not to touch these)
    - very bad things can happen if contract is violated
      - ◊ assuming that there are no bugs in a memory allocator
- Concerns:
  - efficient use of storage
  - efficient use of processor time
- Example:
  - **first-fit** vs. **best-fit** allocation



# Allocation Example



1300 bytes free (first)

1200 bytes free (last)

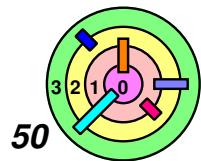
Allocate 1000 bytes:

First Fit

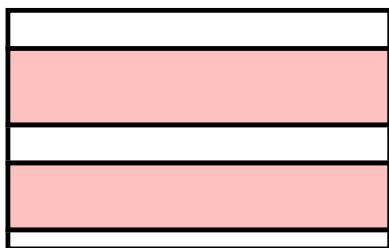
Best Fit

Allocate 1100 bytes:

Allocate 250 bytes:



# Allocation Example

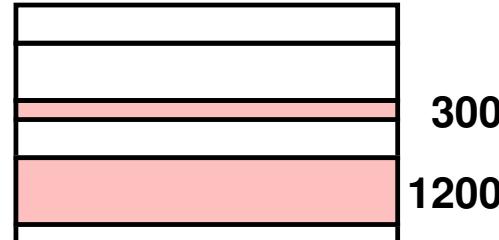


1300 bytes free (first)

1200 bytes free (last)

Allocate 1000 bytes:

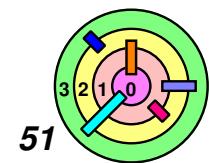
First Fit



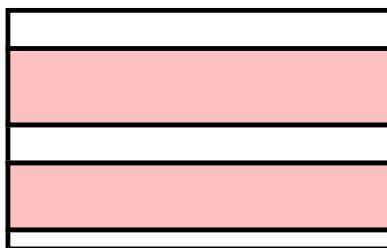
Best Fit

Allocate 1100 bytes:

Allocate 250 bytes:



# Allocation Example

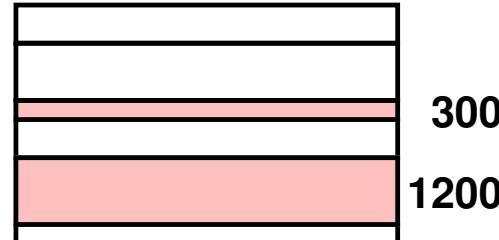


1300 bytes free (first)

1200 bytes free (last)

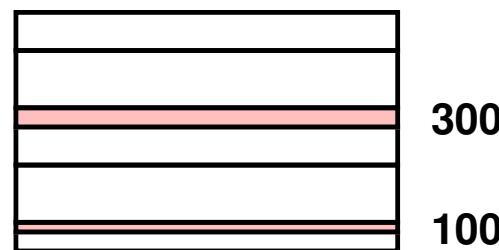
Allocate 1000 bytes:

First Fit



300  
1200

Best Fit

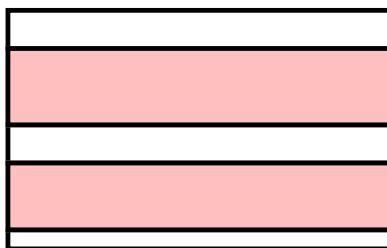


300  
100

Allocate 1100 bytes:

Allocate 250 bytes:

# Allocation Example

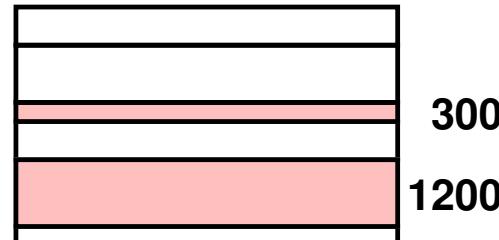


1300 bytes free (first)

1200 bytes free (last)

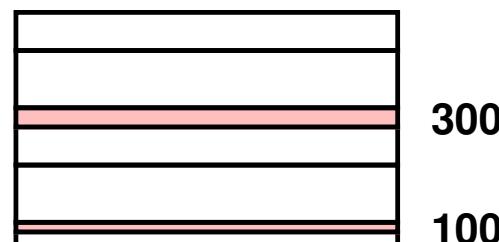
Allocate 1000 bytes:

First Fit

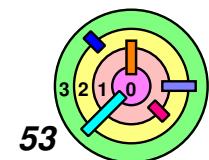
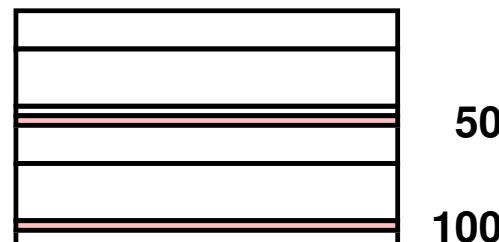


Best Fit

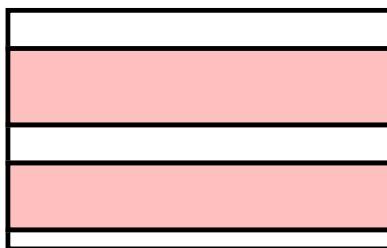
Allocate 1100 bytes:



Allocate 250 bytes:



# Allocation Example

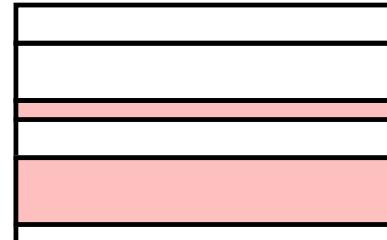


1300 bytes free (first)

1200 bytes free (last)

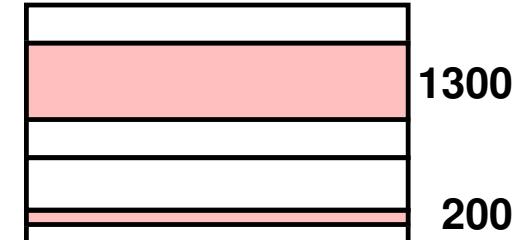
Allocate 1000 bytes:

First Fit



300  
1200

Best Fit



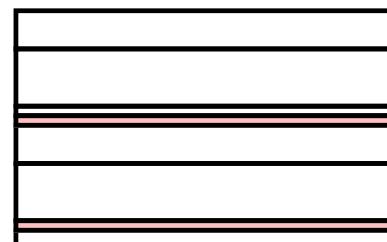
1300  
200

Allocate 1100 bytes:



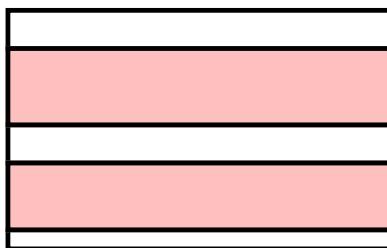
300  
100

Allocate 250 bytes:



50  
100

# Allocation Example

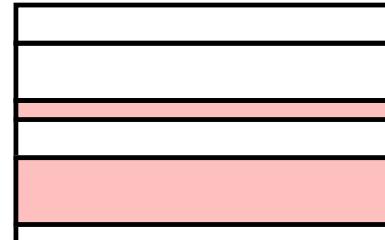


1300 bytes free (first)

1200 bytes free (last)

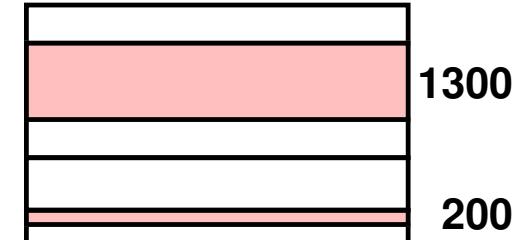
Allocate 1000 bytes:

First Fit



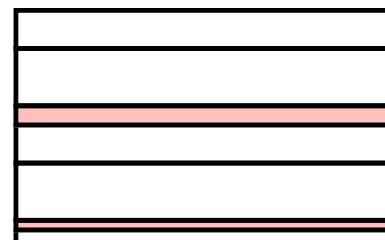
300  
1200

Best Fit



1300  
200

Allocate 1100 bytes:

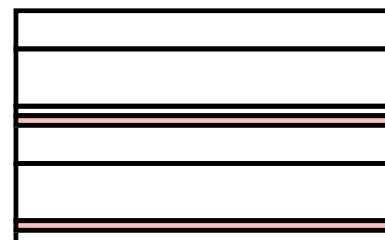


300  
100

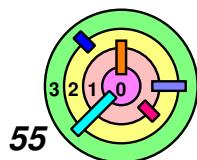


200  
200

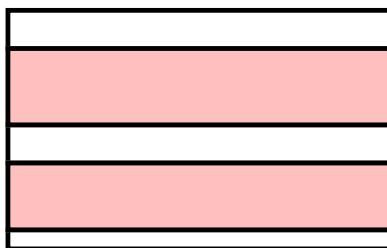
Allocate 250 bytes:



50  
100



# Allocation Example

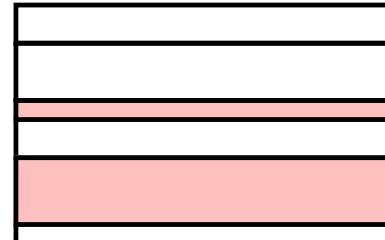


1300 bytes free (first)

1200 bytes free (last)

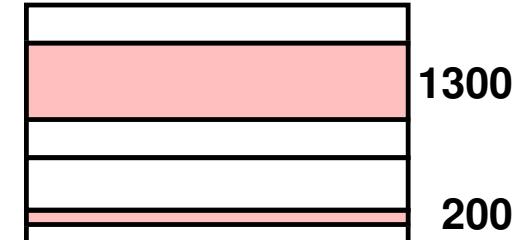
Allocate 1000 bytes:

First Fit



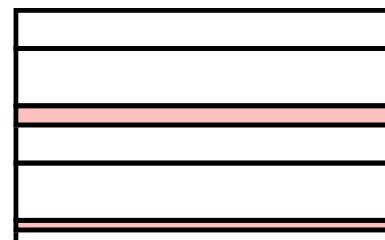
300  
1200

Best Fit



1300  
200

Allocate 1100 bytes:

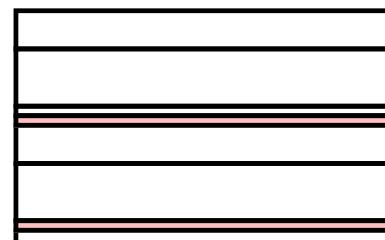


300  
100



200  
200

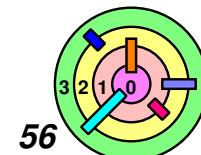
Allocate 250 bytes:



50  
100



200  
200

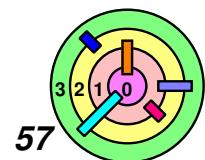


# Fragmentation

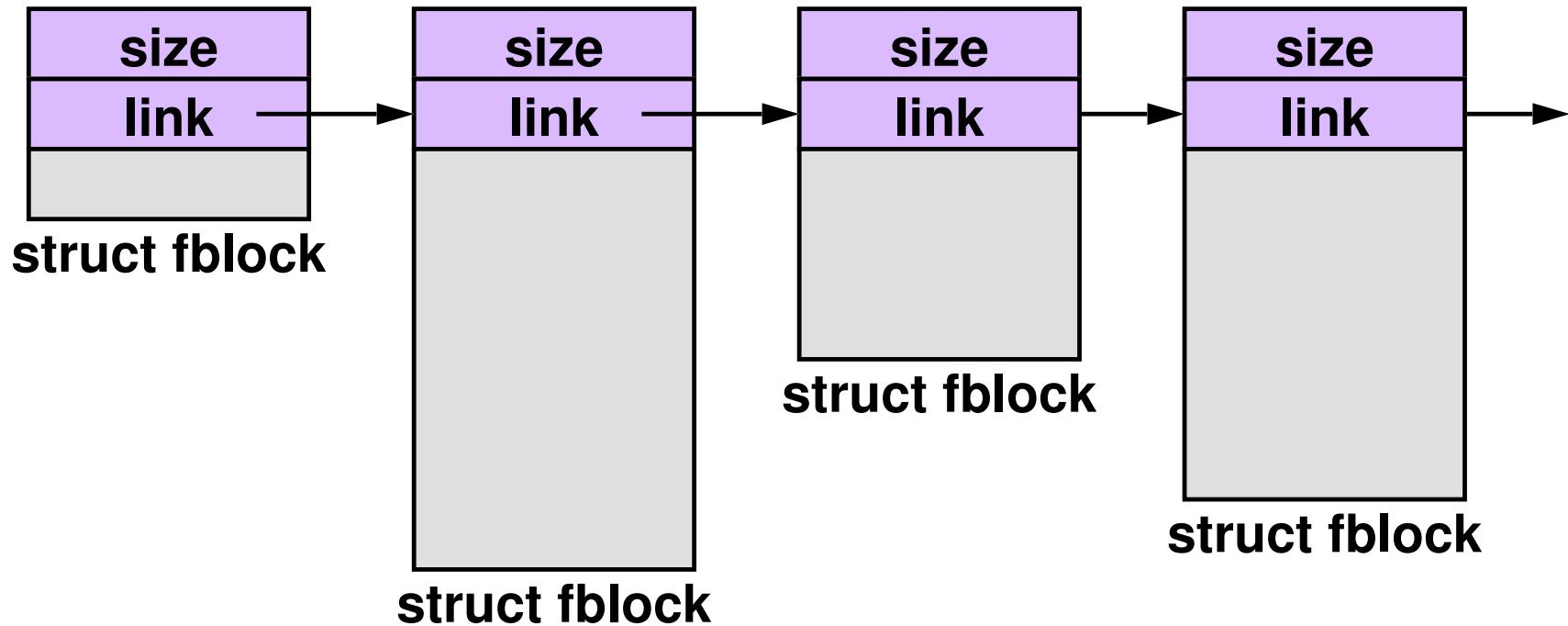


## **First-fit** vs. **best-fit** allocation

- studies have shown that first-fit works better
- best-fit tends to leave behind a large number of regions of memory that are too small to be useful
  - best-fit tends to create smallest left-over blocks!
- this is the general problem of **fragmentation**
  - **internal fragmentation:** unusable memory is contained within an allocated region (e.g., buddy system)
  - **external fragmentation:** unusable memory is separated into small blocks and is interspersed by allocated memory (e.g., best-fit)

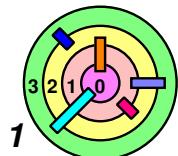


# Implementing First Fit: Data Structures

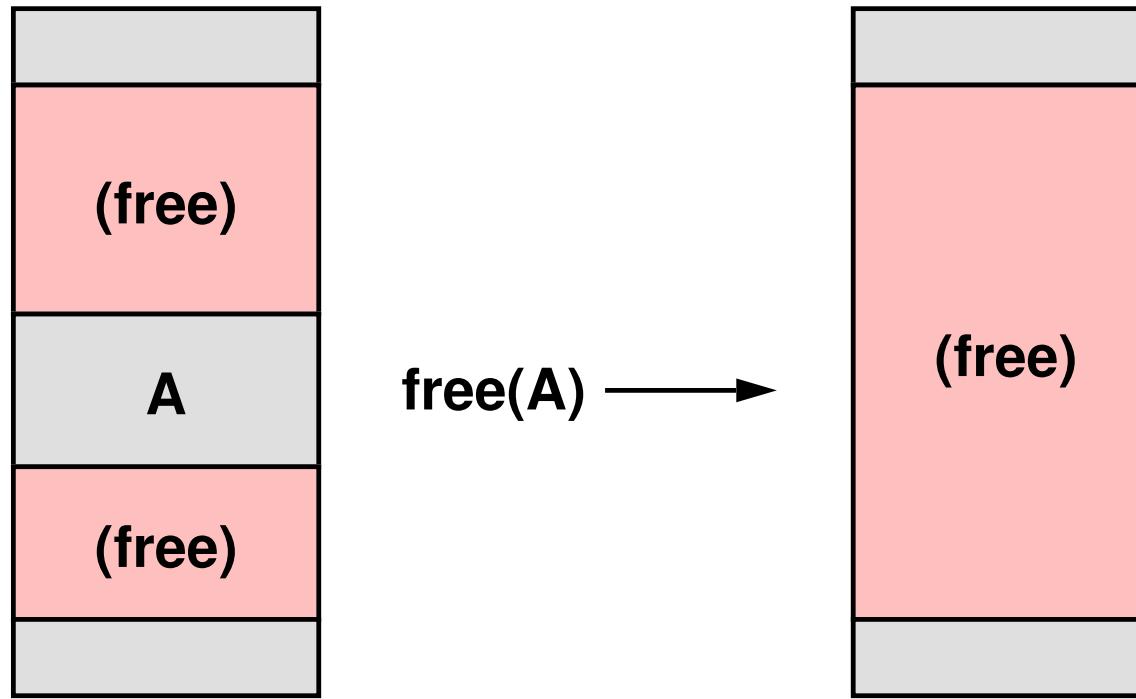


→ ***Free list:*** a linked list of *free blocks*

- *sorted* according to block *addresses*
  - no need to manage *allocated blocks*
- use a *doubly-linked list*
  - insertion and deletion are fast, i.e.,  $O(1)$ , once you know where to insert or delete

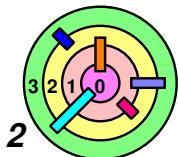


# Liberation of Storage



This is known as *coalescing*

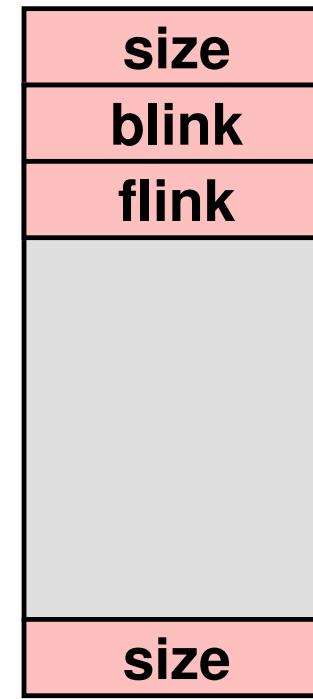
- in order to make coalescing possible, you need to know that *size* of the blocks above and below the block being freed
  - you also need to know if they are *allocated* or *free*



# Boundary Tags



Allocated Block

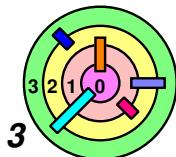


Free Block



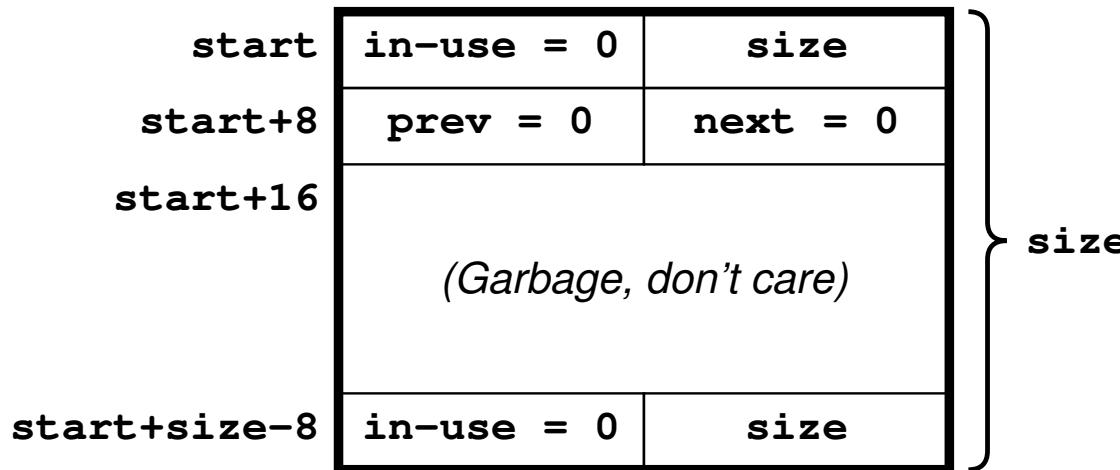
This is known as *coalescing*

- in order to make coalescing possible, you need to know that *size* of the blocks above and below the block being freed
  - you also need to know if they are *allocated* or *free*



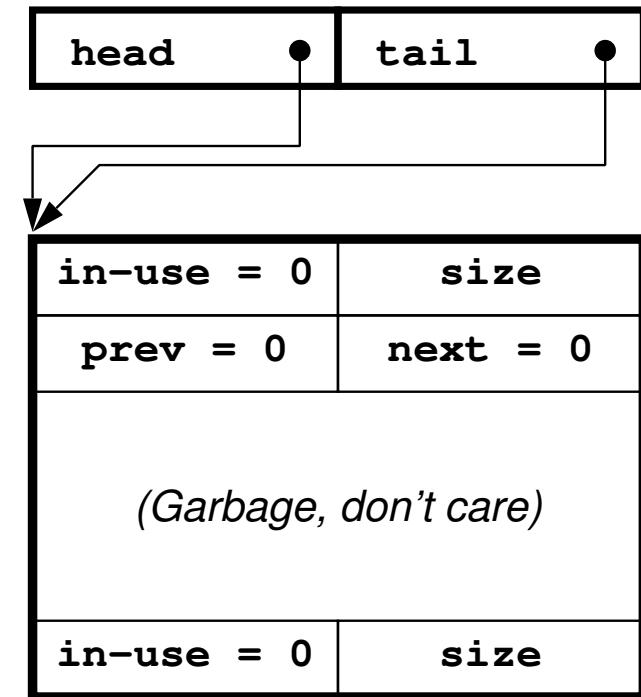
# Detailed Examples

## Free block

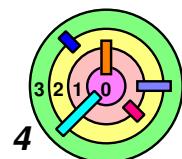
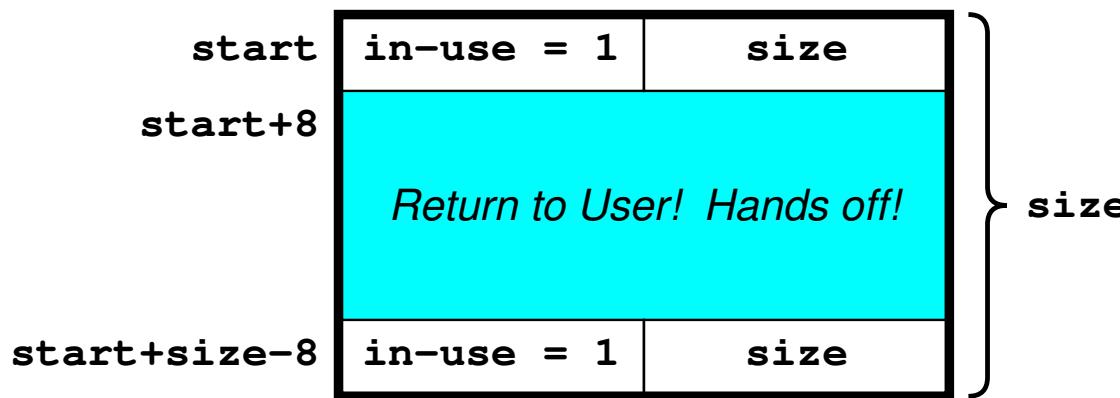


## Free list

Free List

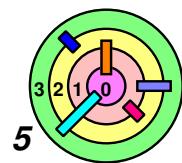
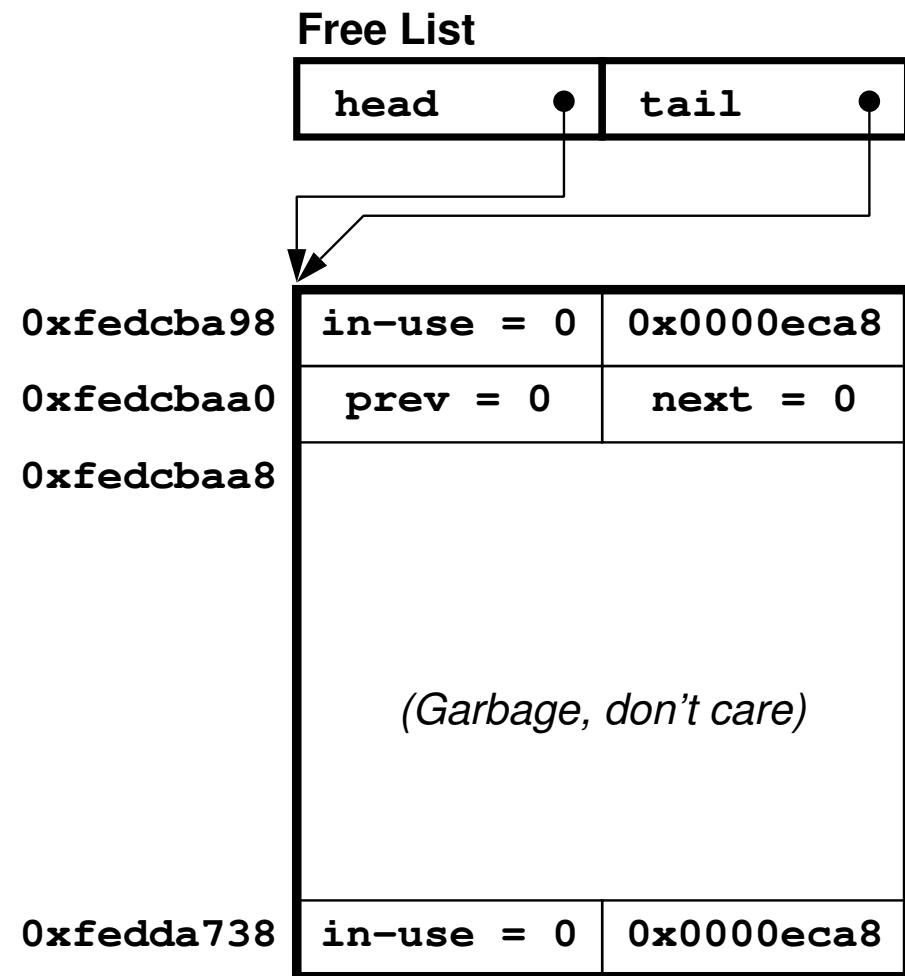


## In-use block



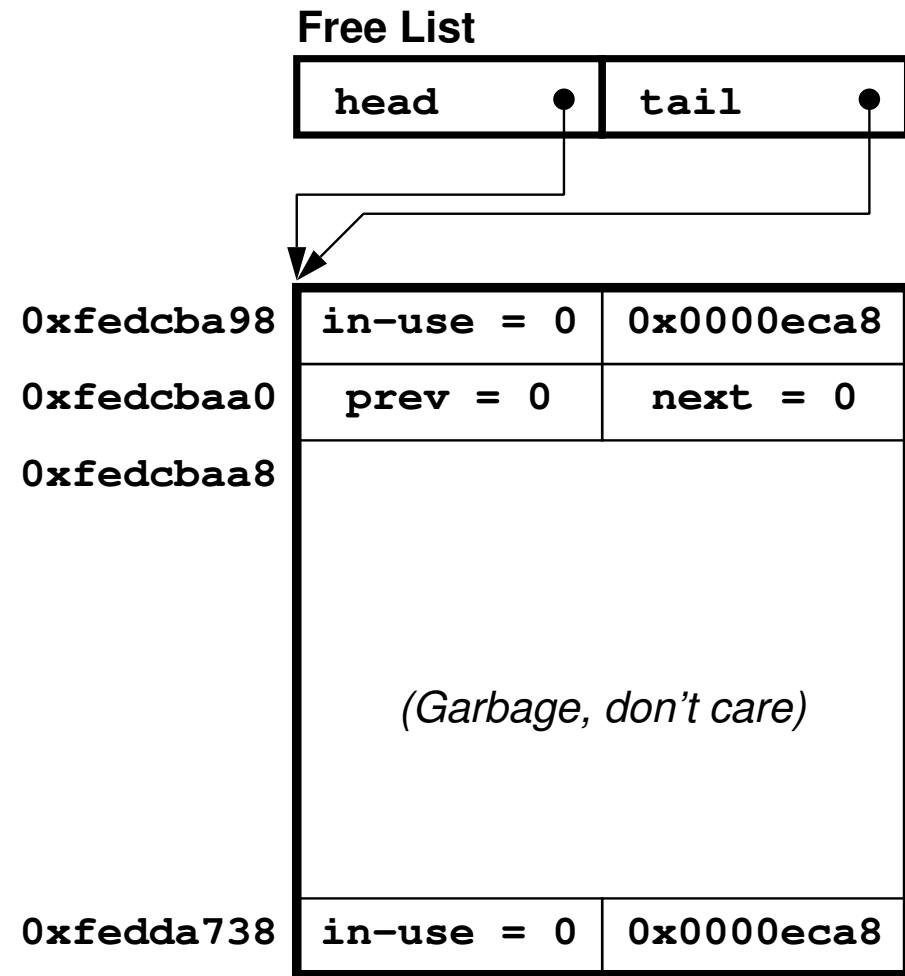
# malloc() Example

- Ex: Heap starts at **0xfedcba98**  
 and size of the heap is  
**0x0000eca8** (60,584) bytes  
 — the Free List contains one free  
 block and it looks like this:

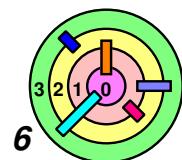


# malloc() Example

- Ex: Heap starts at **0xfedcba98** and size of the heap is **0x0000eca8** (60,584) bytes
- the Free List contains one free block and it looks like this:



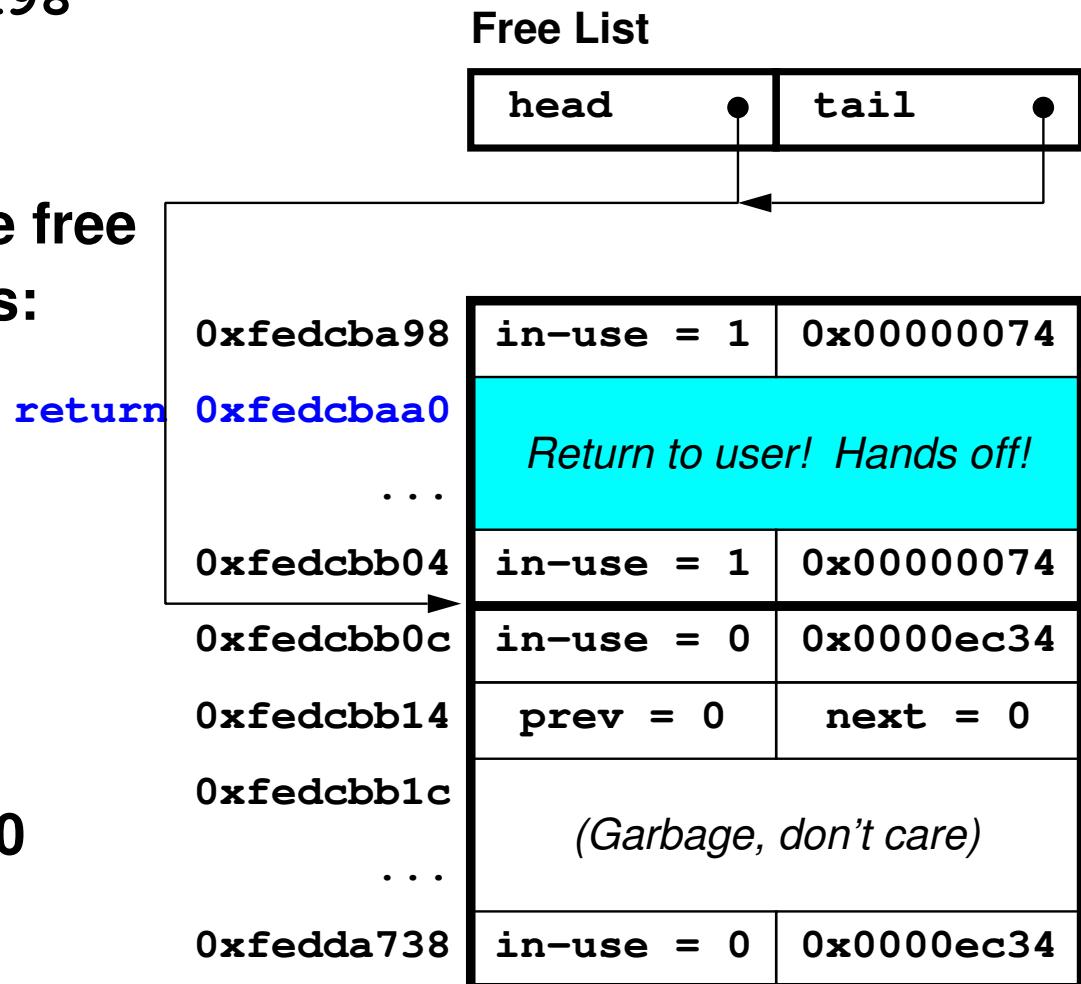
- Ex: Request block size is 100
- split the block into two
  - busy block size is 116
  - remaining free block size is  $60584 - 116 = 60468 = 0xec34$



# malloc () Example

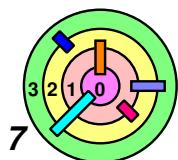
→ Ex: Heap starts at **0xfedcba98** and size of the heap is **0x0000eca8** (60,584) bytes

- the Free List contains one free block and it looks like this:



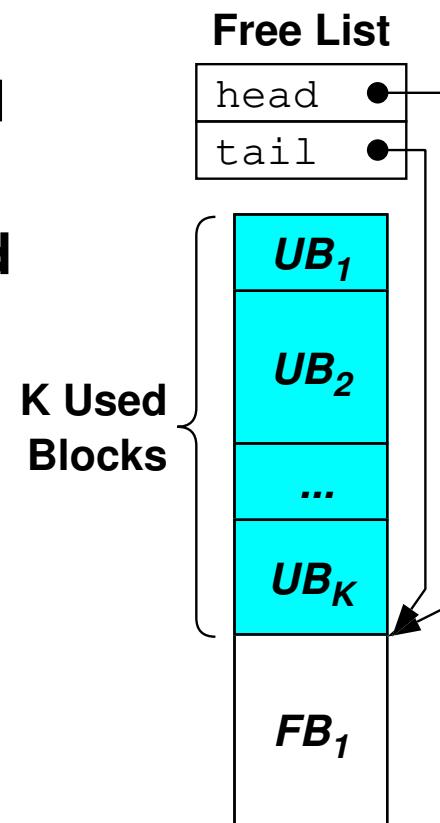
→ Ex: Request block size is 100

- split the block into two
- busy block size is 116
- remaining free block size is  $60584 - 116 = 60468 = 0xec34$



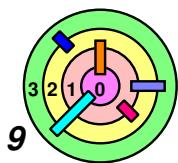
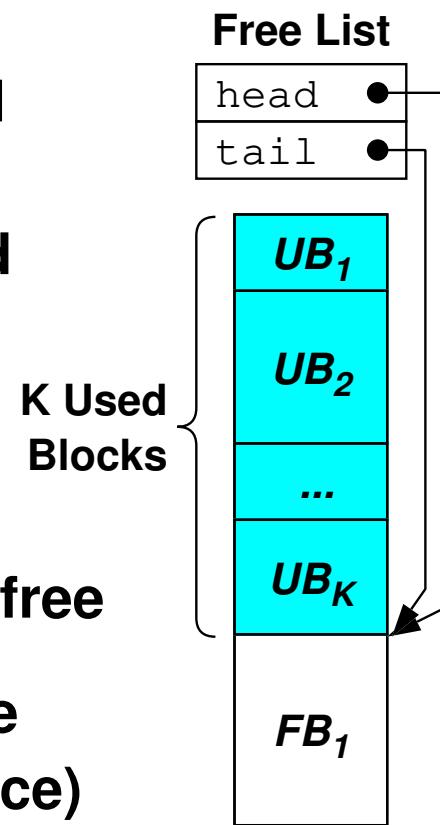
# free() Example

- After  $K$  blocks of memory have been allocated (and assume that none of them have been deallocated)
- in the memory layout, the first  $K$  blocks are used block, followed by one free block



# free() Example

- After **K** blocks of memory have been allocated (and assume that none of them have been deallocated)
  - in the memory layout, the first **K** blocks are used block, followed by one free block
- Memory blocks can be freed in any order
  - when a memory block is freed, we need to check if the blocks before it and after it are also free
- If neither of them are free, we just need to insert the newly freed block into the Free List (at the right place)
  - need to **search** the Free List to find insertion point
  - searching through a linear list is "slow",  $O(n)$
- Otherwise, we can **merge/coalesce** the block in question with neighboring free block(s)



# free() Example

→ Ex: `free(Y)`

- $Y-16$  tells you if the *previous* block is free or not
- $Y-8+z$  tells you if the *next* block is free or not
  - where  $z$  is what's in  $Y-4$

→ *Coalescing:*

- need to make sure that everything is consistent

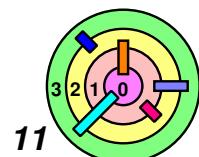
$Y-8-(*(Y-12))$

|         |                            |        |
|---------|----------------------------|--------|
|         | in-use=?                   | size   |
|         |                            | ?      |
| $Y-16$  | in-use=?                   | size   |
| $Y-8$   | in-use=1                   | size=z |
| $Y$     | Return to user! Hands off! |        |
|         | in-use=1                   | size=z |
| $Y-8+z$ | in-use=?                   | size   |
|         |                            | ?      |
|         | in-use=?                   | size   |

# free() Example

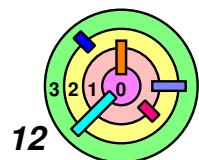
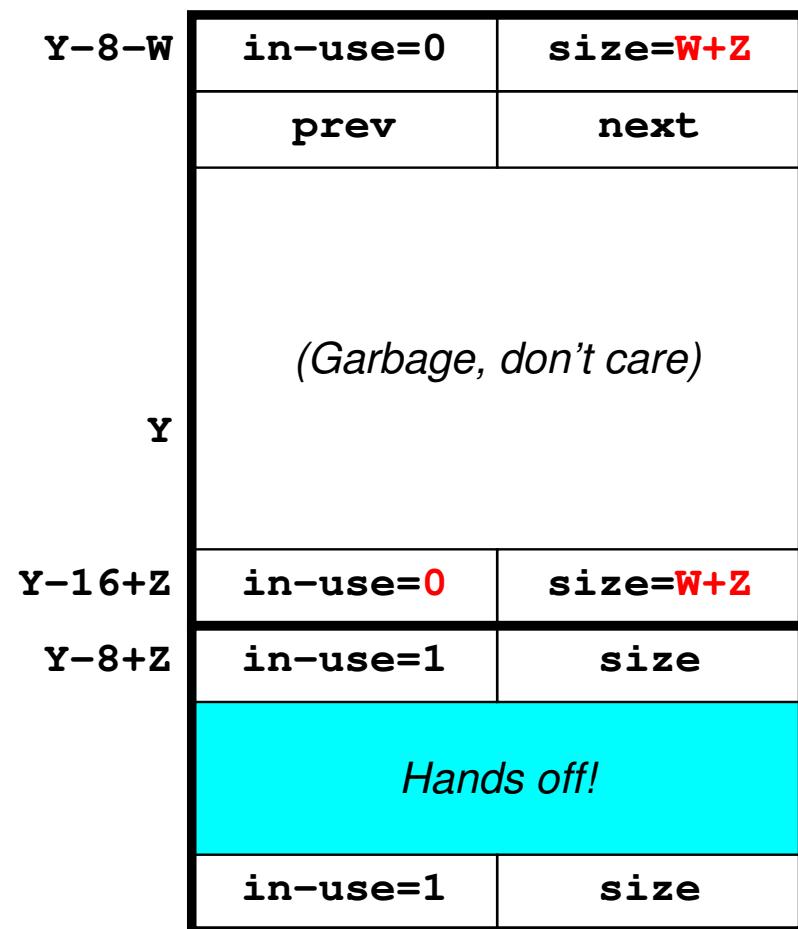
- Ex: `free(Y)` and *previous block is free and next block is busy*
- i.e.,  $Y-16$  is 0 and  $Y-8+z$  is 1
    - where  $z$  is what's in  $Y-4$  and  $w$  is what's in  $Y-12$
  - furthermore,  $Y-8-w$  is on the Free List
  - coalesce this block and the *previous* block

|                              |                                   |                     |
|------------------------------|-----------------------------------|---------------------|
| $Y-8-w$                      | <code>in-use=0</code>             | <code>size=w</code> |
|                              | <code>prev</code>                 | <code>next</code>   |
| <i>(Garbage, don't care)</i> |                                   |                     |
| $Y-16$                       | <code>in-use=0</code>             | <code>size=w</code> |
| $Y-8$                        | <code>in-use=1</code>             | <code>size=z</code> |
| $Y$                          | <i>Return to user! Hands off!</i> |                     |
| $Y-16+z$                     | <code>in-use=1</code>             | <code>size=z</code> |
| $Y-8+z$                      | <code>in-use=1</code>             | <code>size</code>   |
|                              | <i>Hands off!</i>                 |                     |
|                              | <code>in-use=1</code>             | <code>size</code>   |



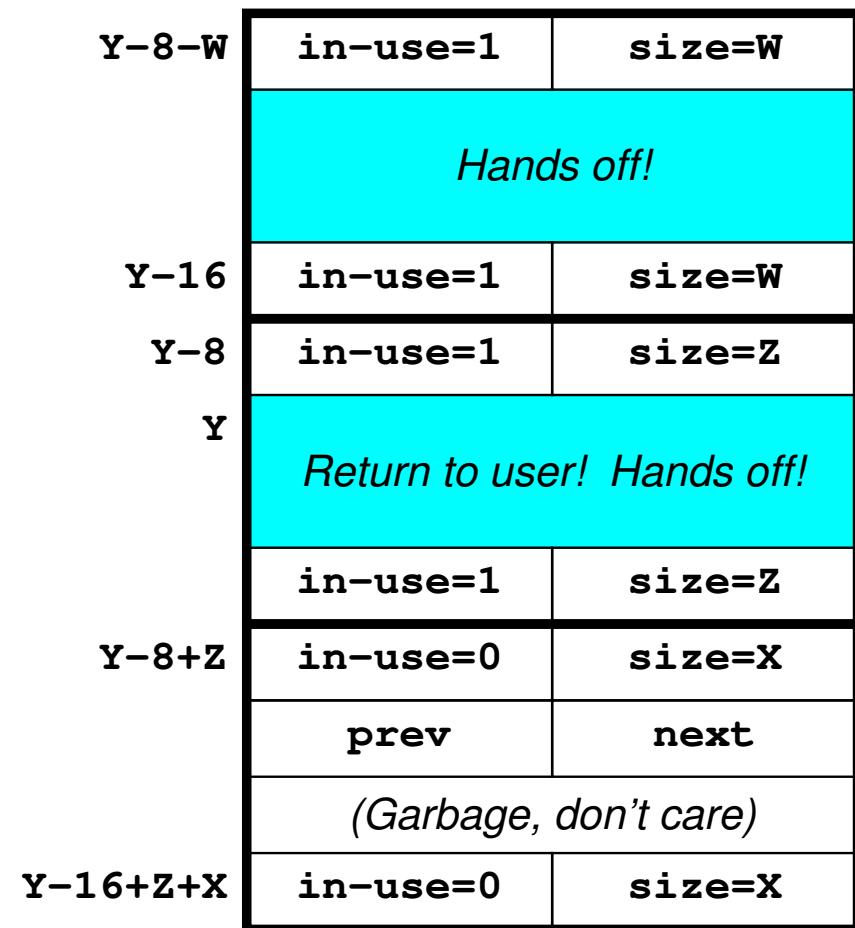
# free() Example

- Ex: `free(Y)` and *previous block is free and next block is busy*
- i.e.,  $Y-16$  is 0 and  $Y-8+z$  is 1
    - where  $z$  is what's in  $Y-4$  and  $w$  is what's in  $Y-12$
  - furthermore,  $Y-8-w$  is on the Free List
  - coalesce this block and the *previous* block
    - easy!
    - just change  $Y-12+z$  and  $Y-4-w$  to  $w+z$  and  $Y-16+z$  to 0
    - don't even need to change prev and next!



# free() Example

- Ex: `free(Y)` and *previous block is busy and next block is free*
- i.e.,  $Y-16$  is 1 and  $Y-8+z$  is 0
    - where  $z$  is what's in  $Y-4$  and  $x$  is what's in  $Y-4+z$
  - furthermore,  $Y-8+z$  is on the Free List
  - coalesce this block and the *next* block

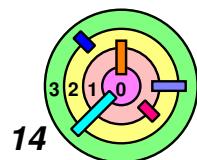
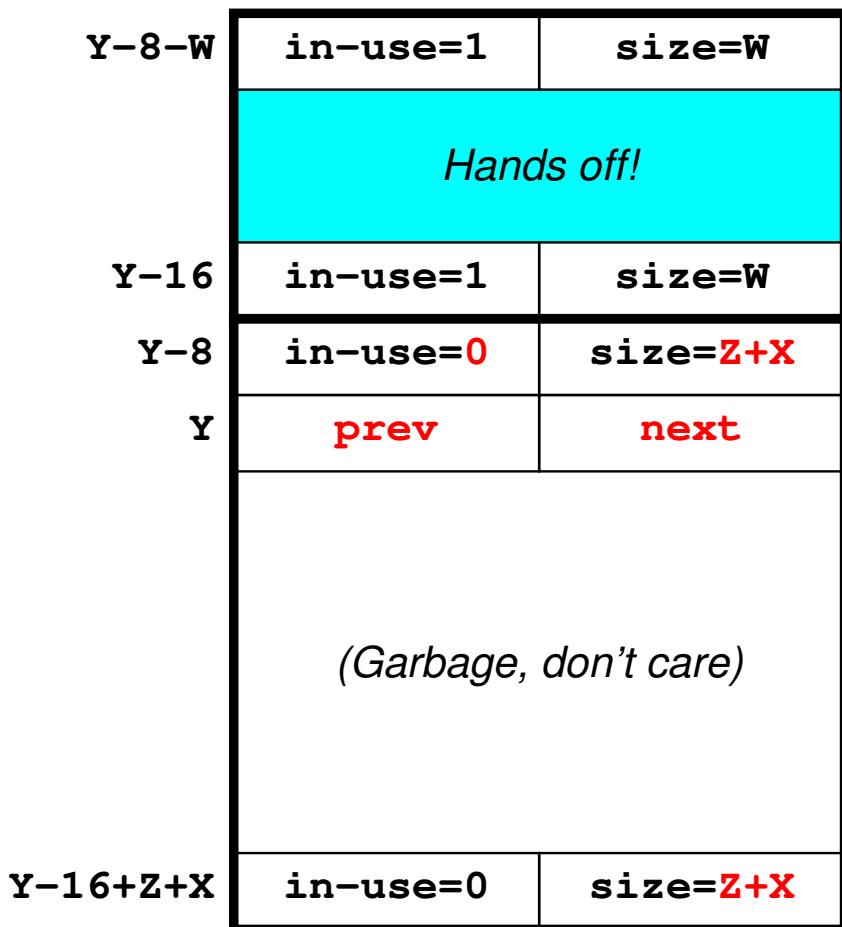


# free() Example



**Ex: `free(Y)` and *previous block is busy* and *next block is free***

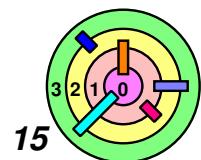
- i.e.,  $Y-16$  is 1 and  $Y-8+z$  is 0
  - where  $z$  is what's in  $Y-4$  and  $x$  is what's in  $Y-4+z$
- furthermore,  $Y-8+z$  is on the Free List
- coalesce this block and the *next* block
  - just change  $Y-4$  and  $Y-12+z+x$  to  $z+x$  and  $Y-8$  to 0
  - move prev and next pointers
  - adjust next field in previous block in Free List
  - adjust prev field in next block in Free List
  - may need to update where Free List points



# free() Example

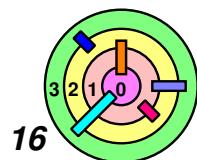
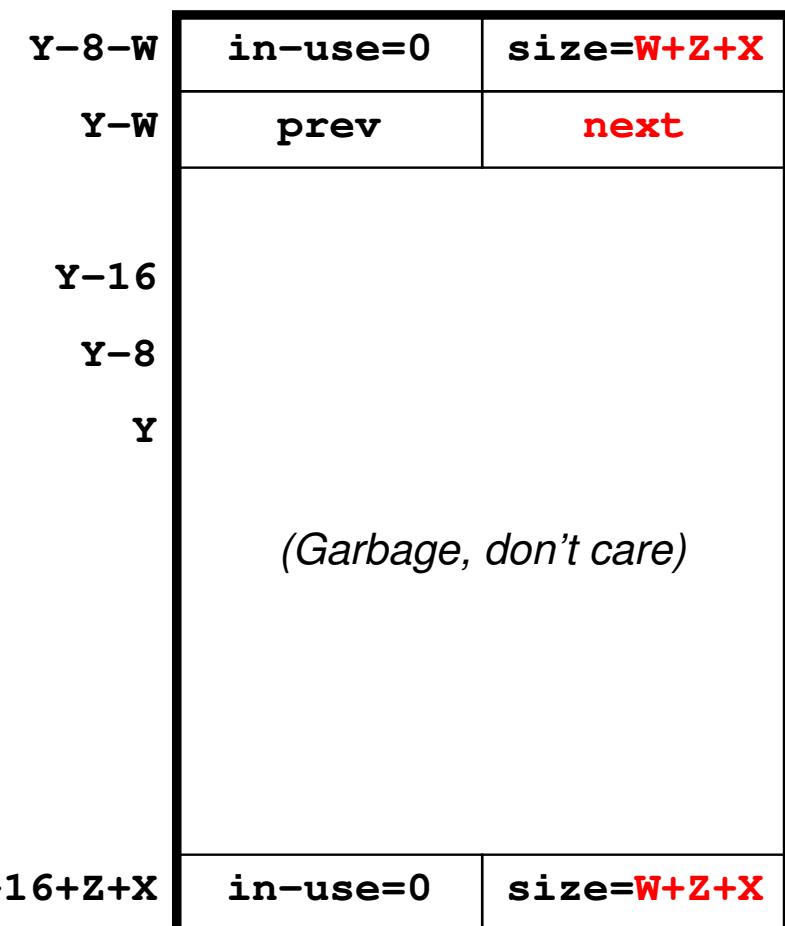
- Ex: `free(Y)` and *previous block is free and next block is also free*
- i.e.,  $Y-16$  is 0 and  $Y-8+z$  is 0
    - where  $z$  is what's in  $Y-4$ ,  $x$  is what's in  $Y-4+z$ , and  $w$  is what's in  $Y-12$
  - blocks starting at  $Y-8-w$  and  $Y-8+z$  are both on the Free List and next to and point at each other
  - coalesce all 3 blocks

|                              |                                   |                     |
|------------------------------|-----------------------------------|---------------------|
| $Y-8-w$                      | <code>in-use=0</code>             | <code>size=w</code> |
| $Y-w$                        | <code>prev</code>                 | $Y-8+z$             |
| <i>(Garbage, don't care)</i> |                                   |                     |
| $Y-16$                       | <code>in-use=0</code>             | <code>size=w</code> |
| $Y-8$                        | <code>in-use=1</code>             | <code>size=z</code> |
| $Y$                          | <i>Return to user! Hands off!</i> |                     |
| $Y-8+z$                      | <code>in-use=1</code>             | <code>size=z</code> |
| $Y+z$                        | <code>in-use=0</code>             | <code>size=x</code> |
| $Y-8-w$                      | <code>next</code>                 |                     |
| <i>(Garbage, don't care)</i> |                                   |                     |
|                              | <code>in-use=0</code>             | <code>size=x</code> |



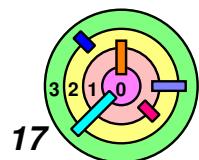
# free() Example

- Ex: `free(Y)` and *previous block is free* and *next block is also free*
- i.e.,  $Y-16$  is 0 and  $Y-8+z$  is 0
    - where  $z$  is what's in  $Y-4$ ,  
 $x$  is what's in  $Y-4+z$ , and  
 $w$  is what's in  $Y-12$
  - blocks starting at  $Y-8-w$  and  $Y-8+z$  are both on the Free List and next to and point at each other
  - coalesce all 3 blocks
    - just change  $Y-4-w$  and  $Y-12+z+x$  to  $w+z+x$
    - copy `next` from  $Y+z+4$  to  $Y-w+4$
    - adjust `prev` field in the new `next` block in Free List to point to  $Y-8-w$
    - may need to update where Free List points



# First-fit & Best-fit Algorithms

- Memory allocator must run fast
  - it does not check if the free list is in a consistent state
    - just like our warmup 1 assignment
- One bad bit in any memory allocator data structure can break the memory allocator code
  - if you write into a *boundary tag*, your program may die later in `malloc()` or `free()`
  - what would happen if you call `free()` twice on the same address?
  - user/application code can *corrupt the memory allocation chain* easily
    - the result can lead to *segmentation faults*
    - unfortunately, the corruption can *stay hidden* for a long time and *eventually* lead to a segmentation fault
      - ◆ memory corruption bugs are very difficult to debug

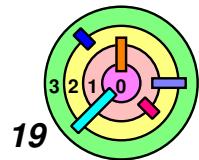


# First-fit Algorithm

- Let  $n$  be the number of free blocks on the free list
  - in the worst case, `malloc()` is  $O(n)$
  - in the worst case, `free(ptr)` is  $O(n)$ 
    - occurs when the blocks around the block containing `ptr` are both in-use
- Such performance is unacceptable in the kernel
  - it is desirable that the kernel's worst-case performance has a bound

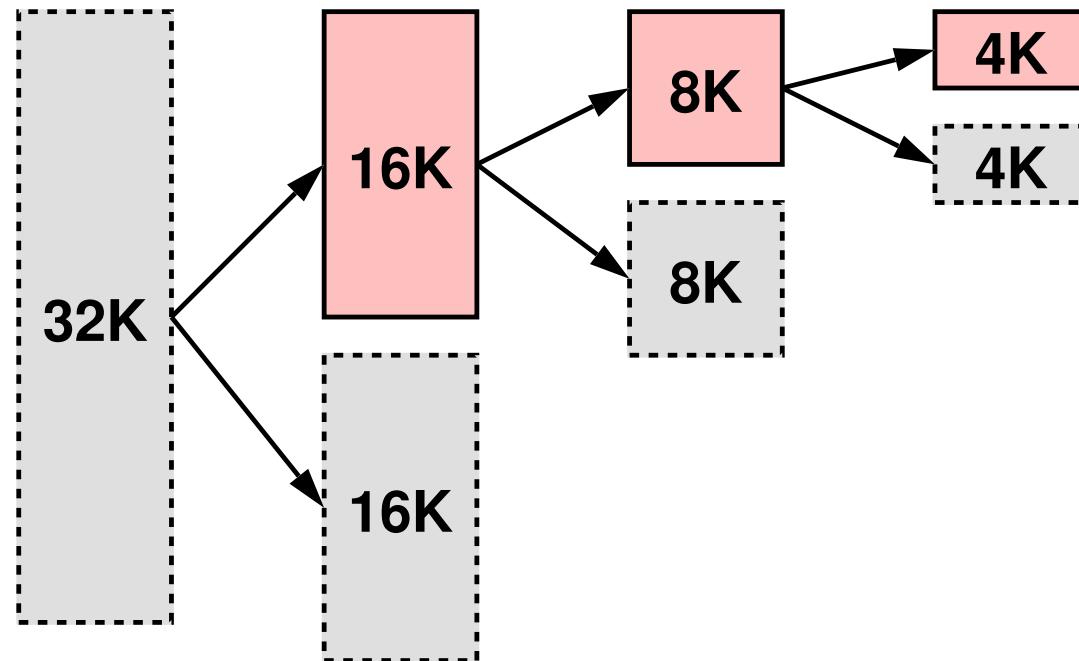
# 3.3 Dynamic Storage Allocation

- ➡ Best-fit & First-fit Algorithms
- ➡ *Buddy System*
- ➡ Slab Allocation



# Buddy Lists

Ex: malloc(4000)



- blocks get evenly divided into two blocks that are buddies with each other
  - can only merge with your buddy if your buddy is also free
- *internal fragmentation*
  - Ex: malloc(4000)
  - return a 4K block

# Buddy Systems



**Faster memory allocation system (at the cost of more fragmentation, internal fragmentation)**

- restrict block size to be a power of 2

1) all blocks of size  $2^k$  start at location  $x$  where  $x \bmod 2^k = 0$

2) given a block starting at location  $x$  such that  $x \bmod 2^k = 0$

    ◊  $\text{BUDDY}_k(x) = x + 2^k$  if  $x \bmod 2^{k+1} = 0$

    ◊  $\text{BUDDY}_k(x) = x - 2^k$  if  $x \bmod 2^{k+1} = 2^k$

    ◊ Ex:  $\text{BUDDY}_2(1010100) = 1010000$

3) only buddies can be merged

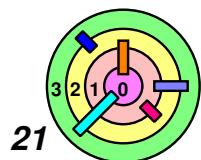
4) try to coalesce buddies when storage is deallocated

○  $k$  different available block lists, one for each block size

○ When request a block of size  $2^k$  and none is available:

1) split smallest block  $2^j > 2^k$  into a pair of blocks of size  $2^{j-1}$

2) place block on appropriate free list and try again



# Buddy Systems



## Data Structure

1) doubly-linked list (not circular) **FREE** list indexed by  $k$

- ◊ links stored in actual blocks
- ◊ **FREE**[ $k$ ] points to first available block of size  $2^k$

