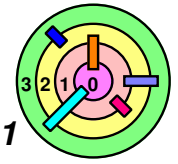
Housekeeping (Lecture 5 - 6/5/2025)

- **PA2 is due at 11:45pm on Tuesday, 7/1/2025**
  - **if you have code from current or a previous semester, *do not look at/copy/share* any code from it**
    - **it's best if you just get rid of it**
  - **if you include files that's not part of the original "`make pa2-submit`" command, the grader will delete them**

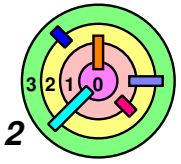
- ***Grading guidelines* is the *ONLY* way we will grade and we can only grade on a *standard 32-bit Ubuntu Linux 16.04 inside VirtualBox/UTM* or on AWS Free Tier**
  - **although not recommended, you can do your development on a different platform**
    - **you must test your code on the "standard" platform because those are the *only* platforms the grader is allowed to grade on**



Housekeeping (Lecture 5 - 6/5/2025)

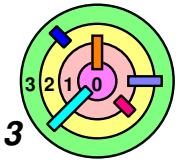
- If you make a submission
  - read and understand the *ticket* in *web page*
    - save the web page as PDF as record of your submission
  - make sure you follow the *"Verify Your Ticket"* and *"Verify Your Submission"* procedure as if you are the grader
  - make sure your README file is perfect
    - if there is anything you are not sure about, please ask me and don't assume that whatever you do will be fine

- Remember that you can use your "free late days"
  - you don't need my permissions to use "free late days"
  - PA1 is only worth 2.4% of your overall grade and a step in the grade is 6%
    - you need to decide if it's worth it to use "free late days" on something that may not affect your letter grade
  - once a "free late day" is used, if assignment is graded, you cannot take back that "free late day"



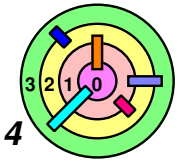
Housekeeping (Lecture 5 - 6/5/2025)

- If you cannot finish PA1, you need to try to get as much partial credit as you can (or submit by PA2 deadline for -50%)
  - if you want the grader to skip a test, you should give yourself 0 points in the corresponding item in `pa1-README.txt`
    - if a grader runs a command and it doesn't work, the grader must deduct points according to the grading guidelines



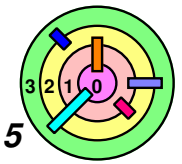
Housekeeping (Lecture 5 - 6/5/2025)

- PA2 and PA3 are sepeate assignments and they will be graded separately
  - if you only make a PA3 submission, you will get a score of 0 for your PA2 submission
  - I would suggest that when you are done with PA2, make a submission and keep the source in the cs350/pa2 directory and don't touch it
  - create a cs350/pa3 directory by copying everything from the cs350/pa2 directory and start working on PA3 in the cs350/pa3 directory
  - when you are working on PA3 and discovered bugs in your PA2 code, you need modify your PA2 code in both the pa2 directory and the pa3 directory and make another PA2 submission



kthread_join()

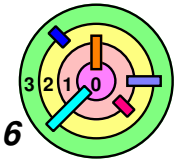
- This function suspends the execution of the calling thread until the target thread (of the same process), indicated by the argument thread_id, terminates
  - if the thread has already exited, execution should not be suspended
  - if successful, the function returns zero
  - otherwise, -1 should be returned to indicate an error



kthread_join()

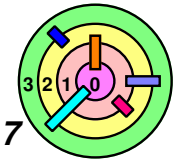
```
kthread_join(int thread_id):

  check if thread_id is valid
  create a thread pointer t
  loop through all threads to find target thread id (parameter)
    make t points to target thread with thread_id
  if not found
    return -1
  while (t->tid == thread_id and t is not in the TZOMBIE state)
    make t sleep using sleep() function with a lock // read sleep() code
  if state of t is zombie
    clearThread(t);
  return 0
```

- Note: the above is not the only way to join threads
- also, this is not a complete pseudocode
  - you have to add locks if necessary



Common Errors

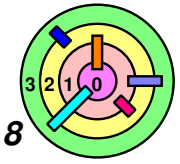
- You are supposed to read the code of the test programs
  - `threadtest1.c`
  - `threadtest2.c`
  - `threadtest3.c`
- You are supposed to be reading the XV6 book: xv6-rev11.pdf and the XV6 source code to understand how the scheduler works



Common Errors

```
$ threadtest1
3 threadtest1: unknown sys call 23
thread in main -1,process 3
3 threadtest1: unknown sys call 22
3 threadtest1: unknown sys call 22
3 threadtest1: unknown sys call 25
Got id : -1
3 threadtest1: unknown sys call 25
Got id : -1
Finished.
3 threadtest1: unknown sys call 24
...
```

Make sure to implement system calls for all kthread functions



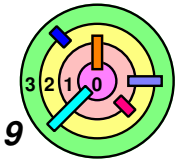
Common Errors

```
cpu with apicid 0: panic: acquire
 80104b85 80104334 80100226 80101a78 80101c4a ...
```

OR:

```
cpu with apicid 0: panic: release
 80104cc8 80103b51 80105dec 80105129 80106338 801060eb ...
```

- **panic: acquire and panic: release errors mean that program fails to acquire lock because it is already acquired earlier or it cannot released lock because it is already released**

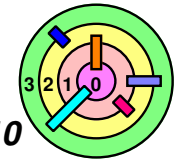


Common Errors

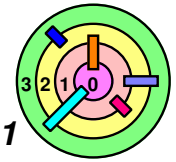
```
panic: sched locks
```

- this means process holding multiple locks
  - before calling `sched()` make sure to release all other locks

```
ac(ptable)
ac(mtable)

rel(mtable)
rel(ptable)
```

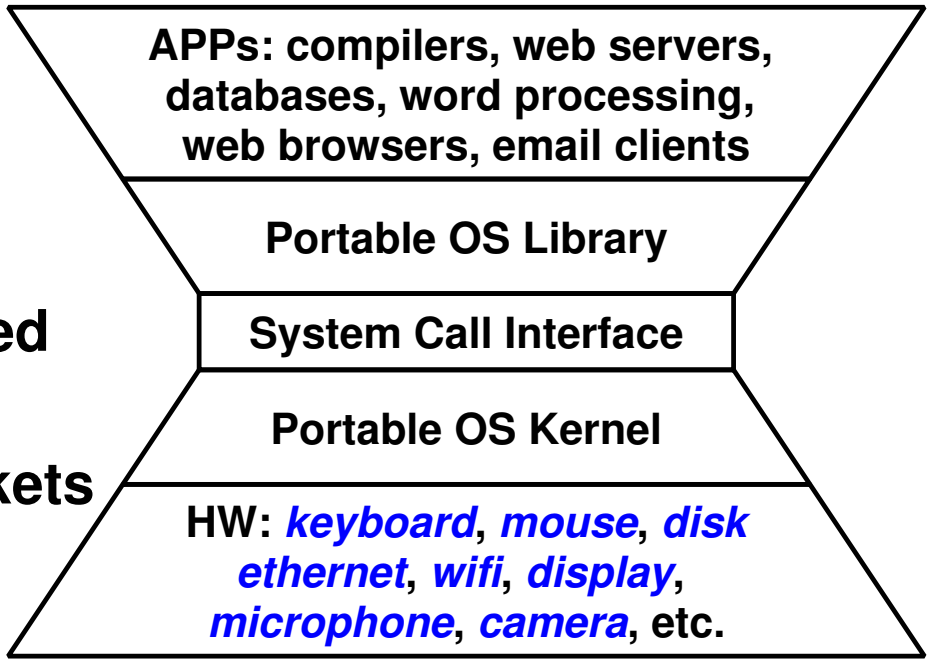


(3.2) Input/Output

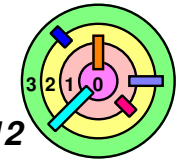


Input/Output: UNIX I/O

- How to interact with I/O devices?
  - a disk is addressed in fixed sized blocks/chunks
    - only returns data when asked
  - a network sends and receives a stream of variable sized packets
    - returns data unprompted
  - keyboard returns individual characters as keys are pressed
    - returns data unprompted

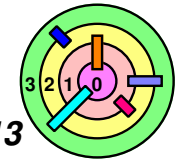


- When a new type of device is invented, it would be bad if new system call interface has to be upgraded to handle that device



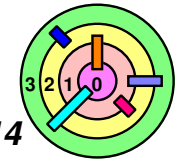
(3.2) Input/Output: UNIX I/O

- One of the primary innovation in UNIX was to regularize all device input and output behind a single common interface
  - UNIX tooks this one step further to use the same interface for reading and writing files and for interprocess communication
- This approach was so successful and pretty much all systems follow this today



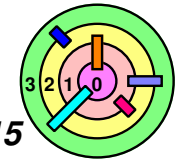
Basic Ideas In UNIX I/O Interface

- ***Uniformity:*** **all operations on all files, devices use the same set of system calls: `open()`, `close()`, `read()`, `write()`**
- ***Open before use:*** **open returns a handle (file descriptor) for use in later calls on the file**
- ***Byte-oriented abstraction:*** **even for block-oriented devices**
- ***Kernel-buffered read:*** **allows the `read()` system calls to be the same for devices with streaming read and block reads**
  - **if no data is available to be returned, `read()` blocks**
- ***Kernel-buffered write:*** **outgoing data is stored in a kernel buffer for transmission when the device becomes available**
  - **this decouples the application from the device so they can go at their own speeds**
- ***Explicit close:*** **to clean up the open file descriptor and related kernel data structures**



UNIX File System Interface

- UNIX file `open()` system call is a Swiss Army knife:
  - open the file, return file descriptor
  - options:
    - if file doesn't exist, return an error
    - if file doesn't exist, create file and open it
    - if file does exist, return an error
    - if file does exist, open file
    - if file exist but isn't empty, nix it then open
    - if file exist but isn't empty, return an error
    - ...



Interface Design Question

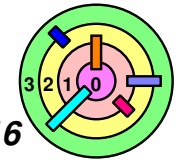
- Why not have separate syscalls for `open()`, `create()`, `exists()`?

```
if (!exists(name)) {
  create(name);  // can create fail?
}
fd = open(name); // does the file exist?
```

  - the problem with the above code is that in a multiuser system, `exists()` and `create()` need to be done in an *atomic operation*
    - another user may create the file before your `create()` call
    - another user may delete the file before your `open()` call

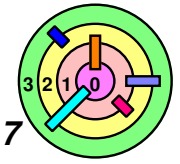
- UNIX's approach is to implement an *atomic* `open()`

- `open()` returns a file descriptor that will continue to work until the application closes the file
  - even if another user has deleted the file from the file system
  - the file system does not actually reclaim the associated disk blocks until the file is closed



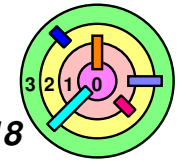
Files

- 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* (i.e., data structures in kernel space for this process)



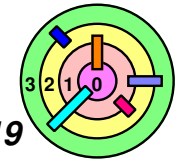
Naming Files

- 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

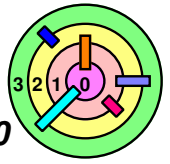
- A file is a simple array of bytes
- Files are made larger by writing beyond their current end
- Files are named by paths in a naming tree
- System calls on files are *synchronous*
- File API
  - `open()`, `read()`, `write()`, `close()`
  - e.g., `cat`



File 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
```

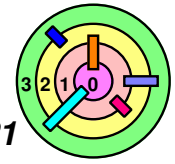
- what is O_RDWR?
- what does `perror()` do?
- *cursor* position in an opened file depends on what functions/system calls you use
  - what about C++?



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)
- The "cat" program:

```
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);
}
```

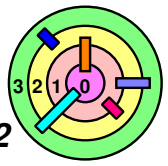


Back to Primes

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

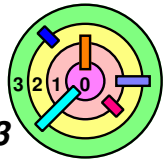
```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
   ...
   for (i=1; i<nprimes; i++) {
      ...
   }
   if (write(1, prime, nprimes*sizeof(int)) == -1) {
     perror("primes output");
     exit(1);
   }
   return(0);
}
```

  - 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);
}
```

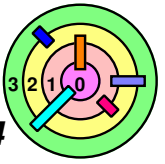


Allocation of File Descriptors

- 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)

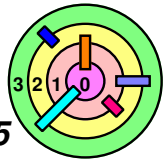


Running It

```
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
```



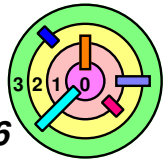
I/O Redirection

```
% 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

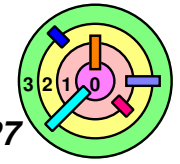
```
% cat < /home/bc/Output
```

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

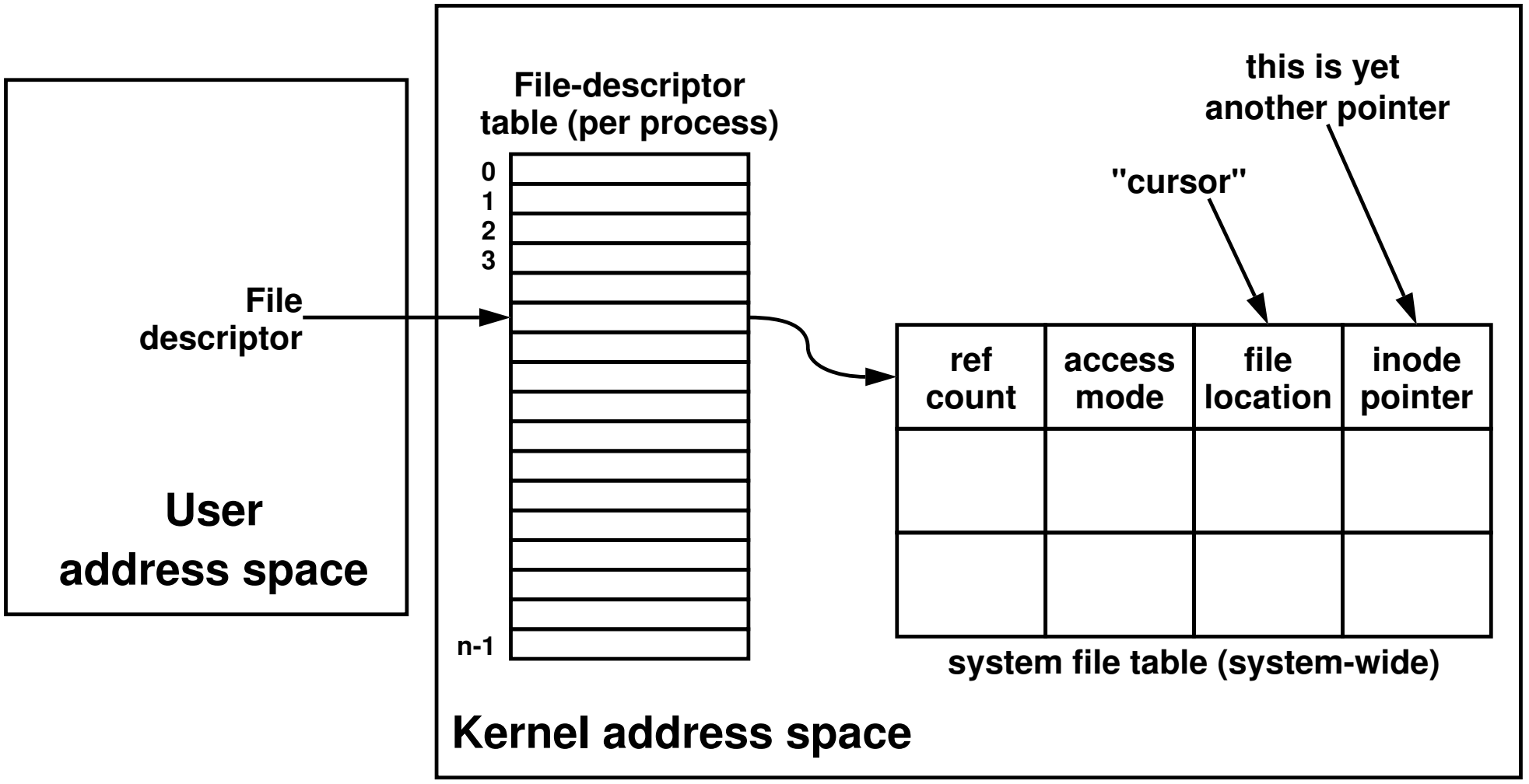


File Descriptor Table

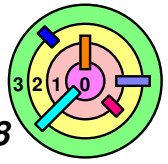
- A file descriptor refers not just to a file
  - it also refers to the *process's* current *context* for that file
    - includes how the file is to be accessed (how `open()` was invoked)
    - *cursor* position
- *Context* information must be maintained by the OS and not directly by the user program
  - 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 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 handle is an *index* into an array maintained for the process in kernel's address space



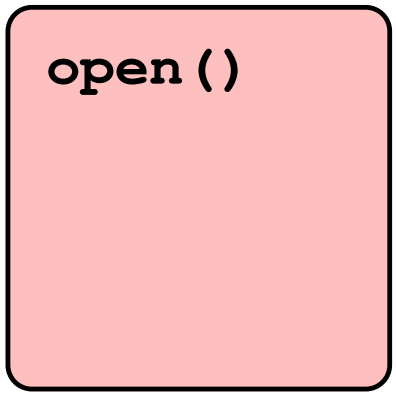
File Descriptor Table



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



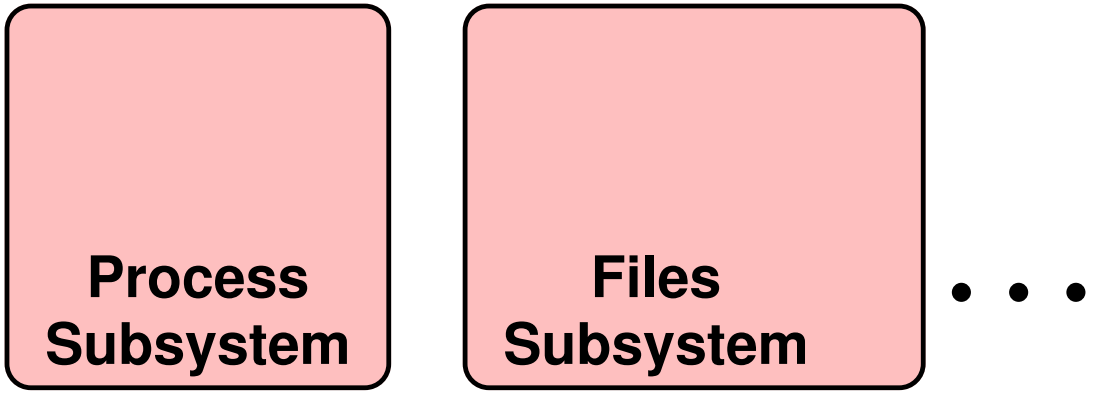
Put It All Together

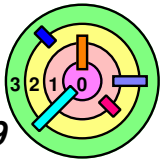



