5.1.1.3 Scheduler Activations Model

Problems With Two-level Model

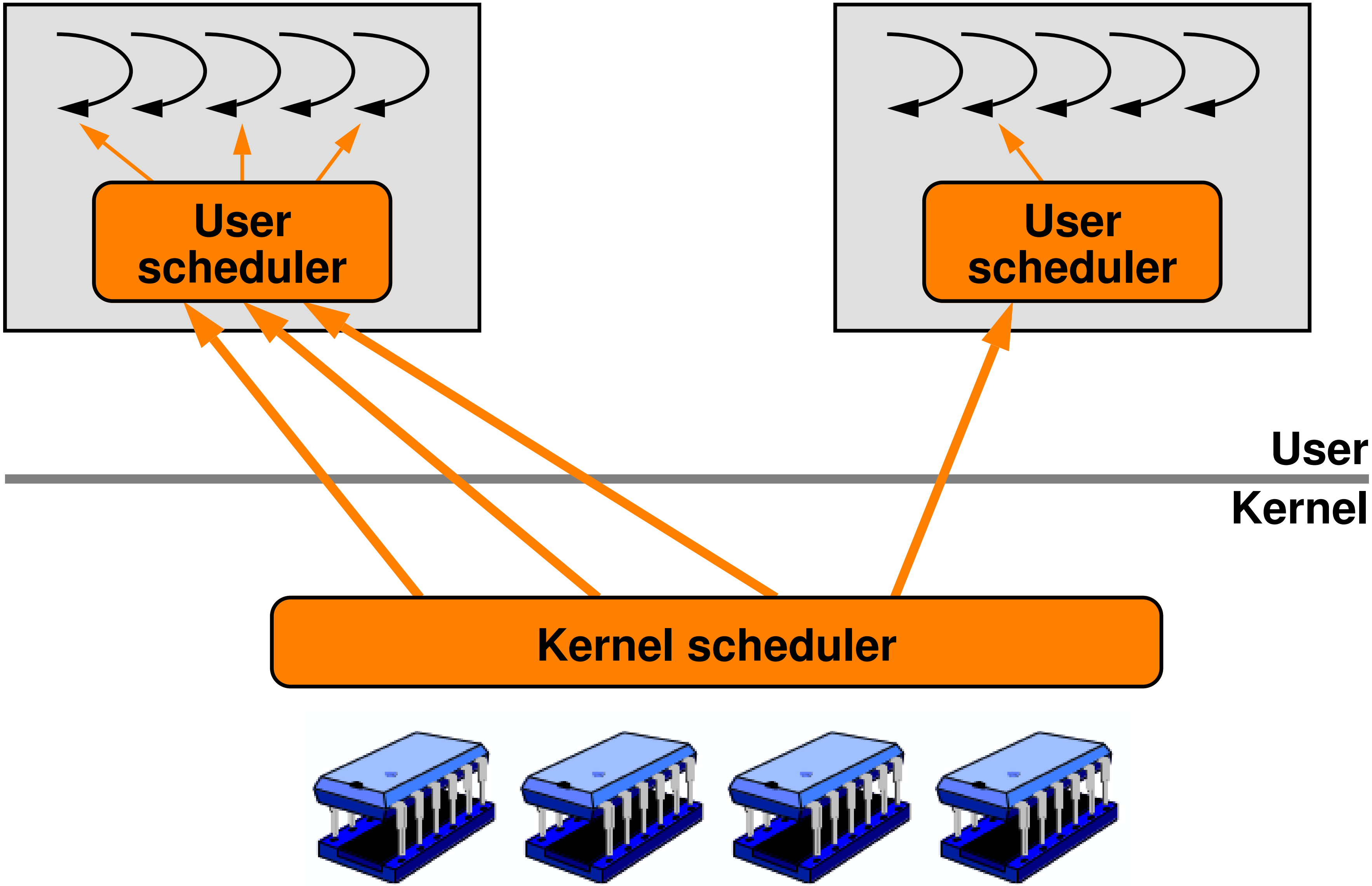
- Two-level model does not solve the I/O blocking problem
  - if there are *N* kernel threads and if *N* user threads are blocked in I/O
    - no other user threads can make progress
    - Solaris solution basically goes back to one-level model

- Another problem: *Priority Inversion*
  - user-level thread schedulers are not aware of the kernel-level thread scheduler
    - it may know the number of kernel threads
  - how can the user-level scheduler talk to the kernel-level scheduler?
    - people have tried this, but it's complicated
  - it's possible to have a higher priority user thread scheduled on a lower priority kernel thread and vice versa

Scheduler Activations Model - Variation on Two-Level Model

- The scheduler activations model is radically different from the other models
  - in other models, we think of the kernel as providing some kernel thread contexts
    - then multiplexing these contexts on processors using the kernel's scheduler
  - in scheduler activations model, we divvy up processors to processes, and processes determine which threads get to use these processors
    - the kernel should supply however many kernel contexts it finds necessary

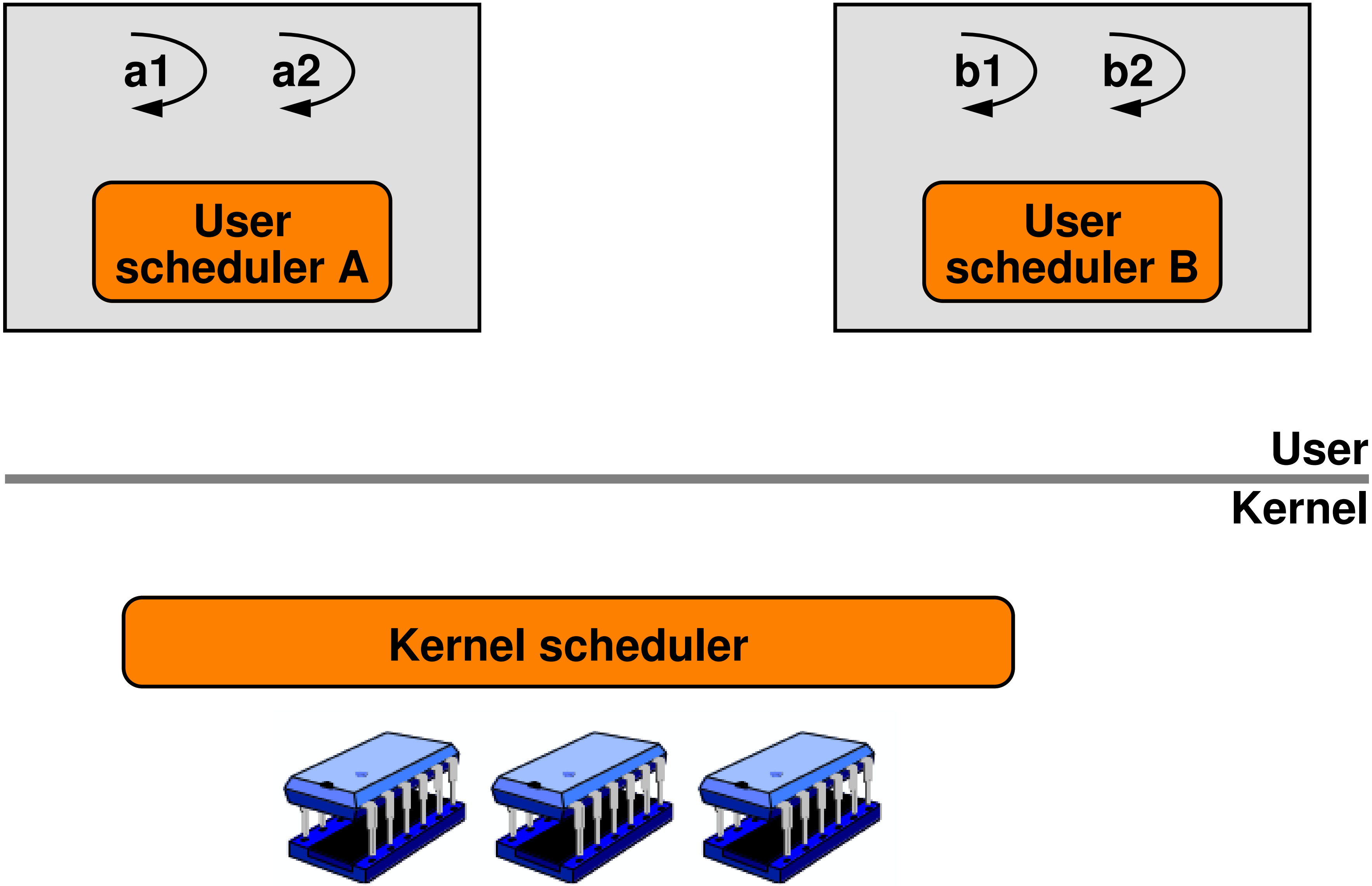
Scheduler Activations Model Example



Scheduler Activations Model Example

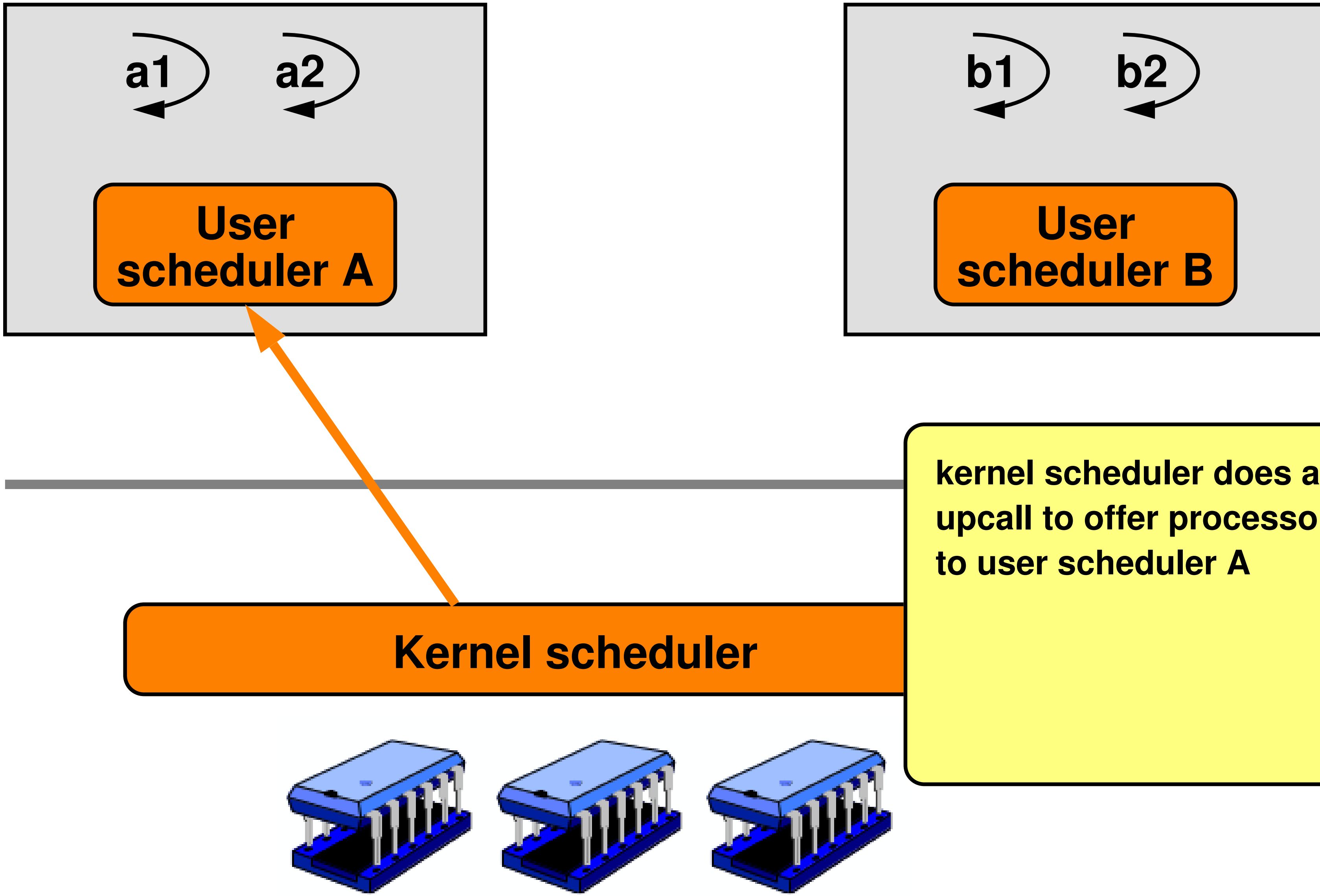
- Let's say a process starts up running a single thread
  - kernel scheduler assigns a processor to the process
  - if the thread blocks, the process gives up the processor to the kernel scheduler
- Suppose the user program creates a new thread and parallelism is desired
  - code in user-level library notifies the kernel that it needs two processors
  - when a processor becomes available, the kernel creates a new kernel context
    - the kernel places an upcall to the user-level library, effectively giving it the processor
    - the user-level library code assigns this processor to the new thread

