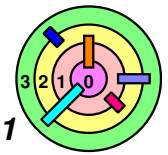
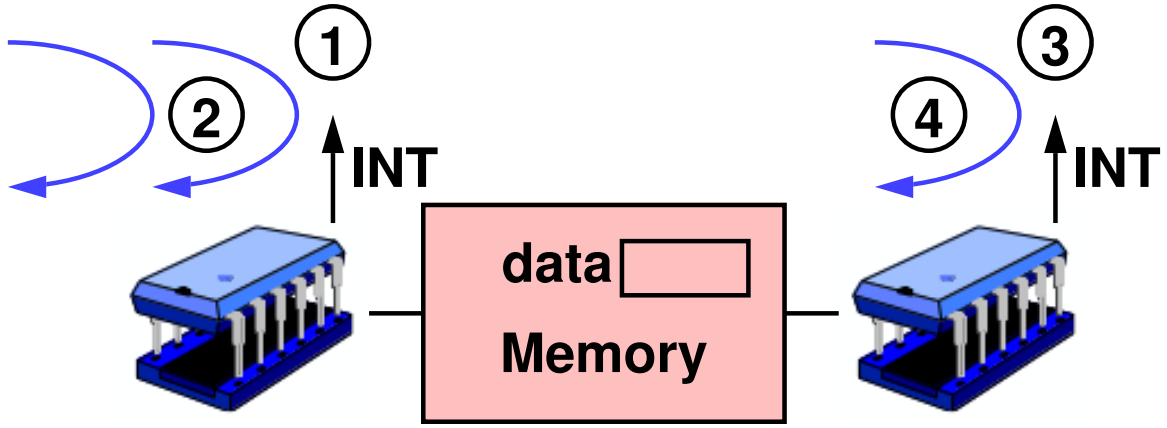
Housekeeping (Lecture 14 - 10/14/2013)

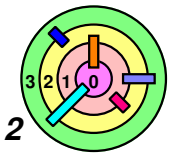
- Kernel #1 due at 11:45pm on Friday, 10/25/2013
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
- *Grading guidelines* for kernel #1 has been posted
  - this is the only way we will grade
  - read the test code there!
  - additional test code can be downloaded from the spec
- Post questions about the kernel #1 to *class Google Group*
  - *extra credit* for posting good responses
- The plan is to have *midterm exam* cover *Ch 1 through Ch 5*
  - materials that correspond to skipped lecture slides will not be part of the midterm coverage
  - lecture slides with a grey X in the lower left corner will not be part of the midterm coverage



Preemptive Kernels

- What's different?
- A thread accesses a shared data structure:
  1) it might be *interrupted* by an interrupt handler (running on its processor) that accesses the same data structure
  2) it might be forced to *give up* its processor to another thread, either because its time slice has expired or it has been preempted by a higher-priority thread
  3) an *interrupt handler* running on *another processor* might access the same data structure
  4) *another thread* running on another processor might access the same data structure





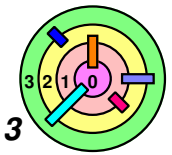
Solution?

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

- Does it work?
  - no, can deadlock in `AccessXInterrupt()` in case (1)



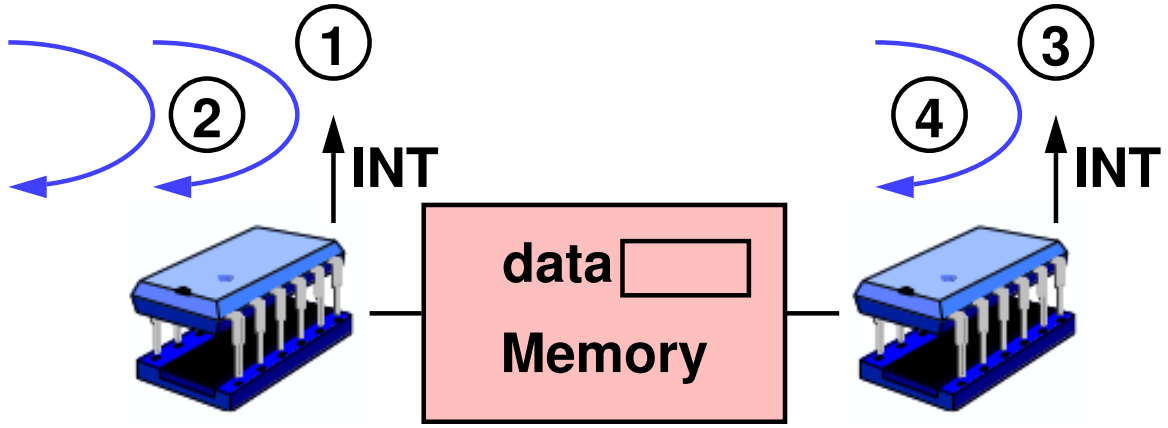
Solution ...

```
int x = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  DisablePreemption();
  MaskInterrupts();
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  UnMaskInterrupts();
  EnablePreemption();
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

Does it work?



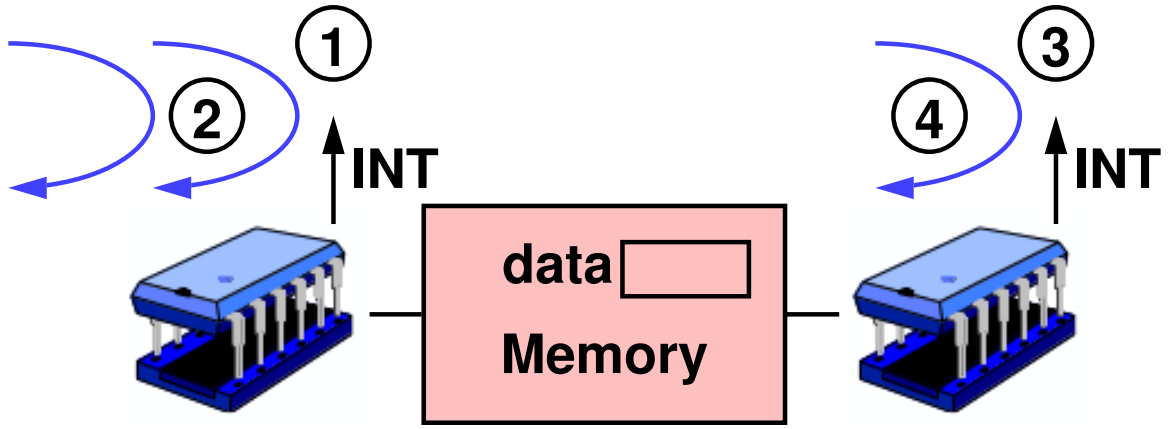
Solution ...

```
int x = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
   DisablePreemption();
   MaskInterrupts();
   SpinLock(&L);
   X = X+1;
   SpinUnlock(&L);
   UnMaskInterrupts();
   EnablePreemption();
}
```

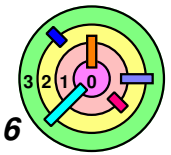
```
void AccessXInterrupt() {
   ...
   SpinLock(&L);
   X = X+1;
   SpinUnlock(&L);
   ...
}
```

- Does it work?
  - yes



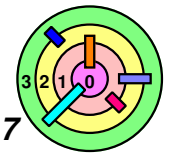
Interrupt Threads?

- Solaris allows interrupts to be handled as threads
- Does it make sense to handle interrupts with threads?
  - perhaps similar to using `sigwait` for handling signals with threads
  - what would be the advantages?
  - what would be the disadvantages?

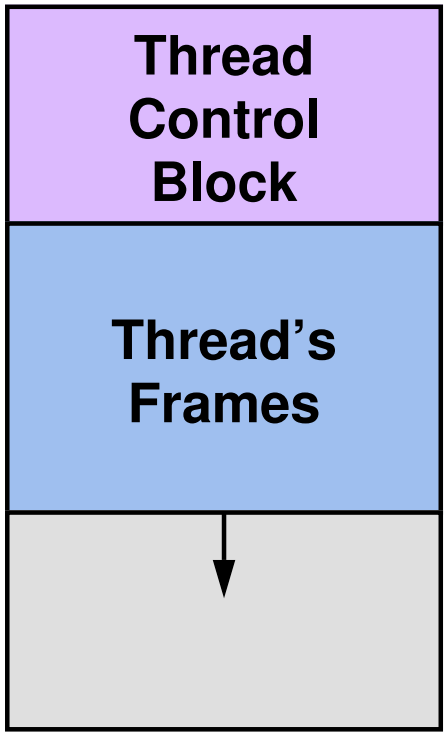


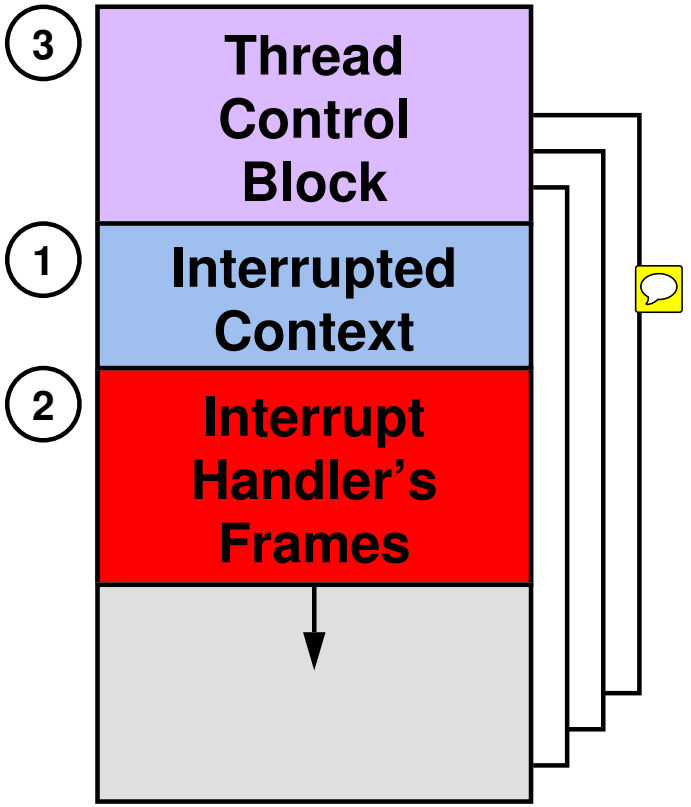
Interrupt Threads

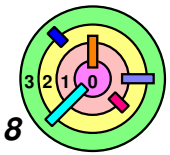
```
void InterruptHandler( ) {
  // deal with interrupt
  ...
  if (!MoreWork)
    return;
  else
    BecomeThread( );
  ...
  P(Semaphore); // sleep!
  ...
}
```



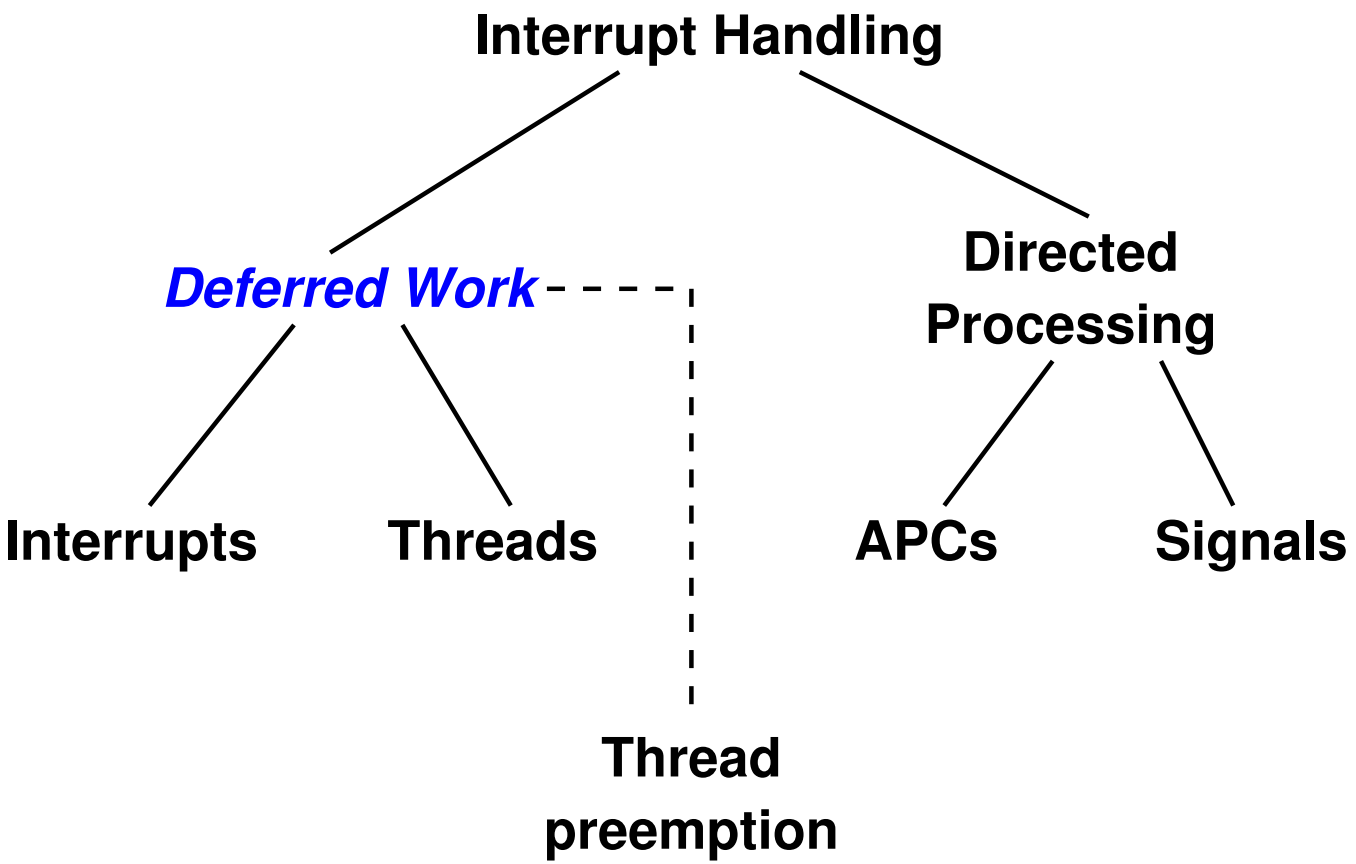
Interrupt Threads In Action

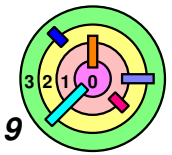






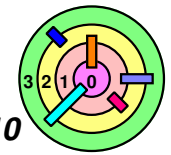
Interrupt Handling - Overview





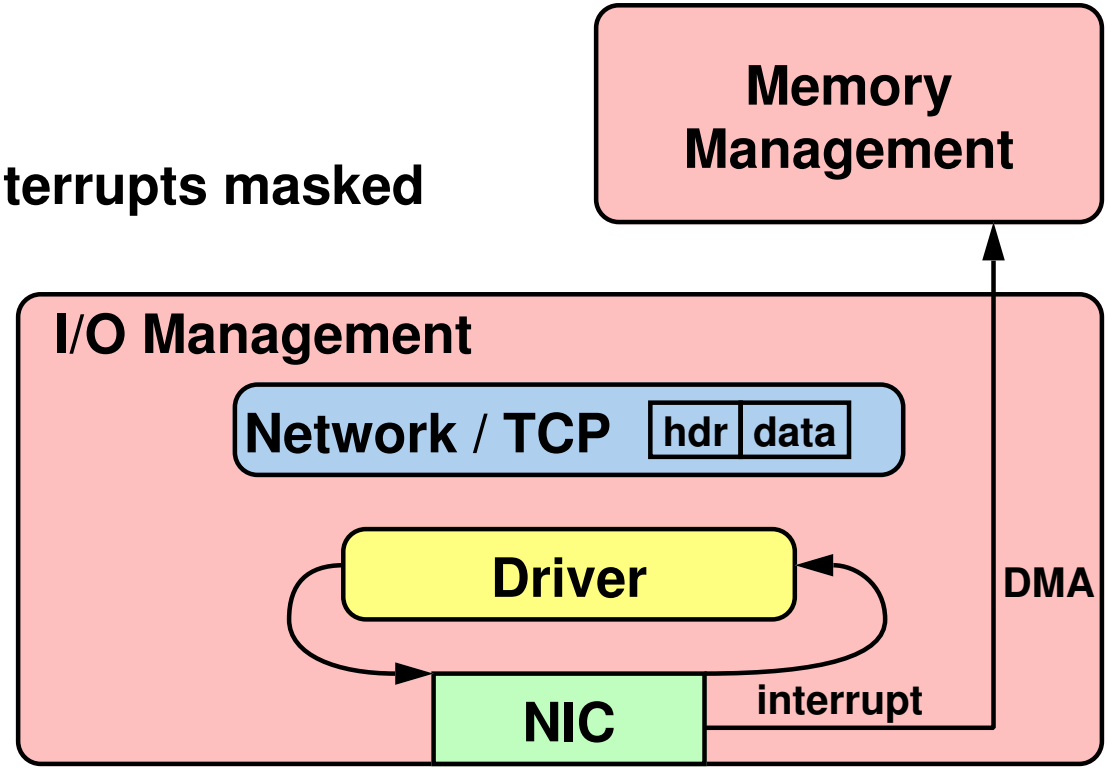
Deferred Work

- Interrupt handlers run with interrupts masked (up to its interrupt priority level)
  - both when executed in interrupt context or thread context
  - may interfere with handling of other interrupts
  - they must run to completion (but may be interrupted by a higher priority interrupt)
    - it must *complete quickly*
- What to do if an interrupt handler has a lot of work to be done?
  - only do what you must do inside the interrupt handler
  - *defer* most of the work to be done after the interrupt handler returns



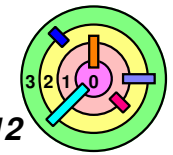
Deferred Work

- Ex: network packet processing
  - TCP header processing can take a long time
    - not suitable to do them in a interrupt handler
- Solution
  - do minimal work now
  - do rest later without interrupts masked
- How?



Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void NetworkInterruptHandler() {
  // deal with interrupt, do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```

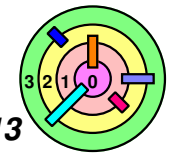


Windows Interrupt Priority Levels

hardware	31	High
	30	Power fail
	29	Inter-processor
	28	Clock
	.	.
	.	.
	.	.
	4	Device 2
	3	Device 1
software	2	*DPC*
	1	APC
	0	Thread

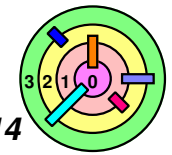
Windows handles deferred work in a special interrupt context

- **DPC (deferred procedure call) is a *software interrupt***



Deferred Procedure Calls

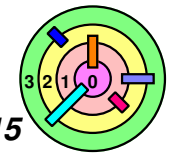
```
void InterruptHandler( ) {
  // deal with interrupt
  ...
  QueueDPC(MoreWork);
  /* enqueues MoreWork on
     the DPC queue and
     requests a DPC
     interrupt
  */
}