```
int fd;
fd = open("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```

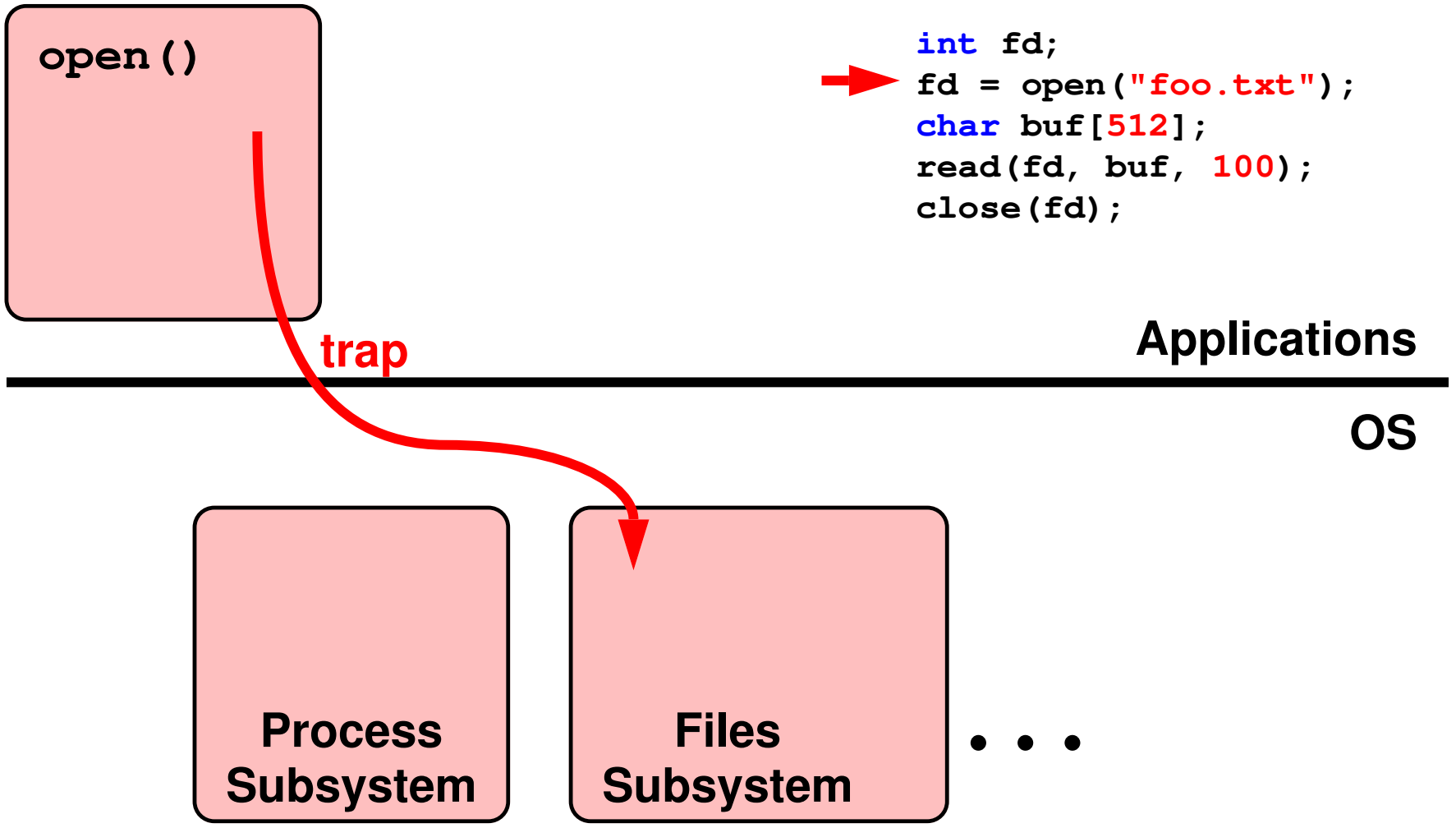
Applications

OS



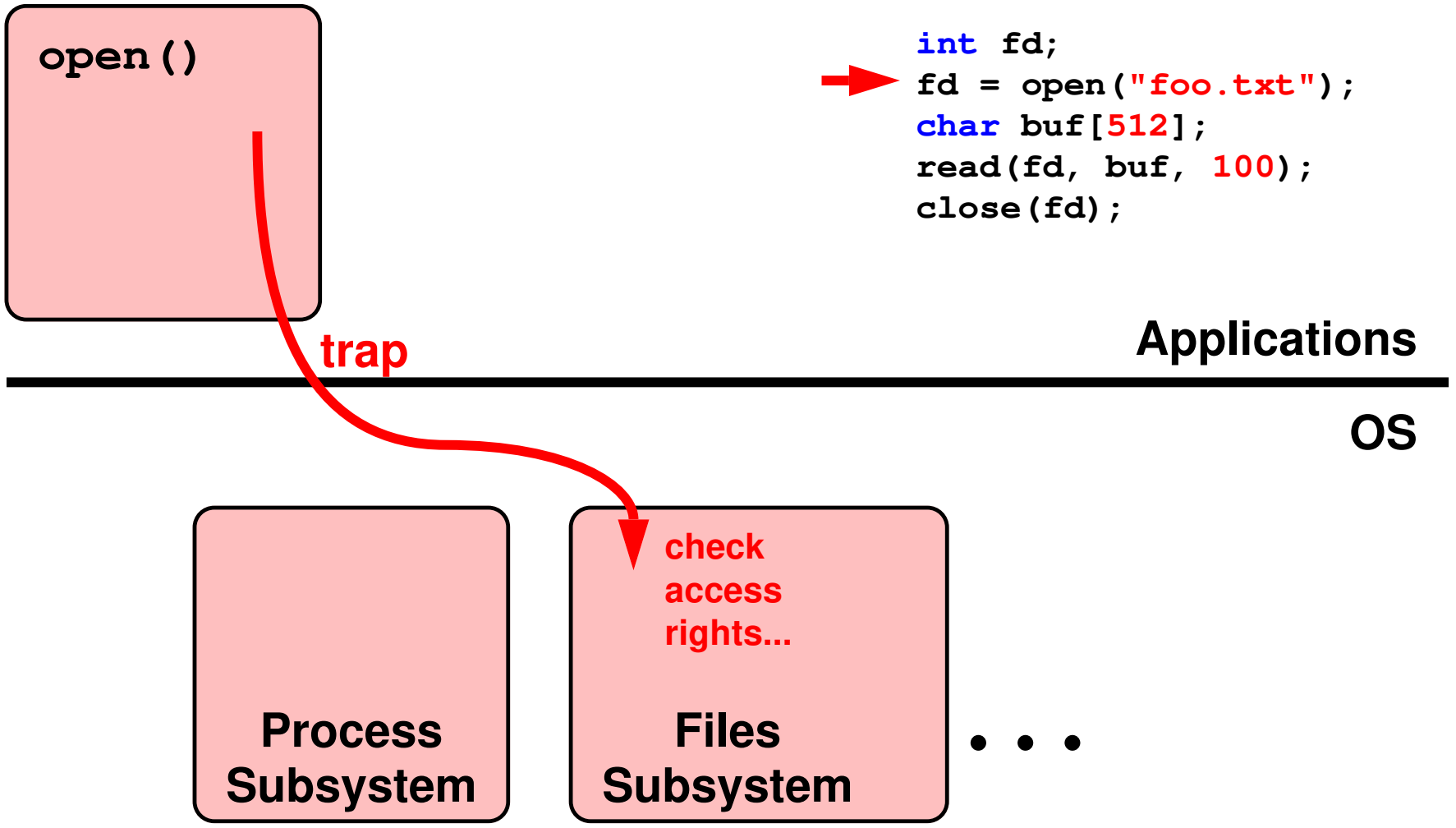


Put It All Together

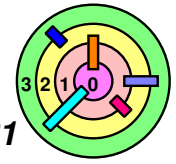



```
int fd;
fd = open("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```

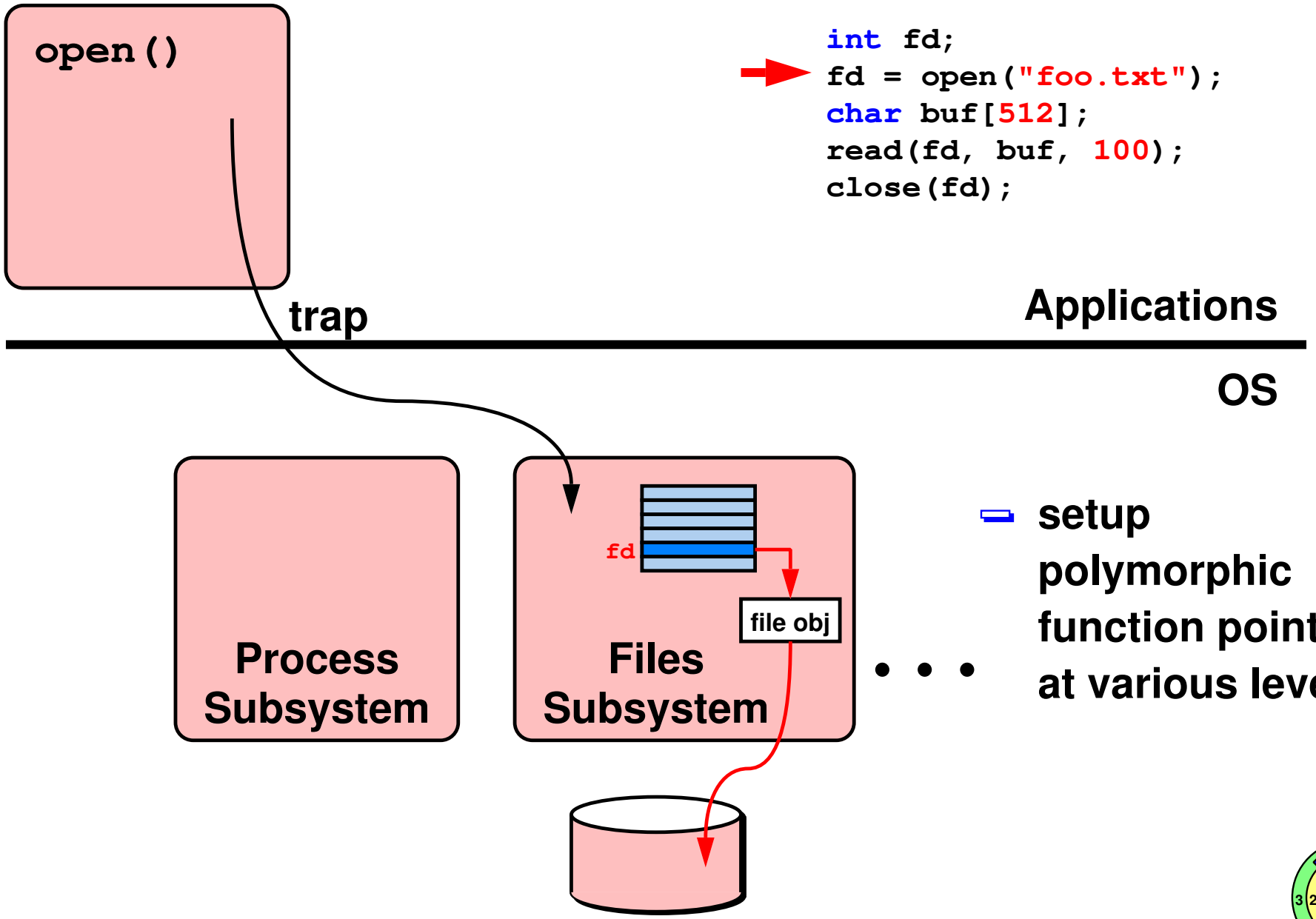
Put It All Together



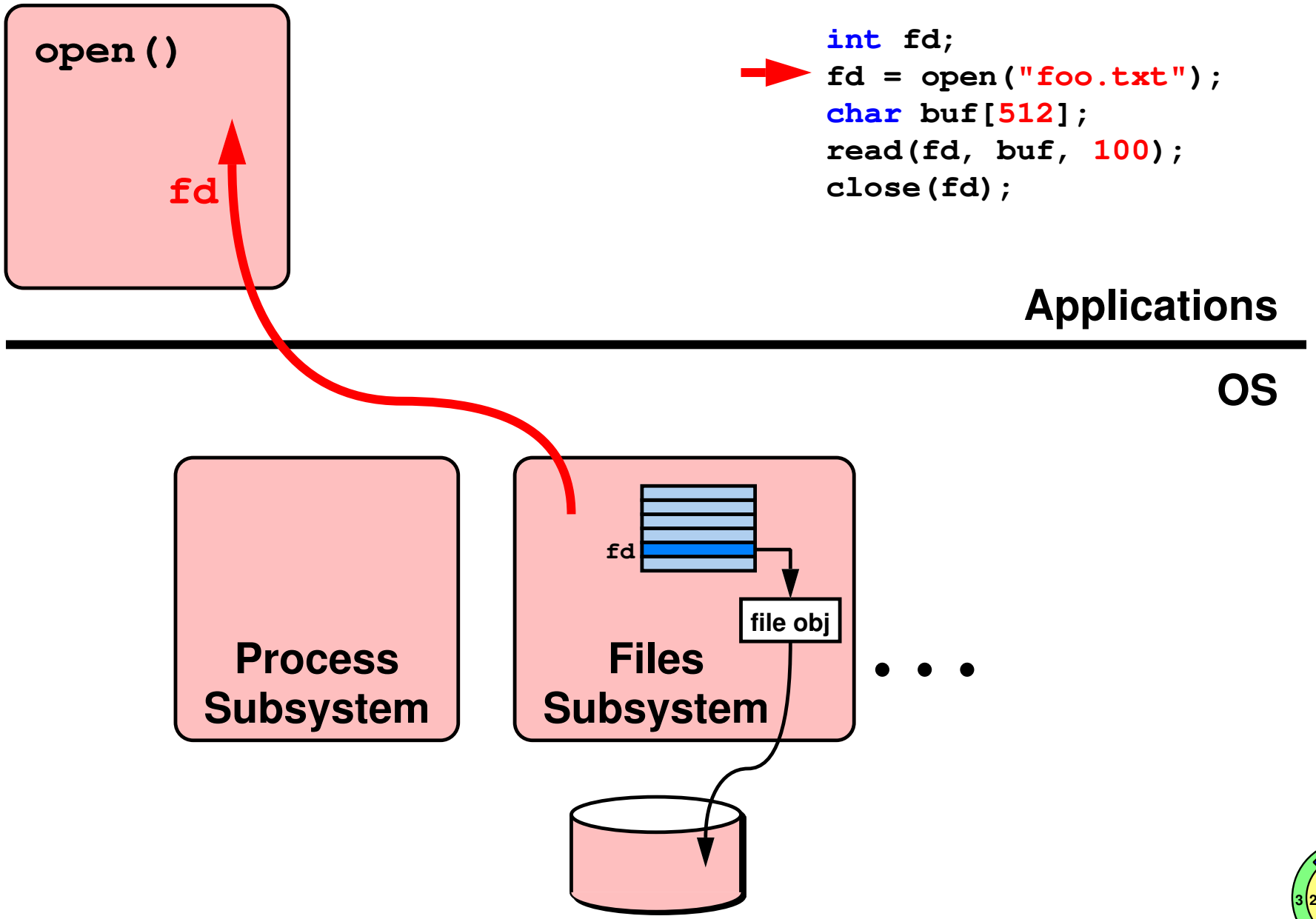

```
int fd;
fd = open("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```



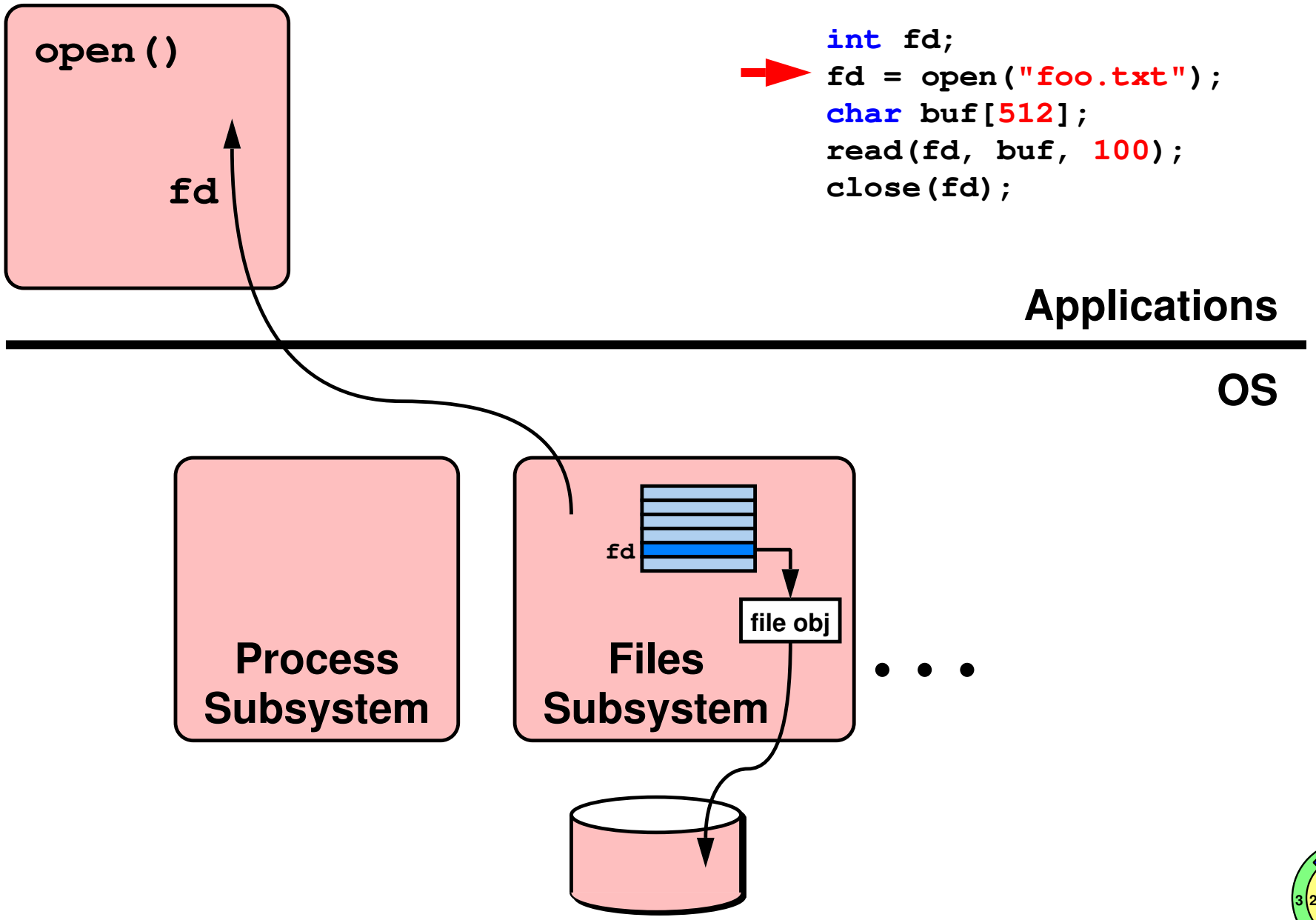
Put It All Together



Put It All Together



Put It All Together



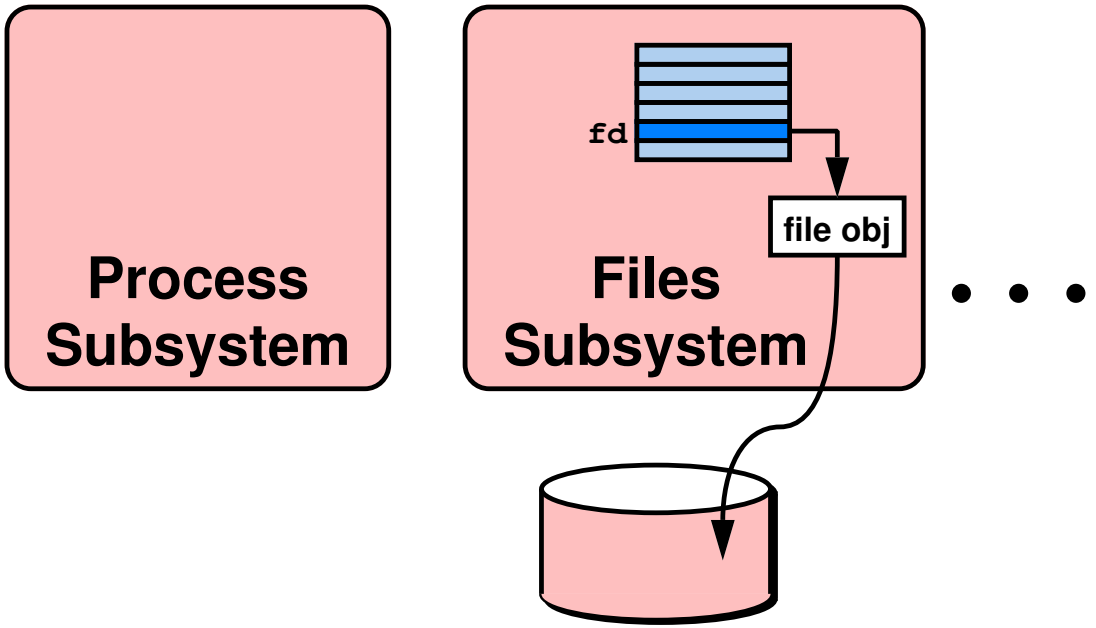
Put It All Together

```
open()
read()
```