Scheduler Activations Model Example



- Kernel scheduler does not schedule threads

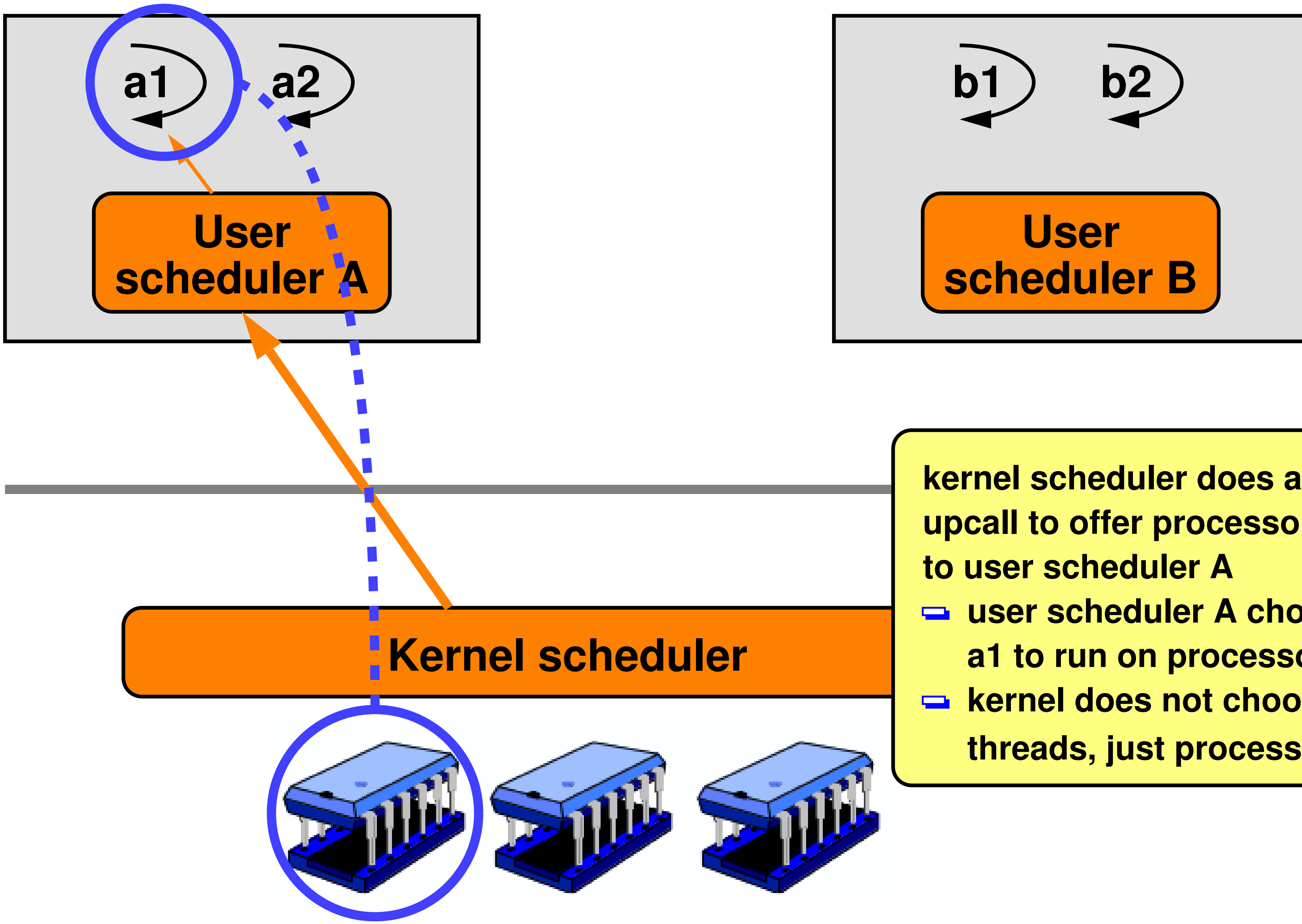
Scheduler Activations Model Example



kernel scheduler does an upcall to offer processor 1 to user scheduler A

- Kernel scheduler does not schedule threads

Scheduler Activations Model Example

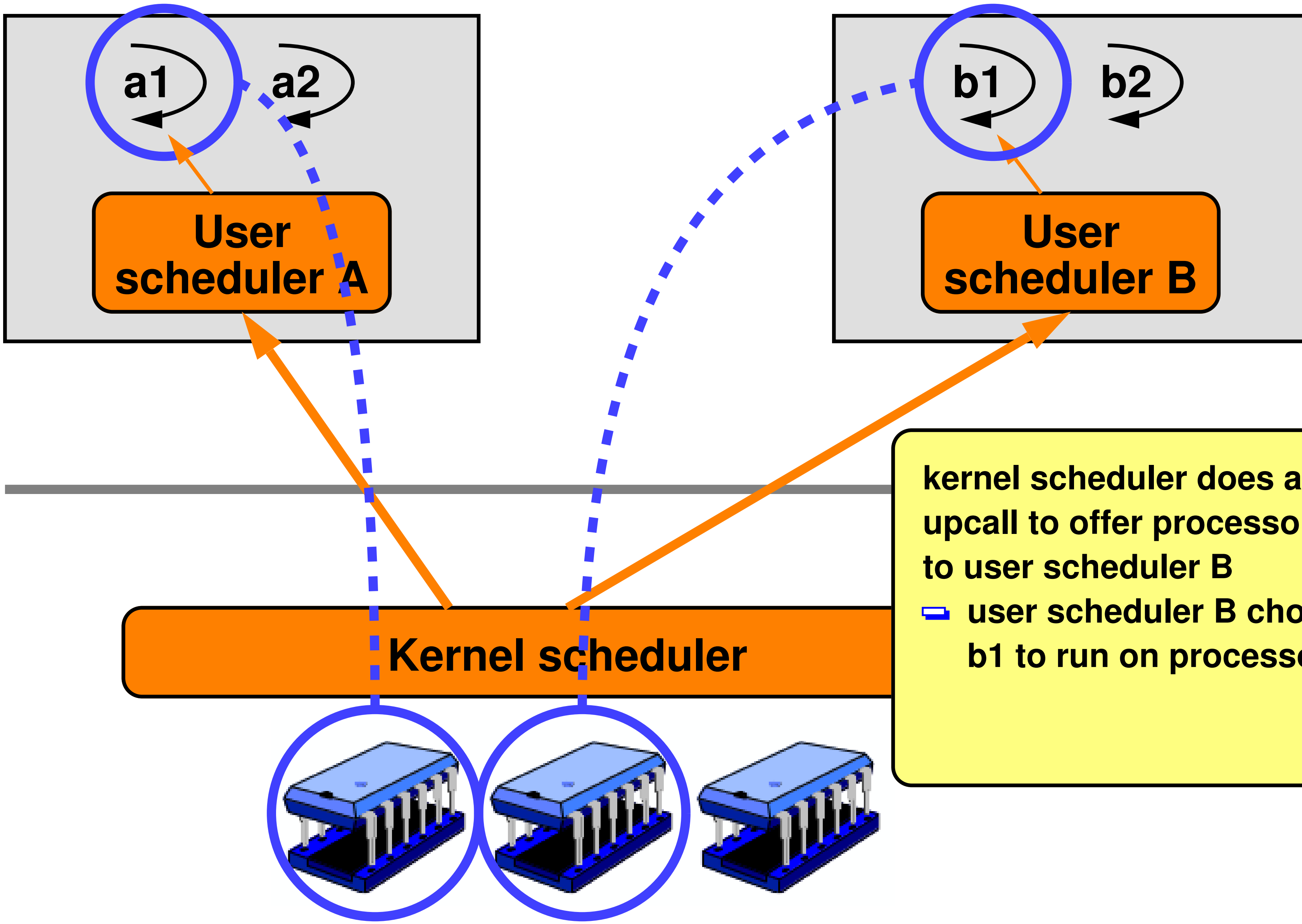


kernel scheduler does an upcall to offer processor 1 to user scheduler A

- user scheduler A chooses a1 to run on processor 1
- kernel does not choose threads, just processes

Kernel scheduler does not schedule threads

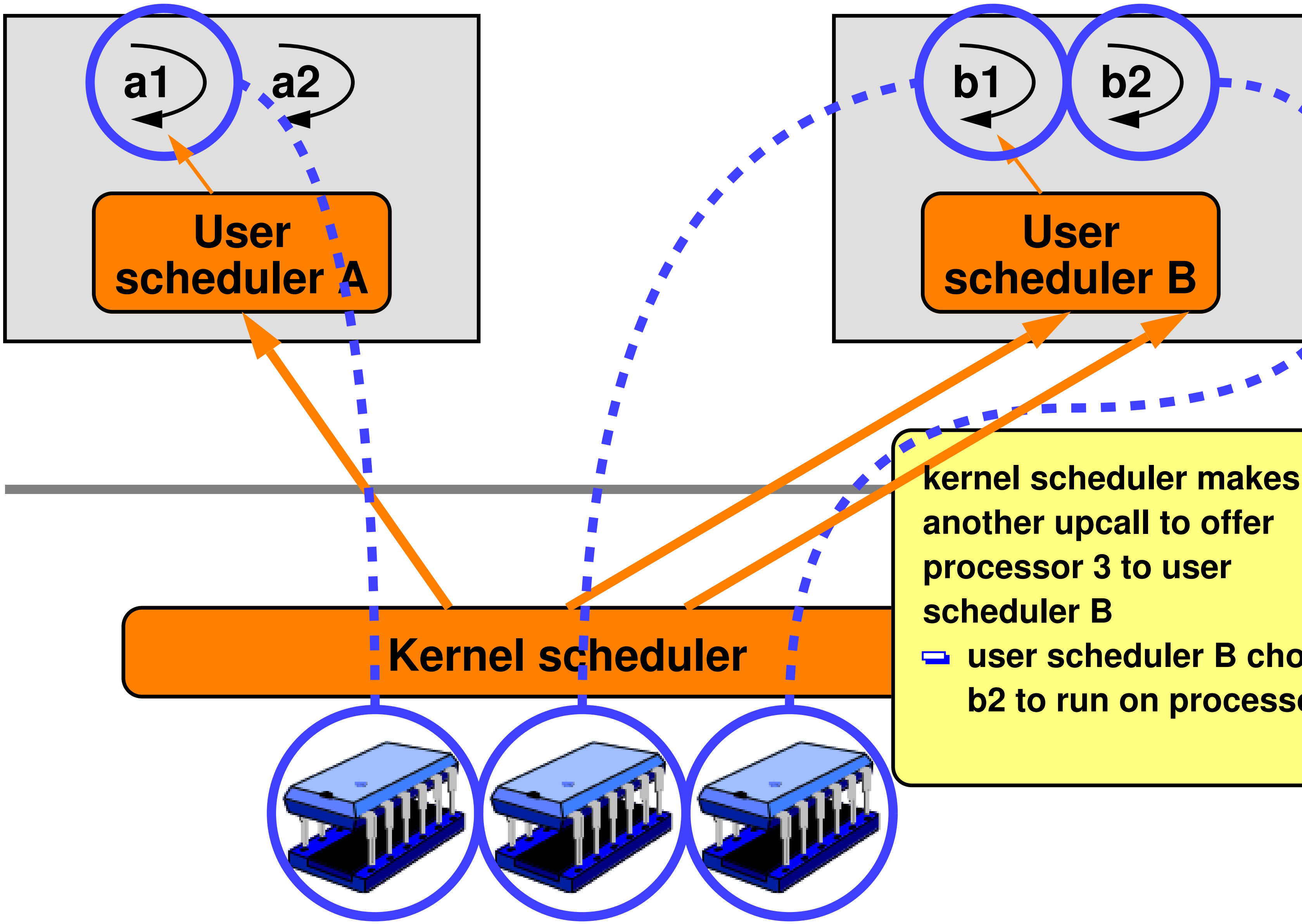
Scheduler Activations Model Example



kernel scheduler does an upcall to offer processor 2 to user scheduler B
- user scheduler B chooses b1 to run on processor 2