2) each block contains

- ◊ **in-use** bit
- ◊ **size**
- ◊ **NEXT** and **PREV** links for **FREE** list

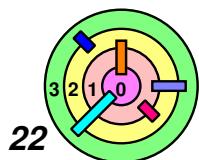
— lots of details

- read **weenix** source code for its "page allocator"



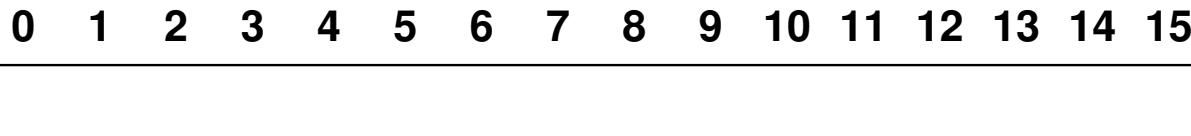
Can get greater variety in block sizes using Fibonacci sequence of block sizes so  $b_j = b_{j-1} + b_{j-2}$

— ratio of successive block sizes is  $2/3$  instead of  $1/2$



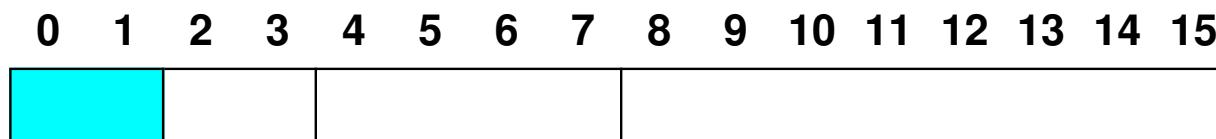
# High-level Example of Buddy Algorithm

- Ex: 16 "pages" (minimum allocation is 1 page)



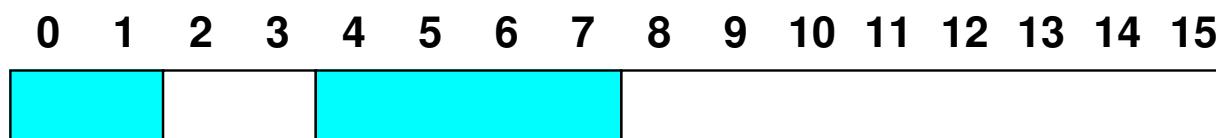
| k | free[k]  |
|---|----------|
| 0 | $\Omega$ |
| 1 | $\Omega$ |
| 2 | $\Omega$ |
| 3 | $\Omega$ |
| 4 | 0        |

- 1) allocate a block of size 2

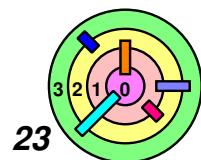


| k | free[k]    |
|---|------------|
| 0 | $\Omega$   |
| 1 | ☒ 2        |
| 2 | ☒ 4        |
| 3 | ☒ 8        |
| 4 | ☒ $\Omega$ |

- 2) allocate a block of size 4



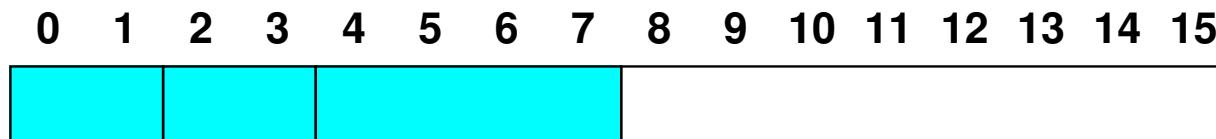
| k | free[k]      |
|---|--------------|
| 0 | $\Omega$     |
| 1 | ☒ 2          |
| 2 | ☒ 4 $\Omega$ |
| 3 | ☒ 8          |
| 4 | ☒ $\Omega$   |



# High-level Example of Buddy Algorithm

- Ex: 16 "pages" (minimum allocation is 1 page)

3) allocate a block of size 2



4) allocate a block of size 2

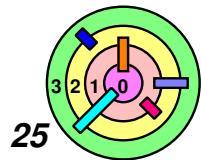


| k | free[k] |
|---|---------|
| 0 | Ω       |
| 1 | ☒ ✗ Ω   |
| 2 | ☒ ✗ 4 Ω |
| 3 | ☒ 8     |
| 4 | ☒ Ω     |

| k | free[k] |
|---|---------|
| 0 | Ω       |
| 1 | ☒ ✗ 10  |
| 2 | ☒ ✗ 12  |
| 3 | ☒ 8 Ω   |
| 4 | ☒ Ω     |

# 3.3 Dynamic Storage Allocation

- ➡ Best-fit & First-fit Algorithms
- ➡ Buddy System
- ➡ *Slab Allocation*



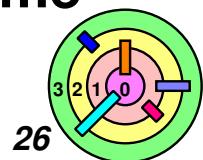
# Slab Allocation

→ Objects are allocated and freed frequently

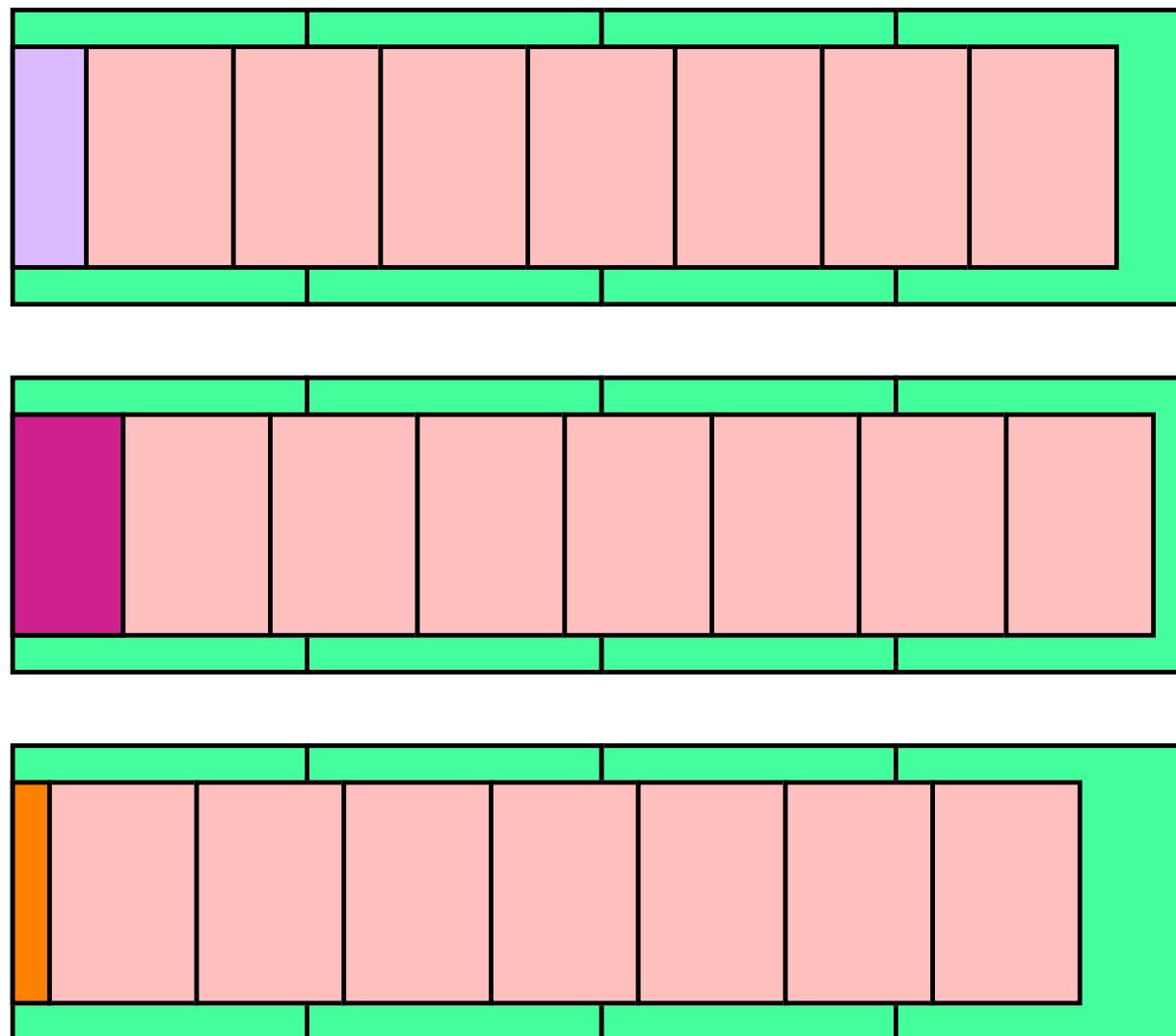
- allocation involves
  - finding an appropriate-sized storage
  - initialize it
    - ◊ pointers need to point at the right places
    - ◊ may even need to initialize synchronization data structures
- deallocation involves
  - tearing down the data structures
  - freeing the storage
- lots of "overhead"

→ Difficulties with dynamic storage allocation

- you cannot predict what an application will ask for
- but it's not true for the kernel
  - e.g., can allocate a slab of process control blocks at a time
    - ◊ return one of them from a slab



# Slab Allocation



— see weenix kernel code!

# Slab Allocation

## → **Slab Allocation**

- sets up a separate cache for each type of object to be managed
- contiguous sets of pages called **slabs**, allocated to hold objects
  - we will cover "pages" later, won't get into too much detail now

## → Whenever a **slab** is allocated, a constructor is called to initialize all the objects it holds

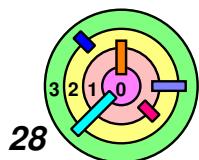
- this is where you pay for initialization, but it's done in a **batch**

## → As **objects** are being allocated, they are taken from the set of existing slabs in the cache

- objects are considered "preallocated" since they have all been initialized already

## → As **objects** are being freed, they are simply marked as free

- don't have to free up storage
- when appropriate can free up an entire slab



# 3.4 Linking & Loading

→ ***Static Linking & Loading***

→ **Shared Libraries**

# Remember This?

```

main:
→ pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    subl $8, %esp
    ...
    pushl $1
    movl -12(%ebp), %eax
    pushl %eax
    call sub
    addl $8, %esp
    movl %eax, -16(%ebp)
    ...
    addl $8, %esp
    movl $0, %eax
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret

```

**set up stack frame**

**push args**

**pop args;  
get result**

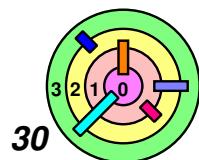
**set return  
value and  
restore frame**



```

int main() {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

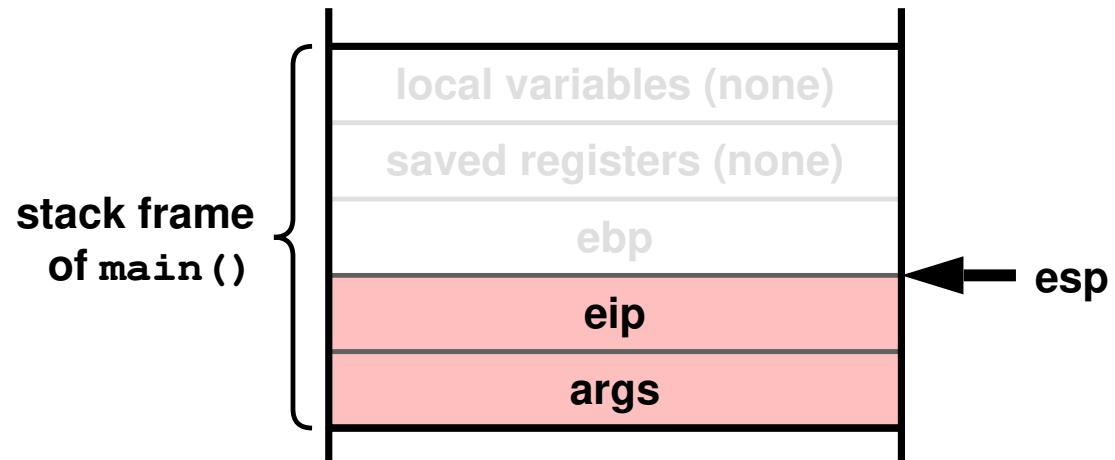
```



# Something Simpler

```
int main(int argc,
         char *[])
{
    return(argc);
}
```

```
main:
→ pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
```



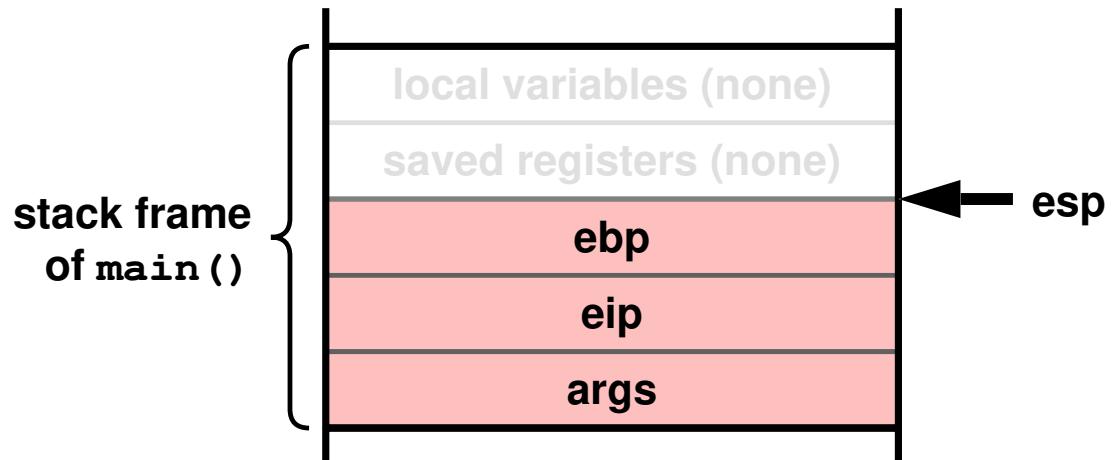
} set up stack frame

} set return value and restore frame

→ Does location matter?

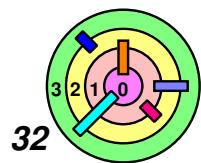
# Something Simpler

```
int main(int argc,
         char *[])
{
    return(argc);
}
```



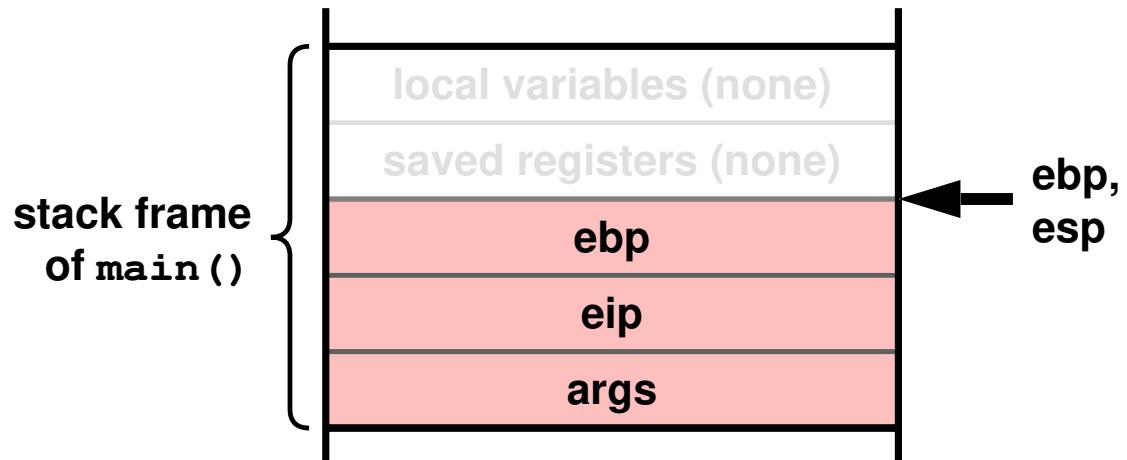
```
main:
    pushl %ebp
    movl %esp, %ebp } set up stack frame
    movl 8(%ebp), %eax
    movl %ebp, %esp } set return value and
    popl %ebp        restore frame
    ret
```

→ Does location matter?



# Something Simpler

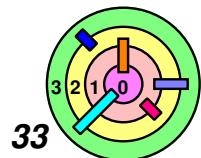
```
int main(int argc,
         char *[])
{
    return(argc);
}
```



`main:`

|                                                                                                                                                                                   |                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| <code>pushl %ebp</code><br><code>movl %esp, %ebp</code><br><b>→</b> <code>movl 8(%ebp), %eax</code><br><code>movl %ebp, %esp</code><br><code>popl %ebp</code><br><code>ret</code> | { set up stack frame<br><br>{ set return value and restore frame |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|

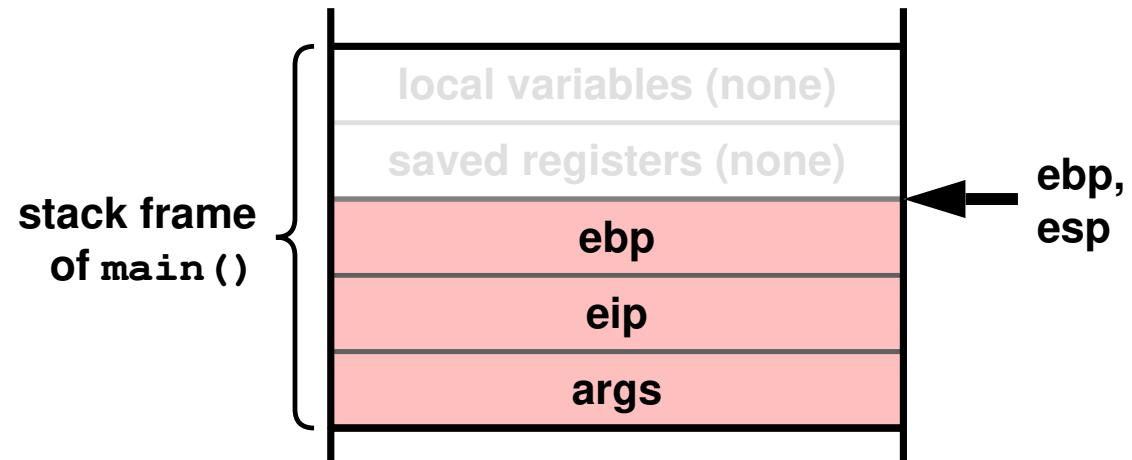
→ Does location matter?



# Something Simpler

```
int main(int argc,
         char *[])
{
    return(argc);
}
```

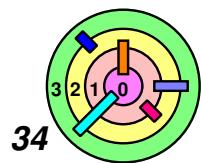
```
main:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
```



} set up stack frame

} set return value and restore frame

→ Does location matter?

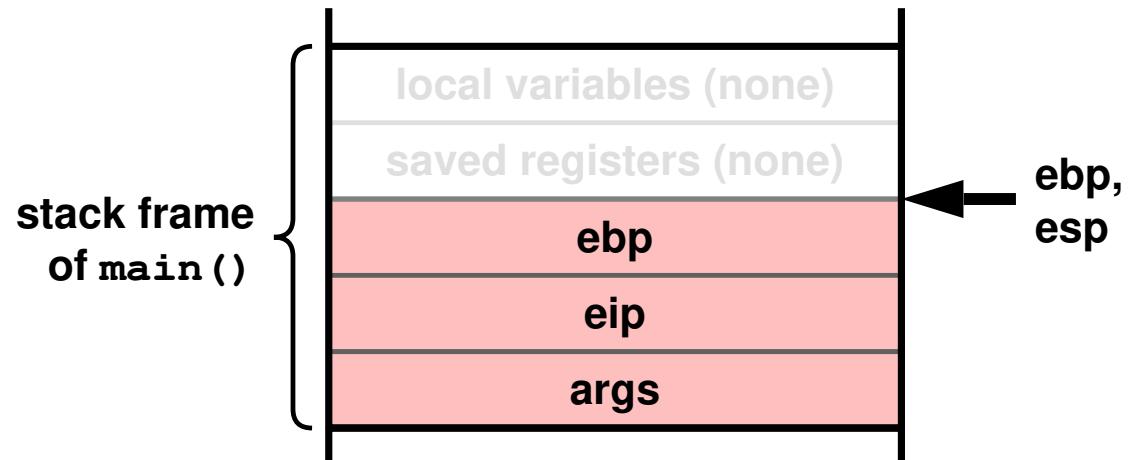


# Something Simpler

```
int main(int argc,
         char *[])
{
    return(argc);
}
```

**main:**

|                                                                                          |                                                                 |
|------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| <pre>pushl %ebp movl %esp, %ebp movl 8(%ebp), %eax movl %ebp, %esp → popl %ebp ret</pre> | { set up stack frame       } set return value and restore frame |
|------------------------------------------------------------------------------------------|-----------------------------------------------------------------|



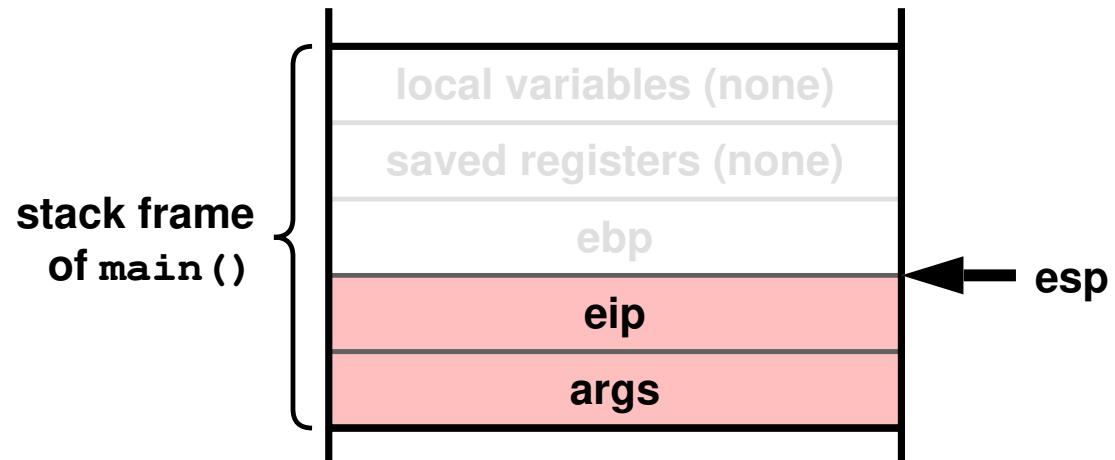
→ Does location matter?

# Something Simpler

```
int main(int argc,
         char *[])
{
    return(argc);
}
```

**main:**

|                                                                                               |                                                                 |
|-----------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| <pre>pushl %ebp movl %esp, %ebp movl 8(%ebp), %eax movl %ebp, %esp popl %ebp <b>ret</b></pre> | { set up stack frame       } set return value and restore frame |
|-----------------------------------------------------------------------------------------------|-----------------------------------------------------------------|



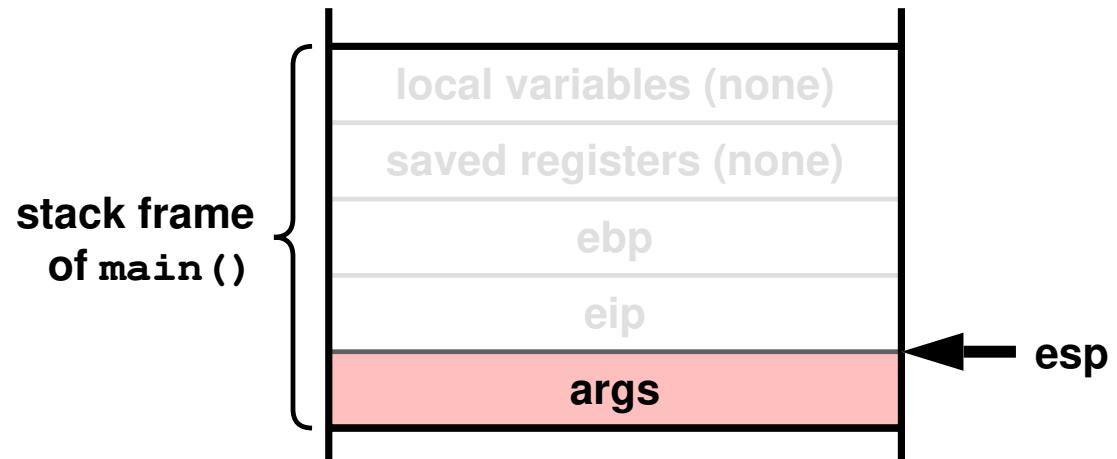
→ Does location matter?

# Something Simpler

```
int main(int argc,
         char *[])
{
    return(argc);
}
```

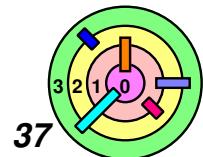
**main:**

|                    |                                      |
|--------------------|--------------------------------------|
| pushl %ebp         | } set up stack frame                 |
| movl %esp, %ebp    |                                      |
| movl 8(%ebp), %eax | } set return value and restore frame |
| movl %ebp, %esp    |                                      |
| popl %ebp          |                                      |
| ret                |                                      |



**Does location matter?**

- if everything can be accessed relative to the **frame pointer**, then you don't need to know the actual address of an object
  - just use **relative-addresses** to **access variables**
  - the **code** is also **location-independent**



# Location Matters ...

```

int x = 6;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = x;
    subr(y);
    return(0);
}

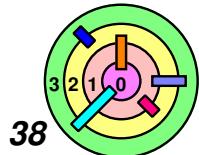
```

```

void subr(int i) {
    printf("i = %d\n", i);
}

```

- Why does it matter here?
- need to put the *address* of x into ax
    - *what* is the address of x?
      - ◊ *when* do you know?
    - remember, both x and ax are in the *data segment*
    - *who* would put the actual value into ax?
  - also need to put the *address* of subr into main()



# Coping

## → Done in two steps

- *compiler* figure out *how many bytes* each object is
  - assign them temporary locations
- *linker* figures out *where* each object is and lays out the entire address space
  - functions, global variables, and more

## → *Relocation*

- modify internal references in memory depending on where module is expected to be *loaded*
  - one of the *exec* system calls loads a program into memory
    - ◊ everything is laid out carefully in memory
- modules requiring relocation are said to be *relocatable*
- the act of modifying such a module to *resolve these references* is called *relocation*
- the program that performs relocation is called a *linker*

textbook  
is wrong

# Linker

→ Two main functions of a *linker*

- 1) *relocation*
- 2) *symbol resolution*
  - ◆ find unresolved symbols and figure out how to resolve them

→ A *loader* loads a program into memory

- "unfolds" a program from disk into memory and "transfer control" to it (i.e., "executes" it)
- a "relocating loader" may perform additional relocation
  - but that's *dynamic* linking

# A Slight Revision

```

extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}

```

**main.c**

```

#include <stdio.h>
int x;

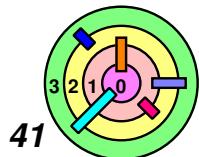
void subr(int i) {
    printf("i = %d\n", i);
}

```

**subr.c**

```
% gcc -o prog main.c subr.c
```

- **main.c** is compiled into **main.o**
- **subr.c** is compiled into **subr.o**
- **ld** is then invoked to combine them into **prog**
  - **ld** knows where to find **printf()**
  - **prog** can be loaded into memory through one of the **exec** system calls



# A Slight Revision

```

extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}

```

**main.c**

```

#include <stdio.h>
int x;

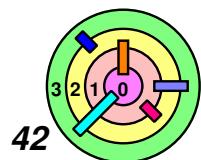
void subr(int i) {
    printf("i = %d\n", i);
}

```

**subr.c**

```
% gcc -o prog main.c subr.c
```

- how does **ld** decides what needs to be done?
- **main.c** contains undefined references to **x** and **subr()**
  - instructions for doing this are provided in **main.o**
- later on, when the actual locations for these are determined, **ld** will modify them when **main.o** is **copied** into **prog**



# A Slight Revision

```

extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}

```

**main.c**

```

#include <stdio.h>
int x;

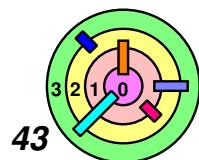
void subr(int i) {
    printf("i = %d\n", i);
}

```

**subr.c**

```
% gcc -o prog main.c subr.c
```

- = **main.o** must contains a list of external symbols, along with their types, and instructions for updating this code



# Can Also Use A Header File

```
#include "subr.c";
int *ax = &x;

int main( ) {
    int y = *ax;
    subr(y);
    return(0);
}
```

main.c

```
extern int x;
void subr(int);
```

subr.h

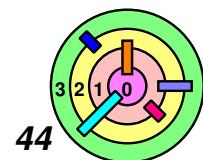
```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```

subr.c

→ A **header** file typically contains

- declaration/definition of ***data structures***
- declaration of ***exported symbols***

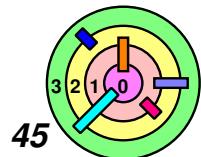


# main.s

| Offset   | Op                                                                  | Arg     |
|----------|---------------------------------------------------------------------|---------|
| 0:       | .data ; what follows is initialized data                            |         |
| 0:       | .globl ax ; ax is global: it may be used<br>; by others             |         |
| 0: ax:   |                                                                     |         |
| 0:       | .long x                                                             |         |
| 4:       |                                                                     |         |
| 0:       | .text ; offset restarts; what follows is<br>; text (read-only code) |         |
| 0:       | .globl main                                                         | F20-Q15 |
| 0: main: |                                                                     |         |
| 0:       | pushl %ebp ; save the frame pointer                                 |         |
| 1:       | movl %esp,%ebp ; point to                                           |         |
| 3:       | subl \$4,%esp ; make space                                          |         |
| 6:       | movl ax,%eax ; put content                                          |         |
| 11:      | movl (%eax),%eax ; put *x                                           |         |
| 13:      | movl %eax,-4(%ebp) ; store                                          |         |
| 16:      | pushl -4(%ebp) ; push y on                                          |         |
| 19:      | call subr                                                           |         |
| 24:      | addl \$4,%esp ; remove y f                                          |         |
| 27:      | movl \$0,%eax ; set return                                          |         |
| 31:      | movl %ebp,%esp ; restore                                            |         |
| 33:      | popl %ebp ; pop frame pointer                                       |         |
| 35:      | ret                                                                 |         |

```
extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}
```



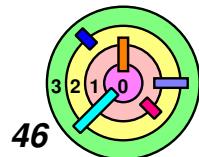
# main.s

| Offset   | Op                                                      | Arg |
|----------|---------------------------------------------------------|-----|
| 0:       | .data ; what follows is initialized data                |     |
| 0:       | .globl ax ; ax is global: it may be used<br>; by others |     |
| 0: ax:   |                                                         |     |
| 0:       | .long x                                                 |     |
| 4:       |                                                         |     |
| 0:       | .text ; offset restarts; w<br>; text (read-only co      |     |
| 0:       | .globl main                                             |     |
| 0: main: |                                                         |     |
| 0:       | pushl %ebp ; save the frame pointer                     |     |
| 1:       | movl %esp,%ebp ; point to                               |     |
| 3:       | subl \$4,%esp ; make space                              |     |
| 6:       | movl ax,%eax ; put conten                               |     |
| 11:      | movl (%eax),%eax ; put *x                               |     |
| 13:      | movl %eax,-4(%ebp) ; stor                               |     |
| 16:      | pushl -4(%ebp) ; push y on                              |     |
| 19:      | call subr                                               |     |
| 24:      | addl \$4,%esp ; remove y f                              |     |
| 27:      | movl \$0,%eax ; set return                              |     |
| 31:      | movl %ebp,%esp ; restore                                |     |
| 33:      | popl %ebp ; pop frame pointer                           |     |
| 35:      | ret                                                     |     |

What follows goes into the  
*data* segment

```
extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}
```



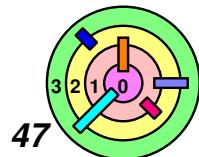
# main.s

| Offset   | Op                                                 | Arg |
|----------|----------------------------------------------------|-----|
| 0:       | .data ; what follows is initialized data           |     |
| 0:       | .globl ax ; ax is global: it may be used by others |     |
| 0: ax:   |                                                    |     |
| 0:       | .long x                                            |     |
| 4:       |                                                    |     |
| 0:       | .text ; offset restarts; w ; text (read-only code) |     |
| 0:       | .globl main                                        |     |
| 0: main: |                                                    |     |
| 0:       | pushl %ebp ; save the frame pointer                |     |
| 1:       | movl %esp,%ebp ; point to stack                    |     |
| 3:       | subl \$4,%esp ; make space                         |     |
| 6:       | movl ax,%eax ; put content                         |     |
| 11:      | movl (%eax),%eax ; put *x                          |     |
| 13:      | movl %eax,-4(%ebp) ; store                         |     |
| 16:      | pushl -4(%ebp) ; push y onto                       |     |
| 19:      | call subr                                          |     |
| 24:      | addl \$4,%esp ; remove y from                      |     |
| 27:      | movl \$0,%eax ; set return                         |     |
| 31:      | movl %ebp,%esp ; restore stack pointer             |     |
| 33:      | popl %ebp ; pop frame pointer                      |     |
| 35:      | ret                                                |     |

What follows goes into the **text** segment

```
extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}
```



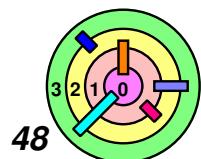
# main.s

| Offset   | Op                                                      | Arg |
|----------|---------------------------------------------------------|-----|
| 0:       | .data ; what follows is initialized data                |     |
| 0:       | .globl ax ; ax is global: it may be used<br>; by others |     |
| 0: ax:   |                                                         |     |
| 0:       | .long x                                                 |     |
| 4:       |                                                         |     |
| 0:       | .text ; offset restarts; w<br>; text (read-only co      |     |
| 0:       | .globl main                                             |     |
| 0: main: |                                                         |     |
| 0:       | pushl %ebp ; save the frame pointer                     |     |
| 1:       | movl %esp,%ebp ; point to                               |     |
| 3:       | subl \$4,%esp ; make space                              |     |
| 6:       | movl ax,%eax ; put conten                               |     |
| 11:      | movl (%eax),%eax ; put *x                               |     |
| 13:      | movl %eax,-4(%ebp) ; stor                               |     |
| 16:      | pushl -4(%ebp) ; push y on                              |     |
| 19:      | call subr                                               |     |
| 24:      | addl \$4,%esp ; remove y f                              |     |
| 27:      | movl \$0,%eax ; set return                              |     |
| 31:      | movl %ebp,%esp ; restore                                |     |
| 33:      | popl %ebp ; pop frame pointer                           |     |
| 35:      | ret                                                     |     |

offset got restarted because  
*segments are relocatable*

```
extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}
```



# main.s

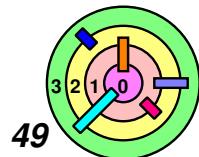
| Offset   | Op                                                 | Arg |
|----------|----------------------------------------------------|-----|
| 0:       | .data ; what follows is initialized data           |     |
| 0:       | .globl ax ; ax is global: it may be used by others |     |
| 0: ax:   |                                                    |     |
| 0:       | .long x                                            |     |
| 4:       |                                                    |     |
| 0:       | .text ; offset restarts; w ; text (read-only code) |     |
| 0:       | .globl main                                        |     |
| 0: main: |                                                    |     |
| 0:       | pushl %ebp ; save the frame pointer                |     |
| 1:       | movl %esp,%ebp ; point to stack                    |     |
| 3:       | subl \$4,%esp ; make space                         |     |
| 6:       | movl ax,%eax ; put content                         |     |
| 11:      | movl (%eax),%eax ; put *x                          |     |
| 13:      | movl %eax,-4(%ebp) ; store                         |     |
| 16:      | pushl -4(%ebp) ; push y onto                       |     |
| 19:      | call subr                                          |     |
| 24:      | addl \$4,%esp ; remove y from                      |     |
| 27:      | movl \$0,%eax ; set return                         |     |
| 31:      | movl %ebp,%esp ; restore stack pointer             |     |
| 33:      | popl %ebp ; pop frame pointer                      |     |
| 35:      | ret                                                |     |

.global directive means that the symbol mentioned is defined *here* and is *exported*

- i.e., can be referenced by other modules
- ax and main are global

```
extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}
```



# main.s

| Offset | Op | Arg |
|--------|----|-----|
|--------|----|-----|

```

0:      .data ; what follows is initialized data
0:      .globl ax ; ax is global: it may be used
          ; by others
0: ax:
0:      .long  x
4:
0:      .text ; offset restarts; w
          ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl %esp,%ebp ; point to
3:      subl $4,%esp ; make space
6:      movl ax,%eax ; put content
11:     movl (%eax),%eax ; put *x
13:     movl %eax,-4(%ebp) ; store
16:     pushl -4(%ebp) ; push y on
19:     call subr
24:     addl $4,%esp ; remove y f
27:     movl $0,%eax ; set return
31:     movl %ebp,%esp ; restore
          ; previous pointer
33:     popl %ebp ; pop frame pointer
35:     ret

```

ax is 4 bytes long and put  
the value of x here

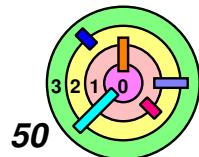
- x here is an *address*
- x will remain *unresolved*

```

extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}

```



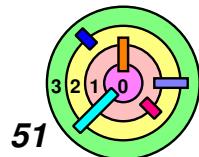
# main.s

| Offset   | Op                                                 | Arg |
|----------|----------------------------------------------------|-----|
| 0:       | .data ; what follows is initialized data           |     |
| 0:       | .globl ax ; ax is global: it may be used by others |     |
| 0: ax:   |                                                    |     |
| 0:       | .long x                                            |     |
| 4:       |                                                    |     |
| 0:       | .text ; offset restarts; w ; text (read-only code) |     |
| 0:       | .globl main                                        |     |
| 0: main: |                                                    |     |
| 0:       | pushl %ebp ; save the frame pointer                |     |
| 1:       | movl %esp,%ebp ; point to stack                    |     |
| 3:       | subl \$4,%esp ; make space                         |     |
| 6:       | movl ax,%eax ; put content of x into eax           |     |
| 11:      | movl (%eax),%eax ; put *x into eax                 |     |
| 13:      | movl %eax,-4(%ebp) ; store eax into memory         |     |
| 16:      | pushl -4(%ebp) ; push y onto stack                 |     |
| 19:      | call subr                                          |     |
| 24:      | addl \$4,%esp ; remove y from stack                |     |
| 27:      | movl \$0,%eax ; set return value                   |     |
| 31:      | movl %ebp,%esp ; restore stack pointer             |     |
| 33:      | popl %ebp ; pop frame pointer                      |     |
| 35:      | ret                                                |     |

these 3 places require relocation

```
extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}
```



# main.s

| Offset | Op | Arg |
|--------|----|-----|
|--------|----|-----|

```

0:      .data ; what follows is initialized data
0:      .globl ax ; ax is global: it may be used
          ; by others
0: ax:
0:      .long  x
4:
0:      .text ; offset restarts; w
          ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the frame pointer
1:      movl %esp,%ebp ; point to
3:      subl $4,%esp ; make space
6:      movl ax,%eax ; put content
11:     movl (%eax),%eax ; put *x
13:     movl %eax,-4(%ebp) ; store
16:     pushl -4(%ebp) ; push y on
19:     call subr
24:     addl $4,%esp ; remove y f
27:     movl $0,%eax ; set return
31:     movl %ebp,%esp ; restore
          ; previous pointer
33:     popl %ebp ; pop frame pointer
35:     ret

```

this call is a PC-relative call

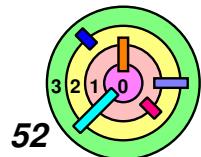
- what's stored at offset 20 is not the absolute address of subr, but a relative address
- this is just how x86 works

```

extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}

```

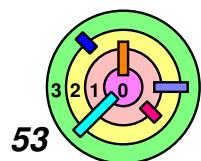


# subr.s

| Offset | Op                                                                  | Arg |
|--------|---------------------------------------------------------------------|-----|
| 0:     | .data ; what follows is initialized data                            |     |
| 0:     | printfarg:                                                          |     |
| 0:     | .string "i = %d\n"                                                  |     |
| 8:     |                                                                     |     |
| 0:     | .comm X, 4 ; 4 bytes in BSS is required<br>; for global X           |     |
| 4:     |                                                                     |     |
| 0:     | .text ; offset restarts; what follows is<br>; text (read-only code) |     |
| 0:     | .globl subr                                                         |     |
| 0:     | subr:                                                               |     |
| 0:     | pushl %ebp ; save the frame pointer                                 |     |
| 1:     | movl %esp, %ebp ; point to new frame                                |     |
| 3:     | pushl 8(%ebp) ; push i onto stack                                   |     |
| 6:     | pushl \$printfarg ; push address of string<br>; onto stack          |     |
| 11:    | call printf                                                         |     |
| 16:    | addl \$8, %esp ; pop argument off stack                             |     |
| 19:    | movl %ebp, %esp ; restore stack pointer                             |     |
| 21:    | popl %ebp ; pop frame pointer                                       |     |
| 23:    | ret                                                                 |     |

```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```



# subr.s

| Offset | Op | Arg |
|--------|----|-----|
|--------|----|-----|

```

0:      .data ; what follows is initialized data
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X, 4 ; 4 bytes in BSS
                  ; for global X
4:
0:      .text ; offset restarts; w
                  ; text (read-only co
0:      .globl subr
0: subr:
0:      pushl %ebp ; save the frame
1:      movl %esp, %ebp ; point to new frame
3:      pushl 8(%ebp) ; push i onto stack
6:      pushl $printfarg ; push address of string
                  ; onto stack
11:     call printf
16:     addl $8, %esp ; pop argument off stack
19:     movl %ebp, %esp ; restore stack pointer
21:     popl %ebp ; pop frame pointer
23:     ret

```

this is how you create a string constant

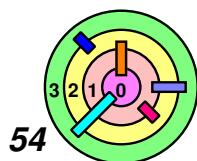
- this one is 8 bytes long
- and local to this module (since it's not global)
- can "live" somewhere else

```

#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}

```



# subr.s

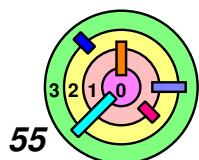
| Offset | Op                                                 | Arg |
|--------|----------------------------------------------------|-----|
| 0:     | .data ; what follows is initialized data           |     |
| 0:     | printfarg:                                         |     |
| 0:     | .string "i = %d\n"                                 |     |
| 8:     |                                                    |     |
| 0:     | .comm X, 4 ; 4 bytes in BSS<br>; for global X      |     |
| 4:     |                                                    |     |
| 0:     | .text ; offset restarts; w<br>; text (read-only co |     |
| 0:     | .globl subr                                        |     |
| 0:     | subr:                                              |     |
| 0:     | pushl %ebp ; save the frame                        |     |
| 1:     | movl %esp, %ebp ; point to                         |     |
| 3:     | pushl 8(%ebp) ; push i onto                        |     |
| 6:     | pushl \$printfarg ; push address<br>; onto stack   |     |
| 11:    | call printf                                        |     |
| 16:    | addl \$8, %esp ; pop argument                      |     |
| 19:    | movl %ebp, %esp ; restore stack pointer            |     |
| 21:    | popl %ebp ; pop frame pointer                      |     |
| 23:    | ret                                                |     |

this is how you create a string constant

- this one is 8 bytes long
- and local to this module (since it's not global)
- can "live" somewhere else
- it is used here

```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```



# subr.s

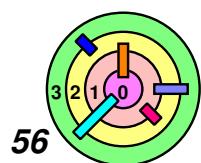
| Offset | Op                                                 | Arg |
|--------|----------------------------------------------------|-----|
| 0:     | .data ; what follows is initialized data           |     |
| 0:     | printfarg:                                         |     |
| 0:     | .string "i = %d\n"                                 |     |
| 8:     |                                                    |     |
| 0:     | .comm X, 4 ; 4 bytes in BSS<br>; for global X      |     |
| 4:     |                                                    |     |
| 0:     | .text ; offset restarts; w<br>; text (read-only co |     |
| 0:     | .globl subr                                        |     |
| 0:     | subr:                                              |     |
| 0:     | pushl %ebp ; save the frame                        |     |
| 1:     | movl %esp, %ebp ; point to                         |     |
| 3:     | pushl 8(%ebp) ; push i onto                        |     |
| 6:     | pushl \$printfarg ; push address<br>; onto stack   |     |
| 11:    | call printf                                        |     |
| 16:    | addl \$8, %esp ; pop argument                      |     |
| 19:    | movl %ebp, %esp ; restore stack pointer            |     |
| 21:    | popl %ebp ; pop frame pointer                      |     |
| 23:    | ret                                                |     |

4 bytes is required in the **bss** segment for this global variable

— .comm directive also means that the symbol mentioned is defined **here** and is **exported**

```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```



# subr.s

| Offset | Op | Arg |
|--------|----|-----|
|--------|----|-----|

```

0:      .data ; what follows is initialized data
0: printfarg:
0:      .string "i = %d\n"
8:
0:      .comm X, 4 ; 4 bytes in BSS
                  ; for global X
4:
0:      .text ; offset restarts; w
                  ; text (read-only co
0:      .globl subr
0: subr:
0:      pushl %ebp ; save the frame
1:      movl %esp, %ebp ; point to new frame
3:      pushl 8(%ebp) ; push i onto stack
6:      pushl $printfarg ; push address of
                  ; onto stack
11:     call printf
16:     addl $8, %esp ; pop argument off stack
19:     movl %ebp, %esp ; restore stack pointer
21:     popl %ebp ; pop frame pointer
23:     ret

```

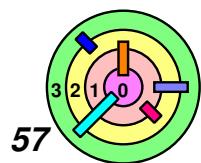
subr is a global symbol  
exported from here

```

#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}

```



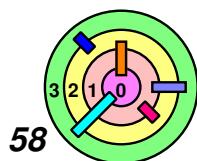
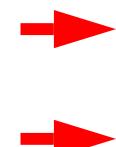
# subr.s

| Offset | Op                                                 | Arg |
|--------|----------------------------------------------------|-----|
| 0:     | .data ; what follows is initialized data           |     |
| 0:     | printfarg:                                         |     |
| 0:     | .string "i = %d\n"                                 |     |
| 8:     |                                                    |     |
| 0:     | .comm X, 4 ; 4 bytes in BSS<br>; for global X      |     |
| 4:     |                                                    |     |
| 0:     | .text ; offset restarts; w<br>; text (read-only co |     |
| 0:     | .globl subr                                        |     |
| 0:     | subr:                                              |     |
| 0:     | pushl %ebp ; save the frame                        |     |
| 1:     | movl %esp, %ebp ; point to                         |     |
| 3:     | pushl 8(%ebp) ; push i onto                        |     |
| 6:     | pushl \$printfarg ; push address<br>; onto stack   |     |
| 11:    | call printf                                        |     |
| 16:    | addl \$8, %esp ; pop argument                      |     |
| 19:    | movl %ebp, %esp ; restore stack pointer            |     |
| 21:    | popl %ebp ; pop frame pointer                      |     |
| 23:    | ret                                                |     |

*relocation* is required for printf and printfarg

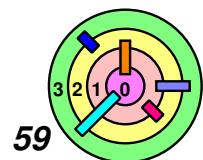
```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```



# Object Files

- ▶ An object file describes what's in the data, bss, and text segments in separate sections
- ▶ Along with each section is a list of:
  - global symbols
  - undefined symbols
  - instructions for relocation
    - these instructions indicate
      - ◊ which locations within the section must be modified
      - ◊ which symbol's value is used to modify the location
    - a symbol's value is the address that is ultimately determined for it
    - typically, this address is added to the location being modified
- ▶ To inspect an object file on Unix
  - nm - list symbols from object files
  - objdump - display information from object files



# subr.o

**Data:**

Size: 8

Contents: "i = %d\n"

**bss:**

Size: 4

Global: x, offset 0

**Text:**

Size: 24

Global: subr, offset 0

Undefined: printf

**Relocation:**

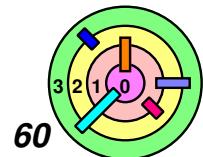
offset 7, size 4, value: addr of printfarg

offset 12, size 4, value: PC-relative addr of  
printf

Contents: [machine instructions]



No tool can generate exactly the above printout



# subr.s

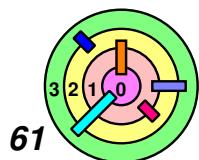
| Offset | Op                                                         | Arg |
|--------|------------------------------------------------------------|-----|
| 0:     | .data ; what follows is initialized data                   |     |
| 0:     | printfarg:                                                 |     |
| 0:     | .string "i = %d\n"                                         |     |
| 8:     |                                                            |     |
| 0:     | .comm X, 4 ; 4 bytes in BSS<br>; for global X              |     |
| 4:     |                                                            |     |
| 0:     | .text ; offset restarts; w<br>; text (read-only code)      |     |
| 0:     | .globl subr                                                |     |
| 0:     | subr:                                                      |     |
| 0:     | pushl %ebp ; save the frame pointer                        |     |
| 1:     | movl %esp, %ebp ; point to new stack                       |     |
| 3:     | pushl 8(%ebp) ; push i onto stack                          |     |
| 6:     | pushl \$printfarg ; push address of string<br>; onto stack |     |
| 11:    | call printf                                                |     |
| 16:    | addl \$8, %esp ; pop arguments from stack                  |     |
| 19:    | movl %ebp, %esp ; restore stack pointer                    |     |
| 21:    | popl %ebp ; pop frame pointer                              |     |
| 23:    | ret                                                        |     |

**relocation** is required for:  
 □ printfarg at offset 7  
 □ printf at offset 12

```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```

- 6: pushl \$printfarg ; push address of string onto stack
- 11: call printf



# subr.o

**Data:**

**Size:** 8

**Contents:** "i = %d\n"

**bss:**

**Size:** 4

**Global:** x, offset 0

**Text:**

**Size:** 24

**Global:** subr, offset 0

**Undefined:** printf

**Relocation:**

offset 7, size 4, value: addr of printfarg

offset 12, size 4, value: PC-relative addr of printf

**Contents:** [machine instructions]

**relocation** is required for:

- ─ printfarg at offset 7
- ─ printf at offset 12

```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```



# subr.o

**Data:**

Size: 8

Contents: "i = %d\n"

x and subr are *exported*

needed in main.o

**bss:**

Size: 4

Global: x, offset 0

**Text:**

Size: 24

Global: subr, offset 0

Undefined: printf

Relocation:

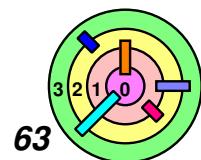
offset 7, size 4, value: addr of printfarg

offset 12, size 4, value: PC-relative addr of  
printf

Contents: [machine instructions]

```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```



# main.s

| Offset | Op | Arg |
|--------|----|-----|
|--------|----|-----|

```

0:      .data ; what follows is initialized data
0:      .globl ax ; ax is global:
                  ; by others
0: ax:
0:      .long  x
4:
0:      .text ; offset restarts; w
                  ; text (read-only co
0:      .globl main
0: main:
0:      pushl %ebp ; save the fram
1:      movl %esp,%ebp ; point to
3:      subl $4,%esp ; make space
6:      movl ax,%eax ; put conten
11:     movl (%eax),%eax ; put *x
13:     movl %eax,-4(%ebp) ; stor
16:     pushl -4(%ebp) ; push y onto stack
19:     call subr
24:     addl $4,%esp ; remove y from stack
27:     movl $0,%eax ; set return value to 0
31:     movl %ebp,%esp ; restore stack pointer
33:     popl %ebp ; pop frame pointer
35:     ret

```

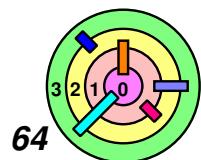
these 2 places remained  
*unresolved*

```

extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}

```



# main.o

**Data:**

**Size:** 4

**Global:** ax, offset 0

**Undefined:** x

**Relocation:** offset 0, size 4, v

**Contents:** 0x00000000

these 2 places remained

*unresolved*

→ they are noted in main.o

**bss:**

**Size:** 0

**Text:**

**Size:** 36

**Global:** main, offset 0

**Undefined:** subr

**Relocation:**

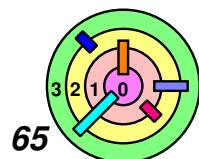
offset 7, size 4, value: addr of ax

offset 20, size 4, value: PC-relative  
addr of subr

**Contents:** [machine instructions]

```
extern int x;
int *ax = &x;

int main( ) {
    void subr(int);
    int y = *ax;
    subr(y);
    return(0);
}
```



# subr.o

**Data:**

Size: 8

Contents: "i = %d\n"

printf remained *unresolved*

**bss:**

Size: 4

Global: x, offset 0

**Text:**

Size: 24

Global: subr, offset 0

Undefined: printf



Relocation:

offset 7, size 4, value: addr of printfarg

offset 12, size 4, value: PC-relative addr of  
printf

Contents: [machine instructions]

```
#include <stdio.h>
int x;

void subr(int i) {
    printf("i = %d\n", i);
}
```

# printf.o

**Data:**

**Size:** 1024

**Global:** StandardFiles

**Contents:** ...

assume that printf.o looks like this

■ write is *unresolved*

**bss:**

**Size:** 256

**Text:**

**Size:** 12000

**Global:** printf, offset 100

...

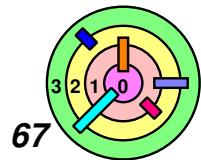
→ **Undefined:** write

**Relocation:**

offset 211, value: addr of StandardFiles

offset 723, value: PC-relative addr of write

**Contents:** [machine instructions]



# write.o

**Data:**

**Size:** 0

**bss:**

**Size:** 4

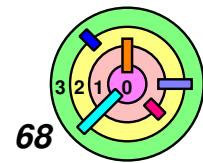
**Global:** errno, offset 0

**Text:**

**Size:** 16

**Contents:** [machine  
               instructions]

and write.o looks like this



# startup function

**Data:**

**Size:** 0

**bss:**

**Size:** 0

**Text:**

**Size:** 36

**Undefined:** main

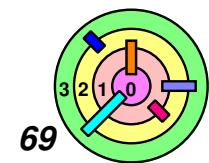
**Relocation:**

**offset 21, value: main**

**Contents:** [machine  
instructions]

every C program contains a **startup** routine that is called first

- it calls `main()`
- if `main()` returns, it calls `exit()`
- our example is incomplete



# prog

## Text

|                      |       |
|----------------------|-------|
| <code>main</code>    | 4096  |
| <code>subr</code>    | 4132  |
| <code>printf</code>  | 4156  |
| <code>write</code>   | 16156 |
| <code>startup</code> | 16172 |

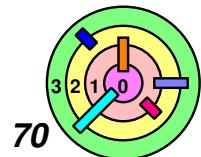
## Data

|                            |       |
|----------------------------|-------|
| <code>ax</code>            | 16384 |
| <code>printfargs</code>    | 16388 |
| <code>StandardFiles</code> | 16396 |

## BSS

|                    |       |
|--------------------|-------|
| <code>x</code>     | 17420 |
| <code>errno</code> | 17424 |

- this is how `ld` might set things up
  - `ld` *lays out* the *address space*
  - `ld` allocates memory in *pages* (typically 4KB each)
- `main` does not start at location 0
  - first "page" is typically made inaccessible so that references to null pointers will fail (get SIGSEG)



# prog

## Text

|         |       |
|---------|-------|
| main    | 4096  |
| subr    | 4132  |
| printf  | 4156  |
| write   | 16156 |
| startup | 16172 |

} R/O

## Data

|               |       |
|---------------|-------|
| ax            | 16384 |
| printfargs    | 16388 |
| StandardFiles | 16396 |

} R/W

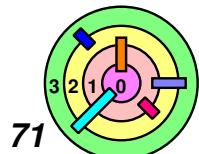
## BSS

|       |       |
|-------|-------|
| x     | 17420 |
| errno | 17424 |

printfargs can be modified by the application

- ─ is that okay?
- ─ where else would you put it?
- ─ who decides?

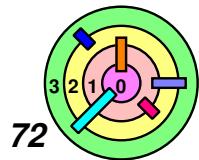
- ─ due to the use of "*pages*", the data segment needs to start at a page boundary (i.e., multiple of page size)
  - this way, the *text segment* can be made *read-only* while the *data* and *bss segments* made *read-write*
  - here we assume 4KB pages (therefore, pages start at 4096, 8192, 12288, 16384, etc.)