void DPCHandler( ... ) {
  while(!Empty(DPCQueue)) {
    Work = DeQueue(DPCQueue);
    Work();
  }
}
```



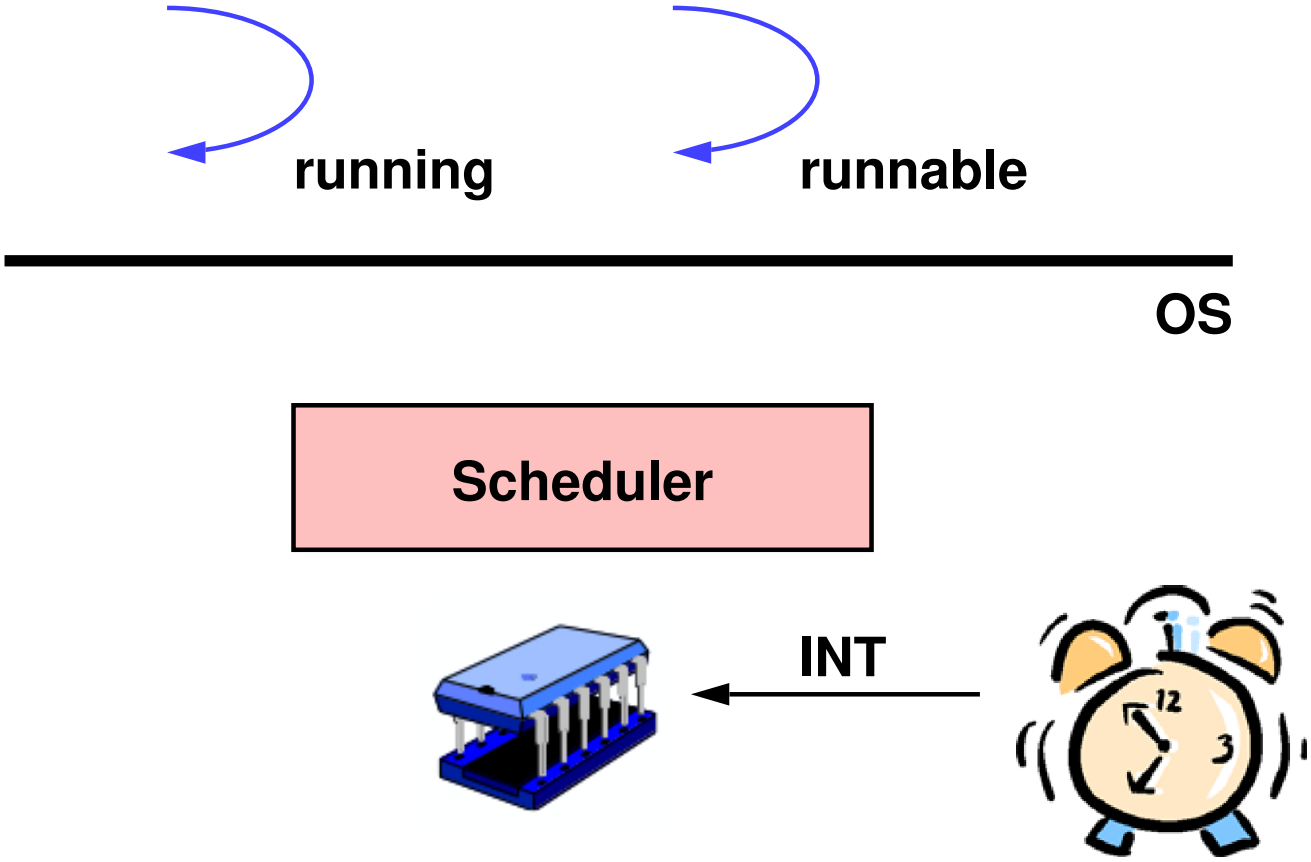
Software Interrupt Threads

- **Linux handles deferred work in a special kernel thread**
  - this kernel thread is scheduled like any other kernel thread

```
void InterruptHandler( ) {
  // deal with interrupt
  ...
  EnQueue(WorkQueue, MoreWork);
  SetEvent(Work);
}

void SoftwareInterruptThread() {
  while(TRUE) {
    WaitEvent(Work)
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}
```



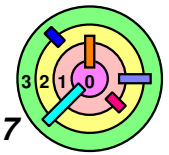
Thread Preemption



Preemption: User-Level Only

- Non-preemptive kernel
  - preempt only threads running in user mode
  - if clock-interrupt happens, just set a global flag

```
void ClockHandler( ) {
  // deal with clock
  //      interrupt
  ...
  if (TimeSliceOver())
    ShouldReschedule = 1;
}
```



Preemption: User-Level Only

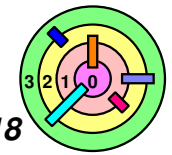
```
void TopLevelTrapHandler(...) {
  SpecificTrapHandler();
  ...
  if (ShouldReschedule) {
    /* the time slice expired
       while the thread was
       in kernel mode */
    Reschedule();
  }
}
```

`Reschedule()` puts the calling thread on the run queue

- then call `thread_switch()` to give up the processor

The work of *rescheduling* is *deferred*

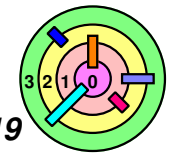
```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev]();
  if (PreviousMode == UserMode) {
    // the clock interrupted user-mode code
      if (ShouldReschedule)
        Reschedule();
  }
  ...
}
```



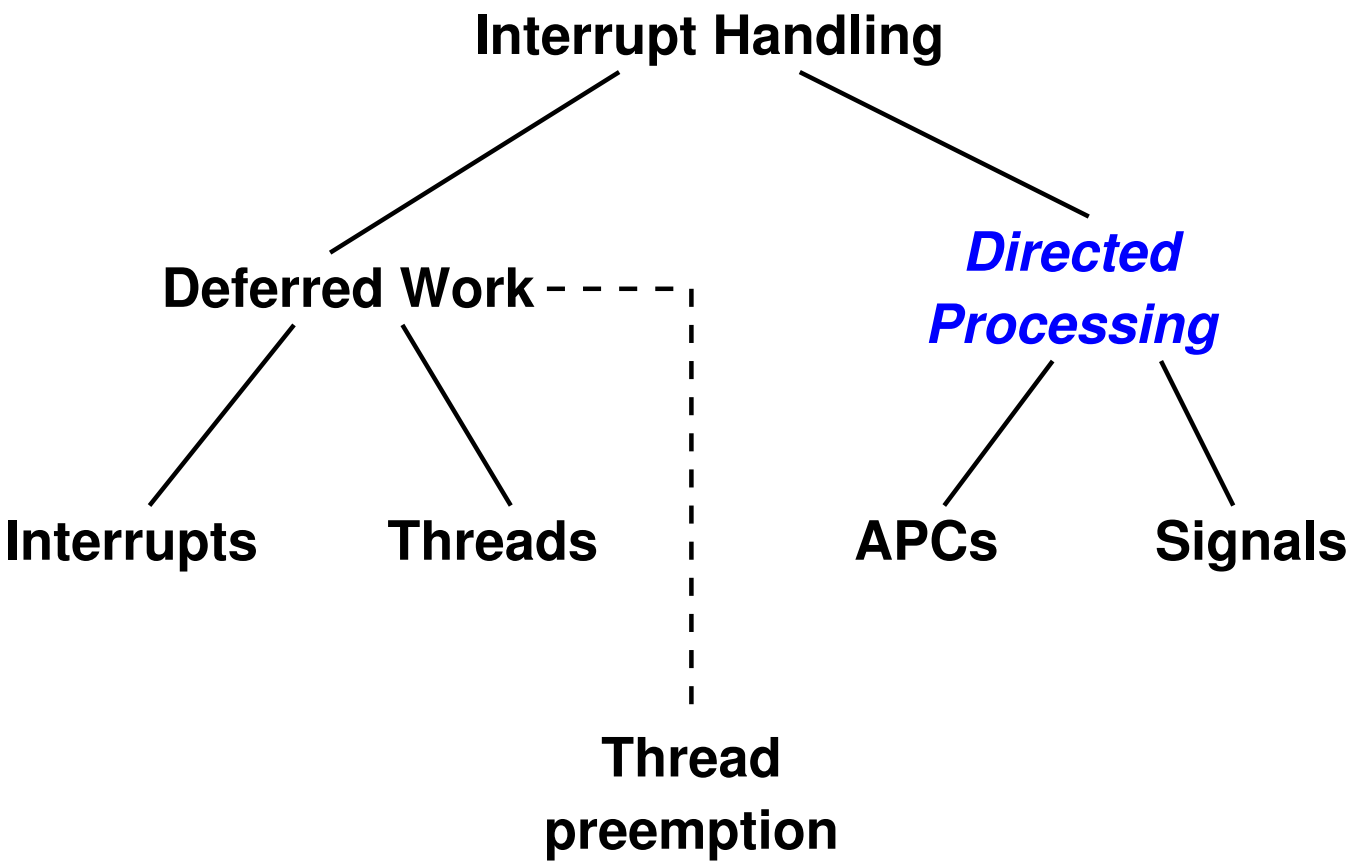
Preemption: Full (i.e., Preemptive Kernel)

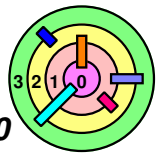
- Preemptive kernel
  - preemption can happen in the kernel mode
  - if clock-interrupt happens, setup the thread to give up the processor when the processor is about to return to the thread's context
  - how?
    - e.g., add the `Reschedule()` function to the DPC queue