- Kernel scheduler does not schedule threads

Scheduler Activations Model Example

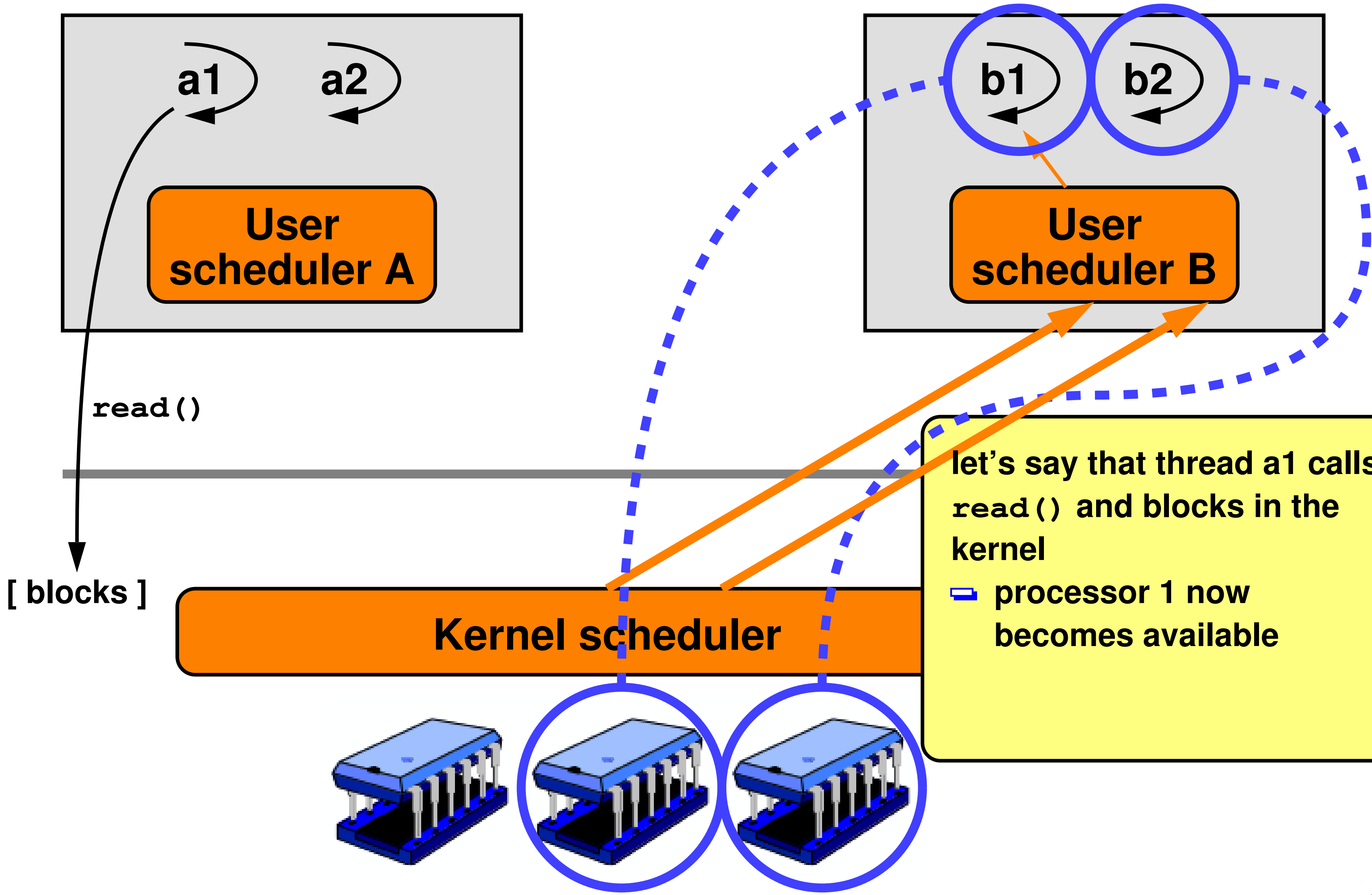


kernel scheduler makes another upcall to offer processor 3 to user scheduler B

- user scheduler B chooses b2 to run on processor 3

- Kernel scheduler does not schedule threads

Scheduler Activations Model Example

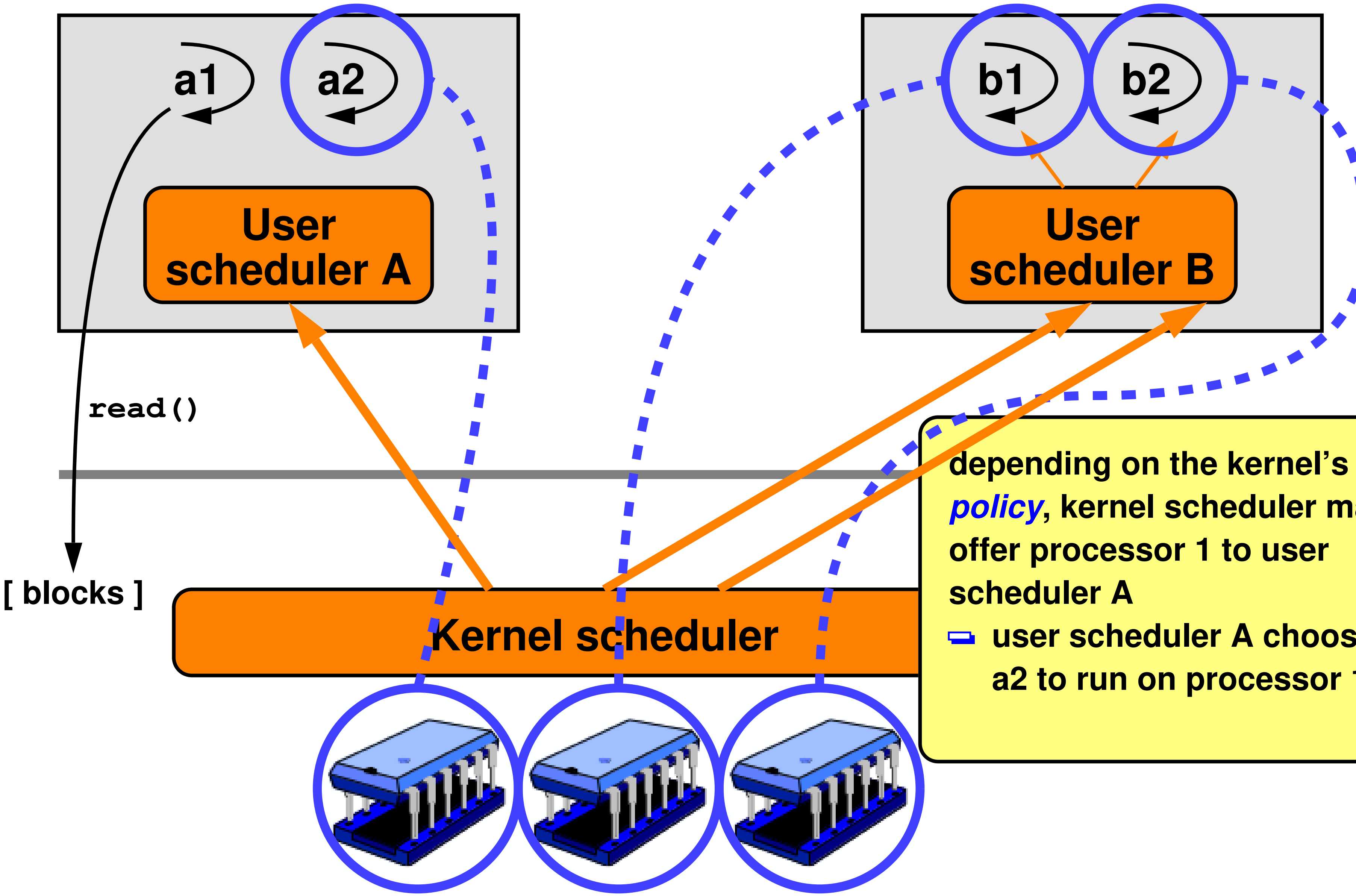


let's say that thread a1 calls `read()` and blocks in the kernel

- processor 1 now becomes available

➡ Kernel scheduler can have various scheduling policies

Scheduler Activations Model Example

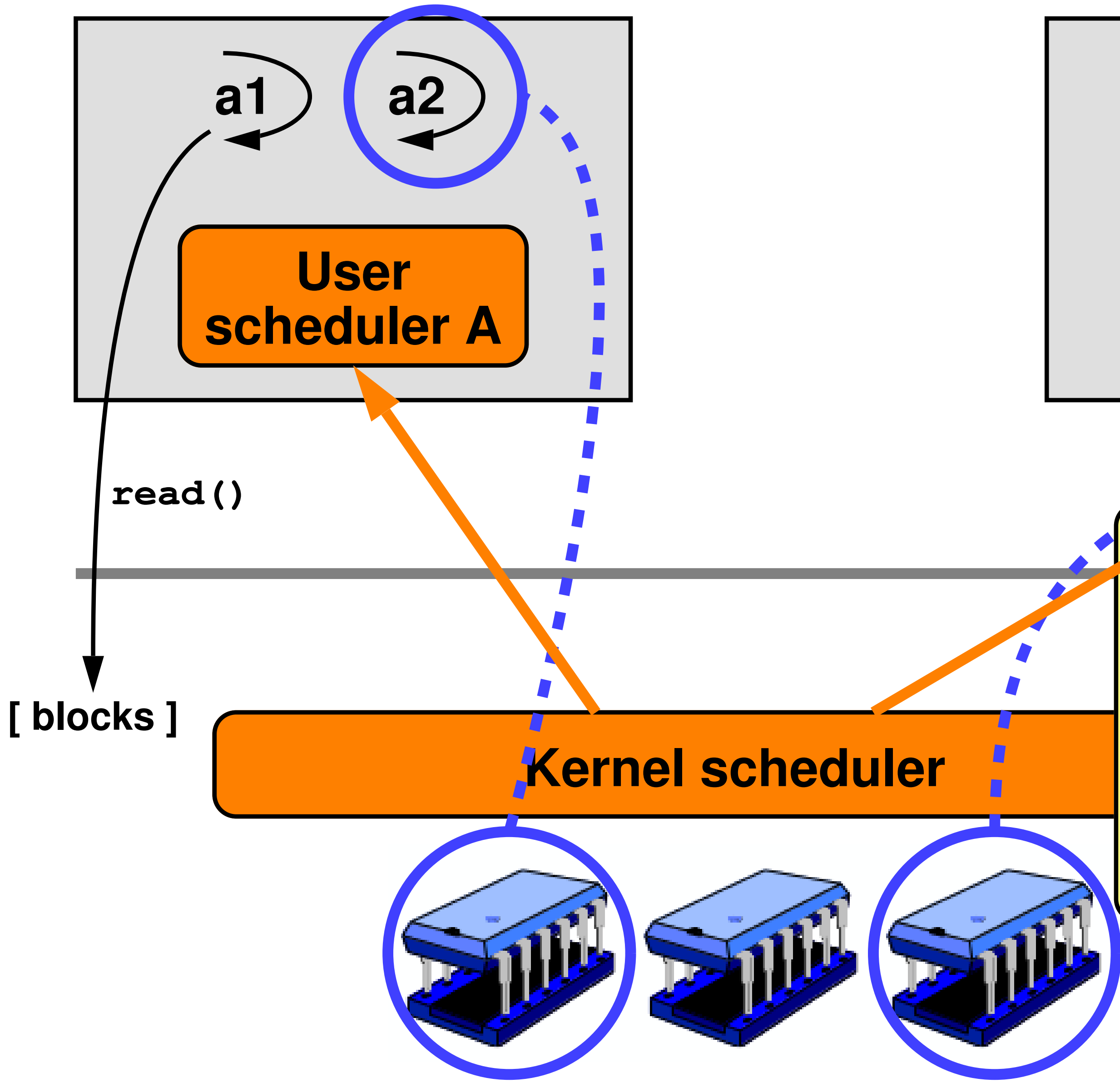


depending on the kernel's *policy*, kernel scheduler may offer processor 1 to user scheduler A

- user scheduler A chooses a2 to run on processor 1

➪ **Kernel scheduler can have various scheduling policies**

Scheduler Activations Model Example



kernel notifies the user schedulers when resources are available/unavailable

- e.g., when b1's quantum expires, kernel can take away processor from b1

➡ Kernel scheduler can have various scheduling policies

Scheduler Activations Model

- Scheduler Activations Model seems like a good solution for two-level models
  - it has not been adopted by a major OS vendor
    - it can take many many years to take something in research and move it into the real world
      - need to conduct extensive experiments to know about all the pluses and minuses of a new approach
      - it may not solve all types of priority inversion problems

4.1 A Simple System (Monolithic Kernel)

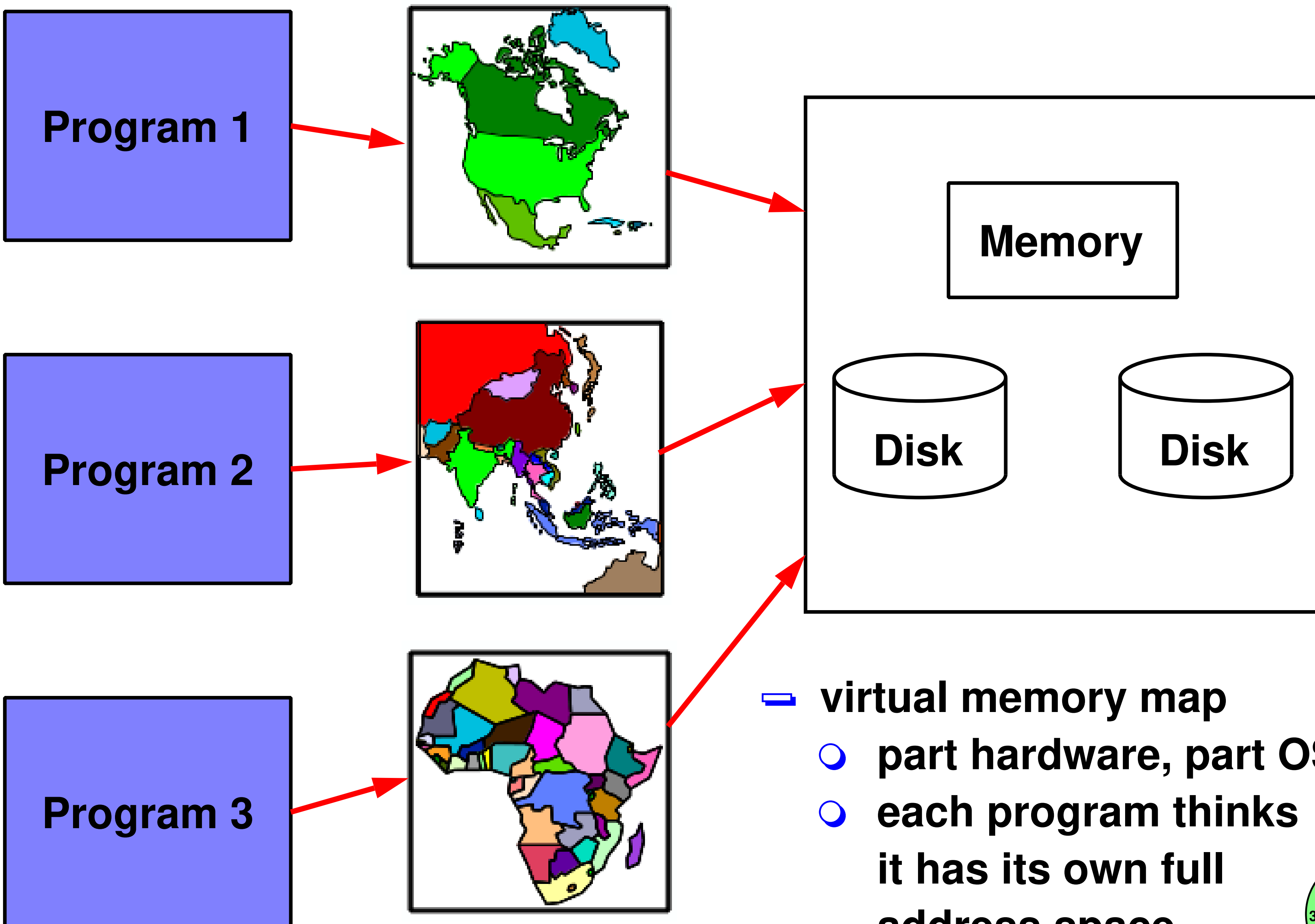
- **A Framework for Devices**
- **Low-level Kernel (will come back to talk about this after Ch 7)**
- **Processes & Threads**
- ***Storage Management***

Storage Management

- What physical "devices" can you use to store data?
  - *primary storage*, i.e., physical memory
    - "directly" addressable (using "one level of indirection")
      - physical memory is not considered a "device"
  - *secondary storage*, i.e., disk-based storage
    - to store files (i.e., implement the abstraction of "files")
    - to support *virtual memory*

- An application is only aware of *virtual memory* (it thinks virtual memory is real memory)
  - applications should not care about how much physical memory is available to it
  - there may not be enough physical memory for all processes
  - the OS makes sure that real primary storage is available *when necessary*
  - e.g., an application can allocate a 1GB block of memory while the machine only has 256MB of physical memory

Virtual Memory Map



- virtual memory map
  - part hardware, part OS
  - each program thinks it has its own full address space

Storage Management

- Two ways for an application to access secondary storage
  - using *sequential I/O:* `open()`, `read()`, `write()`, `close()`
  - using *block I/O:* `open()`, `mmap()`, `close()`

- Memory management concerns
  1) mapping virtual addresses to real ones
  2) determining which addresses are valid, i.e., refer to allocated memory, and which are not
  3) keeping track of which real objects, if any, are mapped into each range of virtual addresses
  4) deciding what should be kept in primary storage (RAM) and what to fetch from elsewhere