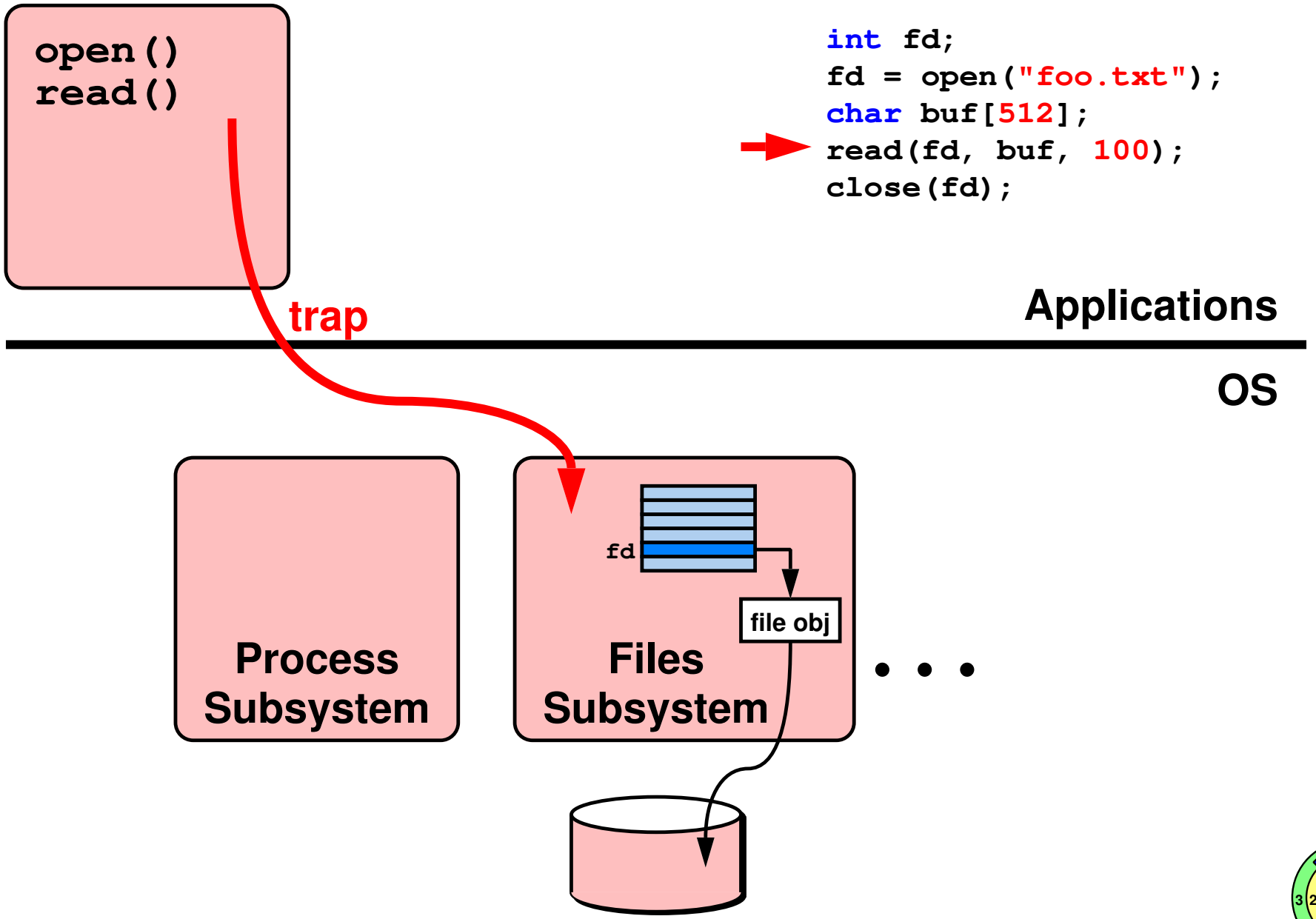
```
int fd;
fd = open("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```

Applications

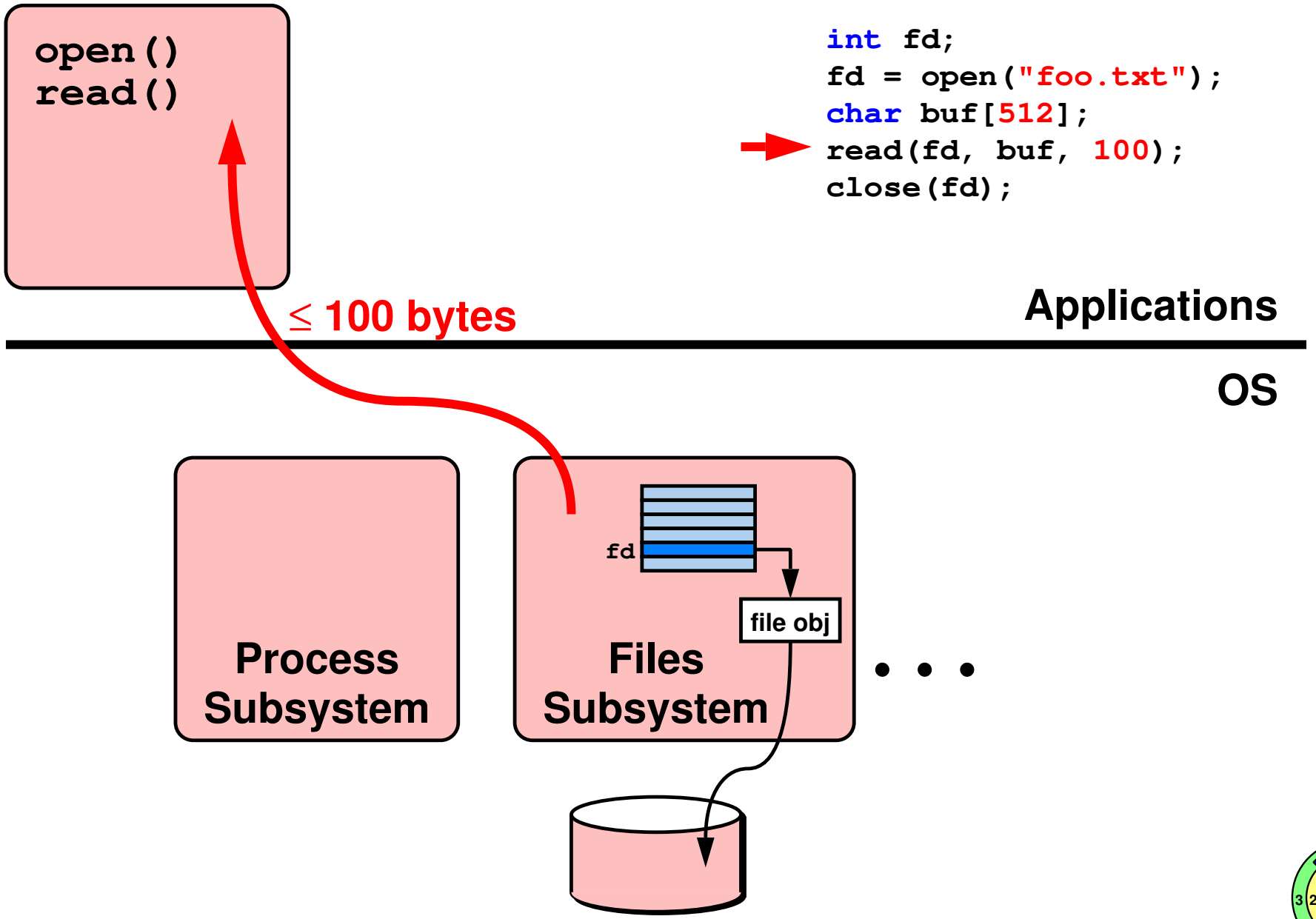
OS



Put It All Together



Put It All Together



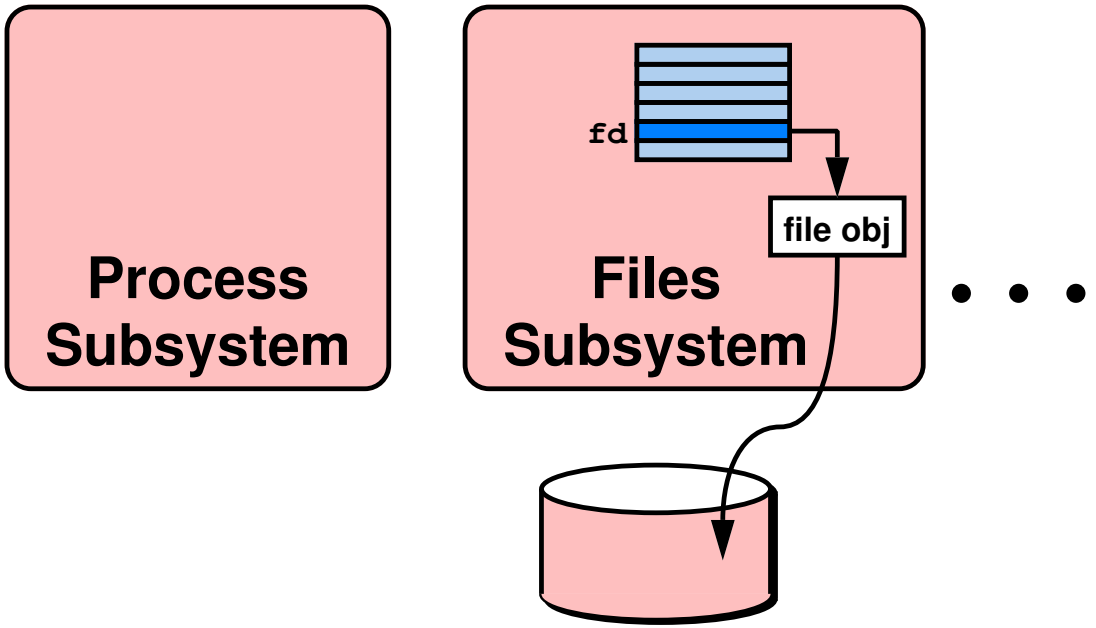
Put It All Together

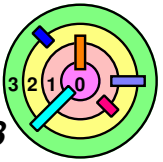
```
open()
read()
close()
```

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

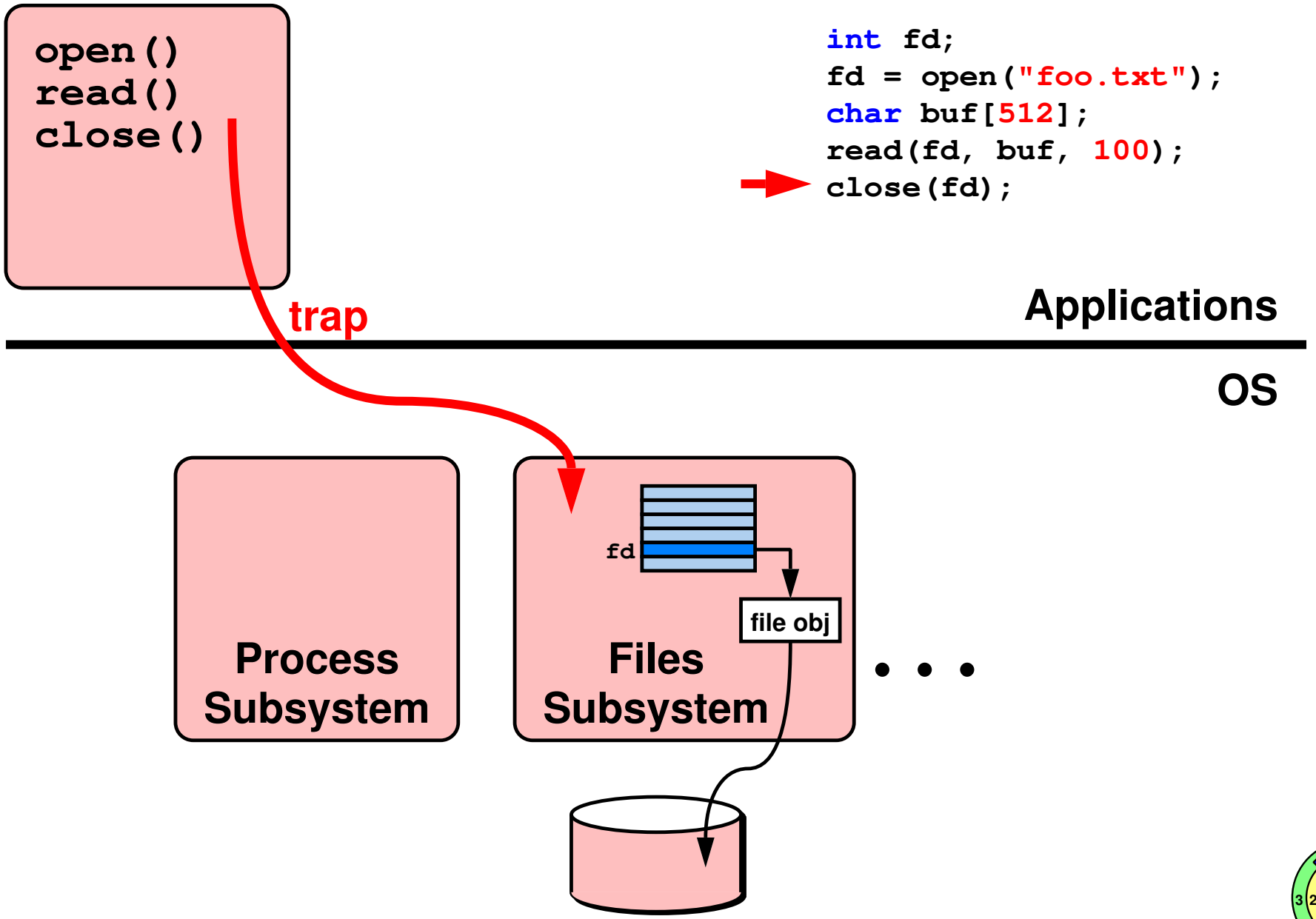
Applications

OS

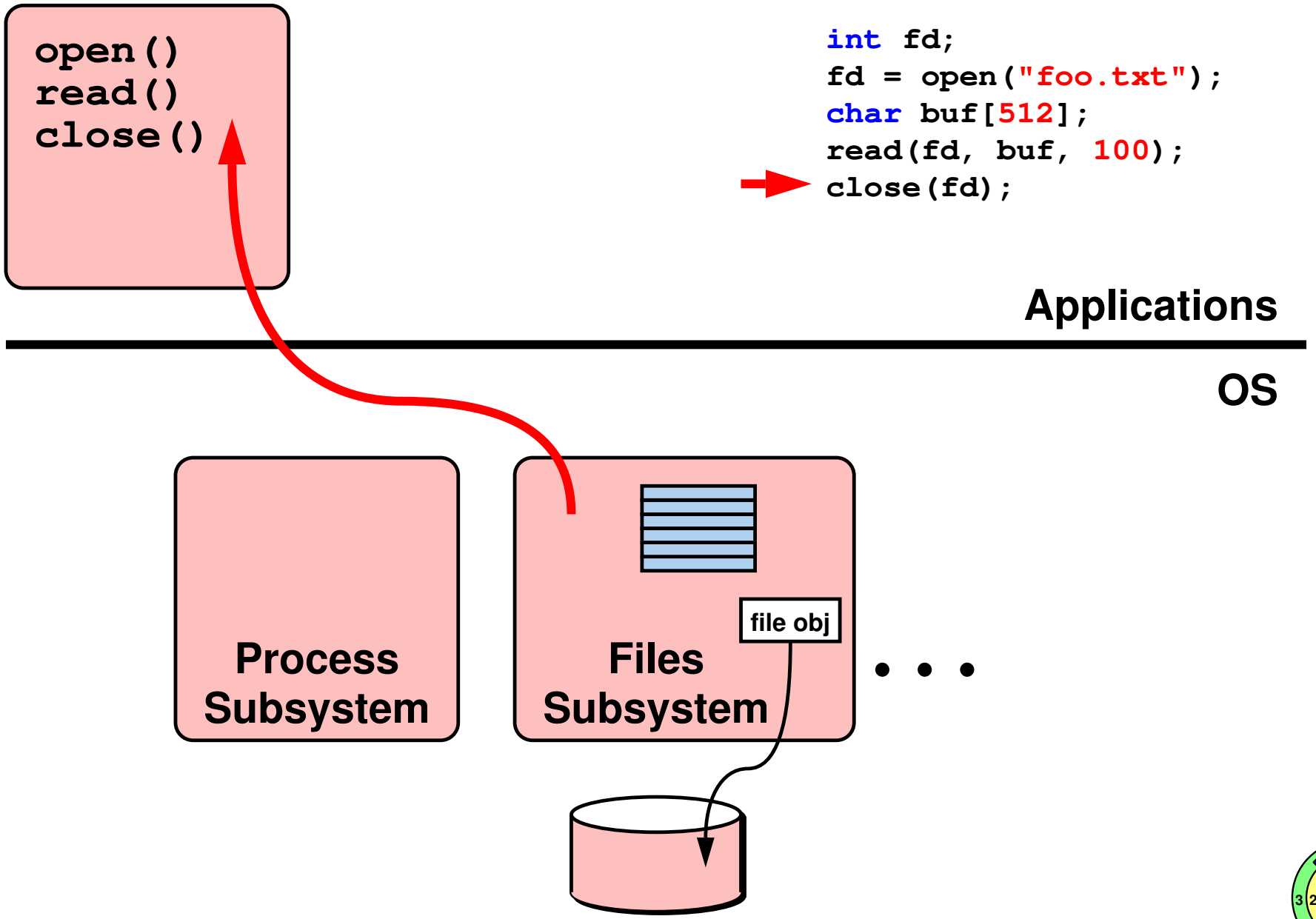




Put It All Together



Put It All Together



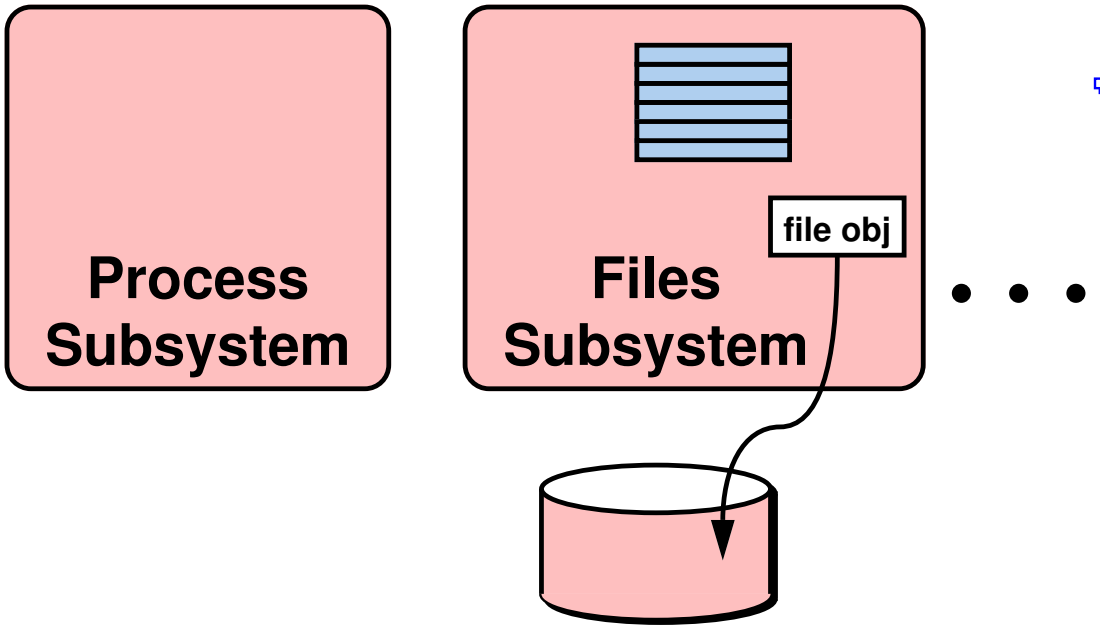
Put It All Together

```
open()
read()
close()
```

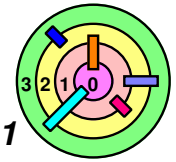
```
int fd;
fd = open("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```

Applications

OS



- file object not deallocated if ref count > 0

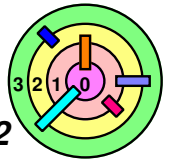


Redirecting Output ... Twice

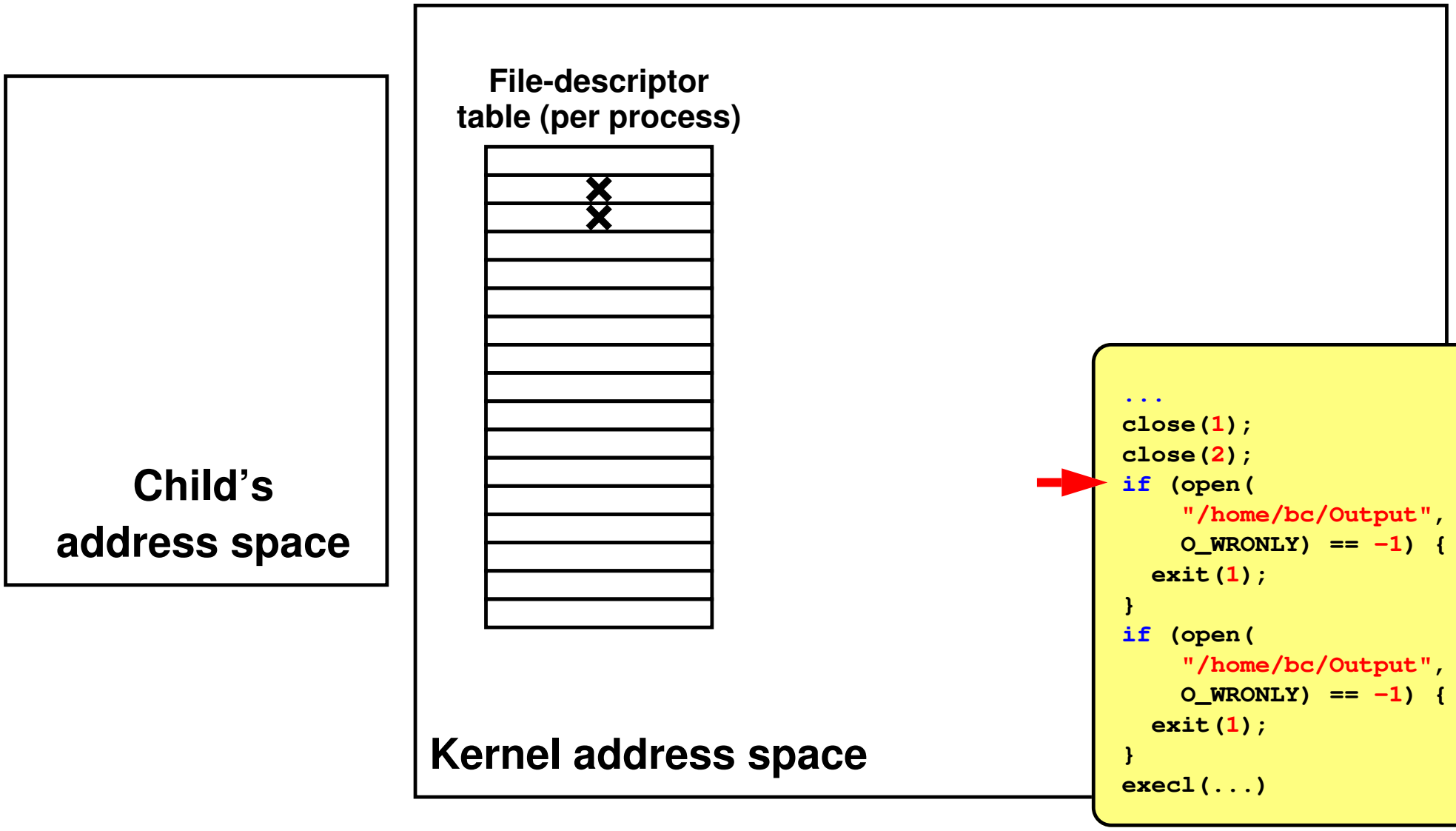
- Every call to `open()` creates a new entry in the system file table

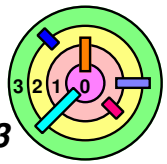
```
if (fork() == 0) {
  /* set up file descriptors 1 and 2 in the child
     process */
  close(1);
  close(2);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    exit(1);
  }
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    exit(1);
  }
  execl("/home/bc/bin/program", "program", 0);
  exit(1);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

  - `stdout` and `stderr` both go into the same file
    - would it cause any problem?

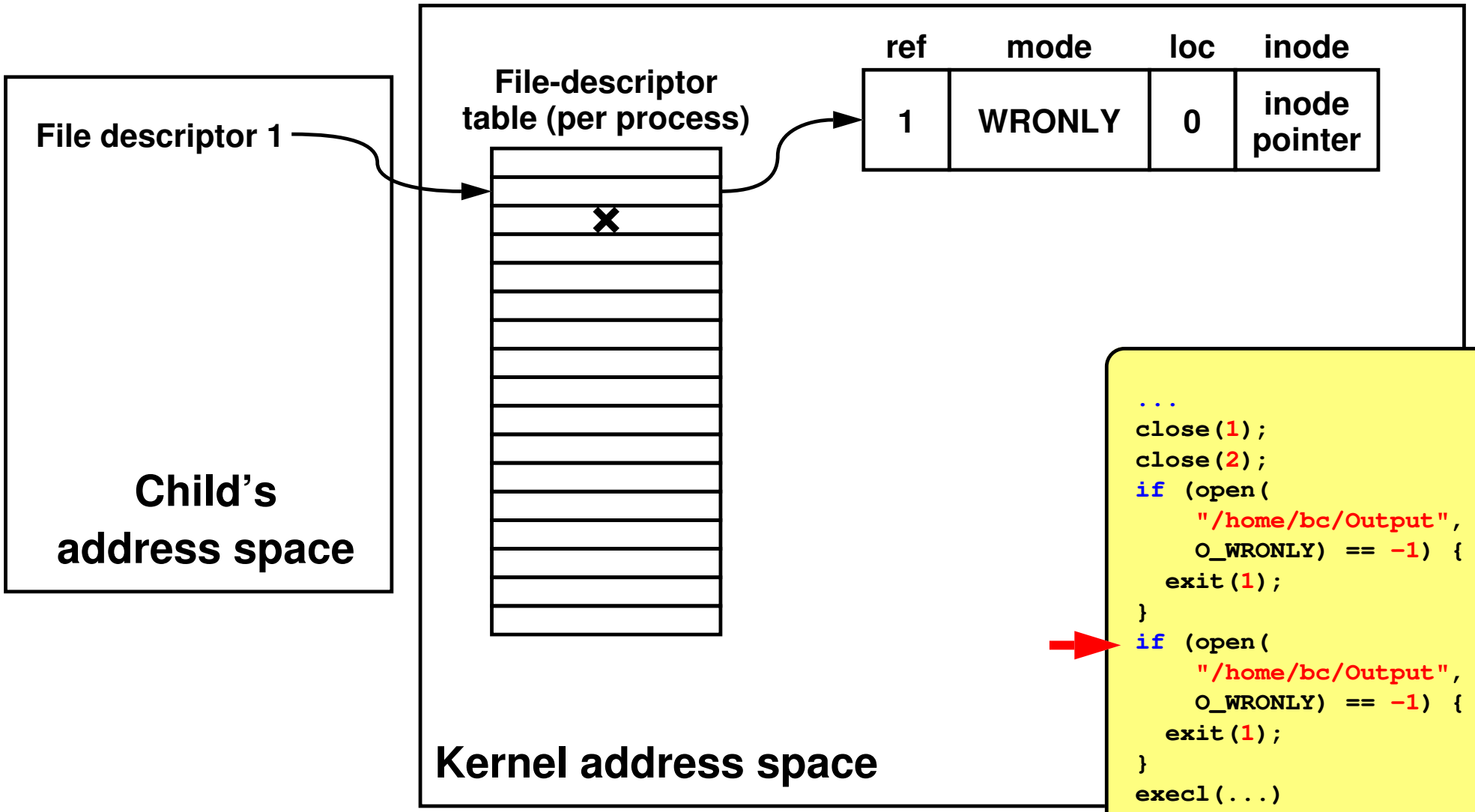


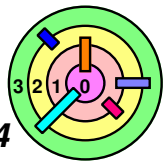
Redirected Output



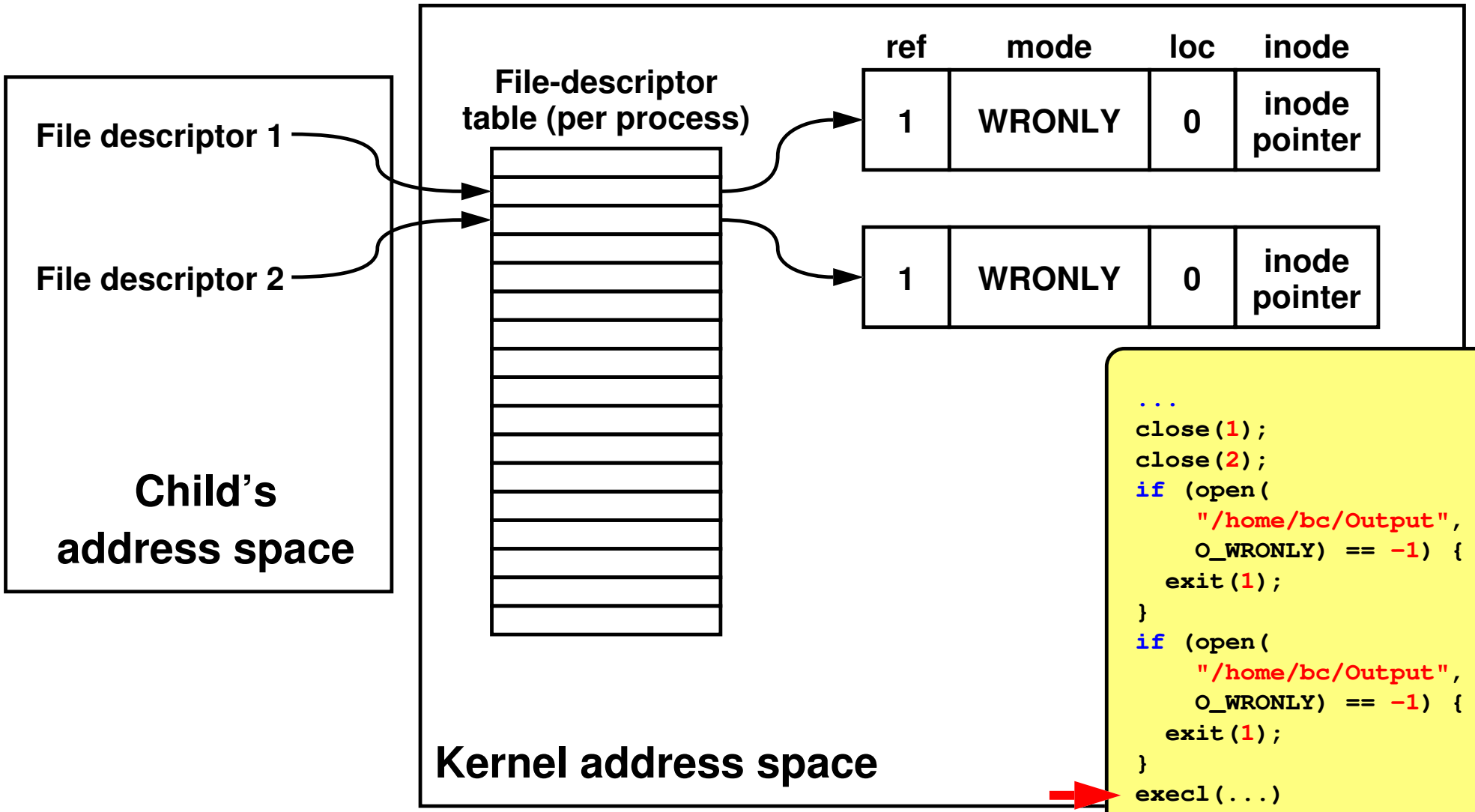


Redirected Output

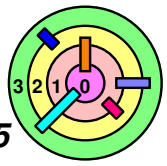




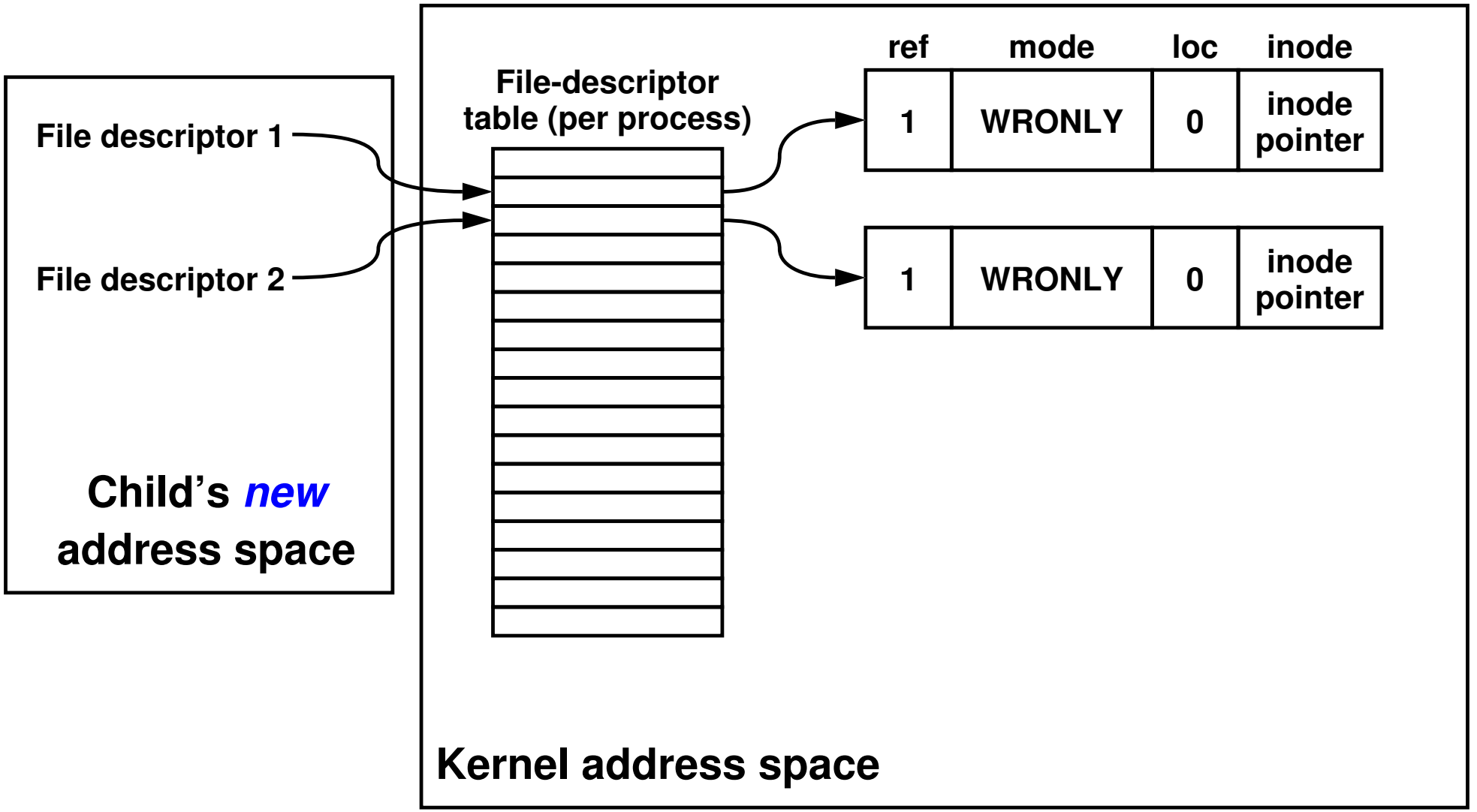
Redirected Output



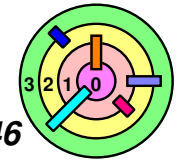