# Page Protection & Virtual Memory Basics

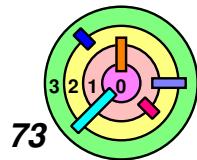


- A process has, say, a 32-bit address space
  - that's 4GB of memory
  - our prog process, when it starts, only needs about 16KB for text+data+bss (plus more for stack)
  - allocating 4GB of physical memory will be a *huge waste*



# Page Protection & Virtual Memory Basics

- ➡ A process has, say, a 32-bit address space
  - that's 4GB of memory
  - our prog process, when it starts, only needs about 16KB for text+data+bss (plus more for stack)
  - allocating 4GB of physical memory will be a *huge waste*
- ➡ Solution: *indirection*
  - use *page table* to *map virtual to physical addresses*
    - a page table is a *kernel data structure*, used by the *CPU*
  - OS allocates *pages* of *physical memory* using the *Buddy System*
    - a page is 4KB in many systems
  - a page corresponds to physical memory that can be located (*or "mapped"*) anywhere in virtual memory
    - "*address translation*" done in hardware
  - some advantages:
    - page protection
    - only allocate needed physical memory pages
  - a lot more to come in Ch 7



# Page Protection & Virtual Memory Basics

## Text

|                |       |
|----------------|-------|
| <b>main</b>    | 4096  |
| <b>subr</b>    | 4132  |
| <b>printf</b>  | 4156  |
| <b>write</b>   | 16156 |
| <b>startup</b> | 16172 |

## Data

|                      |       |
|----------------------|-------|
| <b>ax</b>            | 16384 |
| <b>printfargs</b>    | 16388 |
| <b>StandardFiles</b> | 16396 |

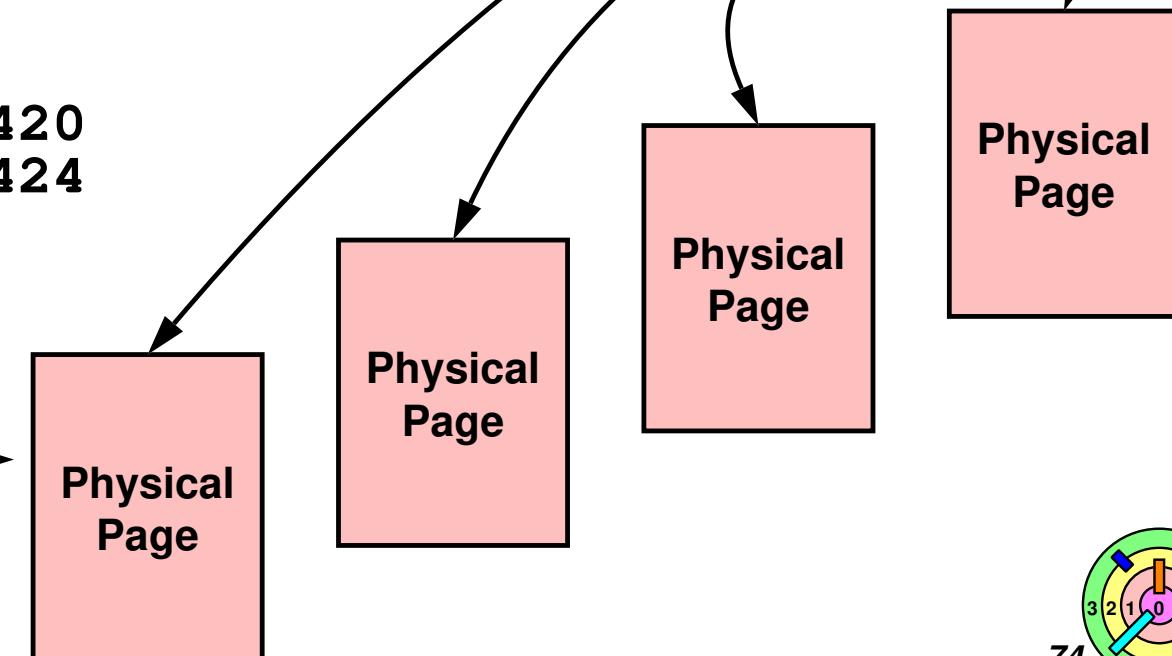
## BSS

|              |       |
|--------------|-------|
| <b>x</b>     | 17420 |
| <b>errno</b> | 17424 |

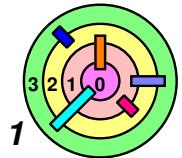
ask buddy system to  
allocate these pages

## Page Table

| # | Start | Access | Physical Addr |
|---|-------|--------|---------------|
| 0 | 0     | -      | -             |
| 1 | 4096  | R      |               |
| 2 | 8192  | R      |               |
| 3 | 12288 | R      |               |
| 4 | 16384 | R/W    |               |

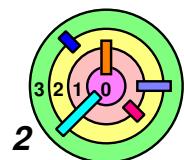


# 3.5 Booting



# Boot

- ➔ Came from the idiomatic expression, "to pull yourself up by your bootstraps"
  - without the help of others
  - it's a difficult situation
- ➔ In OS
  - load its OS into memory
    - which kind of means that you need an OS in memory to do it
- ➔ Solution
  - load a tiny OS into memory
    - known as the *bootstrap loader*
    - then again, who loads this tiny OS into memory?
      - ◊ how about first loading a tiny bootstrap loader?

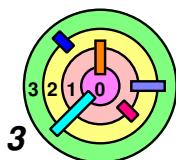


# PDP-8



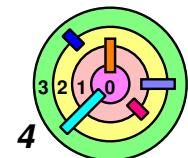
**toggle  
switches**

- How about manually put into memory a simple bootstrap loader?
- approach taken by PDP-8
    - "toggles in" the program
  - read OS from paper tape

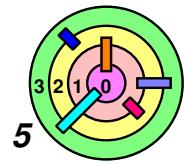


# PDP-8 Boot Code

```
07756 6032 KCC
07757 6031 KSF
07760 5357 JMP .-1
07761 6036 KRB
07762 7106 CLL RTL
07763 7006 RTL
07764 7510 SPA
07765 5357 JMP 7757
07766 7006 RTL
07767 6031 KSF
07770 5367 JMP .-1
07771 6034 KRS
07772 7420 SNL
07773 3776 DCA I 7776
07774 3376 DCA 7776
07775 5356 JMP 7756
07776 0000 AND 0
07777 5301 JMP 7701
```



# VAX-11/780



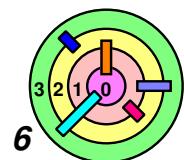
# VAX-11/780 Boot

→ Separate "console computer"

- LSI-11
- hard-wired to always run the code contained in its on-board read-only memory
- then read boot code (i.e., the bootstrap loader) from floppy disk
- then load OS from root directory of first file system on primary disk

→ Code on floppy disk (the bootstrap loader) would handle:

- *disk device*
- *on-disk file system*
- it needs the right *device driver*
- it needs to know how the disk is setup
  - what sort of *file system* is on the disk
  - how the disk is *partitioned*
    - ◊ a disk may hold multiple and different file systems, each in a separate partition

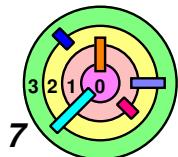


# Configuring the OS



## Early Unix

- OS statically linked to contain all needed device drivers
  - device drivers were statically linked to the OS
- all device-specific info included with drivers
- *disk drivers* contained *partitioning description*
- therefore, the following actions may all require compiling a new version of the OS:
  - adding a new device
  - replacing a device
  - modifying disk-partitioning information



# Configuring the OS



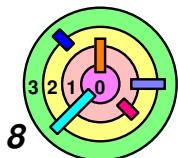
## Later Unix

- OS statically linked to contain all needed device drivers
- at boot time, OS would *probe* to see which devices were present and discover device-specific info
- partition table in first sector of each disk



## Even later Unix

- allowed device drivers to be *dynamically loaded* into a *running system*



# IBM PC

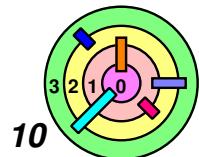


# Issues



## Open architecture

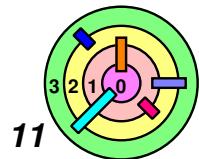
- although MS-DOS was distributed in binary form only
- large market for peripherals, most requiring special drivers
- how to access boot device?
- how does OS get drivers for new devices?



10

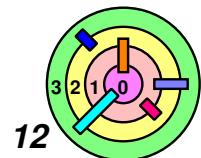
# The Answer: BIOS

- ➡ **Basic Input-Output System (*BIOS*)**
  - code stored in read-only memory (**ROM**)
  - configuration data in non-volatile RAM (**NVRAM**)
    - such as **CMOS**
    - including set of boot-device names
  - the BIOS provides three primary functions
    - power-on self test (**POST**)
      - ◊ so it knows **where** to load the boot program
    - load and transfer control to boot program
    - provide **drivers** for all devices
- ➡ **Main BIOS on motherboard**
  - supplied as a chip on the "motherboard"
  - contains everything necessary to perform the above 3 functions
  - additional BIOSes on other boards
    - provide access to additional devices

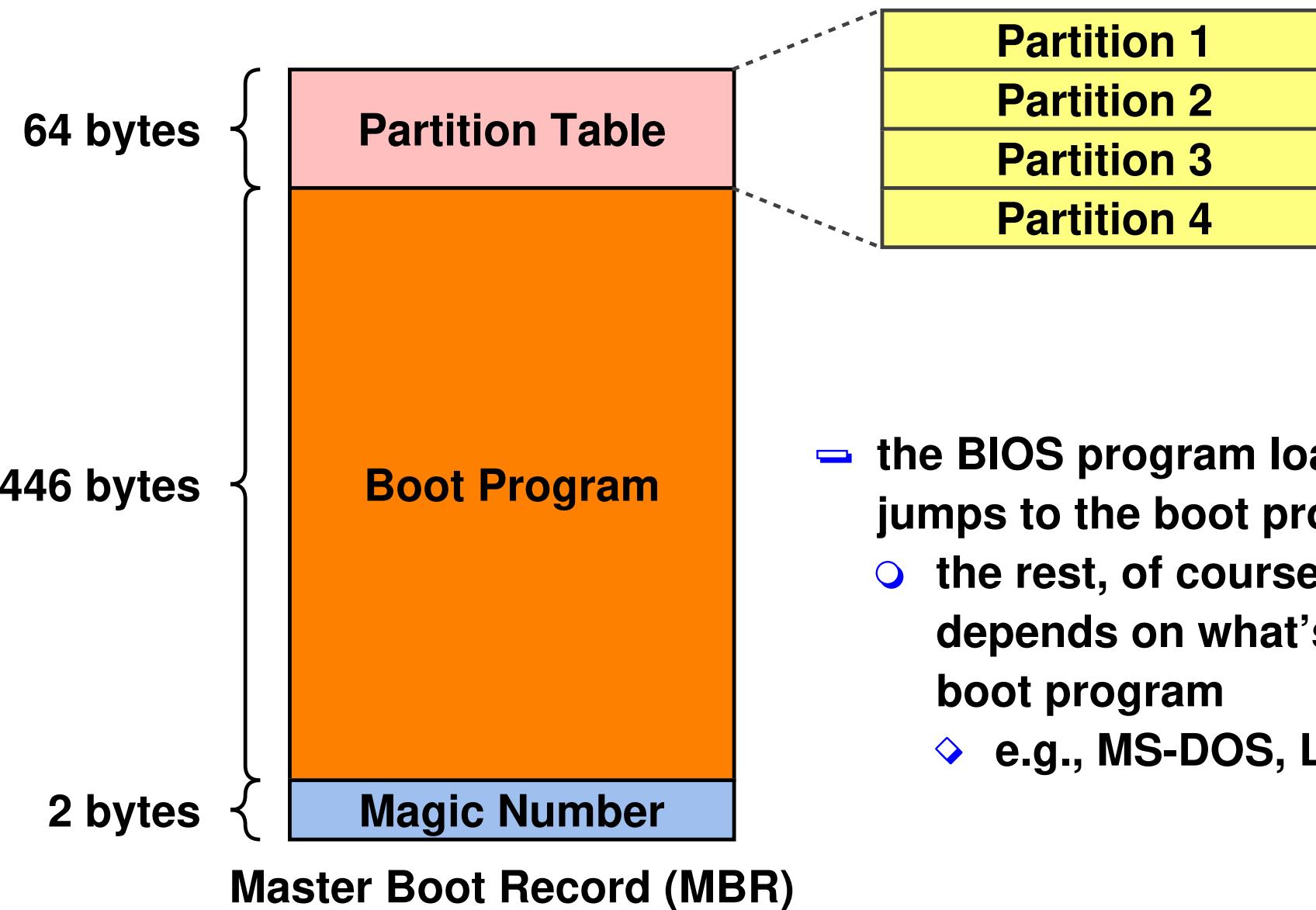


# POST

- ➡ On power-on, CPU executes BIOS code
  - located in last 64KB of first megabyte of address space
    - starting at location 0xf0000
    - CPU is hard-wired to start executing at 0xfffff0 on startup
      - ◊ the last 16 bytes of this region
      - ◊ jump to POST
- ➡ POST
  - initializes hardware
  - **counts** memory locations
    - by testing for working memory
- ➡ Next step is to find a boot device
  - the CMOS is configured with a boot order
- ➡ Next step is to load the Master Boot Record (**MBR**) from the first sector of the boot device, if it's a floppy/diskette
  - or cylinder 0, head 0, sector 1 of a hard disk (Ch 6)

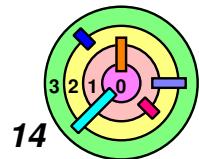


# Getting the Boot Program



# MS-DOS Boot Program

- ➡ One of the hard drive partitions is labeled as the ***active*** partition
- ➡ The MS-DOS boot program finds the active partition
  - loads the first sector from it
    - which contains the "volume boot program"
  - pass control to that program
    - which then load the OS from that partition

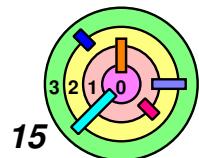


# Linux Booting (1)

→ Two stages of booting provided by one of:

- *lilo* (Linux Loader)
  - uses sector numbers of kernel image
  - therefore, must be modified if a kernel image moves
- *grub* (Grand Unified Boot Manager)
  - understands various file systems
  - can find a kernel image given a file system *path name*
- both allow dual (or greater) booting
- select which system to boot from menu
  - perhaps choice of Linux or Windows

→ The next step is for the kernel to *configure* itself



# Linux Booting (2)

assembler code  
(`startup_32`)

- Kernel image is compressed
  - step 1: set up stack, clear BSS, uncompress kernel, then transfer control to it

assembler code  
(different  
`startup_32`)

- **Process 0** is created
  - step 2: set up initial page tables, turn on *address translation* (Ch 7)
  - process 0 knows how to handle some aspects of paging

C code  
(`start_kernel`)

- Do further initialization
  - step 3: initialize rest of kernel, create the "init" process (i.e., **process 1**, which is the ancestor of all other user processes)
  - invoke the *scheduler*

→ Your kernel 1 assignment starts at step 3 above

- NOTE: **weenix** is not exactly Linux

# BIOS Device Drivers

- ➡ Originally, the BIO provided drivers for all devices
  - OS would call BIOS-provided code whenever it required services of a device driver
- ➡ These drivers sat in low memory and provided minimal functionality
  - later systems would copy them into primary memory
  - even later systems would provide their own drivers
  - nevertheless, BIO drivers are still used for booting
    - how else can you do it?



# Beyond BIOS

## → BIOS

- designed for 16-bit x86 of mid 1980s
- not readily extensible to other architectures

## → Open Firmware

- designed by Sun
- portable
- drivers, boot code in Forth
  - compiled into *bytecode*

## → Intel developed a replacement for BIOS called *EFI (Extensible Firmware Interface)*

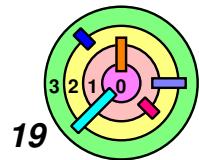
- also uses *bytecode*



# Ch 4: Operating-System Design

Bill Cheng

*<http://merlot.usc.edu/william/uscl/>*



# OS Design

→ We will now look at how OSes are constructed

- what goes into an OS
- how they interact with each other
- how is the software structured
- how performance concerns are factored in

→ We will introduce new components in this chapter

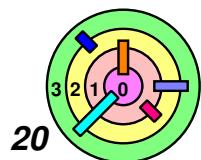
- scheduling (Ch 5)
- file systems (Ch 6)
- virtual memory (Ch 7)

→ We will start with a simple hardware configuration

- what OS is needed to support this

→ Applications views the OS as the "computer"

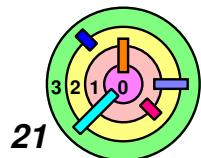
- the OS needs to provide a *consistent* and *usable interface*
  - while being *secure* and *efficient*
- that's a pretty tall order!



# OS Design

→ Our goal is to *build a general-purpose OS*

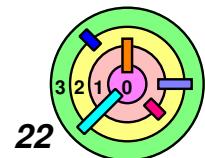
- can run a variety of applications
  - some are interactive
  - many use network communication
  - all read/write to a file system
- it's like most general-purpose OSes
  - Linux                            ○ Solaris
  - FreeBSD                        ○ Mac OS X
  - Chromium OS (has a Linux kernel)
  - Windows (the only one that's not directly based on Unix)
- all these OSes are quite similar, functionally!  
they all provide:
  - processes                      ○ threads
  - file systems                    ○ network protocols with similar APIs
  - user interface with display, mouse, keyboard
  - access control based on file ownership and that file owners can control



F20-Q12

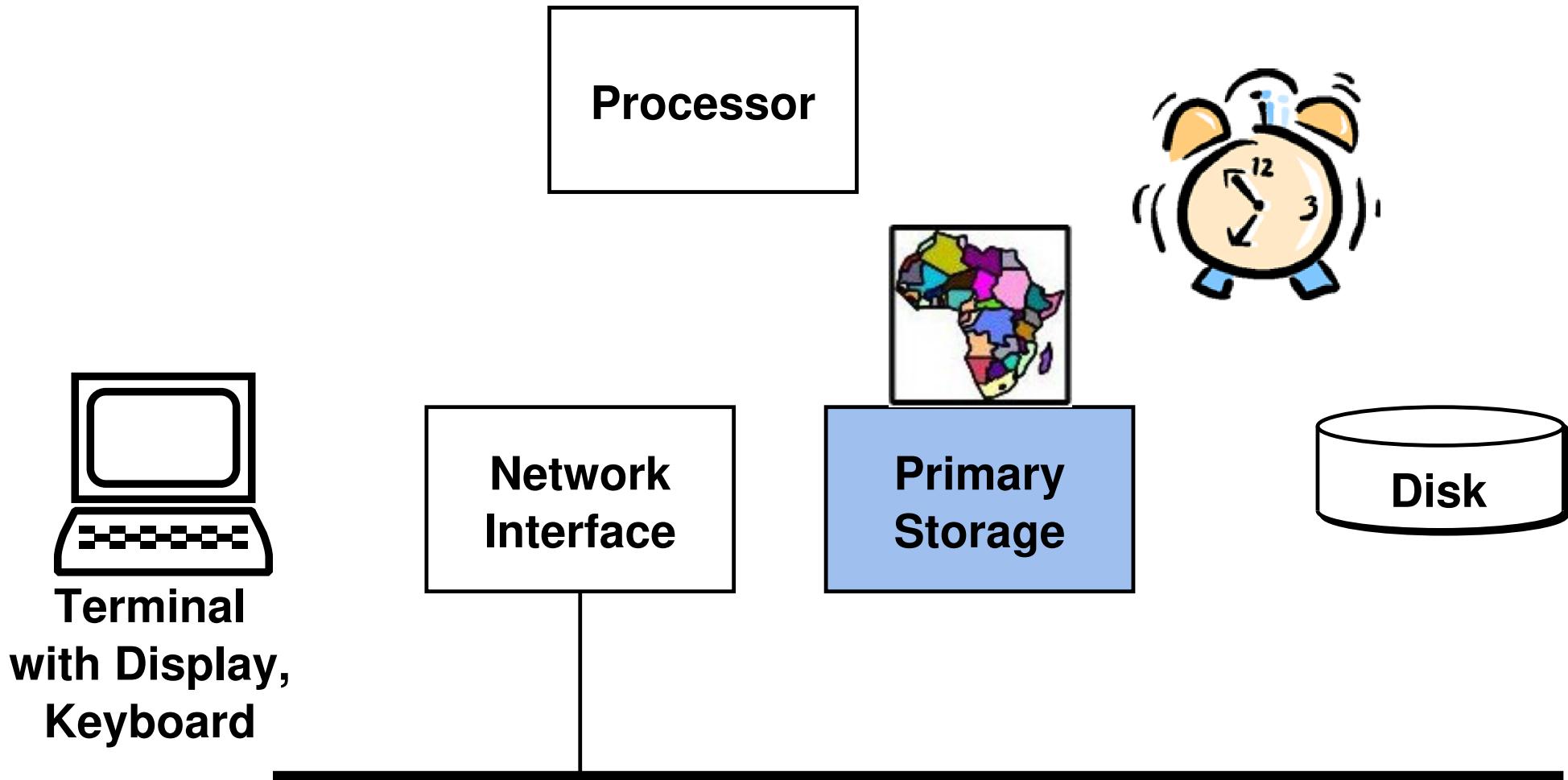
# OS Design Issues

- **Performance**
  - efficiency of application
- **Modularity**
  - tradeoffs between modularity and performance
- **Device independence**
  - for new devices, don't need to write a new OS
- **Security/Isolation**
  - isolate OS from application



22

# Simple Configuration



- Early 1980s OS, so we can focus on the basic OS issues
- no support for bit-mapped displays and mice
  - generally ***less efficient*** design

# OS Components



• • •

**Applications**

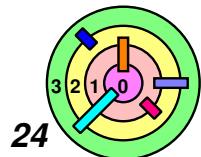
---

**OS**

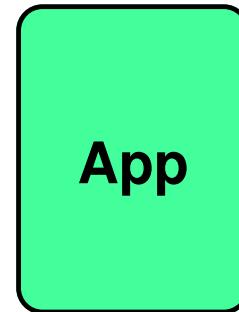
**Processor  
Management**

**Memory  
Management**

**I/O Management**



# OS Components



• • •

**Applications**

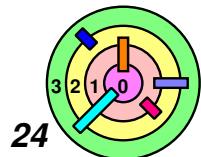
---

**OS**

**Processor  
Management**

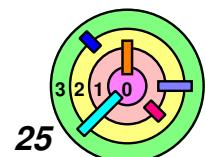
**Memory  
Management**

**I/O Management**

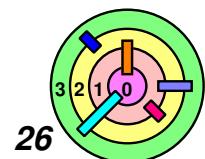
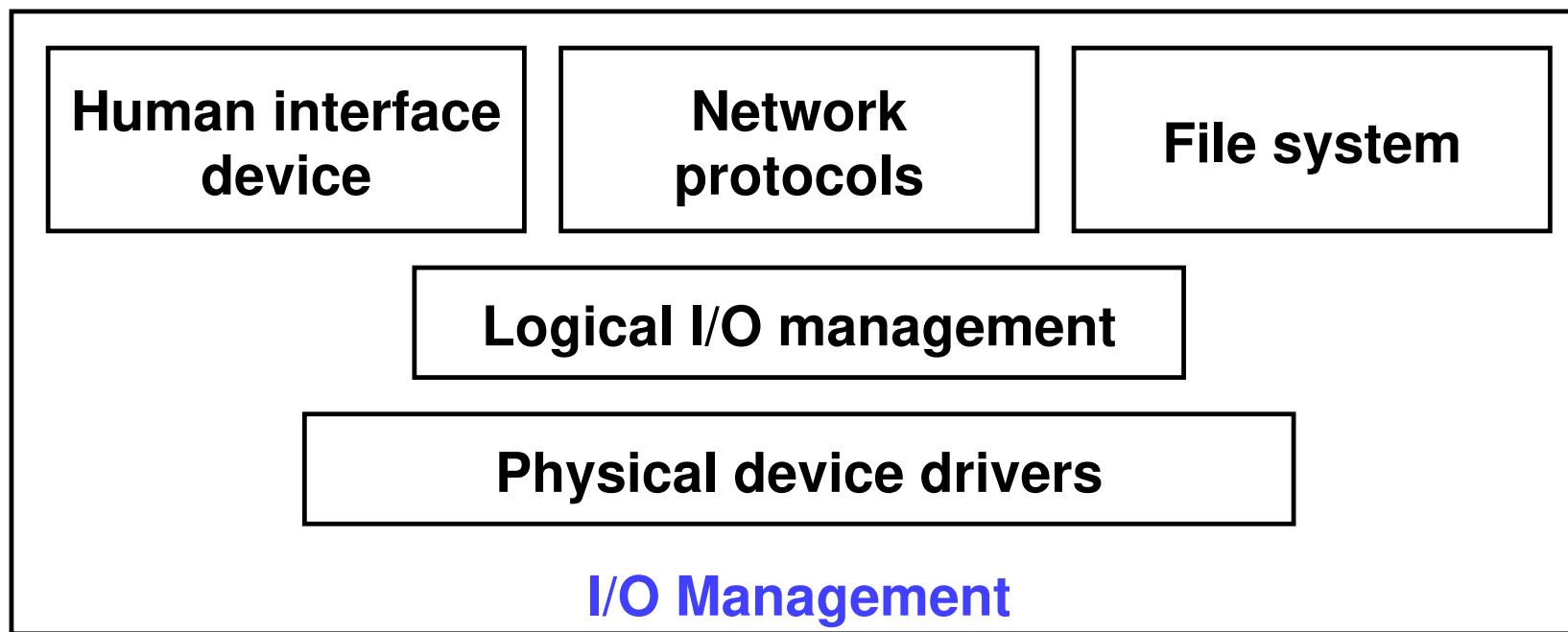
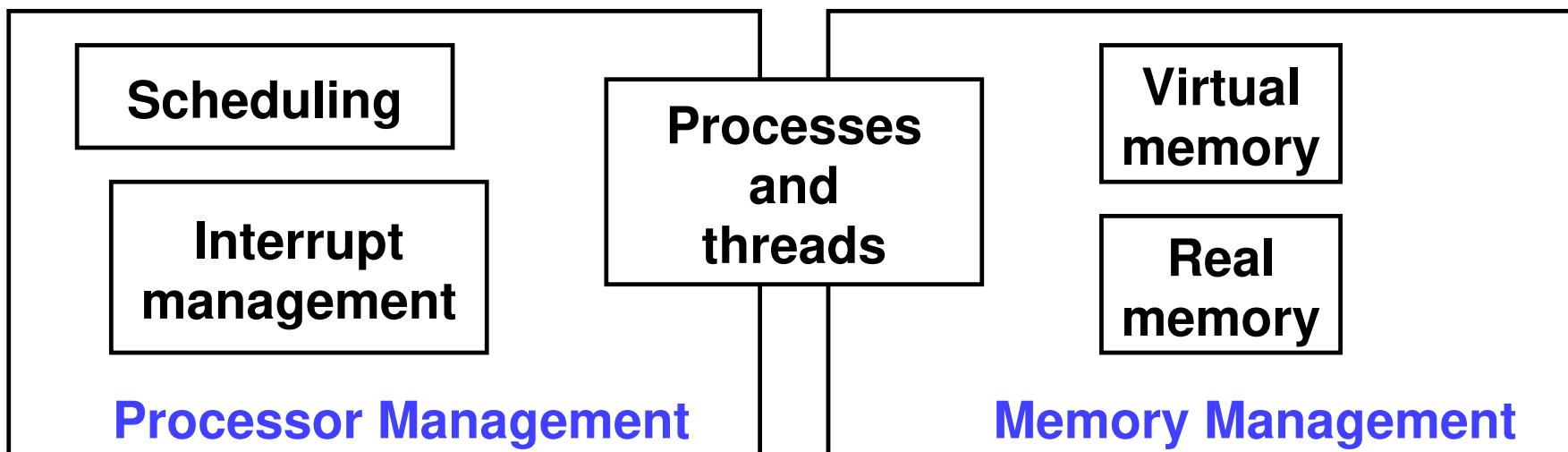


# A Simple System: To Be Discussed

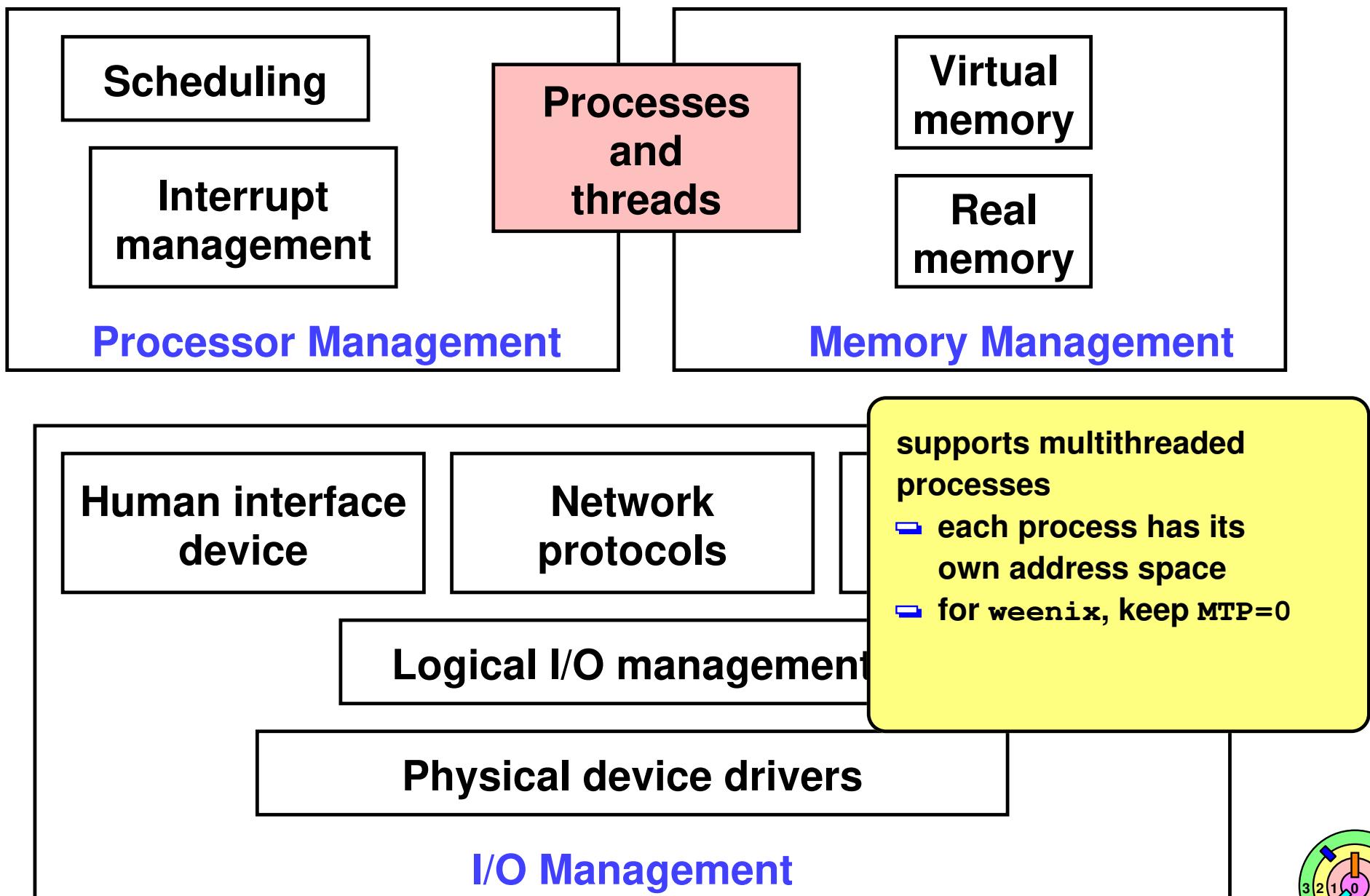
- ➡ What is the functionality of the components?
- ➡ What are the key data structures?
- ➡ What mechanisms are there to support the applications?
- ➡ How is the system broken up into modules?
- ➡ To what extent is the system extensible?
- ➡ What parts run in the OS kernel in privileged mode? What parts run as library code in user applications? What parts run as separate applications?
- ➡ In which execution contexts do the various activities take place?
  - e.g., thread context vs. interrupt context



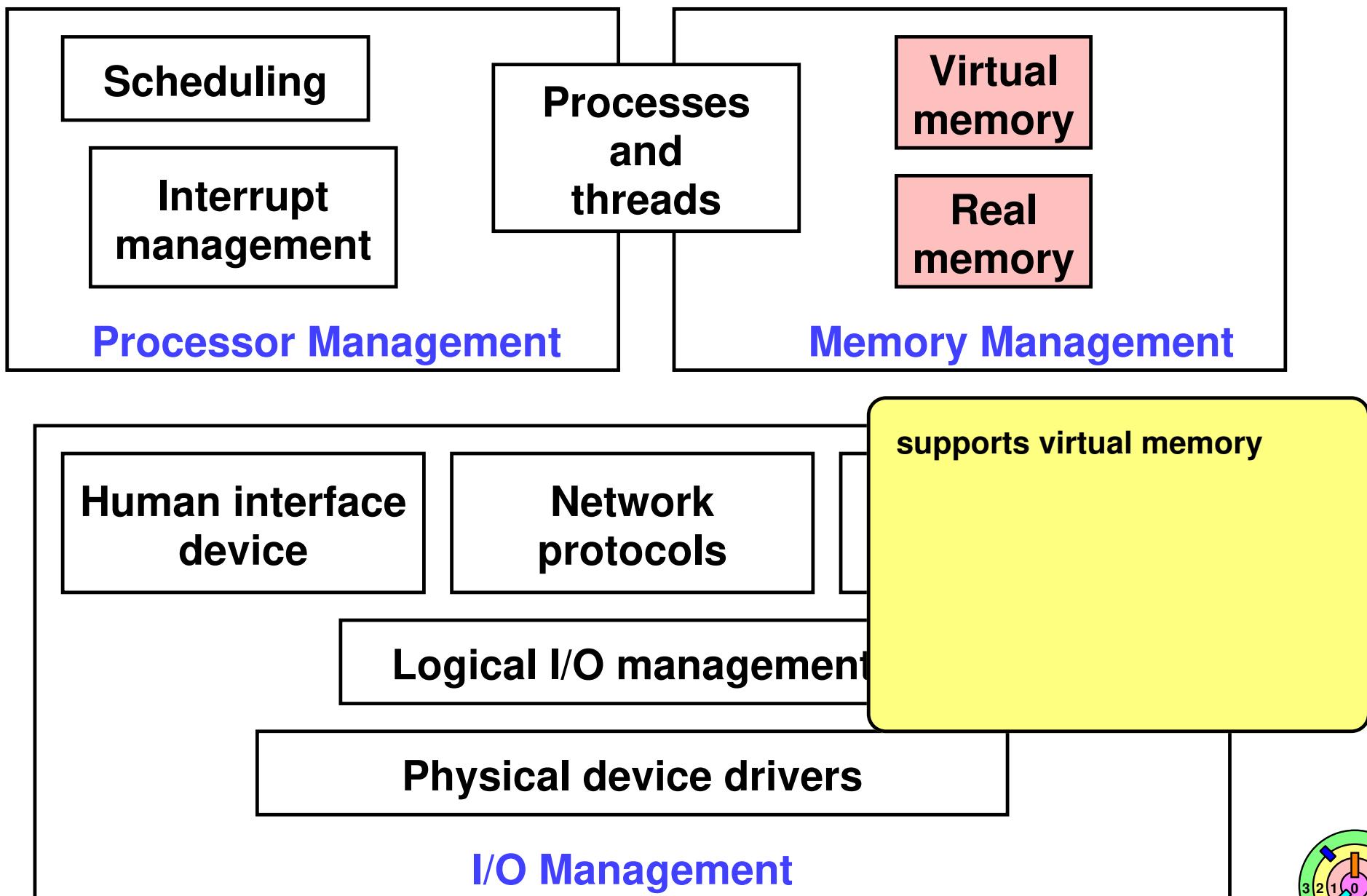
# OS Components



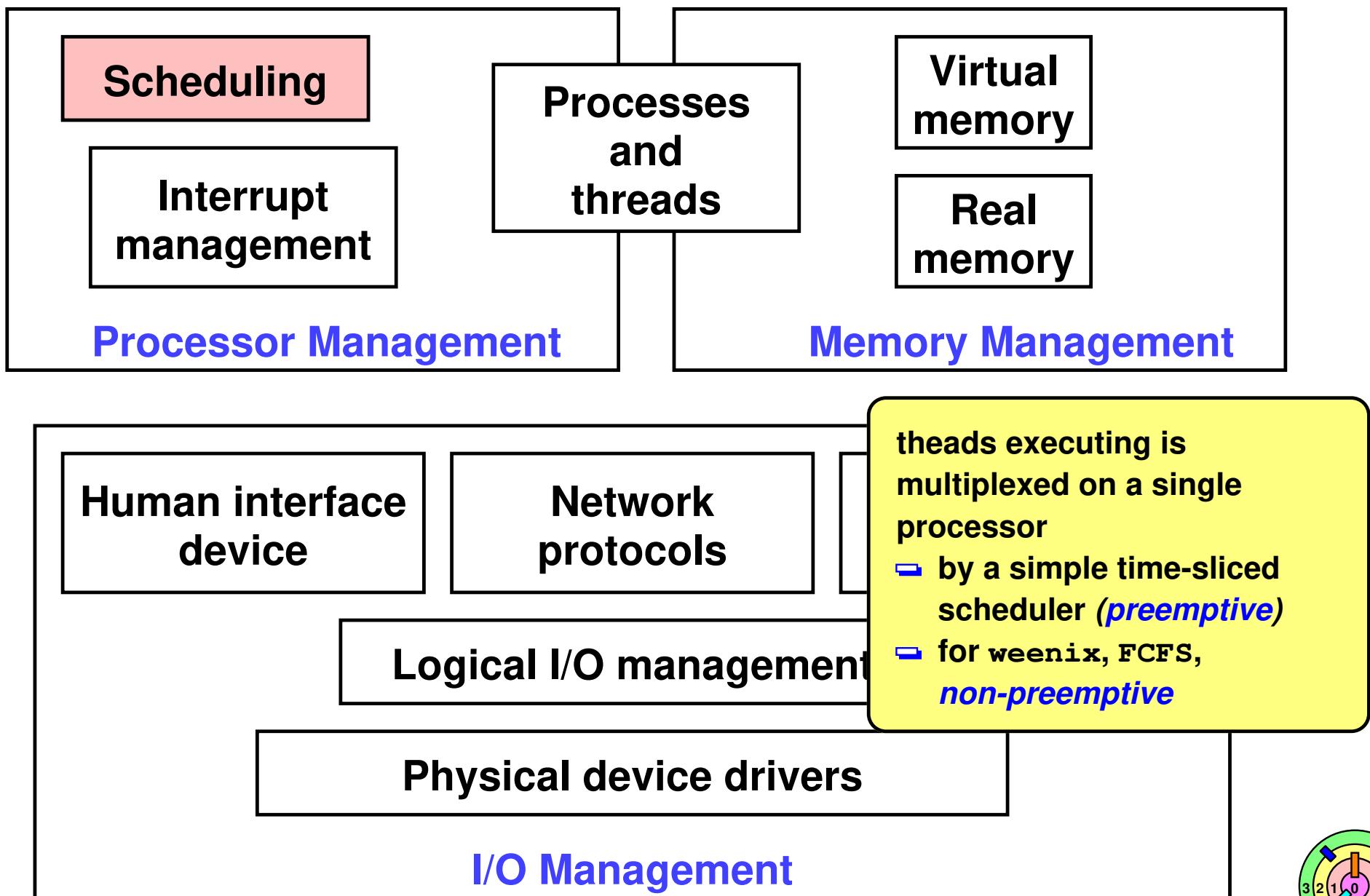
# OS Components



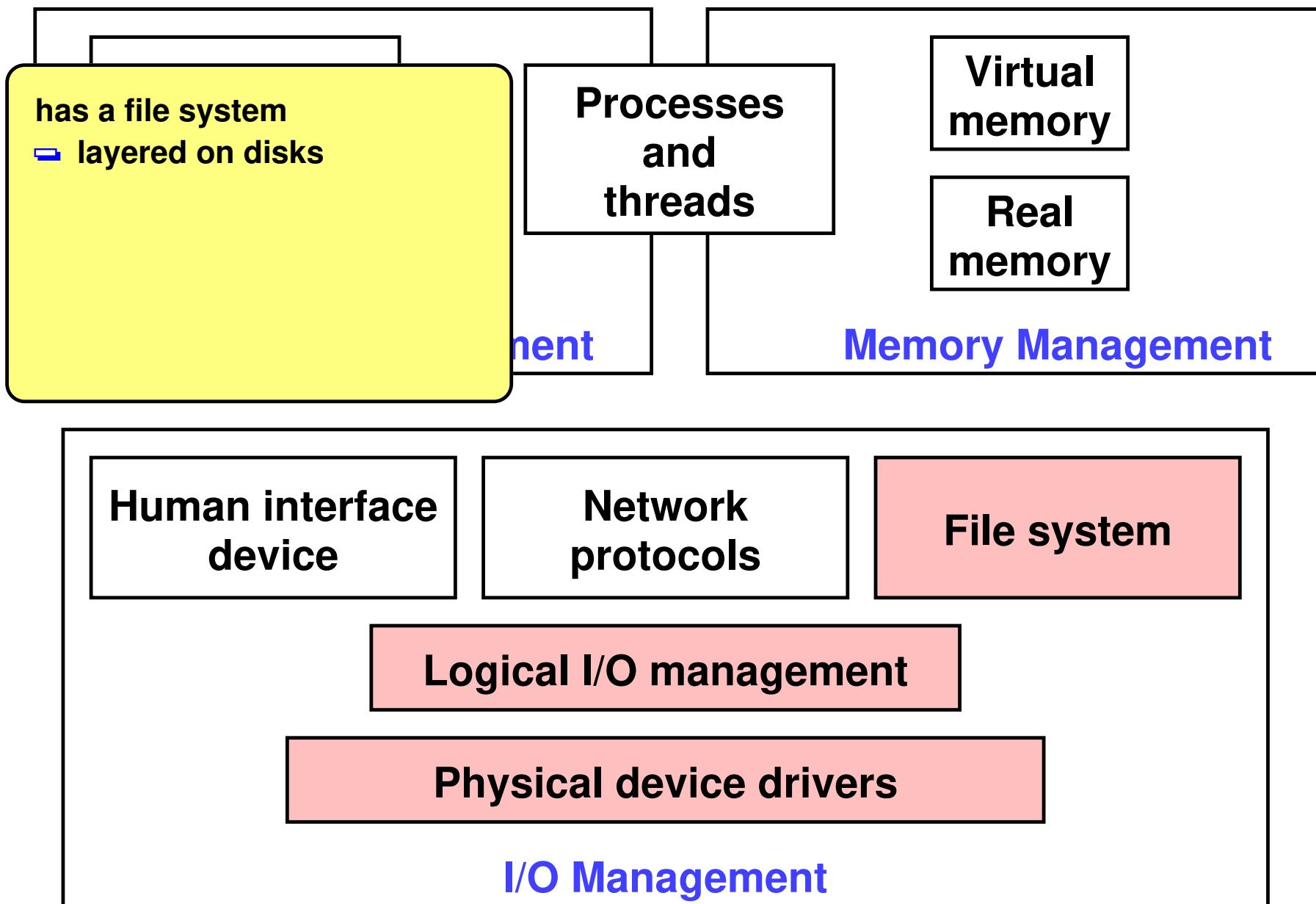
# OS Components



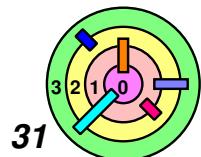
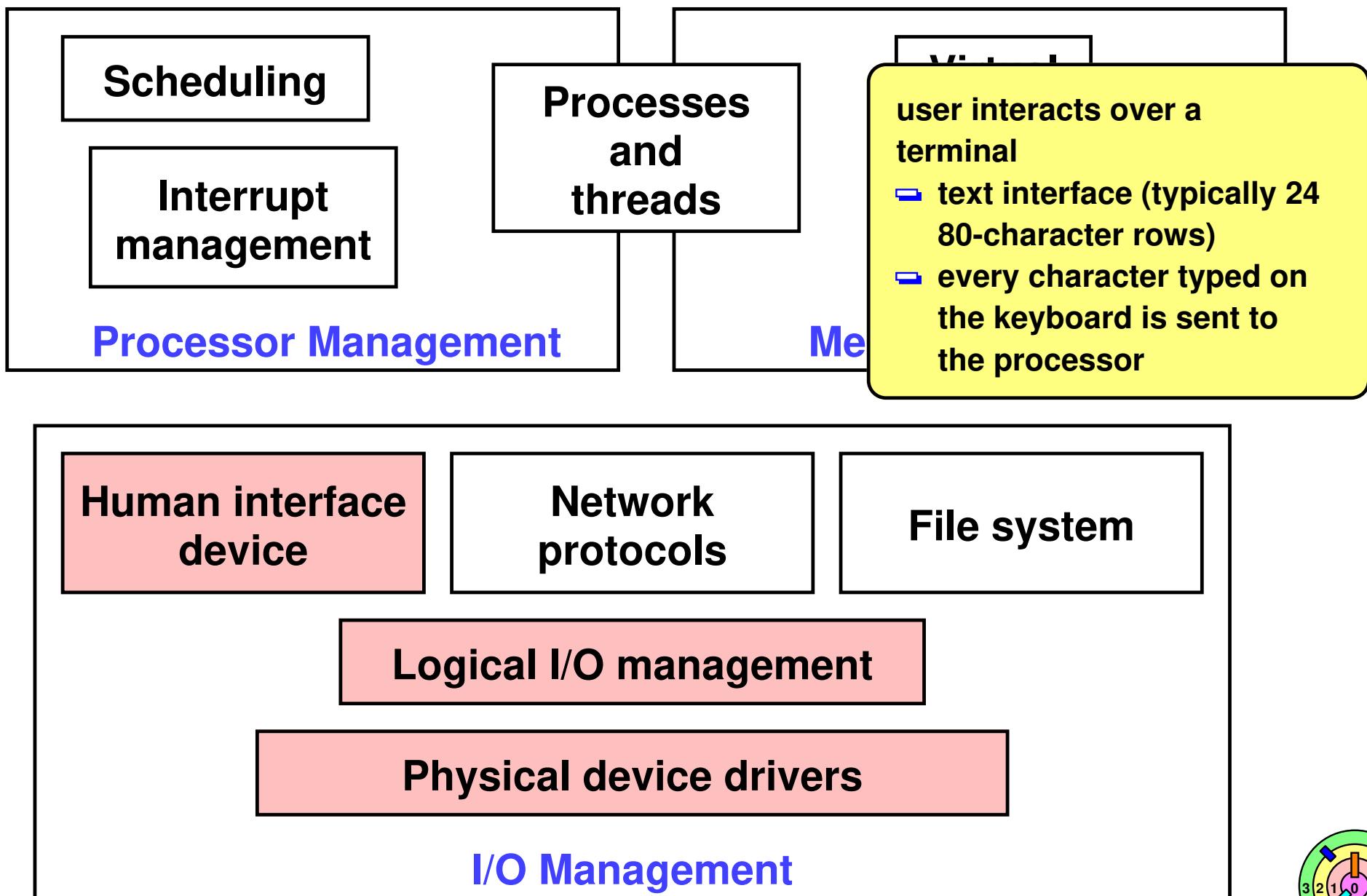
# OS Components



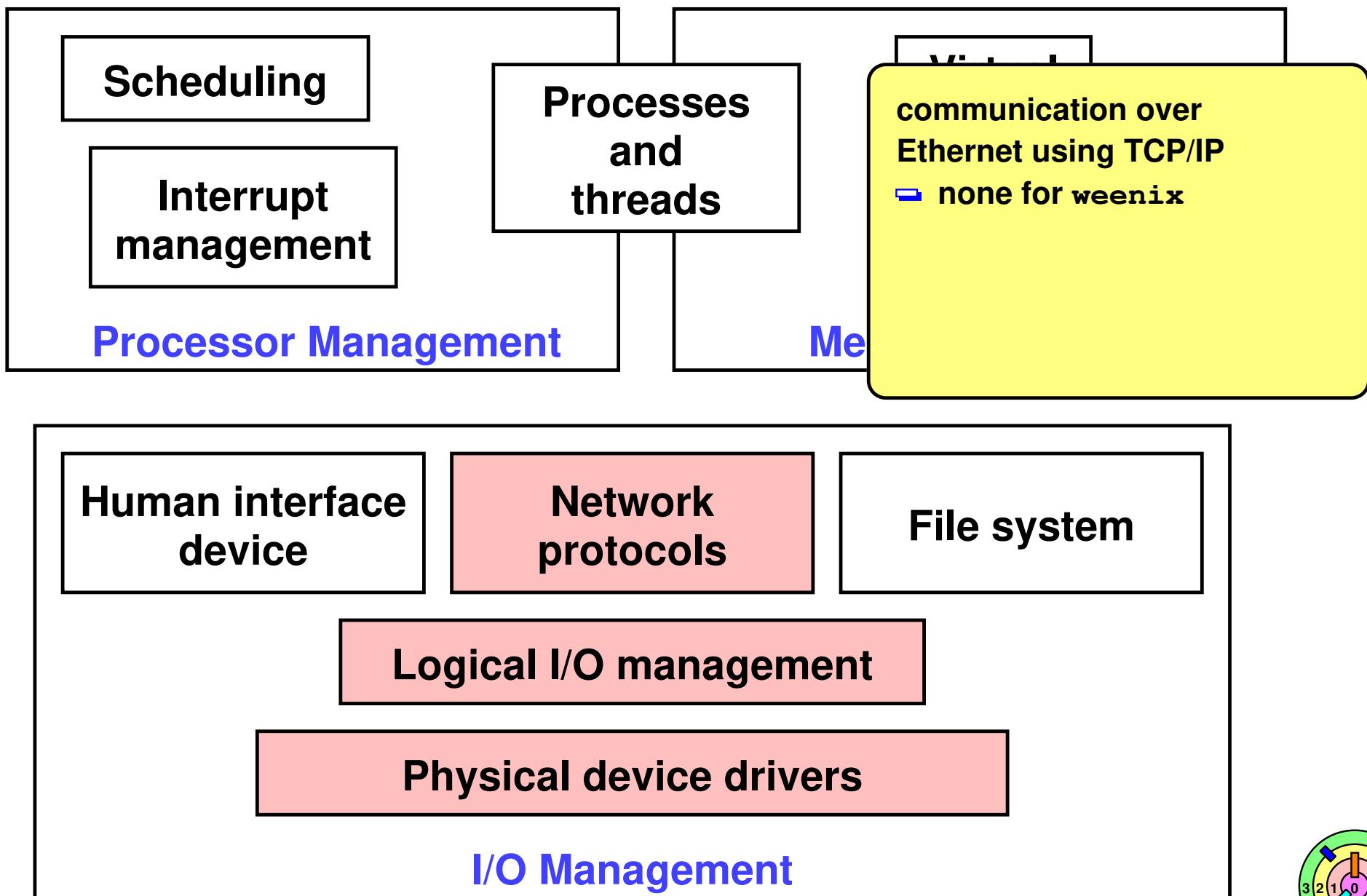
# OS Components



# OS Components

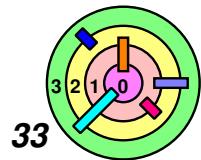


# OS Components



# Some Important OS Concepts

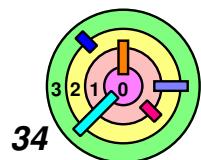
- ➔ **From an application program's point of view, our system has:**
  - processes with threads
  - a file system
  - terminals (with keyboards)
  - a network connection
- ➔ **Need more details on these... Need to look at:**
  - how can they be provided
  - how applications use them
  - how this affects the design of the OS



# Processes And File Systems

- ➡ The purpose of a *process*
  - holds an *address space*
  - holds a group of *threads* that execute within that address space
  - holds a collection of references to *open files* and other "execution context"
- ➡ *Address space:*
  - set of addresses that threads of the process can usefully reference
  - more precisely, it's the content of these addressable locations
    - text, data, bss, dynamic, stack segments/regions and what's in them
      - ◆ a *memory segment/region* contains usable *contiguous* memory addresses

su21-den-Q4



# Address Space Initialization



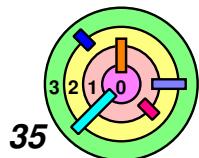
Design issue:

- how should the OS *initialize* these address space regions?