Memory Management Concerns

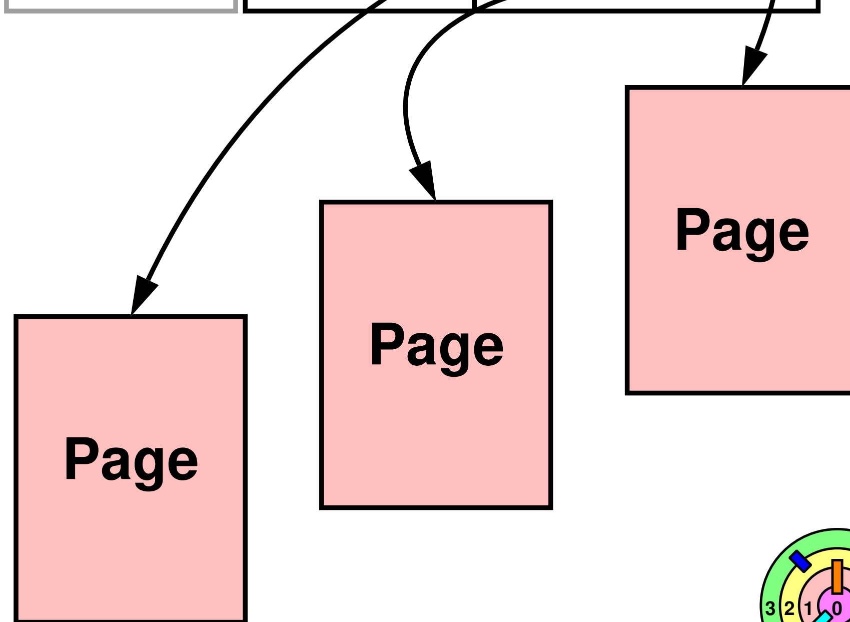
- In reality, the OS is too slow since *every* virtual address needs to be resolved
  - some of the virtual memory mechanisms must be built into the *hardware*
    - in some cases, the hardware is given the complete *"map"* (i.e., mapping from virtual to physical address)
    - in other cases, only a partial map is given to the hardware
    - in either case, OS needs to provide some map to the hardware and needs a *data structure* for the map
      - *page table* is part of the *virtual memory map* (it "maps" virtual address to physical address)
- *Virtual Memory Map (vmmap)* data structure in the OS
  - implements the *address space*
    - only *user part of the address space* needs to be represented
  - implements *memory-mapped files*

Memory Management Concerns

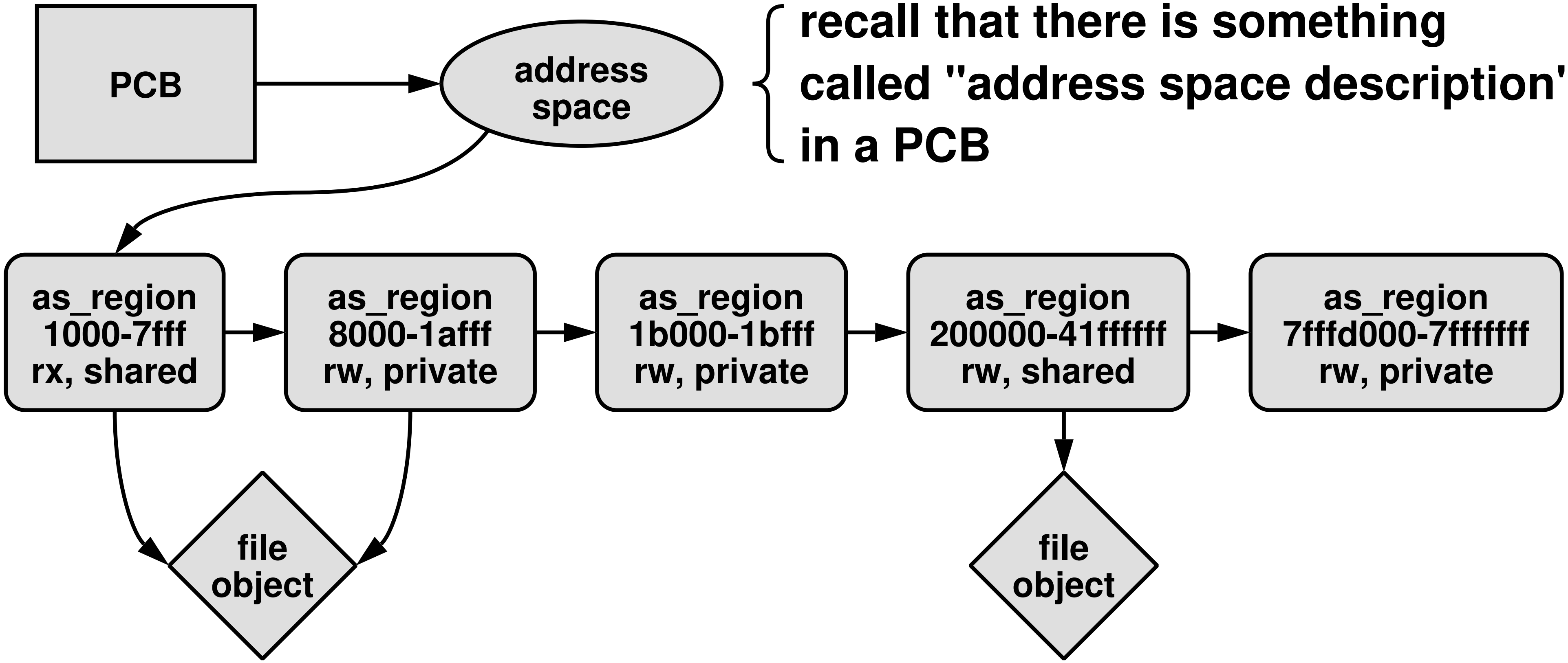
- A valid virtual address must be ultimately ***resolvable*** by the OS to a location in the physical memory
  - if it cannot be resolved, the virtual address is considered an ***invalid*** virtual address
  - referencing an ***unresolvable*** virtual address causes a ***segmentation fault*** (the OS will deliver SIGSEG to the process)
    - the default action would be to terminate the process
  - e.g., virtual address 0
- A ***page fault*** is not a ***segmentation fault*** if it ***can be resolved***

Page Table

Start	*Access*	*Physical Addr*
0	-	-
4096	R	•
8192	-	-
12288	R	•
16384	R/W	•



User Address Space Representation

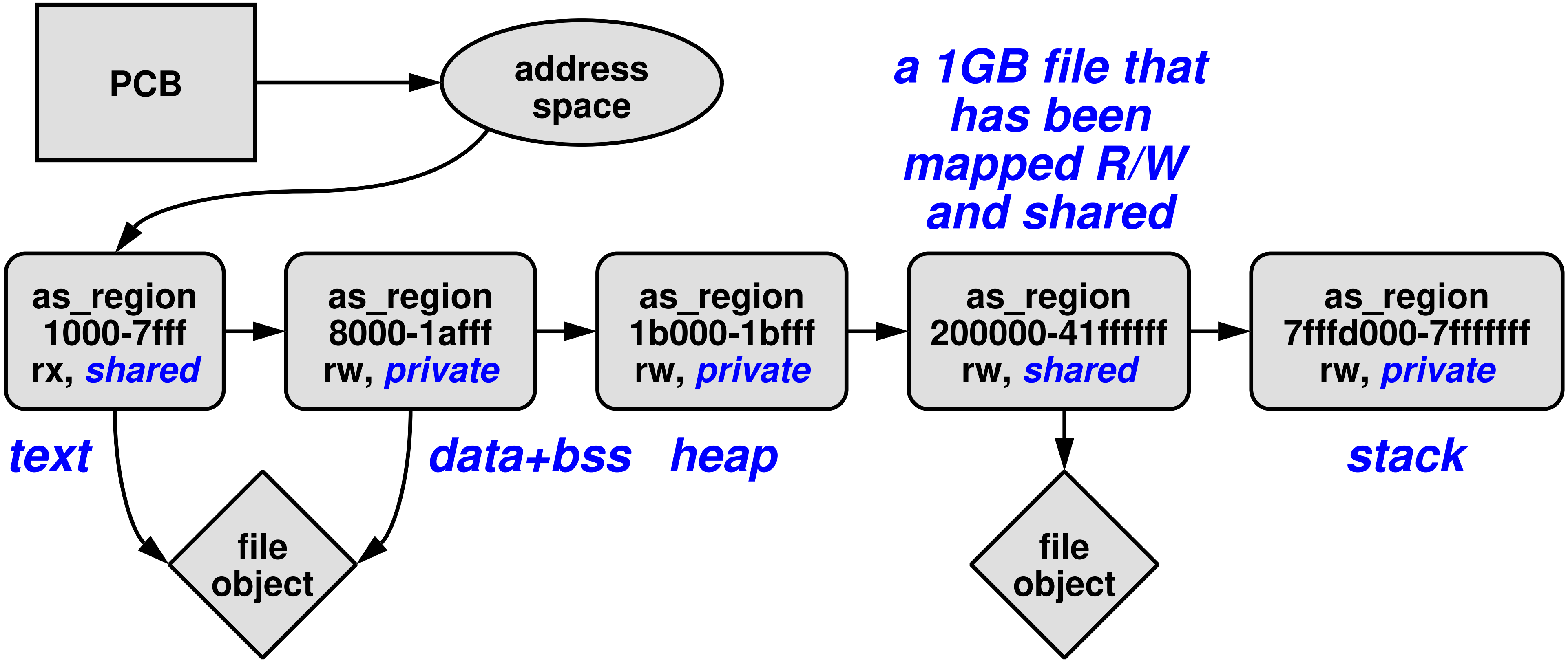


recall that there is something called "address space description" in a PCB

- `as_region` (address space region data structure) contains:
  - *start address*, *length*, *access permissions*, *shared* or *private*
  - if mapped to a file, pointer to the corresponding *file object*

- This is related to Kernel Assignment 3 where you need to create and manage *address spaces* / *virtual memory maps*

User Address Space Representation



- In this example, text and data map portions of the same file
  - *text* is marked read-execute and *shared*
  - *data+bss* is marked read-write and *private* to mean that changes will be private, i.e., will not affect other processes exec'ed from the same file

How OS Makes Virtual Memory Work?

- If a thread access a virtual memory location that's both in primary storage and mapped by the hardware's map
  - no action by the OS
- If a thread access a virtual memory location that's *not in primary storage* or if the *translation is not in the hardware's map*
  - a *page fault* is occurred and the OS is invoked
    - OS checks the `as_region` address space data structures to make sure the reference is valid
      - if it's valid, the OS does whatever that's necessary to locate or create the object of the reference
      - find, or if necessary, make room for it in primary storage if it's not already there, and put it there
      - fix up all the kernel data structures then return from page fault so that application can *retry the memory access*
    - if invalid, it turns into a *segmentation fault* (or bad page fault)

Storage Management

- **Two issues need further discussion**
  - **how is the *primary storage* managed (in terms of "resource management")?**
  - **how are the objects managed in *secondary storage*?**

How Is The Primary Storage "Resource" Managed?

- Who needs primary storage?
  - application processes
  - OS (e.g., terminal-handling subsystem, communication subsystem, I/O subsystem, etc.)
  - they *compete* for primary storage
- If primary storage is managed poorly
  - one subsystem can use up all the available memory
    - then other subsystem won't get to run
    - this can even lead to OS crash when a subsystem uses up all of physical memory
- If there are no mapped files, the solution can be simple
  - assign each process a fixed amount of primary storage
    - this way, they won't compete
    - but is it fair?

In Reality, Have To Deal With Mapped Files

- An example to demonstrate a dilemma
  - one process is using all of its primary storage allocation
  - it then maps a file into its address space and starts accessing that file
  - should the memory that's needed to buffer this file be charged against the files subsystem or charged against the process?
- If charged against the files subsystem
  - if the newly mapped file takes up all the buffer space in the files subsystem, it's unfair to other processes
- If charged against the process
  - if other processes are sharing the same file, other processes are getting a free ride (in terms of memory usage)
  - even worse, another process may increase the memory usage of this process (double unfair!)

In Reality, Have To Deal With Mapped Files

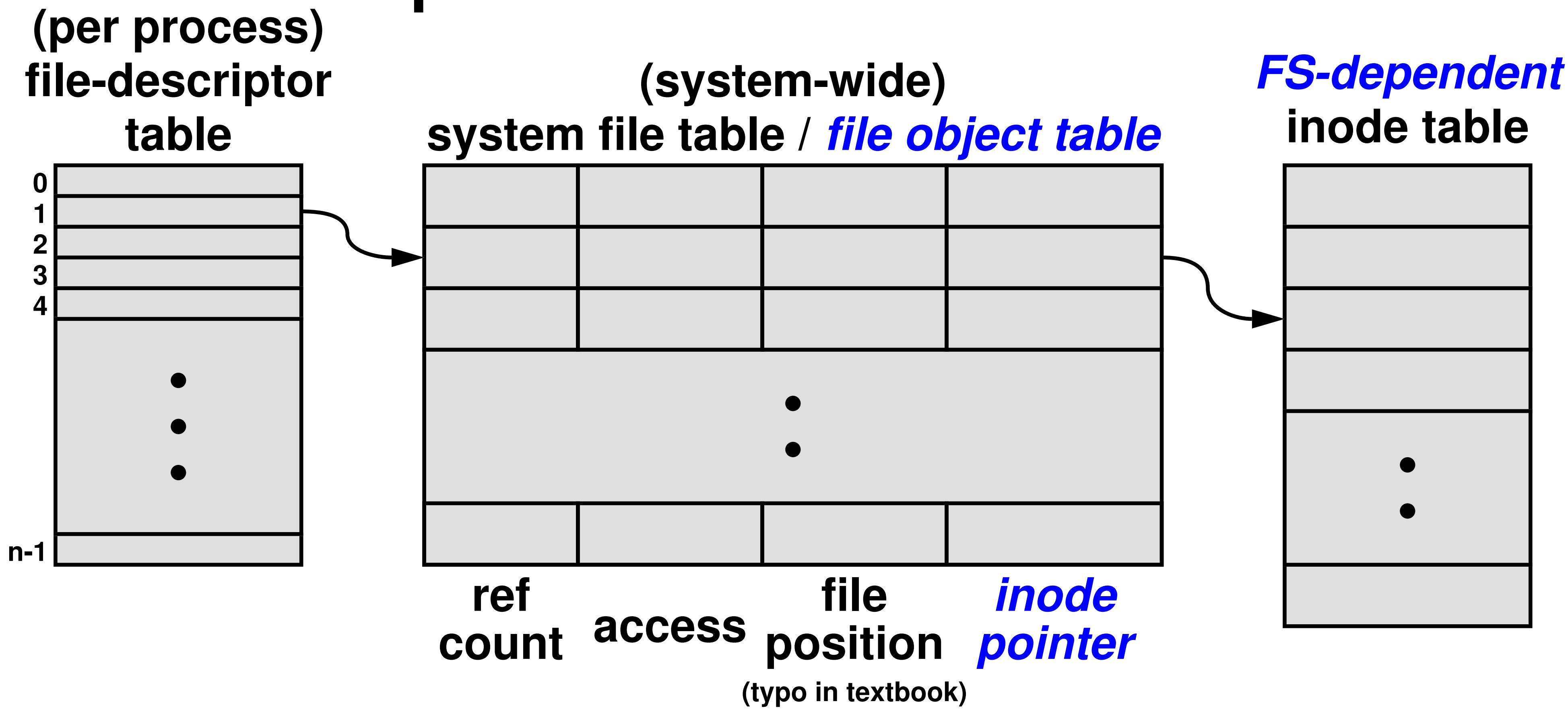
- It's difficult to be *fair*
  - it's difficult to even define what *fair* means
- We will discuss some solutions in Ch 7
  - for now, we use the following solution
    - give each participant (processes, file subsystem, etc.) a minimum amount of storage
    - leave some additional storage available for all to compete

How Are Objects Managed In Secondary Storage?