```
void ClockInterruptHandler( ) {
  // deal with clock interrupt
  ...
  if (TimeSliceOver())
    QueueDPC(Reschedule);
}
```



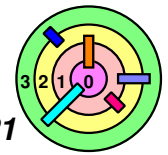
Interrupt Handling - Overview





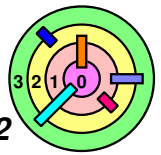
Directed Processing

- **Signals**: Unix
  - perform given action in context of a particular thread in user mode
  - e.g., seg fault
    - generated by hardware and needs to be delievered to the user process to invoke a singal handler
- **APC**: Windows asynchronous procedure calls
  - roughly same thing, but also may be done in kernel mode
    - thus, the APC mechanism is more general than Unix signals



Invoking the Signal Handler

- Basic idea is to set up the user stack so that the handler is called as a subroutine and so that when it returns, normal execution of the thread may continue
- Complications:
  - saving and restoring registers
    - must first save *all* registers and later on restore all of them
  - signal mask
    - must block the signal and later on unblock the signal
  - therefore, when the signal handler returns, it needs to return to some code that restores all the registers, unblocks the signal, then return to the interrupted code

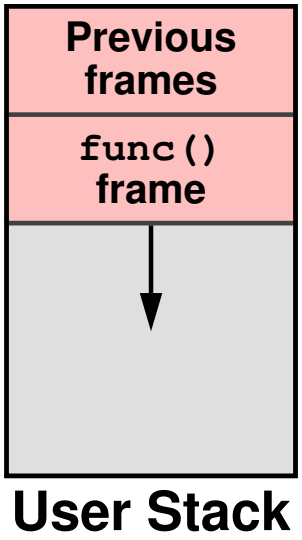


Invoking the Signal Handler (1)

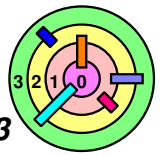
Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

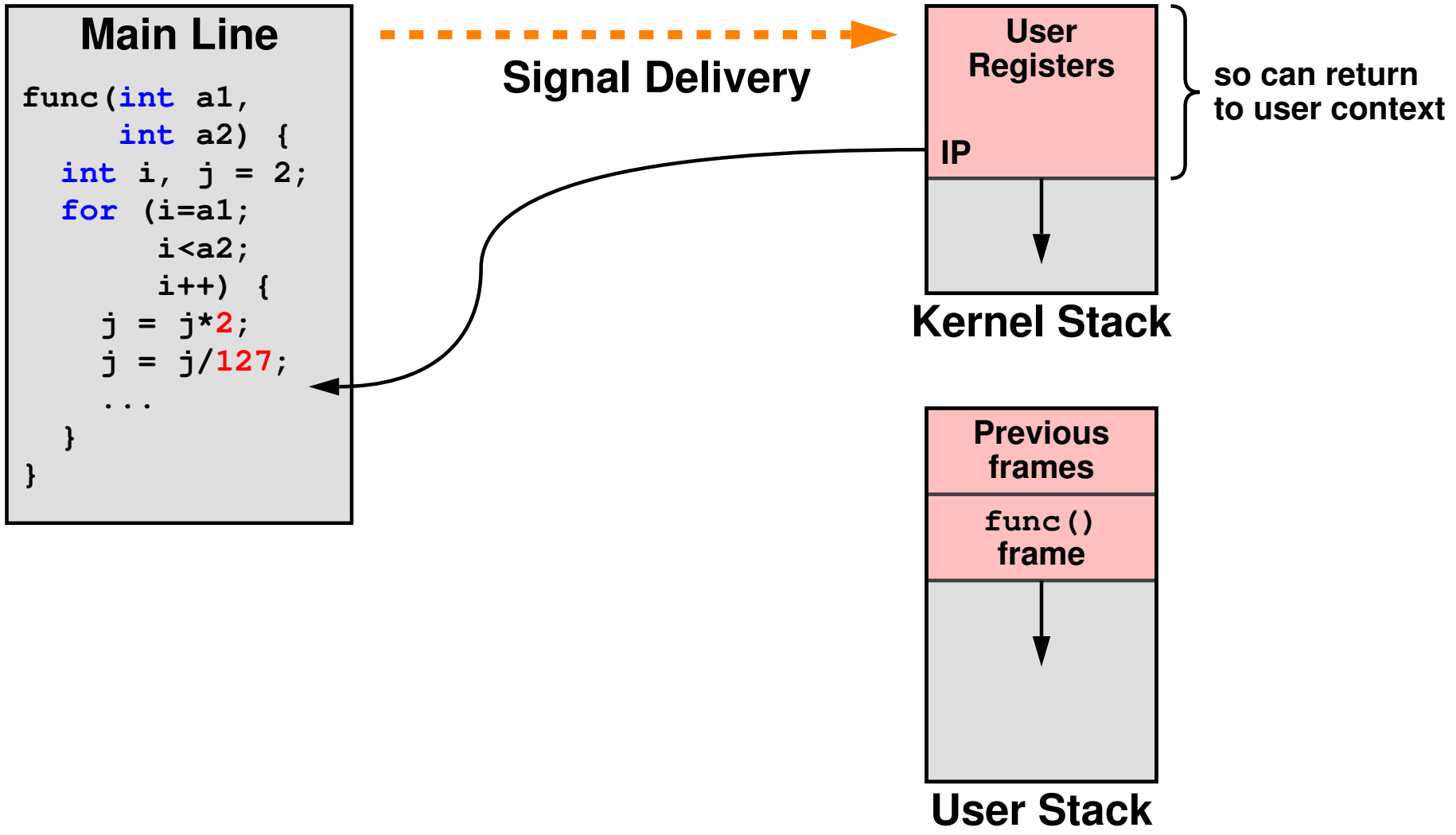
IP



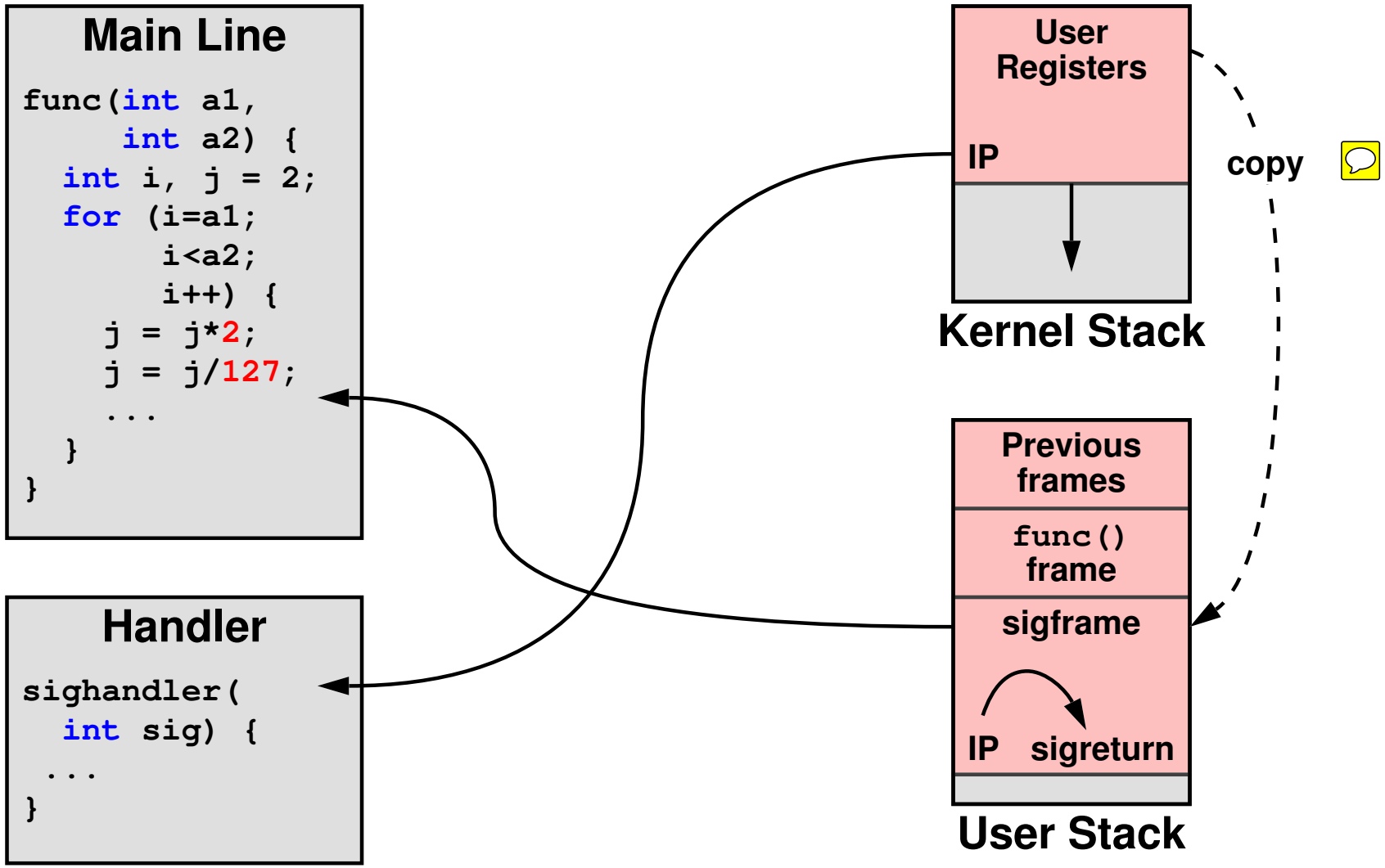
User Stack



Invoking the Signal Handler (2)

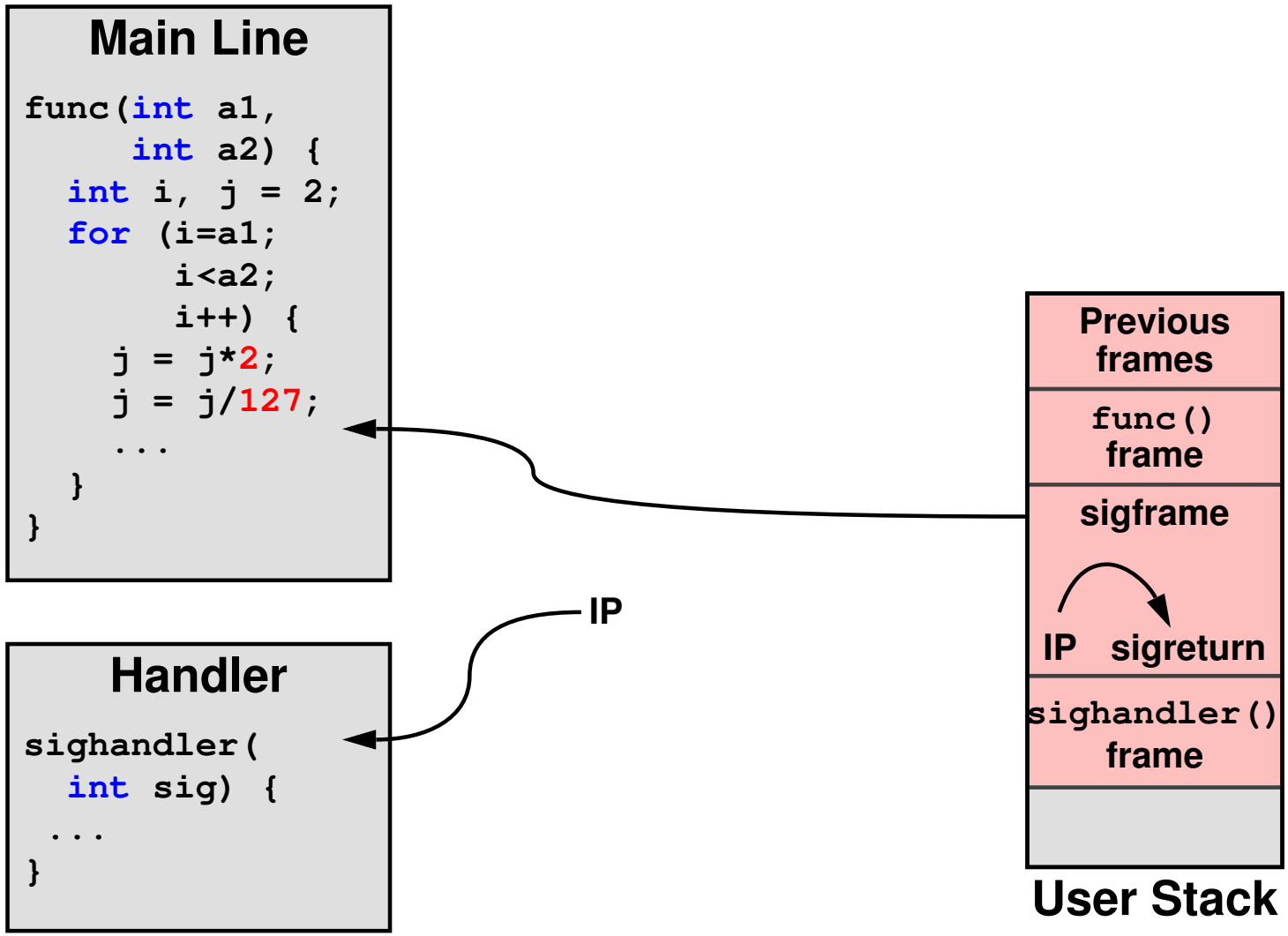


Invoking the Signal Handler (3)



Setup a stack frame to execute a `sigreturn` instruction

Invoking the Signal Handler (4)



Signal handler executed on the user stack

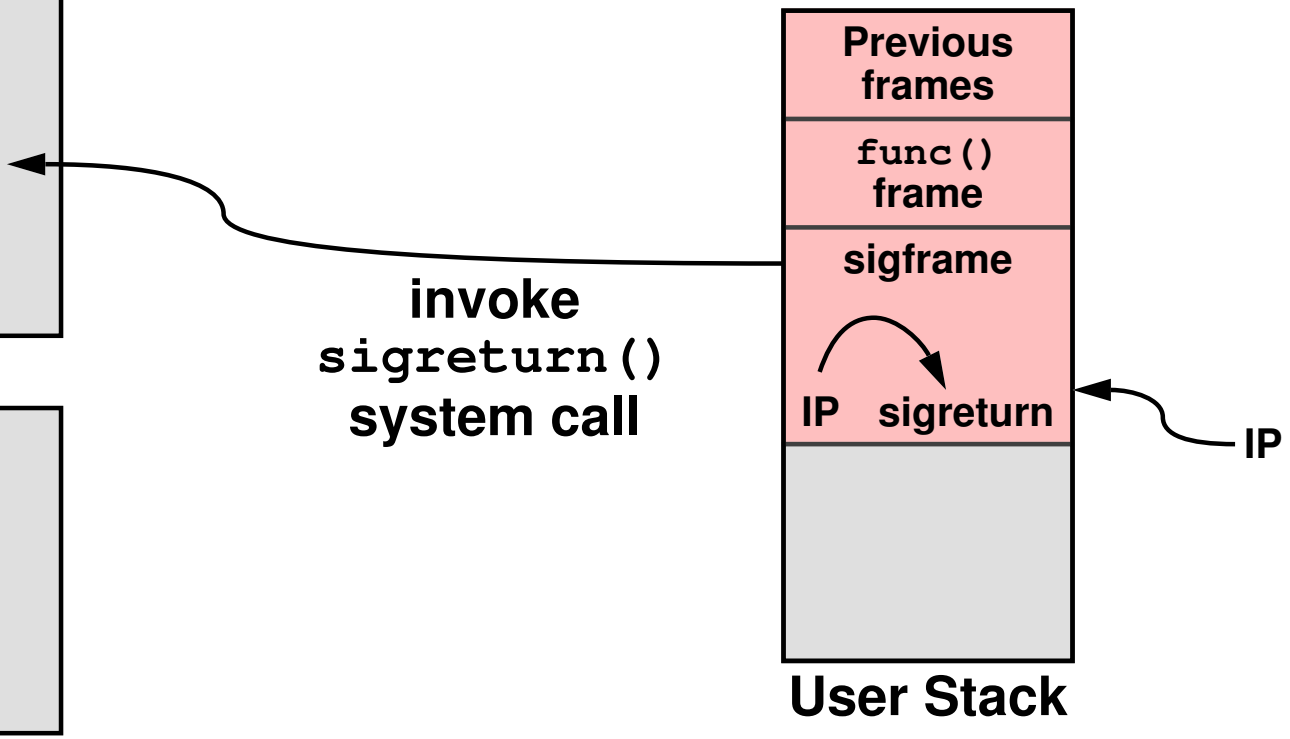
Invoking the Signal Handler (5)

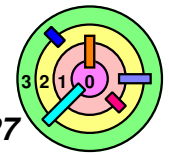
Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```





Invoking the Signal Handler (6)

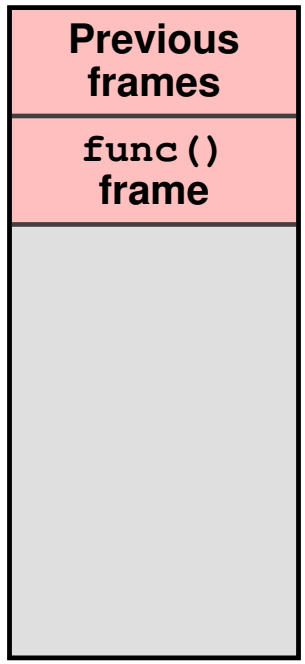
Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

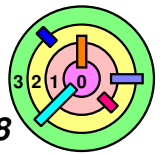
IP

Handler

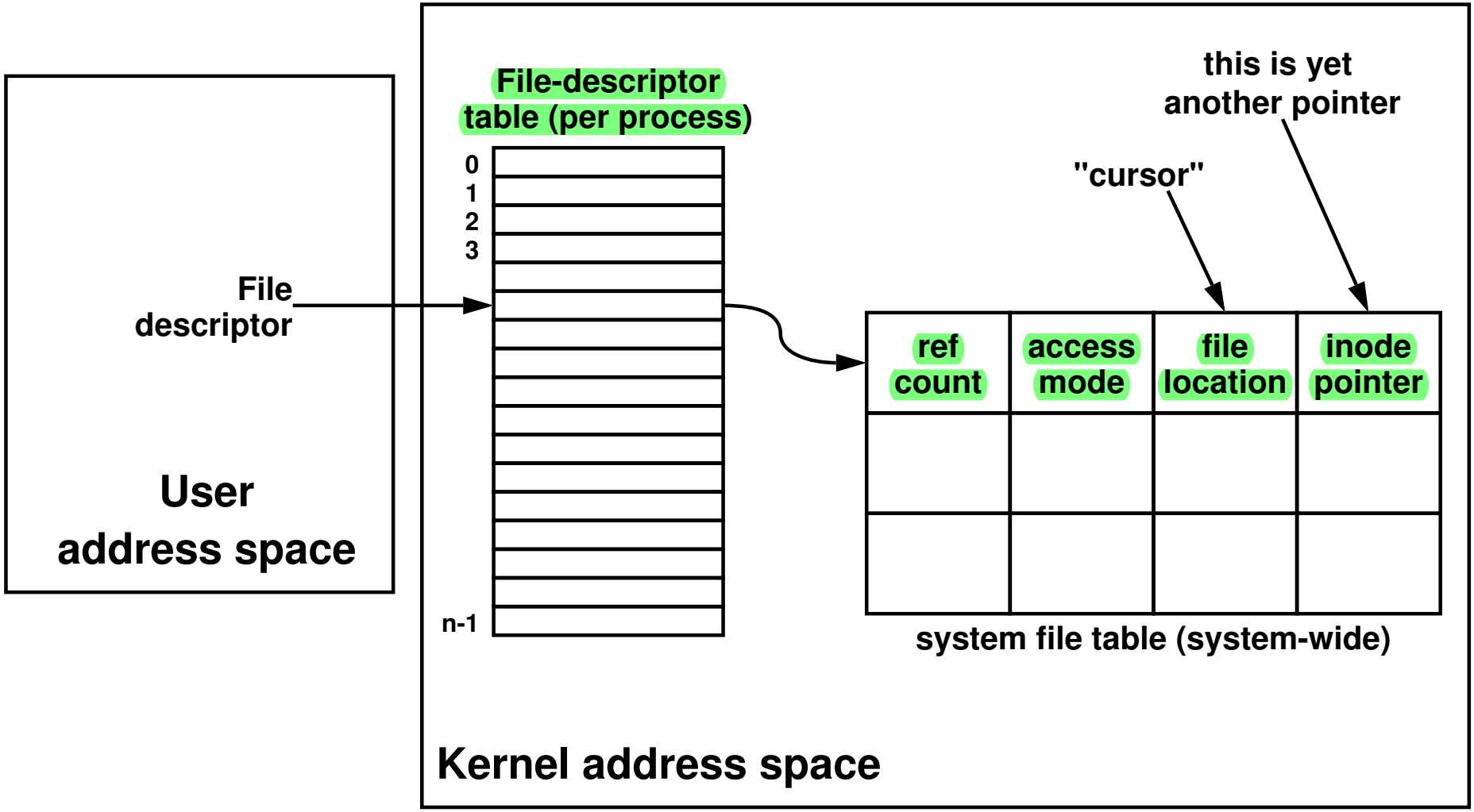
```
signalhandler(
  int sig) {
 ...
}
```



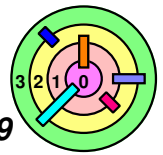
User Stack



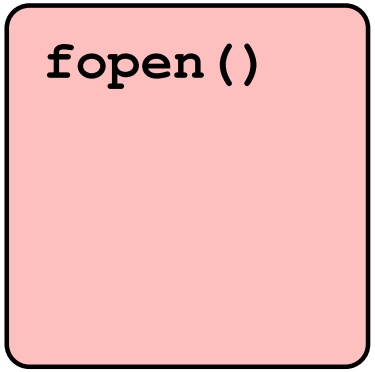
File-Descriptor Table



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



Put It All Together