```
...
close(1);
close(2);
if (open(
    "/home/bc/Output",
    O_WRONLY) == -1) {
  exit(1);
}
if (open(
    "/home/bc/Output",
    O_WRONLY) == -1) {
  exit(1);
}
execl(...)
```



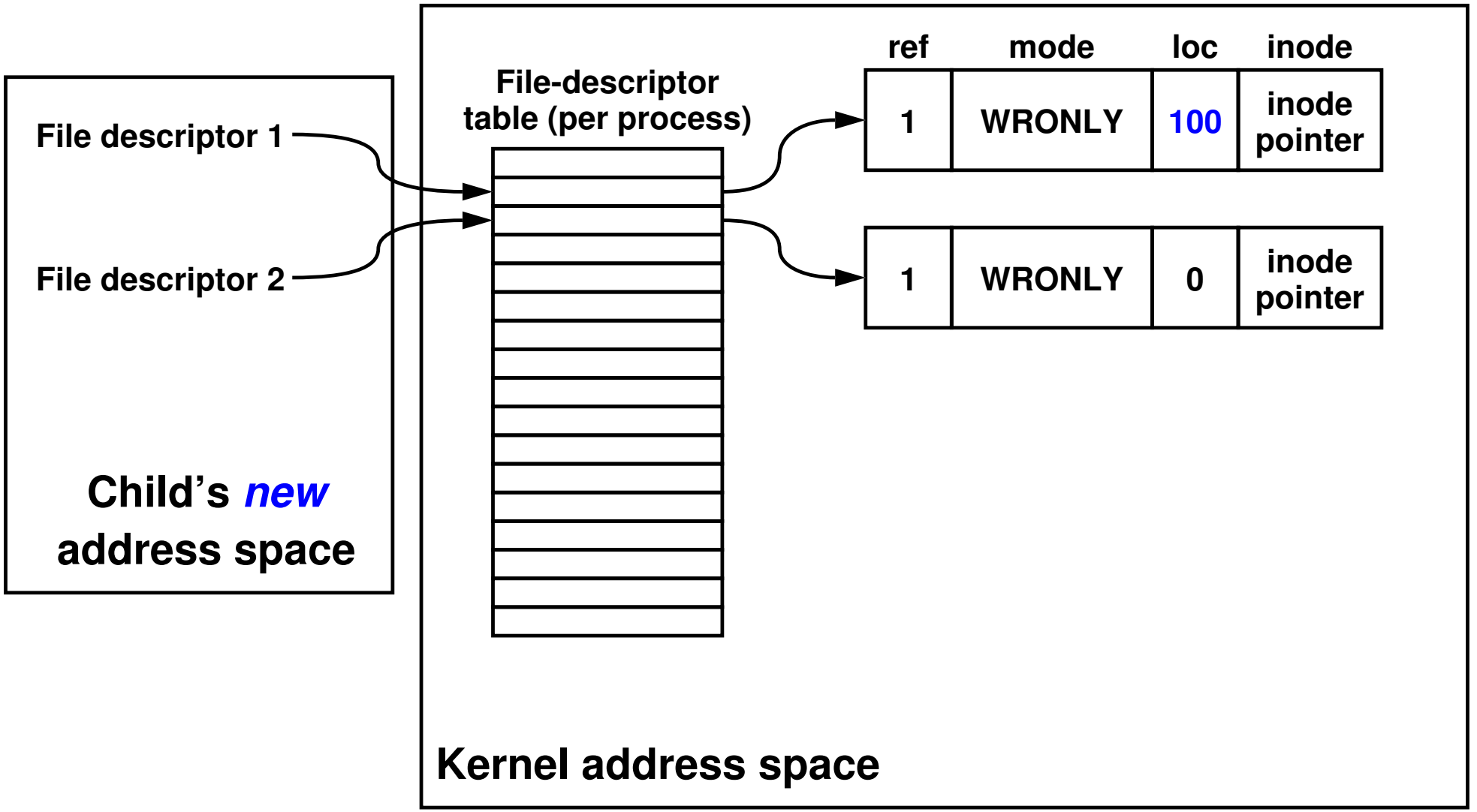
Redirected Output



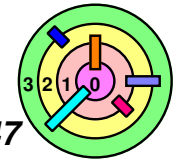
- remember, extended address space survives execs
- let's say we write 100 bytes to `stdout`



Redirected Output After Writing 100 Bytes



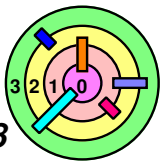
- `write()` to fd=2 will wipe out data in the first 100 bytes
  - that may not be the intent



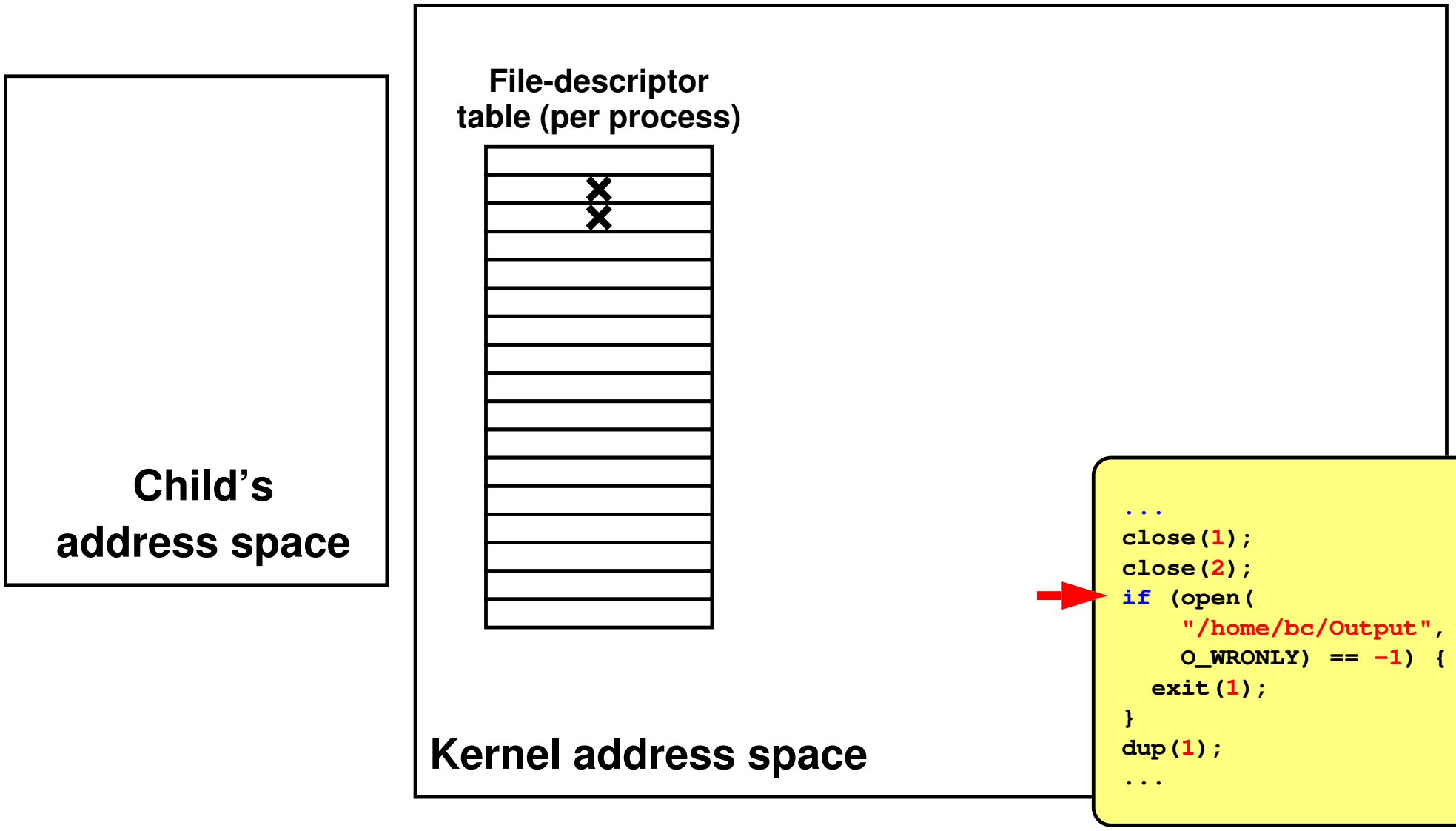
Sharing Context Information

```
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 */
```

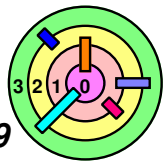
- use the `dup()` system call to ***share*** context information
  - if that's what you want



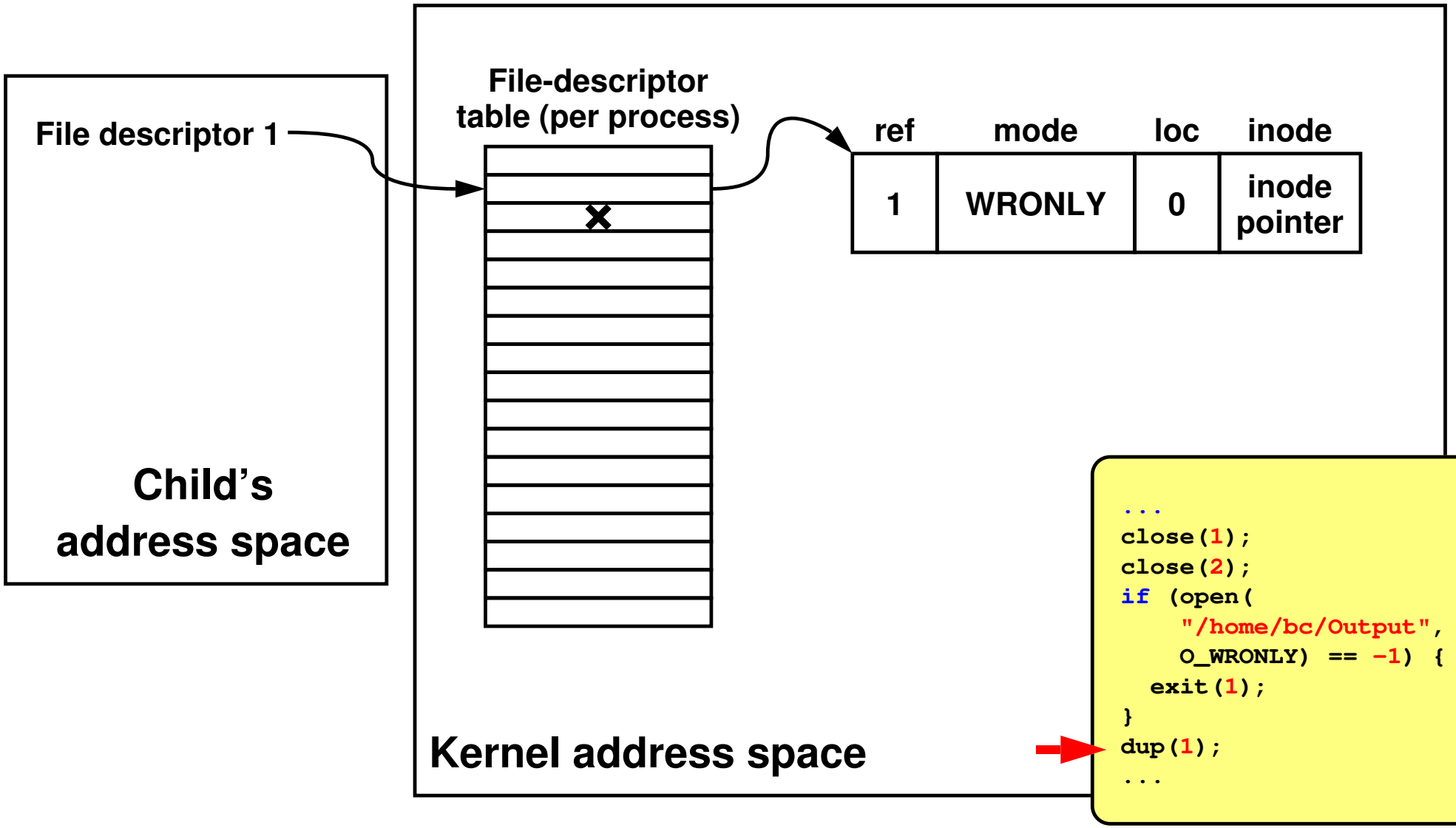
Redirected Output After Dup

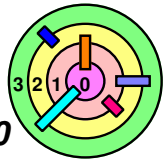



```
...
close(1);
close(2);
if (open(
     "/home/bc/Output",
     O_WRONLY) == -1) {
  exit(1);
}
dup(1);
...
```

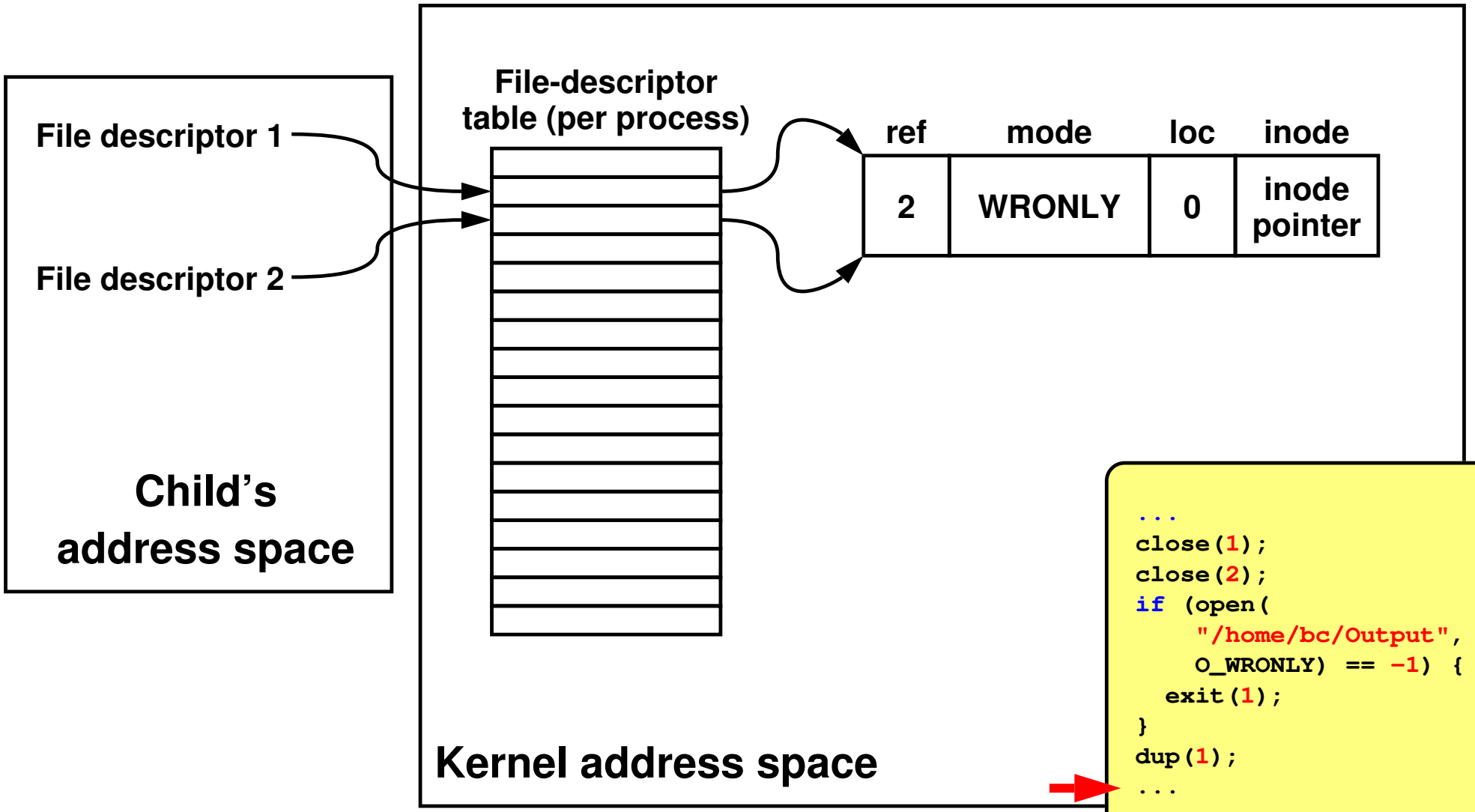


Redirected Output After Dup

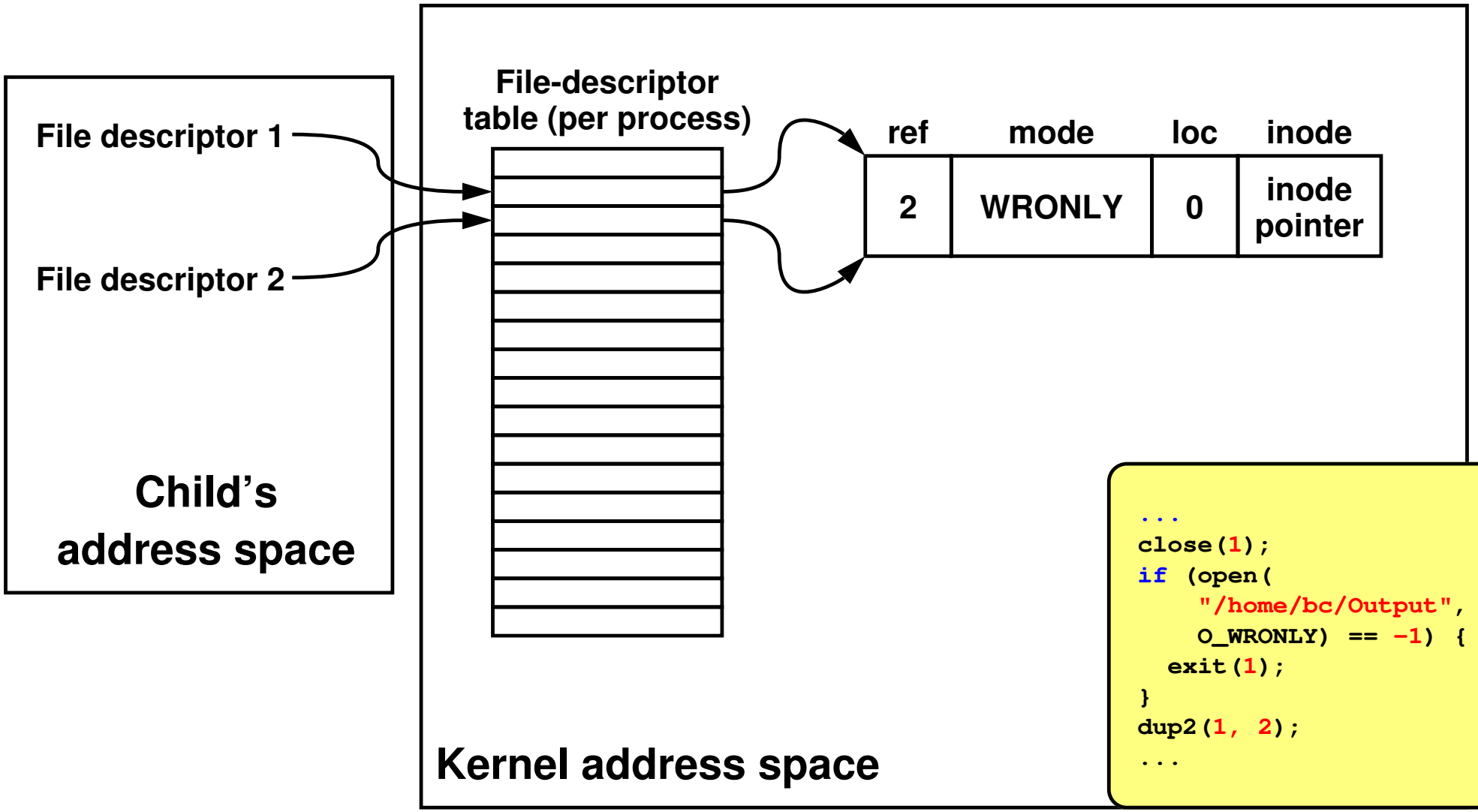




Redirected Output After Dup

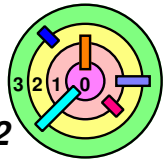


Redirected Output After Dup