- The *file system* is used to manage objects in secondary storage
  - the term "file system" can mean two different things
    1) how to *layout data* on secondary storage (data structures on disk)
    2) how to access those data in data structures (code)

- The file system is usually divided into two parts
  - *file system independent*
    - supports the "file abstraction"
    - on Unix, this is called the *"virtual file system (VFS)"*
      - Kernel Assignment 2
    - on Windows, this is called the *"I/O manager"*
  - *file system dependent*
    - on Unix, this is called the *"actual file system (AFS)"*
      - e.g., ext2, ext3, ext4, etc.
    - on Windows, this is called the *"file system"*
      - e.g., FAT32, NTFS, etc.

Open-File Data Structures



- In the kernel, *each process* has its own *file-descriptor table*
  - the kernel also maintains *system file table (or file object table)*
- The *file object / inode* forms the *boundary* between *VFS* and the *AFS* (i.e., points to *file-system-dependent* stuff)
  - how can this be done?

File Object

- The file object is like an *abstract class* in C++
  - subclasses of file object are the *actual* file objects

```
class FileObject {
  unsigned short refcount, access;
  unsigned int file_pos;
  ...
  virtual int create(const char *, int, FileObject **);
  virtual int read(int, void *, int);
  virtual int write(int, const void *, int);
  ...
};
```

- But wait ...
  - what's this about C++?
    - real operating systems are written in C ...
    - checkout the `DRIVERS` kernel documentation (we skipped this `weenix` assignment)
      - similar trick (polymorphism) used in VFS

File Object in C

```
typedef struct {
  unsigned short refcount, access;
  unsigned int file_pos;
  ...
  void **file_ops; /* to array of function pointers */
} FileObject;
```

- A file object uses an *array of function pointers*
  - this is how C implements *C++ polymorphism*
  - one function pointer for each operation on a file
  - where they point to is (actual) file system dependent
  - but the (virtual) interface is the same to higher level of the OS

- Loose coupling between the actual file system and storage devices
  - the actual file system is written to talk to the devices also in a *device-independent* manner
    - i.e., using major and minor device numbers to reference the device and using standard interface provided by the device driver

File System Cache

- Storage *devices* are slow
  - disks are particularly slow
  - use a lot of tricks to make them look and feel fast
- *Recently used blocks* in a file are kept in a *file system cache*
  - the primary storage holding these blocks might be mapped into one or more address spaces of processes that have this file mapped
    - blocks are available for immediate access by read and write system calls
- Fancier data structures in storage system and file system
  - e.g., *hash table* can be used to locate file blocks in the cache
    - maybe keyed by *inode number*
- More details in Ch 6

1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- Managing Processes
- Loading Program Into Processes
- *Files*
  - (will skip first 13 slides since they overlap with "A Simple OS" slides)

Files

- Our primes program wasn't too interesting
  - it has no output!
  - cannot even verify that it's doing the right thing
  - other program cannot use its result
  - how does a process write to someplace outside the process?

- The notion of a ***file*** is our Unix system's sole abstraction for this concept of "someplace outside the process"
  - modern Unix systems have additional abstractions

- Files
  - abstraction of persistent data storage
  - means for fetching and storing data outside a process
    - including disks, another process, keyboard, display, etc.
    - need to ***name*** these different places
      - hierarchical naming structure
    - part of a process's ***extended address space*** (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 Handles (File Descriptors)

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

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

Standard File Descriptors

- Standard File Descriptors
  - 0 is stdin (by default, the keyboard)
  - 1 is stdout (by default, the display)
  - 2 is stderr (by default, the display)

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

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

Back to Primes

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

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

  - the output is not readable by human

Human-Readable Output

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

Allocation of File Descriptors

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

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

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

Running It

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

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

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

I/O Redirection

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

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

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

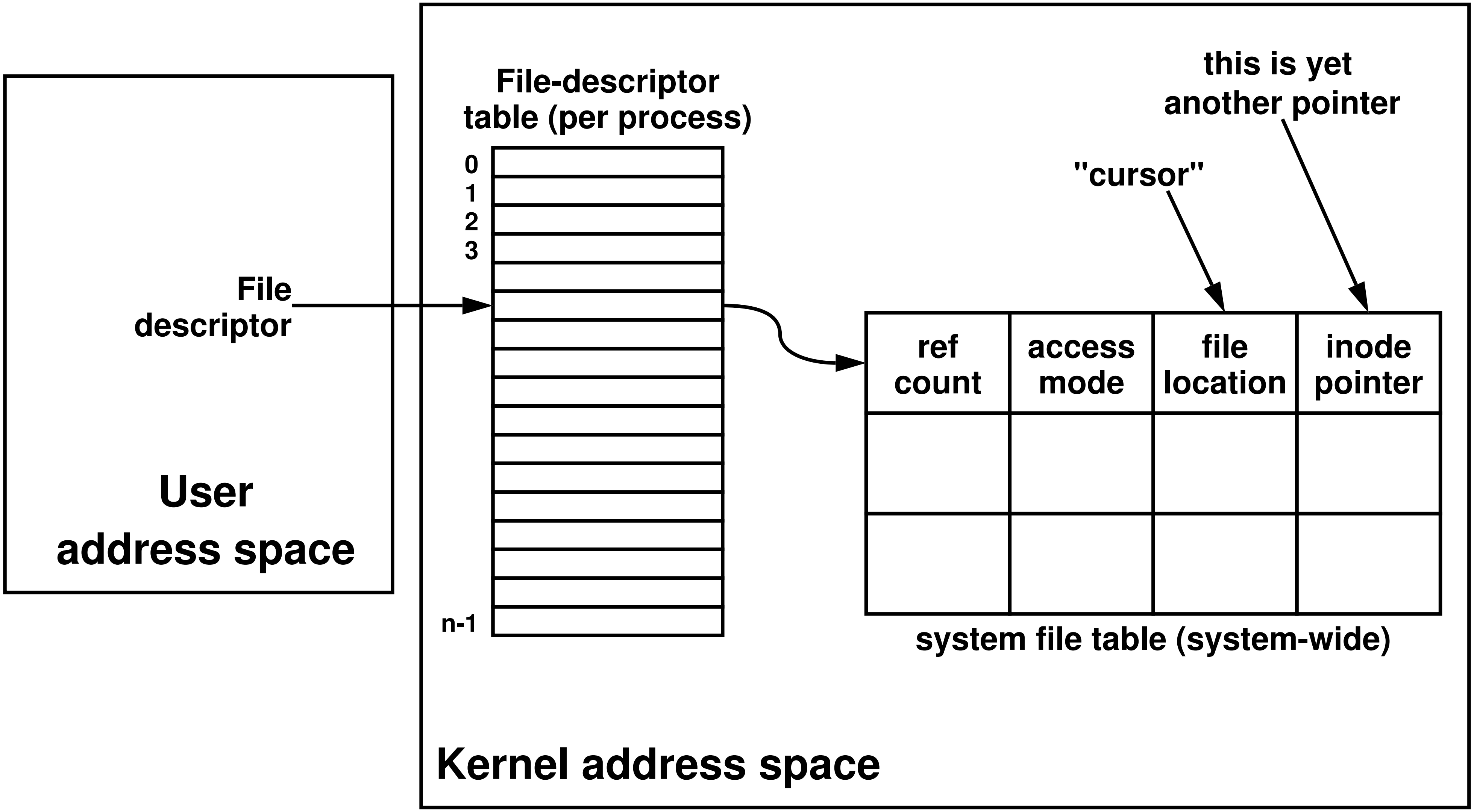
  - when the "`cat`" program reads from file descriptor 0, it would get the data byes from the file "/home/bc/Output"

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 accesses (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

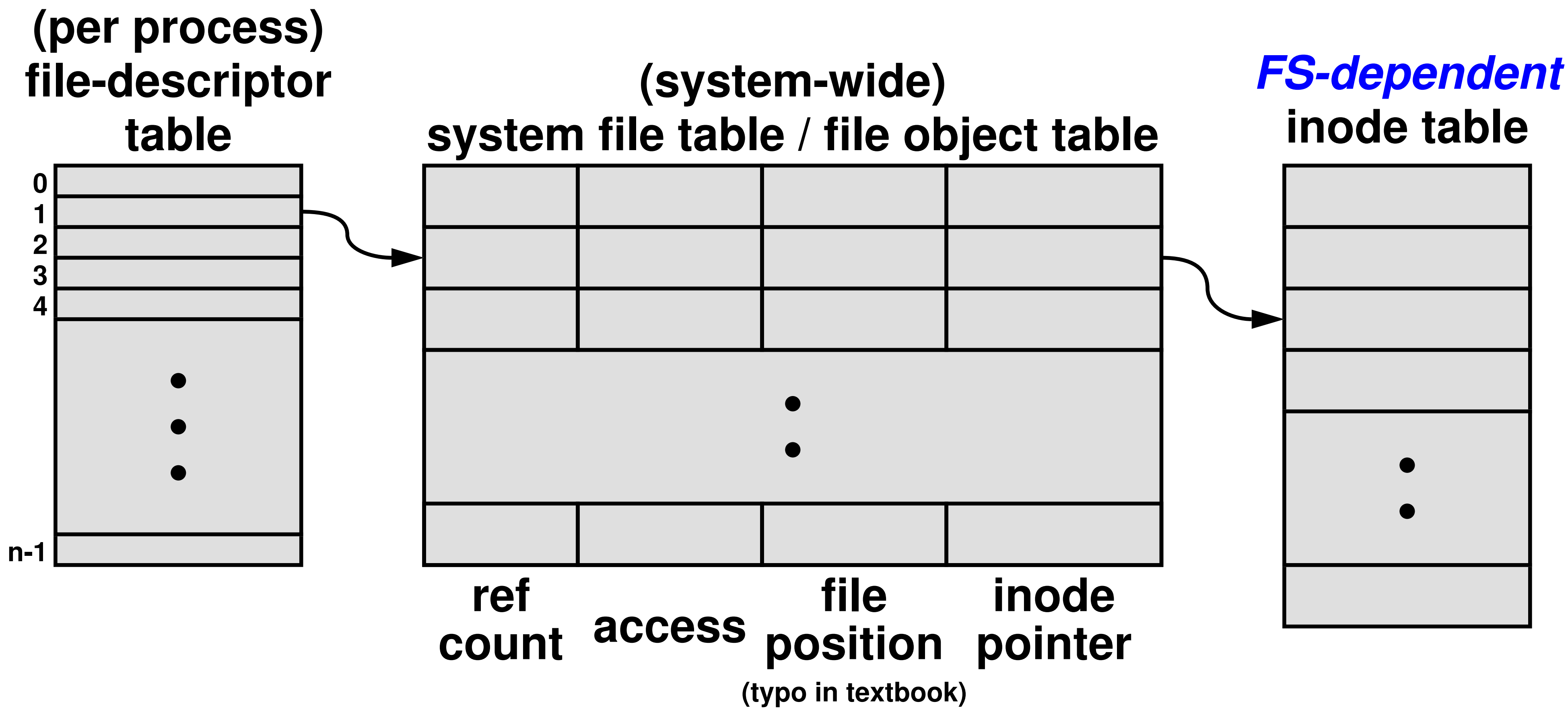
File-Descriptor Table



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

VFS vs. AFS

- We now know more about the "inode pointer"...



- We will focus on VFS in Ch 1
  - AFS will be covered in Ch 6

Put It All Together

`open()`

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

Applications

OS

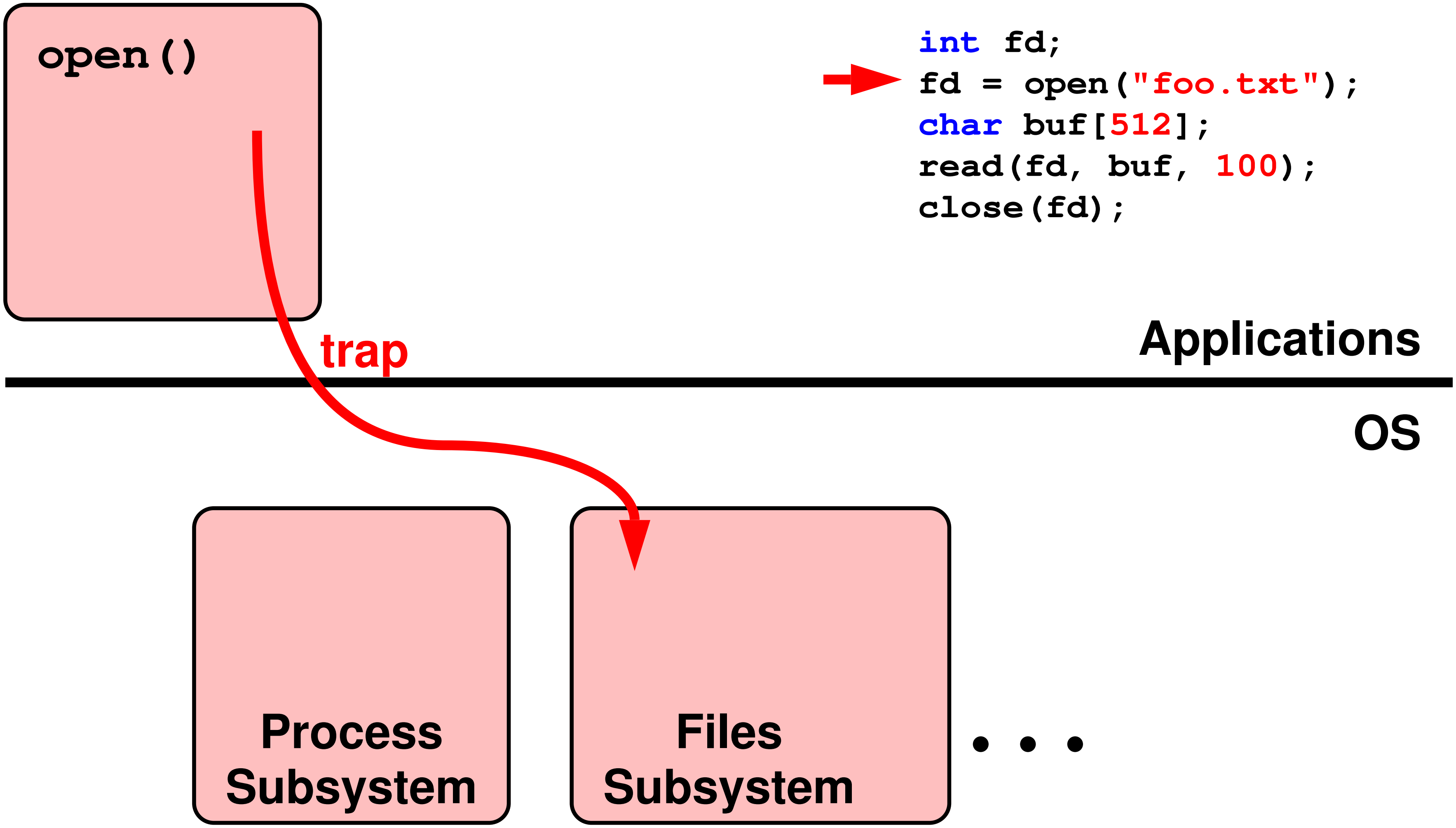
Process Subsystem

Files Subsystem

. . .