Unix does it in two steps

- make a copy of the address space using `fork()`
- then copy contents from the file system to the process address space (as part of the `exec` operation)
- quite wasteful (both in space and time) for the text region since it's read-only data
  - should *share* the text region
- what about data regions? they can potentially be written into
  - can also *share* a portion of a data region if that portion is never modified
  - copy data structures are much faster than copy data



# Remember This?

## Virtual Memory

### Text

|         |       |
|---------|-------|
| main    | 4096  |
| subr    | 4132  |
| printf  | 4156  |
| write   | 16156 |
| startup | 16172 |

### Data

|               |       |
|---------------|-------|
| ax            | 16384 |
| printfargs    | 16388 |
| StandardFiles | 16396 |

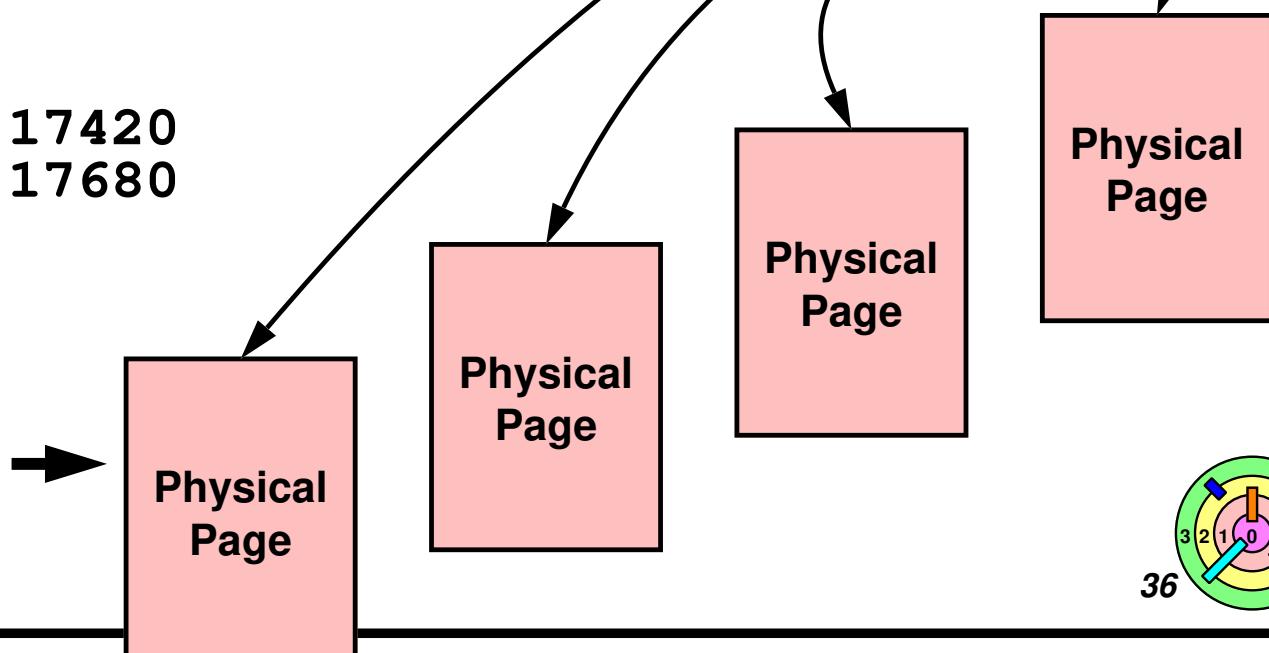
### BSS

|       |       |
|-------|-------|
| x     | 17420 |
| errno | 17680 |

ask buddy system to  
allocate these pages

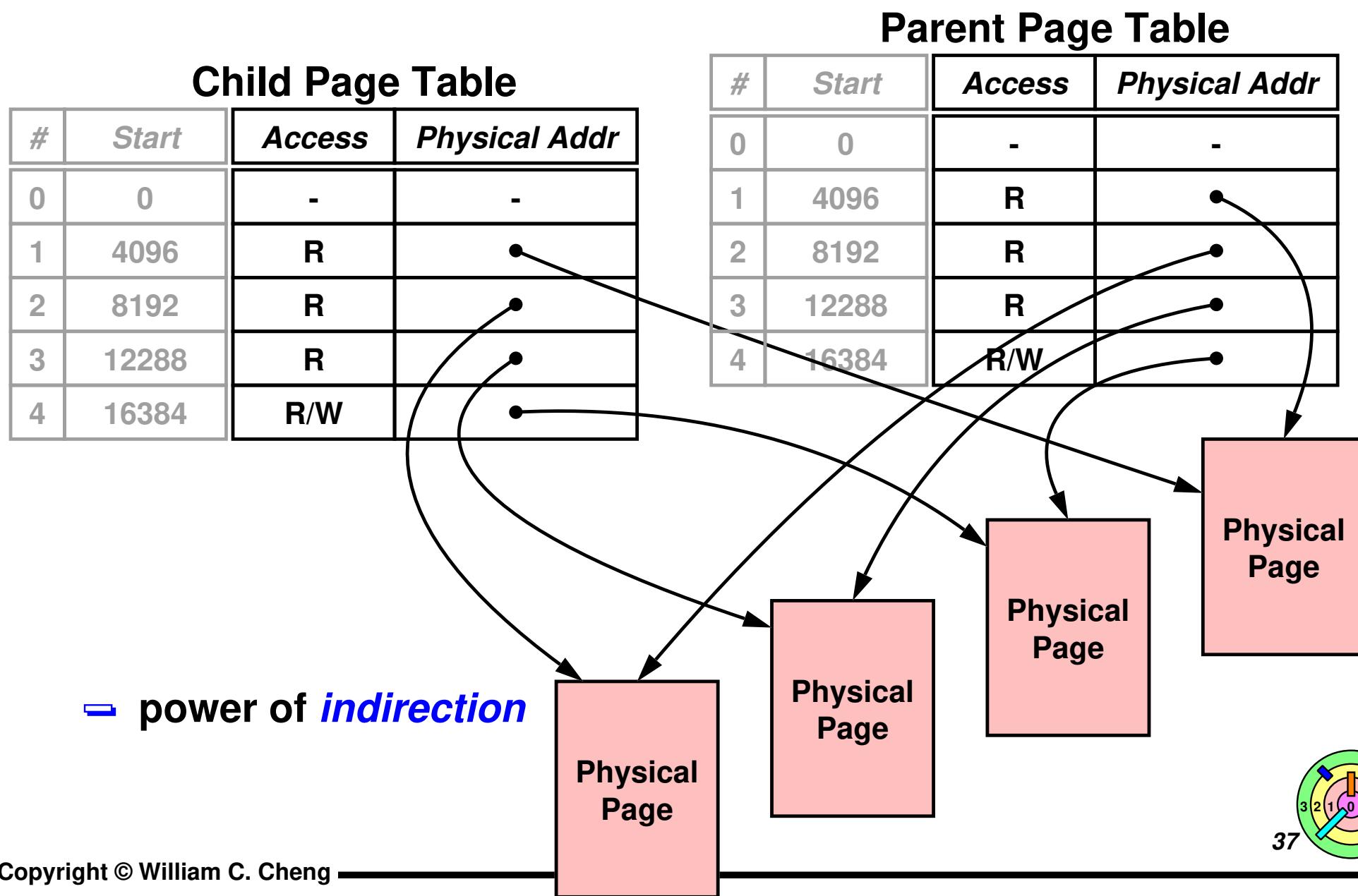
### Page Table

| # | Start | Access | Physical Addr |
|---|-------|--------|---------------|
| 0 | 0     | -      | -             |
| 1 | 4096  | R      | ●             |
| 2 | 8192  | R      | ●             |
| 3 | 12288 | R      | ●             |
| 4 | 16384 | R/W    | ●             |



# Processes Can Share Memory Pages

→ Inside `fork()`, can simply copy parent's page table to child

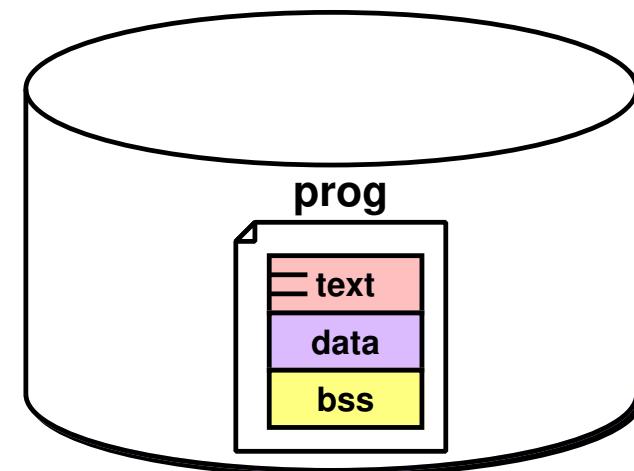


# exec()

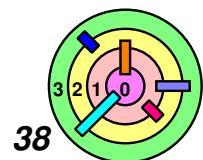
→ Inside `exec()`, need to wipe out the address space (and page table) and create a new address space (and page table)

**Child Page Table**

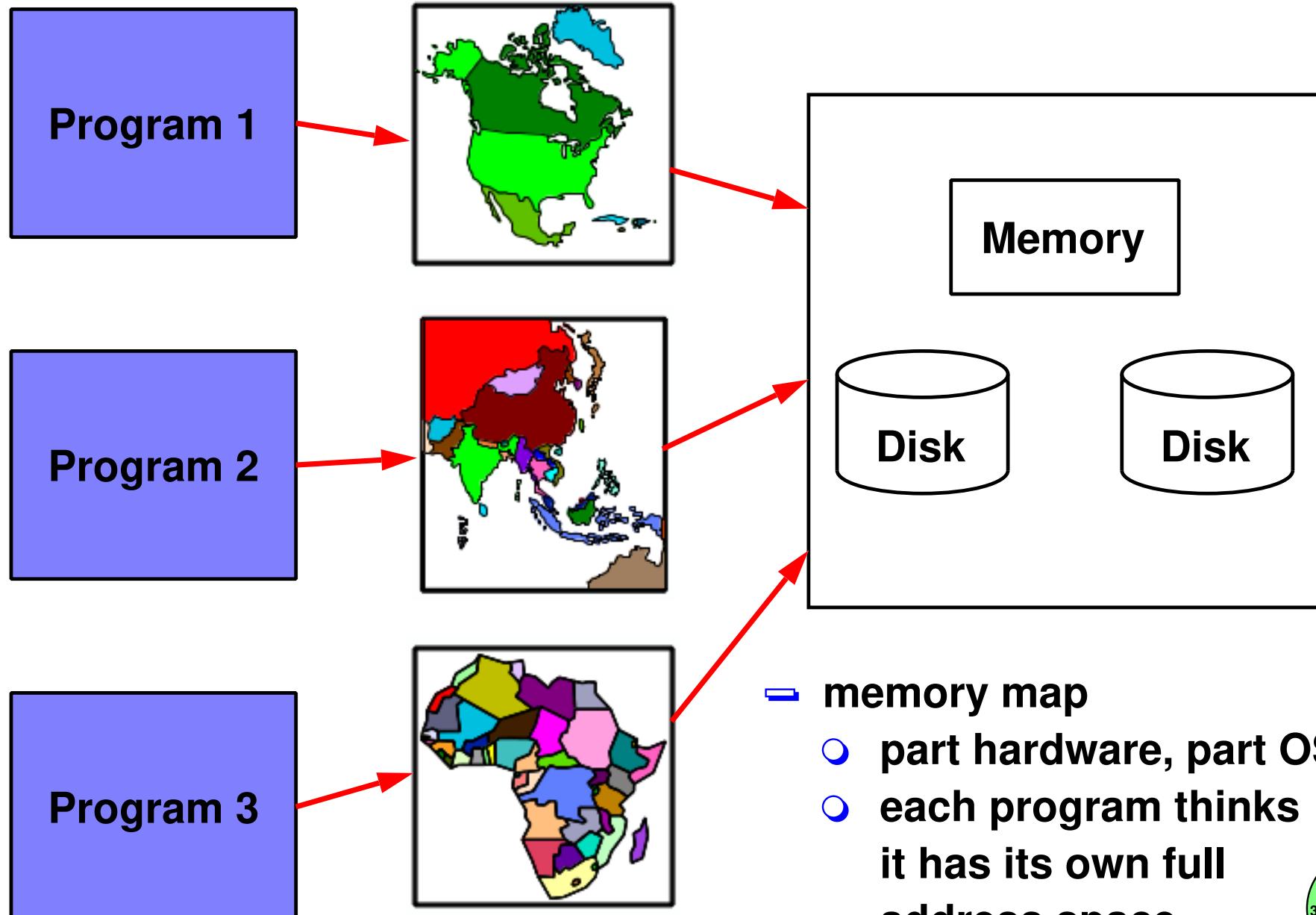
| # | Start | Access | Physical Addr |
|---|-------|--------|---------------|
| 0 | 0     | -      | -             |
| 1 | 4096  | -      | -             |
| 2 | 8192  | -      | -             |
| 3 | 12288 | -      | -             |
| 4 | 16384 | -      | -             |



- should you copy text and data segments of the new program from disk into memory now?
  - can be quite wasteful if you quit your new program quickly (and only use a small amount of the data you just copied from disk)



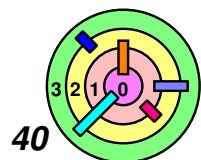
# Memory Map



# Memory Map

→ For the text region, why bother copying the executable file into the address space in the first place?

- can just *map* the *file* into the *address space* (Ch 7)
  - *mapping* is an important concept in the OS
    - ◊ file mapping is *not* the same thing as address translation
    - ◊ some virtual memory pages map to files, and some map to physical memory
  - *mapping* let the OS *tie* the regions of the address space to the file system
  - address space and files are divided into pieces, called *pages*
  - if several processes are executing the same program, then at most one copy of that program's text page is in memory at once
- *text regions* of all processes running this program are setup, using hardware address translation facilities, to share these pages
  - this type of mapping is known as *shared mapping*

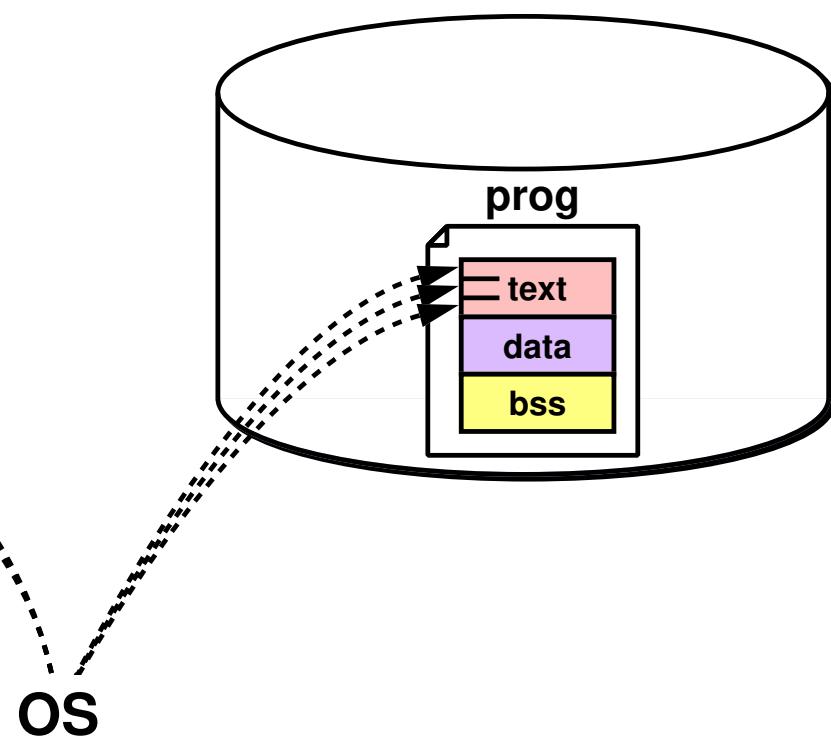


# Memory Map

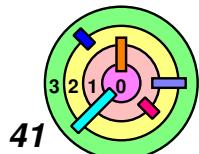
→ The kernel uses a *memory map* to keep track of the mapping from *virtual pages* to *file pages*

**Child Page Table**

| # | Start | Access | Physical Addr |
|---|-------|--------|---------------|
| 0 | 0     | -      | -             |
| 1 | 4096  | -      | -             |
| 2 | 8192  | -      | -             |
| 3 | 12288 | -      | -             |
| 4 | 16384 | -      | -             |



- the kernel also uses *memory map* to keep track of the mapping from *virtual pages* to *physical pages*
  - also use it to maintain the page table data structure

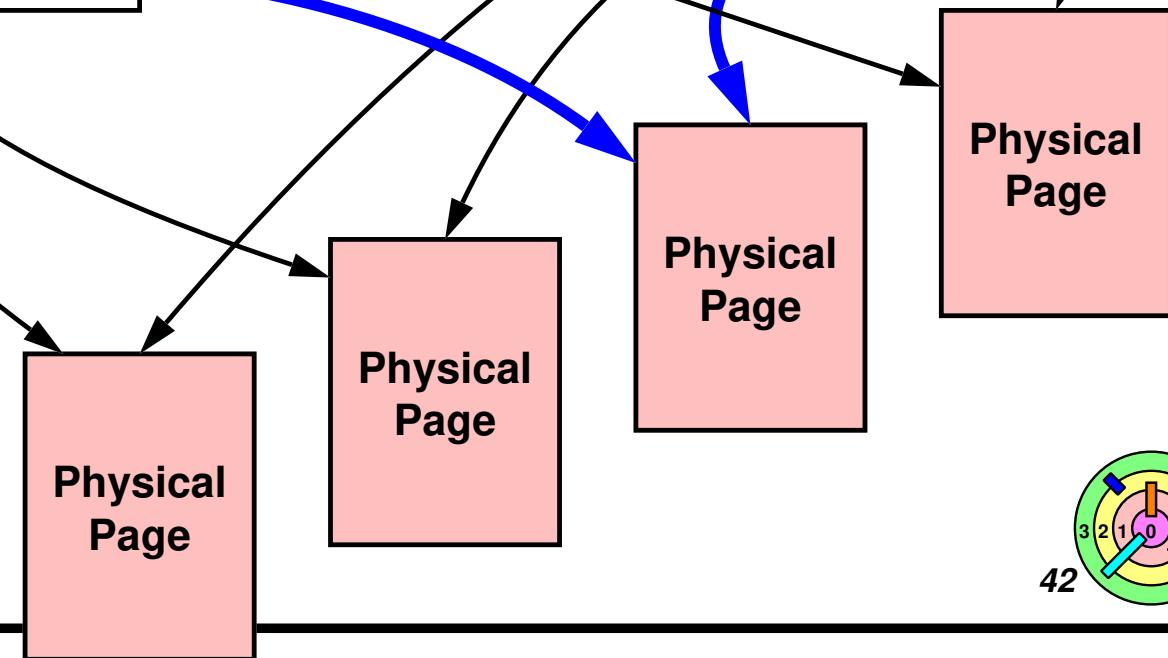


# Processes Can Share Memory Pages

| Child Page Table |       |        |               |
|------------------|-------|--------|---------------|
| #                | Start | Access | Physical Addr |
| 0                | 0     | -      | -             |
| 1                | 4096  | R      | •             |
| 2                | 8192  | R      | •             |
| 3                | 12288 | R      | •             |
| 4                | 16384 | R/W    | •             |

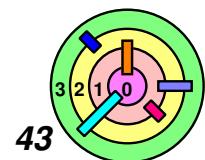
| # | Start | Access | Physical Addr |
|---|-------|--------|---------------|
| 0 | 0     | -      | -             |
| 1 | 4096  | R      | •             |
| 2 | 8192  | R      | •             |
| 3 | 12288 | R      | •             |
| 4 | 16384 | R/W    | •             |

- can we really share *data segment pages*?



# Address Space Initialization

- ▶ ***Text regions*** uses ***shared mapping***
- ▶ ***Data regions*** of all processes running this program ***initially*** refer to pages of memory containing the ***initial*** data region
  - this type of mapping is known as ***private mapping***
    - when does each process really need a private copy of such a page?
      - ◊ when data is ***modified*** by a process, it gets a ***new*** and ***private*** copy of the initial page



F20-Q11 S21-Q16

# Copy-On-Write

su21-a-Q14



## ***Copy-on-write (COW):***

- a process gets a ***private*** copy of the page after a thread in the process performs a ***write*** to that page for the ***first time***
  - the basic idea is that only those pages of memory that are modified are copied



**Use private mapping and copy-on-write for data and bss regions**



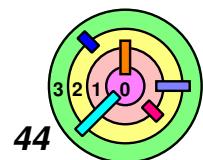
**The dynamic/heap and stack regions use a special form of private mapping**

- their pages are initialized, with zeros (in Linux); ***copy-on-write***
  - these are known as ***anonymous pages***



**If we can implement copy-on-write at the right time, then it's perfectly okay for processes to share address spaces**

- details in Ch 7



# Shared Files



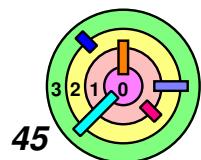
If a bunch of processes *share* a *file*

- we can also *map* the file into the *address space* of each process
- in this case, the mapping is *shared*
- when one process modifies a page, no private copy is made
  - instead, the original page itself is modified
  - everyone gets the changes
  - and changes are written back to the file
    - ◆ more on issues in Ch 6



Can also share a file read-only

- writing through such a map will cause segmentation fault



# Memory Maps Summary

## → File mapping

- shared mapping

- R/W: may change shared data *on disk*

- R/O: read-only

- private mapping

- R/W: copy-on-write (will *not* change data on disk)

- R/O: read-only

## → Anonymous mapping

- shared mapping (may be just shared with child processes)

- R/W: may change shared data *in memory*

- R/O: read-only

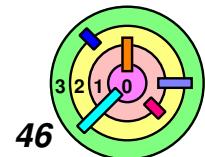
- private mapping

- R/W: copy-on-write

- R/O: read-only

## → Can also use all of the above in an application

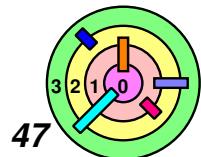
- `mmap()` system call



S21-Q9su21-den-Q7

# Block I/O vs. Sequential I/O

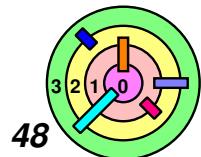
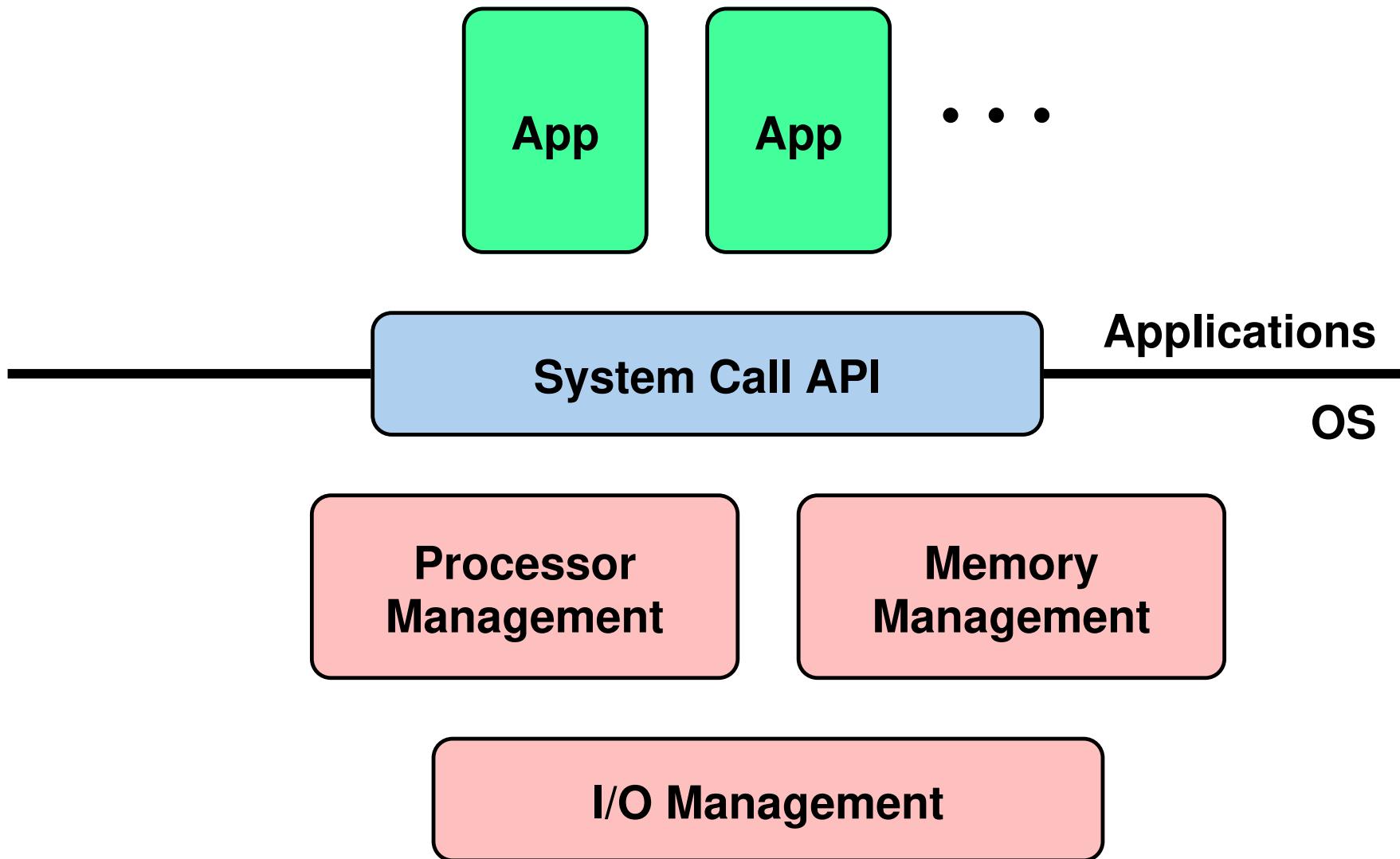
- Mapping files into address space is one way to perform I/O on files
  - block/page is the basic unit
  - this is referred to as **block I/O**
- Some devices cannot be mapped into the address space
  - e.g., receiving characters typed into the keyboard, sending a message via a network connection
  - need a more traditional approach using explicit system calls such as `read()` and `write()`
  - this is referred to as **sequential I/O**
- It also makes sense to be able to read a file like reading from the keyboard
  - similarly, a program that produces lines of text as output should be able to use the same code to write output to a file or write it out to a network connection
  - makes life easier! (and make code more robust)



# System Call API

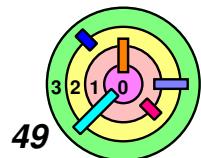
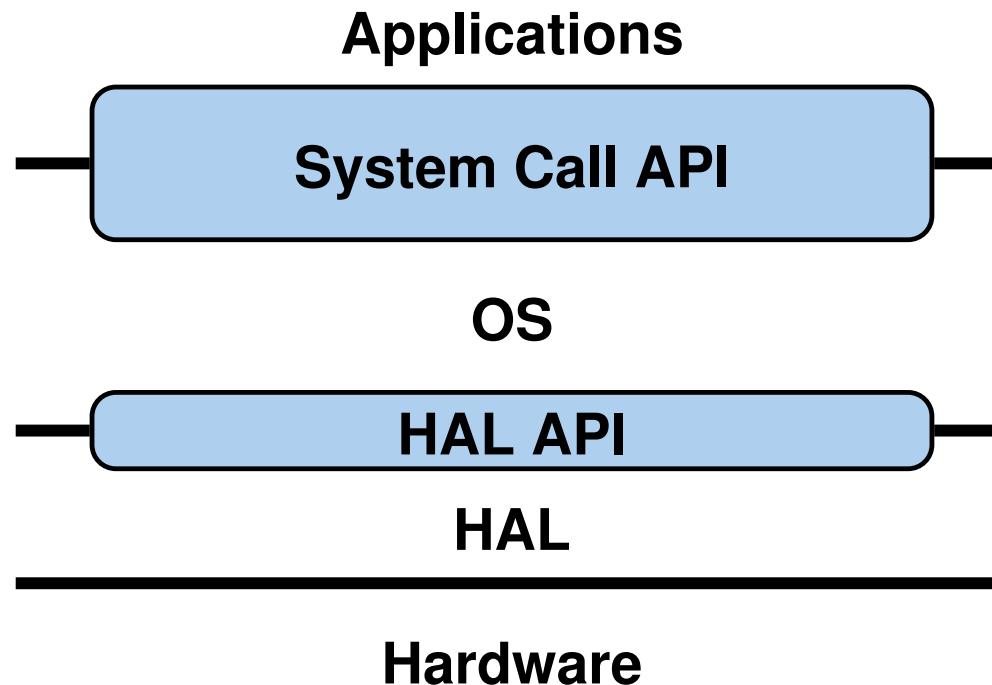


- Backwards compatibility is an important issue
  - try not to change it much (to make developers happy)



# Portability

- It is desirable to have a portable operating system
  - portable across various hardware platforms
- For a monolithic OS, it is achieved through the use of a **Hardware Abstraction Layer (HAL)**
  - a *portable interface to machine configuration* and *processor-specific operations* within the kernel

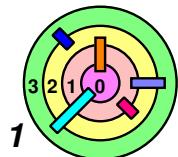


# Hardware Abstraction Layer (HAL)

- ➔ Portability across machine configuration
  - e.g., different manufacturers for x86 machines will require different code to configure interrupts, hardware timers, etc.
- ➔ Portability across processor families
  - e.g., may need additional code for context switching, system calls, interrupting handler, virtual memory management, etc.
- ➔ With a well-defined Hardware Abstraction Layer, most of the OS is *machine and processor independent*
  - porting an OS to a new computer is done by
    - writing new HAL routines
    - relink with the kernel

# 4.1 A Simple System (Monolithic Kernel)

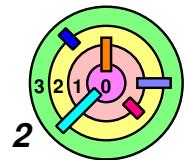
- ▶ **A Framework for Devices**
- ▶ **Low-level Kernel** (will come back to talk about this after Ch 7)
- ▶ **Processes & Threads**
- ▶ **Storage Management** (will come back to talk about this after Ch 5)



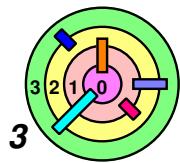
# Computer Terminal



→ VT100



# A "tty"



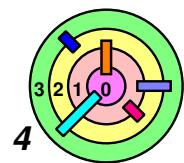
# Devices

## → Challenges in supporting devices

- device independence
- device discovery

## → Device naming

- two choices
  - independent name space (i.e., named independently from other things in the system)
  - devices are named as files



F20-Q2

# A Framework for Devices



## Device driver:

- every device is identified by a device "number", which is actually a pair of numbers
  - a **major device number** - identifies the device driver
  - a **minor device number** - device index for all devices managed by the same device driver



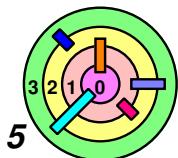
## Special entries were created in the file system to refer to devices

- usually in the `/dev` directory
  - e.g., `/dev/disk1`, `/dev/disk2` each marked as a **special file**
    - ◊ a **special file** does not contain data
    - ◊ it refers to devices by their major and minor device numbers
    - ◊ if you do "`ls -l`", you can see the device numbers



## Data structure in the early Unix systems

- statically allocated array in the kernel called `cdevsw` (character device switch)

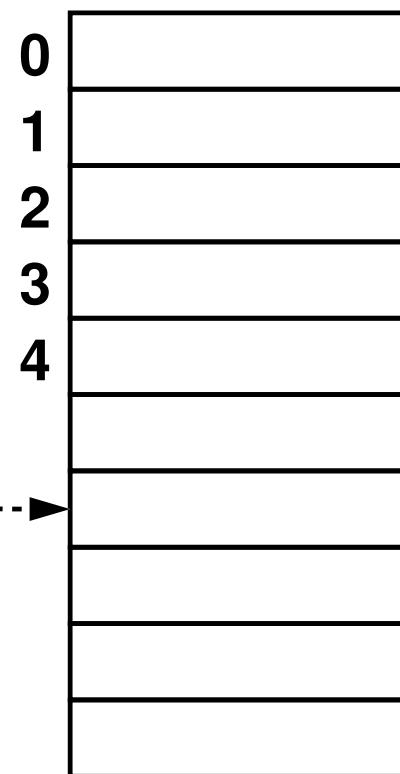


# Finding Devices



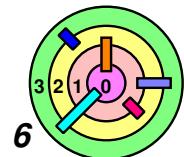
**device number:**  
**major = 6**  
**minor = 1**

- entries in **cdevsw** contains addresses of the device driver entry points
  - a device driver maintains its own data structure



**cdevsw**

{ **read entry point**  
**write entry point**  
**mmap**  
**...** }



# Device Drivers in Early Unix Systems

- The kernel was statically configured to contain device-specific information such as:
  - interrupt-vector locations
  - locations of device-control registers on whatever bus the device was attached to
- Static approach was simple, but cannot be easily extended
  - a kernel must be custom configured for each installation



# Device Probing

- First step to improve the old way
  - allow the devices to be found and automatically configured when the system booted
  - (still require that a kernel contain all necessary device drivers)
- Each device driver includes a *probe routine*
  - invoked at boot time
  - probe the relevant buses for devices and configure them
    - including identifying and recording interrupt-vector and device-control-register locations
- This allowed one kernel image to be built that could be useful for a number of similar but not identical installations
  - boot time is kind of long
  - impractical as the number of supported devices gets big



# Device Probing

→ What's the right thing to do?

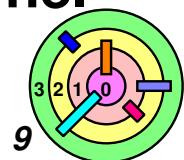
**Step 1:** discover the device without the benefit of having the relevant device driver in the kernel

**Step 2:** find the needed device drivers and dynamically link them into the kernel

- but how do you achieve this?

→ Solution: use meta-drivers

- a meta-drive handles a particular kind of bus
- e.g., USB (Universal Serial Bus)
  - a USB meta-driver is installed into the kernel
  - any device that goes onto a USB (Universal Serial Bus) must know how to interact with the USB meta-driver via the ***USB protocol***
  - once a connected device is identified, system software would select the appropriate device driver and load into the kernel
  - what about applications? how can they reference dynamically discovered devices?



# Discovering Devices

→ So, you plug in a new device to your computer on a particular bus

- OS would notice
- find a device driver
  - what kind of device is it?
  - where is the driver?
- assign a name, but how is it chosen?
- multiple similar devices, but how does application choose?

→ In some Linux systems, entries are added into /dev as the kernel discovers them

- lookup the names from a database of names known as devfs
  - downside of this approach is that device naming conventions not universally accepted
  - what's an application to do?
- some current Linux systems use udev
  - user-level application assigns names based on rules provided by an administrator



# Discovering Devices

→ What about the case where different devices acted similarly?

- e.g., touchpad on a laptop and USB mouse
- how should the choice be presented to applications?

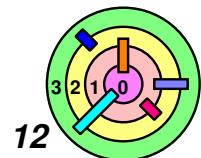
→ Windows has the notion of *interface classes*

- a device can register itself as members of one or more such classes
- an application can *enumerate* all currently connected members of such a class and choose among them (or use them all)



# 4.1 A Simple System (Monolithic Kernel)

- ➔ A Framework for Devices
- ➔ Low-level Kernel (will come back to talk about this after Ch 7)
- ➔ *Processes & Threads*
- ➔ Storage Management (will come back to talk about this after Ch 5)



# Processes and Threads



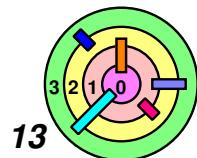
A process is:

- a holder for an *address space*
- a collection of other information shared by a set of *threads*
- a collection of references to *open files* and other "execution context"

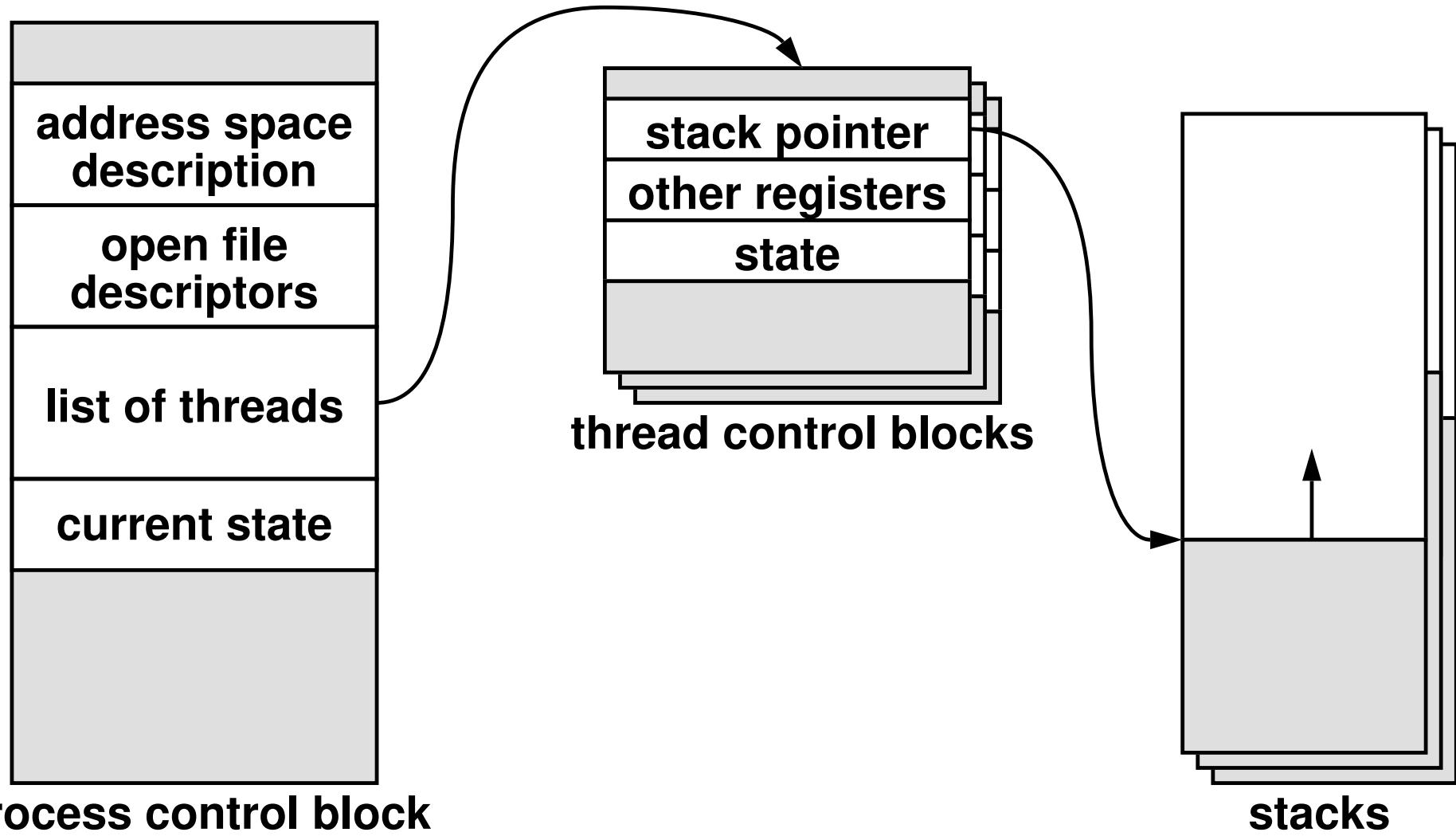


As discussed in Ch 1, processes related APIs include

- `fork()`, `exec()`, `wait()`, `exit()`



# Processes and Threads

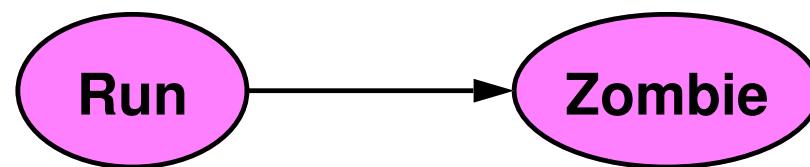


→ Note: all these are relevant to your Kernel Assignment 1  
— although we are only doing *one thread per process*

# Process Life Cycle

→ Pretty simple

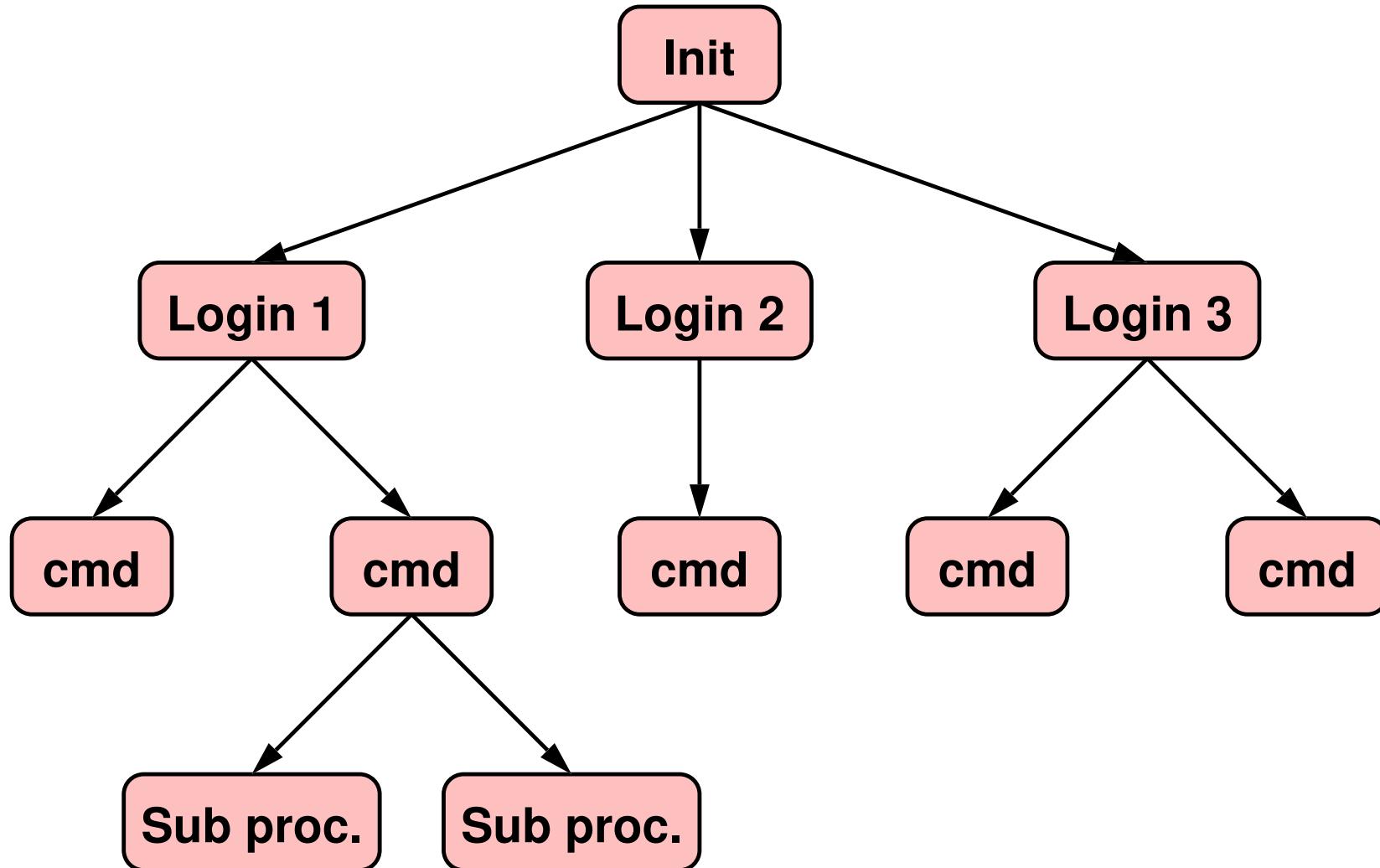
- a process starts in the *run* state



# Process Relationships (1)

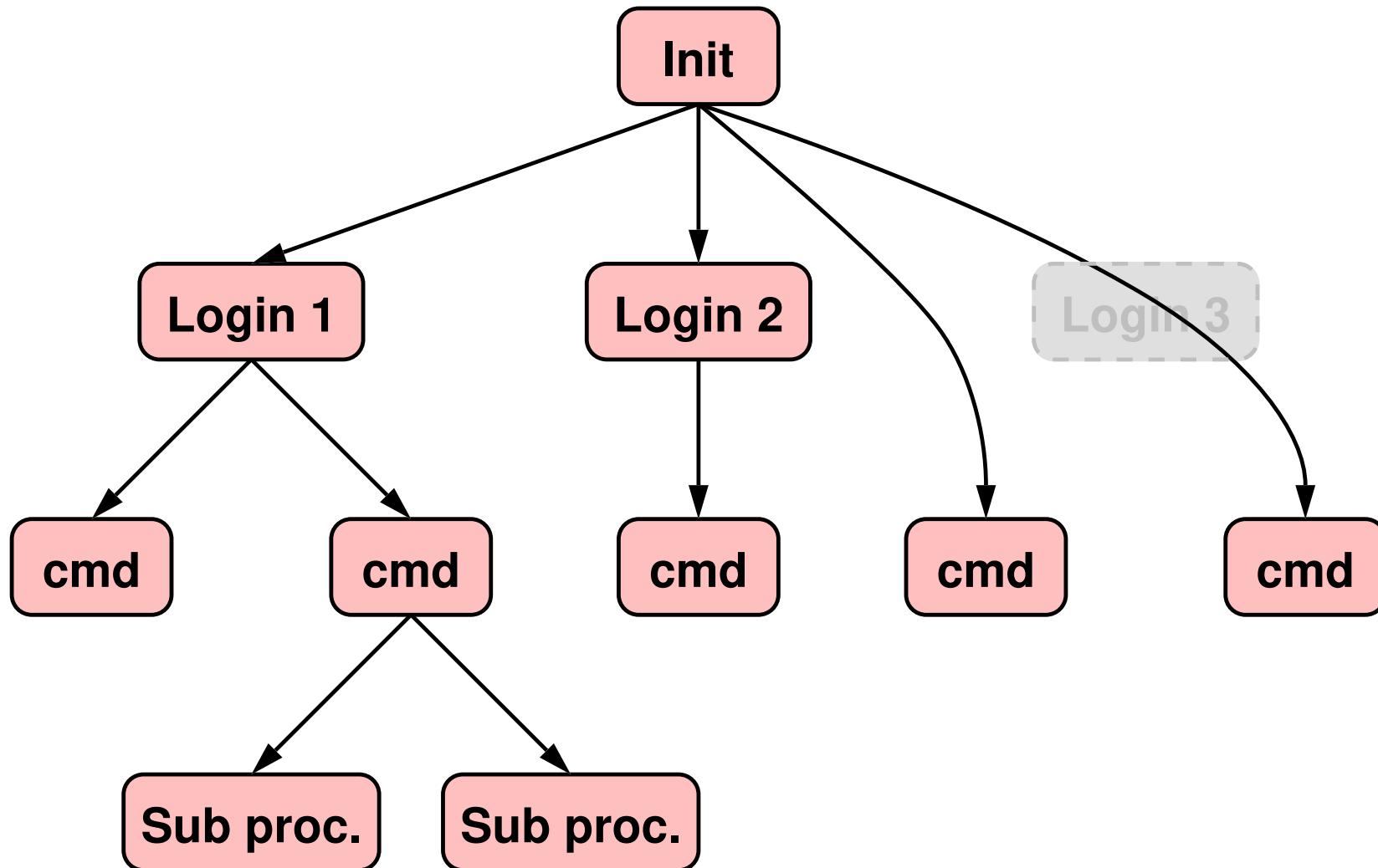
## Process hierarchy

— run "pstree" on Linux



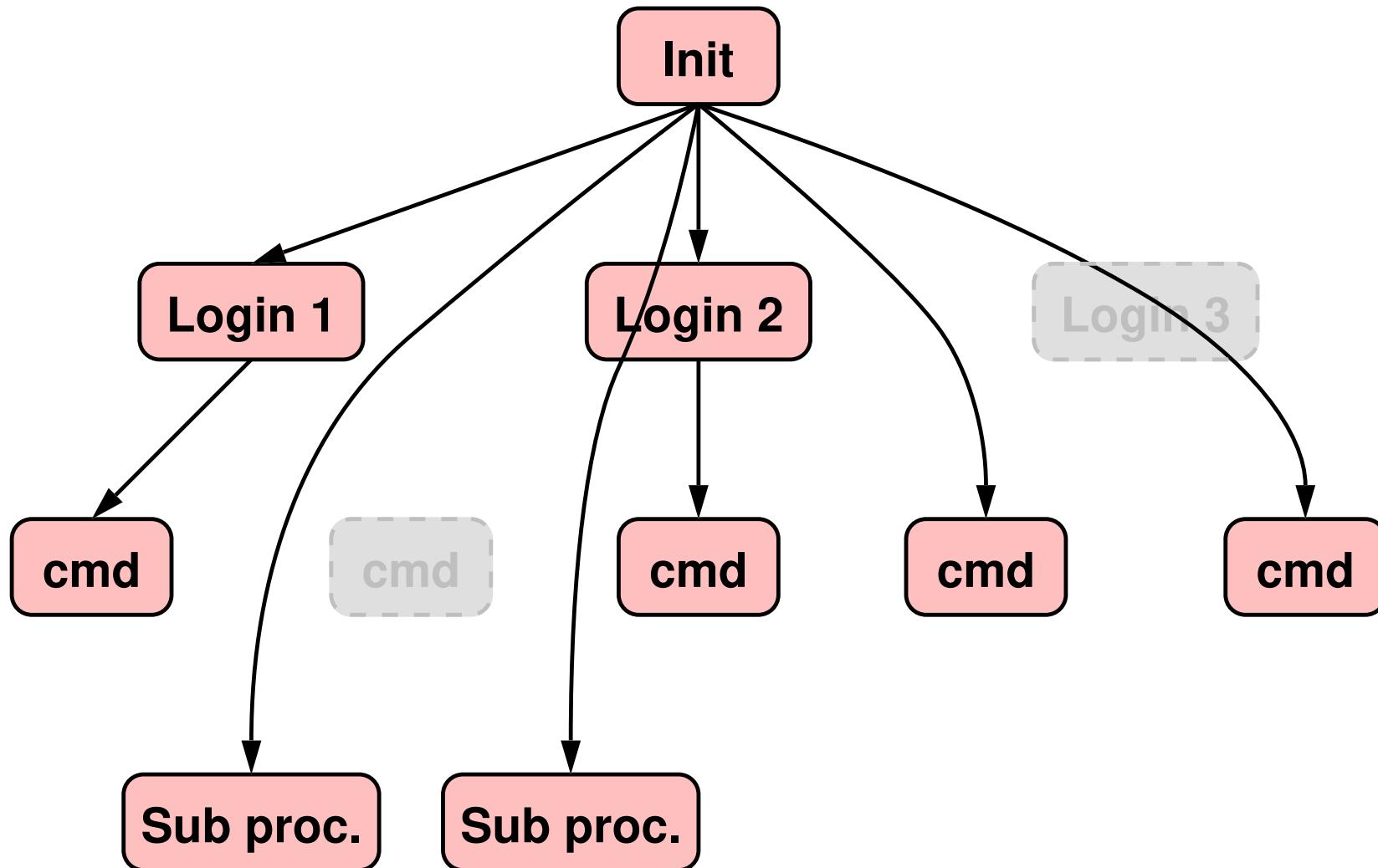
# Process Relationships (2)

→ If a process dies, you must *reparent* all its child processes

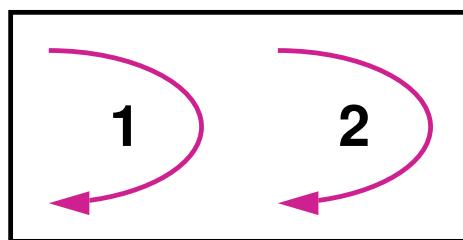


# Process Relationships (3)

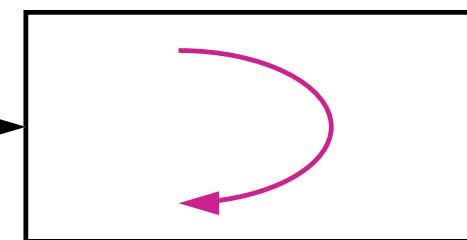
- If a process dies, you must *reparent* all its child processes
  - new parent is the INIT process



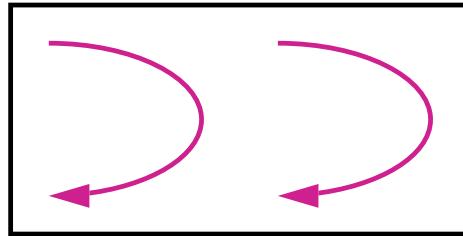
# Fork and Threads



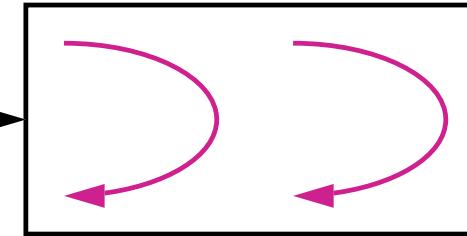
fork



Or

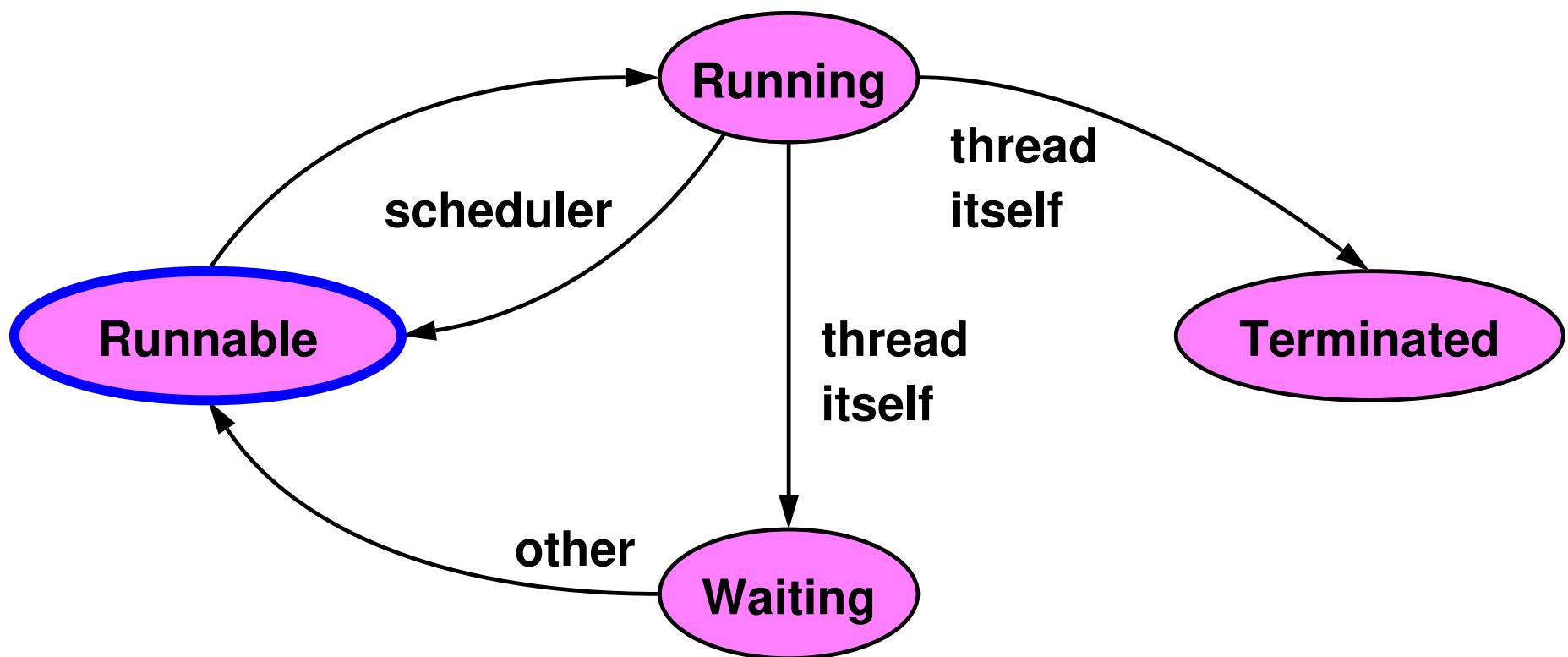


fork

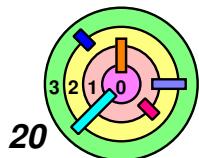


- ▶ Solaris uses the 2nd approach
  - expensive to fork a process
- ▶ Problem with 1st approach
  - thread 1 called `fork()` and thread 2 has a mutex locked
    - who will unlock the mutex?
  - POSIX solution is to provide a way to unlock all mutex before `fork()`

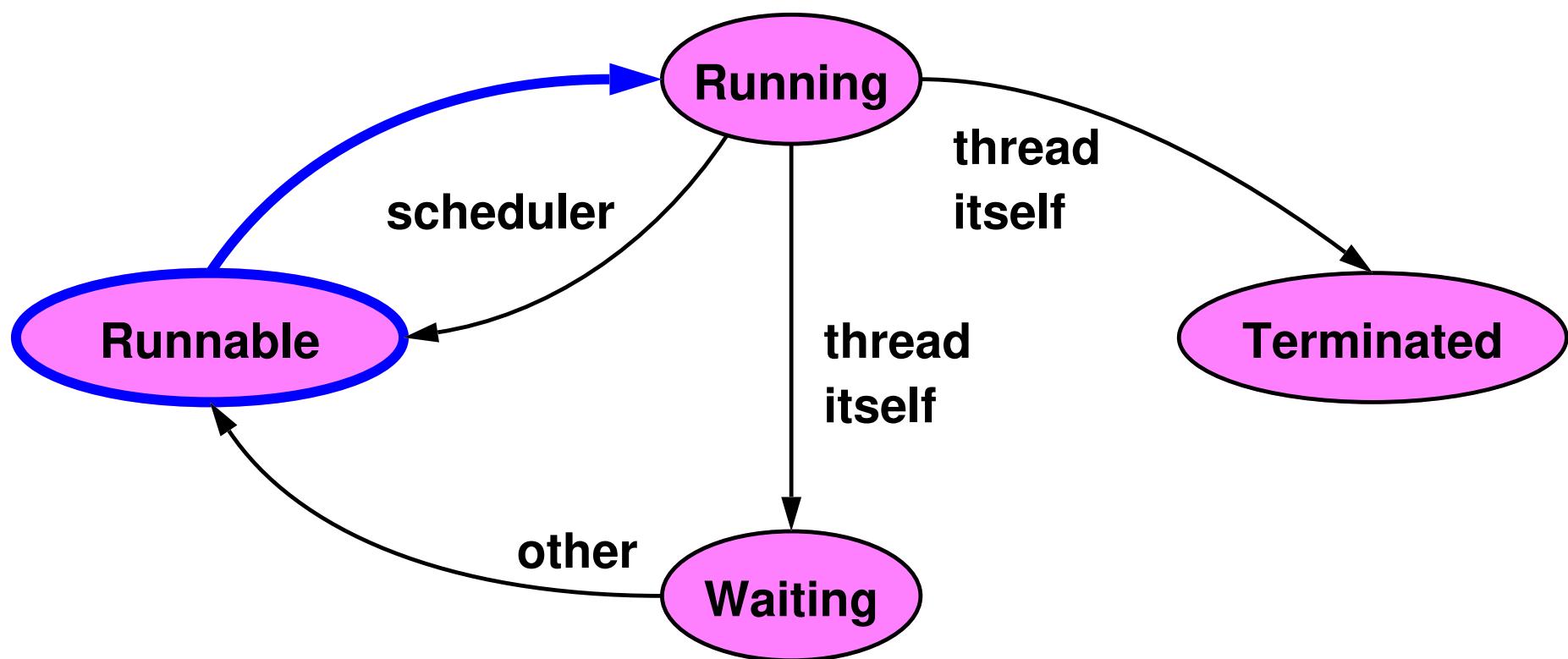
# Thread Life Cycle



- a thread starts in the *Runnable* state
  - *sleeps* in the *run queue* (or "ready queue")
    - ◊ threads sleep in the run queue to wait to use the CPU

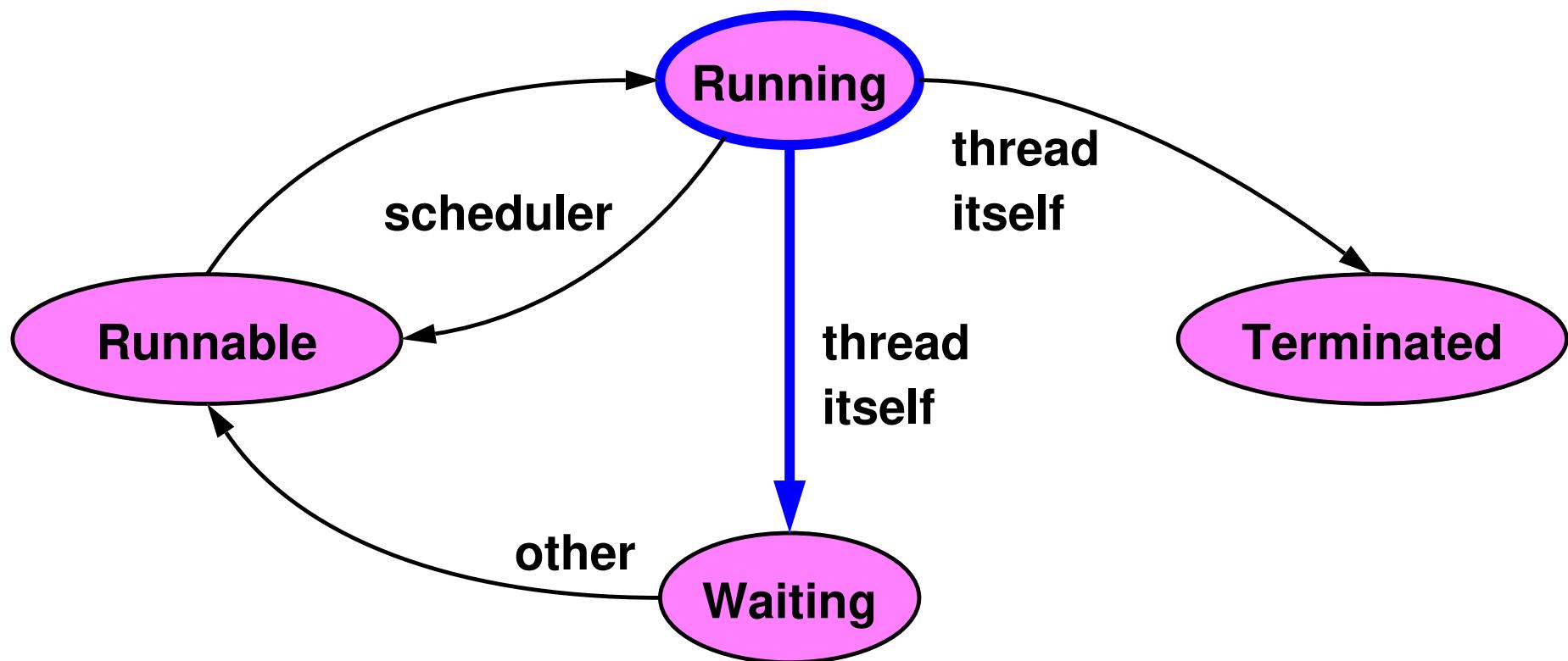


# Thread Life Cycle



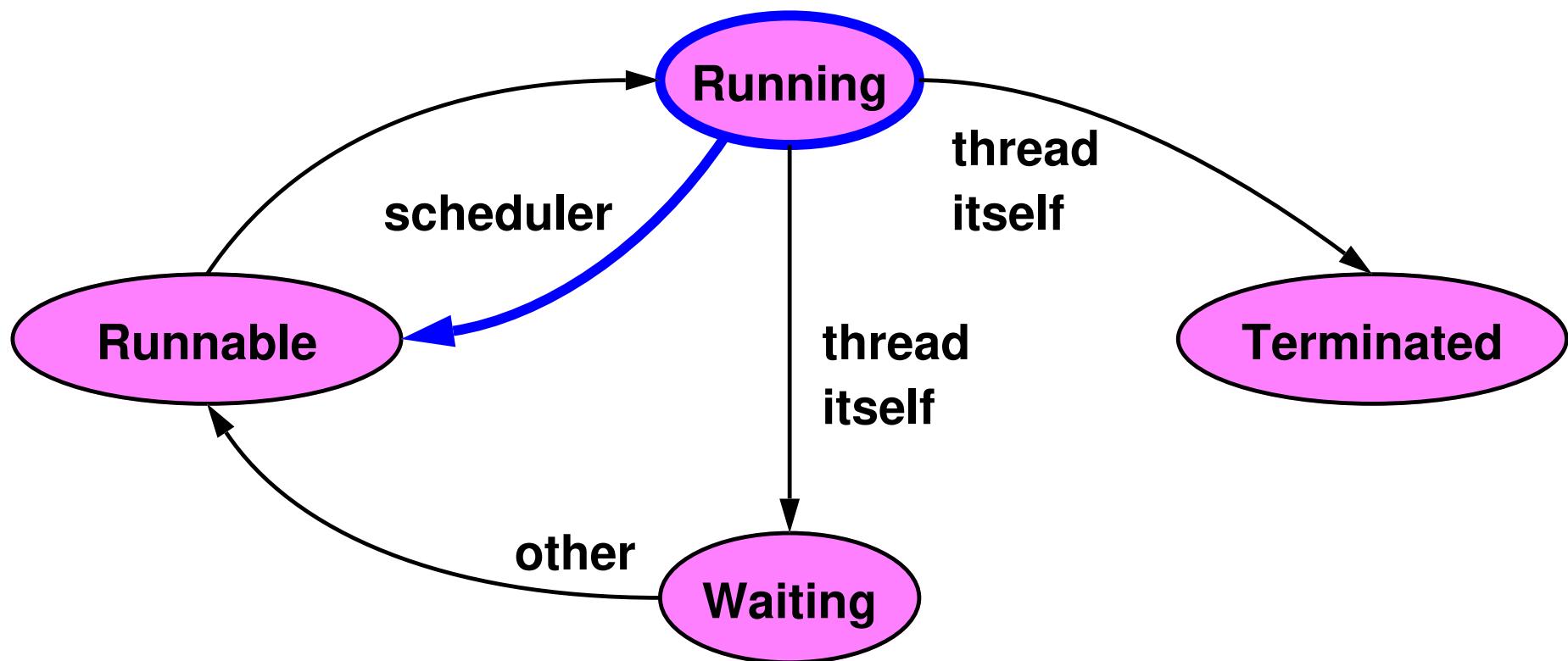
- the **scheduler** switches a thread's state from runnable to running