```
...
close(1);
if (open(
     "/home/bc/Output",
     O_WRONLY) == -1) {
  exit(1);
}
dup2(1, 2);
...
```

- `dup2(oldfd,newfd)` specifies a new file descriptor (if `newfd` is currently open, close it first)

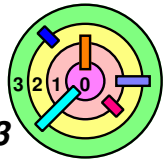


Parent And Child Communicate Over A File

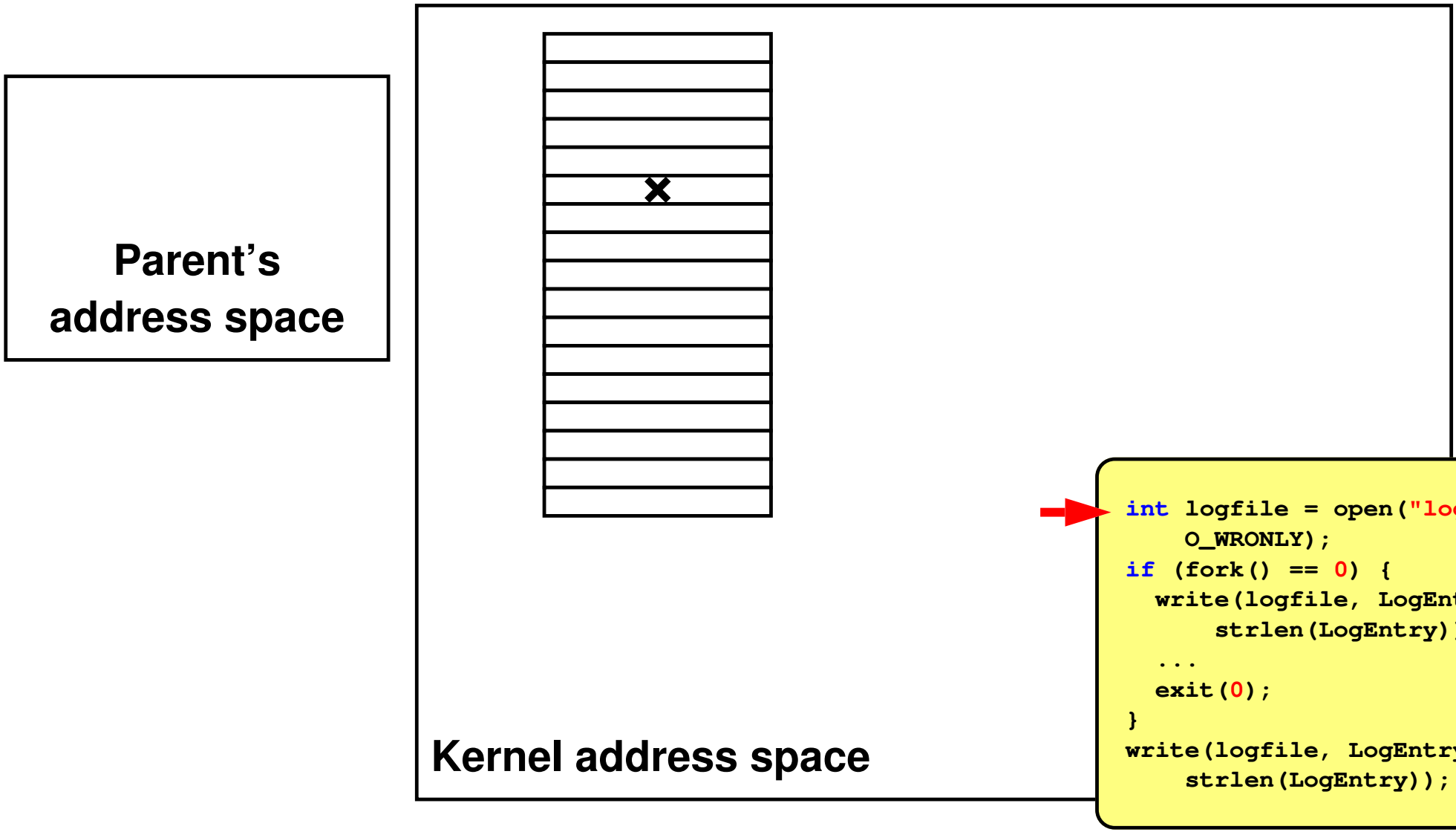
- It would be useful to be able to share file context information with a child process
  - when `fork()` is called, the child process gets a *copy* of the parent's file descriptor table

```
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()`

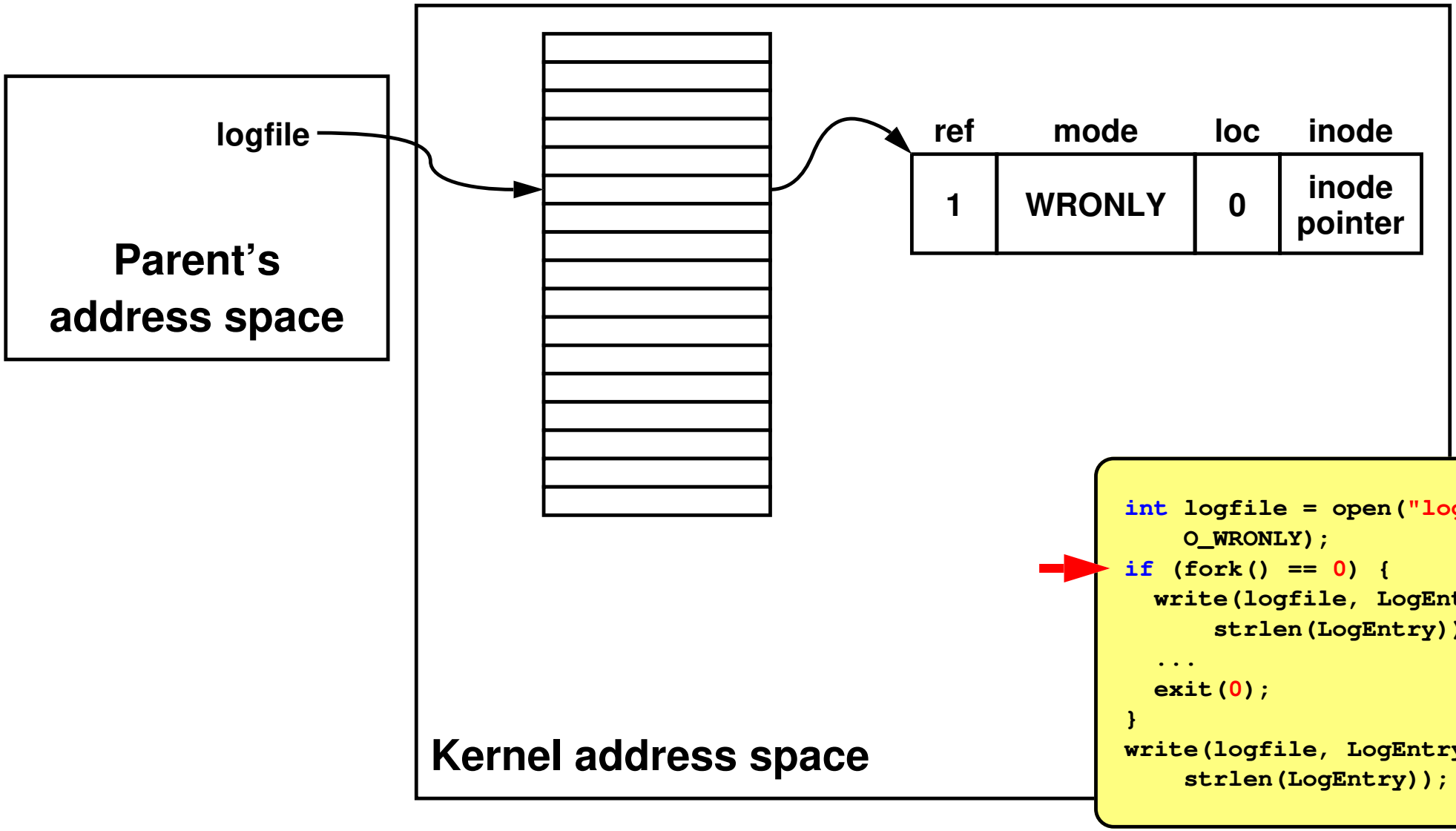


Parent And Child Communicate Over A File



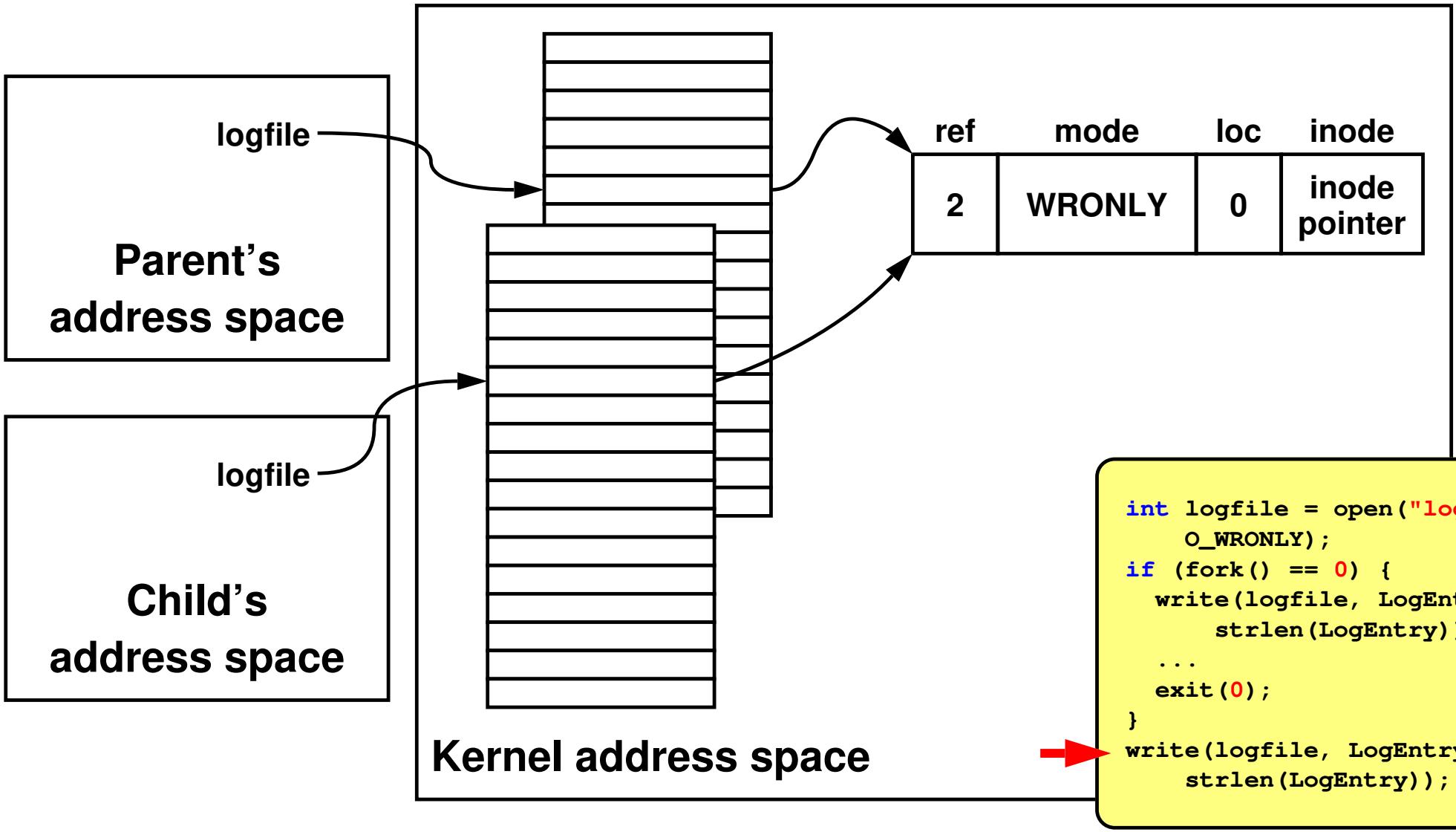
- parent and child processes get *separate* file descriptor table but *share* extended address space

Parent And Child Communicate Over A File



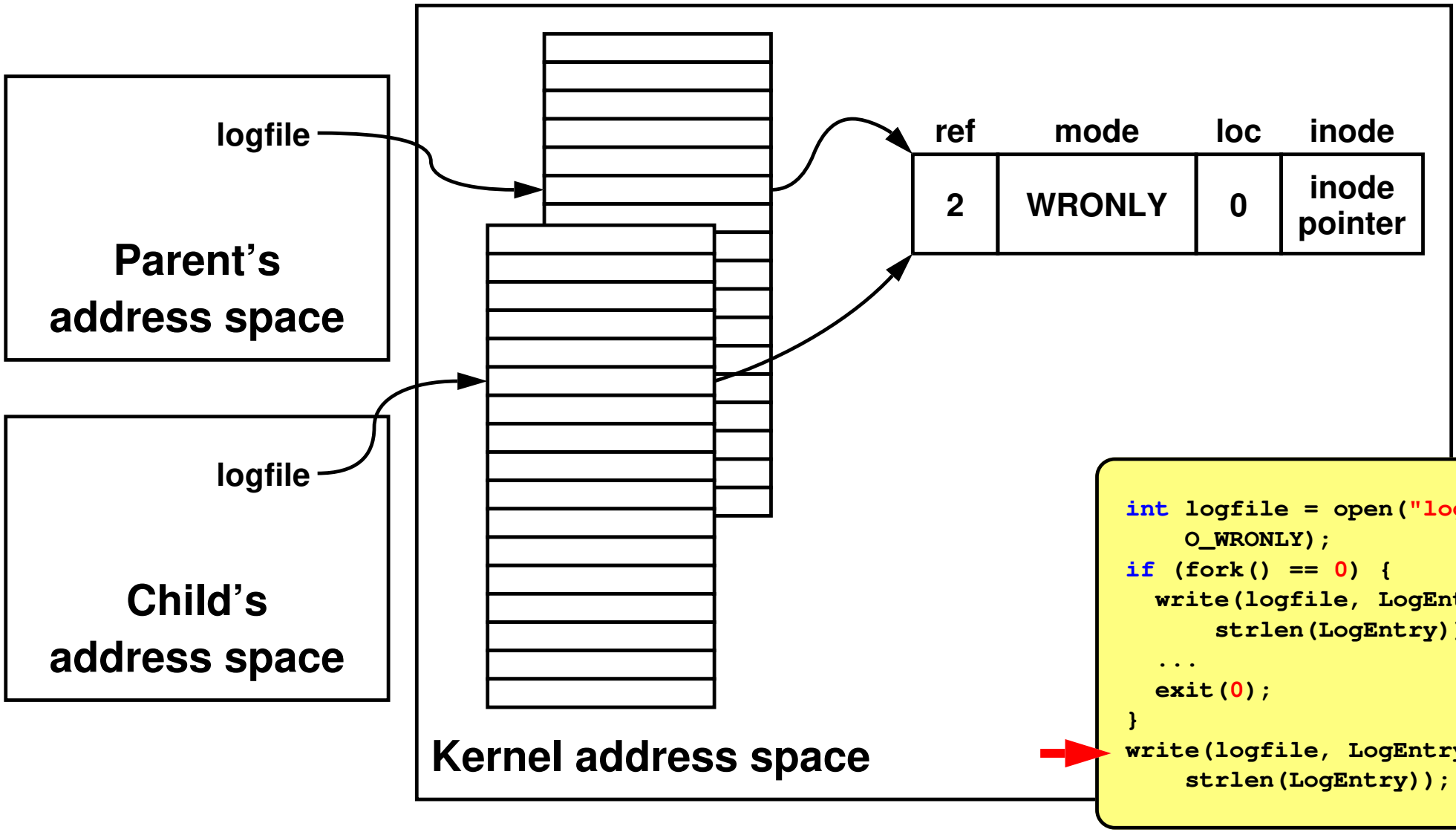
- parent and child processes get *separate* file descriptor table but *share* extended address space

Parent And Child Communicate Over A File

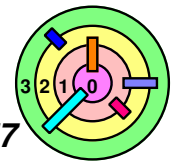


- parent and child processes get *separate* file descriptor table but *share* extended address space *indirectly*

Parent And Child Communicate Over A File

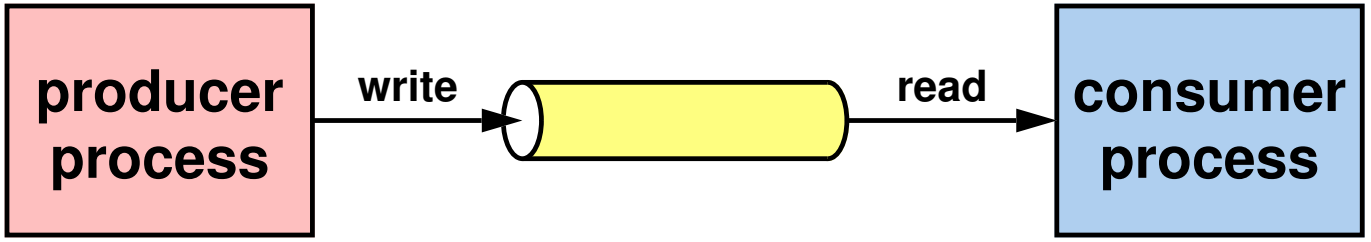


- parent and child processes can communicate using such a shared file descriptor, although *difficult to synchronize*



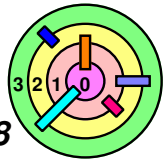
Interprocess Communication With Pipes

- Pipes: a UNIX pipe is a kernel buffer with two file descriptors, one for writing and one for reading



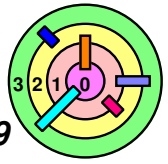
  - the producer process behaves as if it has a file descriptor to a file that has been opened for writing
  - the consumer process behaves as if it has a file descriptor to a file that has been opened for reading

- 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

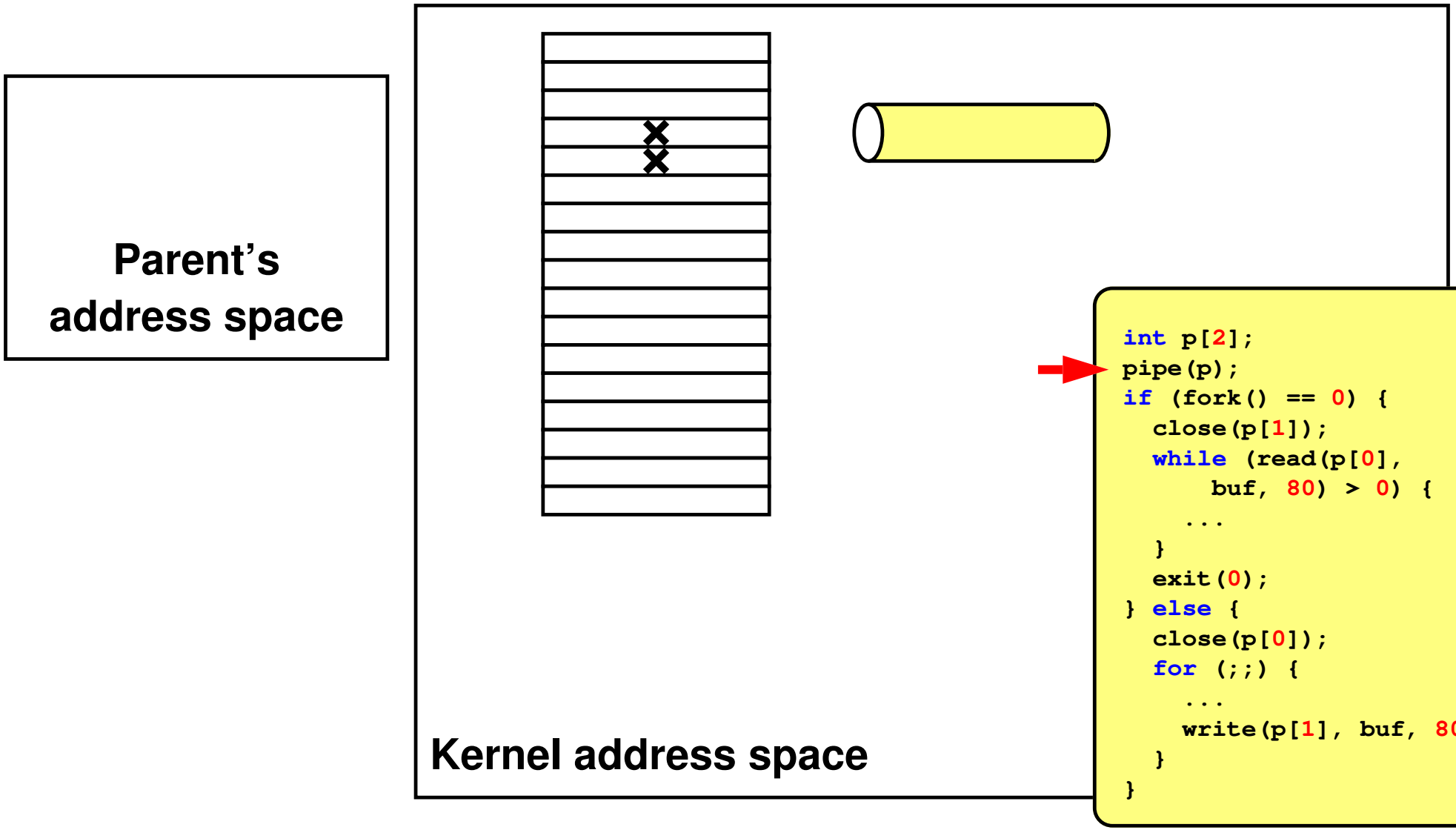


Pipes

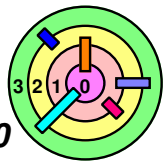
```
int p[2]; // array to hold pipe's file descriptors
pipe(p);  // creates a pipe, assume no errors
  // p[0] refers to the read/output end of the pipe
  // p[1] refers to the write/input end of the pipe
if (fork() == 0) {
  char buf[80];
  close(p[1]); // not needed by the child
  while (read(p[0],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



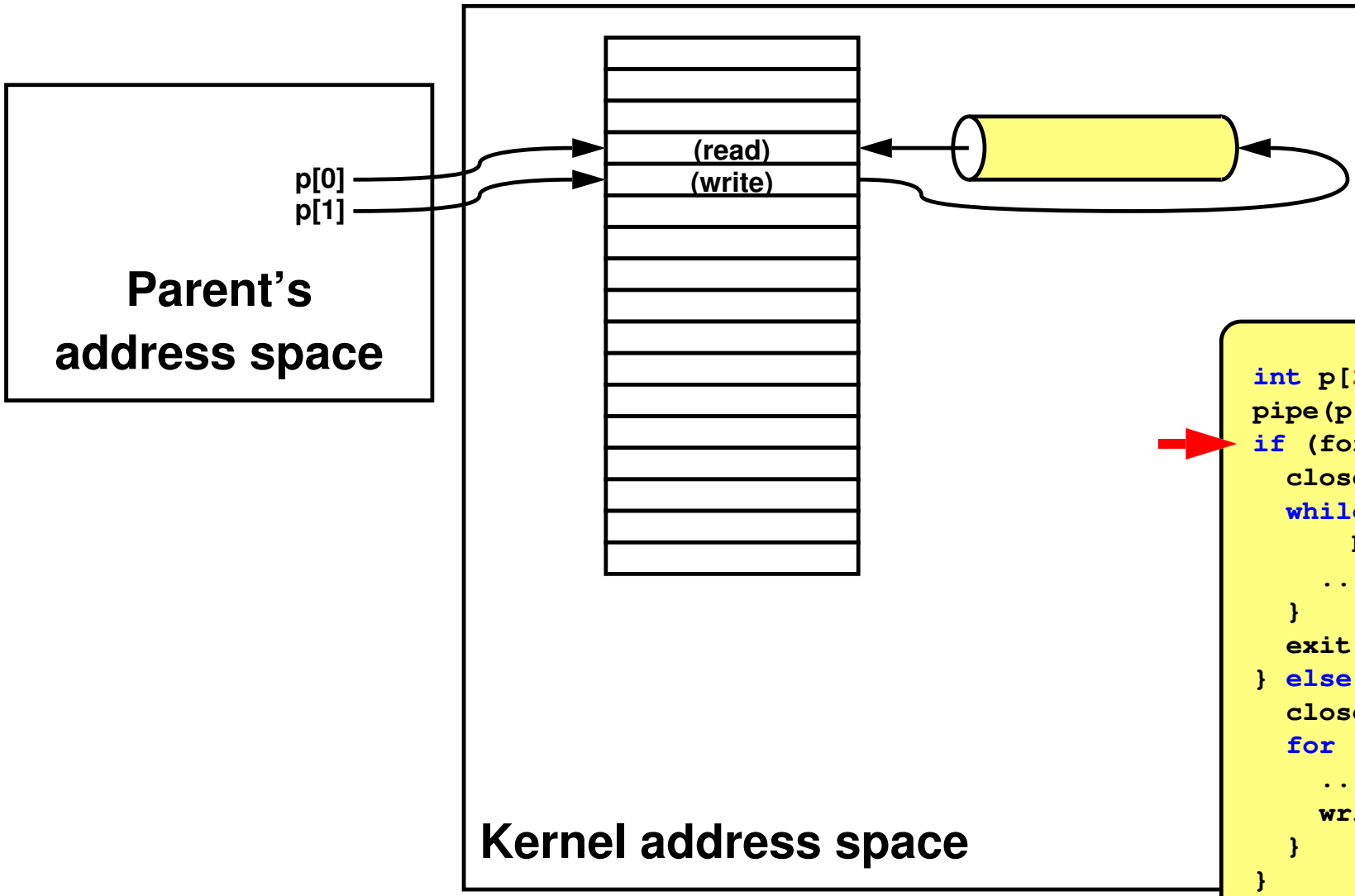
Pipes



- parent creates a pipe object in the kernel

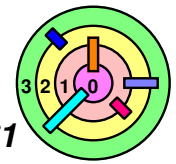


Pipes

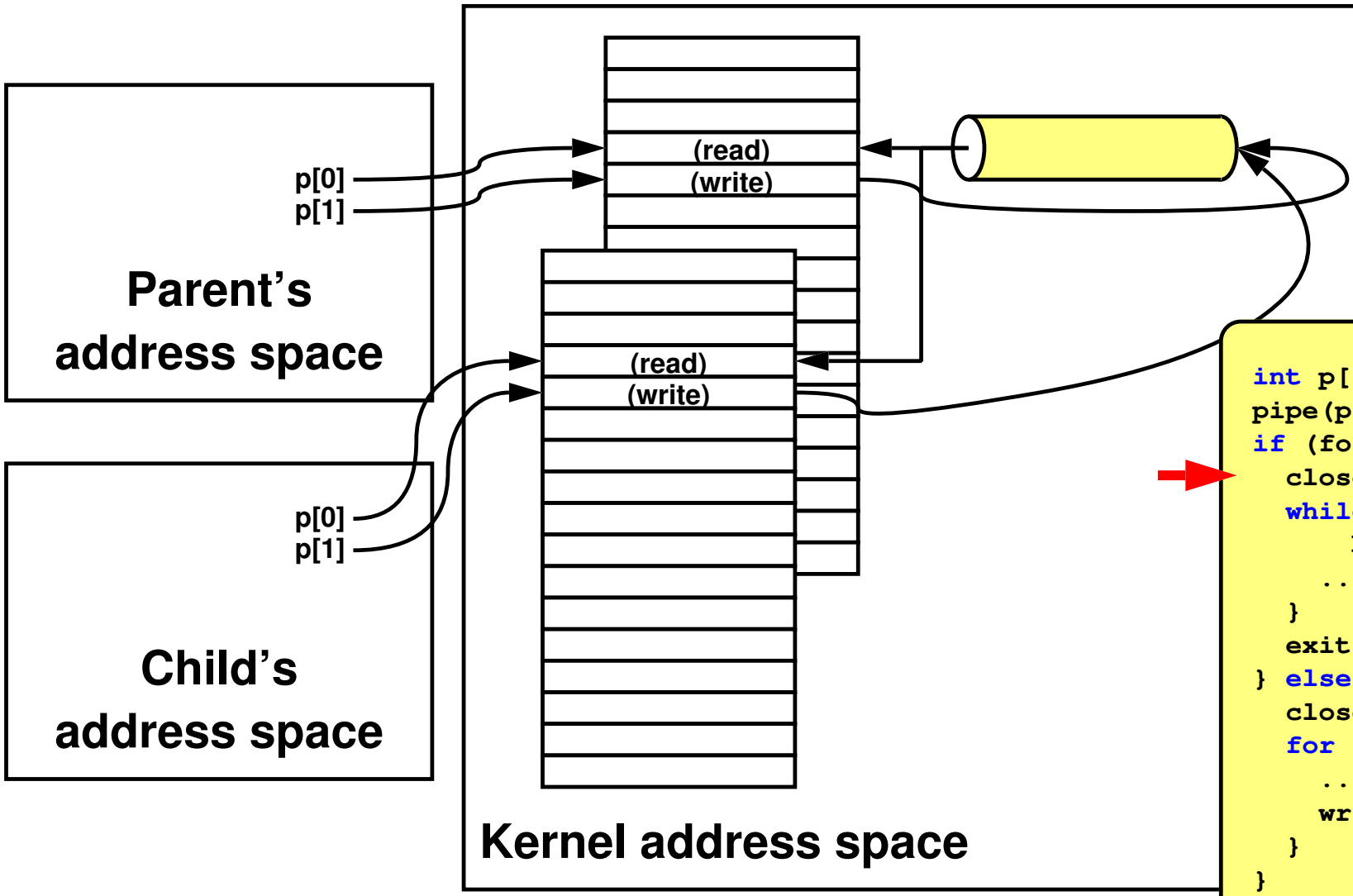