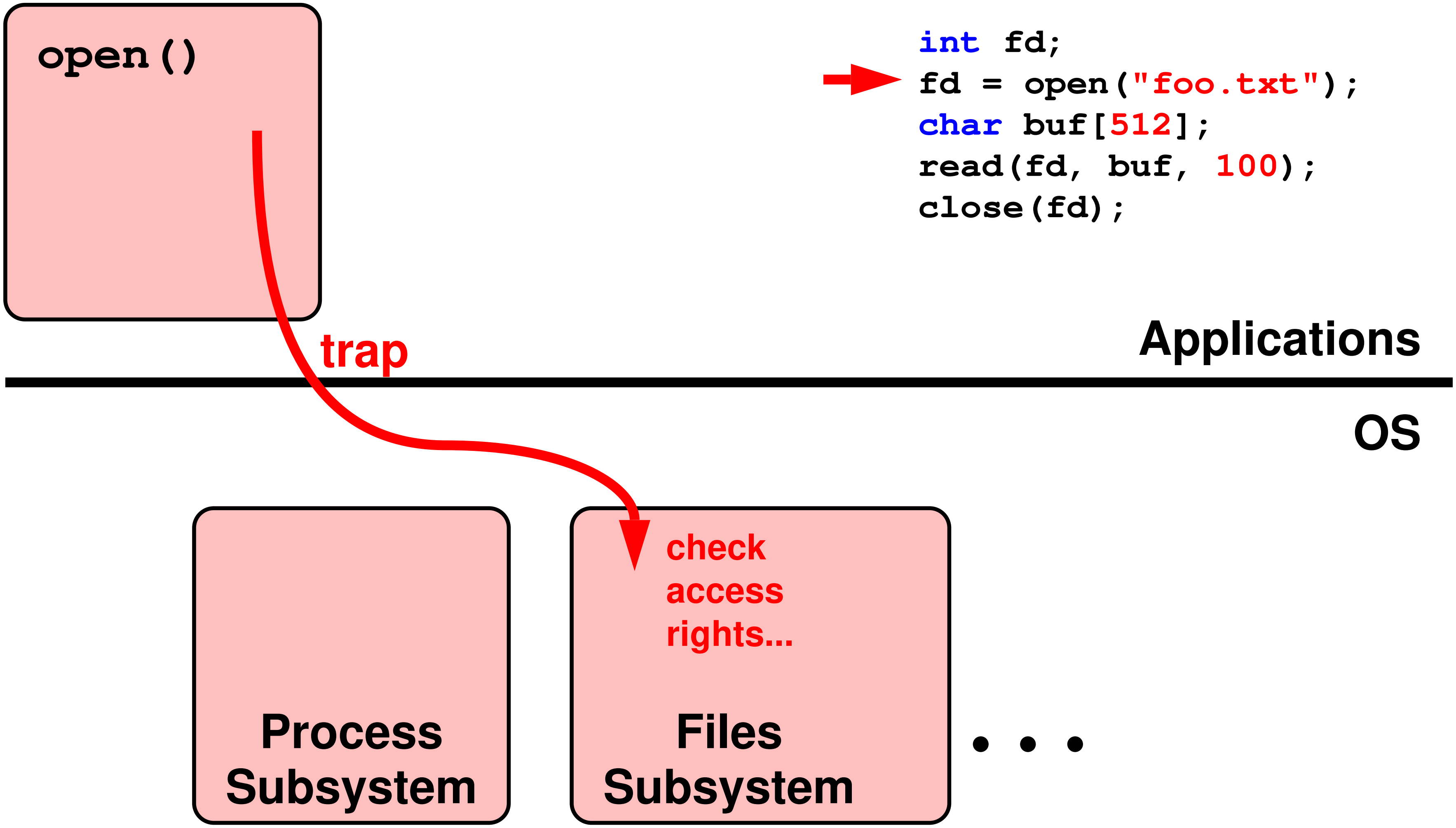
Put It All Together

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



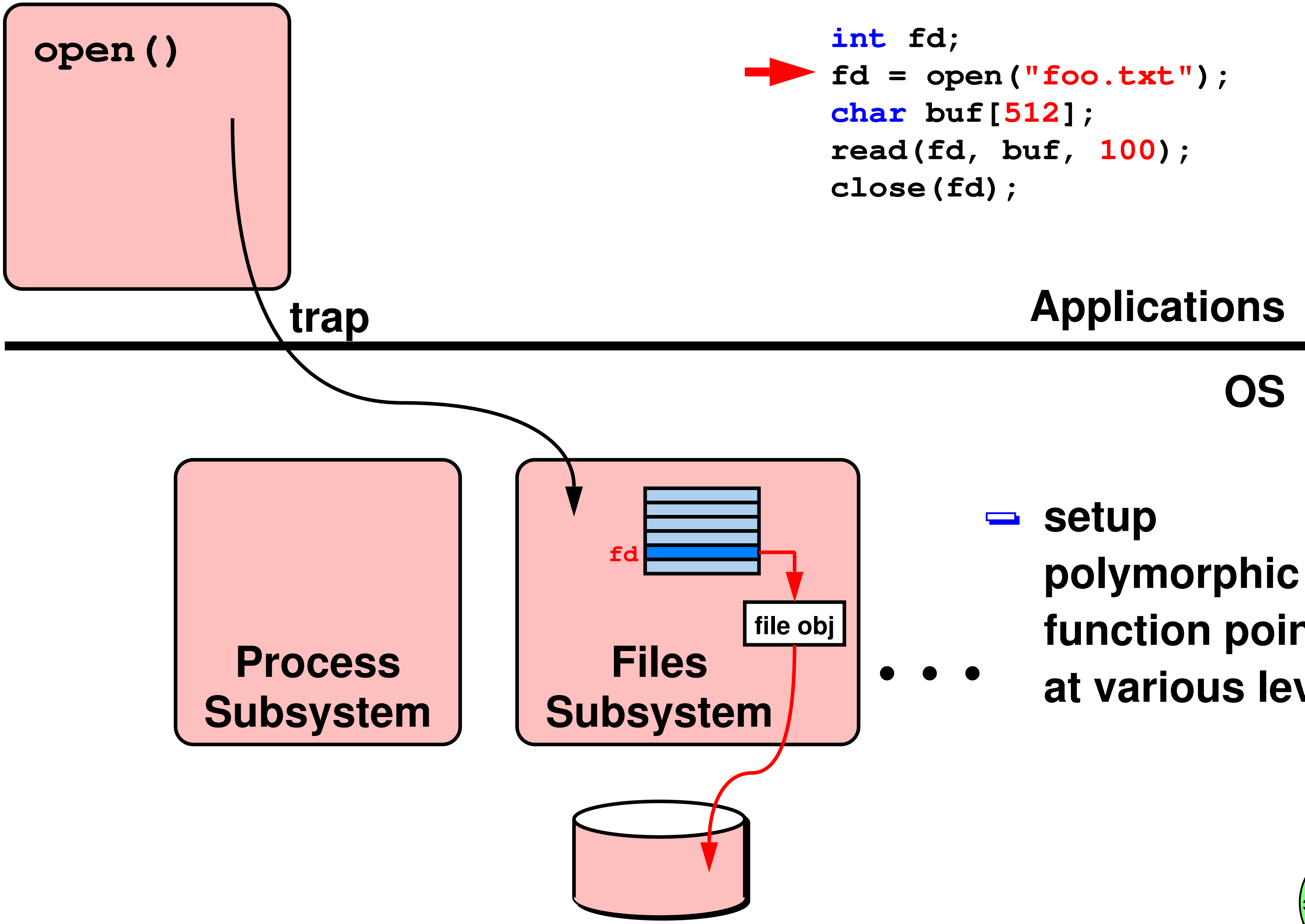
Put It All Together

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



Put It All Together

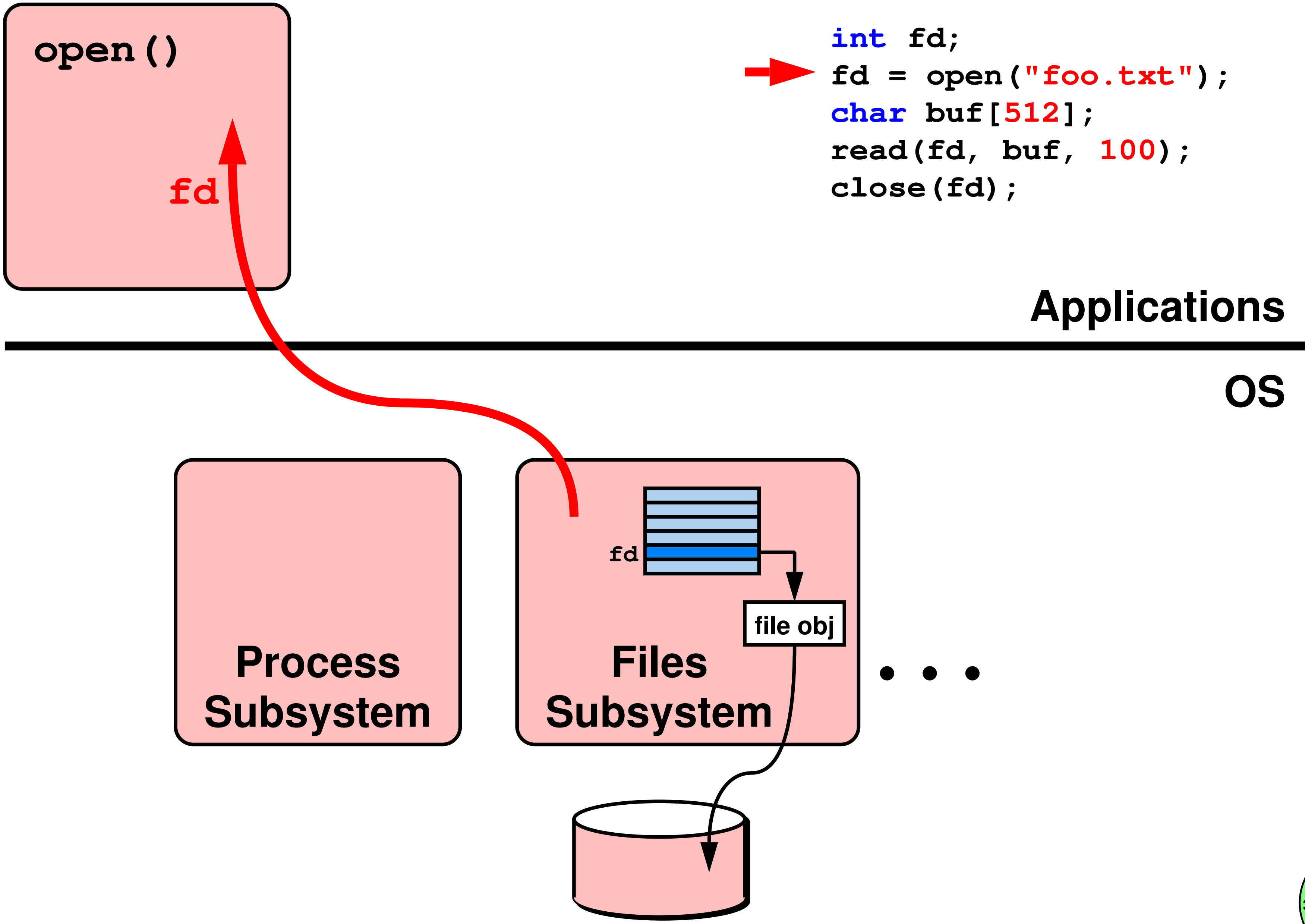
```
int fd;
fd = open("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```



- setup polymorphic function pointers at various levels

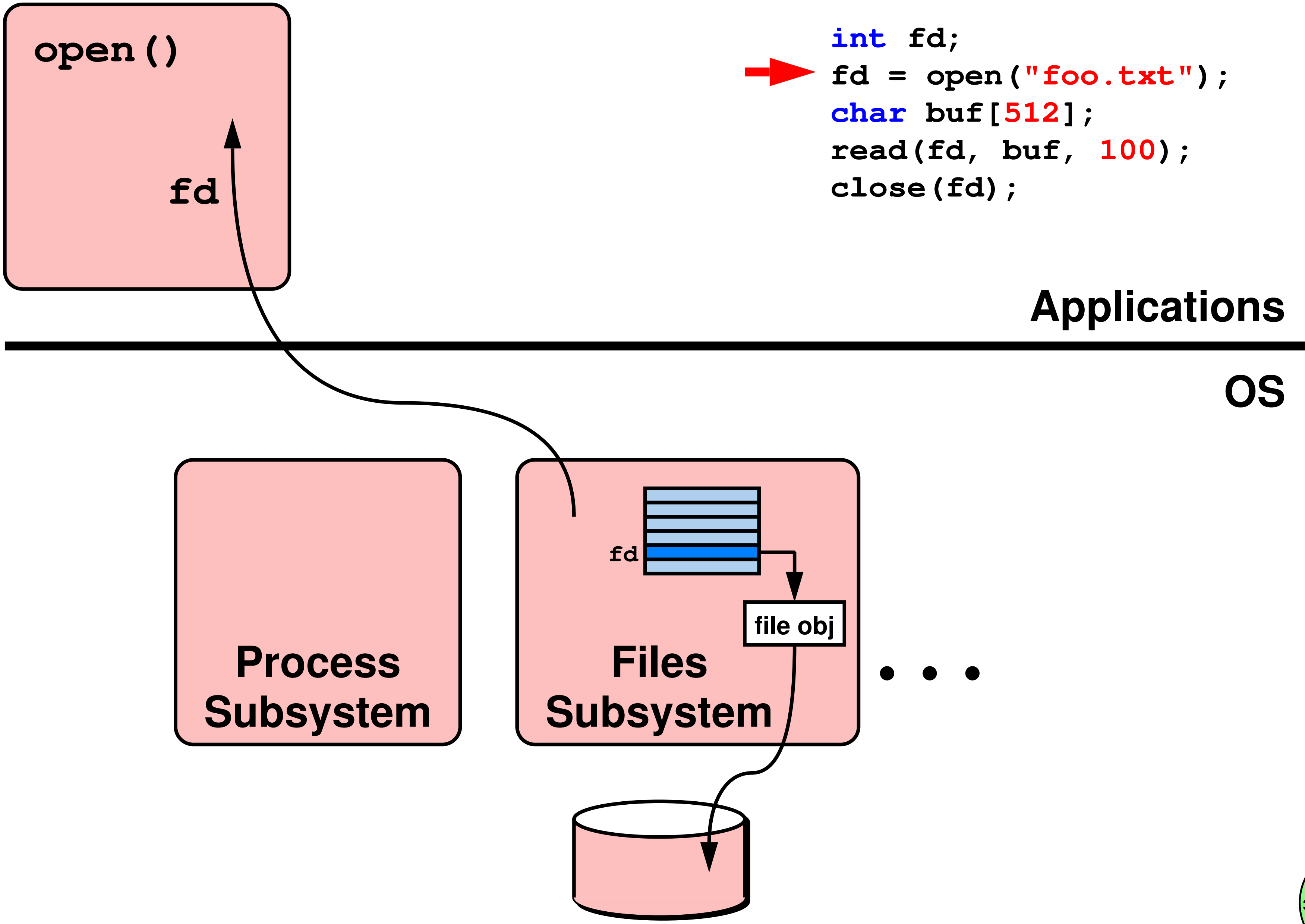
Put It All Together

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



Put It All Together

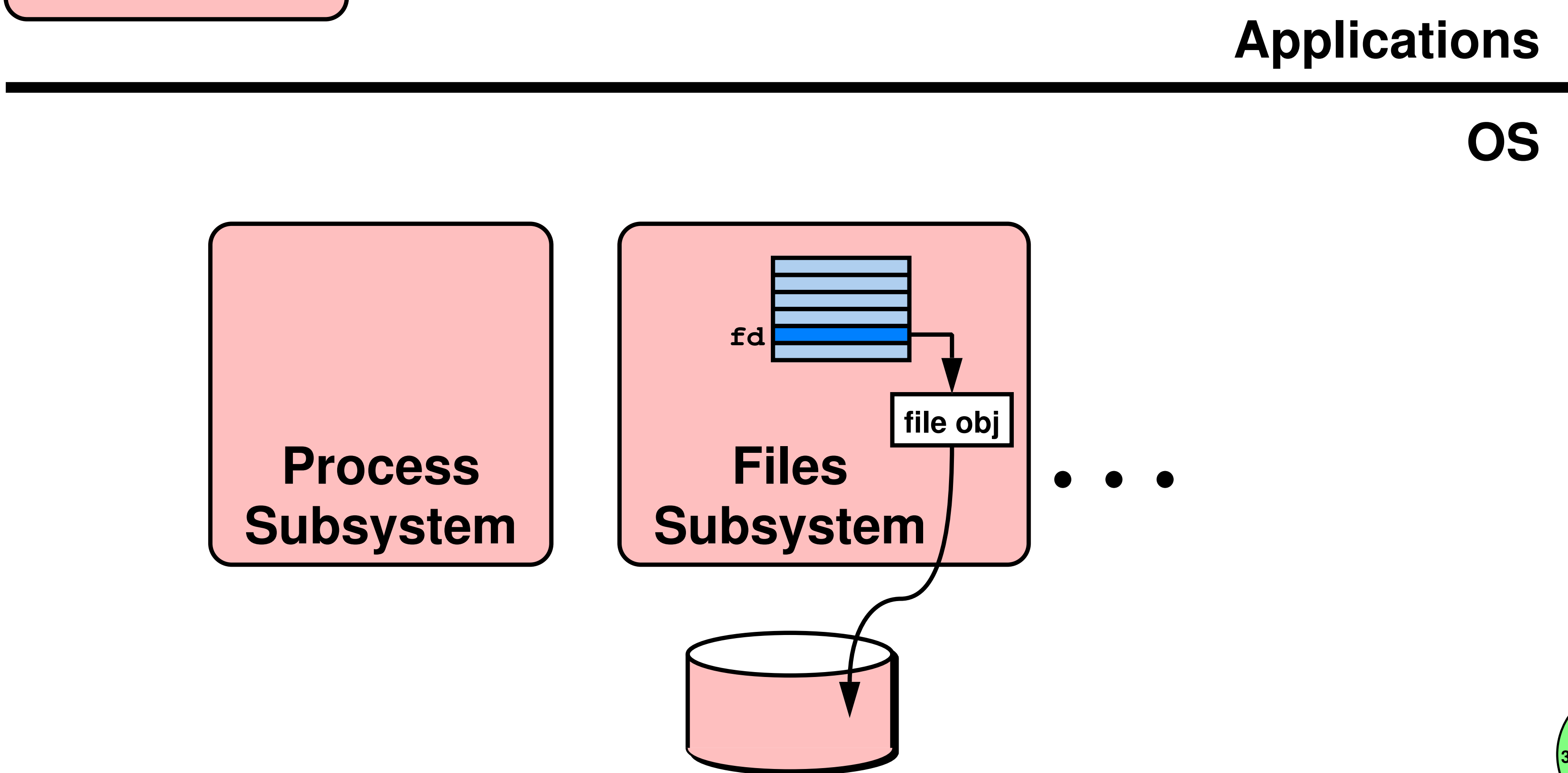
```
int fd;
fd = open("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```