# Thread Life Cycle

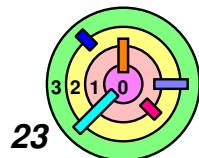


- a thread goes from running to waiting when a **blocking call** is made by the thread itself
  - the scheduler is **not involved** here

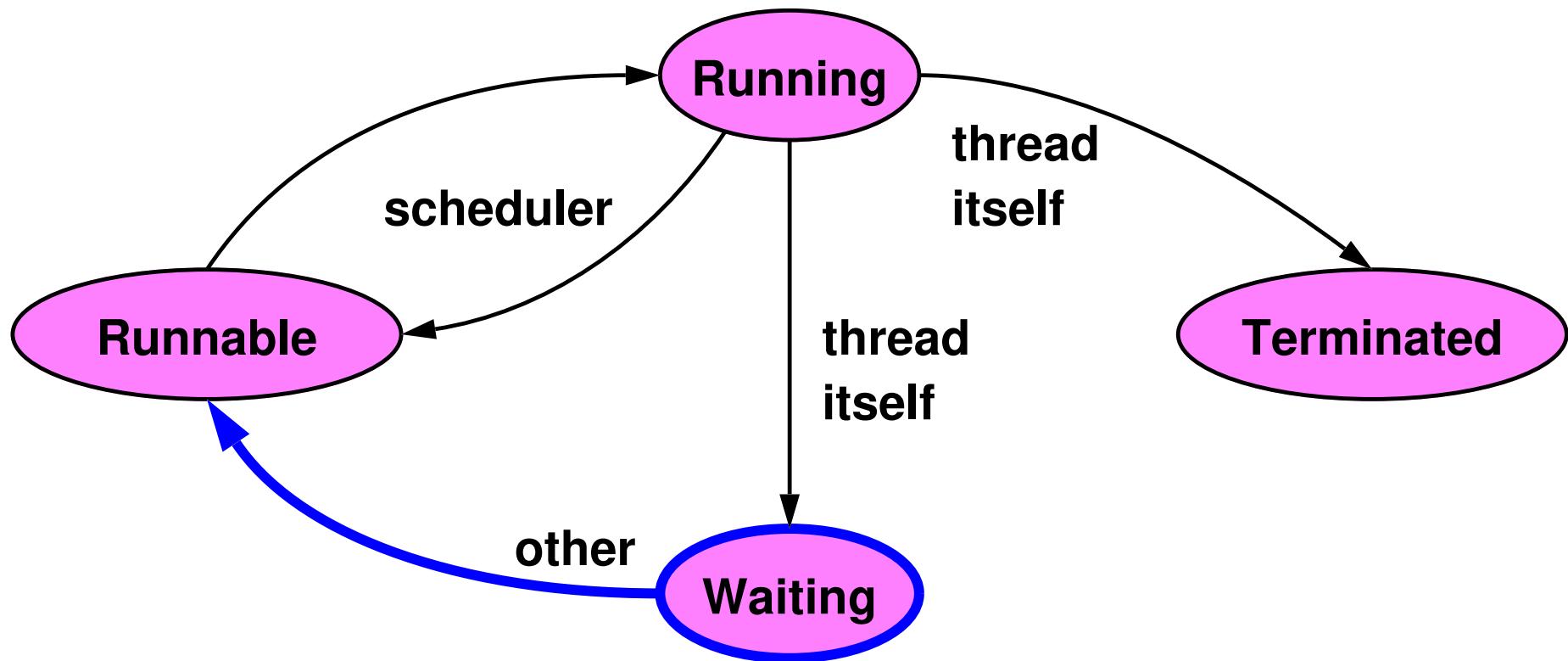
# Thread Life Cycle



- the ***scheduler*** switches a thread's state from running to runnable when the thread used up its execution quantum
  - a thread can also "***yield***" the CPU (see examples in `faber_thread_test()` in kernel 1)

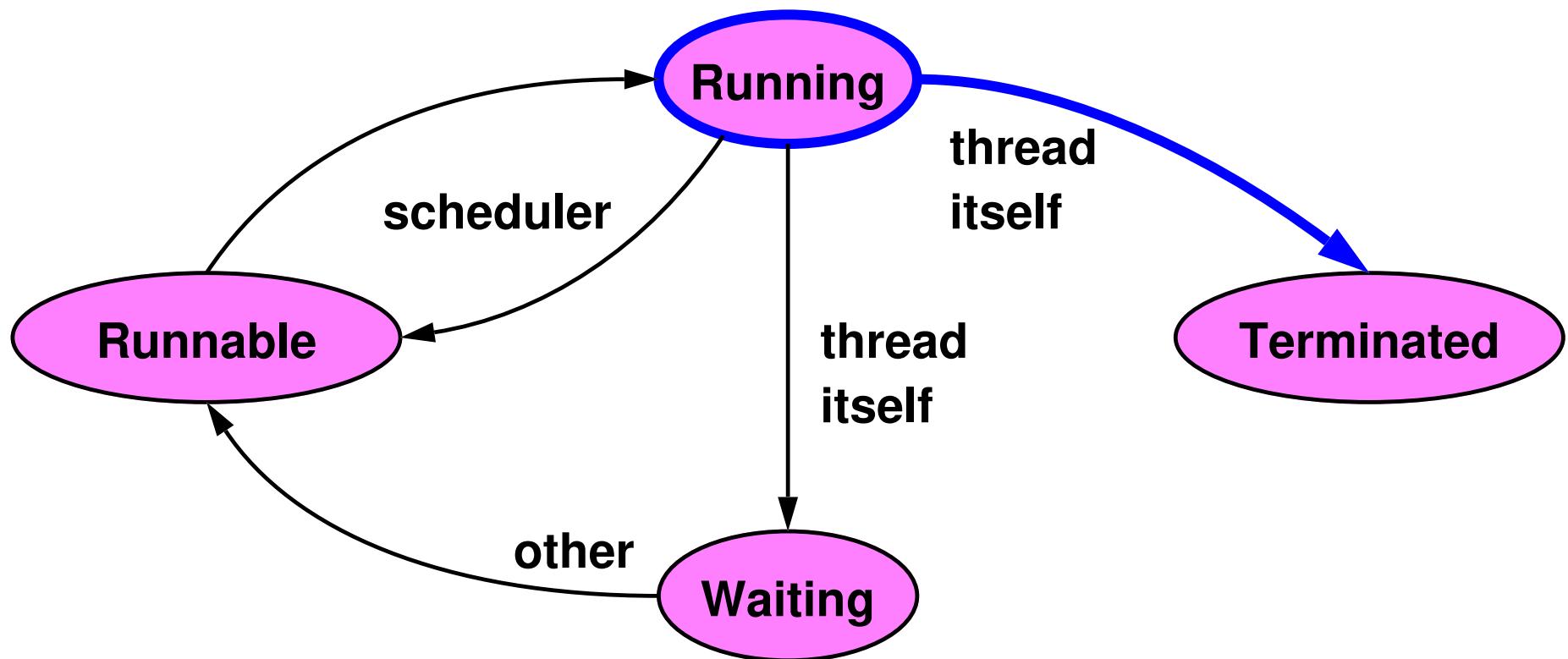


# Thread Life Cycle

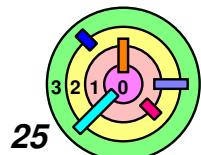


- a thread gets *unblocked* by the action of another thread or by an interrupt handler
  - the scheduler is *not involved* here

# Thread Life Cycle

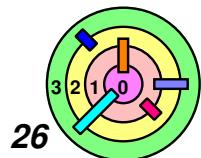


- in order for a thread to enter the terminated state, it has to be in the running state just before that
  - what if something like `pthread_cancel()` is invoked when the thread is not in the running state?



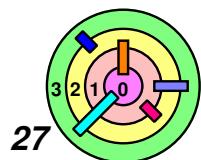
# Thread Life Cycle

- Does `pthread_exit()` delete the thread (completely) that calls it?
  - no, the thread goes into a **zombie state** (i.e., "terminated")
- What's left in the thread after it calls `pthread_exit()`?
  - its **thread control block**
    - needs to keep thread ID and return code around
  - its **stack**
    - how can a thread **delete its own stack**? no way!
      - ◆ which stack are we talking about anyway?



# Thread Life Cycle

- Who is deleting the *thread control block* and freeing up the thread's *stack* space?
- If a thread is not detached
  - it can be taken care of in the `pthread_join()` code
    - the thread that calls `pthread_join()` does the clean up
- If a thread is detached (our simple OS does not support this)
  - can do this in one of two ways
    - 1) use a special *reaper thread*
      - ◊ basically doing `pthread_join()`
    - 2) queue these threads on a list and have other threads free them when it's convenient (e.g., when the scheduler schedule a thread to run)



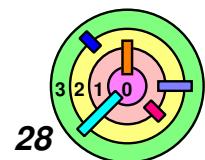
# Kernel 1 Process & Thread Life Cycles

→ Part of the kernel 1 assignment is to implement the *life cycles* of processes and threads

- process/thread creation/termination
  - since we are only doing one thread per process ( $MTP=0$ ), when a thread dies, the process must die as well
- process/thread cancellation
- process waiting (and no thread joining since  $MTP=0$ )
- etc.

→ Unlike warmup2, in kernel assignments, first procedures of almost all kernel threads have been written for you already!

- the thread code there make function calls and some of these functions are *not-yet-implemented*
  - your job is to implement those functions so that these kernel threads can run *perfectly*

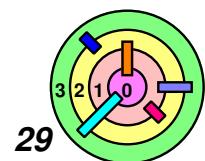


# Kernel 1 Process & Thread Life Cycles

→ Hint on how to do this is by reading kernel code

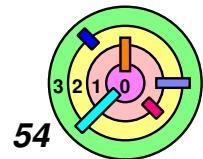
= ***read the code*** in "kernel/proc/faber\_test.c"

- if it calls a function that you are suppose to implement, it's telling you what it's expecting from that function!
  - ◊ feel free to discuss things like that in the class Google Group
- you need to understand what every line of code is doing there
- you need to pass every test there (see grading guidelines)
  - ◊ you must not change anything there
  - ◊ make sure the printout is correct (you may want to discuss it in the class Google Group)
- if you need to do something similar in another module, just ***copy*** the code from it
  - ◊ you can copy code that's given to you as course material and you don't have to cite your source



# 4.1 A Simple System (Monolithic Kernel)

- ➔ A Framework for Devices
- ➔ Low-level Kernel (will come back to talk about this after Ch 7)
- ➔ Processes & Threads
- ➔ *Storage Management*



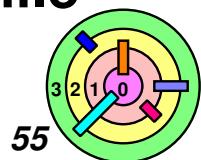
# Storage Management

→ What physical "devices" can you use to store data?

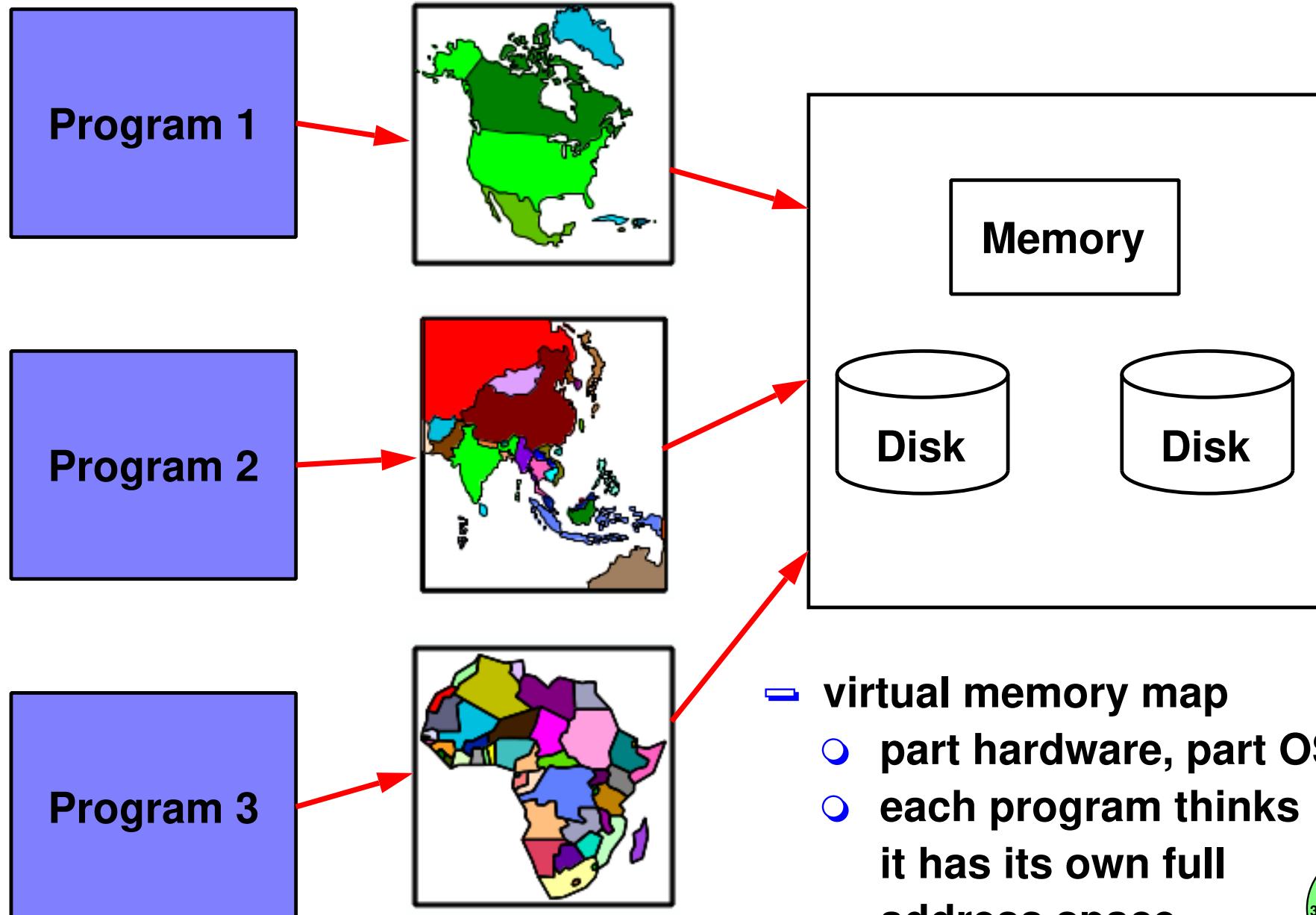
- **primary storage**, i.e., physical memory
  - "directly" addressable (using "one level of indirection")
    - ◊ physical memory is not considered a "device"
- **secondary storage**, i.e., disk-based storage
  - to store files (i.e., implement the abstraction of "files")
  - to support **virtual memory**

→ An application is only aware of **virtual memory** (it thinks virtual memory is real memory)

- applications should not care about how much physical memory is available to it
- there may not be enough physical memory for all processes
- the OS makes sure that real primary storage is available **when necessary**
- e.g., an application can allocate a 1GB block of memory while the machine only has 256MB of physical memory



# Virtual Memory Map

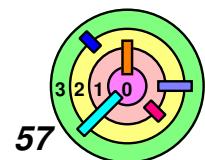


# Storage Management

- Two ways for an application to access secondary storage
  - using **sequential I/O**: `open()`, `read()`, `write()`, `close()`
  - using **block I/O**: `open()`, `mmap()`, `close()`

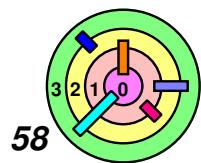
- Memory management concerns

- 1) mapping virtual addresses to real ones
- 2) determining which addresses are valid, i.e., refer to allocated memory, and which are not
- 3) keeping track of which real objects, if any, are mapped into each range of virtual addresses
- 4) deciding what should be kept in primary storage (RAM) and what to fetch from elsewhere



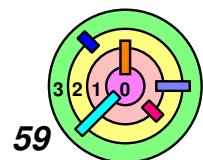
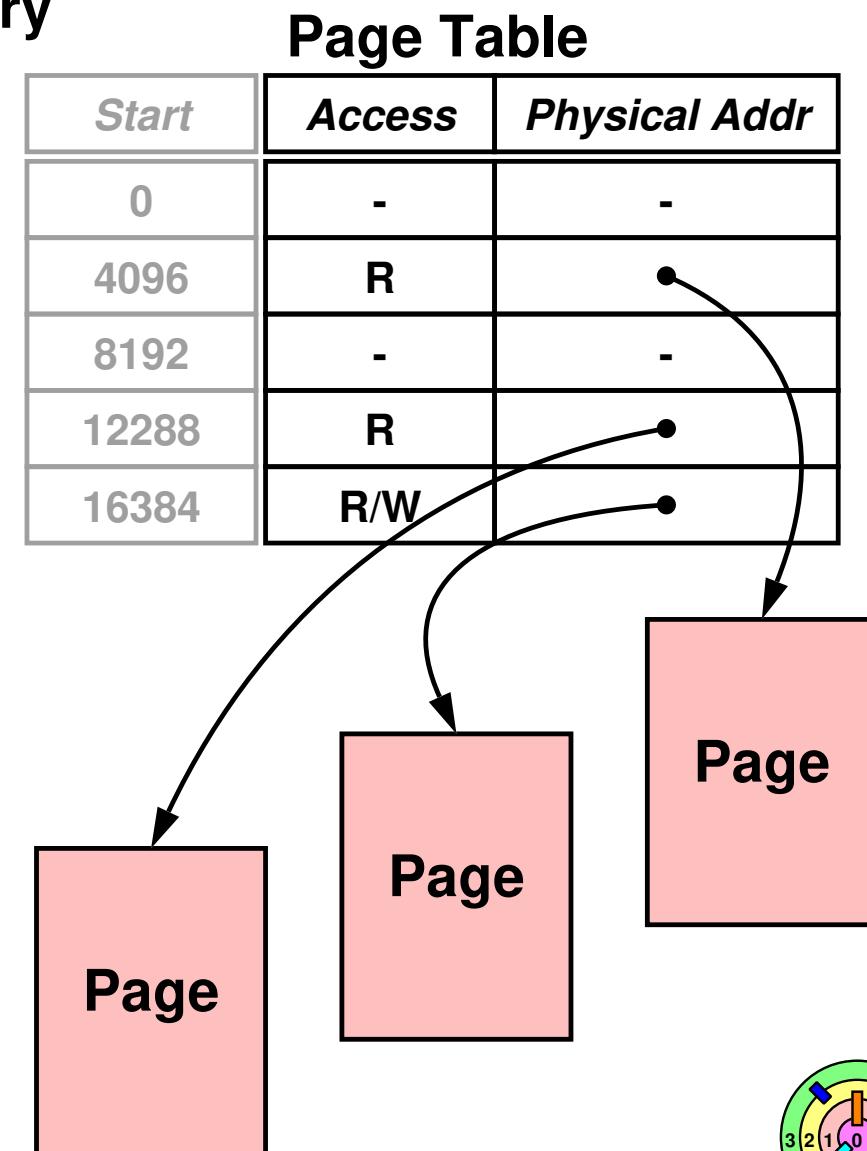
# Memory Management Concerns

- In reality, the OS is too slow since **every** virtual address needs to be resolved
  - some of the virtual memory mechanisms must be built into the **hardware**
    - in some cases, the hardware is given the complete "**map**" (i.e., mapping from virtual to physical address)
    - in other cases, only a partial map is given to the hardware
    - in either case, OS needs to provide some map to the hardware and needs a **data structure** for the map
      - ◆ **page table** is part of the **virtual memory map** (it "maps" virtual address to physical address)
- **Virtual Memory Map (vmmap) data structure in the OS**
  - implements the **address space**
    - only **user part of the address space** needs to be represented
  - implements **memory-mapped files**

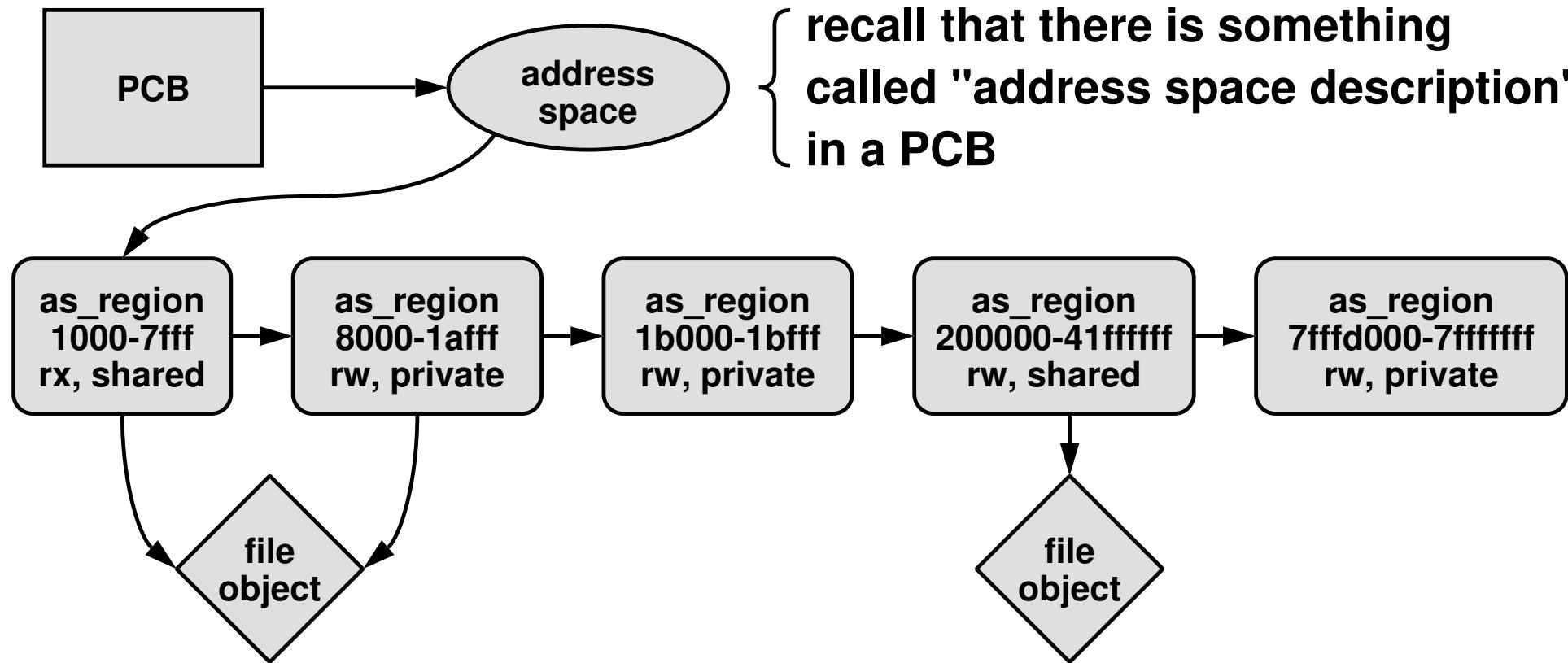


# Memory Management Concerns

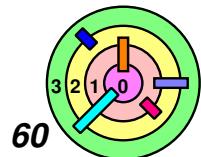
- ➡ A valid virtual address must be ultimately *resolvable* by the OS to a location in the physical memory
  - if it cannot be resolved, the virtual address is considered an *invalid* virtual address
  - referencing an *unresolvable* virtual address causes a *segmentation fault* (the OS will deliver SIGSEG to the process)
    - the default action would be to terminate the process
  - e.g., virtual address 0
- ➡ A *page fault* is not a *segmentation fault* if it *can be resolved*



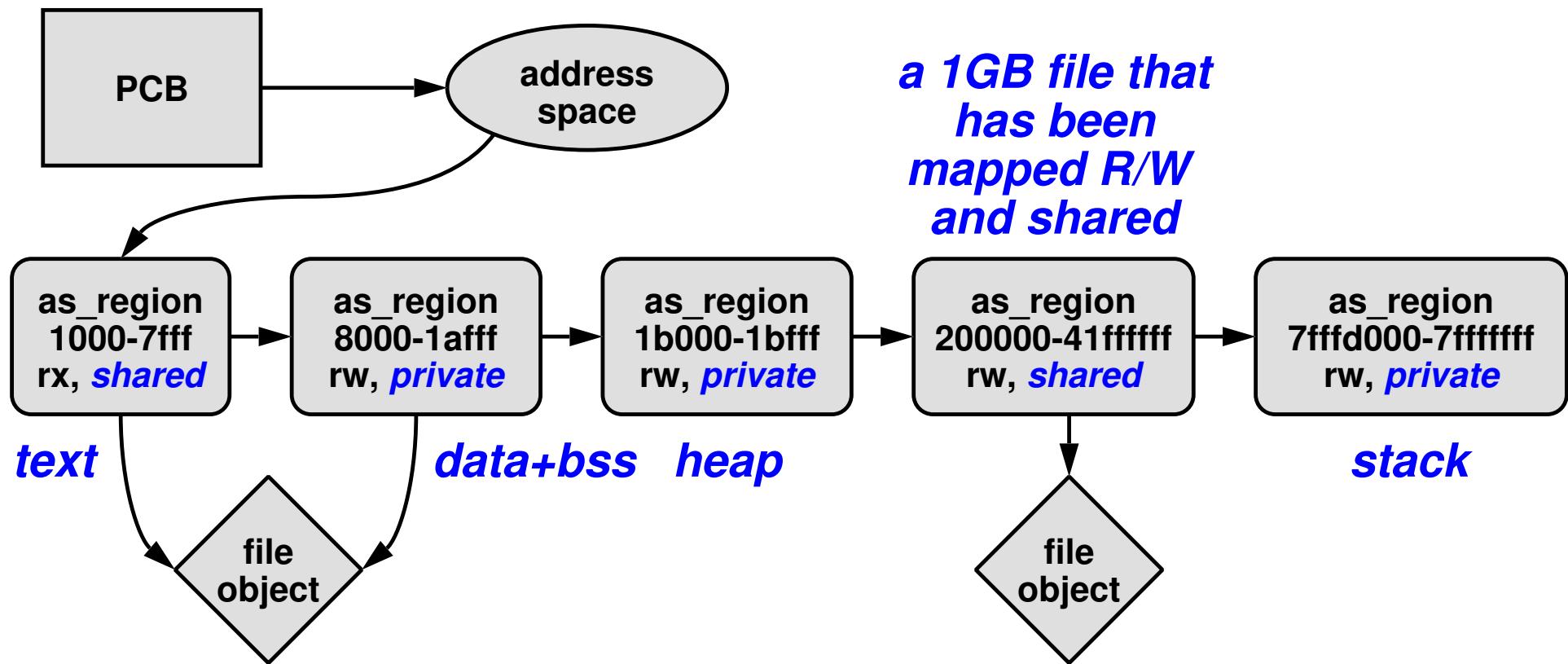
# User Address Space Representation



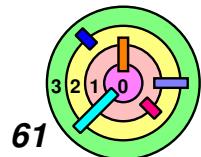
- **as\_region** (address space region data structure) contains:
  - = **start address**, **length**, **access permissions**, **shared** or **private**
  - = if mapped to a file, pointer to the corresponding **file object**
- This is related to Kernel Assignment 3 where you need to create and manage **address spaces** / **virtual memory maps**



# User Address Space Representation

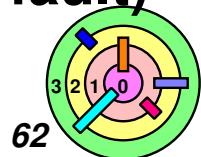


- In this example, text and data map portions of the same file
- = **text** is marked read-execute and **shared**
  - = **data+bss** is marked read-write and **private** to mean that changes will be private, i.e., will not affect other processes exec'ed from the same file



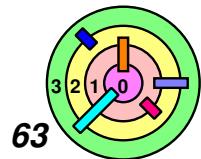
# How OS Makes Virtual Memory Work?

- If a thread access a virtual memory location that's both in primary storage and mapped by the hardware's map
  - no action by the OS
- If a thread access a virtual memory location that's *not in primary storage* or if the *translation is not in the hardware's map*
  - a **page fault** is occurred and the OS is invoked
    - OS checks the `as_region` address space data structures to make sure the reference is valid
      - ◊ if it's valid, the OS does whatever that's necessary to locate or create the object of the reference
      - ◊ find, or if necessary, make room for it in primary storage if it's not already there, and put it there
      - ◊ fix up all the kernel data structures then return from page fault so that application can *retry the memory access*
    - if invalid, it turns into a **segmentation fault** (or bad page fault)



# Storage Management

- Two issues need further discussion
- how is the *primary storage* managed (in terms of "resource management")?
  - how are the objects managed in *secondary storage*?



# How Is The Primary Storage "Resource" Managed?

→ Who needs primary storage?

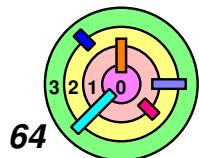
- application processes
- OS (e.g., terminal-handling subsystem, communication subsystem, I/O subsystem, etc.)
- they **compete** for primary storage

→ If primary storage is managed poorly

- one subsystem can use up all the available memory
  - then other subsystem won't get to run
  - this can even lead to OS crash when a subsystem uses up all of physical memory

→ If there are no mapped files, the solution can be simple

- assign each process a fixed amount of primary storage
  - this way, they won't compete
  - but is it fair?



# In Reality, Have To Deal With Mapped Files



An example to demonstrate a dilemma

- one process is using all of its primary storage allocation
- it then maps a file into its address space and starts accessing that file
- should the memory that's needed to buffer this file be charged against the files subsystem or charged against the process?



If charged against the files subsystem

- if the newly mapped file takes up all the buffer space in the files subsystem, it's unfair to other processes



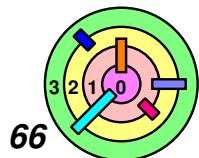
If charged against the process

- if other processes are sharing the same file, other processes are getting a free ride (in terms of memory usage)
- even worse, another process may increase the memory usage of this process (double unfair!)



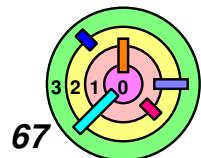
# In Reality, Have To Deal With Mapped Files

- ➡ It's difficult to be *fair*
  - it's difficult to even define what *fair* means
- ➡ We will discuss some solutions in Ch 7
  - for now, we use the following solution
    - give each participant (processes, file subsystem, etc.) a minimum amount of storage
    - leave some additional storage available for all to compete



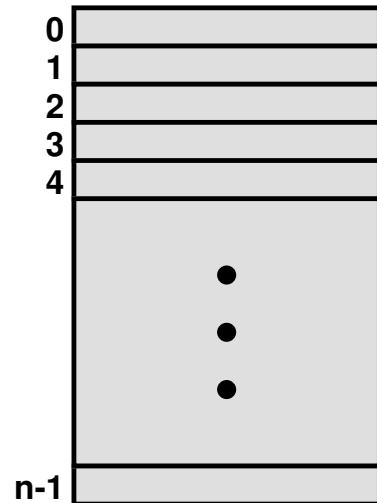
# How Are Objects Managed In Secondary Storage?

- ➡ The **file system** is used to manage objects in secondary storage
  - the term "file system" can mean two different things
    - 1) how to **layout data** on secondary storage (data structures on disk)
    - 2) how to access those data in data structures (code)
- ➡ The file system is usually divided into two parts
  - **file system independent**
    - supports the "file abstraction"
    - on Unix, this is called the "**virtual file system (VFS)**"
      - ◆ Kernel Assignment 2
    - on Windows, this is called the "**I/O manager**"
  - **file system dependent**
    - on Unix, this is called the "**actual file system (AFS)**"
      - ◆ e.g., ext2, ext3, ext4, etc.
    - on Windows, this is called the "**file system**"
      - ◆ e.g., FAT32, NTFS, etc.

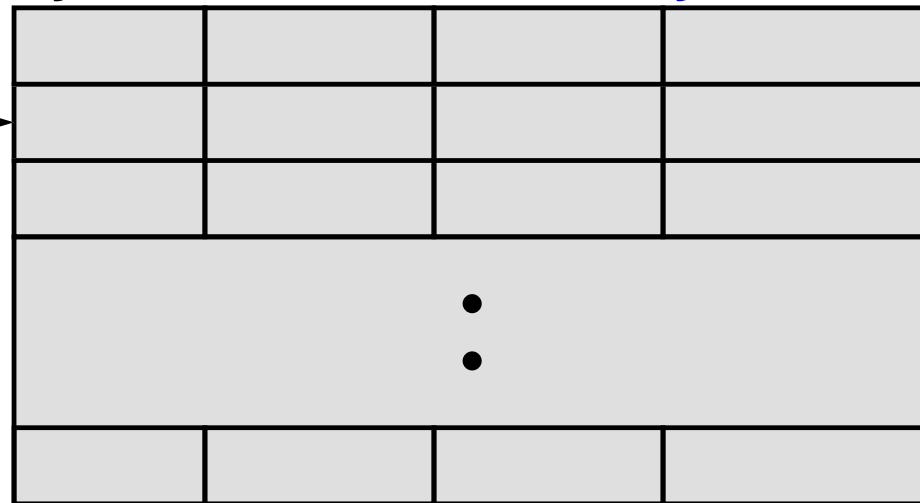


# Open-File Data Structures

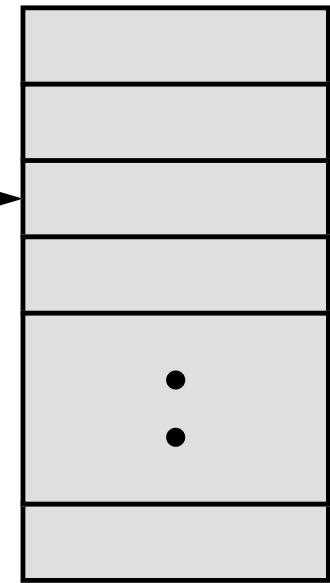
(per process)  
file-descriptor  
table



(system-wide)  
system file table / *file object table*



*FS-dependent*  
inode table



ref count    access    file position    *inode pointer*  
(typo in textbook)

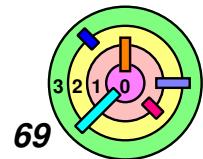
- ▶ In the kernel, *each process* has its own *file-descriptor table*
  - the kernel also maintains *system file table* (or *file object table*)
- ▶ The *file object* / *inode* forms the *boundary* between *VFS* and the *AFS* (i.e., points to *file-system-dependent* stuff)
  - how can this be done?

# File Object

- The file object is like an *abstract class* in C++
  - subclasses of file object are the *actual* file objects

```
class FileObject {  
    unsigned short refcount, access;  
    unsigned int file_pos;  
    ...  
    virtual int create(const char *, int, FileObject **);  
    virtual int read(int, void *, int);  
    virtual int write(int, const void *, int);  
    ...  
};
```

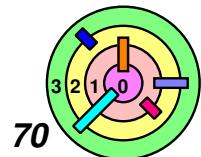
- But wait ...
  - what's this about C++?
    - real operating systems are written in C ...
    - checkout the **DRIVERS** kernel documentation (we skipped this **weenix** assignment)
      - ◆ similar trick (**polymorphism**) used in VFS



# File Object in C

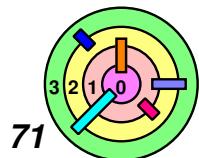
```
typedef struct {
    unsigned short refcount, access;
    unsigned int file_pos;
    ...
    void **file_ops; /* to array of function pointers */
} FileObject;
```

- A file object uses an *array of function pointers*
  - this is how C implements *C++ polymorphism*
  - one function pointer for each operation on a file
  - where they point to is (actual) file system dependent
  - but the (virtual) interface is the same to higher level of the OS
- Loose coupling between the actual file system and storage devices
  - the actual file system is written to talk to the devices also in a *device-independent* manner
    - i.e., using major and **minor device numbers** to reference the device and using standard interface provided by the device driver



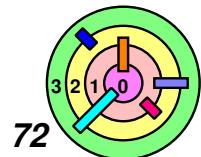
# File System Cache

- ➔ Storage **devices** are slow
  - disks are particularly slow
  - use a lot of tricks to make them look and feel fast
- ➔ **Recently used blocks** in a file are kept in a **file system cache**
  - the primary storage holding these blocks might be mapped into one or more address spaces of processes that have this file mapped
    - blocks are available for immediate access by read and write system calls
- ➔ Fancier data structures in storage system and file system
  - e.g., **hash table** can be used to locate file blocks in the cache
    - maybe keyed by **inode number**
- ➔ More details in Ch 6



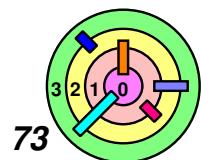
# 1.3 A Simple OS

- ▶ OS Structure
- ▶ Processes, Address Spaces, & Threads
- ▶ Managing Processes
- ▶ Loading Program Into Processes
- ▶ *Files*
  - = (will skip first 13 slides since they overlap with "A Simple OS" slides)



# Files

- Our primes program wasn't too interesting
  - it has no output!
  - cannot even verify that it's doing the right thing
  - other program cannot use its result
  - how does a process write to someplace outside the process?
- The notion of a **file** is our Unix system's sole abstraction for this concept of "someplace outside the process"
  - modern Unix systems have additional abstractions
- Files
  - abstraction of persistent data storage
  - means for fetching and storing data outside a process
    - including disks, another process, keyboard, display, etc.
    - need to **name** these different places
      - ◊ hierarchical naming structure
    - part of a process's **extended address space**



# Naming Files

## → Directory system

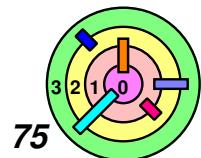
- shared by all processes running on a computer
  - although each process can have a different view
  - Unix provides a means to restrict a process to a subtree
    - ◆ by redefining what "root" means for the process
- name space is outside the processes
  - a user process provides the name of a file to the OS
  - the OS returns a **handle** to be used to access the file
    - ◆ after it has verified that the process is allowed **access** along the **entire path**, starting from root
  - user process uses the handle to read/write the file
    - ◆ avoid access checks

## → Using a handle to refer to an object managed by the kernel is an important concept

- handles are essentially an **extension** to the process's **address space**
  - can even survive execs!

# The File Abstraction

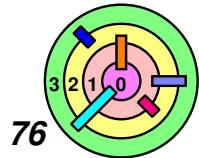
- A file is a simple array of bytes
- Files are made larger by writing beyond their current end
- Files are named by paths in a naming tree
- System calls on files are *synchronous*
- File API
  - `open()`, `read()`, `write()`, `close()`
  - e.g., `cat`



# File Handles (File Descriptors)

```
int fd;
char buffer[1024];
int count;
if ((fd = open("/home/bc/file", O_RDWR) == -1) {
    // the file couldn't be opened
    perror("/home/bc/file");
    exit(1);
}
if ((count = read(fd, buffer, 1024)) == -1) {
    // the read failed
    perror("read");
    exit(1);
}
// buffer now contains count bytes read from the file
```

- what is O\_RDWR?
- what does perror() do?
- *cursor* position in an opened file depends on what functions/system calls you use
  - what about C++?



# Standard File Descriptors

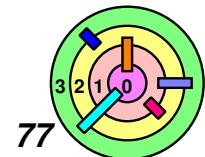


## Standard File Descriptors

- 0 is stdin (by default, the keyboard)
- 1 is stdout (by default, the display)
- 2 is stderr (by default, the display)

```
main() {
    char buf[BUFSIZE];
    int n;
    const char *note = "Write failed\n";

    while ((n = read(0, buf, sizeof(buf))) > 0)
        if (write(1, buf, n) != n) {
            (void)write(2, note, strlen(note));
            exit(EXIT_FAILURE);
        }
    return(EXIT_SUCCESS);
}
```

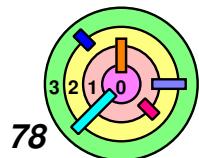


# Back to Primes

- Have our primes program write out the solution, i.e., the `primes[]` array

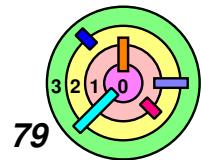
```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    ...
    for (i=1; i<nprimes; i++) {
        ...
    }
    if (write(1, prime, nprimes*sizeof(int)) == -1) {
        perror("primes output");
        exit(1);
    }
    return(0);
}
```

- the output is not readable by human



# Human-Readable Output

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    ...
    for (i=1; i<nprimes; i++) {
        ...
    }
    for (i=0; i<nprimes; i++) {
        printf("%d\n", prime[i]);
    }
    return(0);
}
```

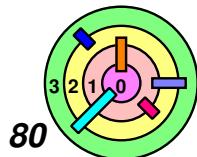


# Allocation of File Descriptors

- Whenever a process requests a new file descriptor, the lowest numbered file descriptor not already associated with an open file is selected; thus

```
#include <fcntl.h>
#include <unistd.h>
...
close(0);
fd = open("file", O_RDONLY);
```

- will always associate "file" with file descriptor 0 (assuming that the open succeeds)



# Running It

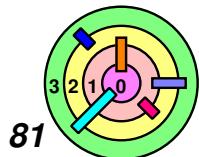
```

if (fork() == 0) {
    /* set up file descriptor 1 in the child process */
    close(1);
    if (open("/home/bc/Output", O_WRONLY) == -1) {
        perror("/home/bc/Output");
        exit(1);
    }
    execl("/home/bc/bin/primes", "primes", "300", 0);
    exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
;

```

- `close(1)` removes file descriptor 1 from ***extended address space***
- file descriptors are allocated ***lowest first*** on `open()`
- ***extended address space*** survives `execs`
- new code is same as running

% `primes 300 > /home/bc/Output`



# I/O Redirection

```
% primes 300 > /home/bc/Output
```

- The ">" parameter in a shell command that instructs the command shell to *redirect* the output to the given file
  - If ">" weren't there, the output would go to the display

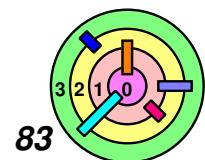
- Can also redirect input

- ```
% cat < /home/bc/Output
```

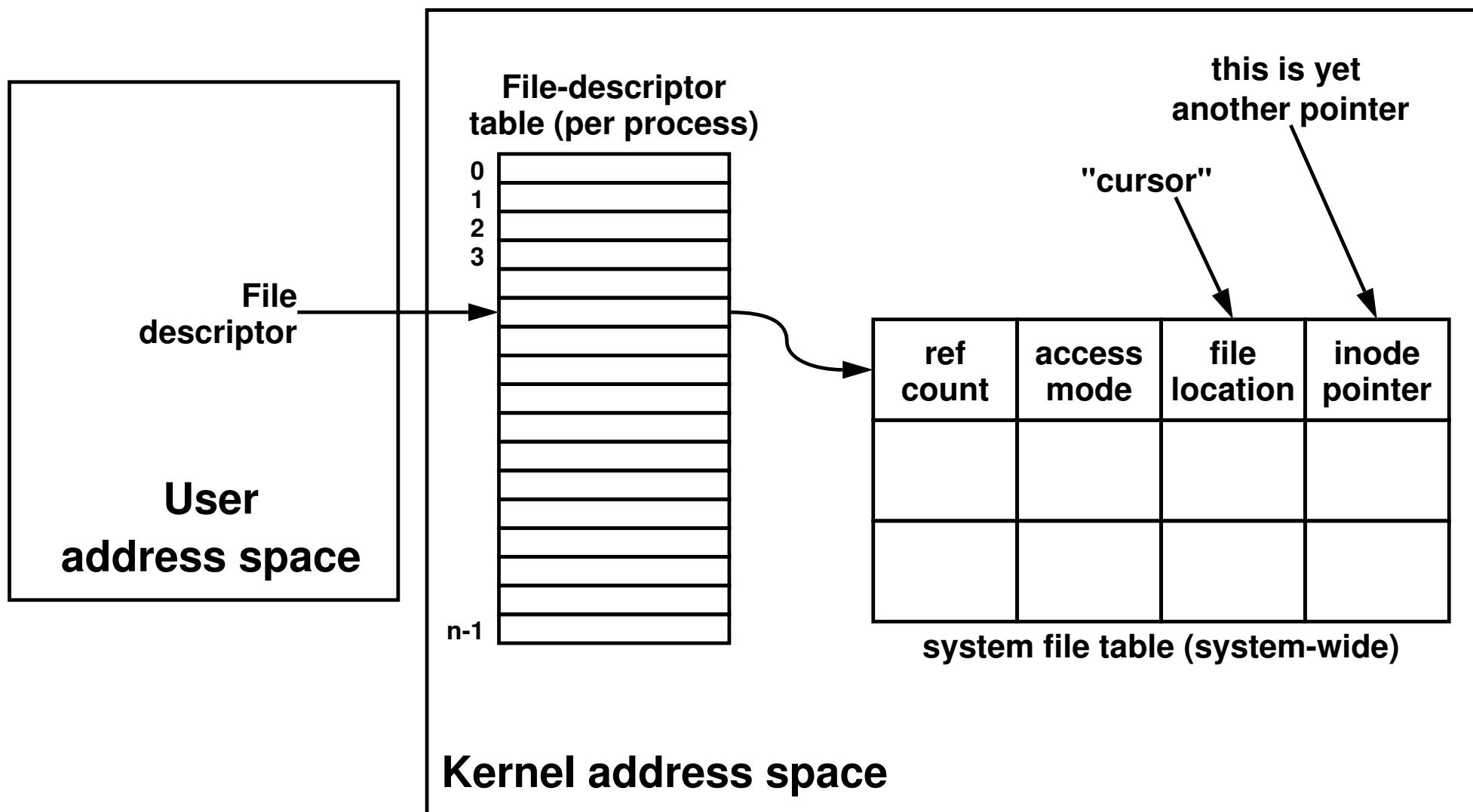
    - when the "cat" program reads from file descriptor 0, it would get the data bytes from the file "/home/bc/Output"

# File-Descriptor Table

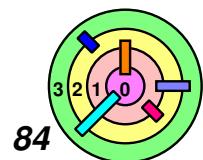
- ➡ A file descriptor refers not just to a file
  - it also refers to the *process's* current *context* for that file
    - includes how the file is to be accessed (how `open()` was invoked)
    - *cursor* position
- ➡ *Context* information must be maintained by the OS and not directly by the user program
  - let's say a user program opened a file with `O_RDONLY`
  - later on it calls `write()` using the opened file descriptor
  - how does the OS know that it doesn't have write access?
    - stores `O_RDONLY` in context
  - if the user program can manipulate the context, it can change `O_RDONLY` to `O_RDWR`
  - therefore, user program must not have access to context!
    - all it can see is the handle
    - the handle is an *index* into an array maintained for the process in kernel's address space



# File-Descriptor Table

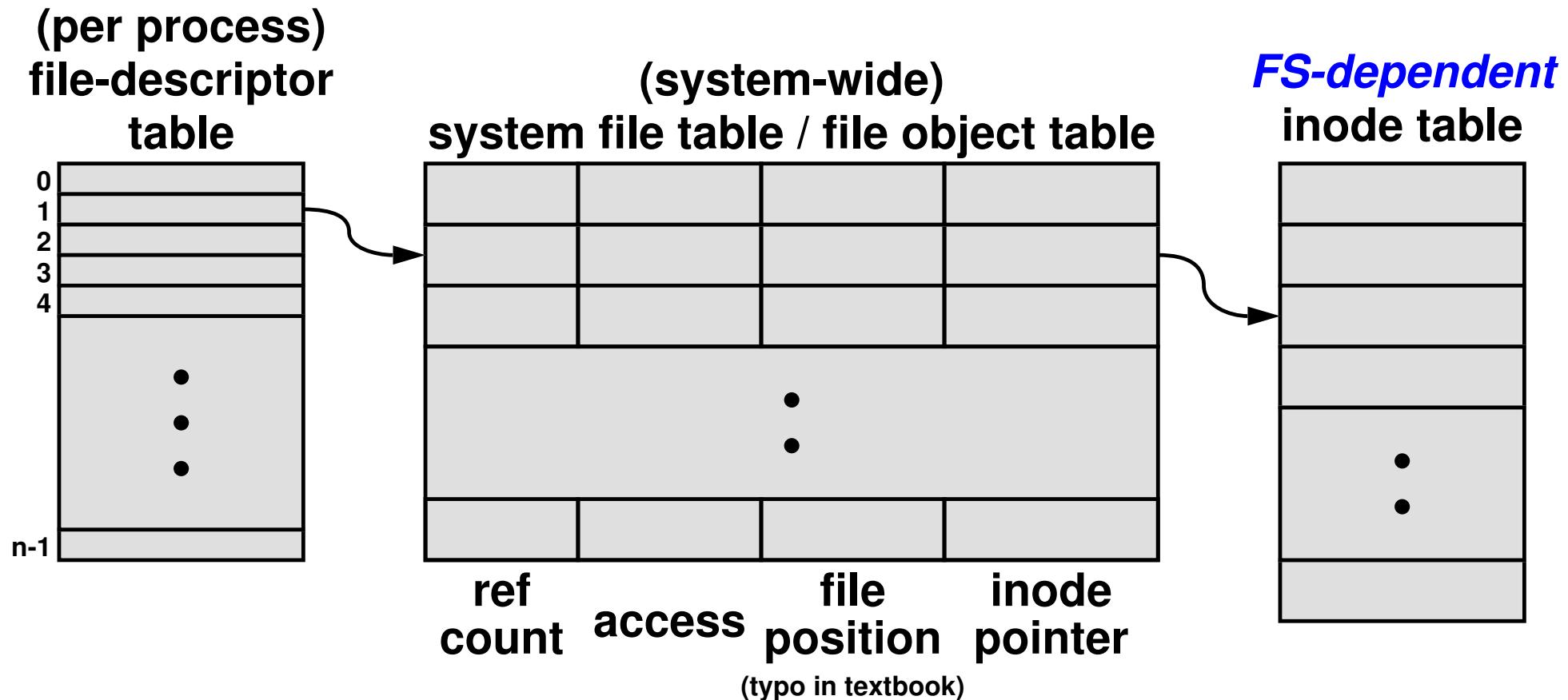


- context is not stored directly into the file-descriptor table
  - one-level of *indirection*



# VFS vs. AFS

→ We now know more about the "inode pointer"...



→ We will focus on VFS in Ch 1  
— AFS will be covered in Ch 6

# Put It All Together

open()