```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

- parent creates a pipe object in the kernel

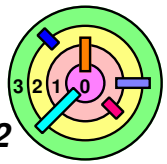


Pipes

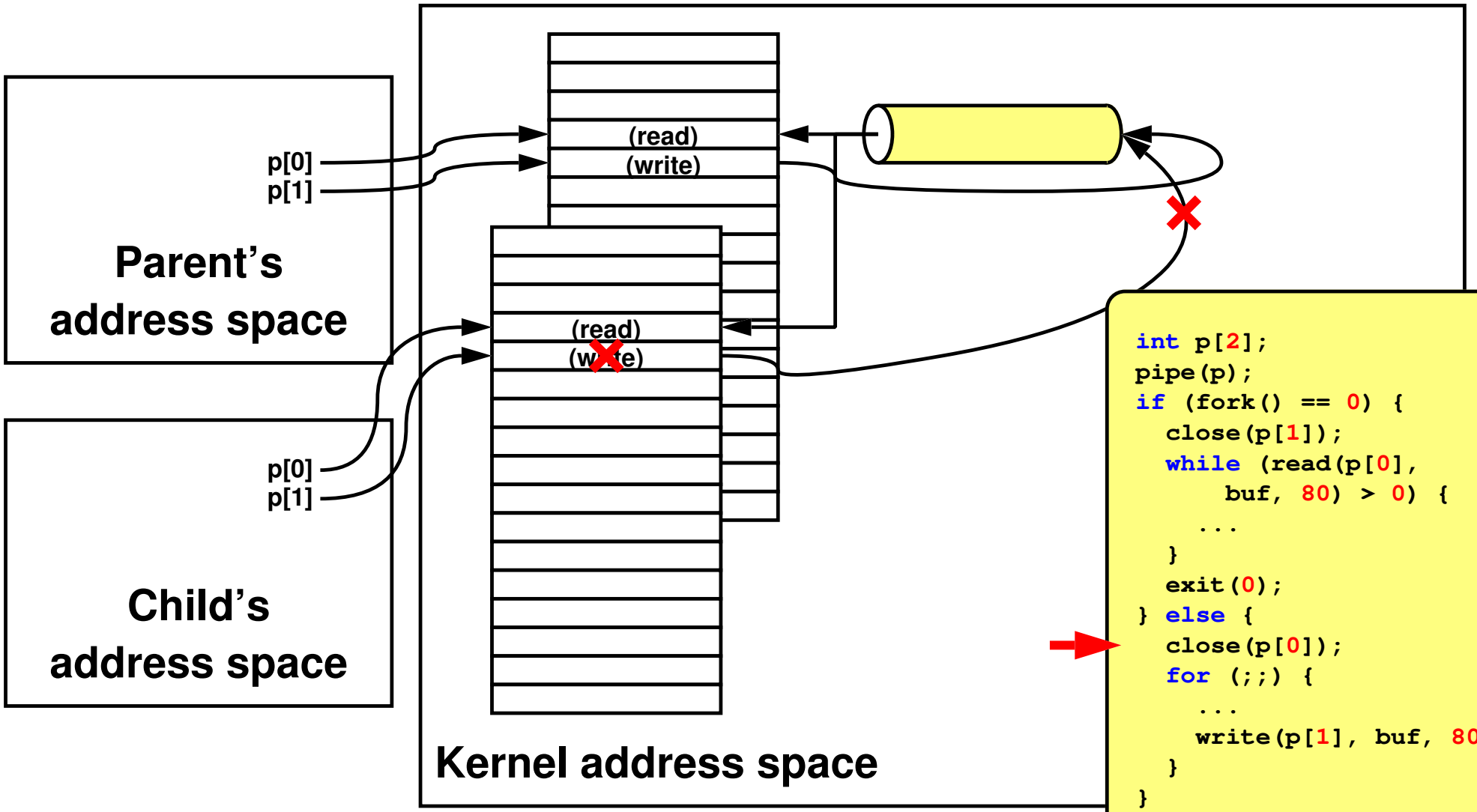



```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

- child processes gets a *copy* of the parent's file descriptor table and end up sharing the pipe object

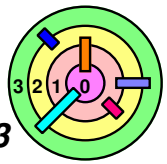


Pipes

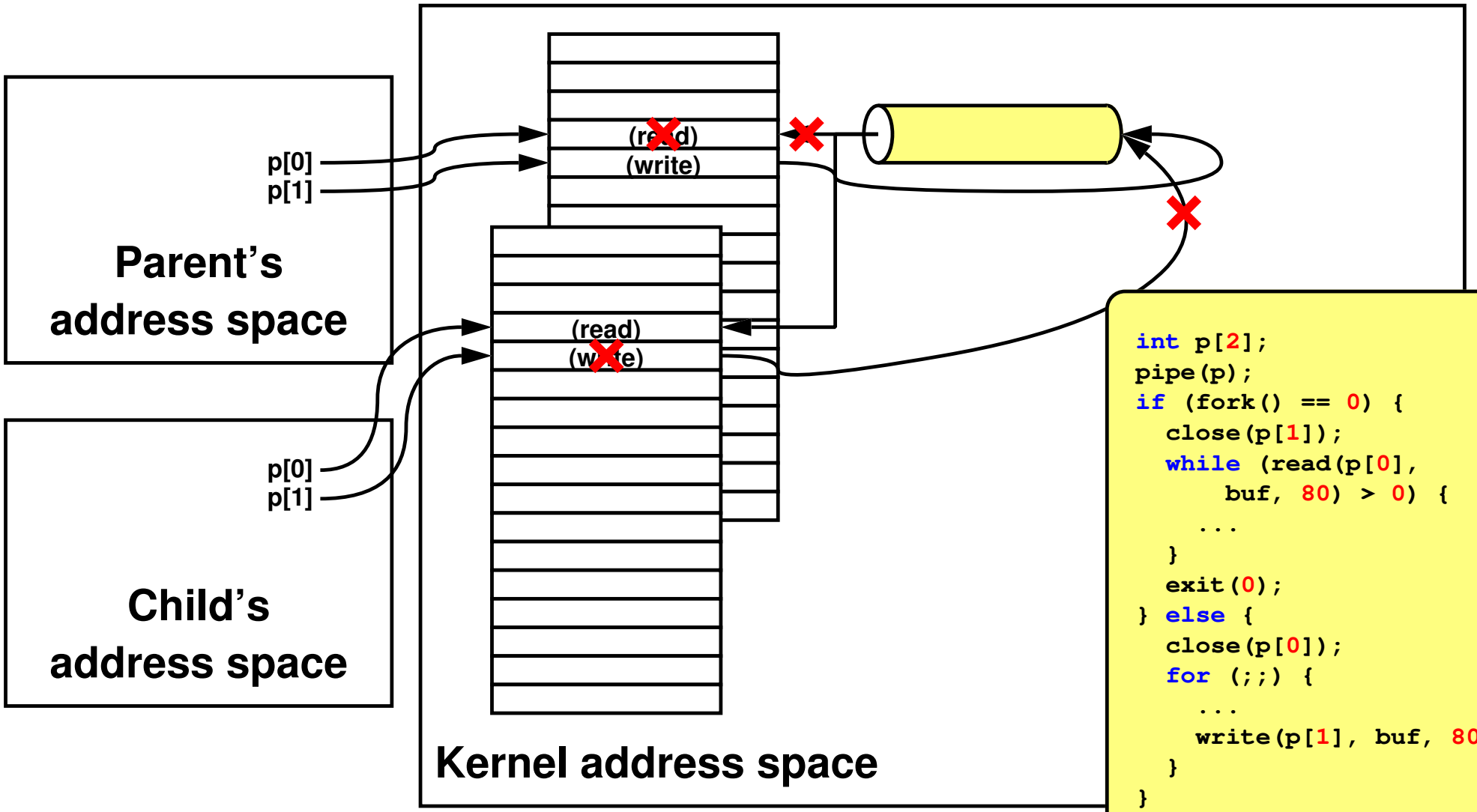



```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

- if the child just wants to read from the pipe, it must close the write-end of the pipe

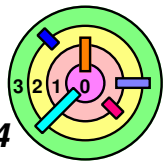


Pipes

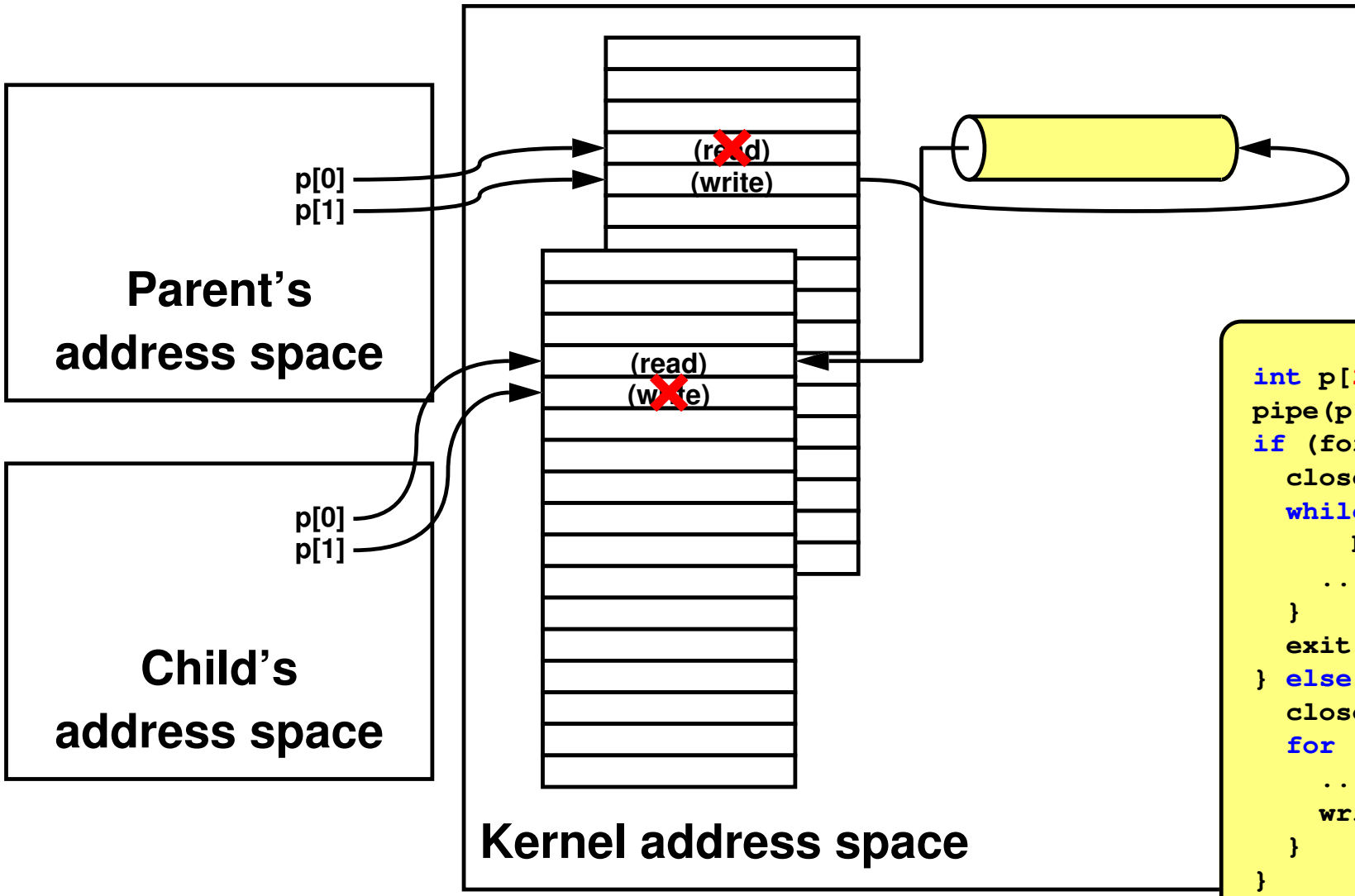



```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

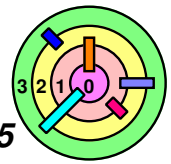
- if the parent just wants to write into the pipe, it must close the read-end of the pipe



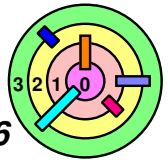
Pipes




```
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);
  }
}
```



(3.3) Case Study: Implementing a Shell

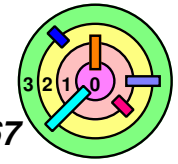


(3.3) Implementing a Shell

- A basic shell:

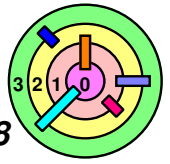
```
char *prog, **args;

while (readAndParseCmdLine(&prog, &args)) {
  int child_pid = fork();
  if (child_pid == 0) {
    execv(prog, args);
    exit(1);
  } else {
    while(child_pid != wait(0))
      ;
  }
}
```

- A program can send its output to a file (using I/O redirection)
- A program can read its input from a file (using I/O redirection)



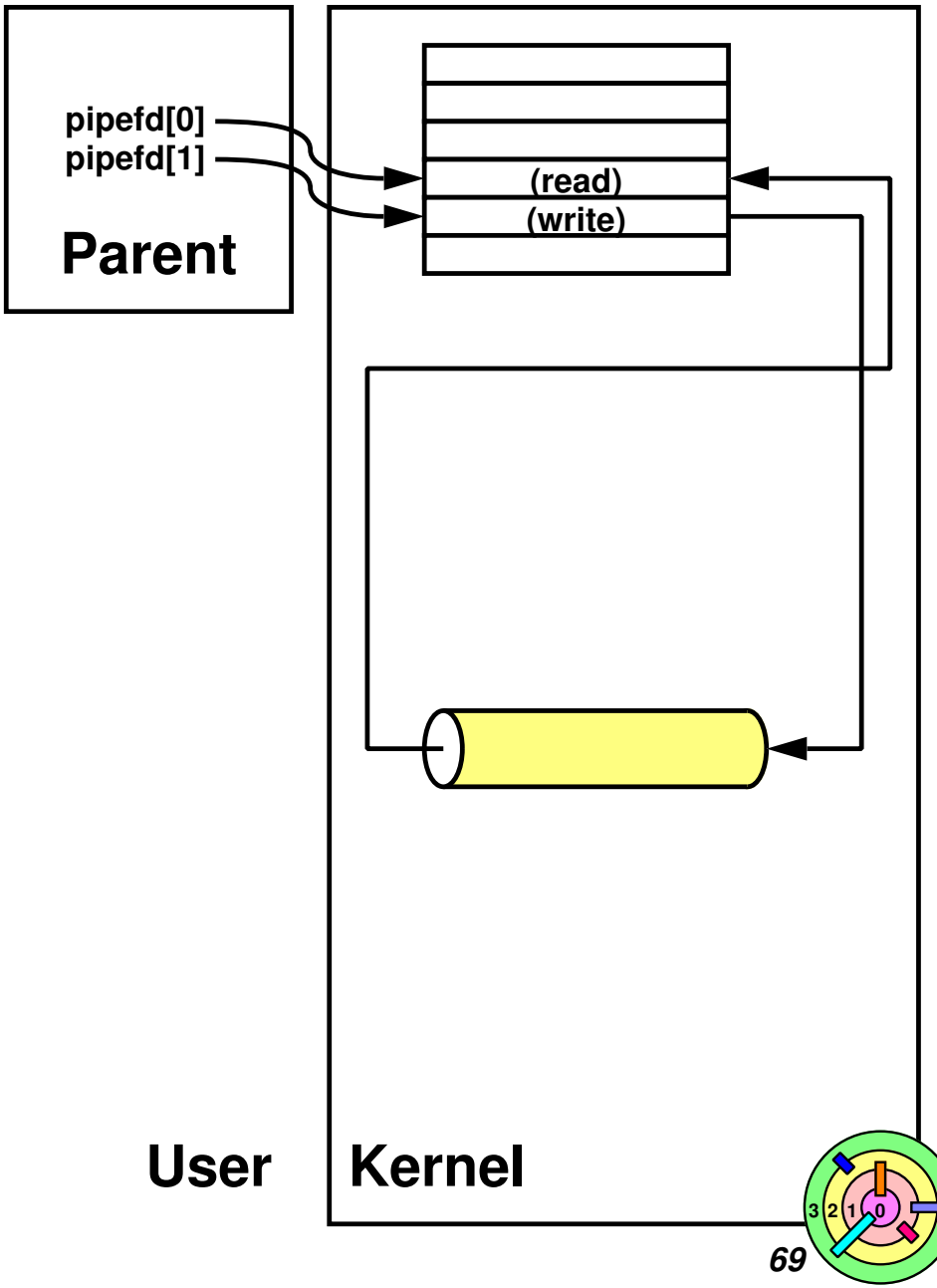
Implementing a Shell

- A program can be a file of commands (i.e., a shell script)
  - the first line of a shell script must be "`#!interpreter`", e.g.,
    - `#!/bin/bash`
    - `#!/bin/tcsh`
    - `#!/bin/tcsh -f`
    - `#!/usr/bin/perl`
    - `#!/usr/bin/python`
    - `#!/usr/bin/python3`
  - the shell would pass the content of the file to the interpreter
- The output of one program can be the input to another program
  - if user types, "`ls -l | wc`", the code in the shell will be more complicated since you have to create two child processes and a pipe object and hook things up just right
  - can use this trick to create a *pipeline*
    - Ex: "`cpp file.c | cparse | cgen | as > file.o`"
      - *pipelined parallelism* (at the process level)



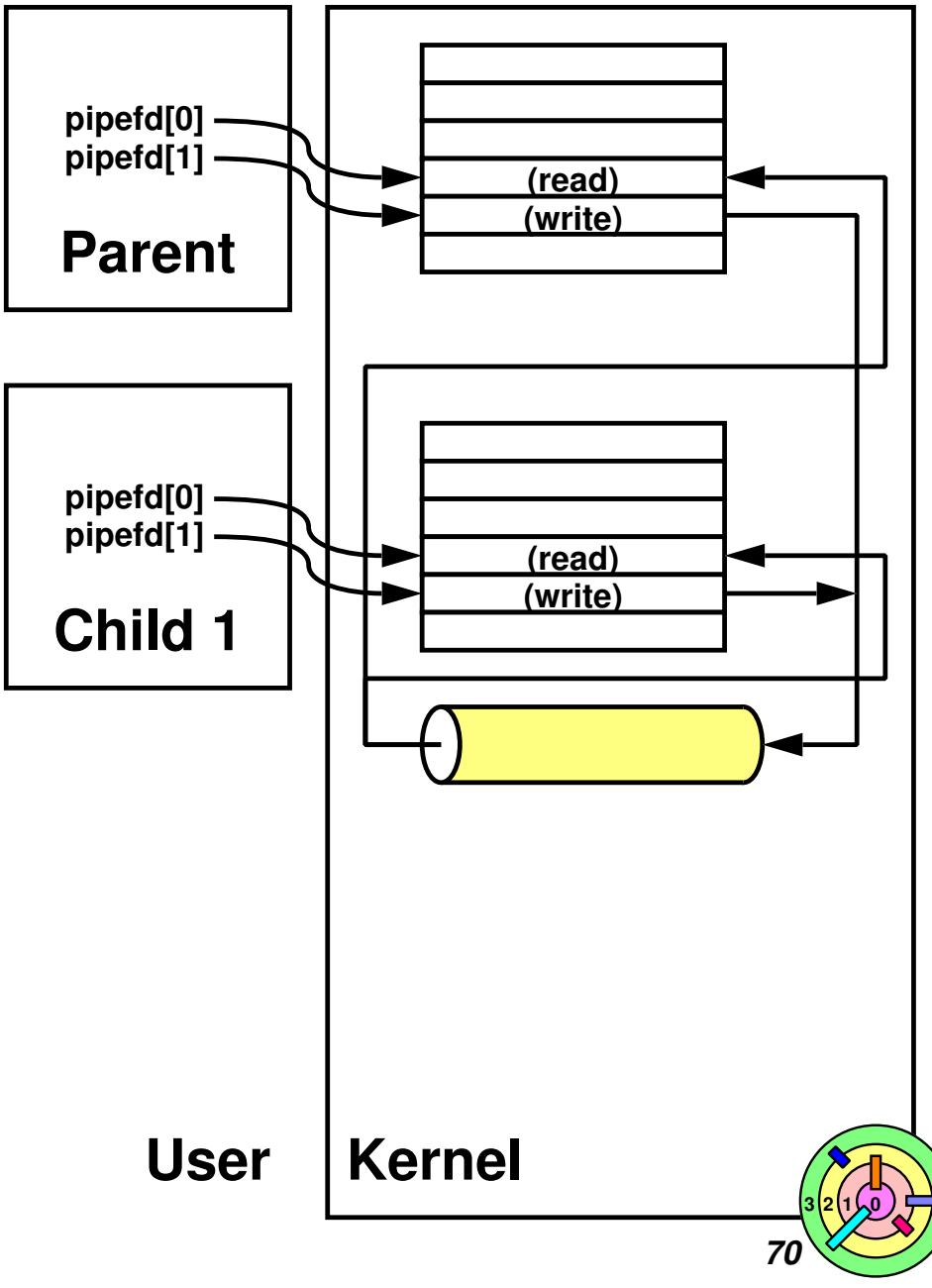
ls -l | wc