```
int fd;
fd = fopen("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```

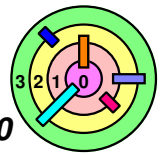
Applications

OS

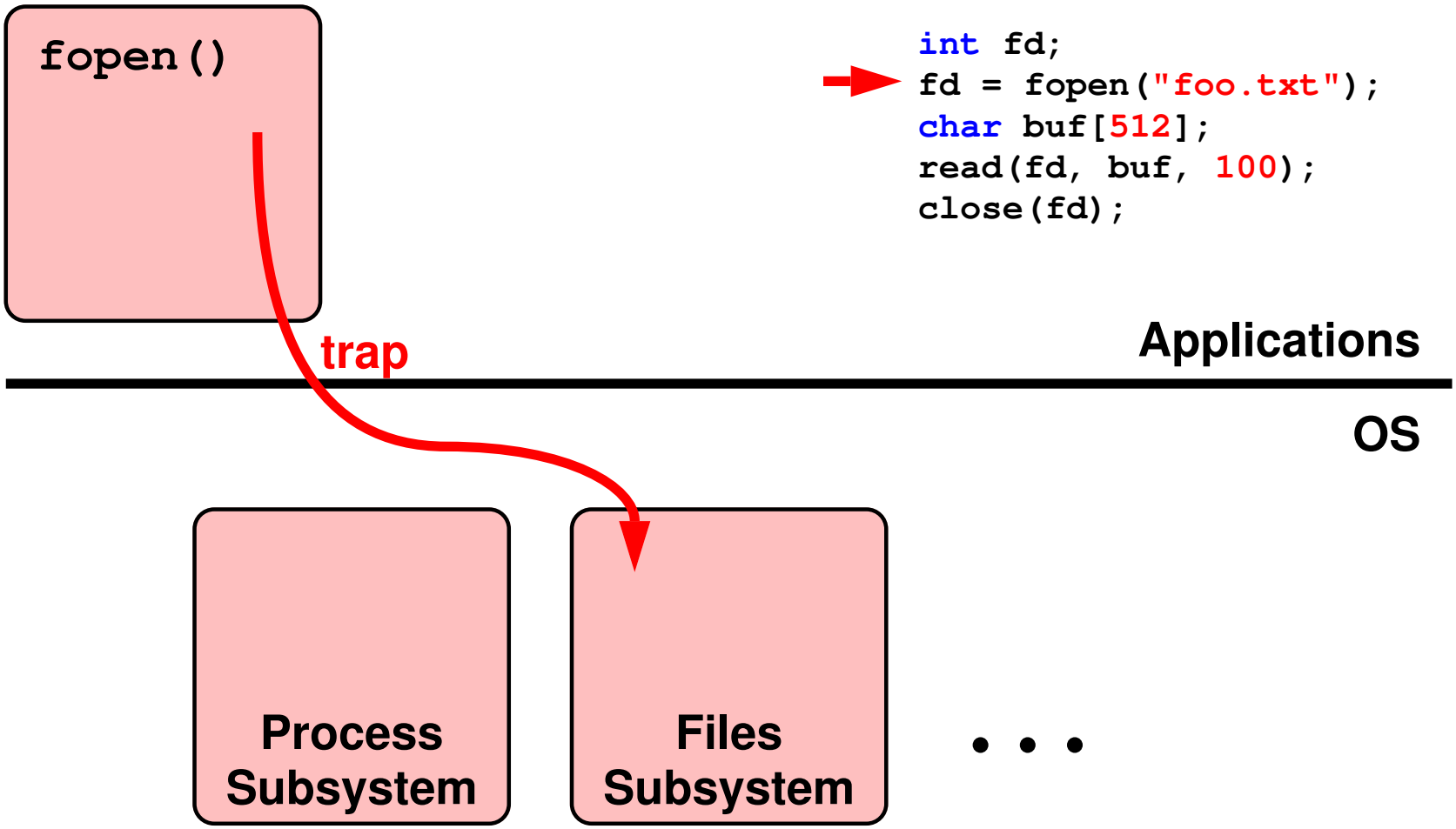
Process Subsystem

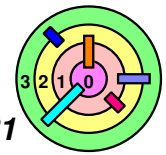
Files Subsystem

• • •

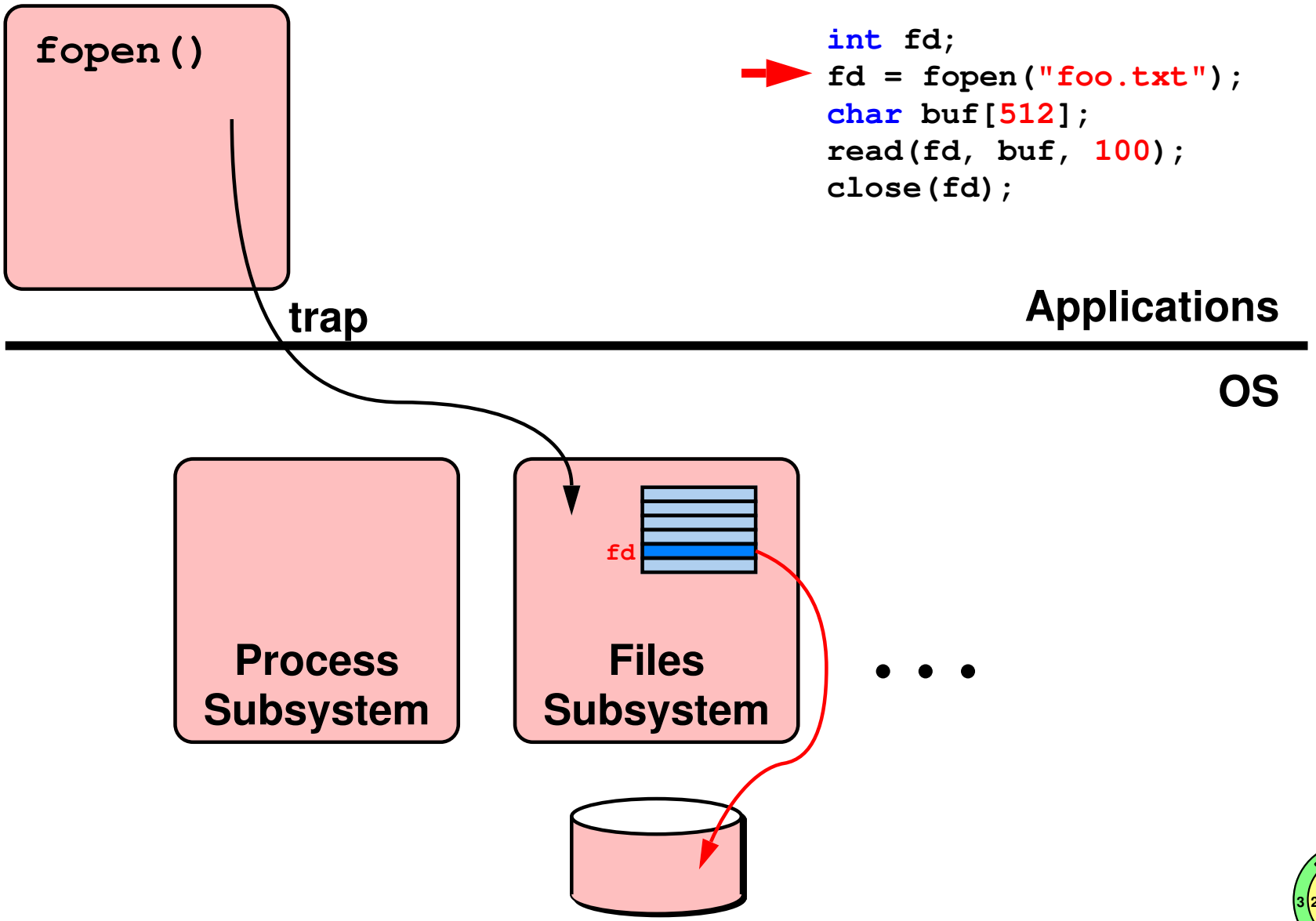


Put It All Together

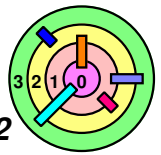




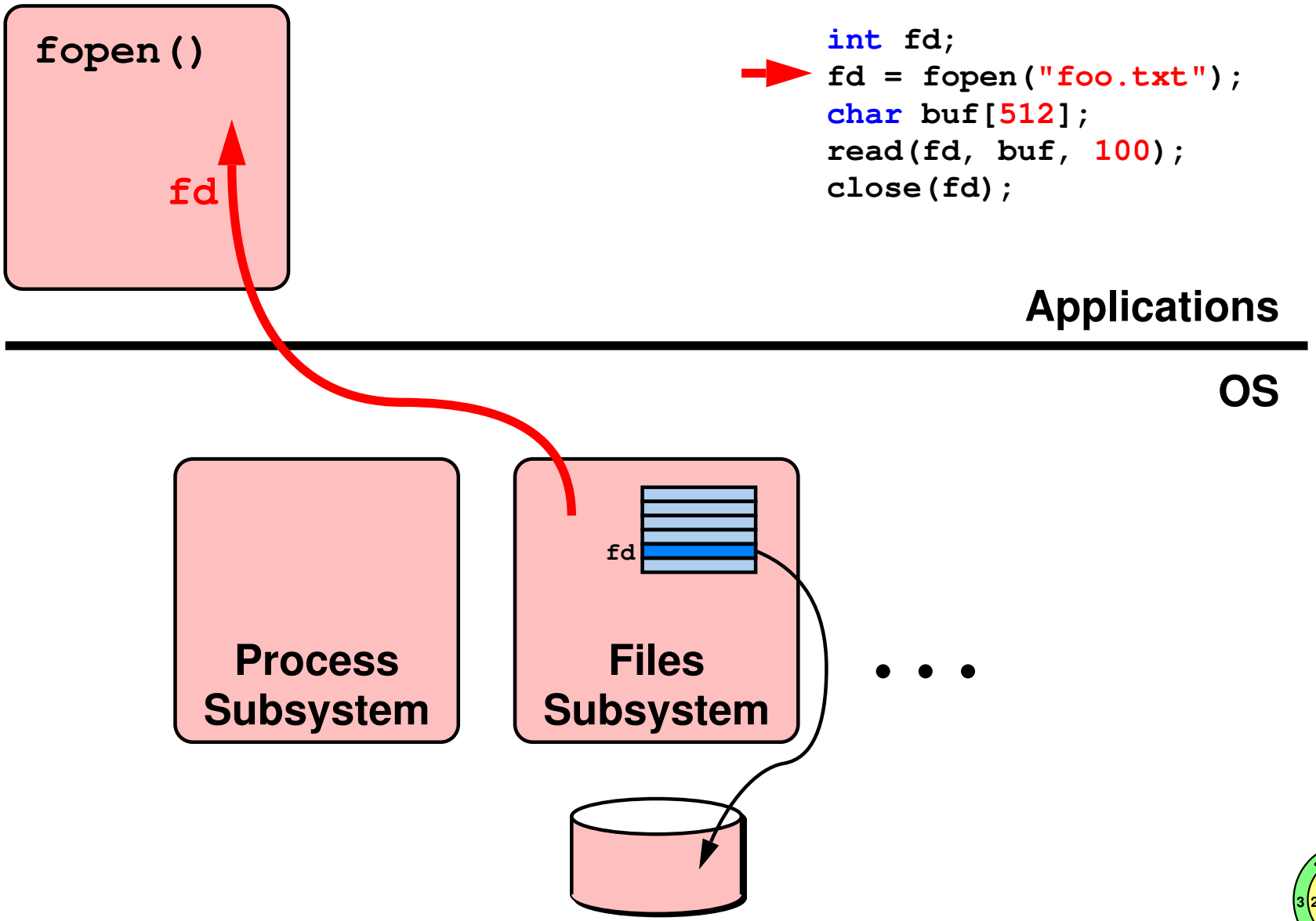
Put It All Together



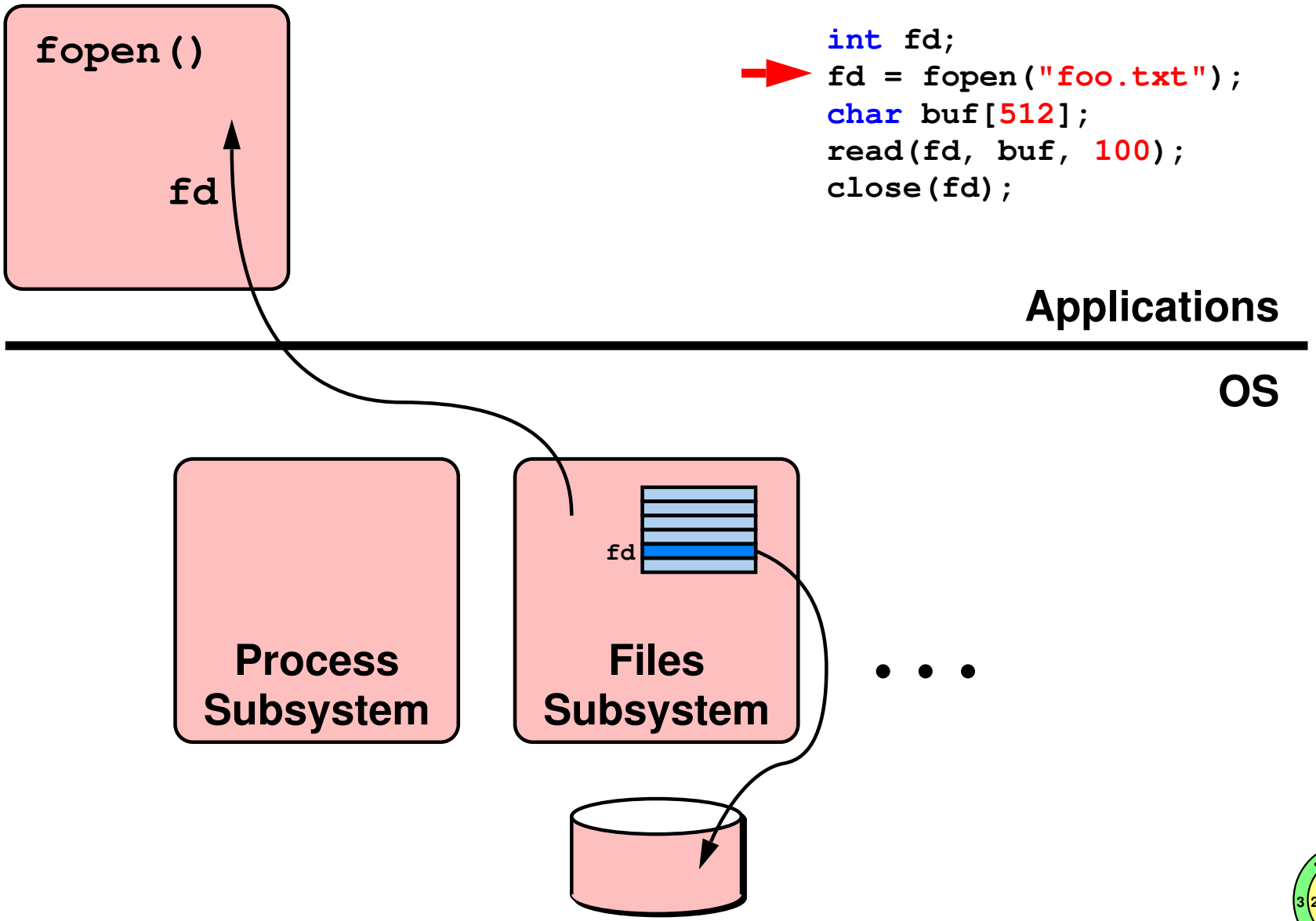

```
int fd;
fd = fopen("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```



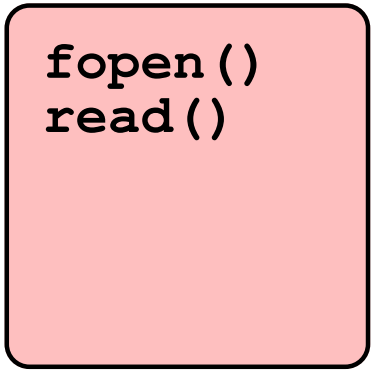
Put It All Together



Put It All Together



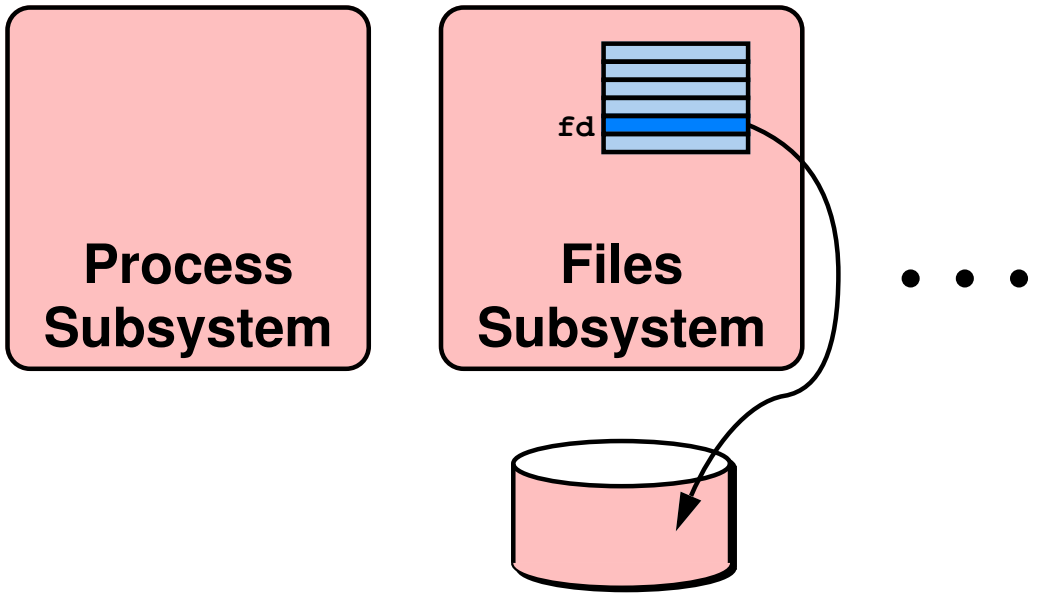
Put It All Together

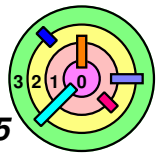