```
int fd;  
→ fd = open("foo.txt");  
char buf[512];  
read(fd, buf, 100);  
close(fd);
```

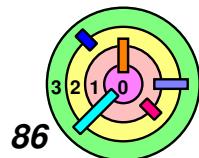
Applications

OS

Process  
Subsystem

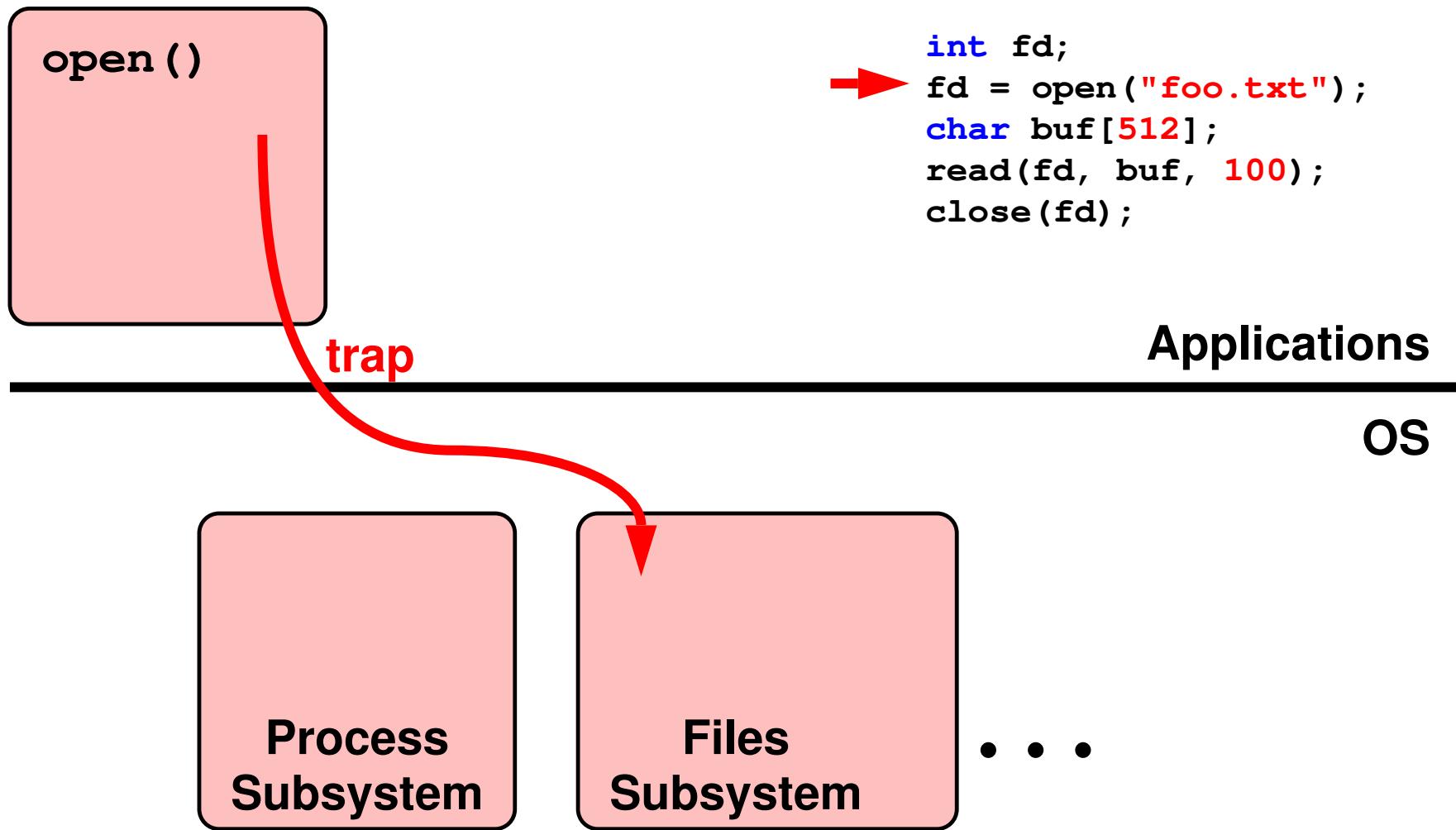
Files  
Subsystem

• • •

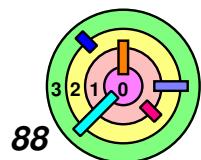
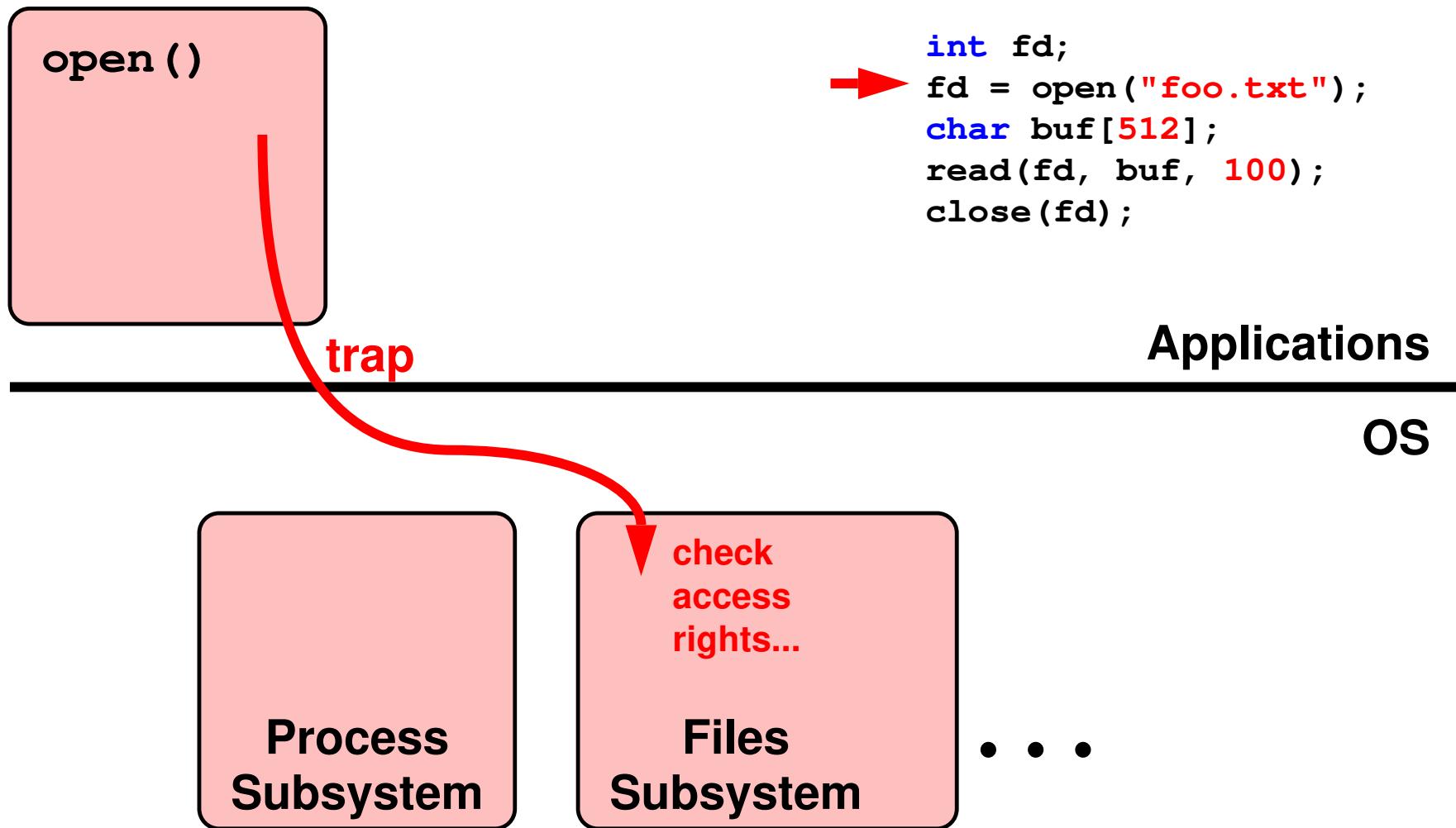


86

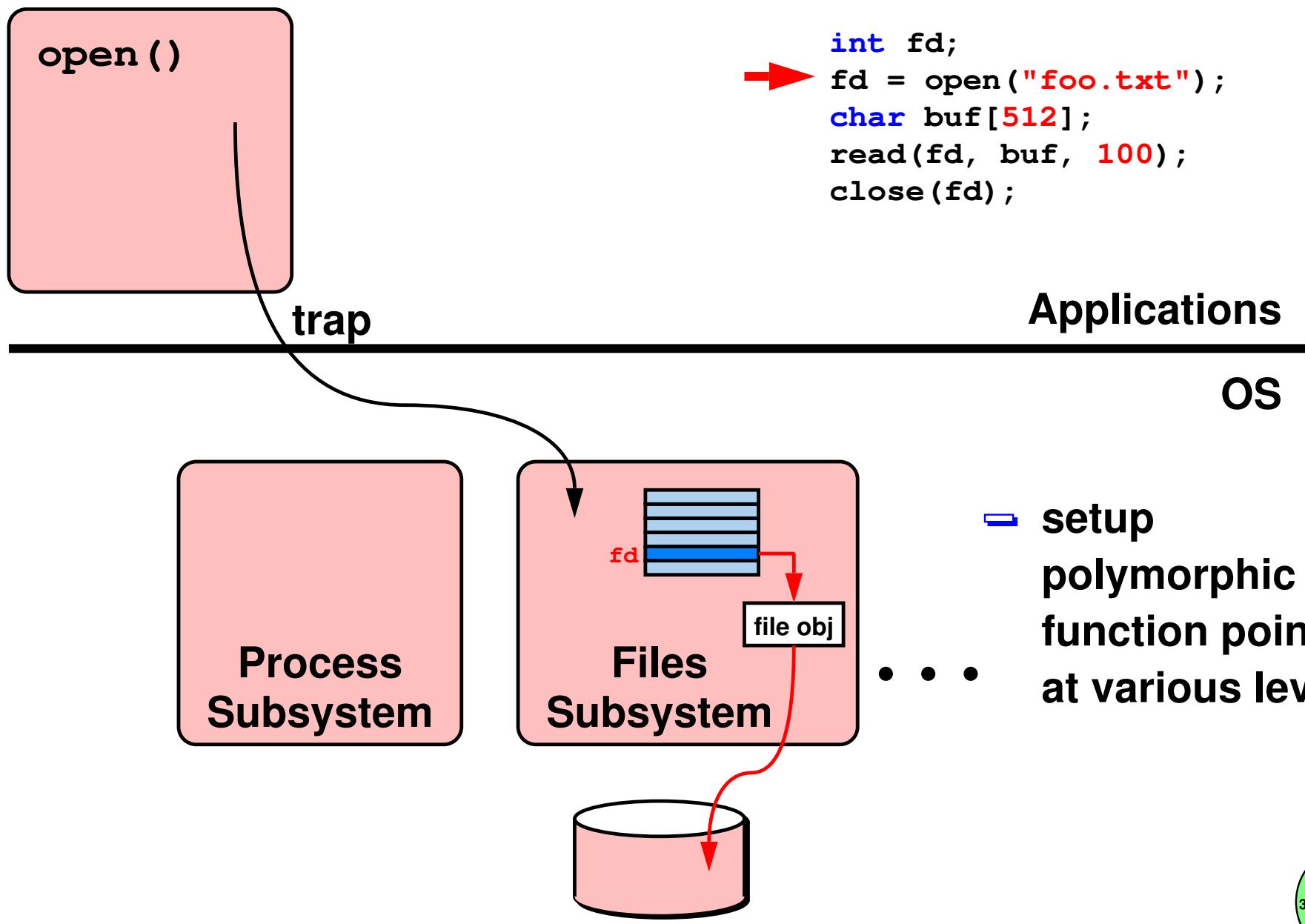
# Put It All Together



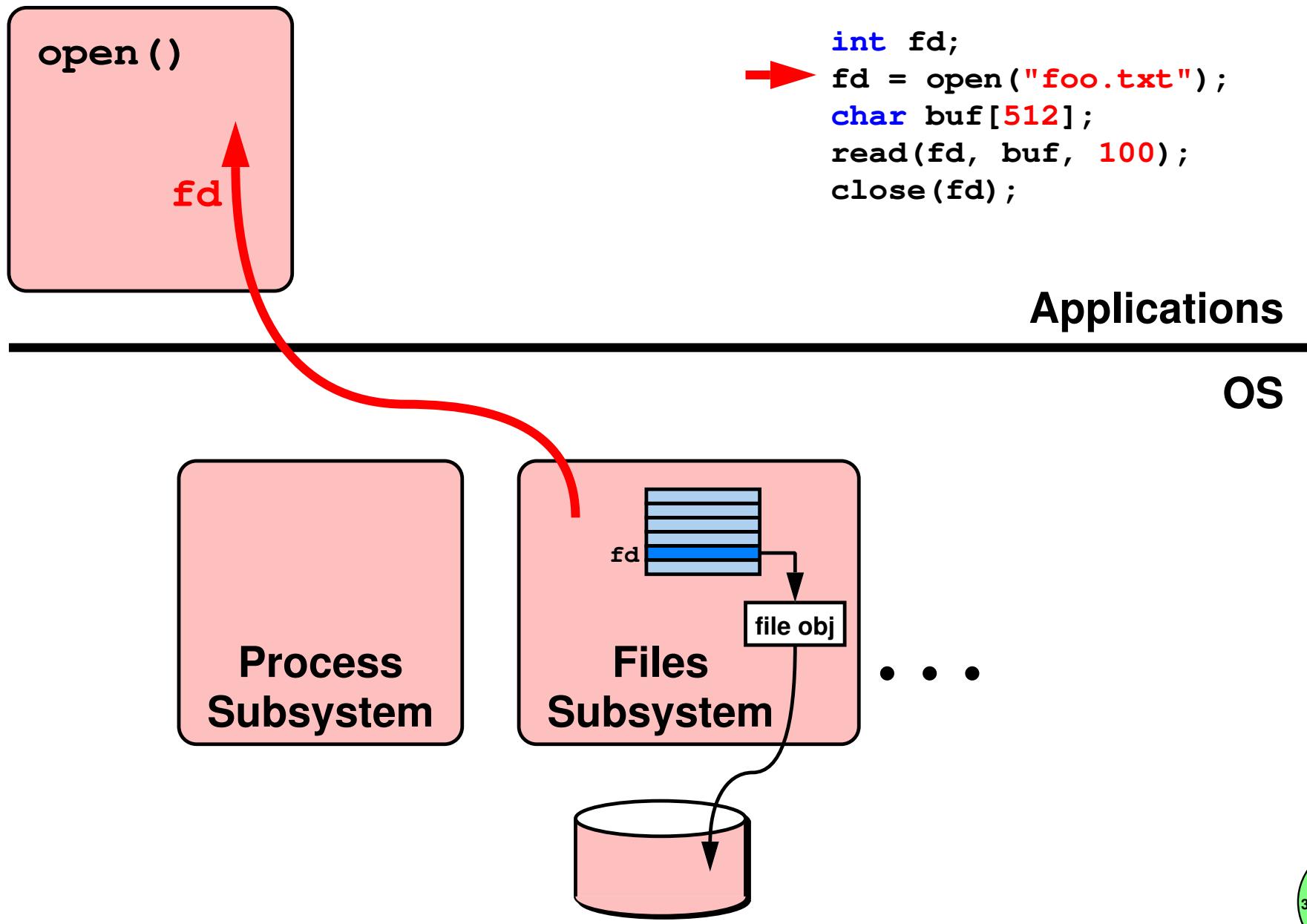
# Put It All Together



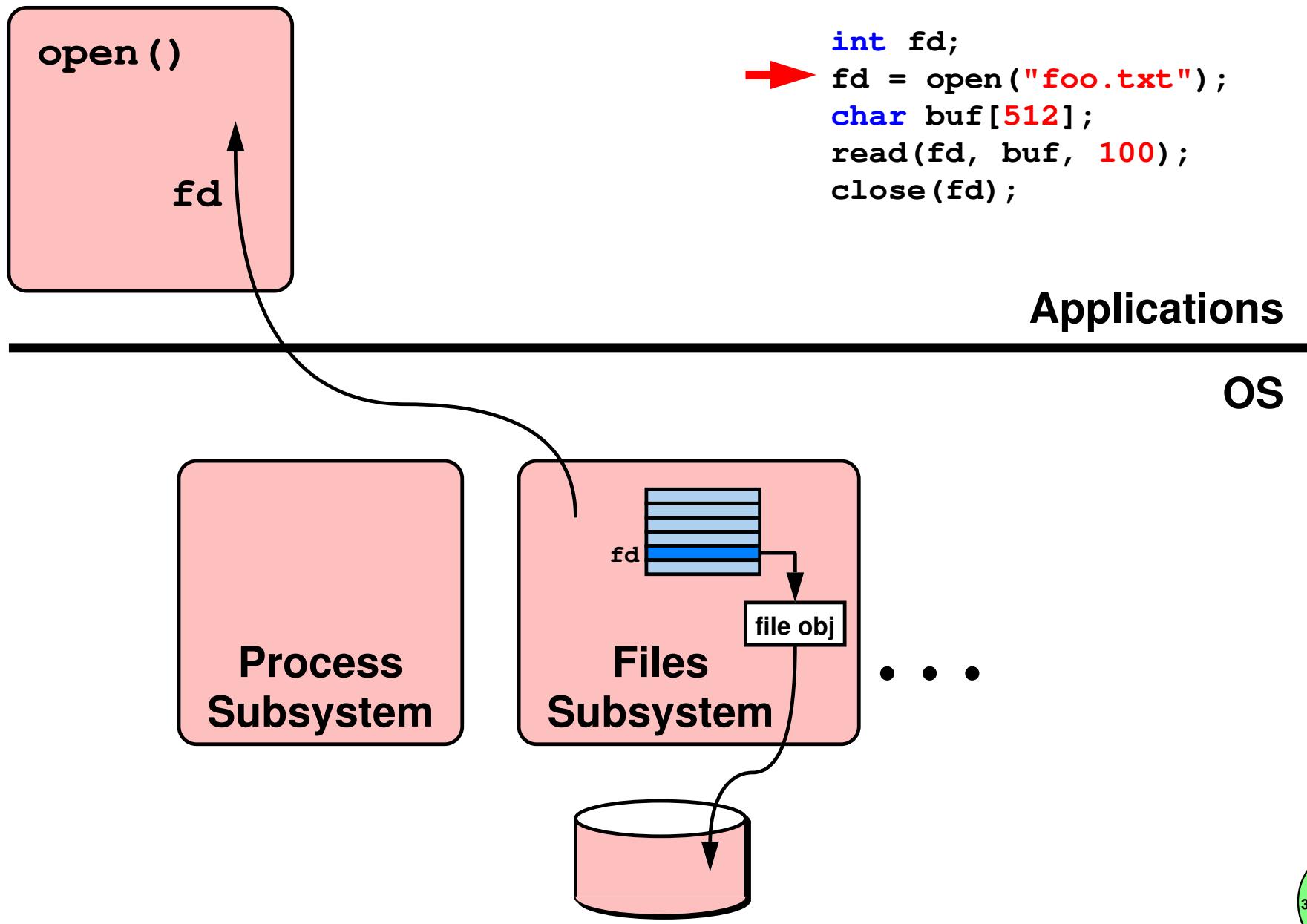
# Put It All Together



# Put It All Together



# Put It All Together



# Put It All Together

`open()`  
`read()`

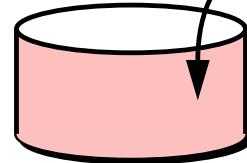
```
int fd;  
fd = open("foo.txt");  
char buf[512];  
→ read(fd, buf, 100);  
close(fd);
```

Applications

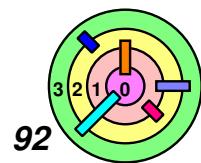
OS

Process  
Subsystem

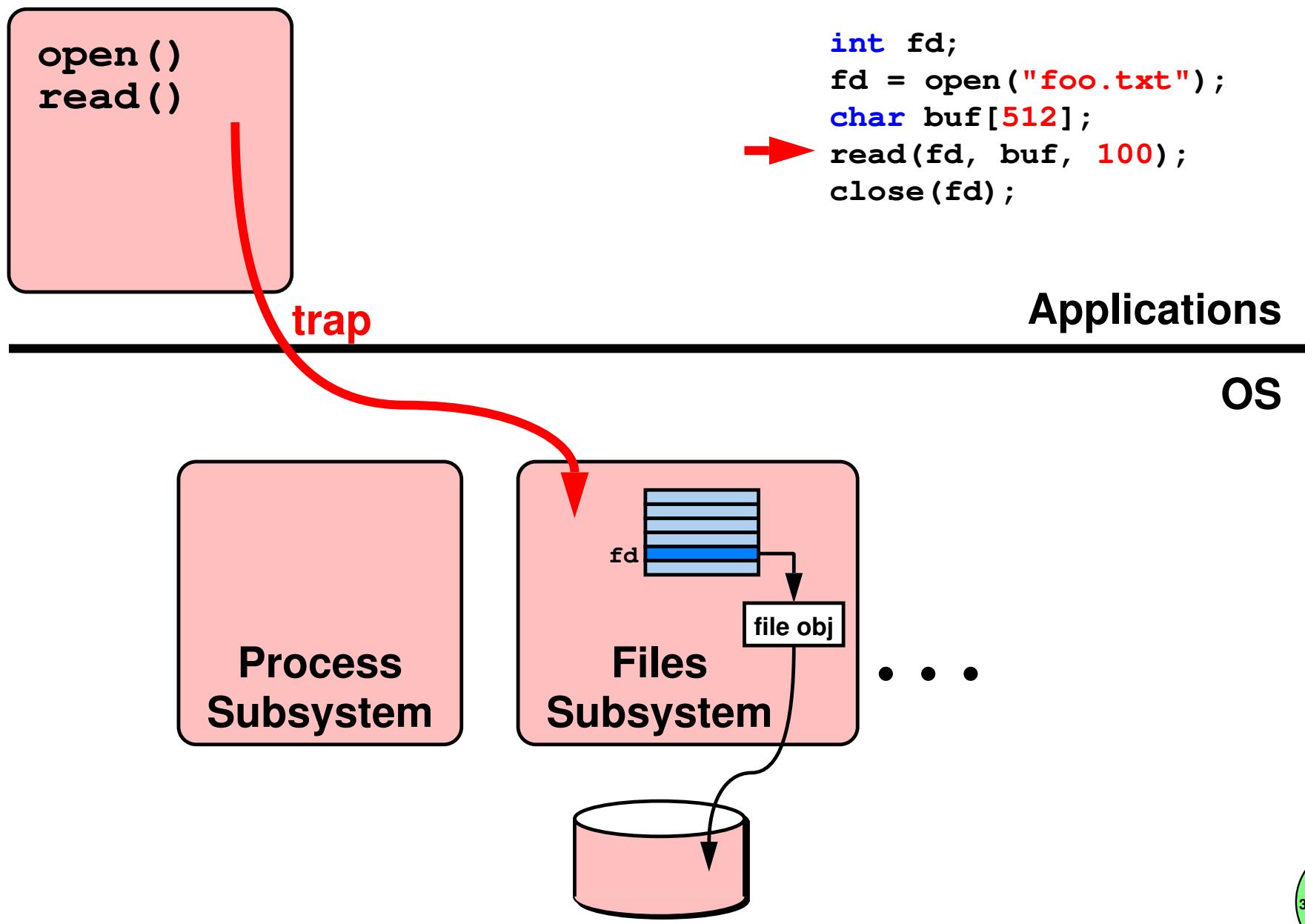
Files  
Subsystem



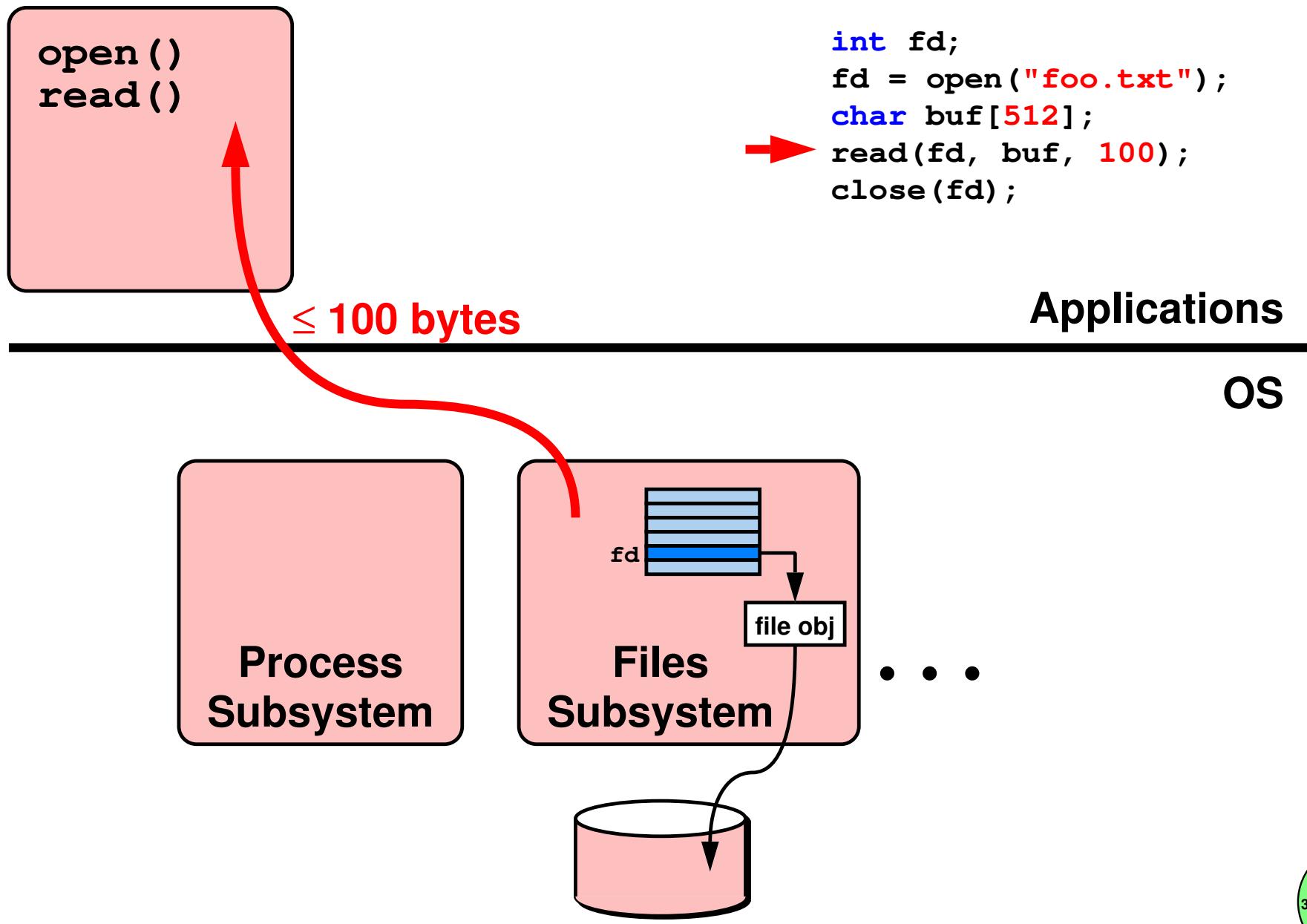
• • •



# Put It All Together



# Put It All Together



# Put It All Together

`open()`  
`read()`  
`close()`

```
int fd;  
fd = open("foo.txt");  
char buf[512];  
read(fd, buf, 100);  
close(fd);
```



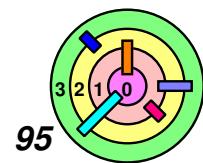
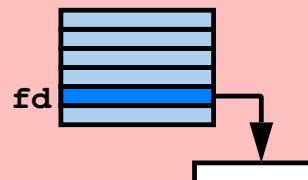
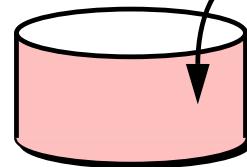
Applications

OS

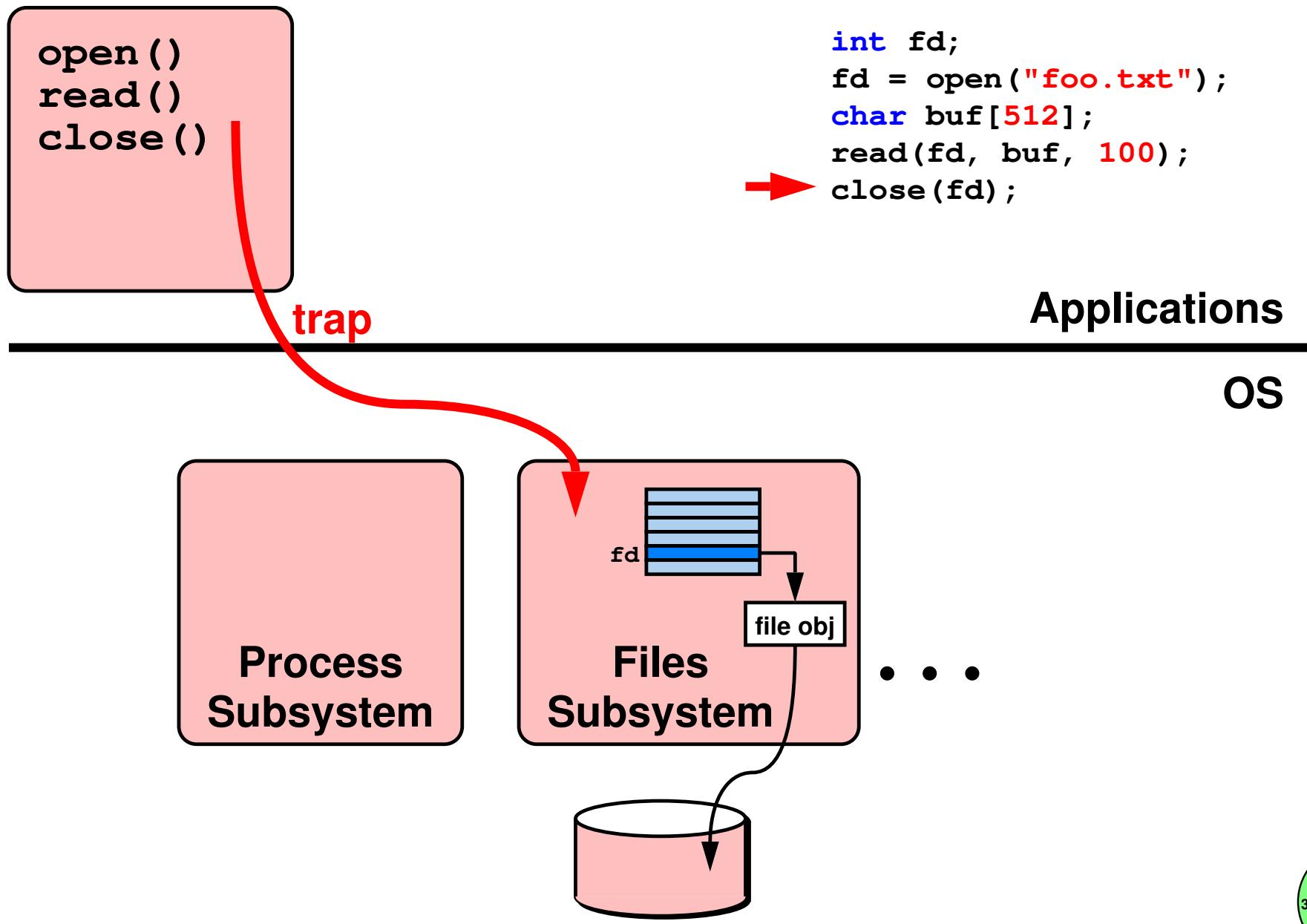
Process  
Subsystem

Files  
Subsystem

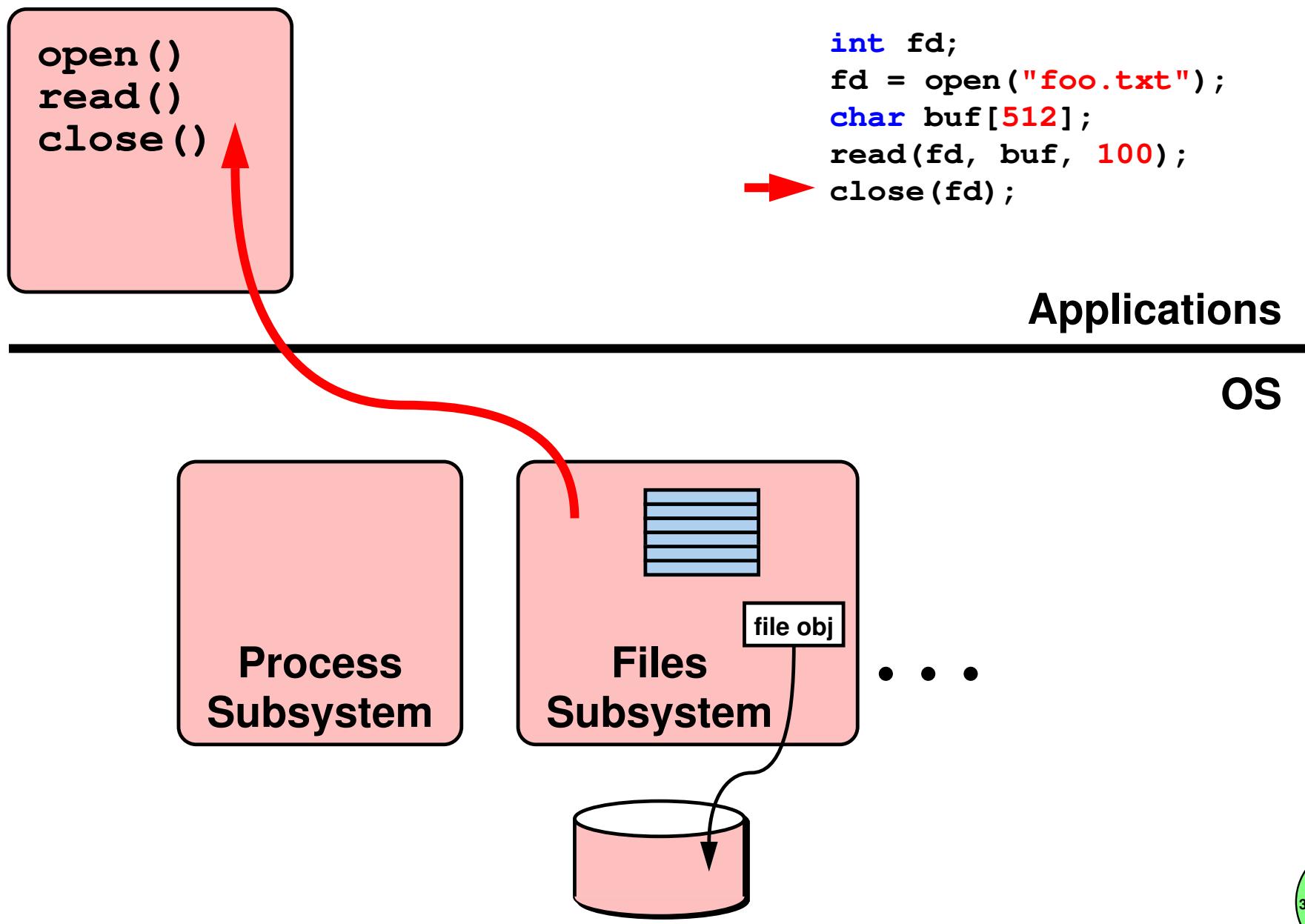
• • •



# Put It All Together



# Put It All Together



# Put It All Together

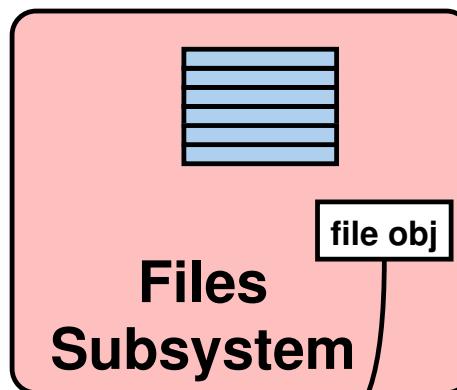
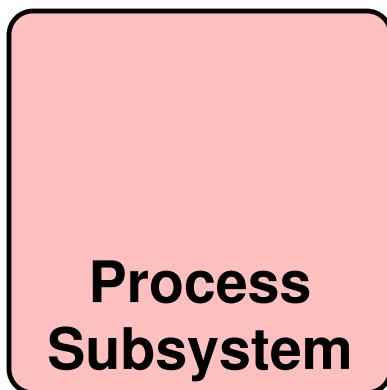
```
open()  
read()  
close()
```

```
int fd;  
fd = open("foo.txt");  
char buf[512];  
read(fd, buf, 100);  
close(fd);
```



Applications

OS



- file object not deallocated if ref count > 0

# Redirecting Output ... Twice

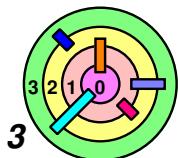
→ Every call to `open()` creates a new entry in the system file table

```

if (fork() == 0) {
    /* set up file descriptors 1 and 2 in the child
       process */
    close(1);
    close(2);
    if (open("/home/bc/Output", O_WRONLY) == -1) {
        exit(1);
    }
    if (open("/home/bc/Output", O_WRONLY) == -1) {
        exit(1);
    }
    execl("/home/bc/bin/program", "program", 0);
    exit(1);
}
/* parent continues here */

```

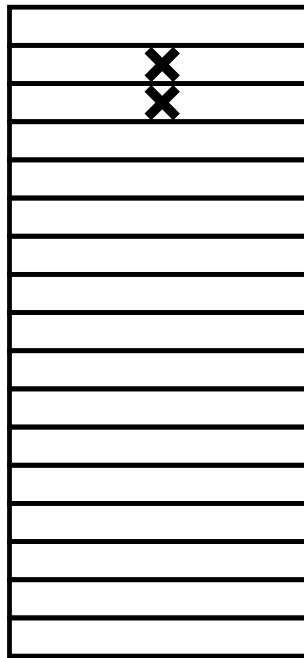
- `stdout` and `stderr` both go into the same file
  - would it cause any problem?



# Redirected Output

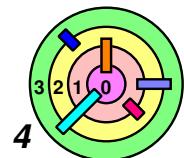
Child's  
address space

File-descriptor  
table (per process)

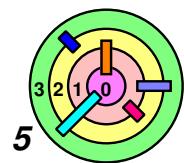
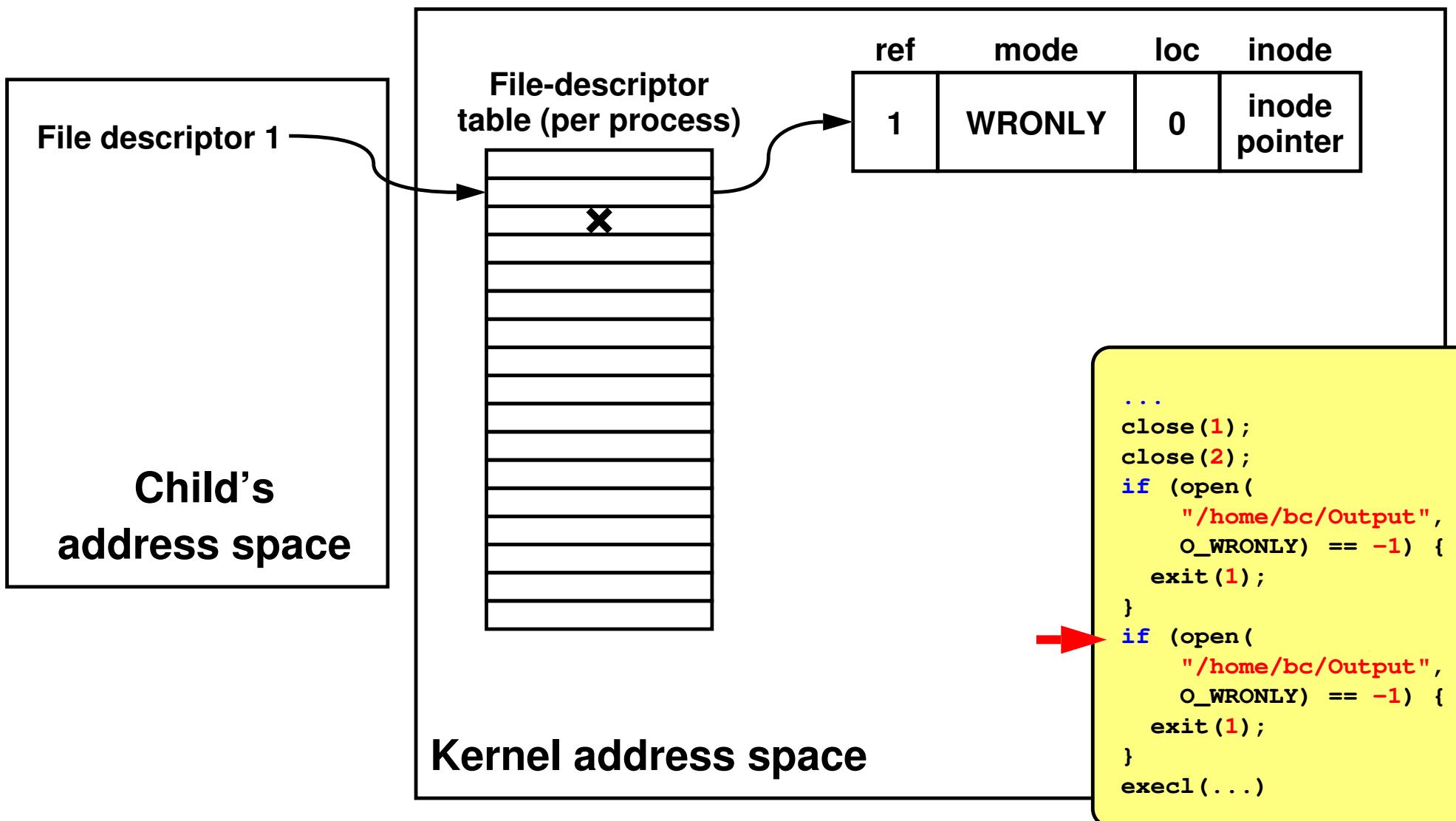


Kernel address space

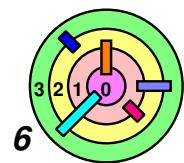
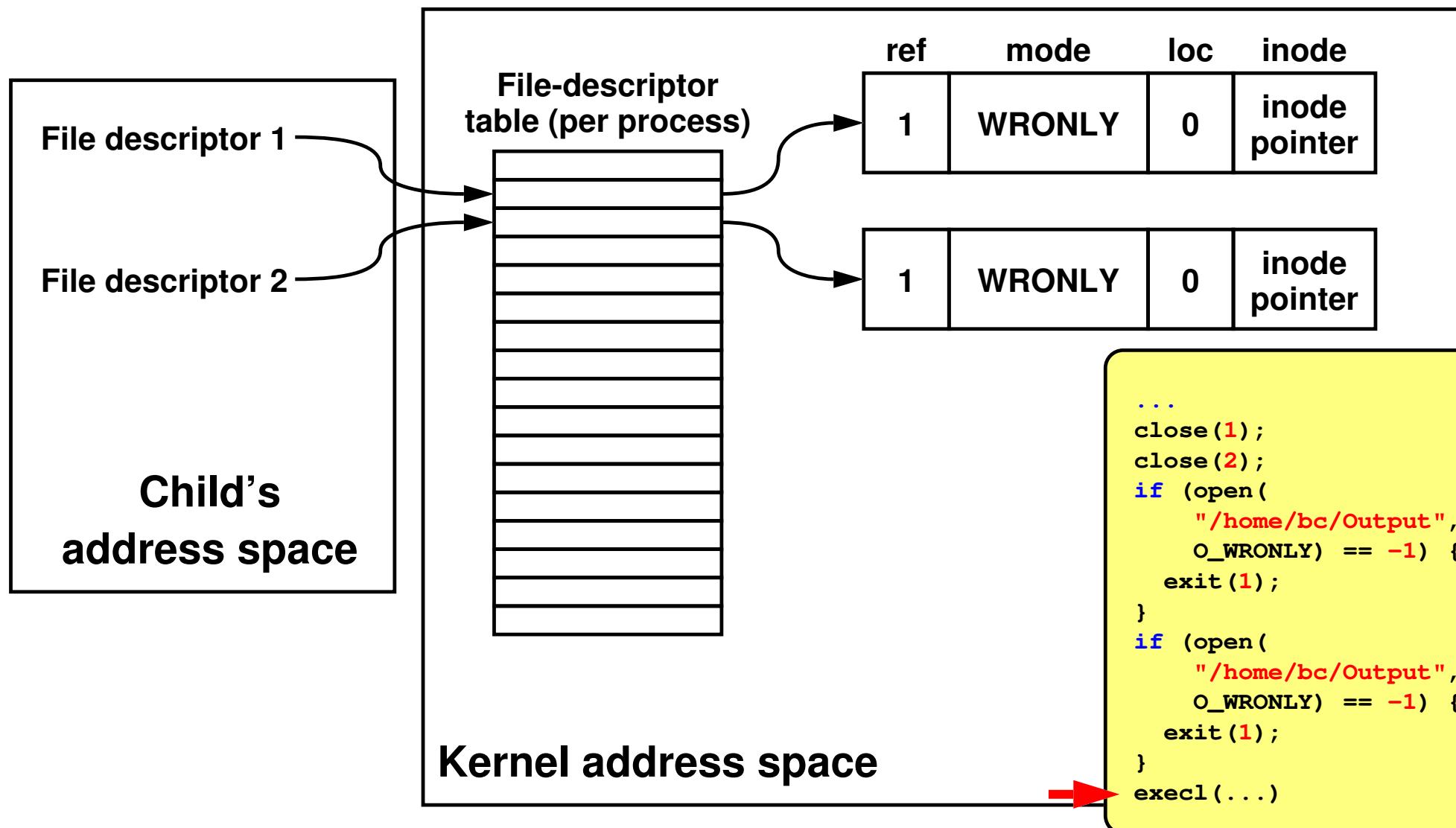
```
...
close(1);
close(2);
if (open(
    "/home/bc/Output",
    O_WRONLY) == -1) {
    exit(1);
}
if (open(
    "/home/bc/Output",
    O_WRONLY) == -1) {
    exit(1);
}
execl(...)
```



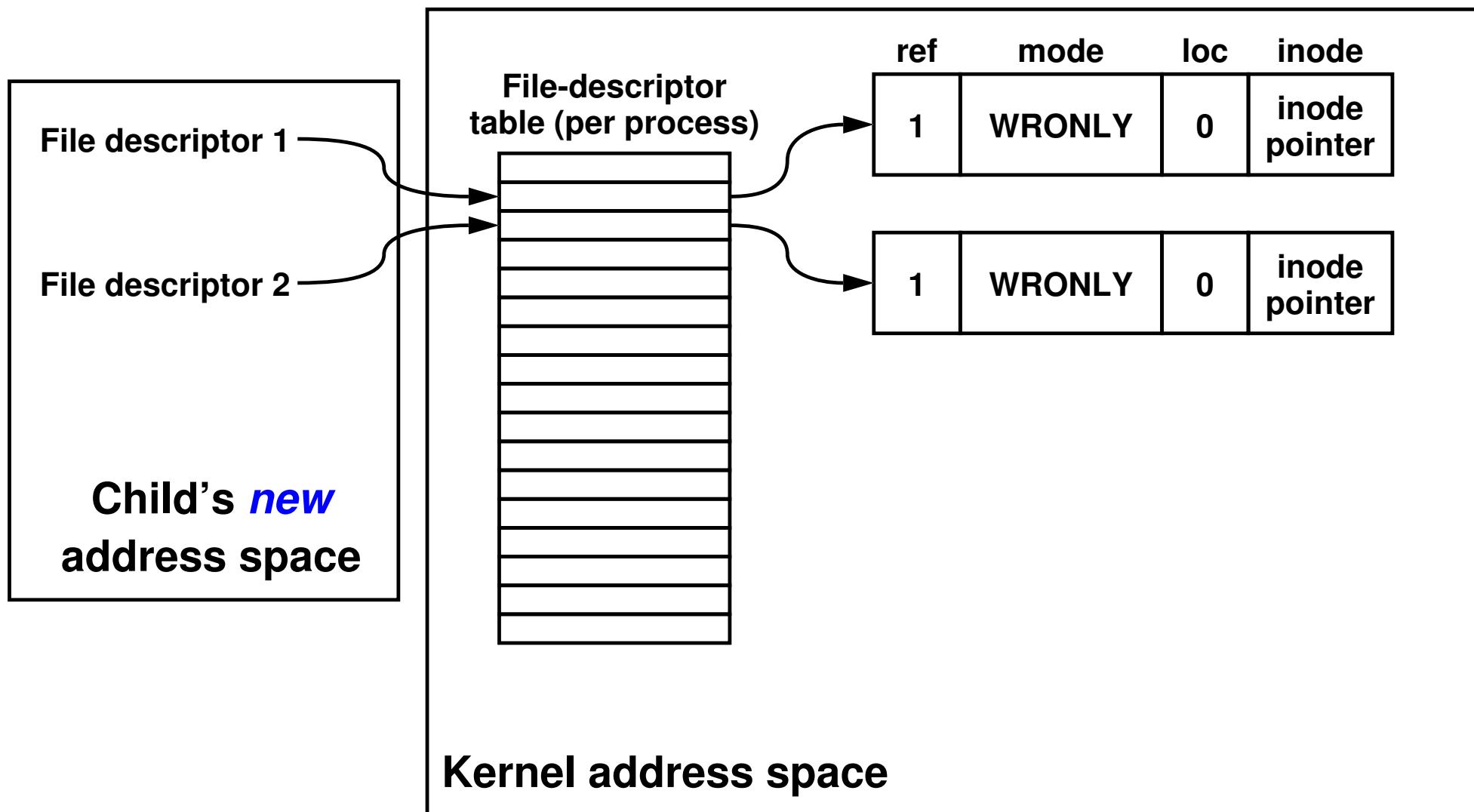
# Redirected Output



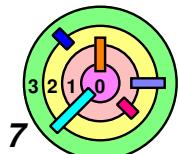
# Redirected Output



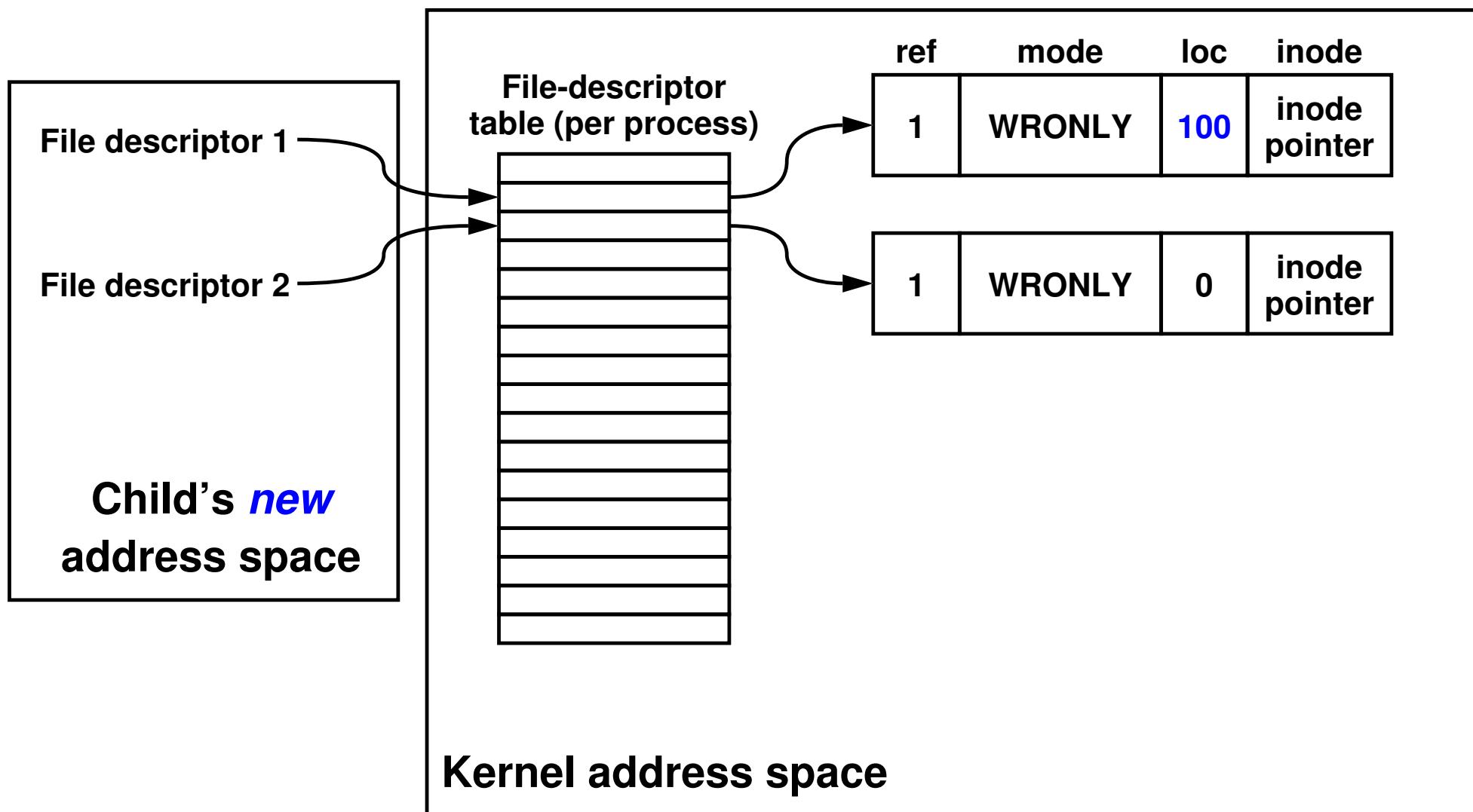
# Redirected Output



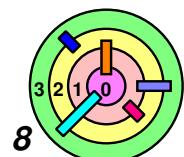
- remember, extended address space survives execs
- let's say we write 100 bytes to `stdout`



# Redirected Output After Writing 100 Bytes



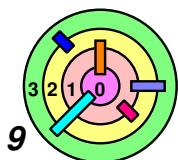
- `write()` to `fd=2` will wipe out data in the first 100 bytes
  - that may not be the intent



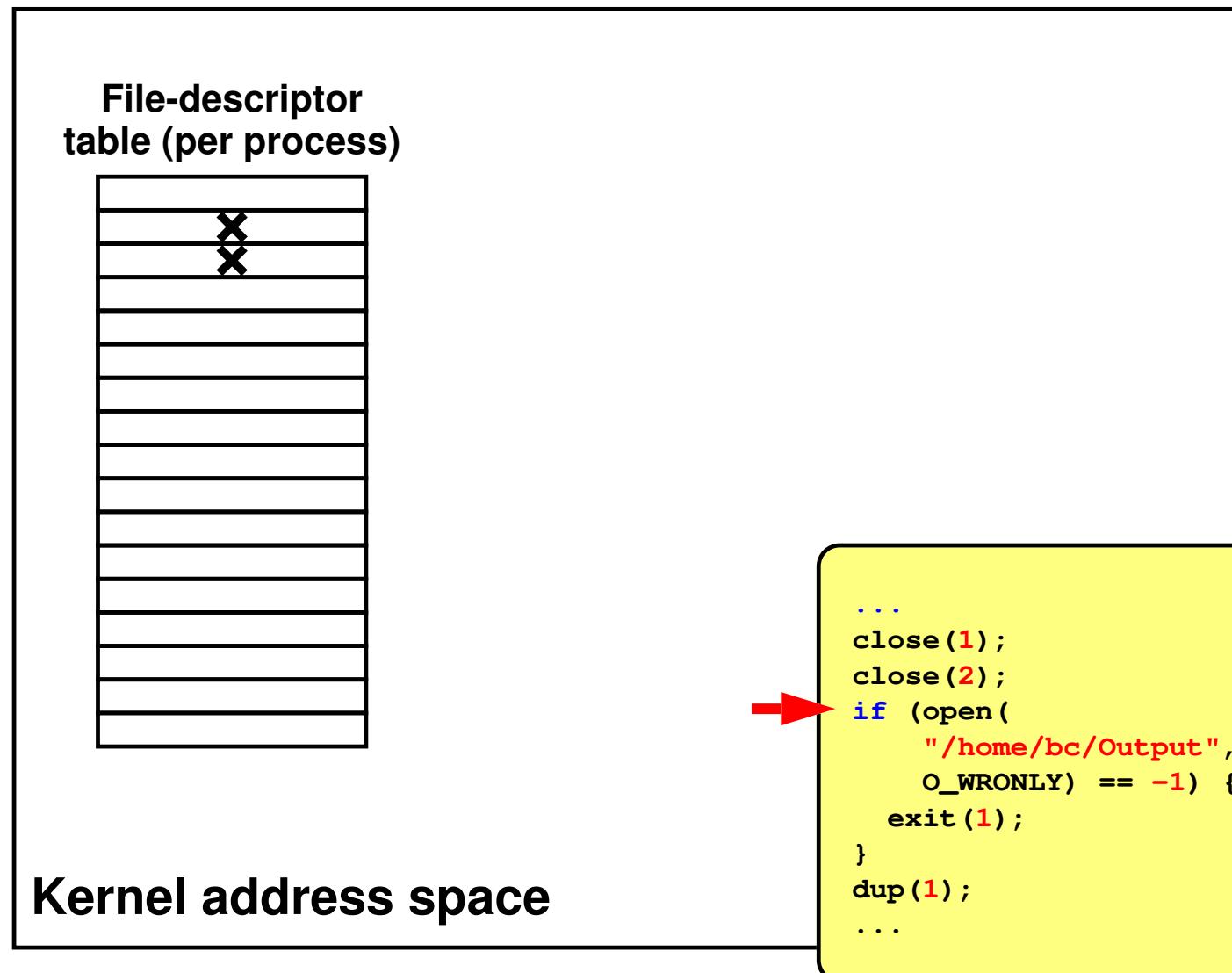
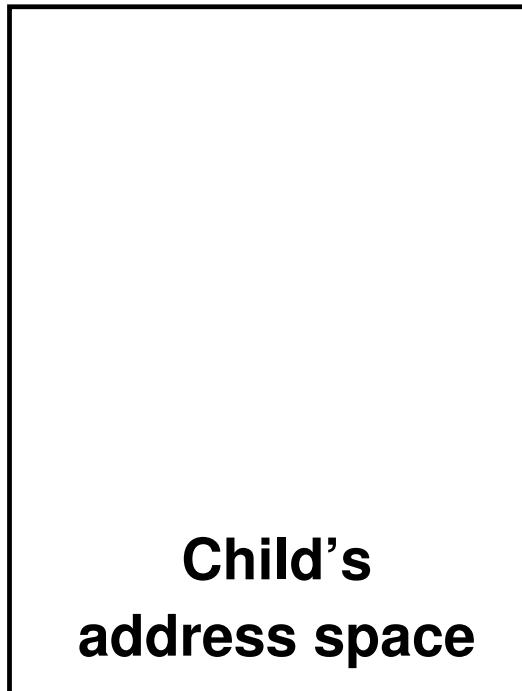
# Sharing Context Information