```
int pipefd[2];
pid_t ls_pid, wc_pid;

pipe(pipefd);
// CHILD PROCESS: ls
if ((ls_pid = fork()) == 0) {
  close(1);
  dup2(pipefd[1], 1);
  close(pipefd[1]);
  close(pipefd[0]);
  execl("/bin/ls", "ls", "-l", NULL);
  exit(1);
}
// CHILD PROCESS: wc
if ((wc_pid = fork()) == 0) {
  close(0);
  dup2(pipefd[0], 0);
  close(pipefd[0]);
  close(pipefd[1]);
  execl("/usr/bin/wc", "wc", NULL);
  exit(1);
}
// PARENT PROCESS
close(pipefd[0]);
close(pipefd[1]);
```



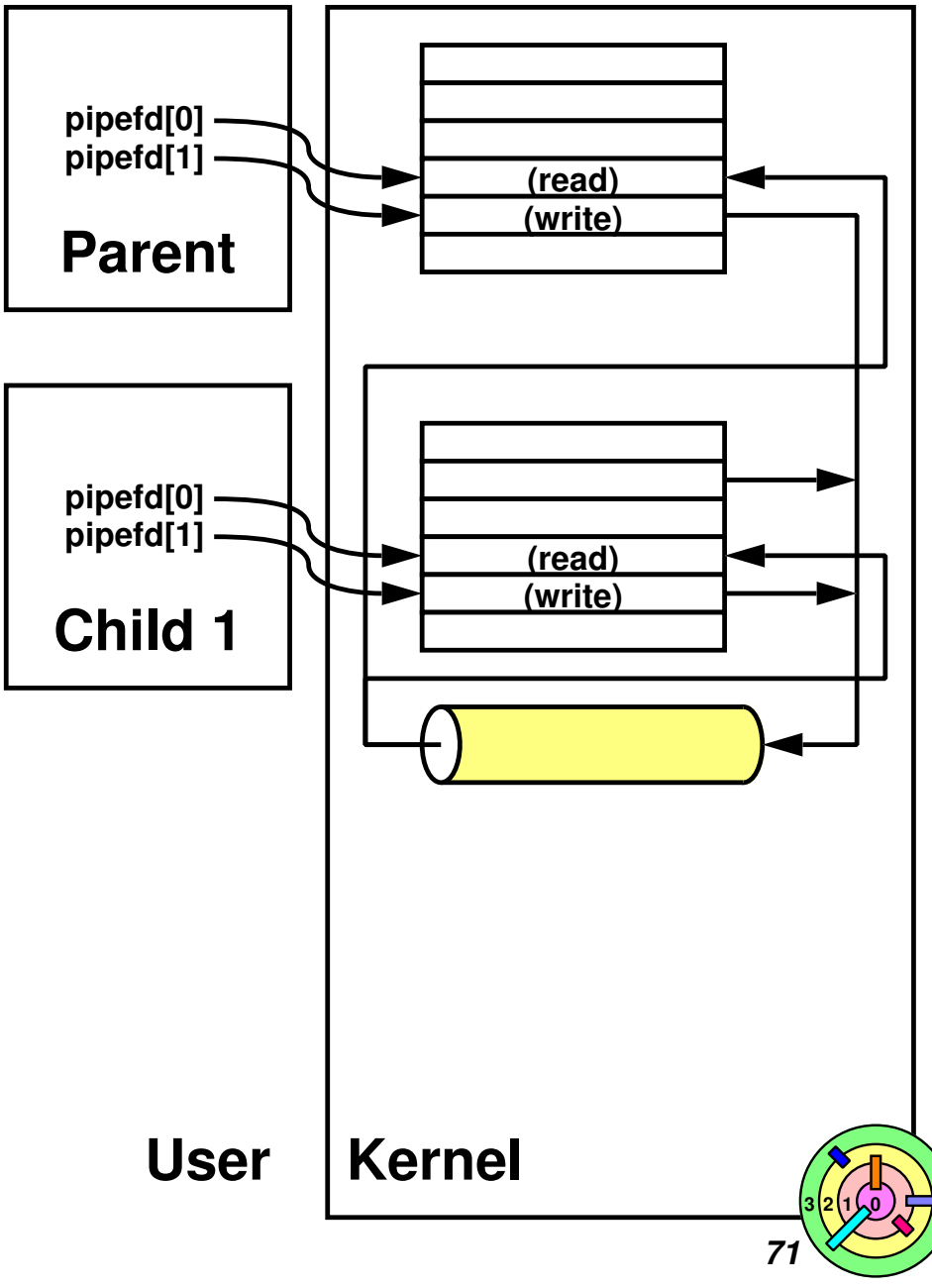
ls -l | wc

```
int pipefd[2];
pid_t ls_pid, wc_pid;

pipe(pipefd);
// CHILD PROCESS: ls
if ((ls_pid = fork()) == 0) {
  close(1);
  dup2(pipefd[1], 1);
  close(pipefd[1]);
  close(pipefd[0]);
  execl("/bin/ls", "ls", "-l", NULL);
  exit(1);
}
// CHILD PROCESS: wc
if ((wc_pid = fork()) == 0) {
  close(0);
  dup2(pipefd[0], 0);
  close(pipefd[0]);
  close(pipefd[1]);
  execl("/usr/bin/wc", "wc", NULL);
  exit(1);
}
// PARENT PROCESS
close(pipefd[0]);
close(pipefd[1]);
```



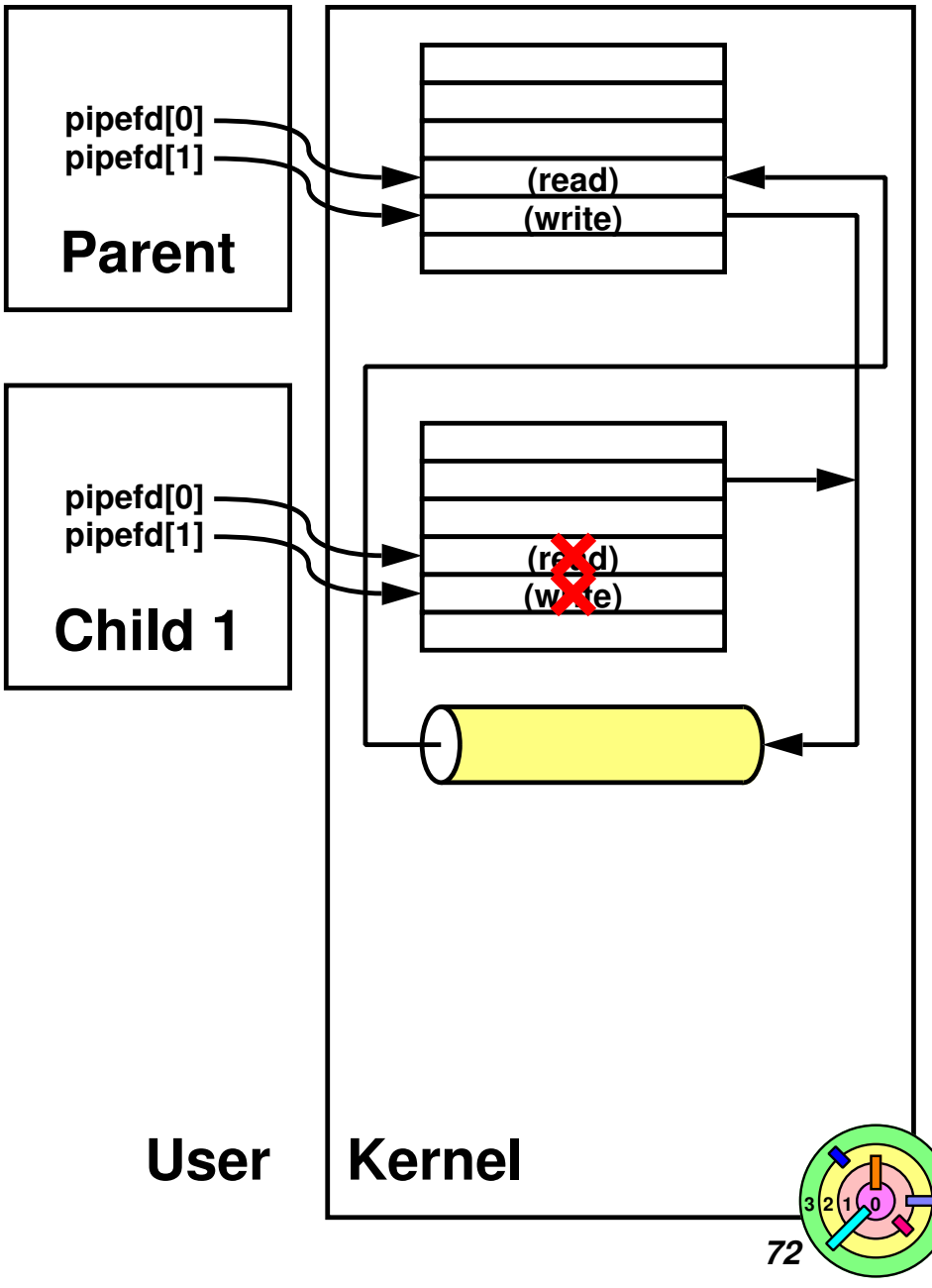
ls -l | wc

```
int pipefd[2];
pid_t ls_pid, wc_pid;

pipe(pipefd);
// CHILD PROCESS: ls
if ((ls_pid = fork()) == 0) {
  close(1);
  dup2(pipefd[1], 1);
  close(pipefd[1]);
  close(pipefd[0]);
  execl("/bin/ls", "ls", "-l", NULL);
  exit(1);
}
// CHILD PROCESS: wc
if ((wc_pid = fork()) == 0) {
  close(0);
  dup2(pipefd[0], 0);
  close(pipefd[0]);
  close(pipefd[1]);
  execl("/usr/bin/wc", "wc", NULL);
  exit(1);
}
// PARENT PROCESS
close(pipefd[0]);
close(pipefd[1]);
```



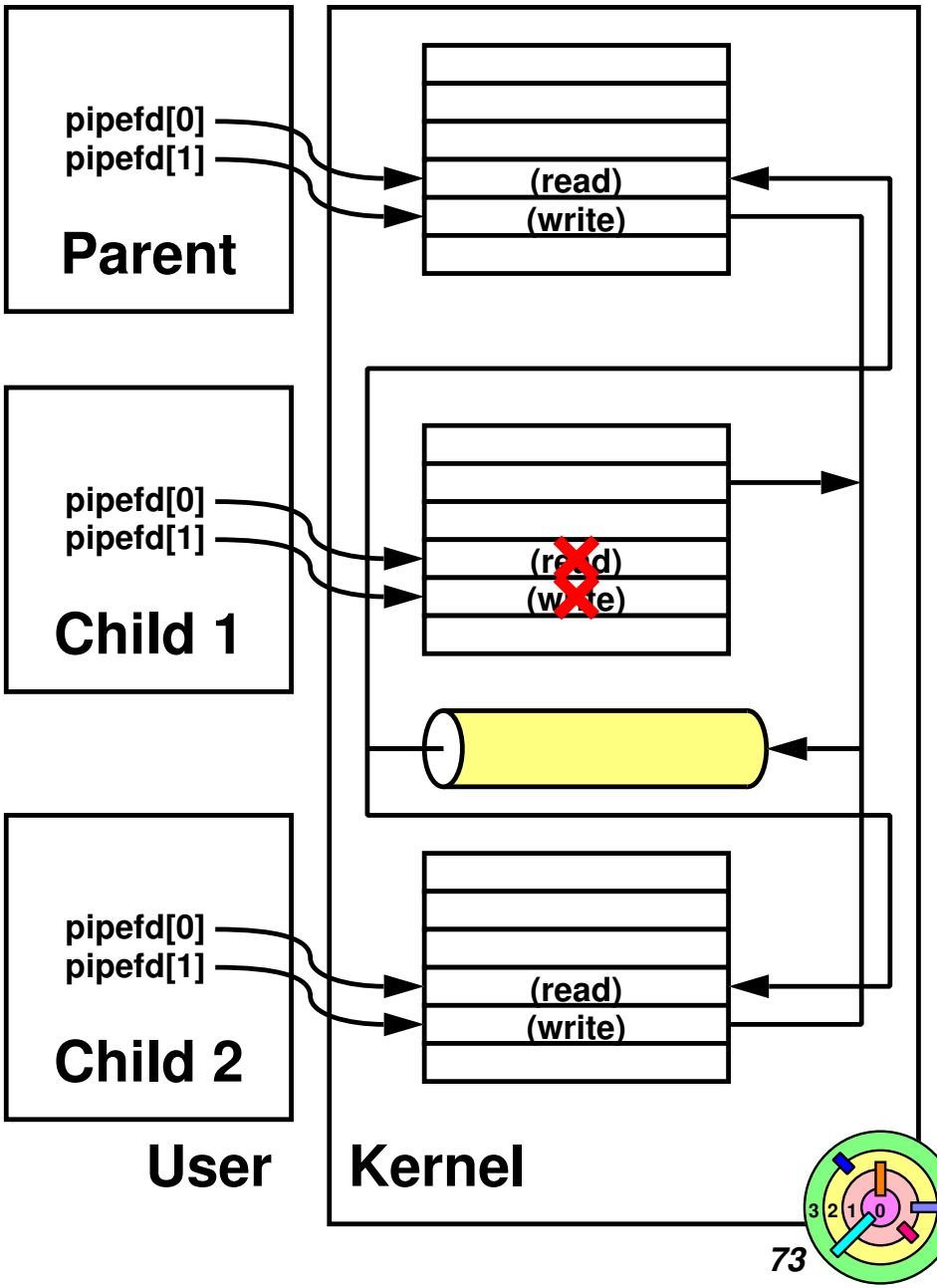
ls -l | wc

```
int pipefd[2];
pid_t ls_pid, wc_pid;

pipe(pipefd);
// CHILD PROCESS: ls
if ((ls_pid = fork()) == 0) {
  close(1);
  dup2(pipefd[1], 1);
  close(pipefd[1]);
  close(pipefd[0]);
  execl("/bin/ls", "ls", "-l", NULL);
  exit(1);
}
// CHILD PROCESS: wc
if ((wc_pid = fork()) == 0) {
  close(0);
  dup2(pipefd[0], 0);
  close(pipefd[0]);
  close(pipefd[1]);
  execl("/usr/bin/wc", "wc", NULL);
  exit(1);
}
// PARENT PROCESS
close(pipefd[0]);
close(pipefd[1]);
```



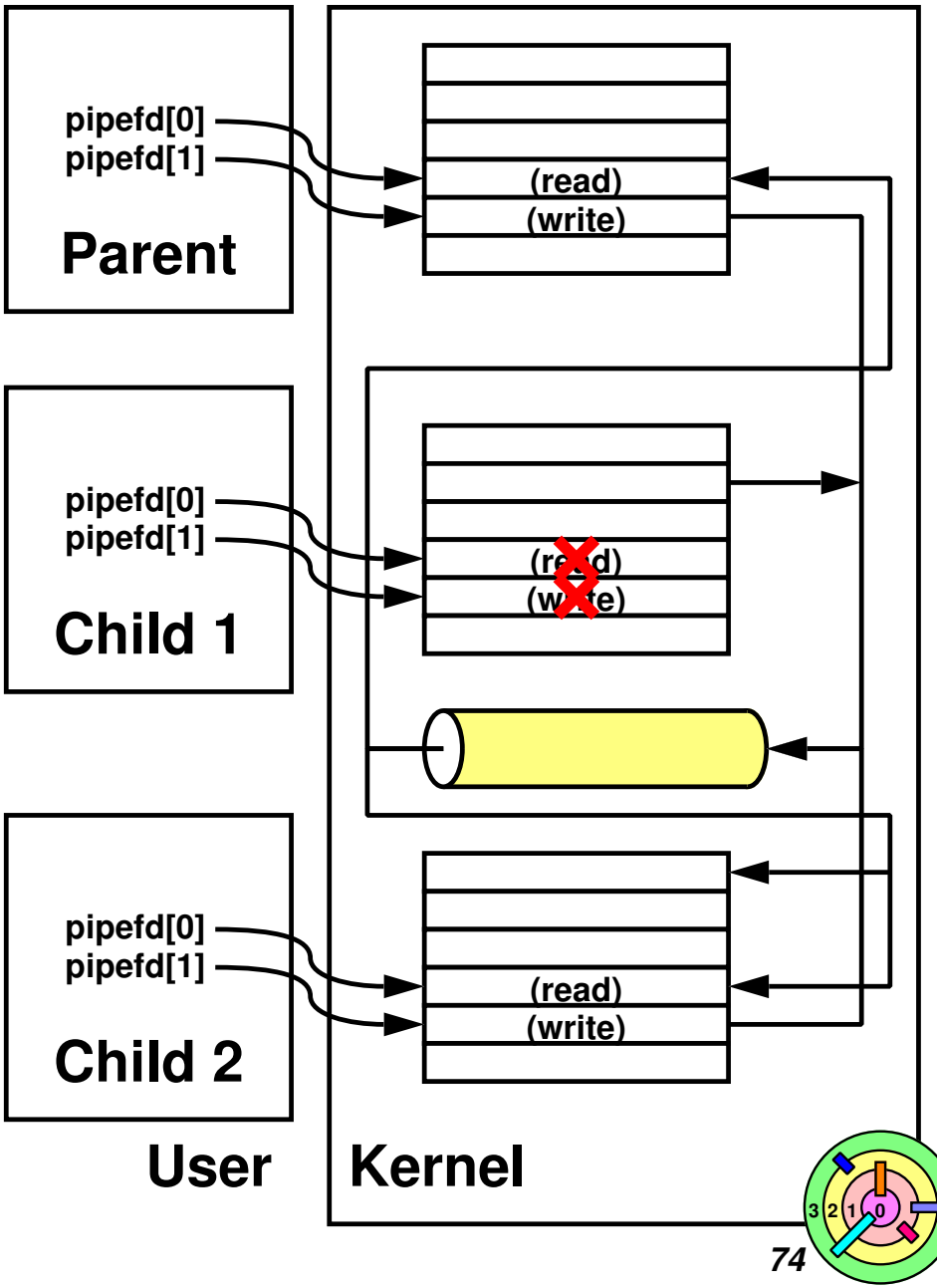
ls -l | wc

```
int pipefd[2];
pid_t ls_pid, wc_pid;

pipe(pipefd);
// CHILD PROCESS: ls
if ((ls_pid = fork()) == 0) {
  close(1);
  dup2(pipefd[1], 1);
  close(pipefd[1]);
  close(pipefd[0]);
  execl("/bin/ls", "ls", "-l", NULL);
  exit(1);
}
// CHILD PROCESS: wc
if ((wc_pid = fork()) == 0) {
  close(0);
  dup2(pipefd[0], 0);
  close(pipefd[0]);
  close(pipefd[1]);
  execl("/usr/bin/wc", "wc", NULL);
  exit(1);
}
// PARENT PROCESS
close(pipefd[0]);
close(pipefd[1]);
```



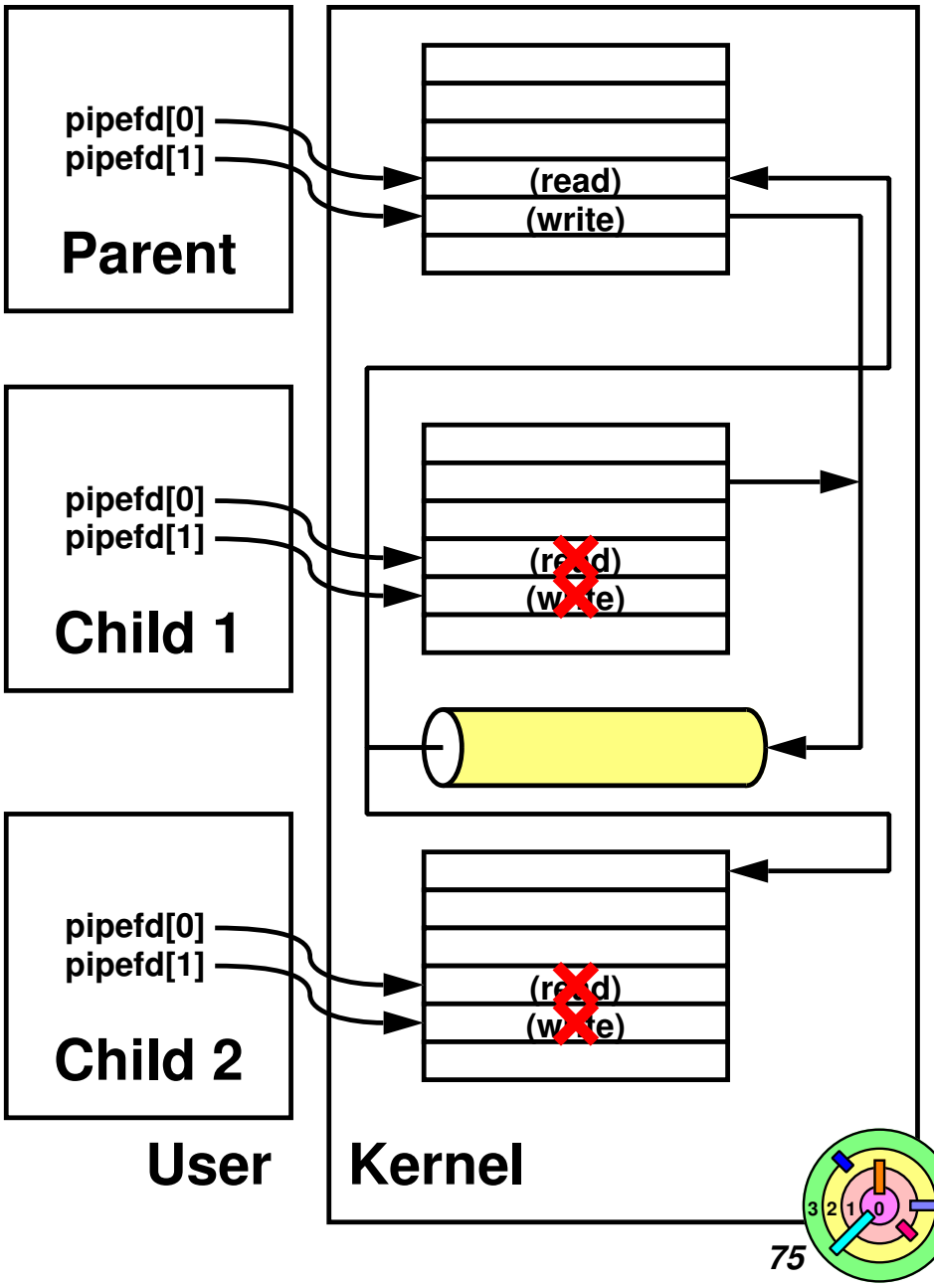
ls -l | wc

```
int pipefd[2];
pid_t ls_pid, wc_pid;

pipe(pipefd);
// CHILD PROCESS: ls
if ((ls_pid = fork()) == 0) {
  close(1);
  dup2(pipefd[1], 1);
  close(pipefd[1]);
  close(pipefd[0]);
  execl("/bin/ls", "ls", "-l", NULL);
  exit(1);
}
// CHILD PROCESS: wc
if ((wc_pid = fork()) == 0) {
  close(0);
  dup2(pipefd[0], 0);
  close(pipefd[0]);
  close(pipefd[1]);
  execl("/usr/bin/wc", "wc", NULL);
  exit(1);
}
// PARENT PROCESS
close(pipefd[0]);
close(pipefd[1]);
```



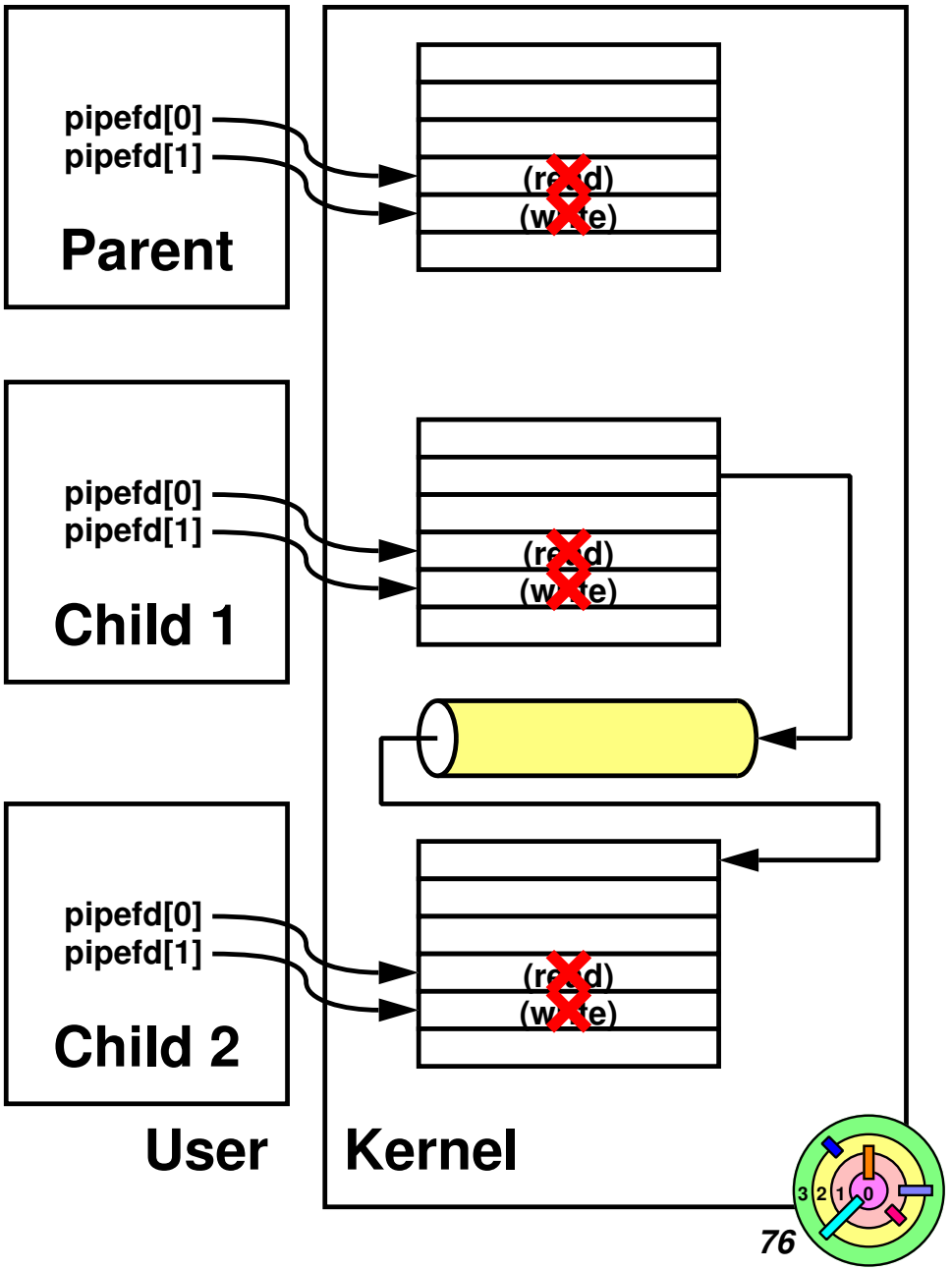
ls -l | wc