```
int fd;
fd = fopen("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```

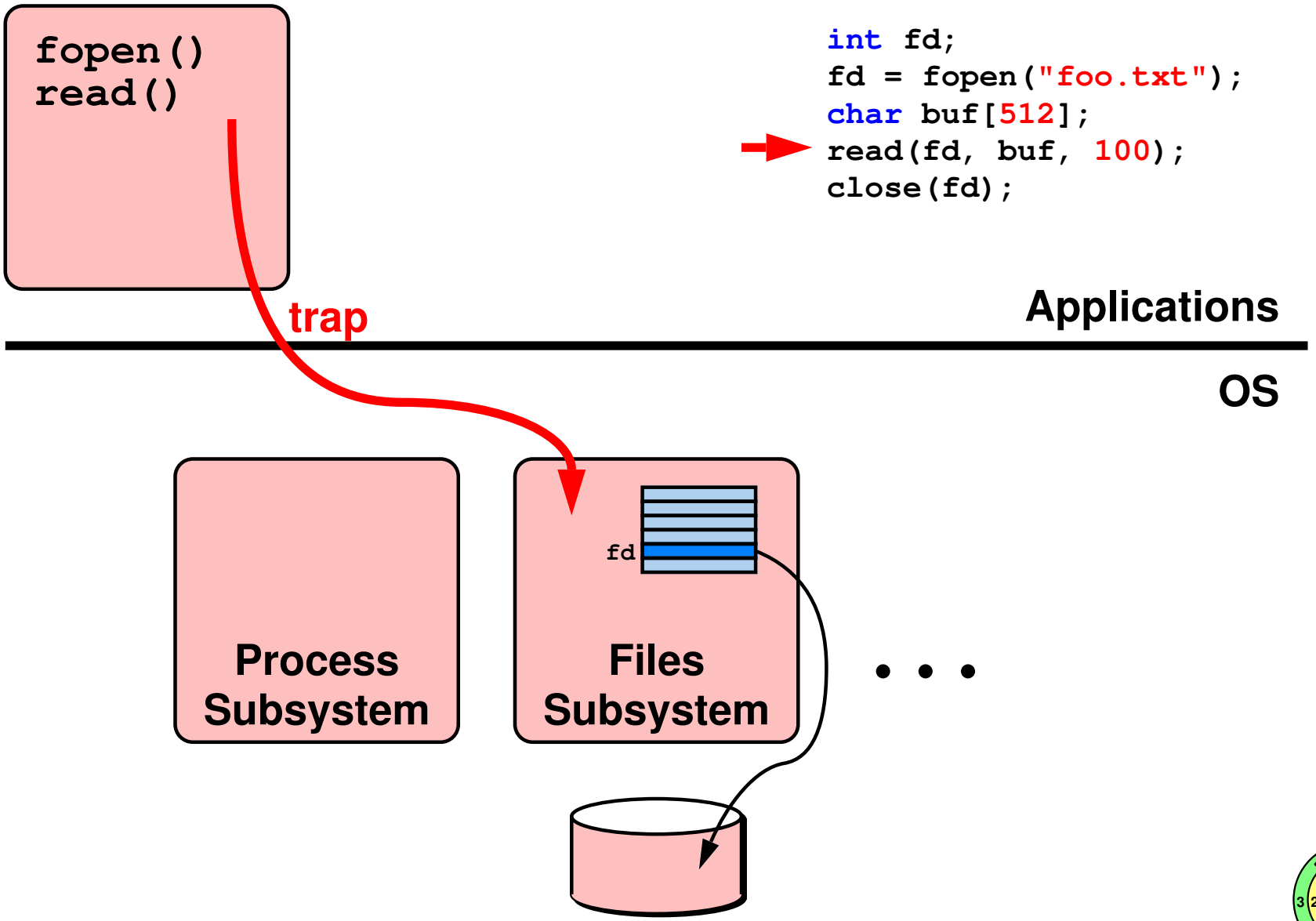
Applications

OS

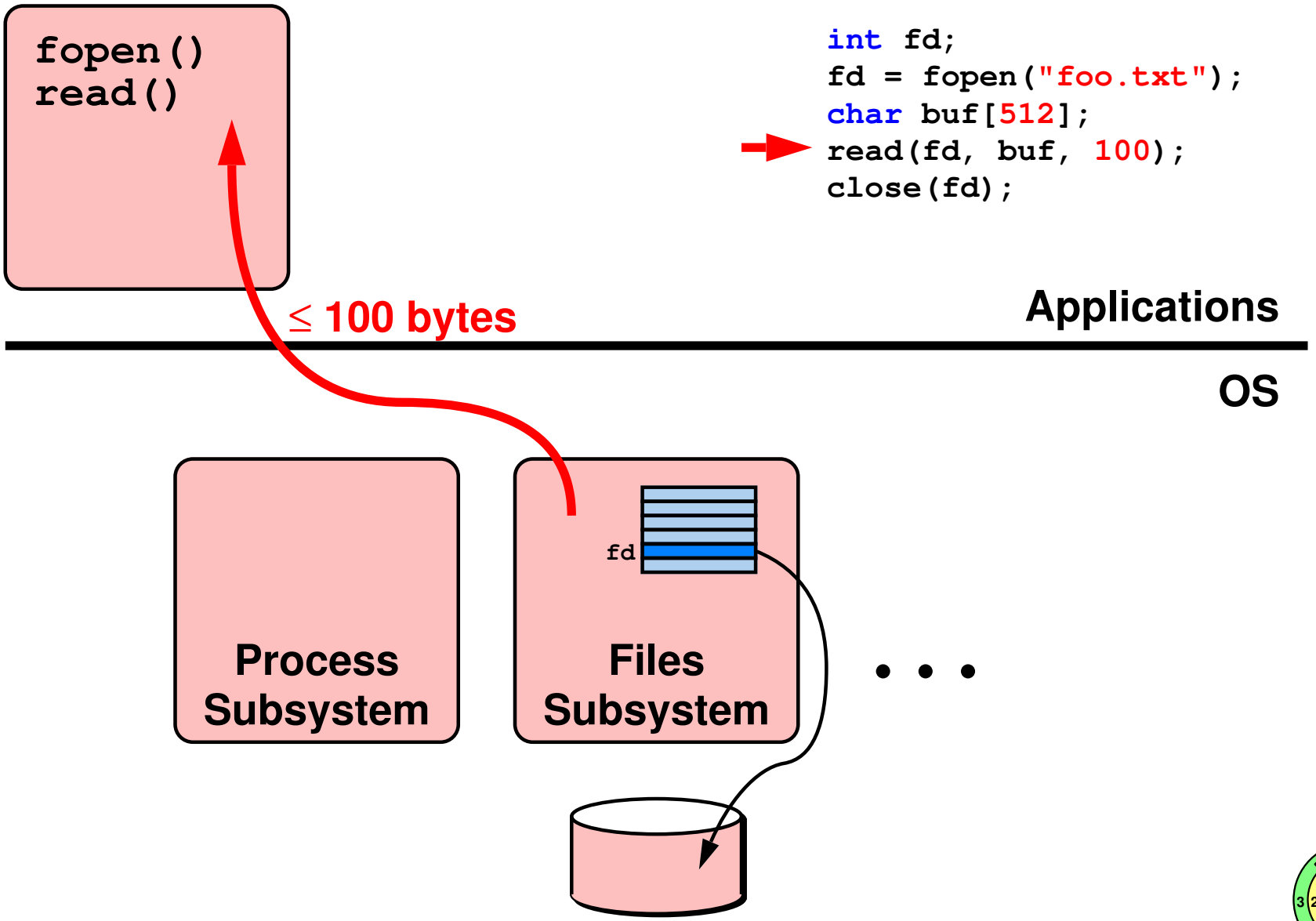




Put It All Together



Put It All Together



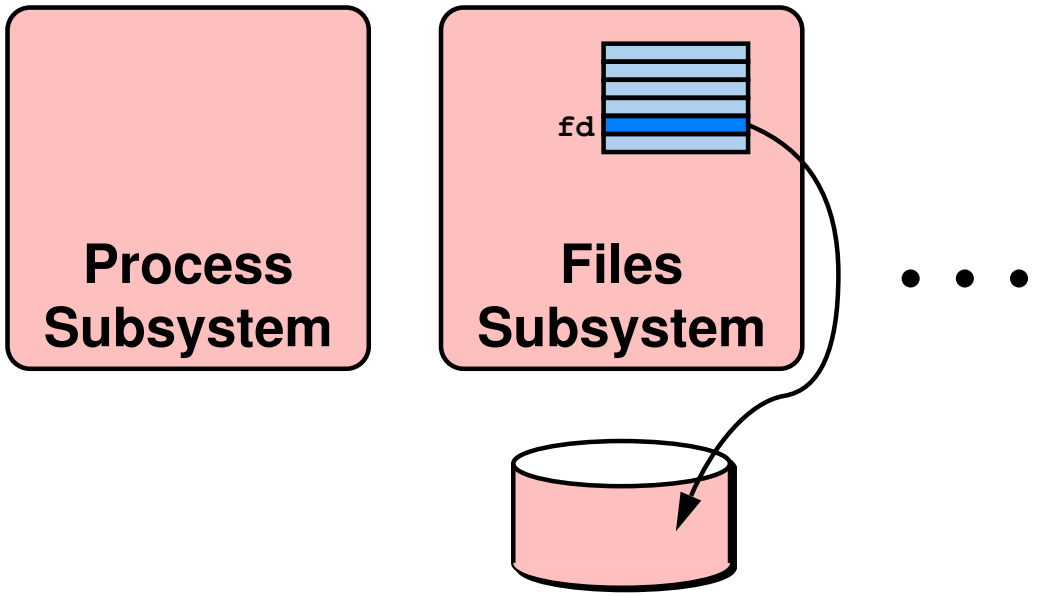
Put It All Together

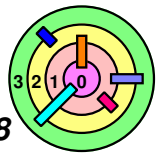
```
fopen()
read()
close()
```

```
int fd;
fd = fopen("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```

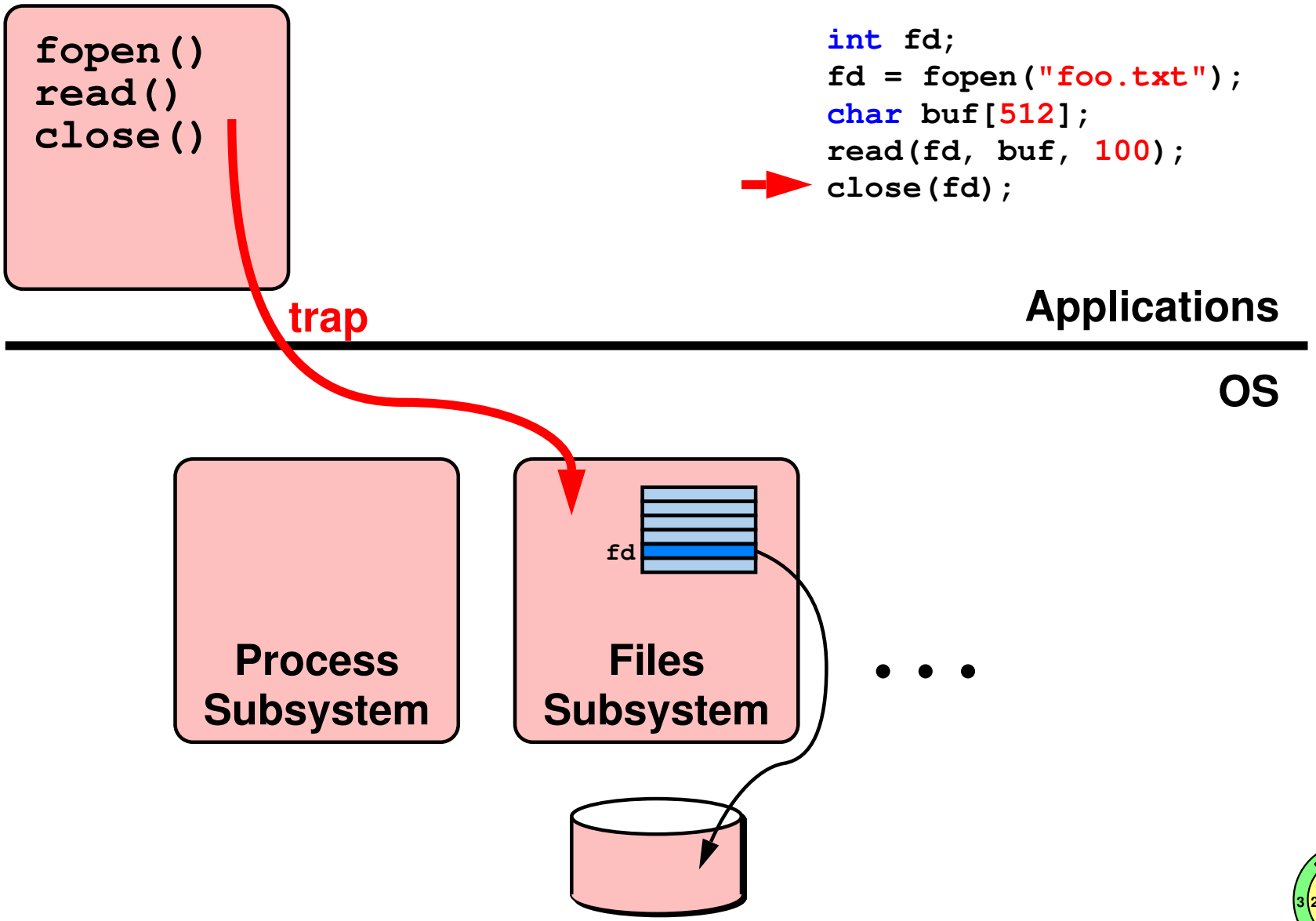
Applications

OS

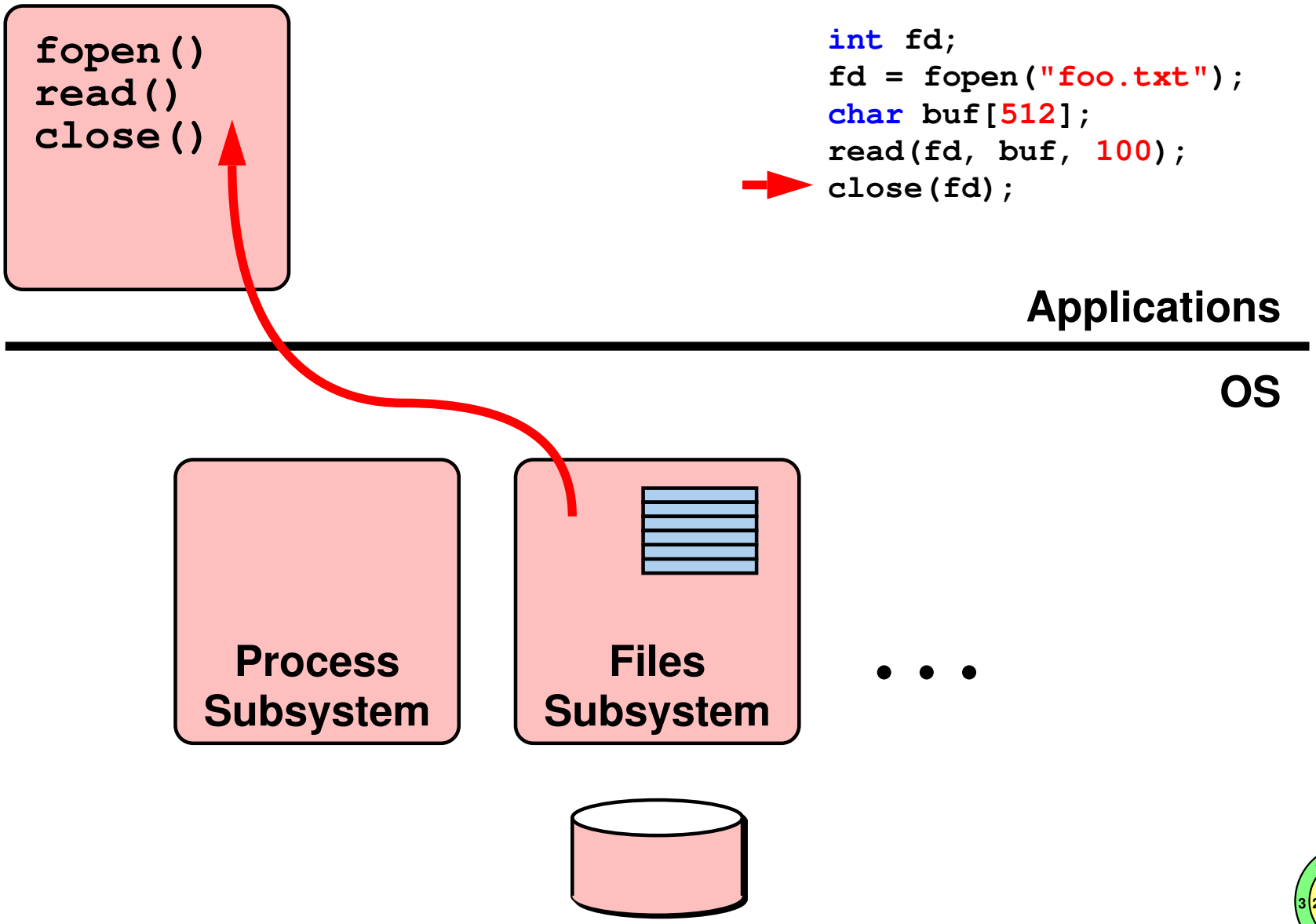




Put It All Together



Put It All Together



Put It All Together

```
fopen()
read()
close()
```

```
int fd;
fd = fopen("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```

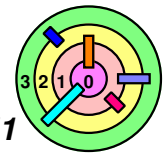
Applications

OS

Process Subsystem

Files Subsystem

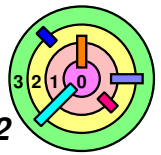
• • •



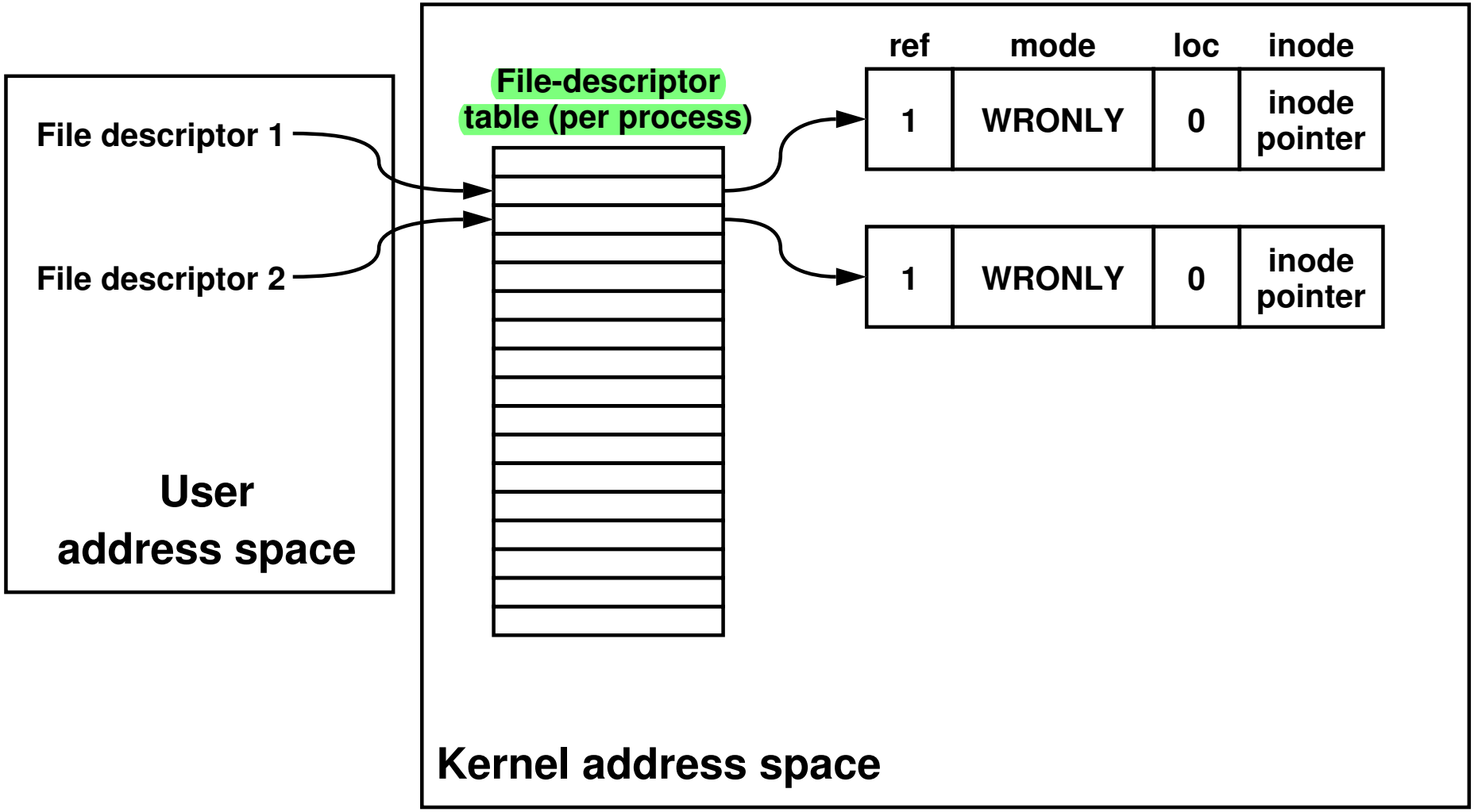
Redirecting Output ... Twice

```
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 */
```

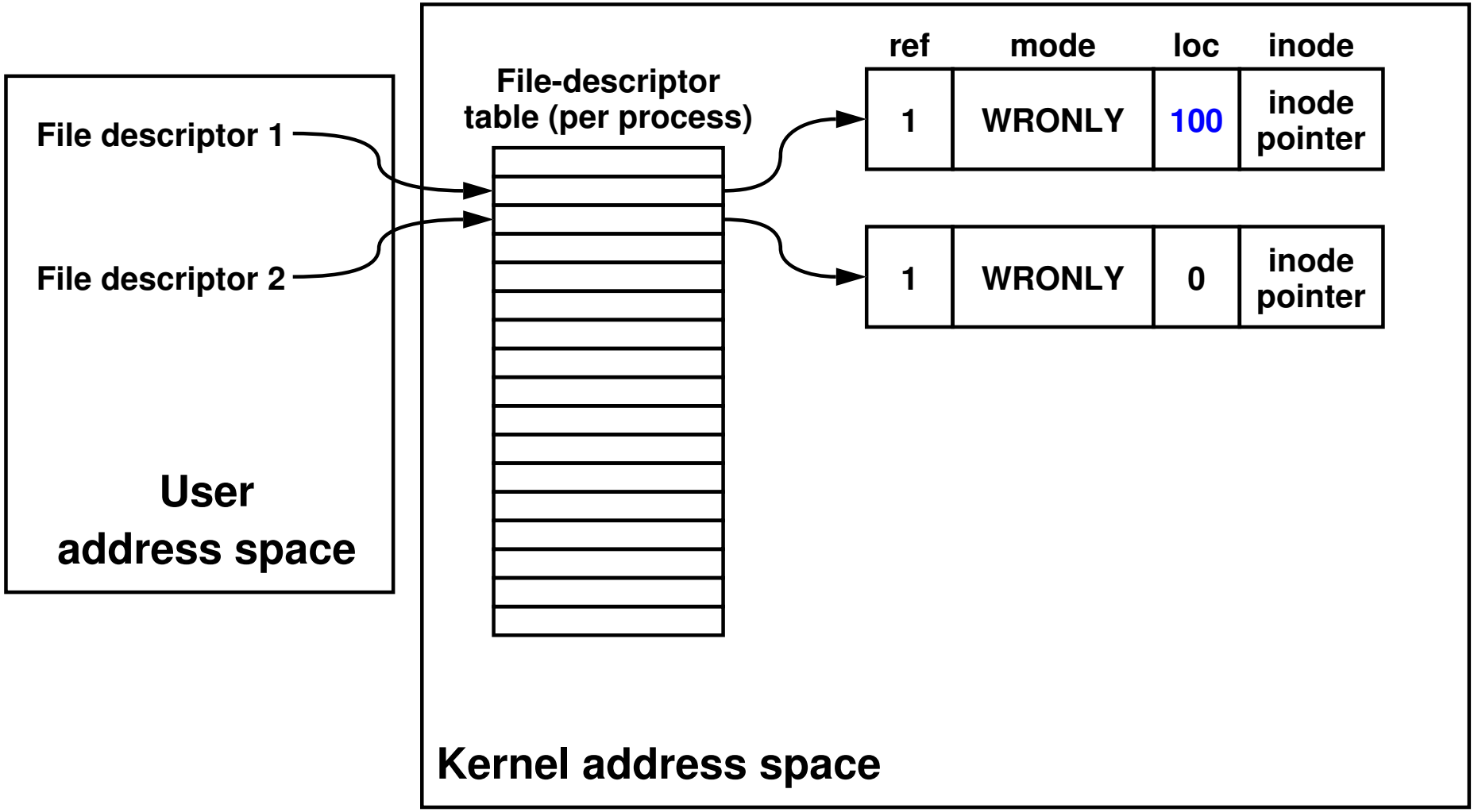
- `stdout` and `sterr` both goes into the same file
  - would it cause any problem?