```
if (fork() == 0) {  
    /* set up file descriptors 1 and 2 in the child  
       process */  
    close(1);  
    close(2);  
    if (open("/home/bc/Output", O_WRONLY) == -1) {  
        exit(1);  
    }  
    dup(1);  
    execl("/home/bc/bin/program", "program", 0);  
    exit(1);  
}  
/* parent continues here */
```

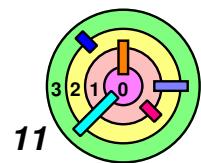
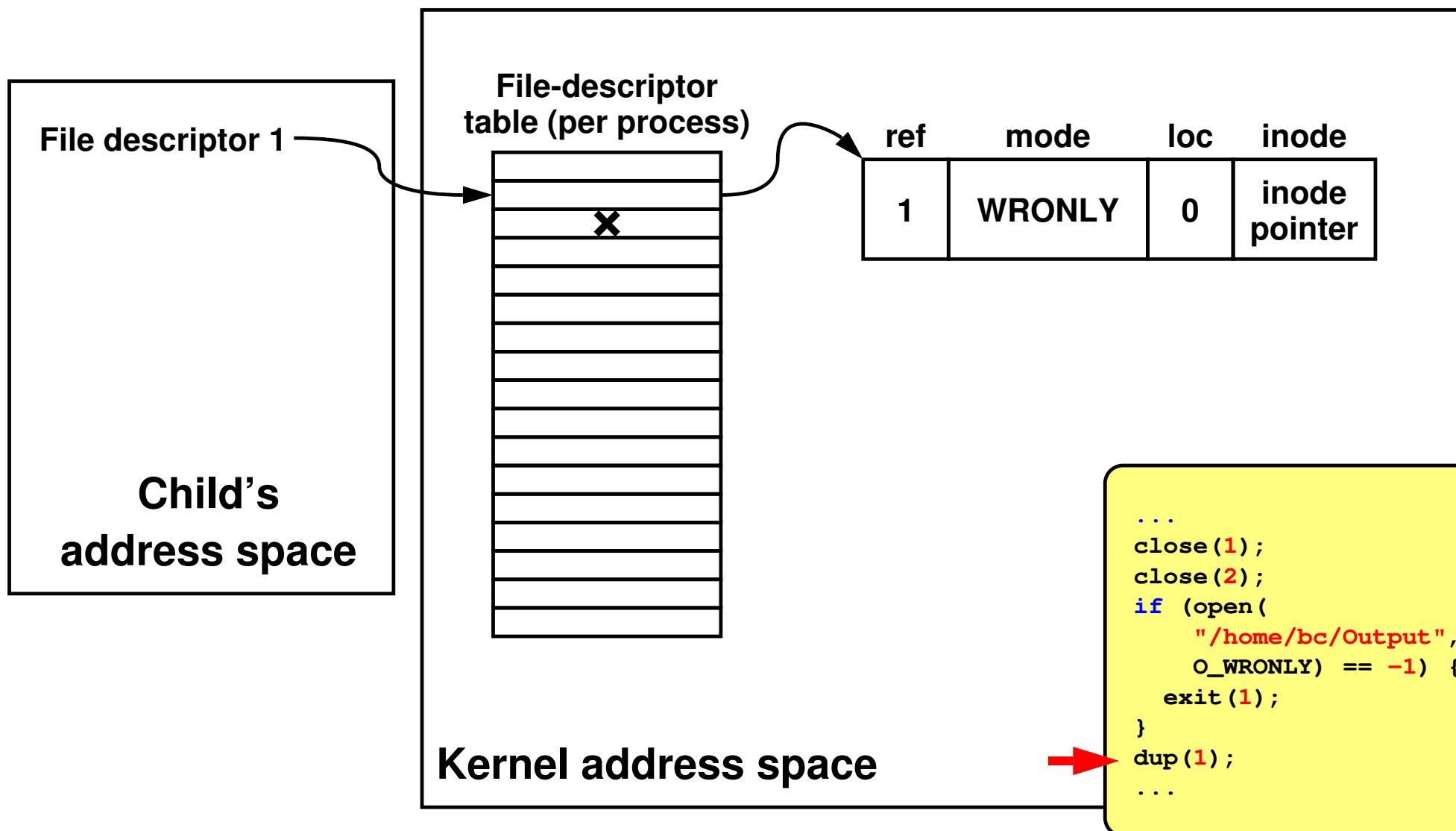
- use the `dup()` system call to **share** context information
  - if that's what you want



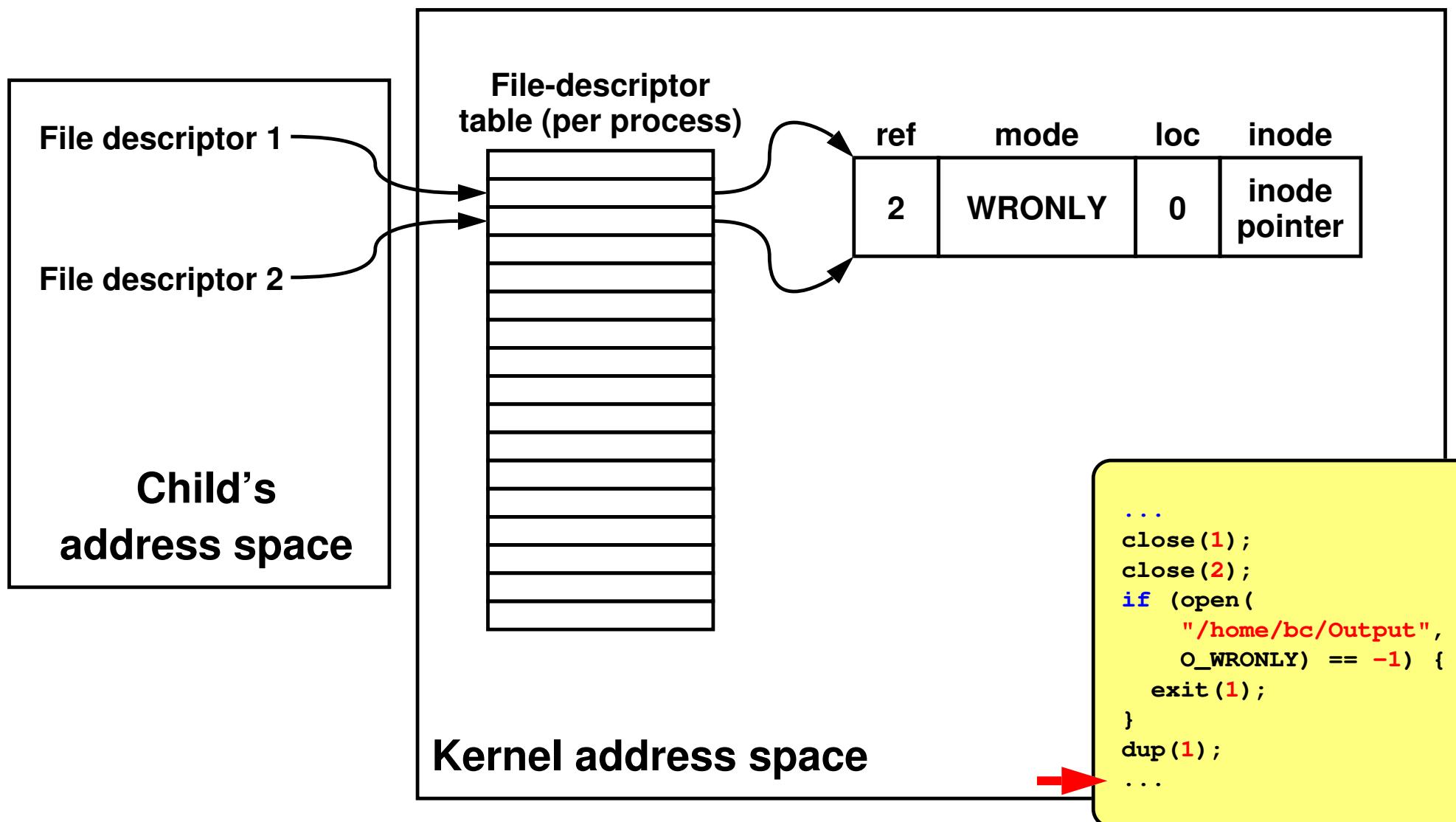
# Redirected Output After Dup



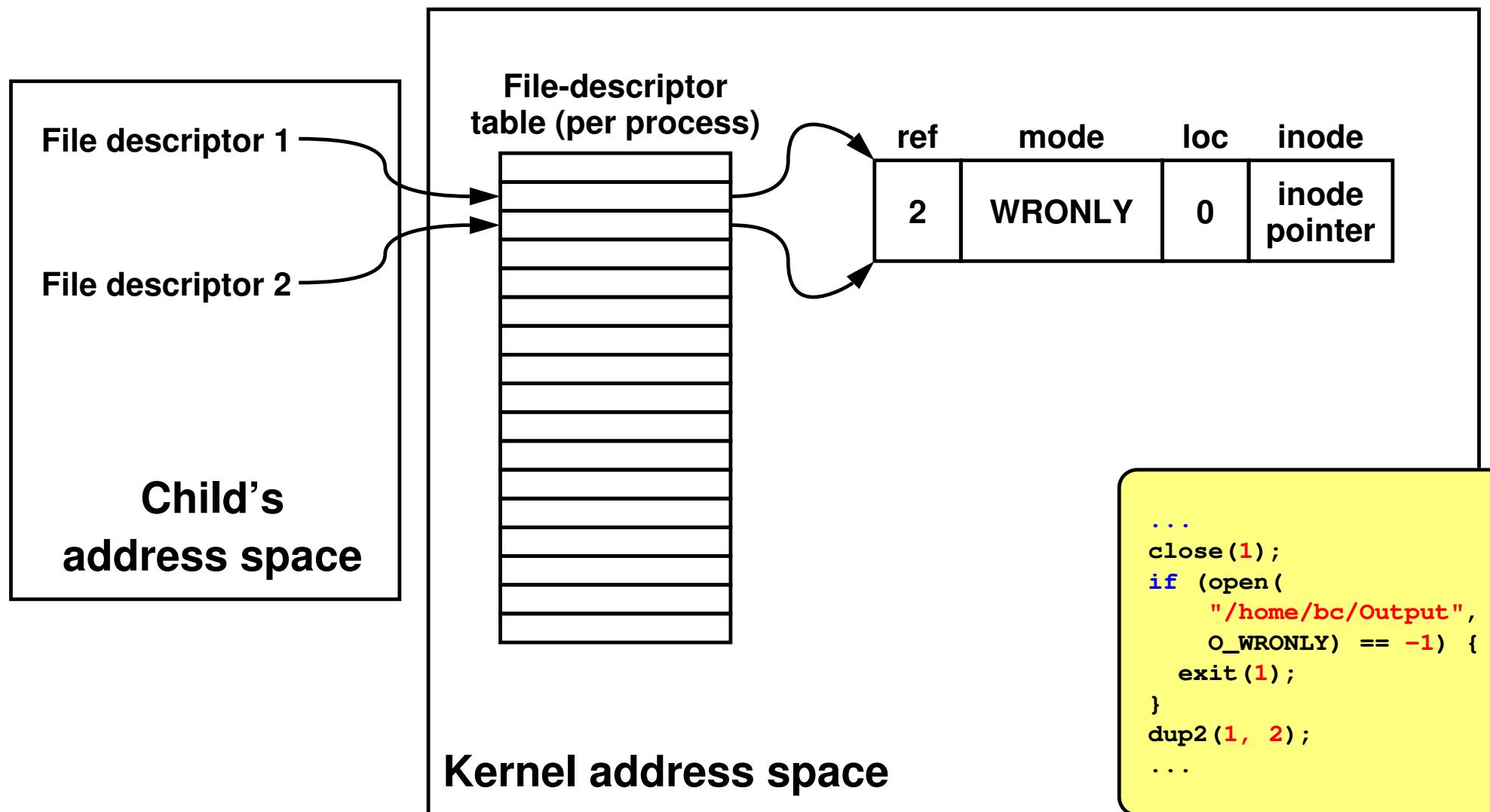
# Redirected Output After Dup



# Redirected Output After Dup



# Redirected Output After Dup



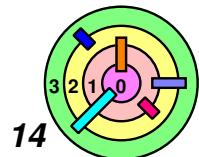
→ There is also a `dup2()` system call

# Fork and File Descriptors

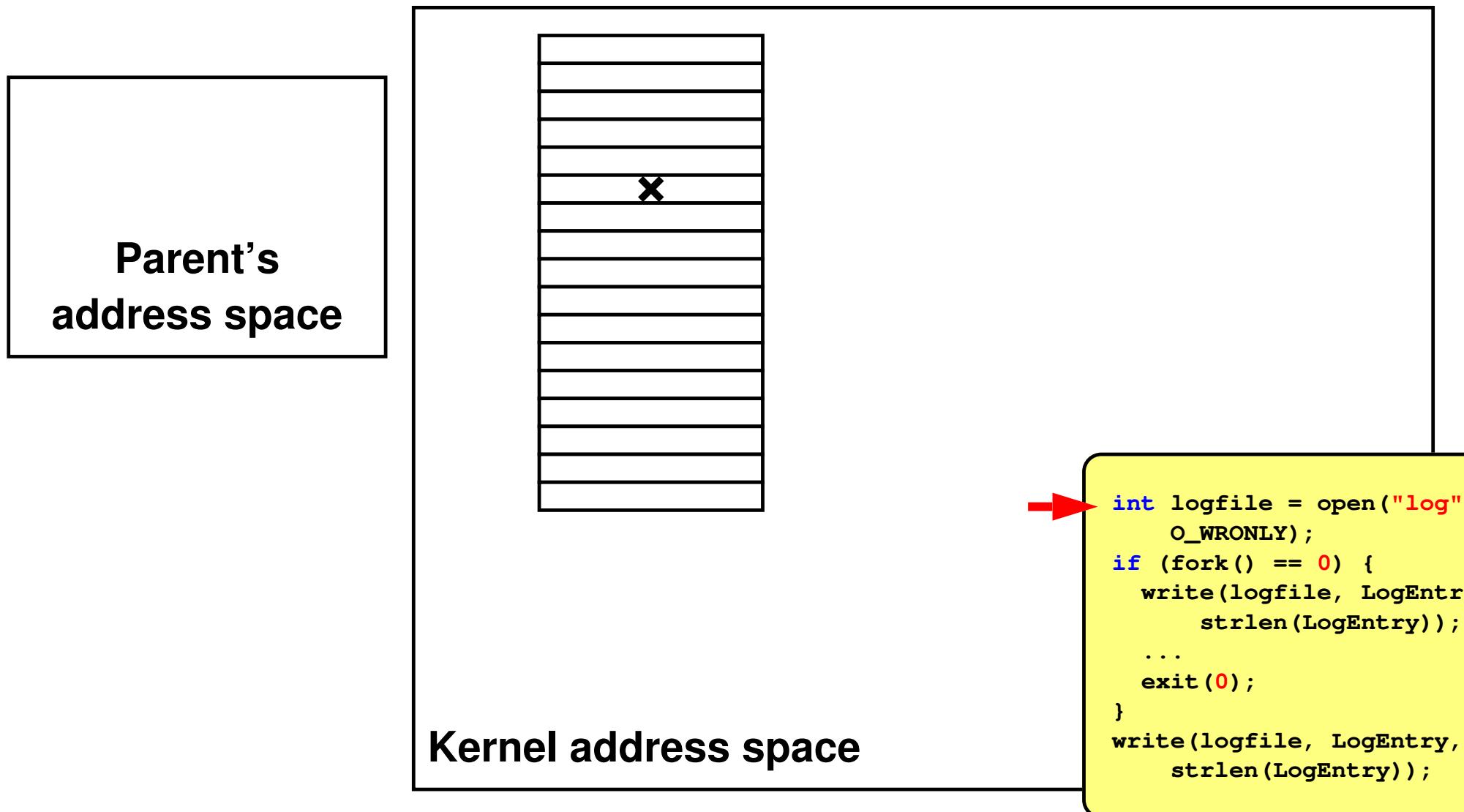
- It would be useful to be able to share file context information with a child process
- when `fork()` is called, the child process gets a *copy* of the parent's **file descriptor table**

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
    /* child process computes something, then does: */
    write(logfile, LogEntry, strlen(LogEntry));
    ...
    exit(0);
}
/* parent process computes something, then does: */
write(logfile, LogEntry, strlen(LogEntry));
...
```

- remember, extended address space survives execs
  - also `fork()`

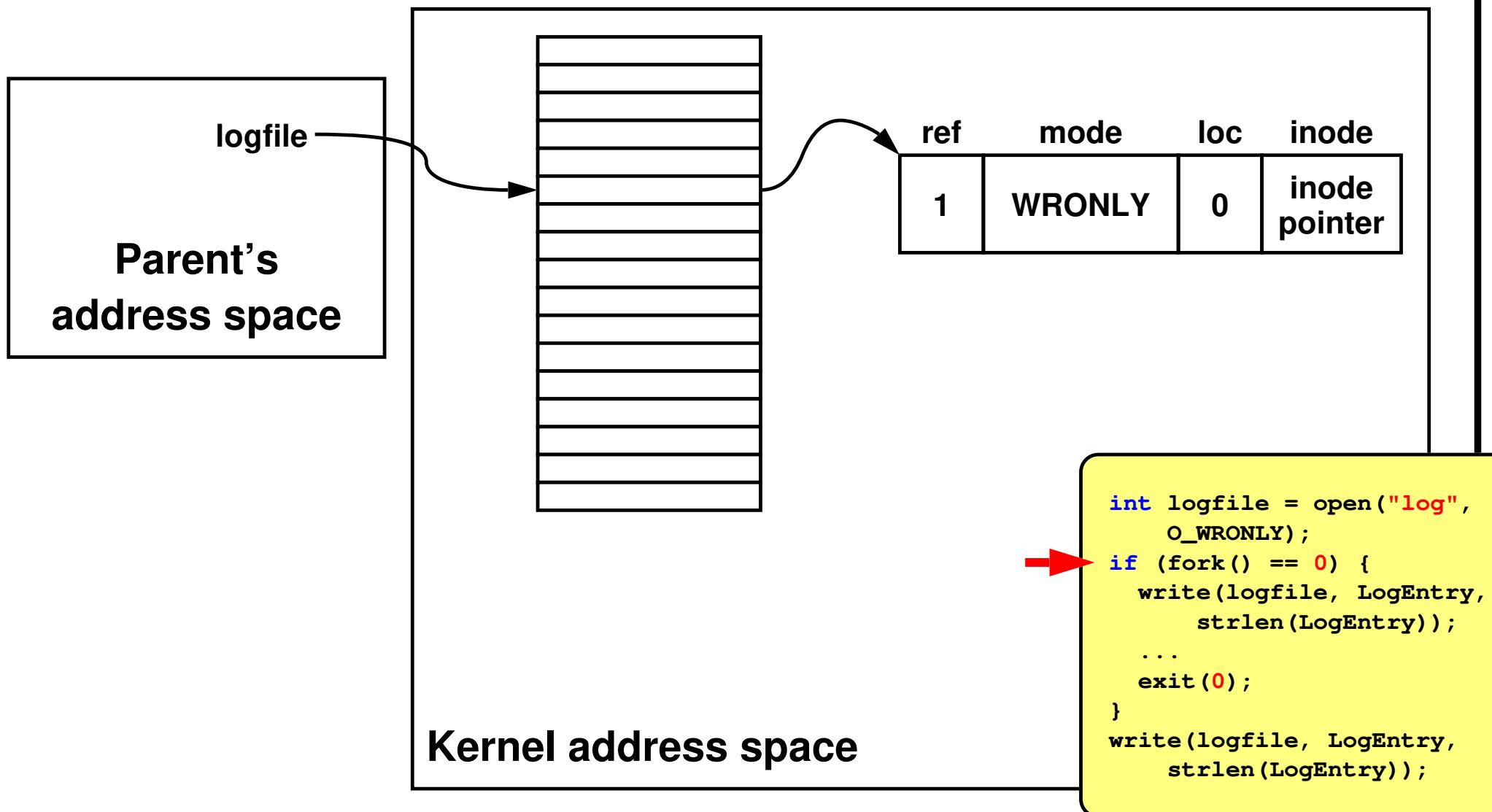


# File Descriptors After Fork



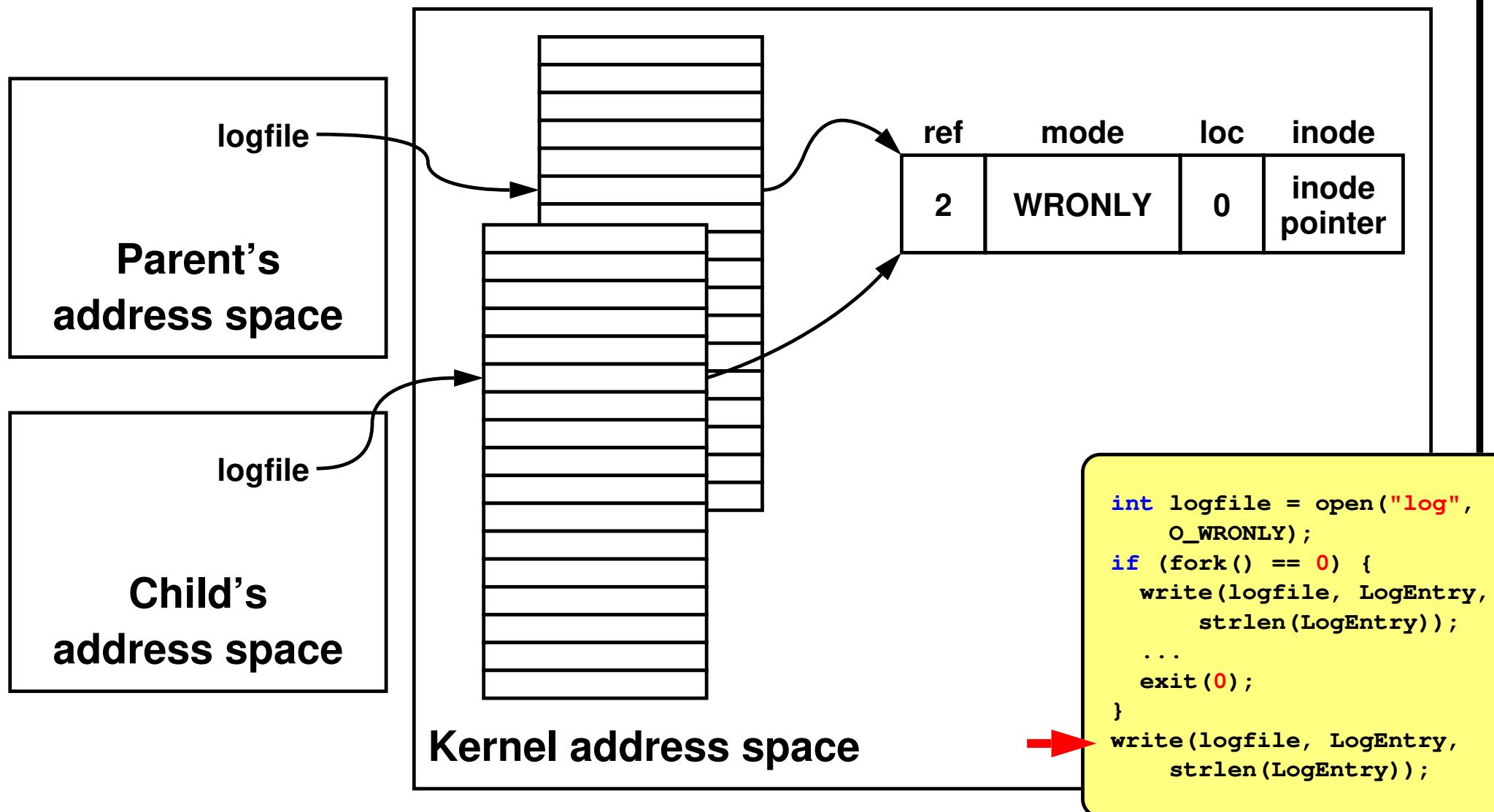
- = parent and child processes get **separate** file descriptor table but **share** extended address space

# File Descriptors After Fork

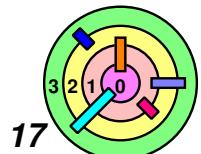


- = parent and child processes get **separate** file descriptor table but **share** extended address space

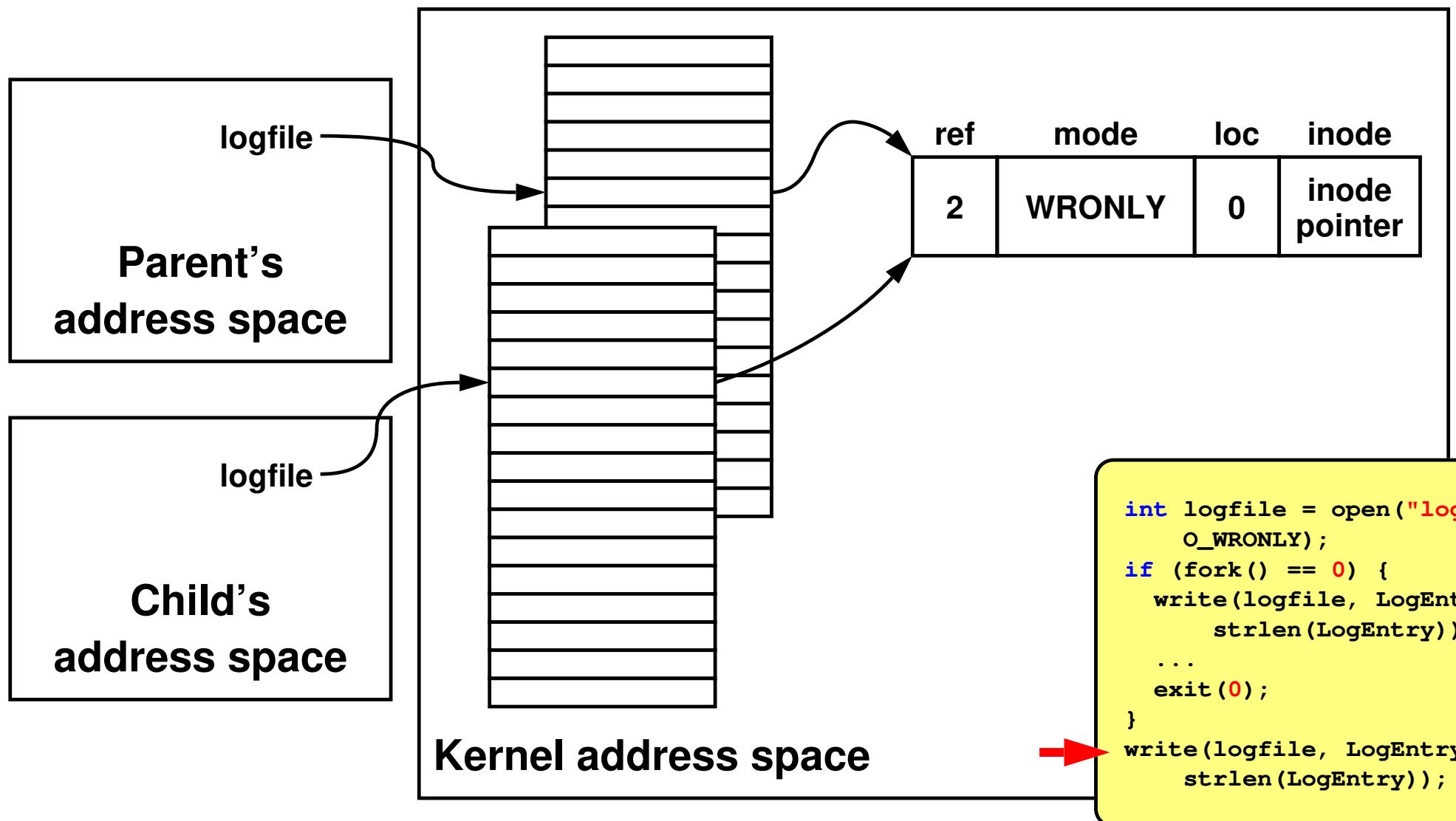
# File Descriptors After Fork



- = parent and child processes get **separate** file descriptor table but **share** extended address space **indirectly**



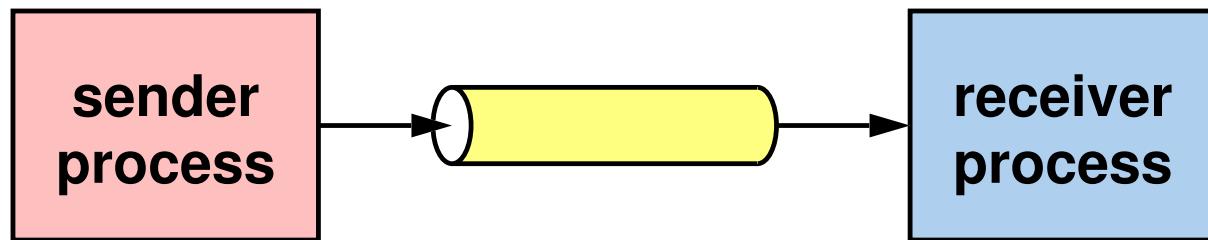
# File Descriptors After Fork



- = parent and child processes can communicate using such a shared file descriptor, although *difficult to synchronize*

# Pipes

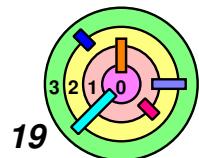
→ A pipe is a means for one process to send data to another directly, as if it were writing to a file



- the sending process behaves as if it has a file descriptor to a file that has been opened for writing
- the receiving process behaves as if it has a file descriptor to a file that has been opened for reading

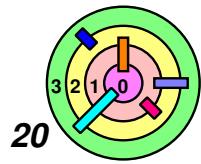
→ The `pipe()` system call creates a pipe object in the kernel and returns (via an output parameter) the two file descriptors that refer to the pipe

- one, set for write-only, refers to the input side
- the other, set for read-only, refers to the output side
- a pipe has no name, cannot be passed to another process

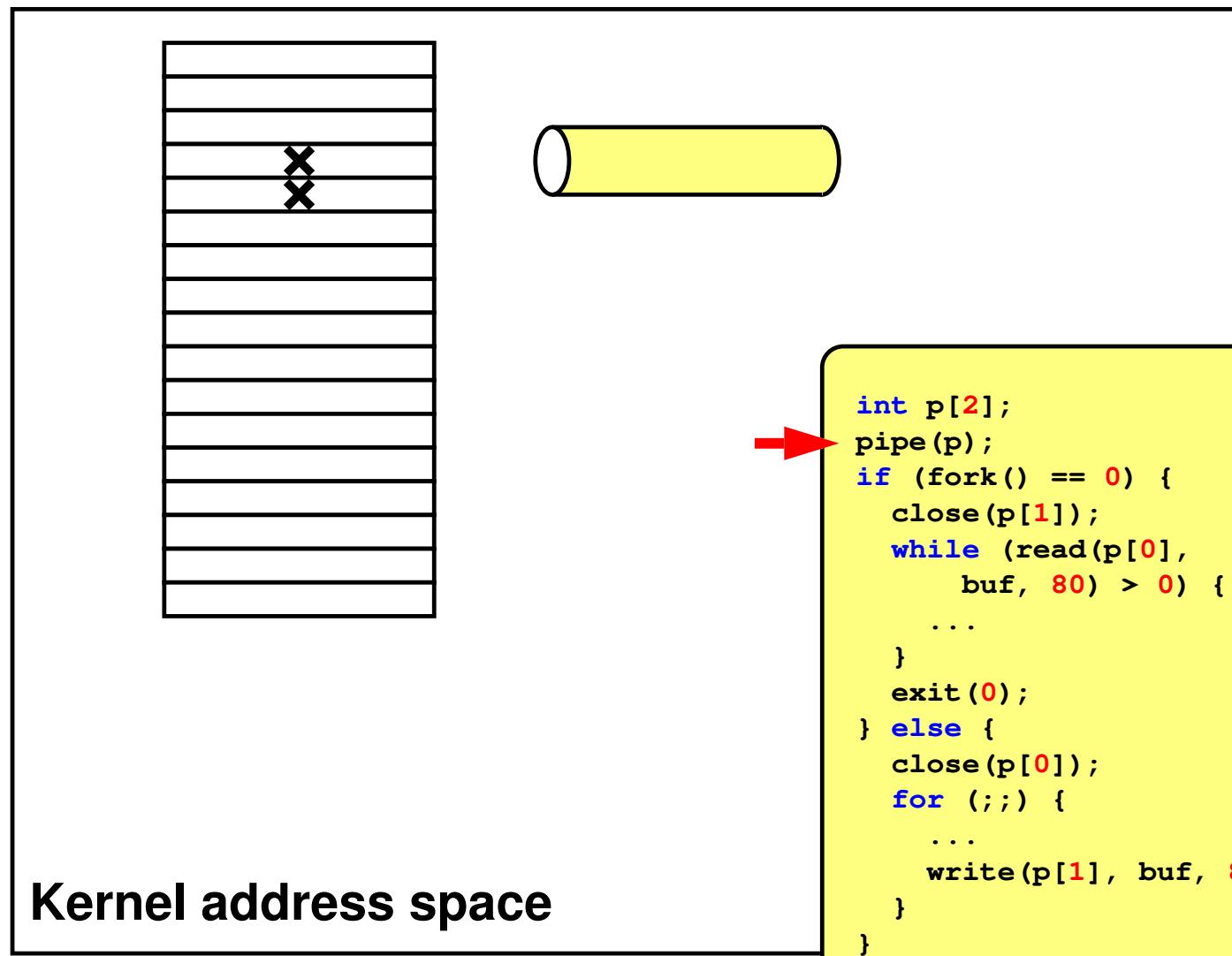


# Pipes

```
int p[2]; // array to hold pipe's file descriptors
pipe(p); // creates a pipe, assume no errors
    // p[0] refers to the read/output end of the pipe
    // p[1] refers to the write/input end of the pipe
if (fork() == 0) {
    char buf[80];
    close(p[1]); // not needed by the child
    while (read(p[0], buf, 80) > 0) {
        // use data obtained from parent
        ...
    }
    exit(0); // child done
} else {
    char buf[80];
    close(p[0]); // not needed by the parent
    for (;;) {
        // prepare data for child
        ...
        write(p[1], buf, 80);
    }
}
```

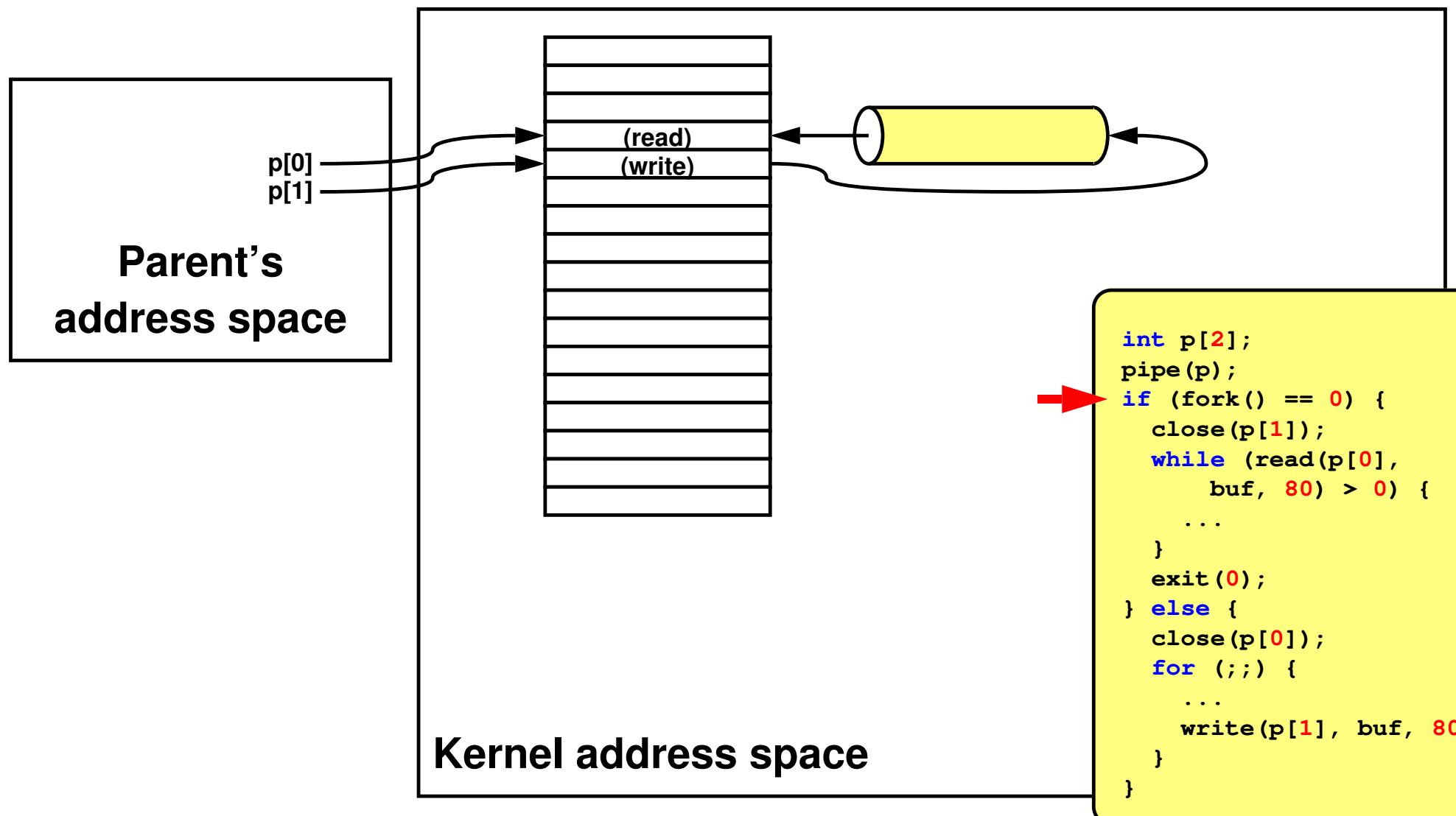


# Pipes



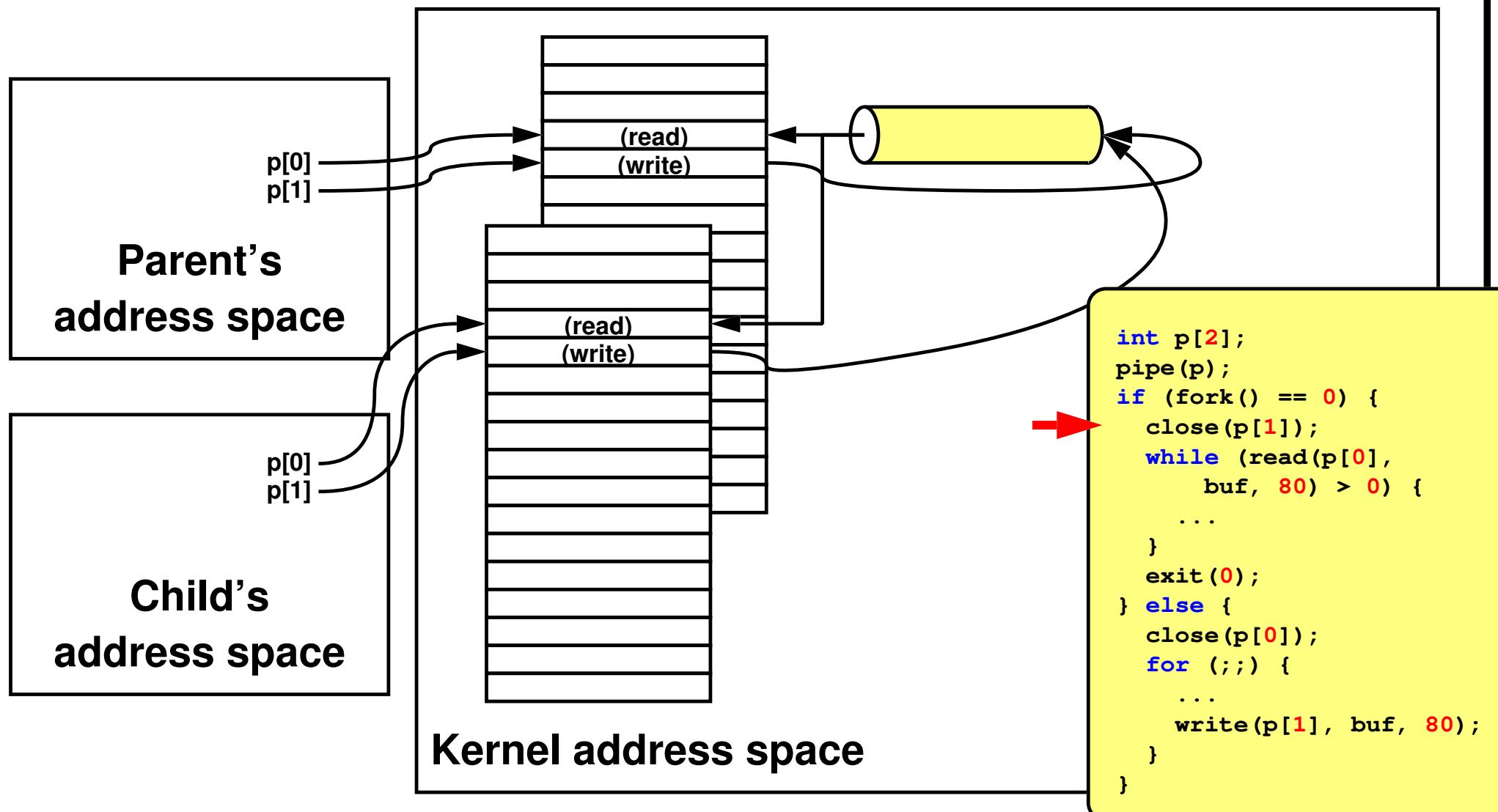
- parent creates a pipe object in the kernel

# Pipes

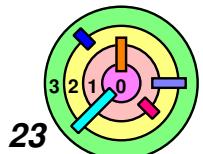


- parent creates a pipe object in the kernel

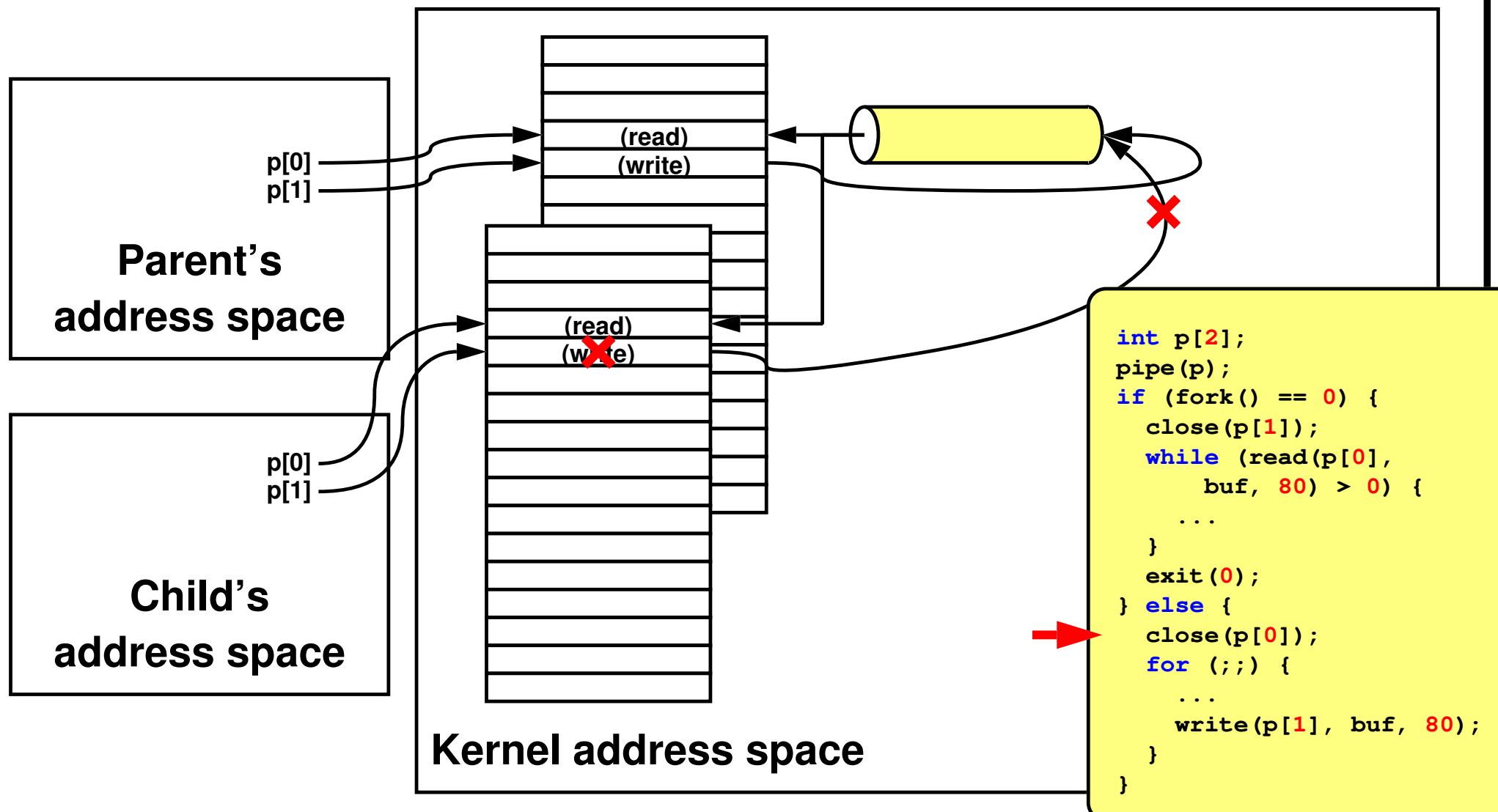
# Pipes



- parent and child processes get **separate** file descriptor tables but **share** extended address space

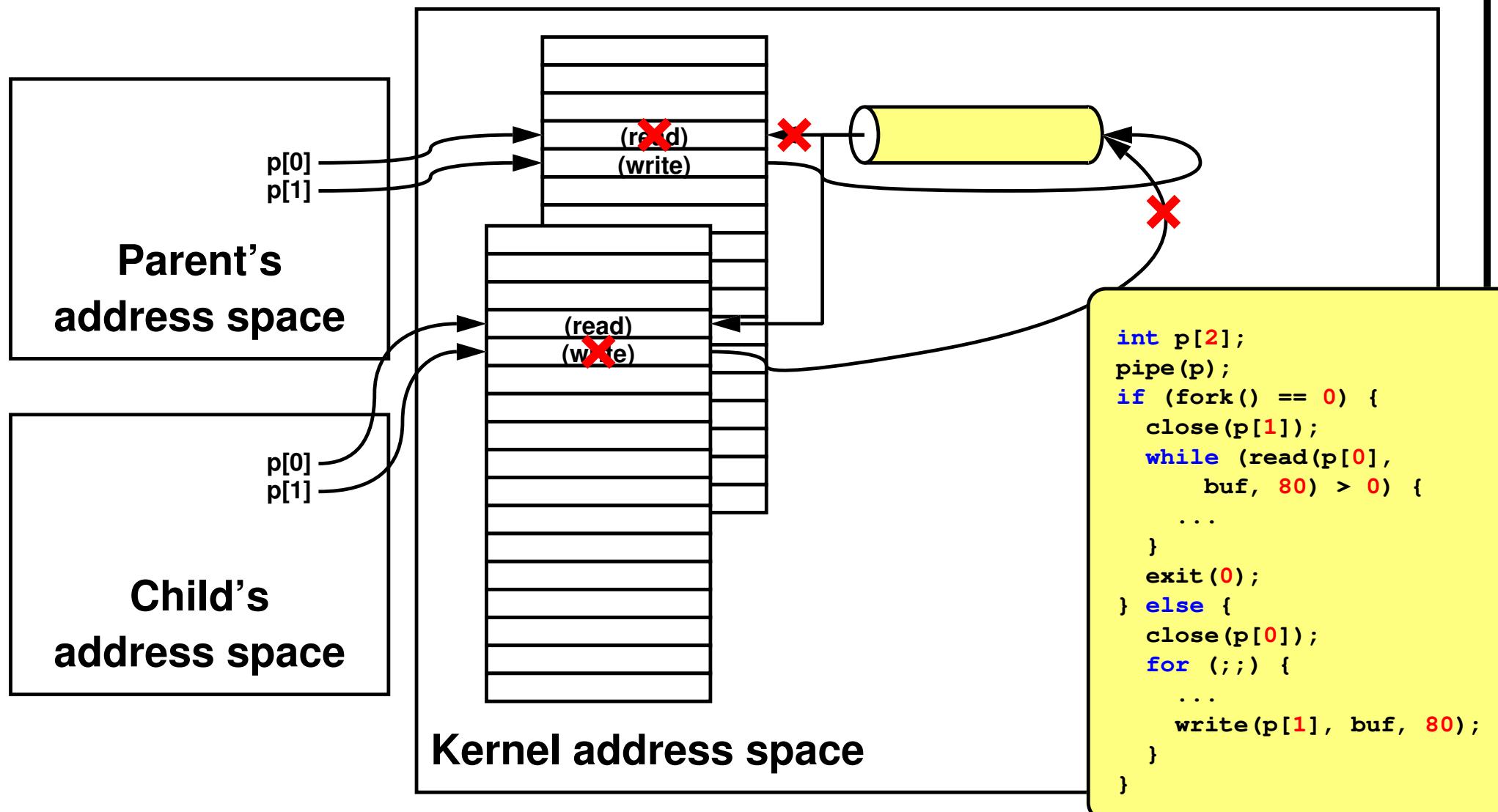


# Pipes



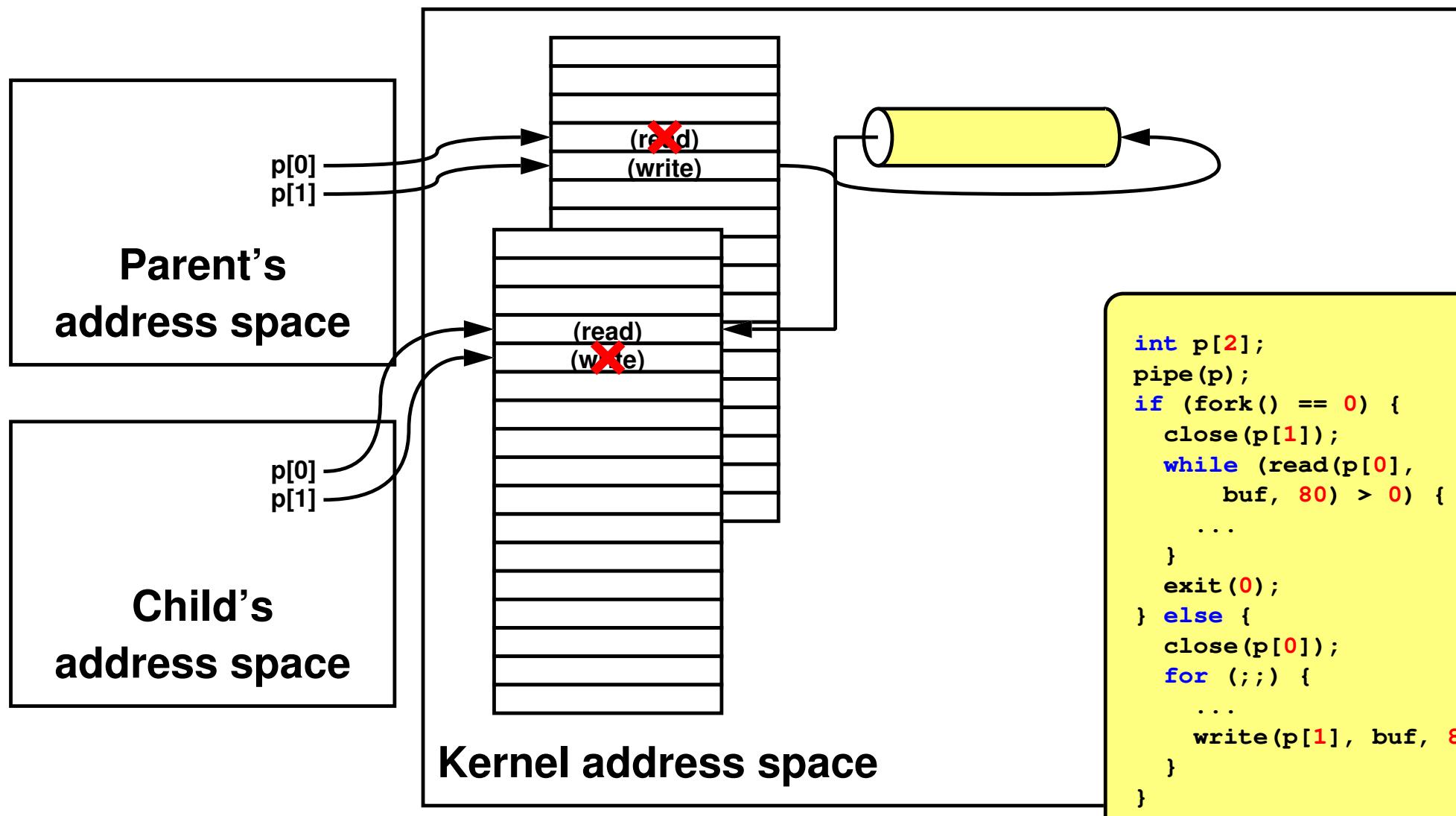
- child closes the write-end of the pipe

# Pipes



- child closes the write-end of the pipe
- parent closes the read-end of the pipe

# Pipes



```

int p[2];
pipe(p);
if (fork() == 0) {
    close(p[1]);
    while (read(p[0], buf, 80) > 0) {
        ...
    }
    exit(0);
} else {
    close(p[0]);
    for (;;) {
        ...
        write(p[1], buf, 80);
    }
}

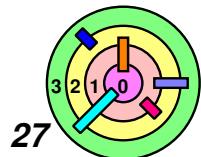
```

- child closes the write-end of the pipe
- parent closes the read-end of the pipe

# Command Shell

- Now you know enough to write a command shell
  - execute a command
  - redirect I/O
  - pipe the output of one program to another
    - cat f0 | ./warmup1 sort
    - the shell needs to create a pipe
    - create two child processes
    - in the first child
      - ◊ have stdout go to the write-end of the pipe
      - ◊ close the read-end of the pipe
      - ◊ exec "cat f0"
    - in the 2nd child
      - ◊ have stdin come from the read-end of the pipe
      - ◊ close the write-end of the pipe
      - ◊ exec "./warmup1 sort"
  - run a program in the background

```
primes 1000000 > primes.out &
```



# Random Access In Sequential I/O

```

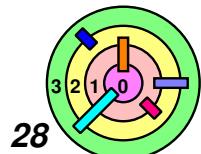
fd = open("textfile", O_RDONLY);
// go to last char in file
fptr = lseek(fd, (off_t)(-1), SEEK_END);
while (fptr != -1) {
    read(fd, buf, 1);
    write(1, buf, 1);
    fptr = lseek(fd, (off_t)(-2), SEEK_CUR);
}

```

- "man lseek" gives

off\_t lseek(int fd, off\_t offset, int whence);

- whence can be SEEK\_SET, SEEK\_CUR, SEEK\_END
- if succeeds, returns cursor position (always measured from the beginning of the file)
  - otherwise, returns (-1)
  - errno is set to indicate the error
- read(fd, buf, 1) advances the cursor position by 1, so we need to move the cursor position back 2 positions



# More On Naming

→ (Almost) everything has a path name

- files
- directories
- devices (*known as special files*)
  - keyboards, displays, disks, etc.