```
int pipefd[2];
pid_t ls_pid, wc_pid;

pipe(pipefd);
// CHILD PROCESS: ls
if ((ls_pid = fork()) == 0) {
   close(1);
   dup2(pipefd[1], 1);
   close(pipefd[1]);
   close(pipefd[0]);
   execl("/bin/ls", "ls", "-l", NULL);
   exit(1);
}
// CHILD PROCESS: wc
if ((wc_pid = fork()) == 0) {
   close(0);
   dup2(pipefd[0], 0);
   close(pipefd[0]);
   close(pipefd[1]);
   execl("/usr/bin/wc", "wc", NULL);
   exit(1);
}
// PARENT PROCESS
close(pipefd[0]);
close(pipefd[1]);
```

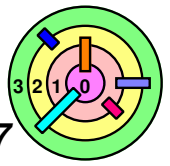


ls -l | wc

```
int pipefd[2];
pid_t ls_pid, wc_pid;

pipe(pipefd);
// CHILD PROCESS: ls
if ((ls_pid = fork()) == 0) {
  close(1);
  dup2(pipefd[1], 1);
  close(pipefd[1]);
  close(pipefd[0]);
  execl("/bin/ls", "ls", "-l", NULL);
  exit(1);
}
// CHILD PROCESS: wc
if ((wc_pid = fork()) == 0) {
  close(0);
  dup2(pipefd[0], 0);
  close(pipefd[0]);
  close(pipefd[1]);
  execl("/usr/bin/wc", "wc", NULL);
  exit(1);
}
// PARENT PROCESS
close(pipefd[0]);
close(pipefd[1]);
```

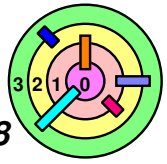


(3.4) Case Study: Interprocess Communication

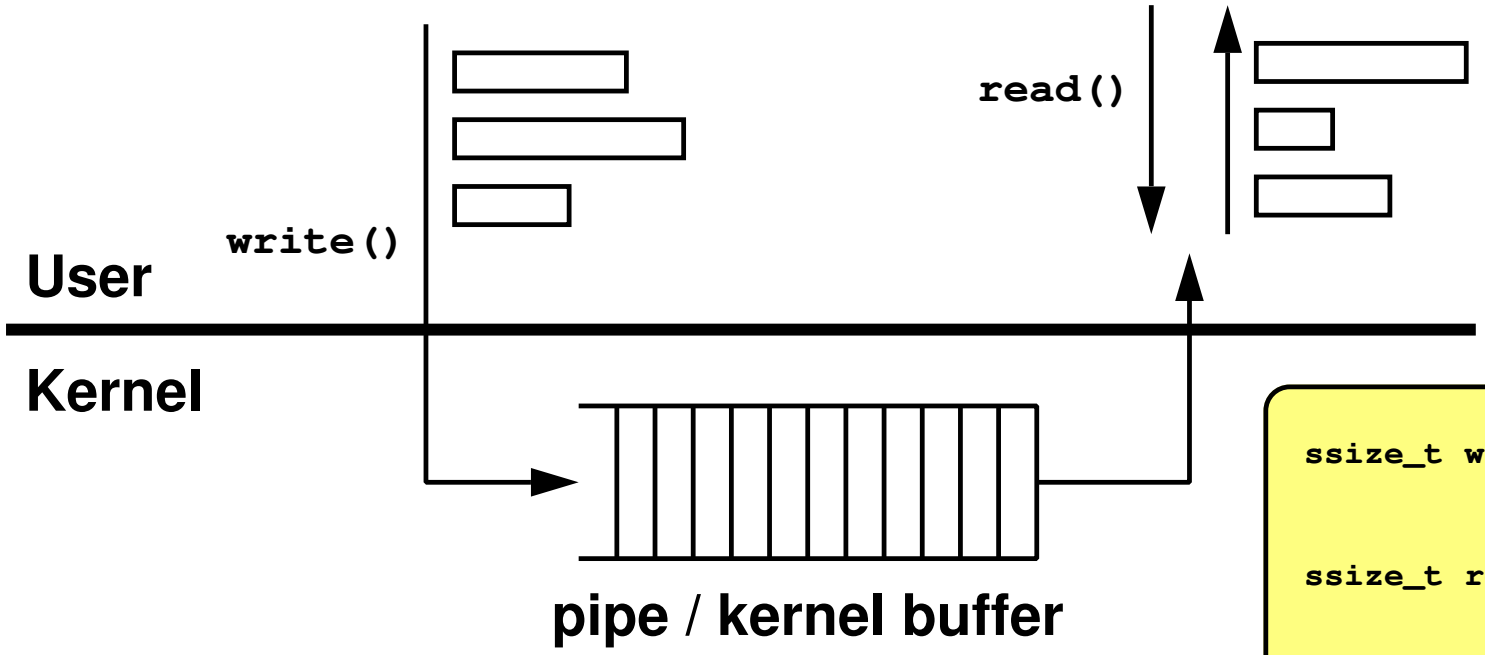


Interprocess Communication

- How to communicate with a print server on the same machine?
  - *producer-consumer:* one-way communication, i.e., producer writes date and consumer reads data
    - can use pipes and get pipelined parallelism
  - *client-server:* two-way communication, i.e., client makes requests and server sends responses
  - *file system:* one program creates a file to be processed by another program at a later time
    - these programs do not have to be running at the same time
    - data needs to be stored persistently (e.g., on disk) and needs to be named so that it can be located
- How to communicate with a print server on a remote machine?
  - *producer-consumer:* e.g., Google MapReduce
  - *client-server:* e.g., the web
  - *file system:* e.g., distributed file server

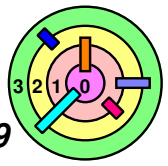


Producer-Consumer Communication

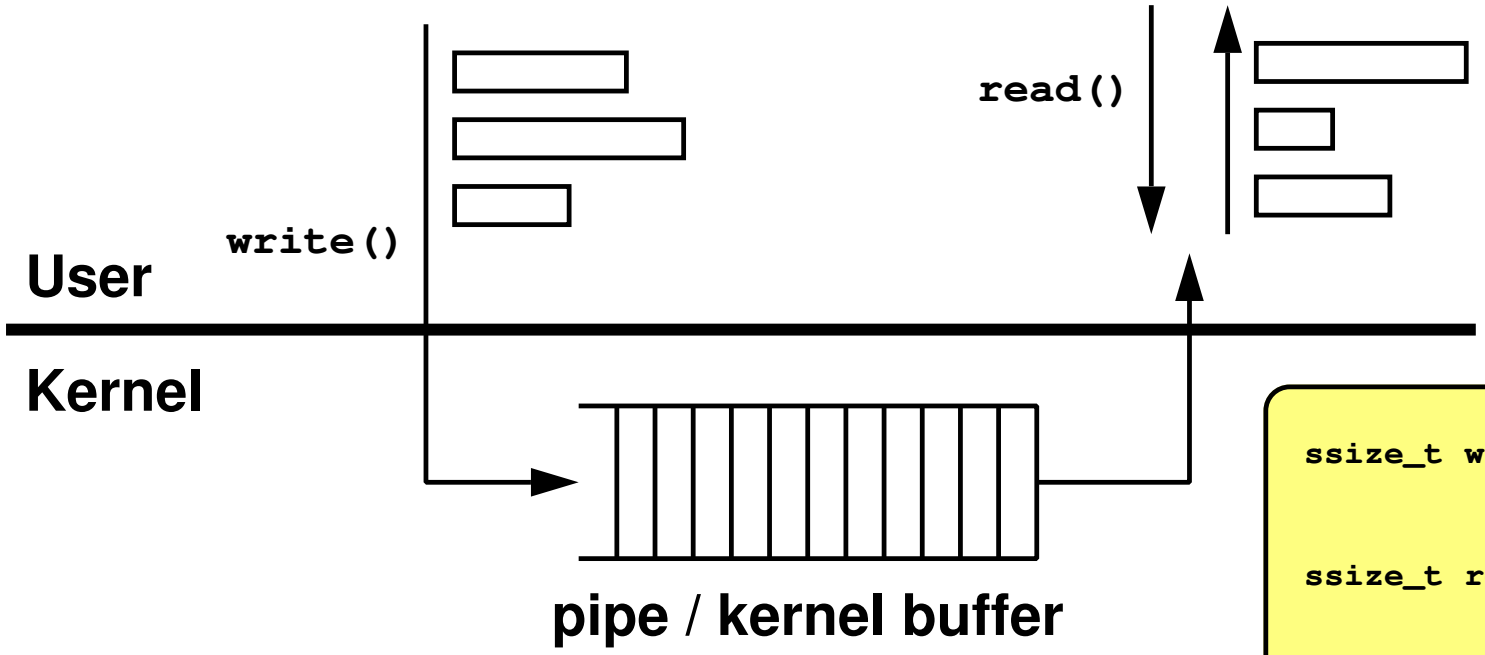



```
ssize_t write(int fd,
              const void *buf,
              size_t count);
ssize_t read(int fd,
             void *buf,
             size_t count);
```

- The pipe is implemented with a *finite buffer*
  - data being written and read can be of different sizes
  - `write()` returns immediately if there is space in the buffer
  - `write()` blocks if there isn't enough space in the buffer
  - `read()` returns immediately if there is data in the buffer
    - may return less data
  - `read()` blocks if the buffer is empty

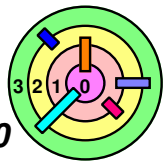


Producer-Consumer Communication

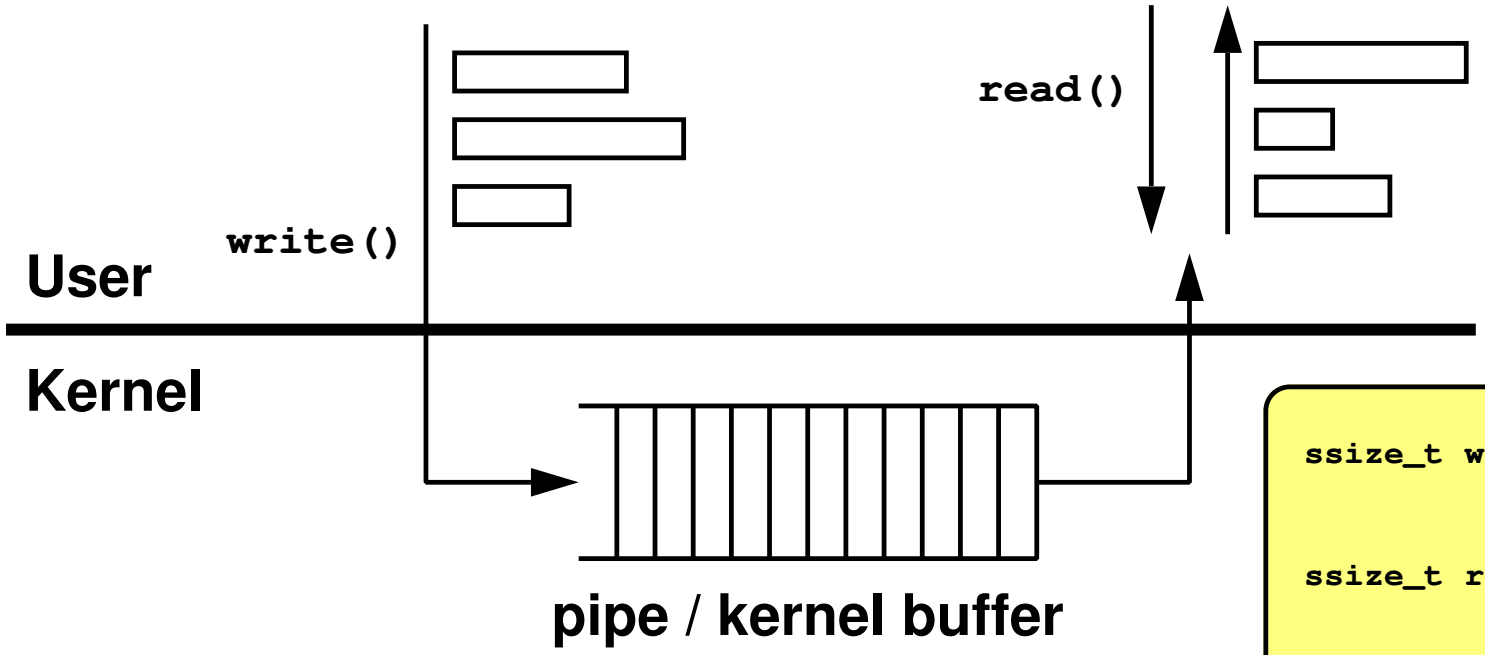



```
ssize_t write(int fd,
              const void *buf,
              size_t count);
ssize_t read(int fd,
             void *buf,
             size_t count);
```

- The pipe is implemented with a ***finite buffer***
  - producer and consumer can run at their own pace (they don't have to be synchronized or run in lock steps)
  - producer and consumer may run in parallel, if permitted (see last page)
  - when producer finishes, it closes the write end of the pipe
  - when consumer finishes, it closes the read end of the pipe

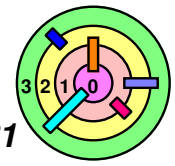


Producer-Consumer Communication

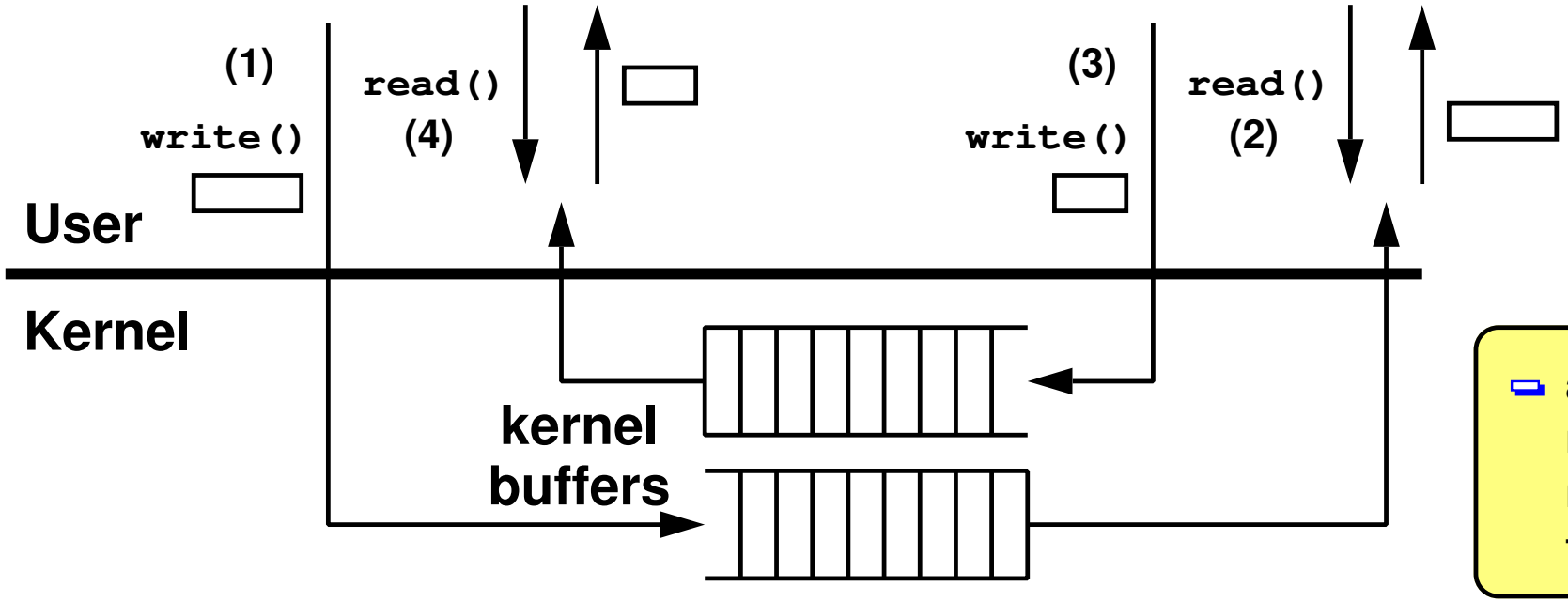



```
ssize_t write(int fd,
              const void *buf,
              size_t count);
ssize_t read(int fd,
             void *buf,
             size_t count);
```

- The pipe is implemented with a *finite buffer*
  - there is no difference reading from the pipe or writing to the pipe
  - but there are differences regarding what system calls are more likely to work on a file
    - `rewind()` and `lseek()` may return error on pipes if you try to move the file cursor position beyond the end of the buffer



Client-Server Communication



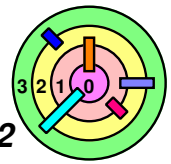
- assuming that requests and responses are fixed-size

Client code:

```
char req[ReqSize];
char rsp[RspSize]
// ..compute..
// put the request into the buffer
// send the buffer to the server
write(output, req, ReqSize);
// wait for response
read(input, rsp, RspSize);
// ..compute..
```

Server code:

```
char req[ReqSize];
char rsp[RspSize];
// loop waiting for requests
while (1) {
  // read incoming command
  read(input, req, ReqSize);
  // do operation
  // send result
  write(output, rsp, RspSize);
}
```



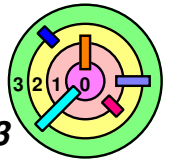
Client-Server Communication

- What if you want to allow multiple clients to talk to the same server at the same time?
  - the server can use the `select()` system call to identify the pipe containing the request
  - client code stays the same

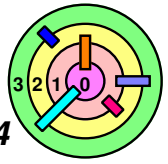
Server code:

```
char req[ReqSize];
char rsp[RspSize];
FileDescriptor clientInput[NumClients];
FileDescriptor clientOutput[NumClients];
// loop waiting for a req from any client
while ((int index=select(clientInput, NumClients)) >= 0) {
  // read incoming command from a specific client
  read(clientInput[index], req, ReqSize);
  // do operation
  // send result
  write(clientOutput[index], rsp, RspSize);
}
```

  - the real `select()` system call is more messy
    - Windows has `WaitForMultipleObjects()`

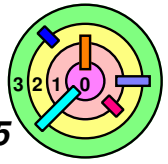


(3.5) Operating System Structure

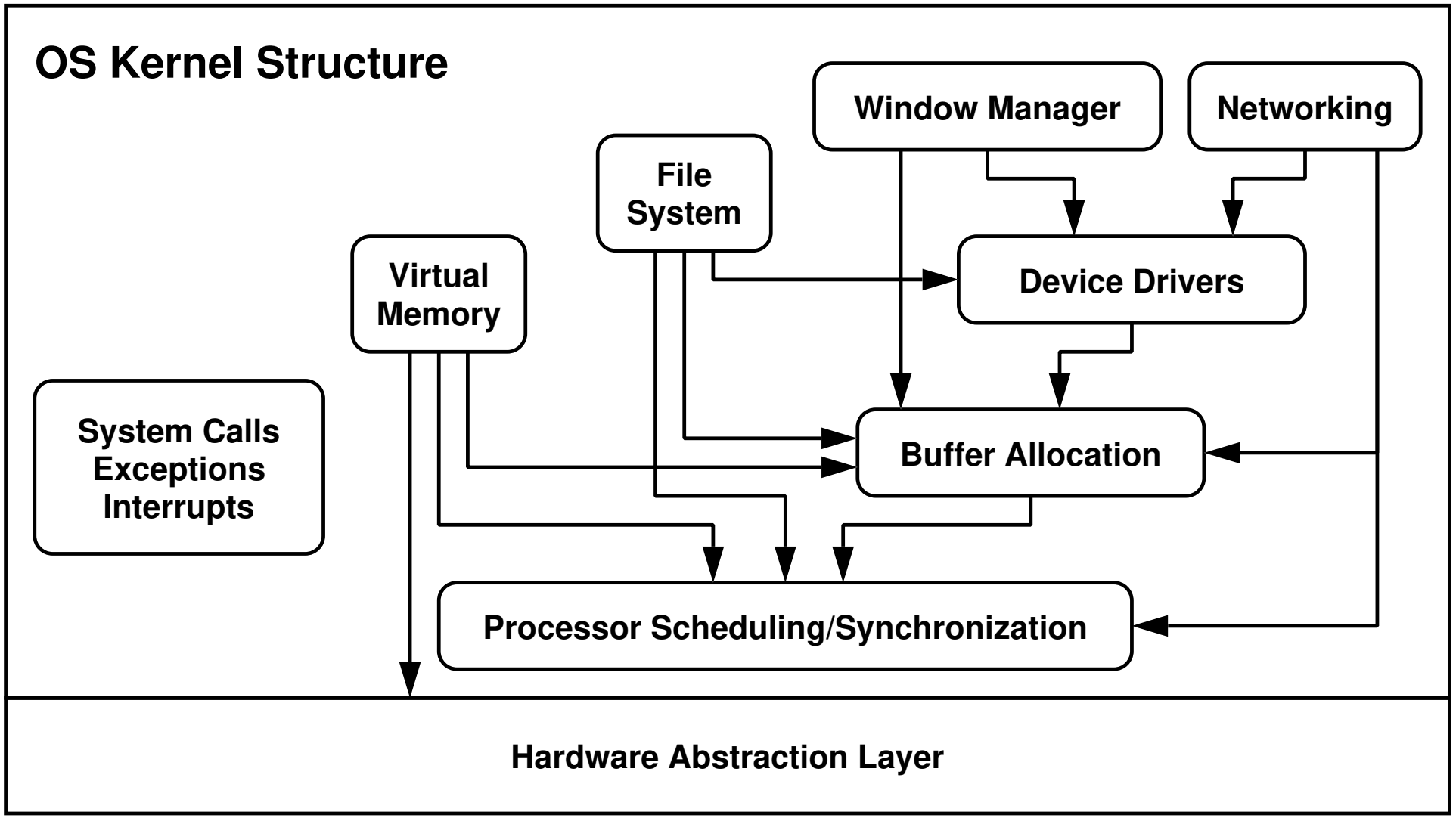


Operating System Structure

- As demonstrated, some OS functionalities can be done in user space
  - e.g., shell, print server, file server
  - natural question: how much OS functionalities should be done in user space
    - advantage of keeping things in the kernel: fast
    - disadvantage of keeping things in the kernel: less flexible, more difficult to innovate
- Is it easy to move OS modules into user space?
  - not so easy because modules in kernel have dependencies, and some require frequent interactions
- We will take a look at some OS architectures
  - *Monolithic Kernels*
  - *Microkernel*



Monolithic Kernels



Monolithic Kernel: most of the OS functionalities is linked together inside the kernel