Redirected Output



Redirected Output After Writing 100 Bytes

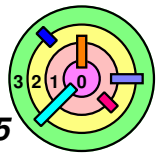


- write(2) will wipe out something in the first 100 bytes
  - that may not be the intent

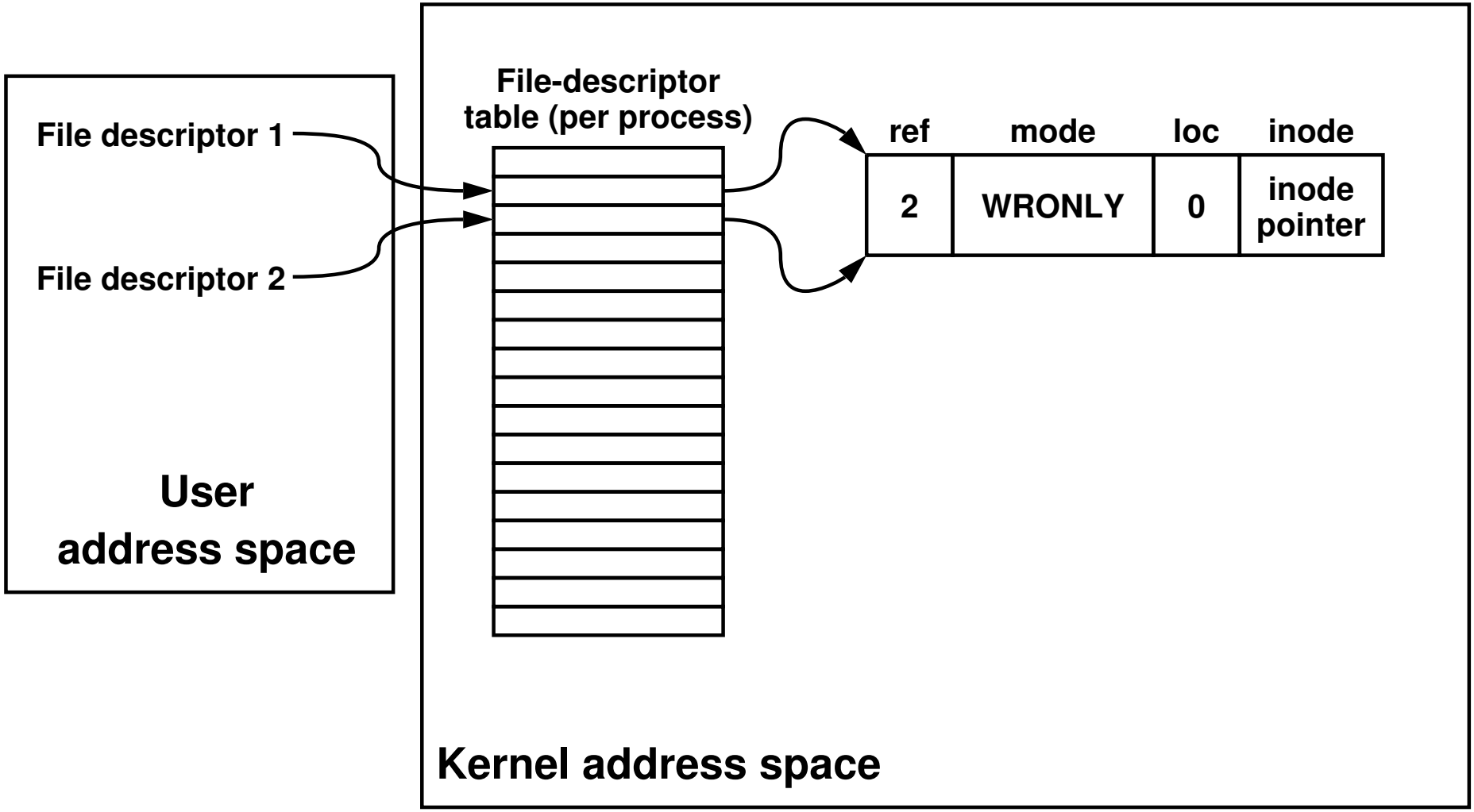
Sharing Context Information

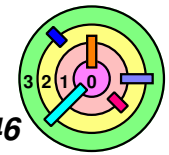
```
if (fork() == 0) {
  /* set up file descriptors 1 and 2 in the child
    process */
  close(1);
  close(2);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
     exit(1);
  }
  dup(1);
  execl("/home/bc/bin/program", "program", 0);
  exit(1);
}
/* parent continues here */
```

- use the `dup()` system call to ***share*** context information
  - if that's what you want



Redirected Output After Dup



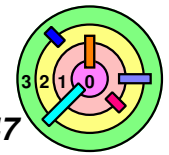


Fork and File Descriptors

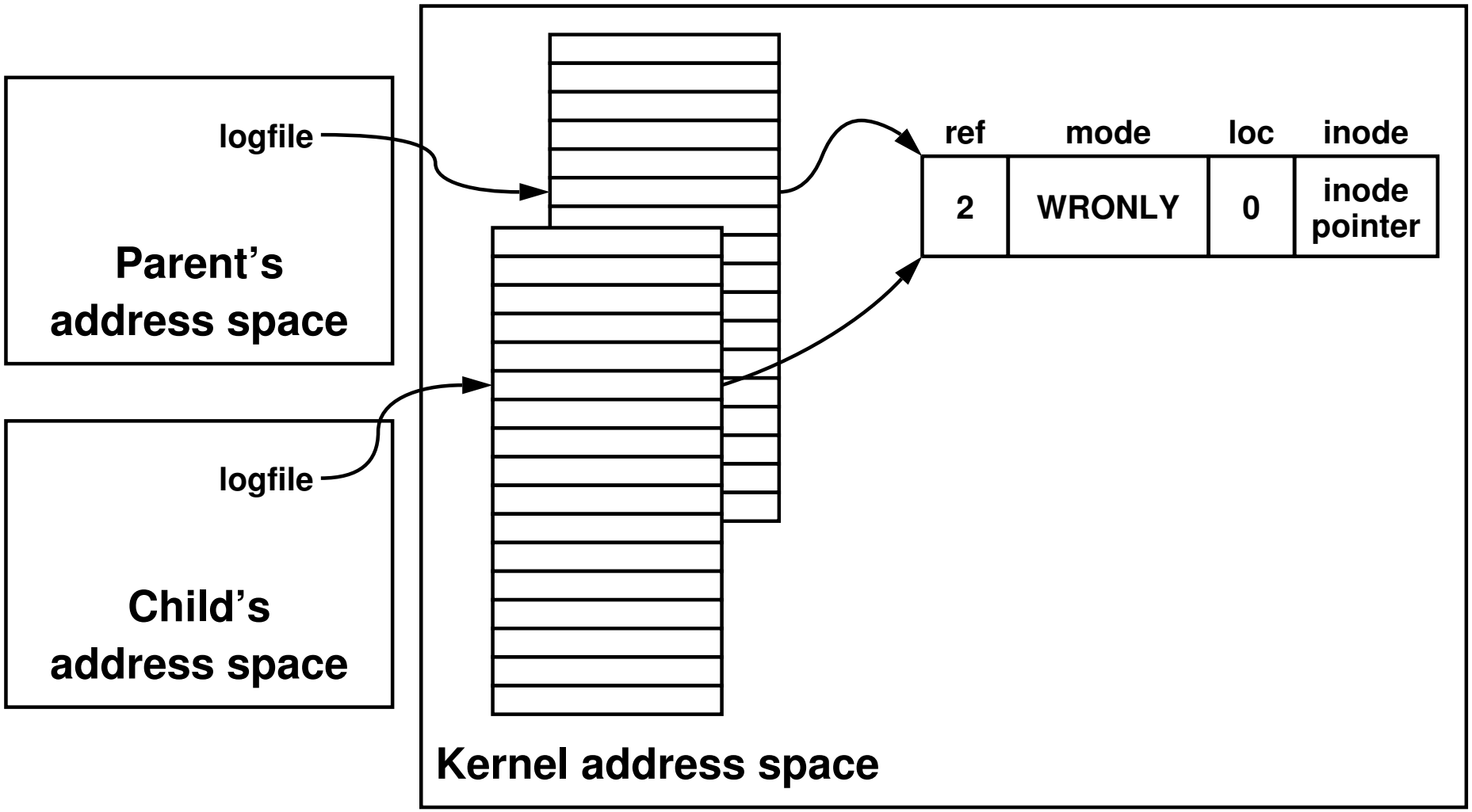
- When `fork()` is called, the child process gets a *copy* of the parent's file descriptor table

```
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()`



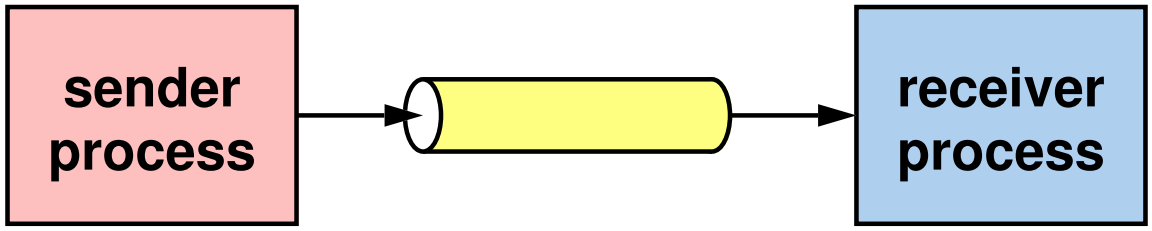
File Descriptors After Fork



- parent and child processes get *separate* file descriptor table but *share* context information

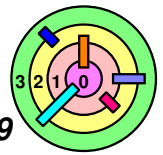
Pipes

- A pipe is a means for one process to send data to another directly, as if it were writing to a file



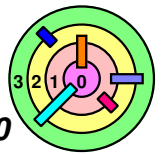
  - the sending process behaves as if it has a file descriptor to a file that has been opened for writing
  - the receiving process behaves as if it has a file descriptor to a file that has been opened for reading

- 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

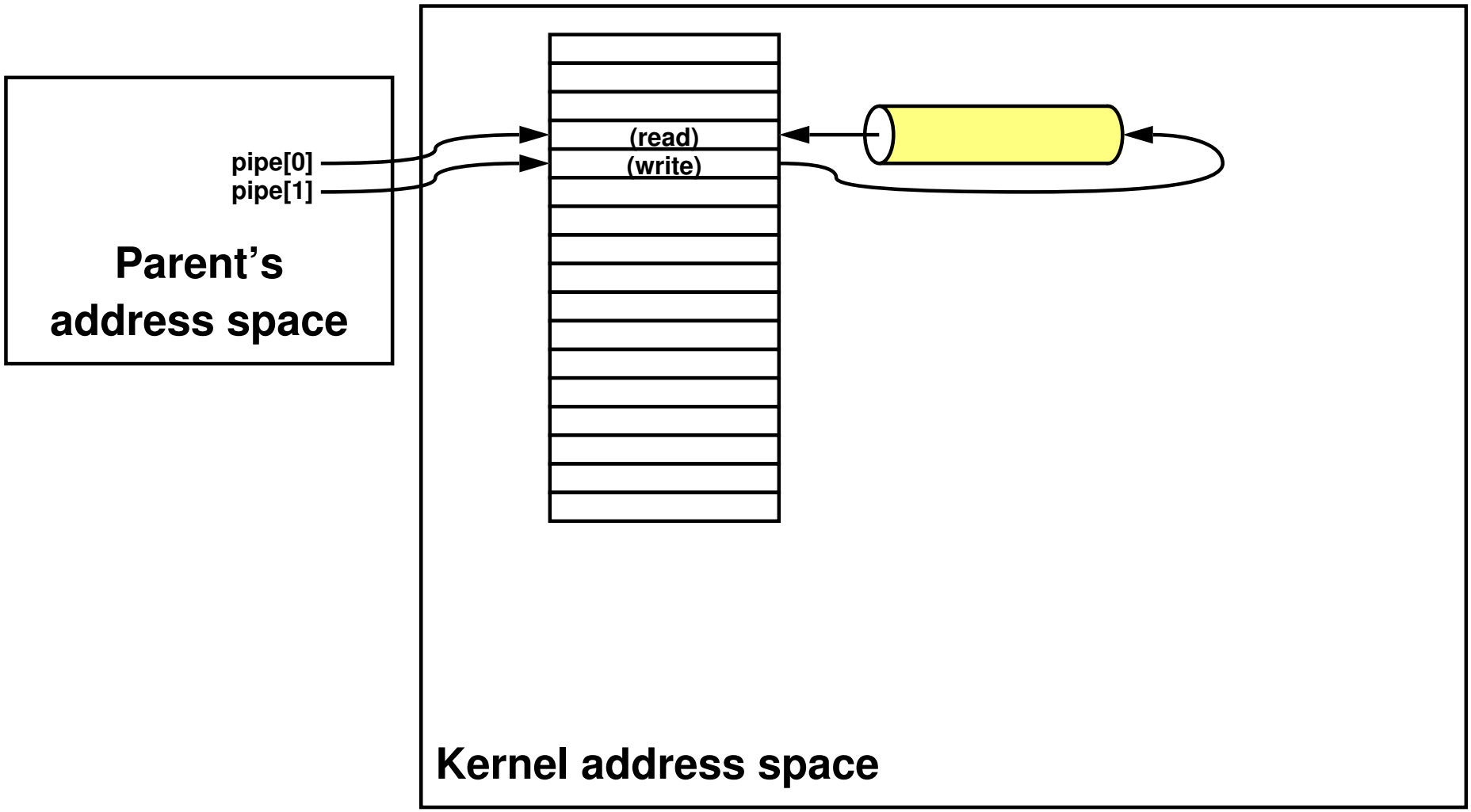


Pipes

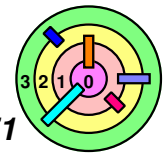
```
int p[2]; // array to hold pipe's file descriptors
pipe(p);  // creates a pipe, assume no errors
   // p[0] refers to the read/output end of the pipe
   // p[1] refers to the write/input end of the pipe
if (fork() == 0) {
   char buf[80];
   close(p[1]); // not needed by the child
   while (read(p[0], 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);
   }
}
```



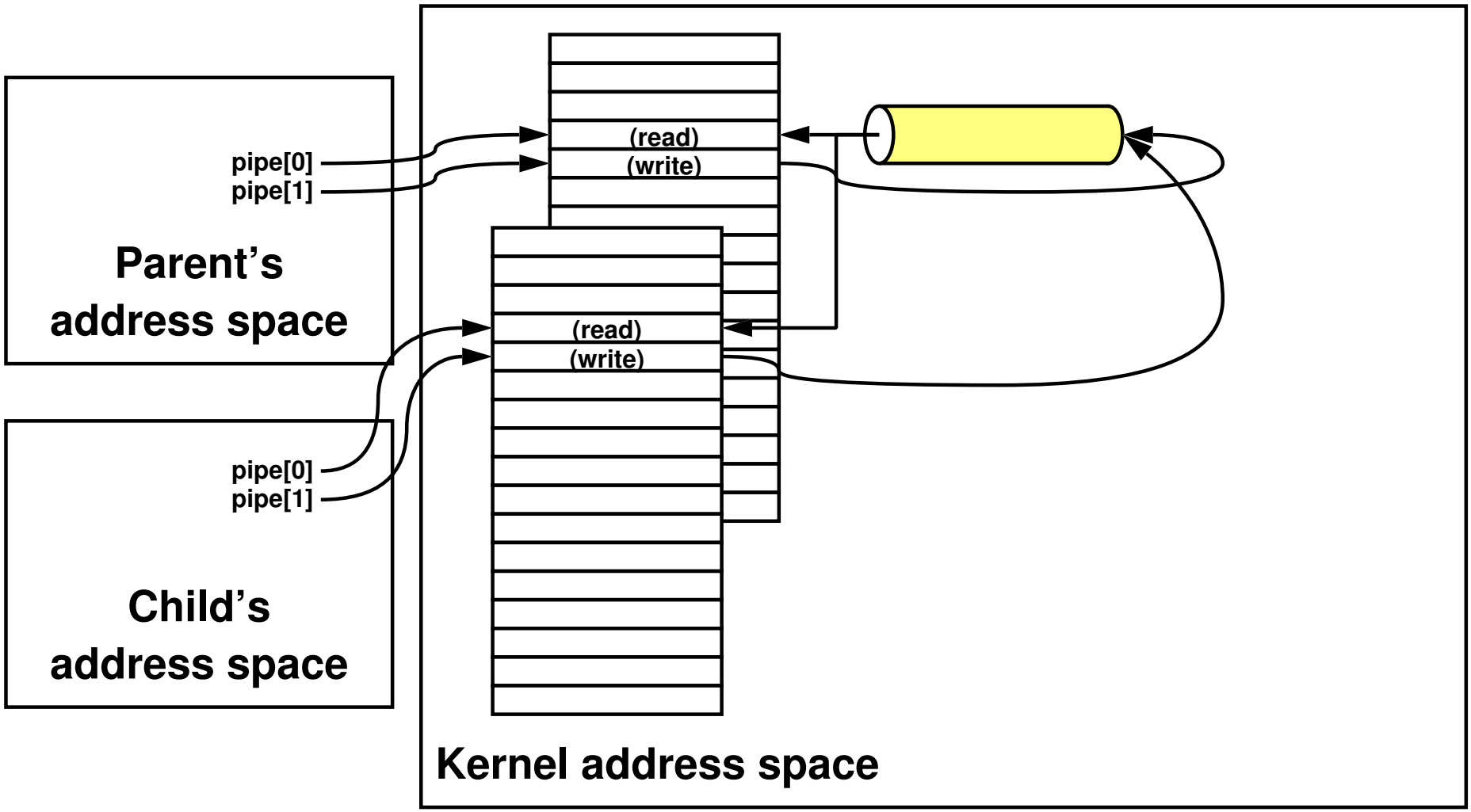
Pipes



- parent creates a pipe object in the kernel

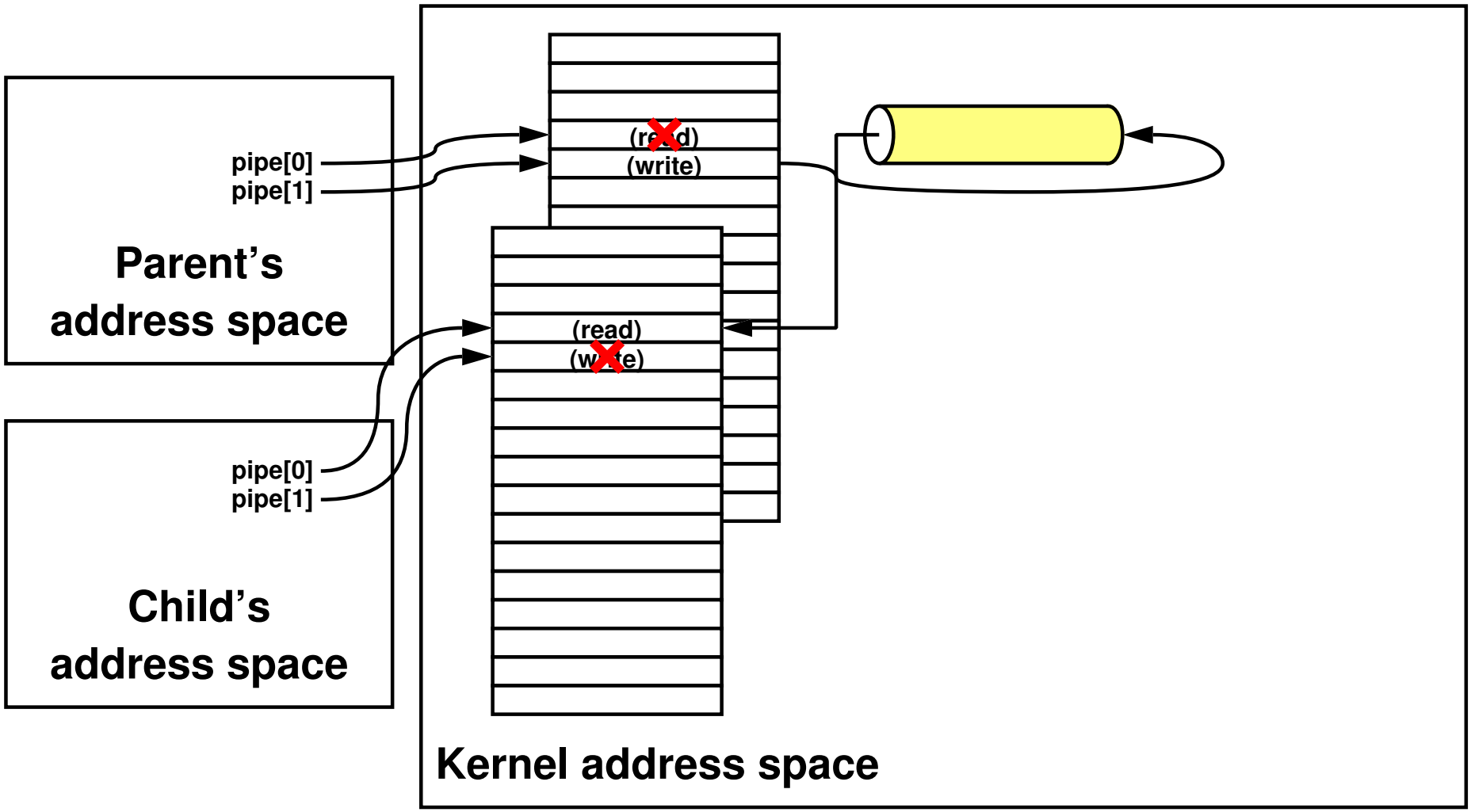


Pipes

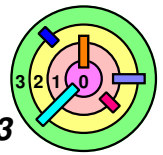


- parent and child processes get *separate* file descriptor table but *share* context information

Pipes



- parent closes the read-end of the pipe
  child closes the write-end of the pipe



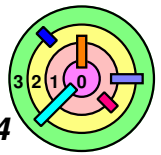
Command Shell

- 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 &
```