→ Uniformity

```
// opening a normal file
int file = open("/home/bc/data", O_RDWR);

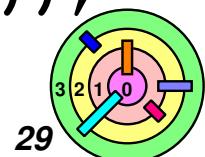
// opening a device (one's terminal or window)
int device = open("/dev/tty", O_RDWR);

int bytes = read(file, buffer, sizeof(buffer));
write(device, buffer, bytes);
```

- in warmup1, we saw that you can open a directory

```
read(open("/etc", O_RDONLY), buf, sizeof(buf));
```

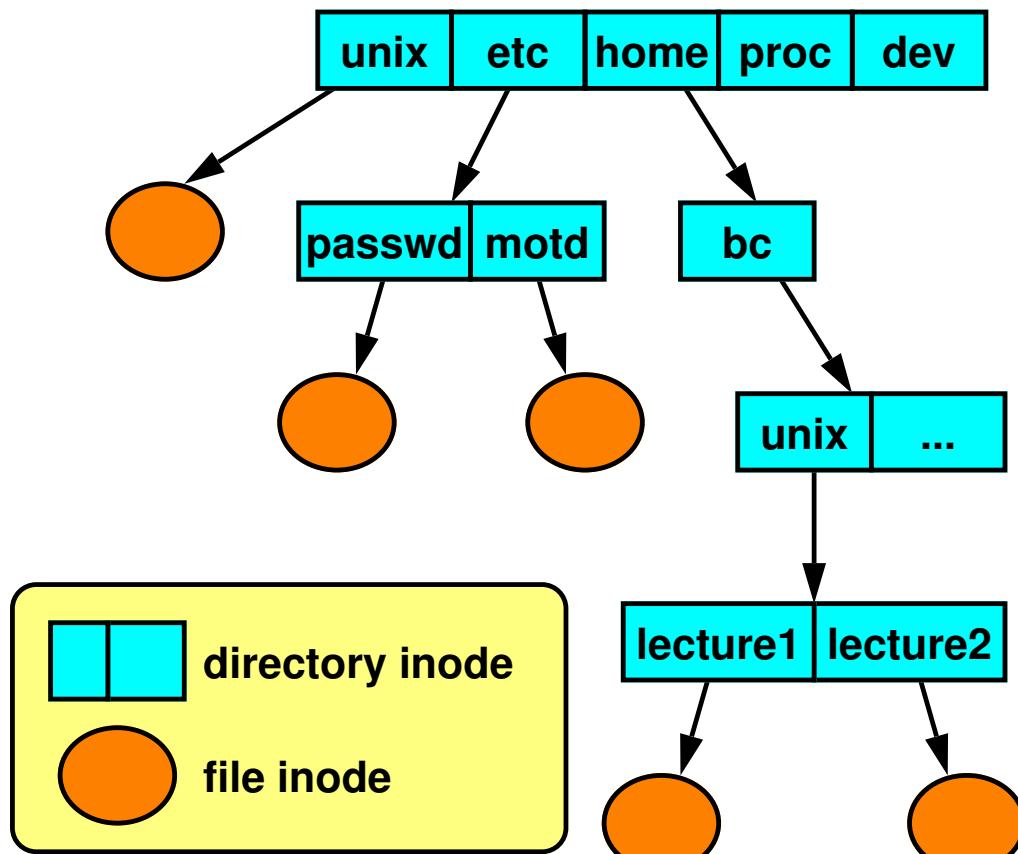
- although *reading from a directory is not standardized*



# Directories

→ A **directory** is a **file**

- interprets differently by the OS as containing **references** to other files/directories
- a file is represented as an **index node** (or **inode**) in the file system

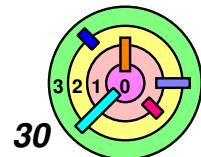


→ A **directory maps** a **file name** to an **inode number**

- maps a string to an integer
- done inside Virtual File System in weenix

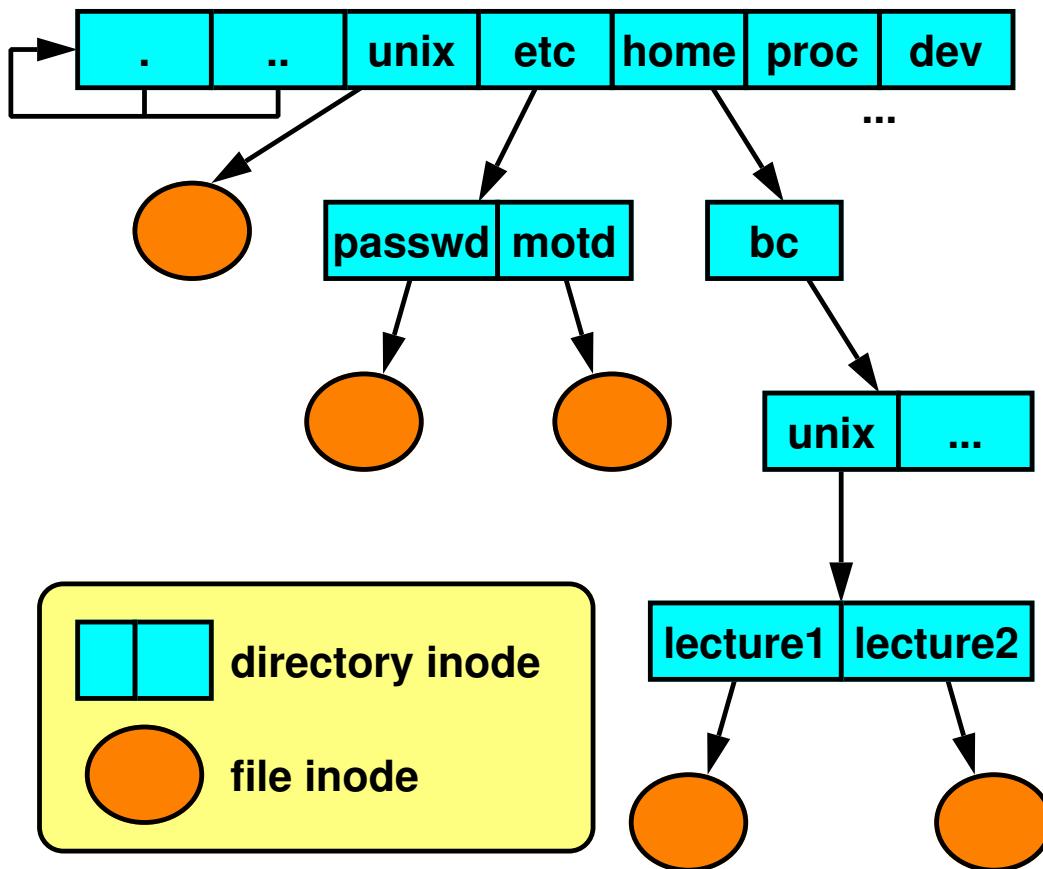
→ An **inode** maps an **inode number** to **disk address**

- done inside Actual File System in weenix



# Directory Representation

- A root directory entry example
- parent inode number = its own inode number



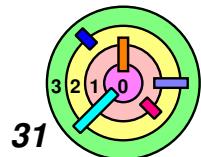
→ Tree structured hierarchy

Component Name	Inode number
----------------	--------------

directory entry

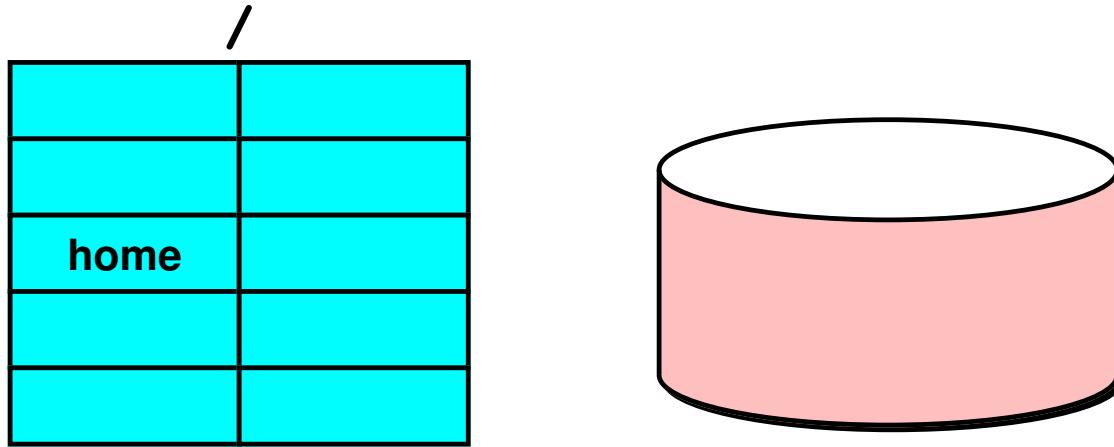
.	1
..	1
unix	117
etc	4
home	18
proc	36
dev	93

this is what a S5FS (and RAMFS) directory looks like in weenix

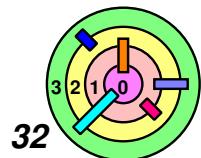


# Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "/home/bc/foo.c"?  
— this process is called *pathname resolution*

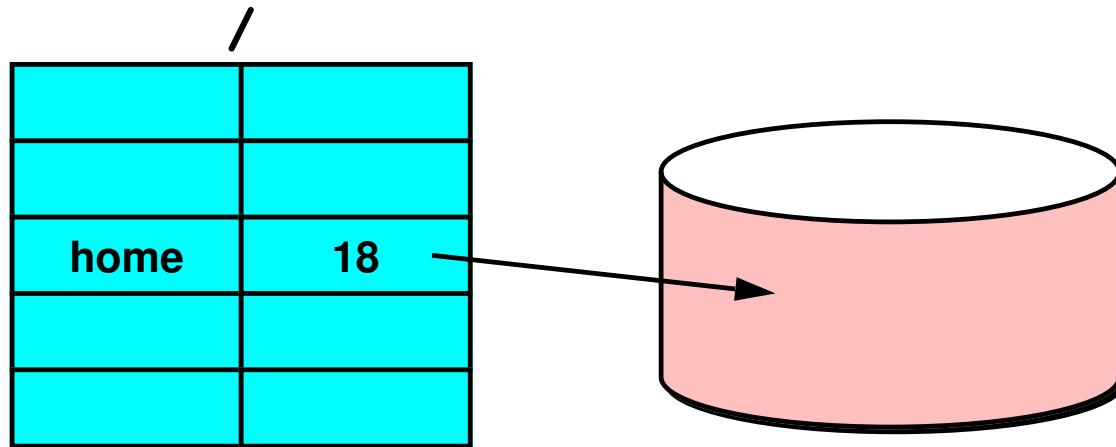


- AFS creates an *inode pointer* to represent the directory file



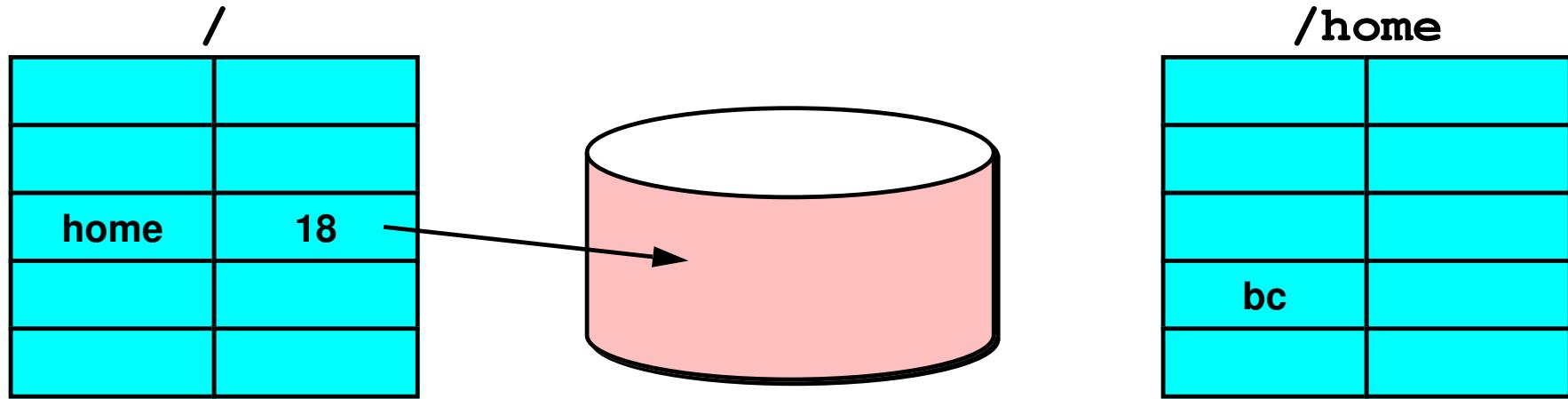
# Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "/home/bc/foo.c"?  
— this process is called *pathname resolution*



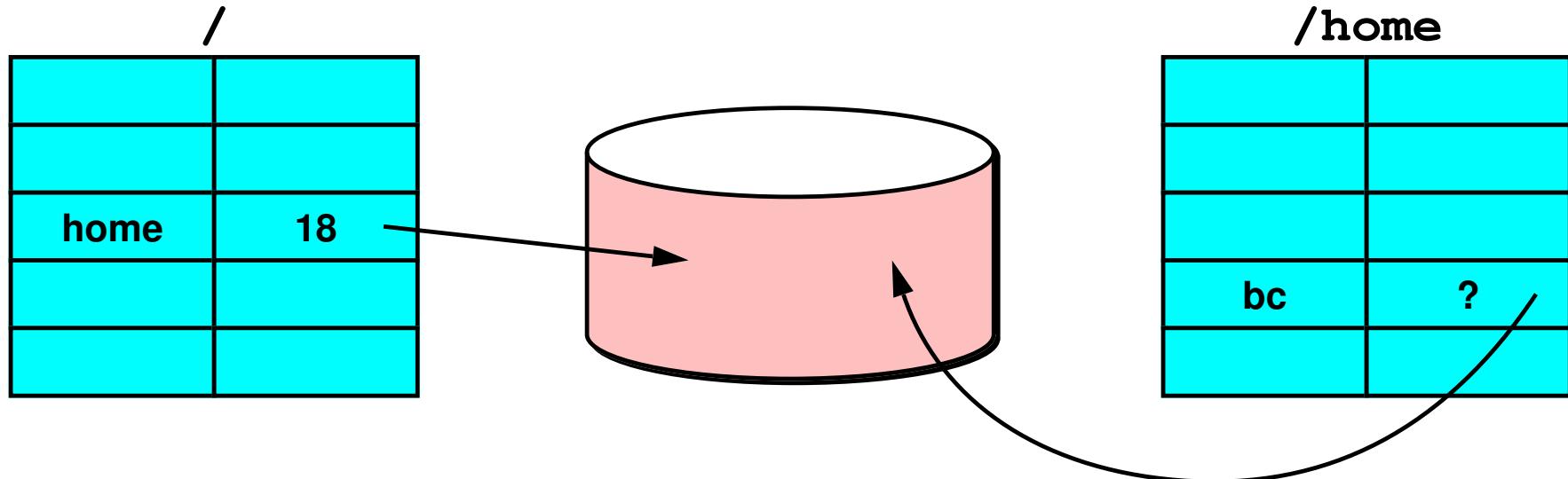
# Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "/home/bc/foo.c"?  
— this process is called *pathname resolution*



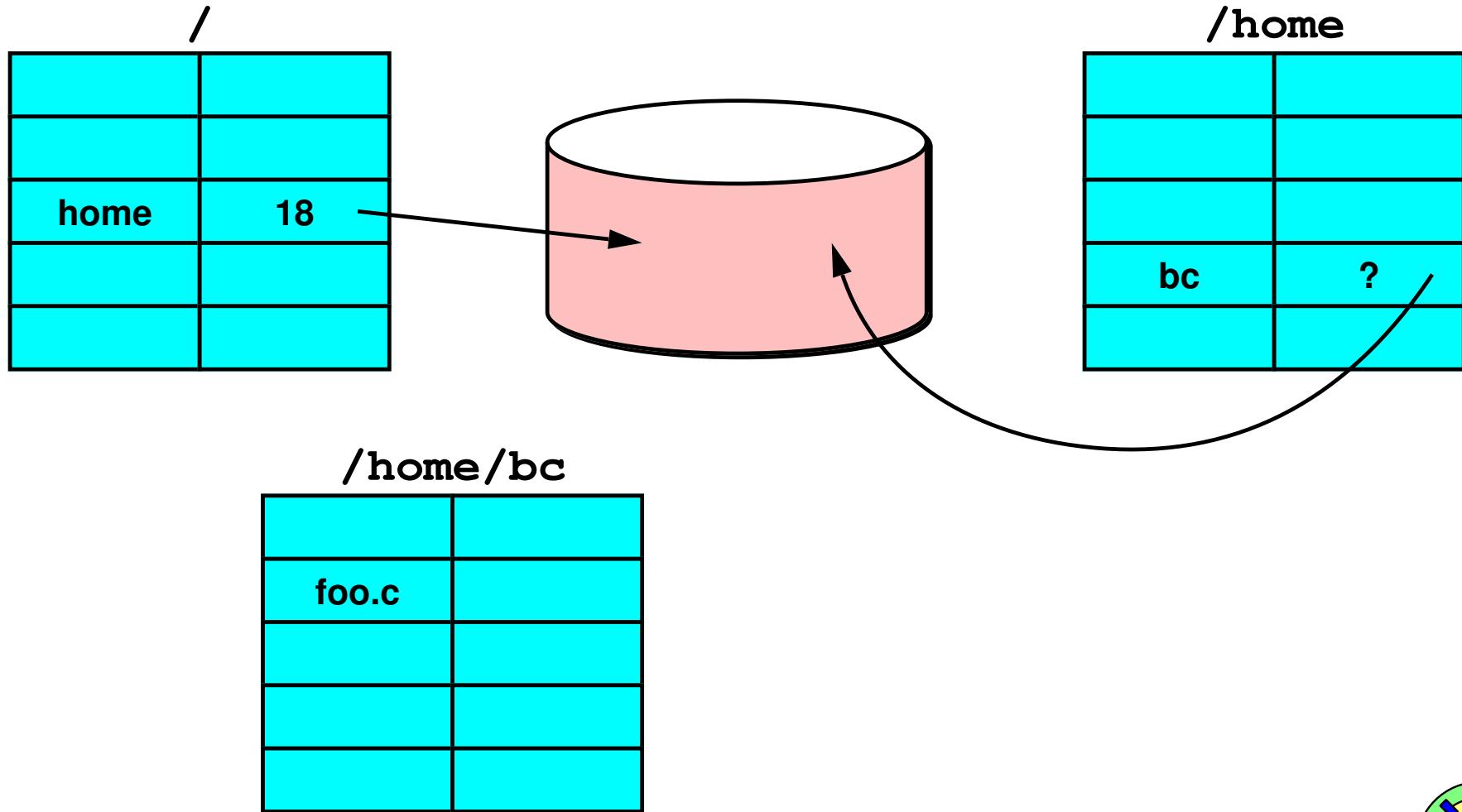
# Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "/home/bc/foo.c"?  
— this process is called *pathname resolution*



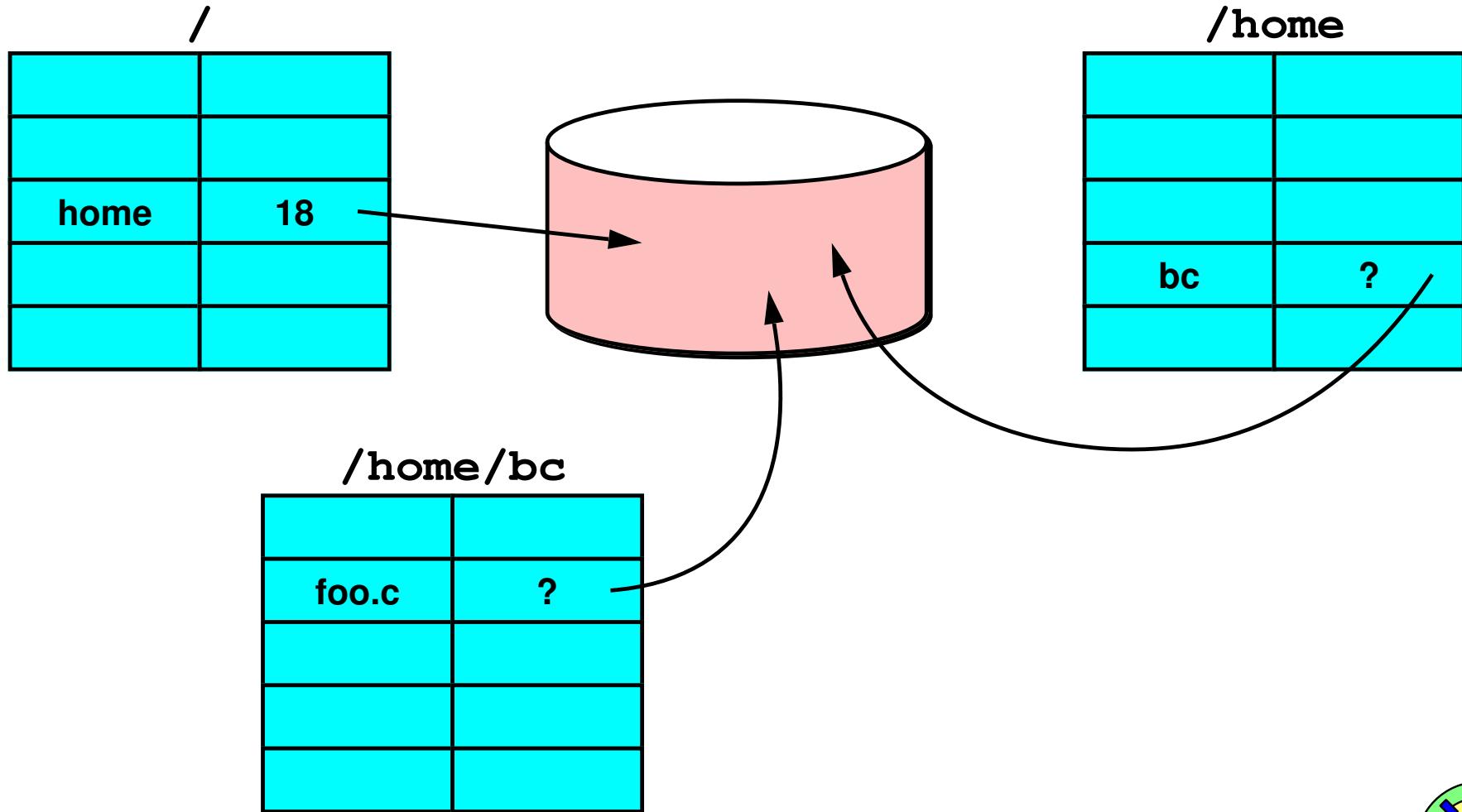
# Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "/home/bc/foo.c"?  
— this process is called *pathname resolution*



# Look Up Inode Number Of A Path

- Ex: how do figure out the inode number of "/home/bc/foo.c"?  
— this process is called *pathname resolution*



su21-a-Q16

# Directory Hierarchy

→ Unix and many other OSes allow limited deviation from trees

- **hard links**

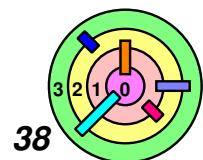
- reference to a file in one directory that also appears in another
- can use the `link()` system call or the "`ln`" shell command to *add* a hard link to a *file* (but not a directory)

- **soft links** or **symbolic links**

- a special kind of file containing the *name* of another file or directory
- can use the `symlink()` system call or the "`ln -s`" shell command to add a symbolic link to any type of file

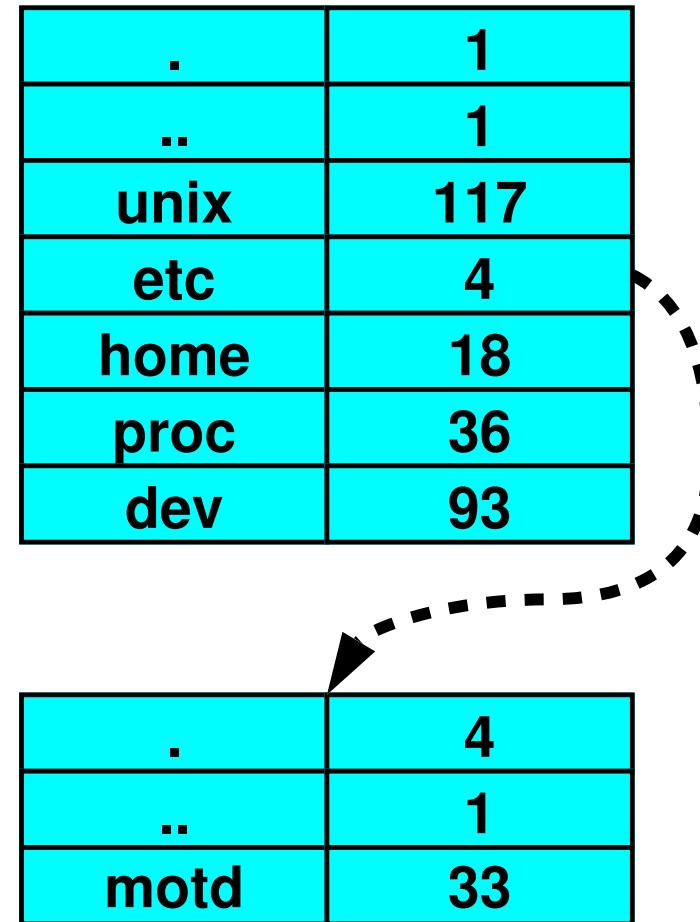
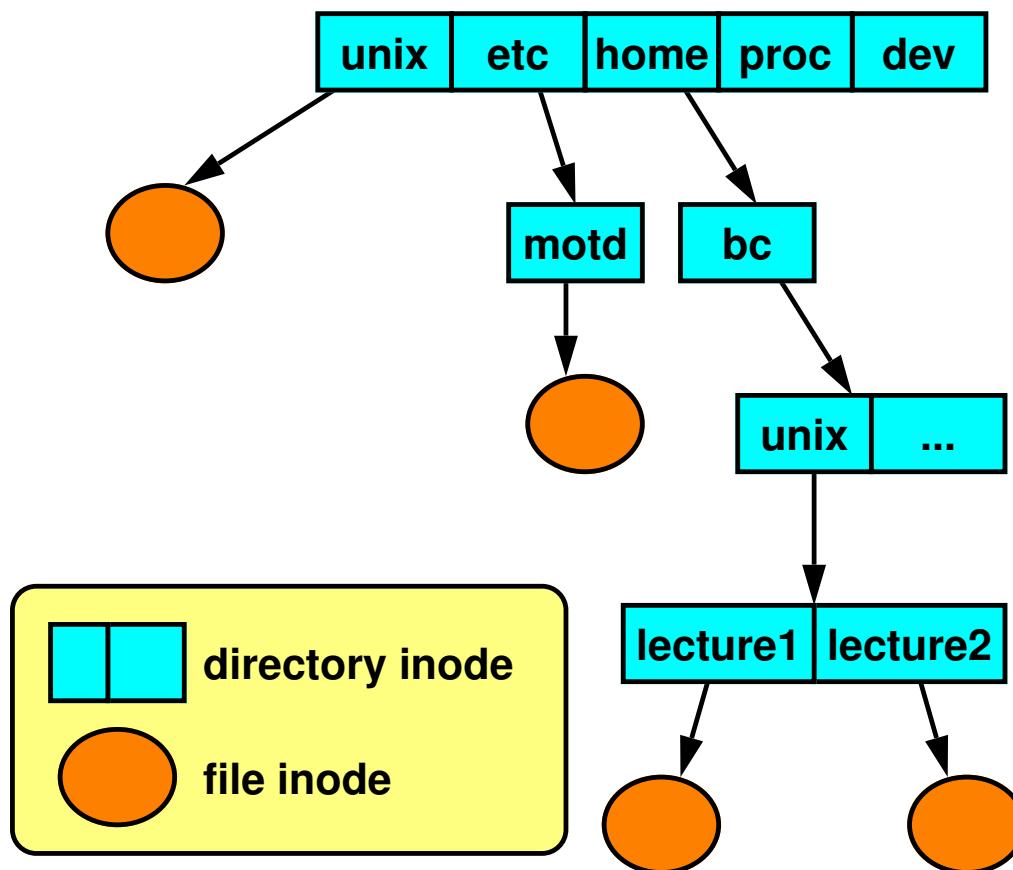
→ Why `link()` cannot be used on a directory?

- to avoid cycles
- Unix directory hierarchy can be viewed as a *directed acyclic graph* (DAG)
- straight-forward algorithm to traverse directory hierarchy efficiently



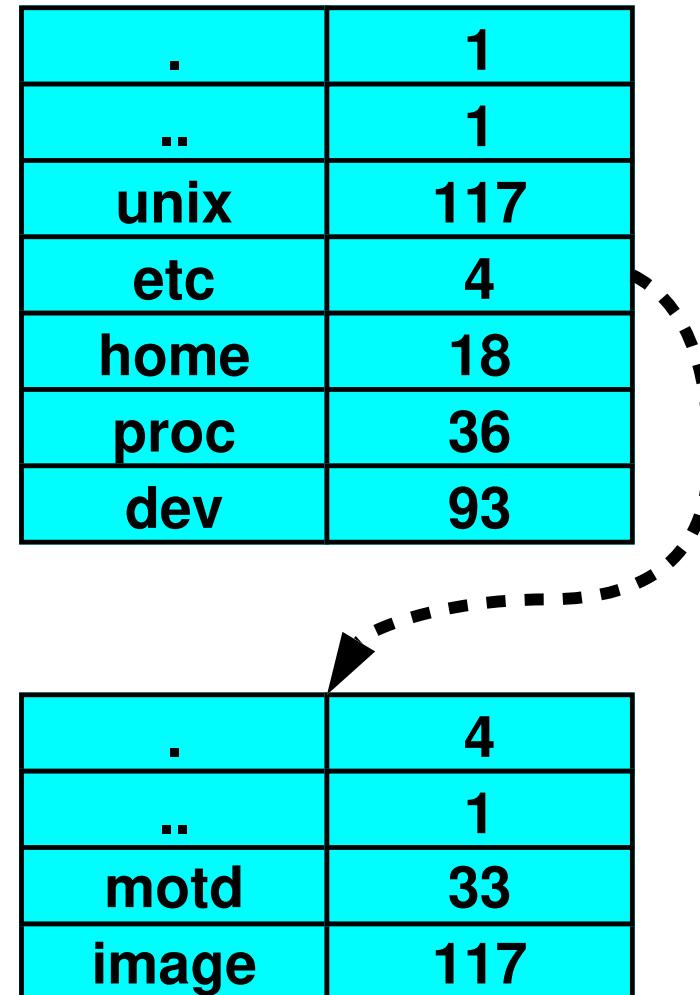
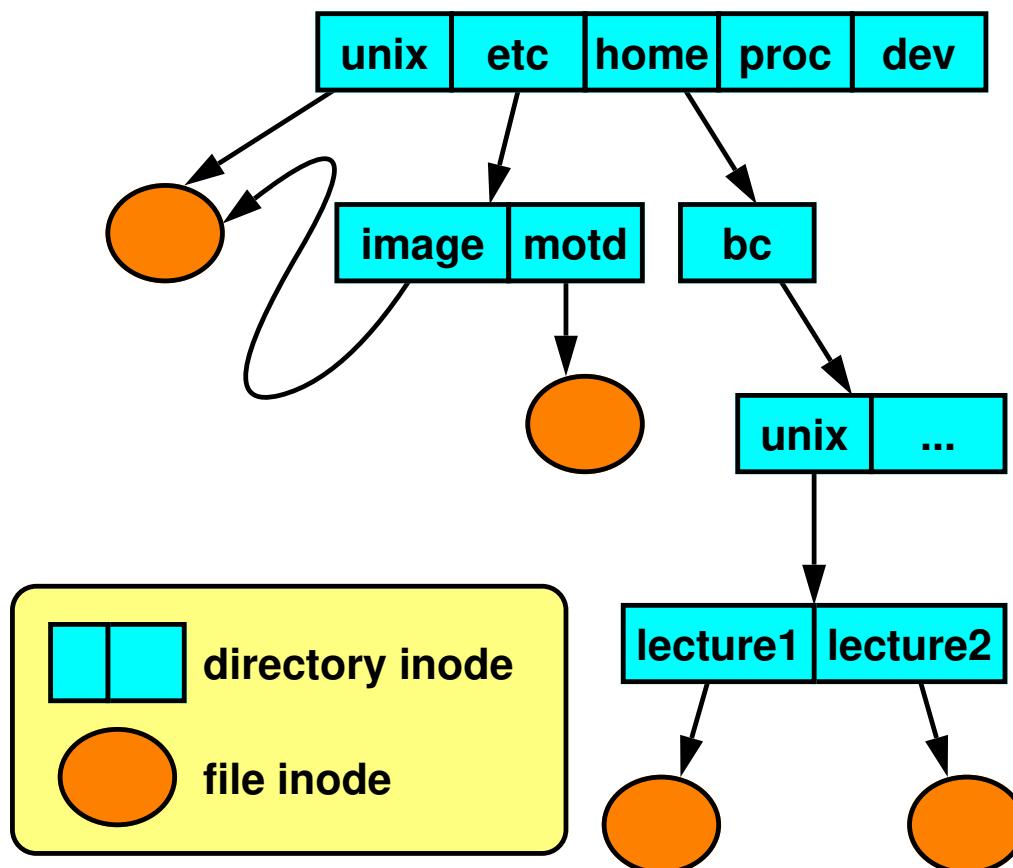
# Hard Links

```
% ln /unix /etc/image
  - create "image" in "/etc" to
    link to "/unix"
```



# Hard Links

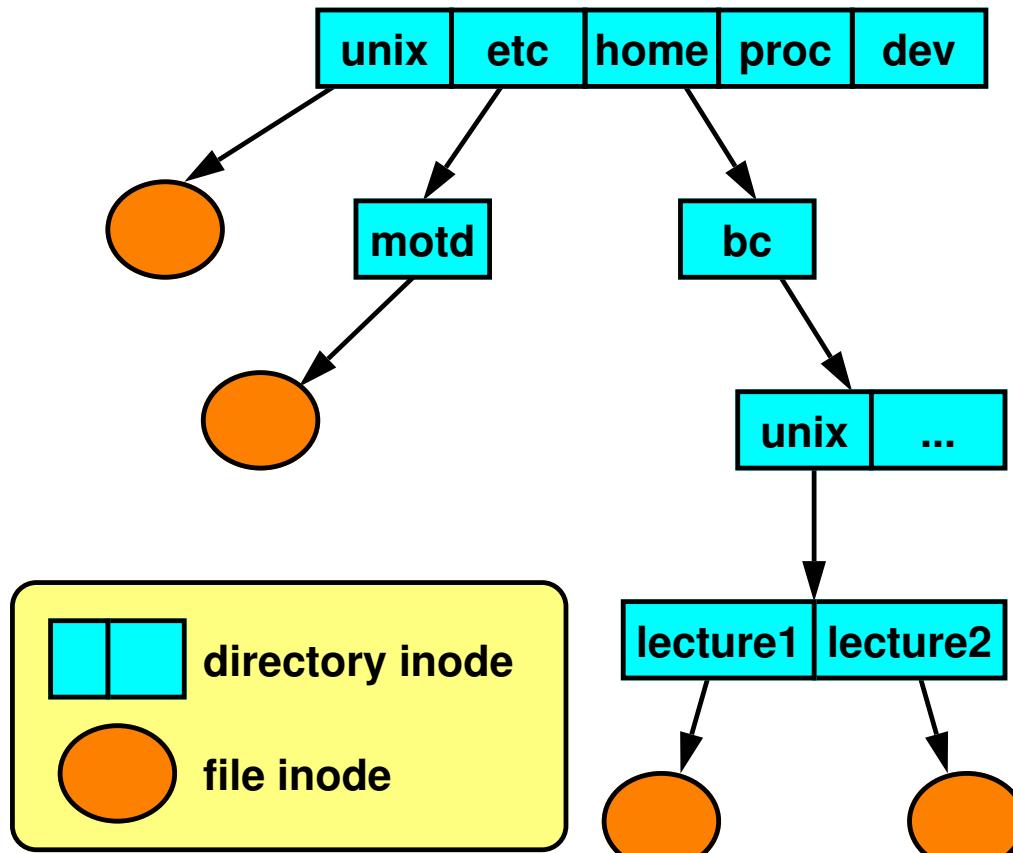
```
% ln /unix /etc/image
  - create "image" in "/etc" to
    link to "/unix"
```



# Soft Links

```
% ln -s /unix /home/bc/mylink
```

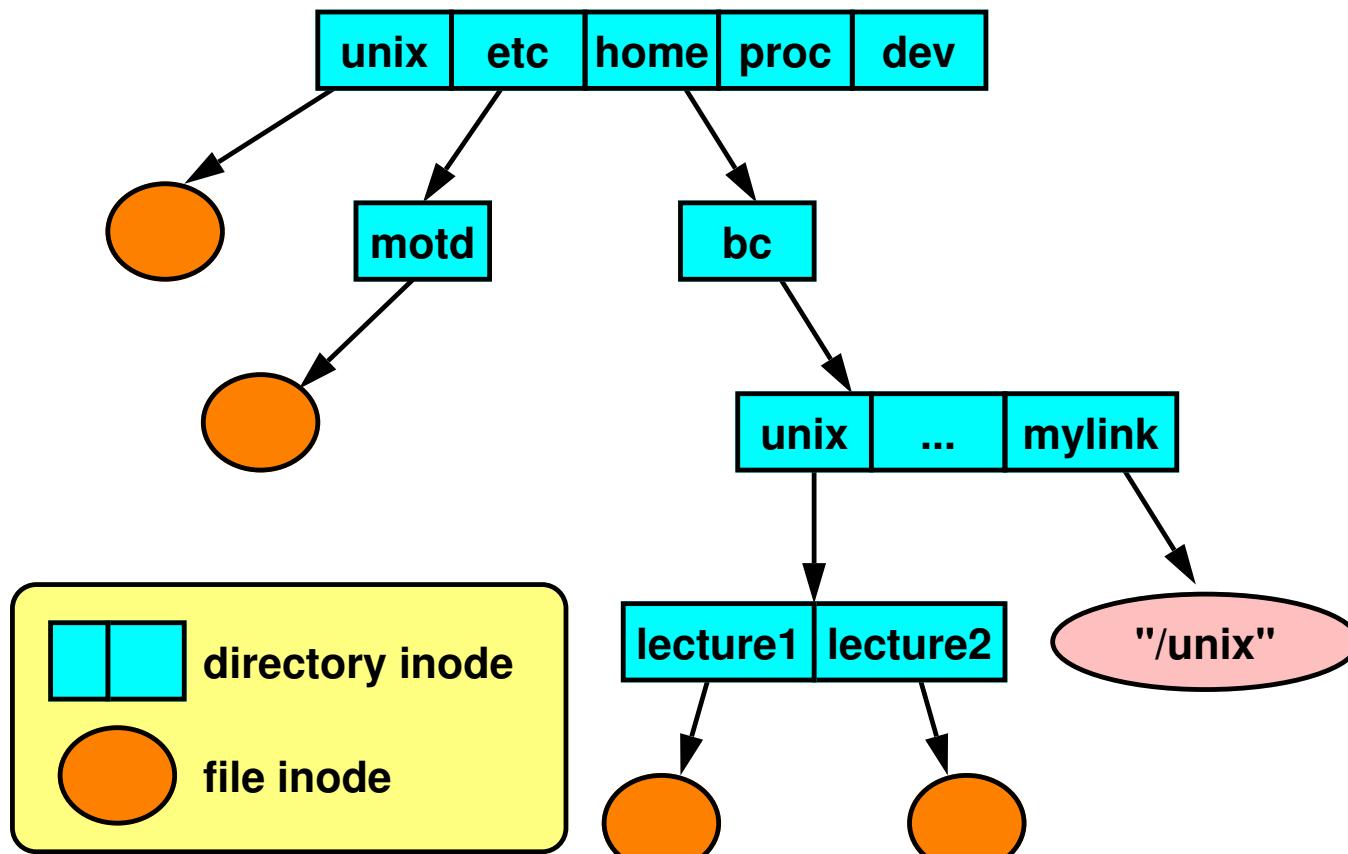
- create "mylink" in "/home/bc" to **soft-link** to "/unix"



# Soft Links

```
% ln -s /unix /home/bc/mylink
```

- create "mylink" in "/home/bc" to **soft-link** to "/unix"

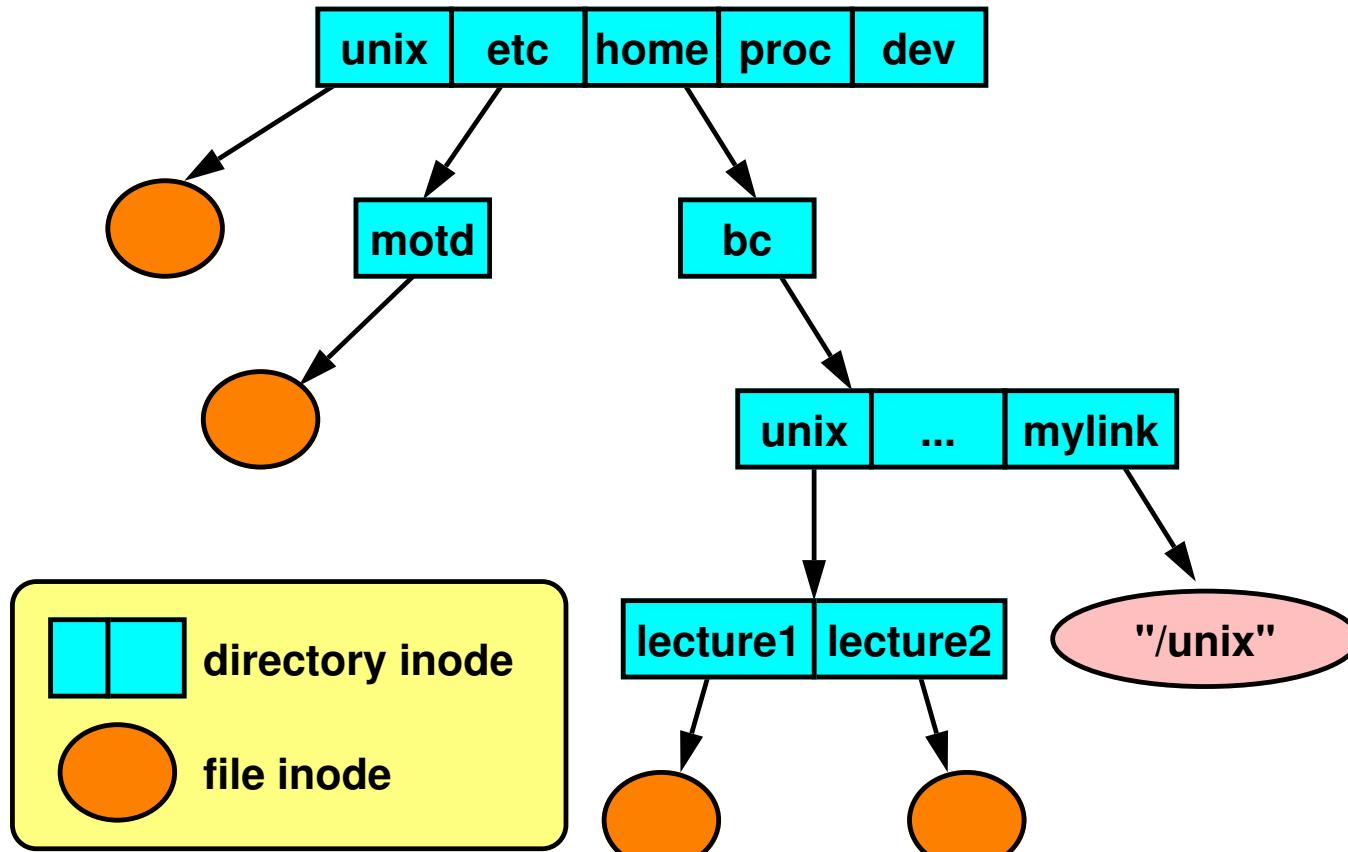


# Soft Links

```
% ln -s /unix /home/bc/mylink
```

```
% ln -s /home/bc /etc/bc
```

- create "bc" in "/etc" to **soft-link** to "/home/bc"

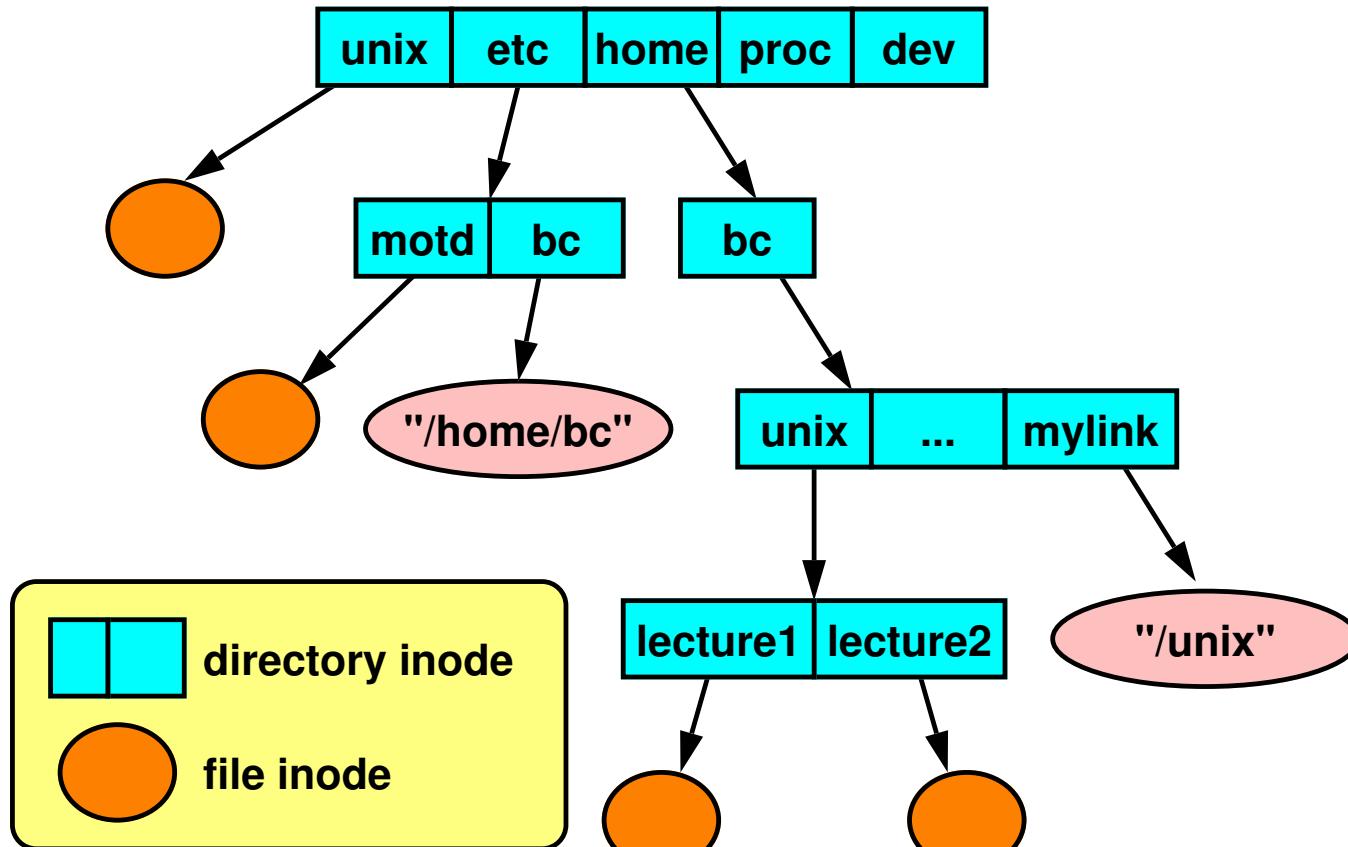


# Soft Links

```
% ln -s /unix /home/bc/mylink
```

```
% ln -s /home/bc /etc/bc
```

- create "bc" in "/etc" to **soft-link** to "/home/bc"

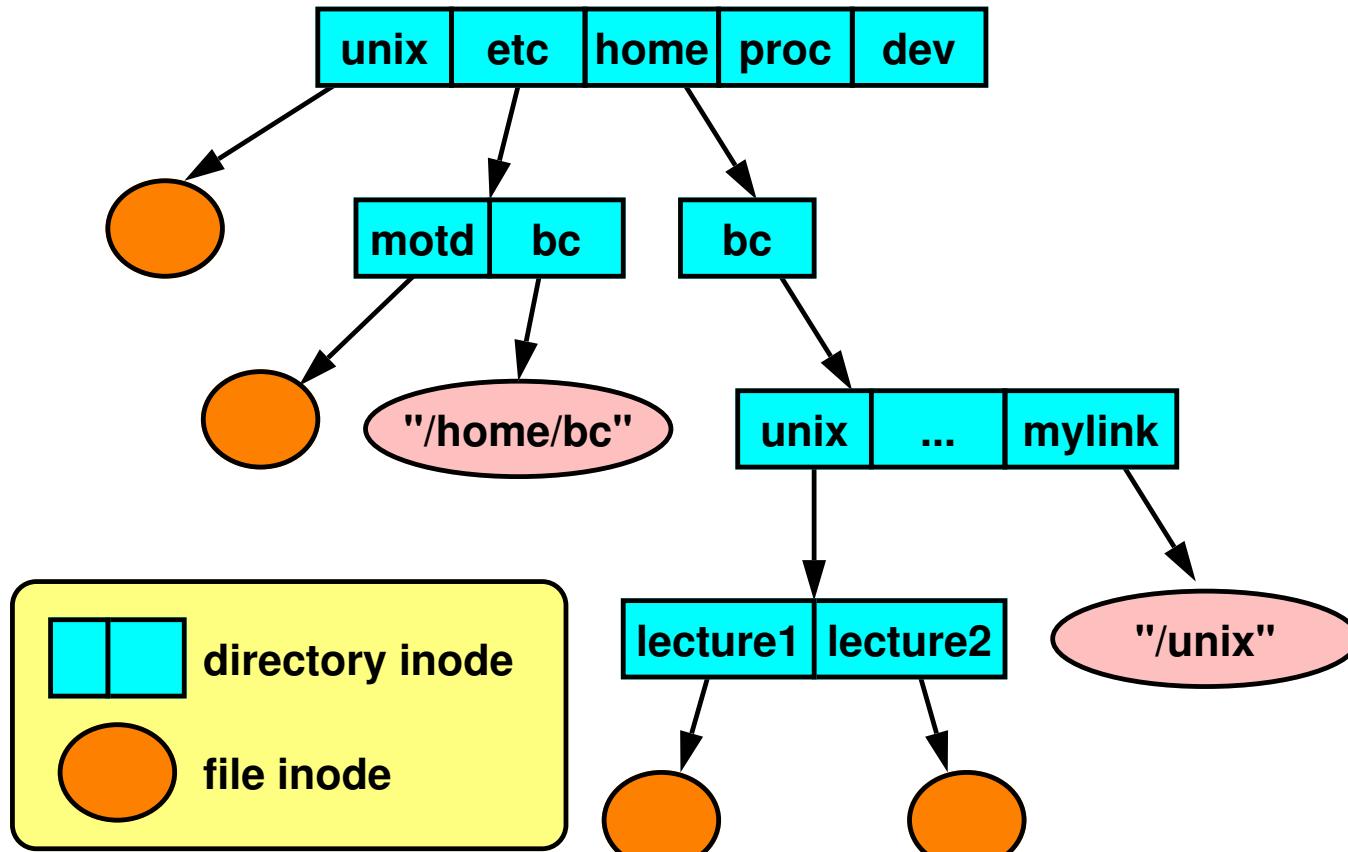


# Soft Links

```
% ls -l /etc/bc/unix/lecture1
```

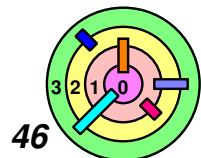
— same as "ls -l /home/bc/unix/lecture1", or is it?

- yes for the "root" account, may be no for the "bc" account
- ◊ see "access protection" later



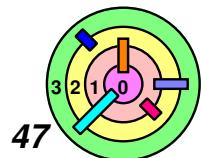
# Working Directory

- ➔ When you type "ls" in a Terminal, what directory content are you listing?
  - how does the shell know what directory content to list?
- ➔ **Working Directory:** maintained *in kernel for each process*
  - paths not starting from "/" start with the working directory
  - get by using the `getcwd()` system call
  - set by using the `chdir()` system call
    - that's what the "cd" shell command does (clearly, "cd" cannot be a program)
  - displayed (via shell) using "pwd"



# Kernel 2

- ➡ Now you have everything you need to complete kernel 2
  - if you are not familiar with a function, look at Linux man page
    - keep in mind that weenix is not Linux
    - your goal is to pass all the tests in the grading guidelines
- ➡ New things in kernel 2
  - *reference counting* (not as easy as it sounds)
    - if you have a reference counting bug, will get a kernel panic
  - *C++ polymorphism implemented in C*
    - the VFS layer is AFS-independent
    - can the VFS layer read data from disk?
      - ◊ no way
      - ◊ it needs to ask AFS to do it because only AFS knows what data structure is used on disk
      - ◊ need to invoke AFS in a FS-independent way
    - read the `ramfs` code
  - read *kernel 2 FAQ*



# Access Protection

- OS needs to make sure that only *authorized processes* are allowed access to system resources
  - various ways to provide this
- Unix (and many other systems, such as Windows) associates with files some indication of which *security principals* (i.e., the "who") are allowed access
  - along with what sort of access is allowed (i.e., the "what")
- A *security principal* is normally *a user* or *a group of users*
  - a file typically contains two pieces of information
    - which *user* owns the file (*uid*)
    - which *group* owns the file (*gid*)
  - each running process can have several security principals associated with it
    - for Sixth-Edition Unix, one user ID and one group ID
    - a process in some other OSes can have more than one group IDs



# Access Protection

```
% ls -l z  
-rw-r--r-- 1 bill adm 593 Dec 17 13:34 z
```

→ Each file has associated with it a set of access permissions

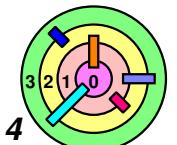
- there are 3 classes of security principals:

- *user*: owner of the file
  - *group*: group owner of the file
  - *others*: everyone else

- for each of the 3 classes of principals, specify what sorts of operations on the file are allowed

- the operations are grouped into 3 classes:

- *read*: can read a file or directory
  - *write*: can write a file or directory
  - *execute*: one must have *execute permission* for a *directory* in order to *follow a path through it*



# Access Protection

## → *Rules for checking permissions*

- 1) determines the *smallest class of principals the requester belongs to* ("user" being smallest and "others" being largest)
- 2) then it checks for appropriate permissions within that class

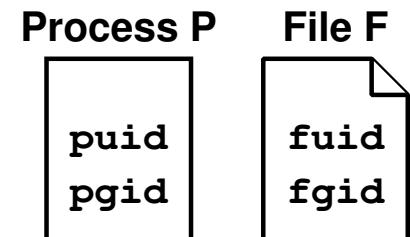
## → Can process P access file F?

- in (1), we need to determine *which class* is P
  - puid: process uid       fuid: file uid
  - pgid: process gid       fgid: file gid

```

if (puid == fuid) {
    /* requester is "user/owner" */
} else if (pgid == fgid) {
    /* requester is "group-owner" */
} else {
    /* requester is "others" */
}

```



# Permissions Example

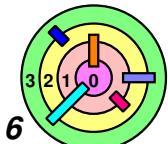
```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

→ Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not

1) Q: May **andy** list the contents of directory **A**?



# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

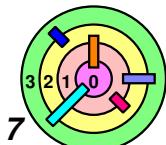
./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

→ Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not

1) Q: May **andy** list the contents of directory **A**?

A: No



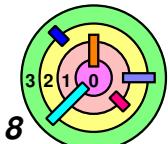
# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

- Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not
- 2) Q: May `andy` read `A/x`?



# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm    1024 Dec 17 13:34 A
drwxr-----  2 bill      adm    1024 Dec 17 13:34 B

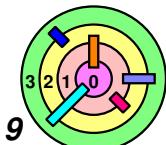
./A:
total 1
-rw-rw-rw-  1 bill      adm    593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 bill      adm    446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm    446 Dec 17 13:45 y
```

→ Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not

2) Q: May `andy` read `A/x`?

A: Yes



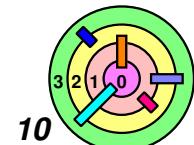
# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

- Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not
- 3) Q: May **trina** list the contents of directory **B**?



# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

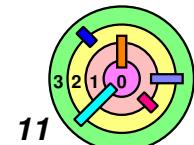
./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

→ Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not

3) Q: May **trina** list the contents of directory **B**?

A: Yes



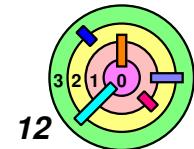
# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

- Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not
- 4) Q: May **trina** modify **B/y**?



# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

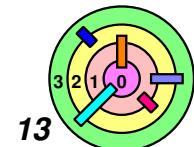
./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

→ Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not

4) Q: May **trina** modify **B/y**?

A: No



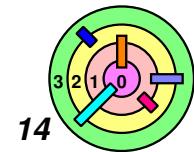
# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

- Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not
- 5) Q: May **bill** modify **B/x**?



# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm    1024 Dec 17 13:34 A
drwxr-----  2 bill      adm    1024 Dec 17 13:34 B

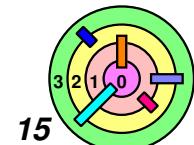
./A:
total 1
-rw-rw-rw-  1 bill      adm    593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm    446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm    446 Dec 17 13:45 y
```

→ Suppose that `bill` and `trina` are members of the `adm` group and `andy` is not

5) Q: May `bill` modify `B/x`?

A: No



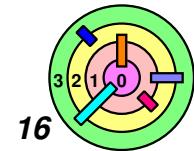
# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw-----rw-  1 trina    adm        446 Dec 17 13:45 y
```

- Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not
- 6) Q: May **bill** read **B/y**?



# Permissions Example

```
% ls -lR
.:
total 2
drwxr-x--x  2 bill      adm        1024 Dec 17 13:34 A
drwxr-----  2 bill      adm        1024 Dec 17 13:34 B

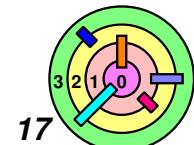
./A:
total 1
-rw-rw-rw-  1 bill      adm        593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-   1 bill      adm        446 Dec 17 13:34 x
-rw----rw-   1 trina    adm        446 Dec 17 13:45 y
```

→ Suppose that **bill** and **trina** are members of the **adm** group and **andy** is not

6) Q: May **bill** read **B/y**?

A: No



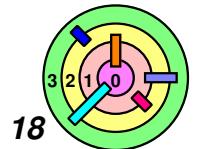
# Open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int options [, mode_t mode])
```



## options

- **O\_RDONLY** open for reading only
- **O\_WRONLY** open for writing only
- **O\_RDWR** open for reading and writing
- **O\_APPEND** set the file offset to end of file prior to each write
- **O\_CREAT** if the file does not exist, then create it, setting its mode to mode adjusted by user mask (umask)
- **O\_EXCL**: if **O\_EXCL** and **O\_CREAT** are set, then open fails if the file exists
- **O\_TRUNC** delete any previous contents of the file
- **O\_NONBLOCK** don't wait if I/O cannot be done immediately
- some options are not compatible with other options

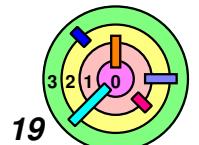


# Setting File Permissions

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode)
```

- sets the file permissions of the given file to those specified in mode
- only the owner of a file and the superuser may change its permissions
- nine combinable possibilities for mode (read/write/execute for user, group, and others)
  - S\_IRUSR (0400), S\_IWUSR (0200), S\_IXUSR (0100)
  - S\_IRGRP (040), S\_IWGRP (020), S\_IXGRP (010)
  - S\_IROTH (04), S\_IWOTH (02), S\_IXOTH (01)
    - note: numeric prefix of 0 means the number is in *octal* format

```
% chmod 0640 z
% ls -l z
-rw-r----- 1 bill    adm      593 Dec 17 13:34 z
```



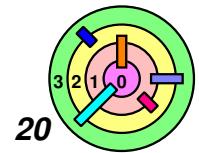
# Creating a File

→ Use either `open` or `creat`

- `open(const char *pathname, int flags, mode_t mode)`
  - flags must include `O_CREAT` to create a file
- `creat(const char *pathname, mode_t mode)`
- `open` is preferred

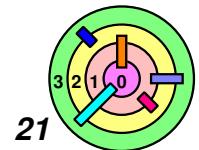
→ The `mode` parameter helps specify the permissions of the newly created file

- `permissions = mode & ~umask`



# Umask

- Standard programs create files with "maximum needed permissions" as *mode*
  - compilers: 0777
  - editors: 0666
- Per-process parameter, *umask*, used to *turn off* undesired permission bits
  - e.g., turn off all permissions for others, write permission for group: set umask to 027
  - compilers: permissions = 0777 & ~ (027) = 0750
  - editors: permissions = 0666 & ~ (027) = 0640
  - set with `umask()` system call or (usually) `umask` shell command



# Midterm Exam Coverage



- Midterm exam covers everything from the beginning of the semester to this slide**
  - Ch 1 through Ch 4 only**
    - Ch 5 materials are excluded from the midterm
  - final exam coverage will not overlap midterm coverage**
    - since the topics covered by the final exam is not independent of the midterm coverage, we say the final exam "focuses" on Ch 5 plus everything beyond this slide