Put It All Together

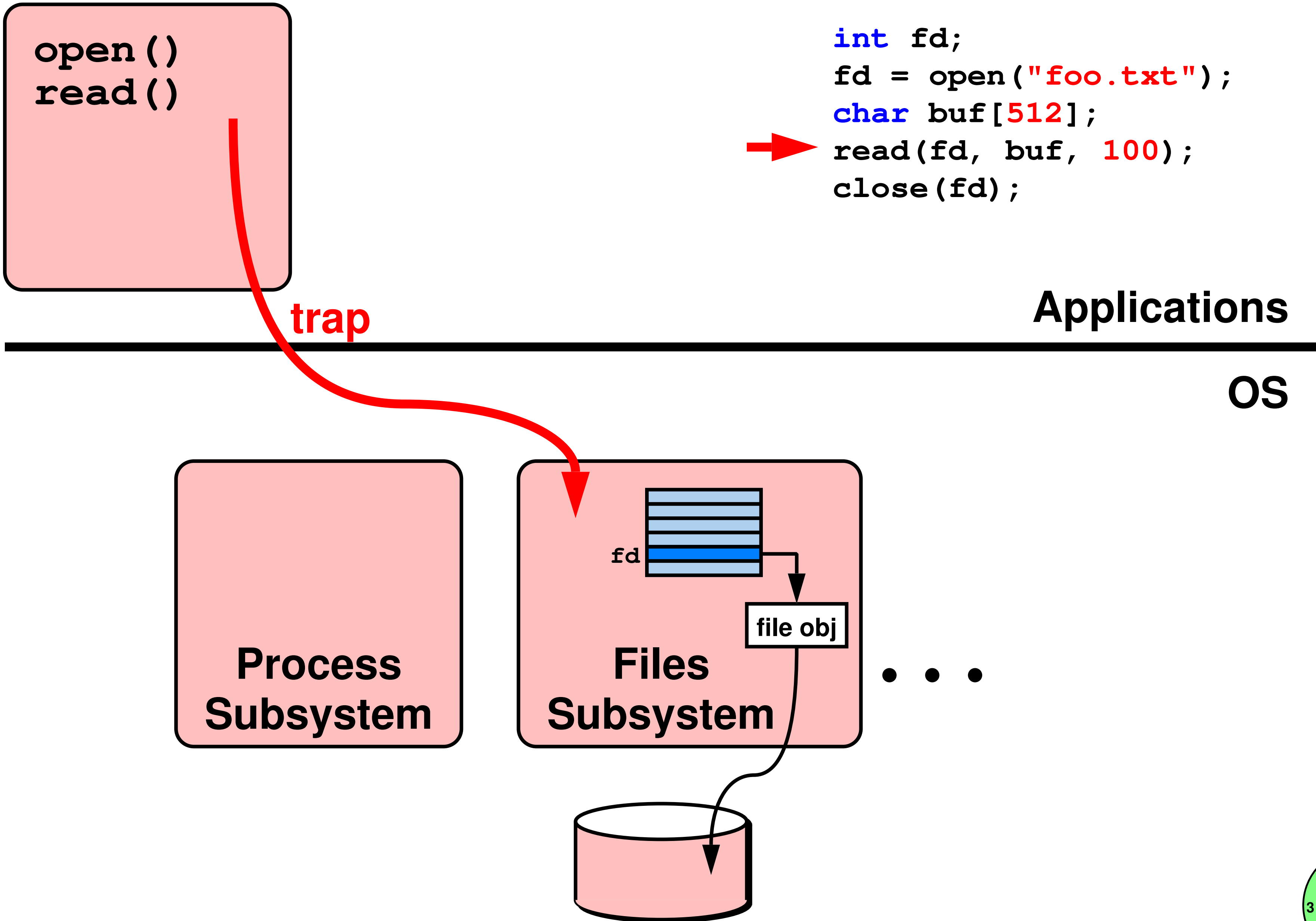
```
open()
read()
```

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



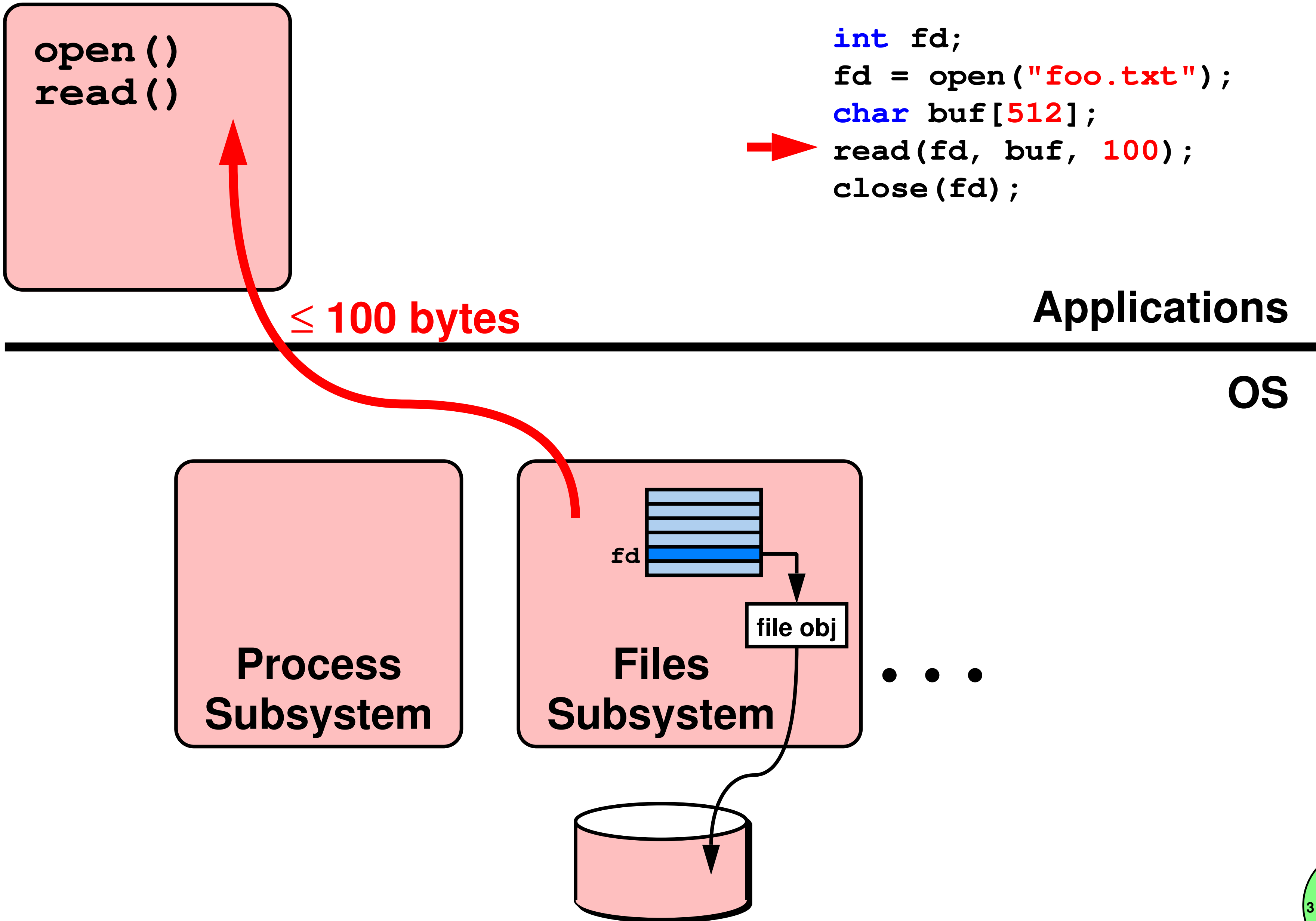
Put It All Together

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



Put It All Together

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



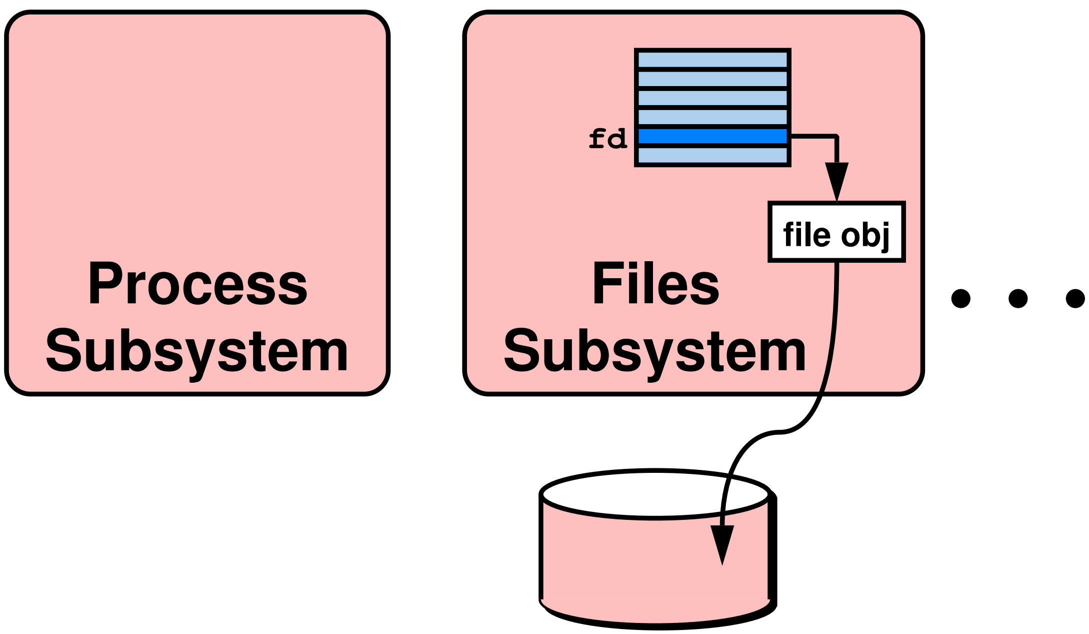
Put It All Together

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

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