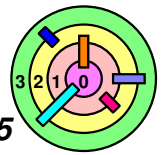
Random Access

```
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

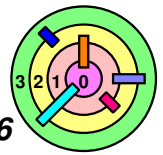
  ```
  off_t lseek(int fd, off_t offset, int whence);
  ```

- `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

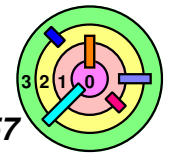
- (Almost) everything has a path name
  - files
  - directories
  - devices (known as *special files*)
  - keyboards
  - displays
  - disks
  - etc.



Uniformity

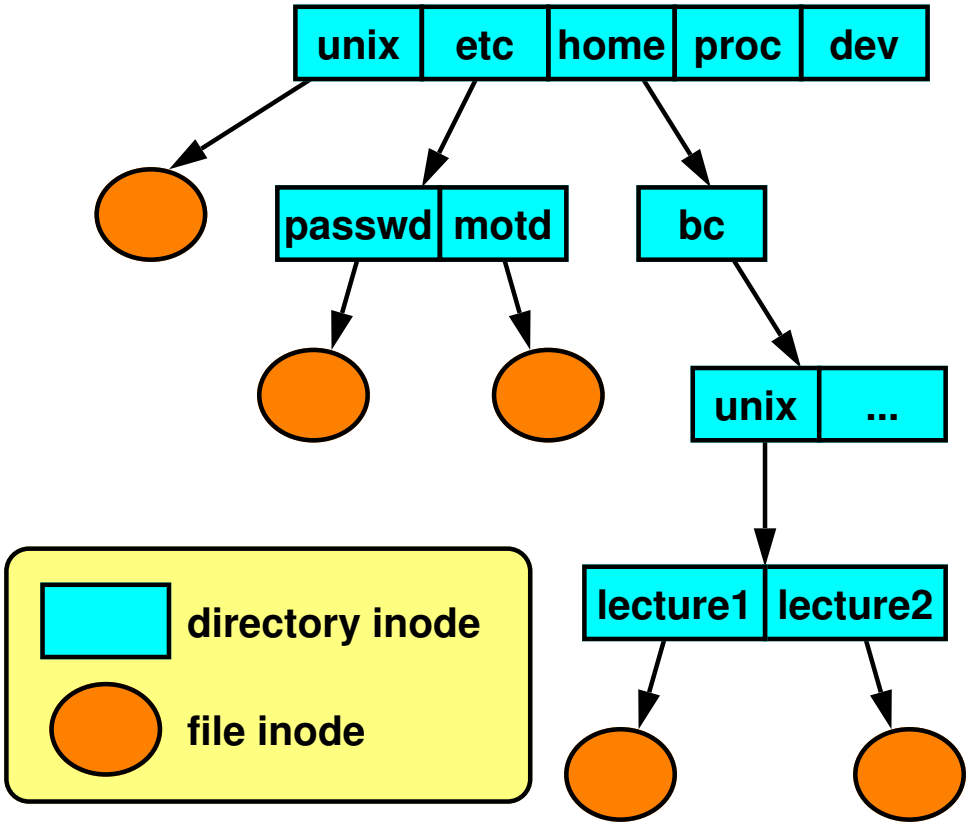
```
// opening a normal file
int file = open("/home/bc/data", O_RDWR);

// opening a device (one's terminal or window)
int device = open("/dev/tty", O_RDWR);
```

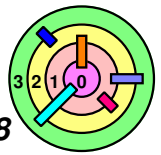
```
int bytes = read(file, buffer, sizeof(buffer));
write(device, buffer, bytes);
```



Directories

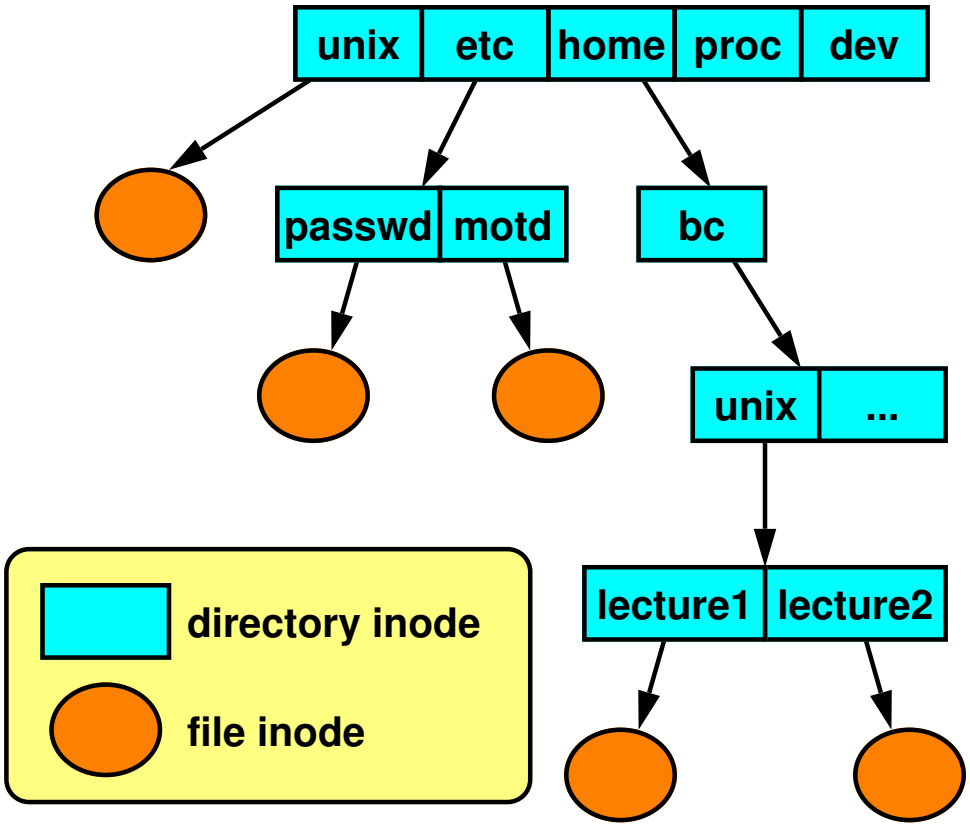
- **A directory is a file**
  - interprets differently by the OS as containing references to other files/directories
  - a file is represented as an ***index node*** *(*or ***inode****)* in the file system





Directory Representation

- A root directory entry example
  - parent inode number = its own inode number

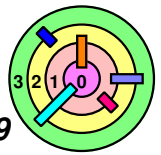


Component Name	Inode number

directory entry

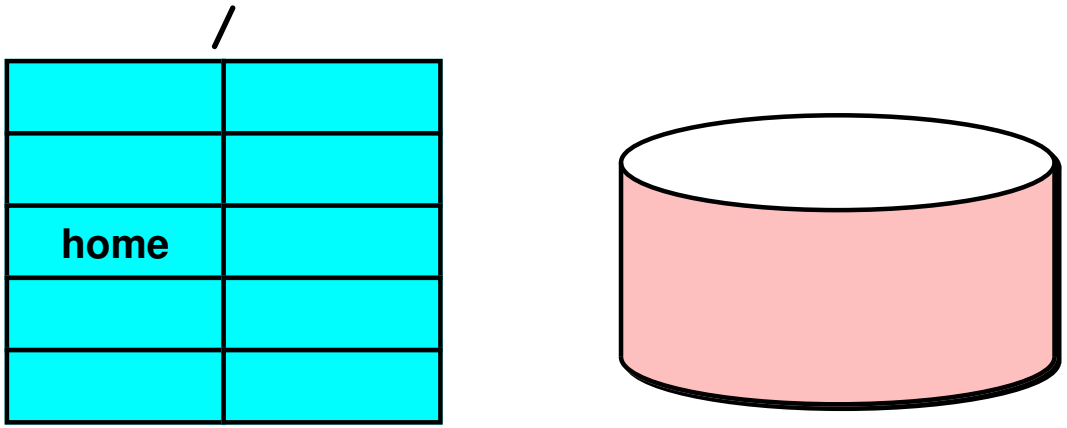
.	1
..	1
unix	117
etc	4
home	18
proc	36
dev	93

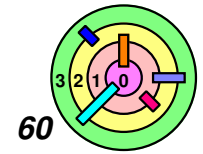
- Tree structured hierarchy



Directory Hierarchy

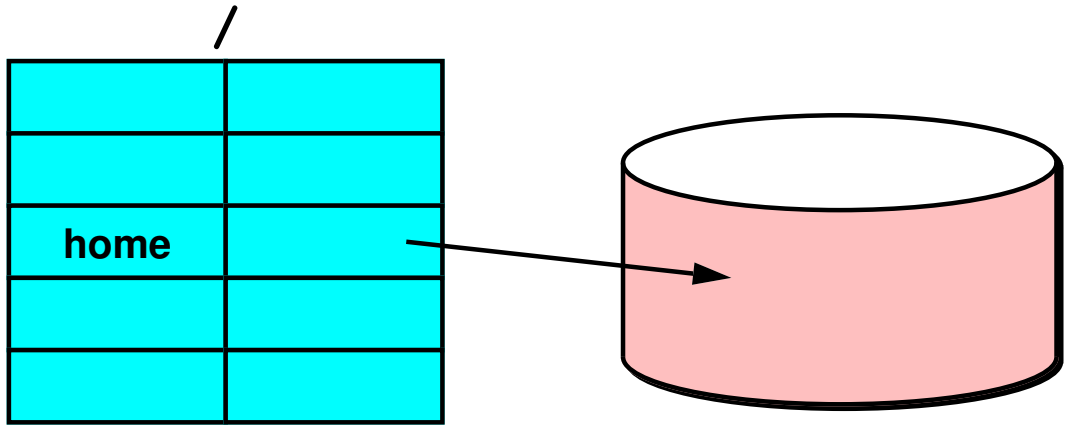
Ex: how do you get to `/home/bc/foo.c`?

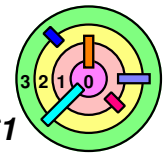




Directory Hierarchy

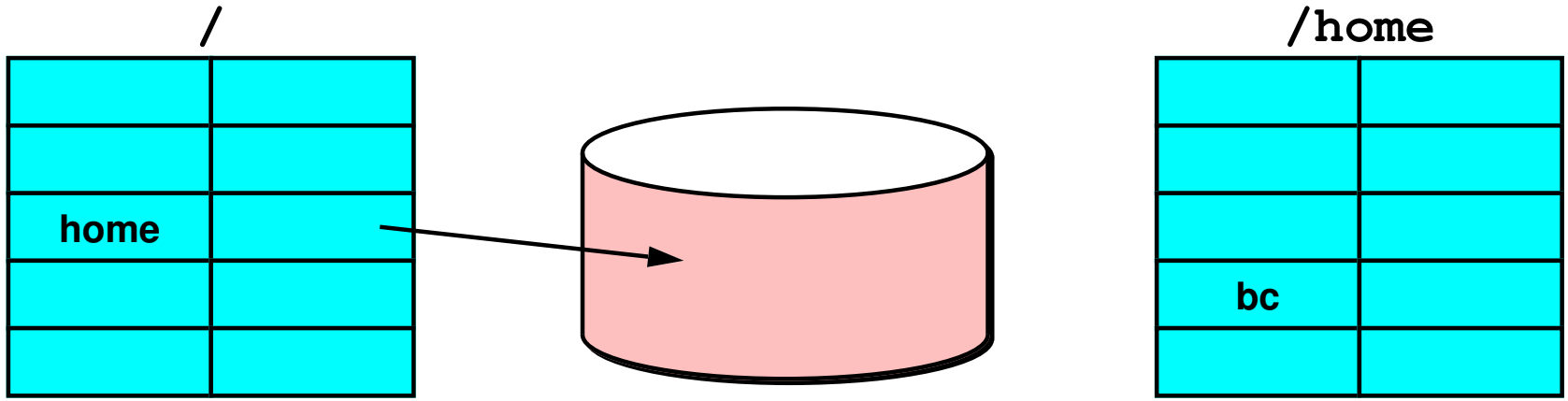
- Ex: how do you get to `/home/bc/foo.c`?

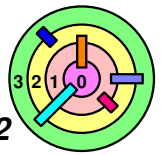




Directory Hierarchy

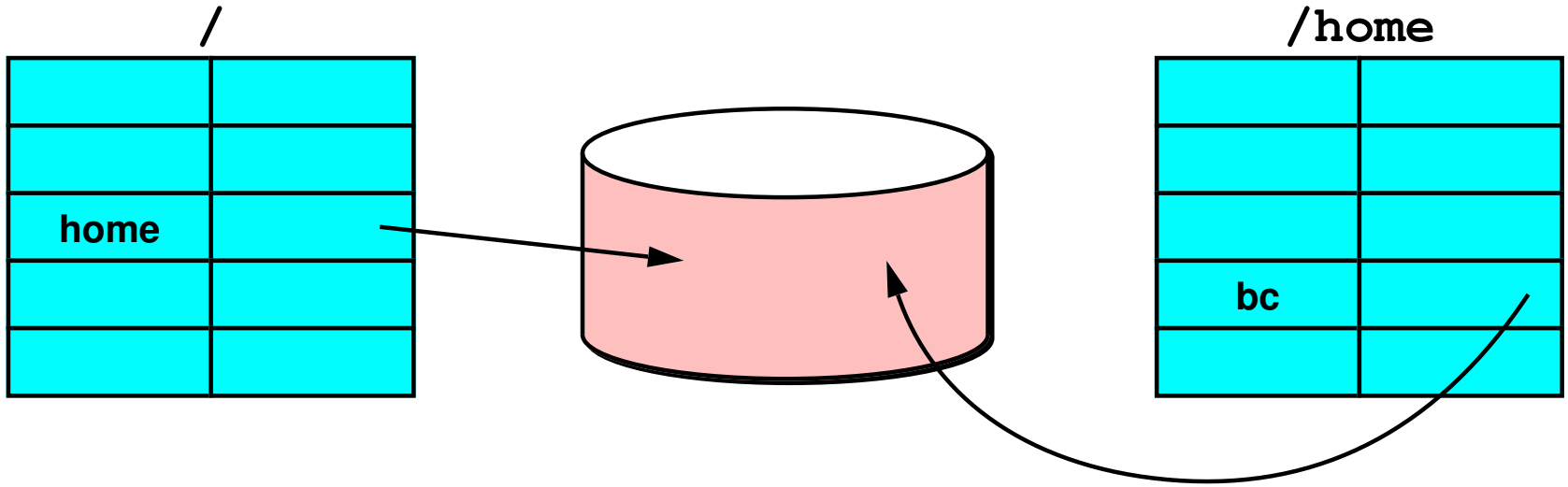
Ex: how do you get to `/home/bc/foo.c`?

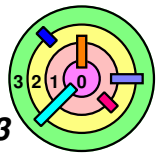




Directory Hierarchy

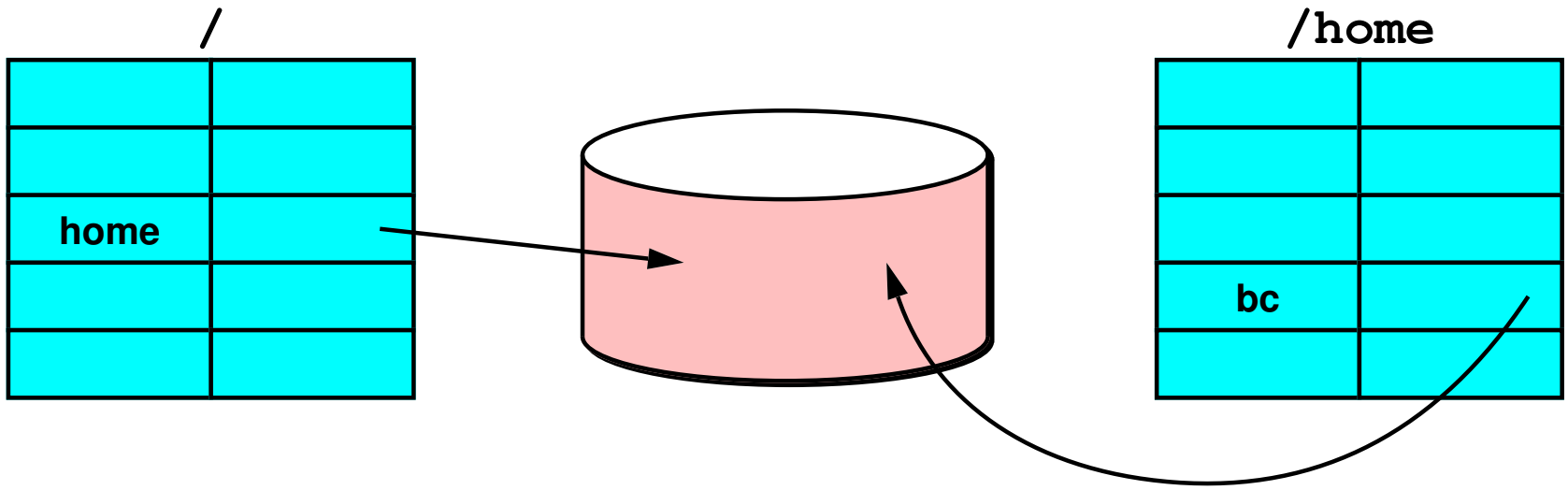
Ex: how do you get to `/home/bc/foo.c`?

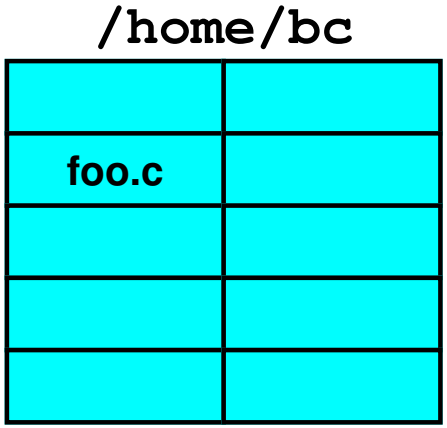


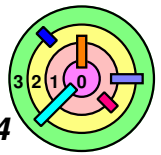


Directory Hierarchy

- Ex: how do you get to `/home/bc/foo.c`?

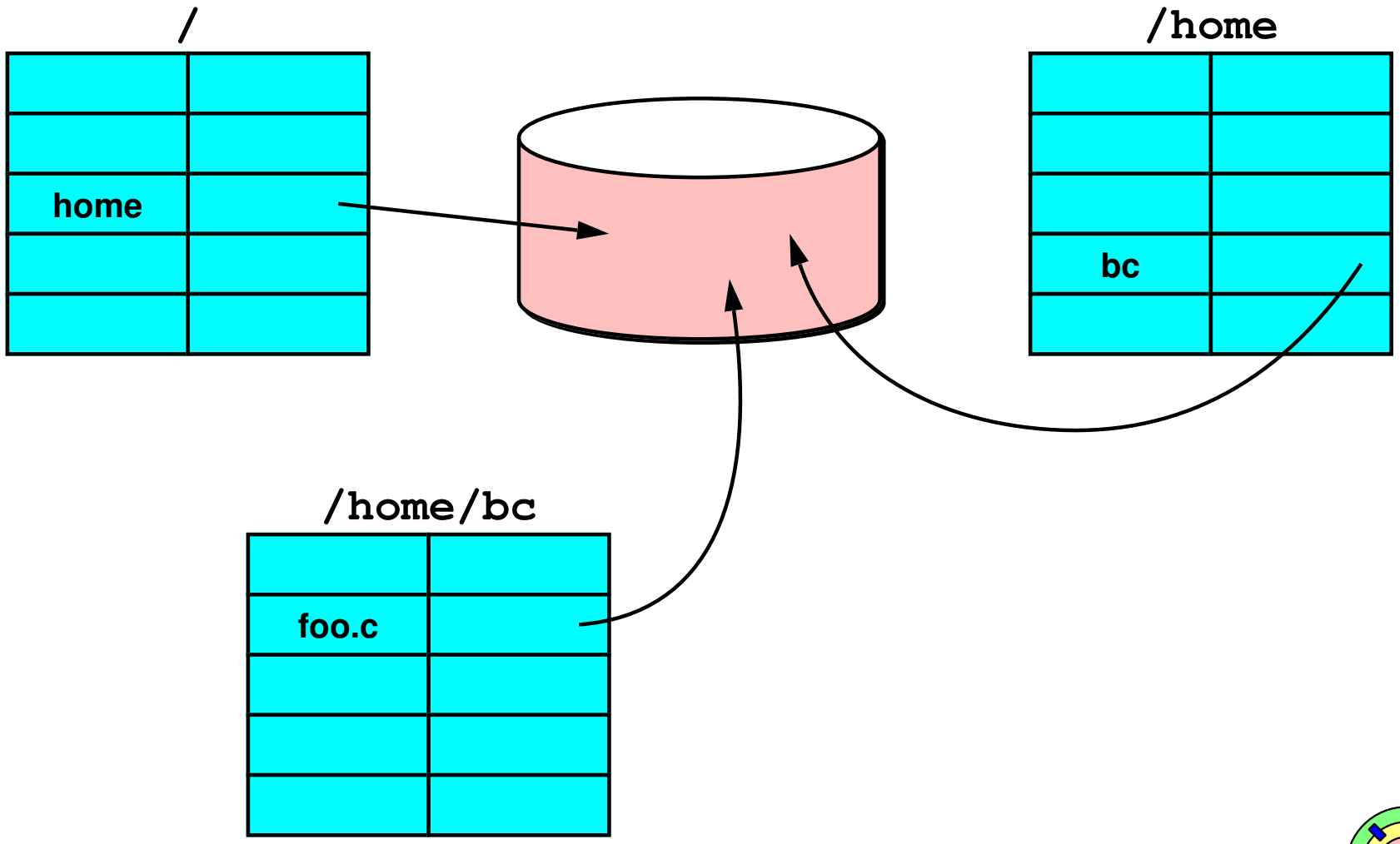






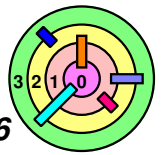
Directory Hierarchy

Ex: how do you get to `/home/bc/foo.c`?



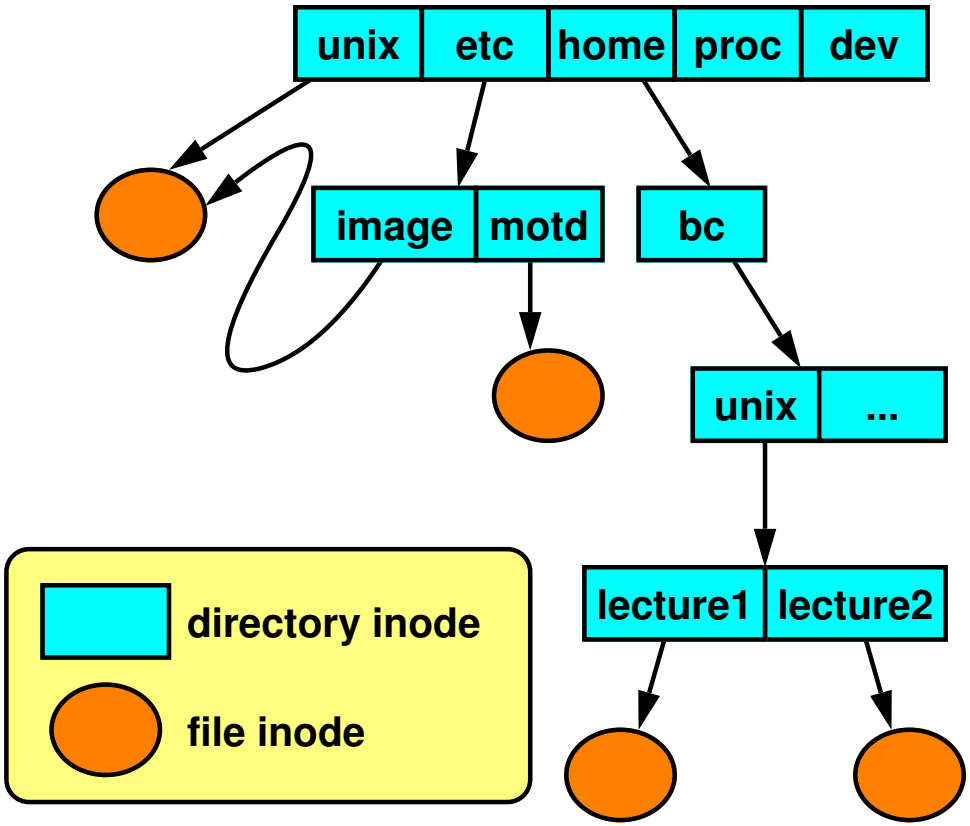
Directory Hierarchy

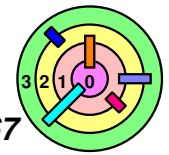
- Unix and many other OSes allow limited deviation from trees
  - ***hard links***
    - reference to a ***file*** (not a directory) in one directory that also appears in another
    - using the `link()` system call or the "ln" shell command
  - ***soft links*** or ***symbolic links***
    - a special kind of ***file*** containing the ***name*** of another file or directory
    - using the `symlink()` system call or the "ln -s" shell command



Hard Links

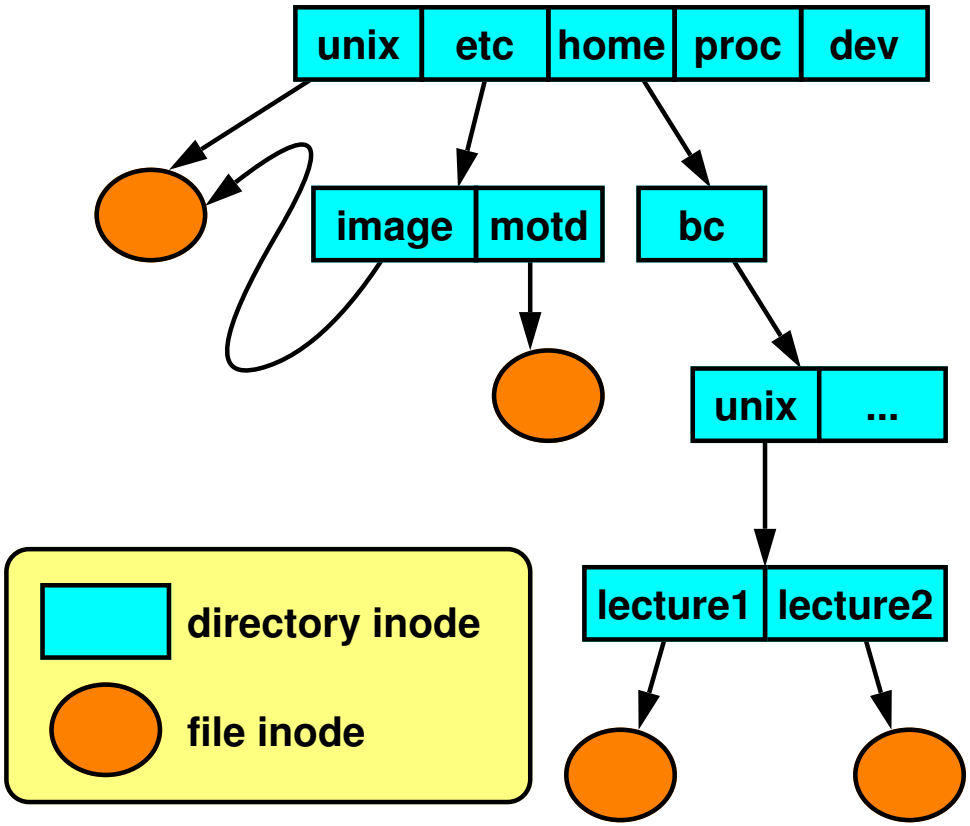
```
% ln /unix /etc/image
```





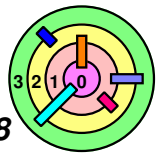
Hard Links

```
% ln /unix /etc/image
```



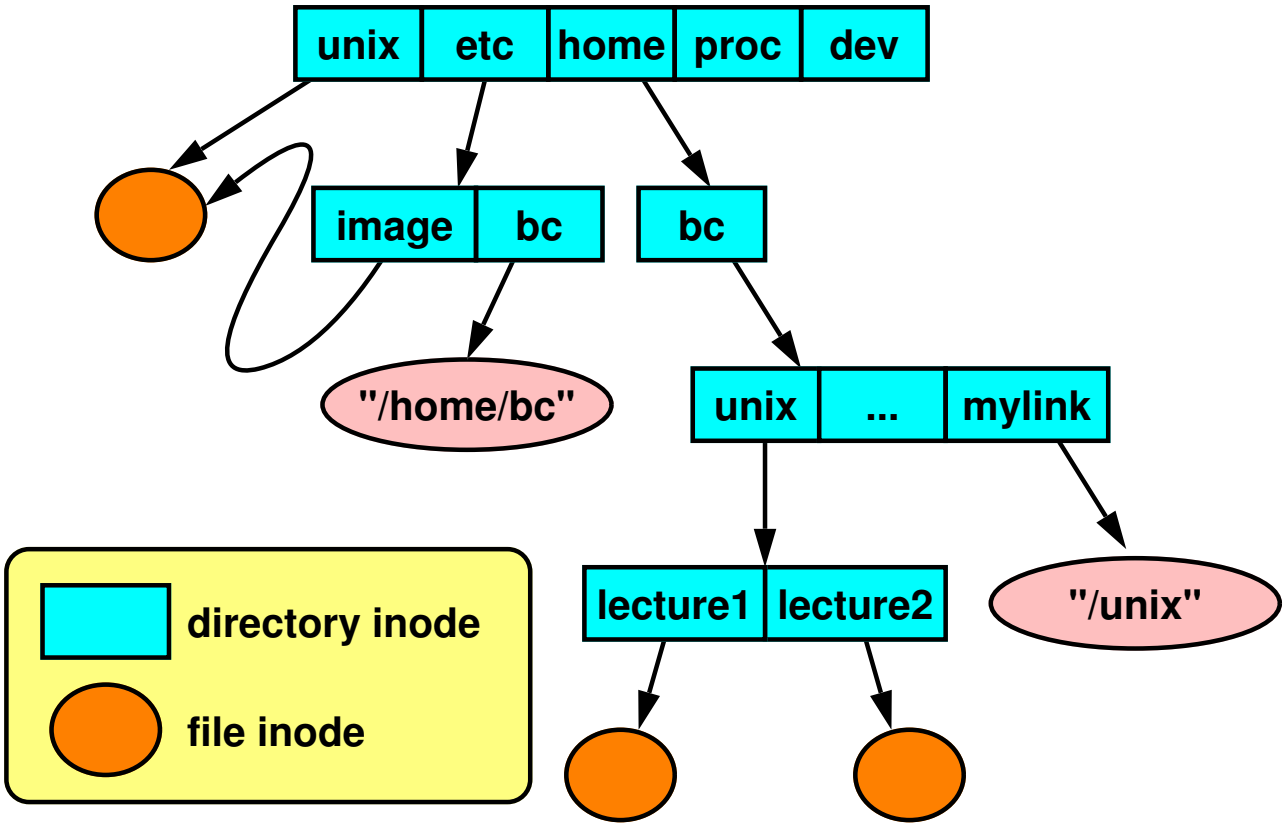
.	1
..	1
unix	117
etc	4
home	18
proc	36
dev	93

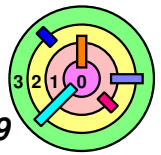
.	4
..	1
image	117
motd	33



Soft Links

```
% ln -s /unix /home/bc/mylink
% ln -s /home/bc /etc/bc
```





Working Directory

- Maintained in kernel for each process
  - paths not starting from "/" start with the working directory
  - changed by use of the `chdir` system call
  - displayed (via shell) using "`pwd`"