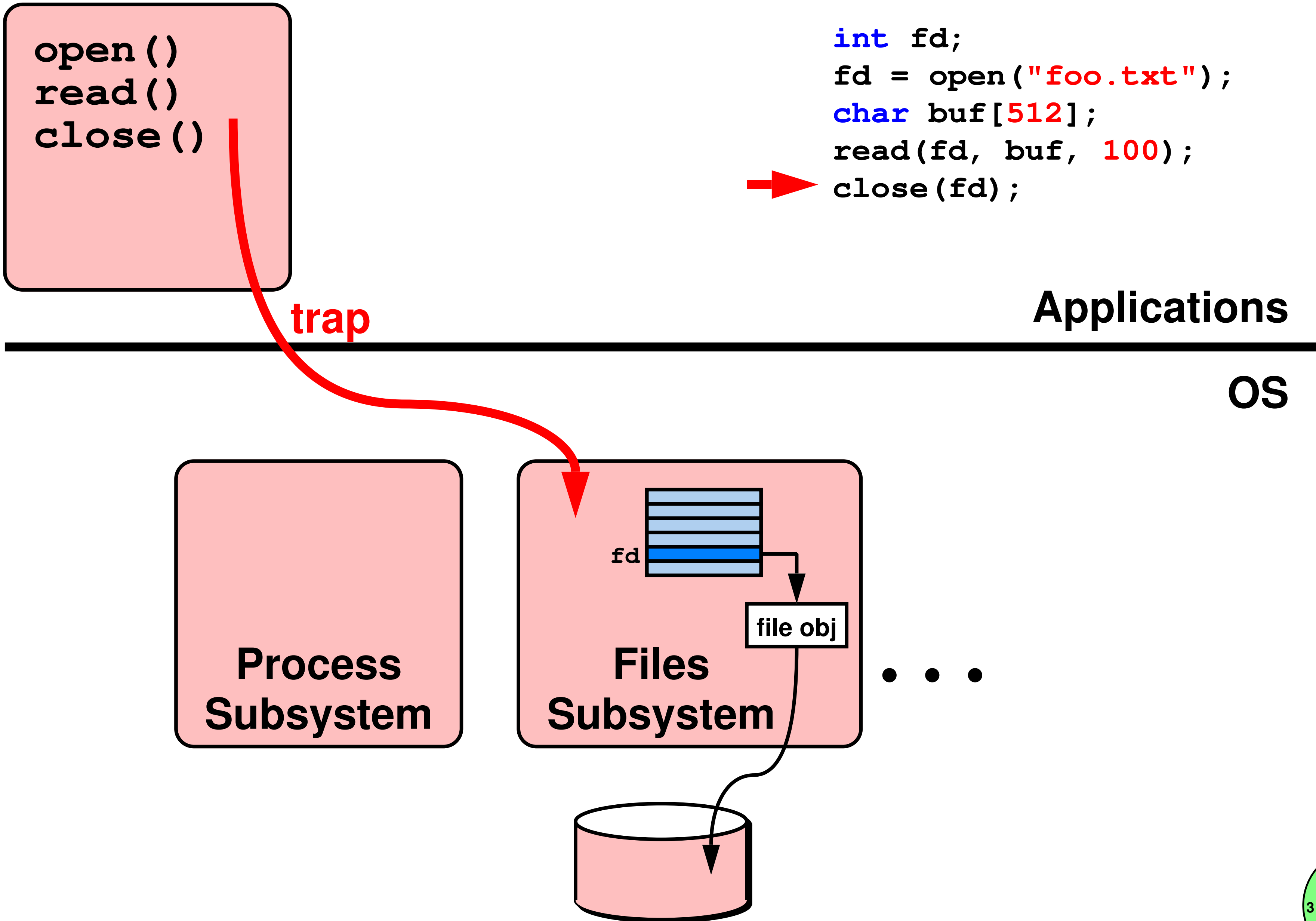
Applications

OS



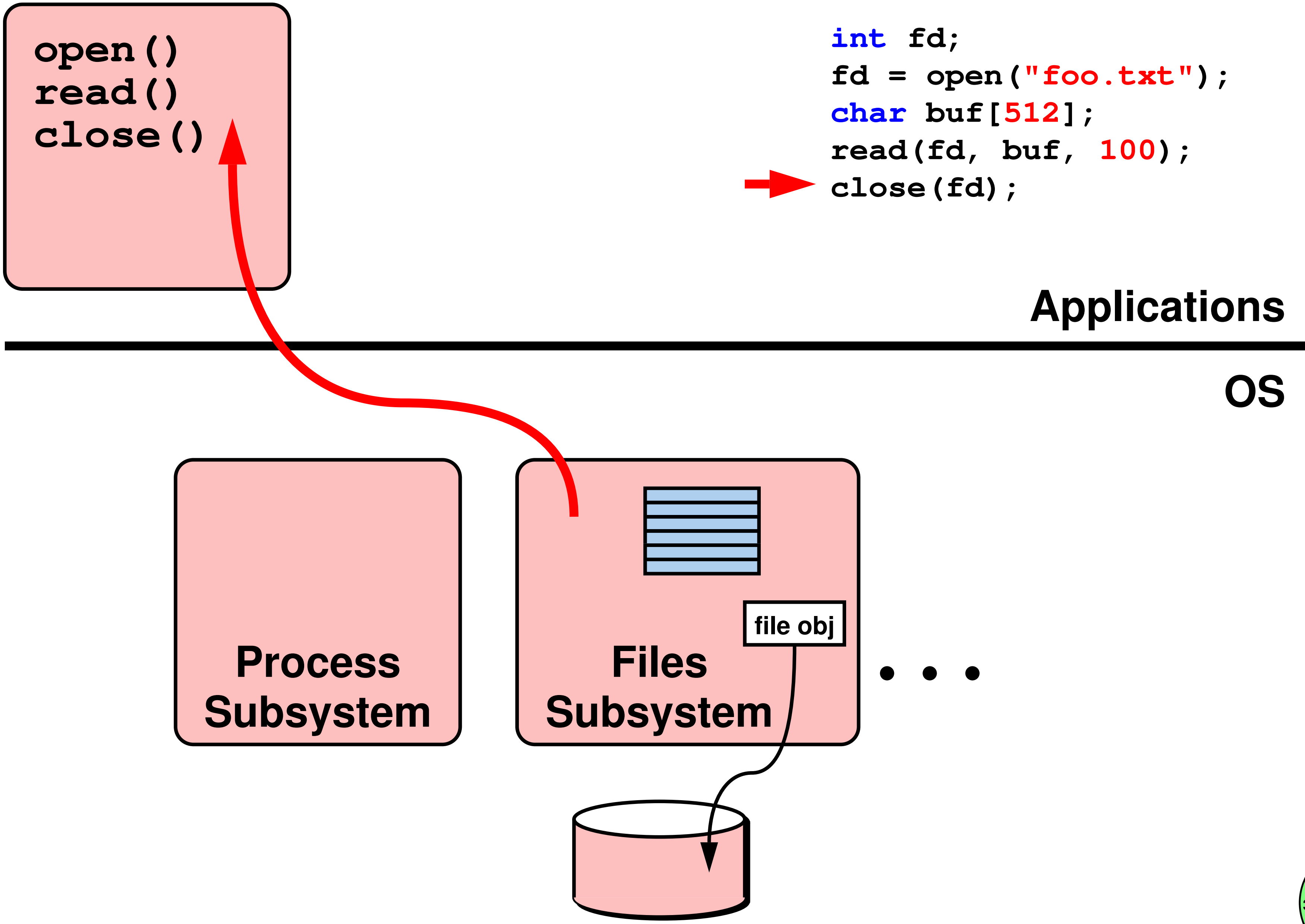
Put It All Together

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

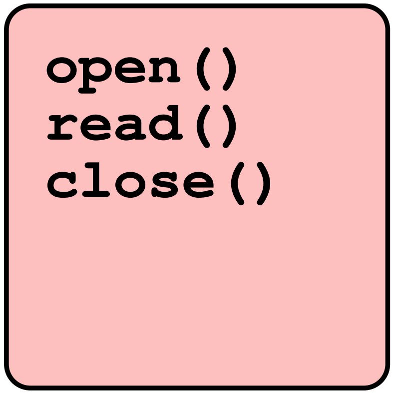


Put It All Together

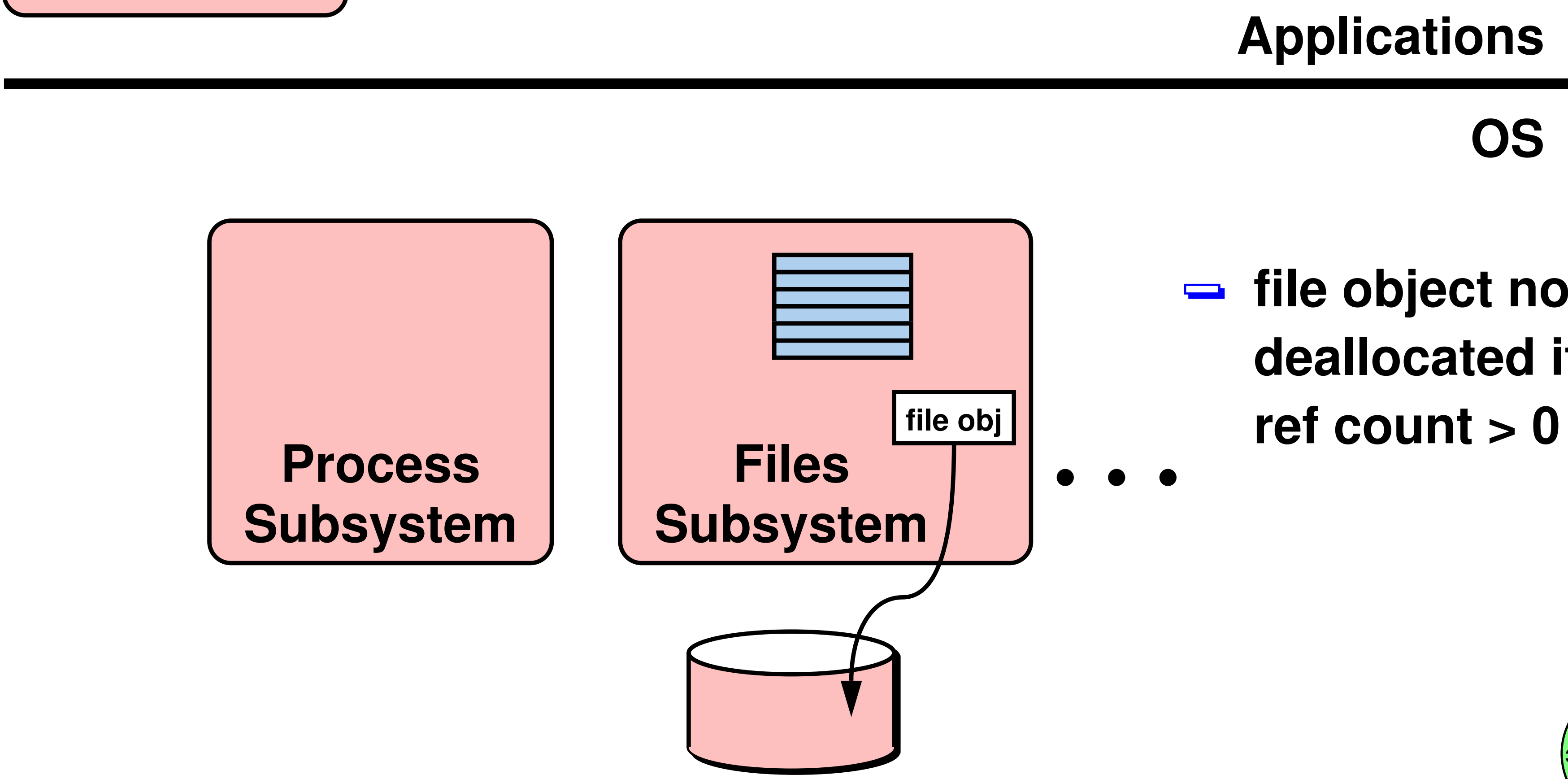
```
int fd;
fd = open("foo.txt");
char buf[512];
read(fd, buf, 100);
close(fd);
```



Put It All Together




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



- file object not deallocated if ref count > 0