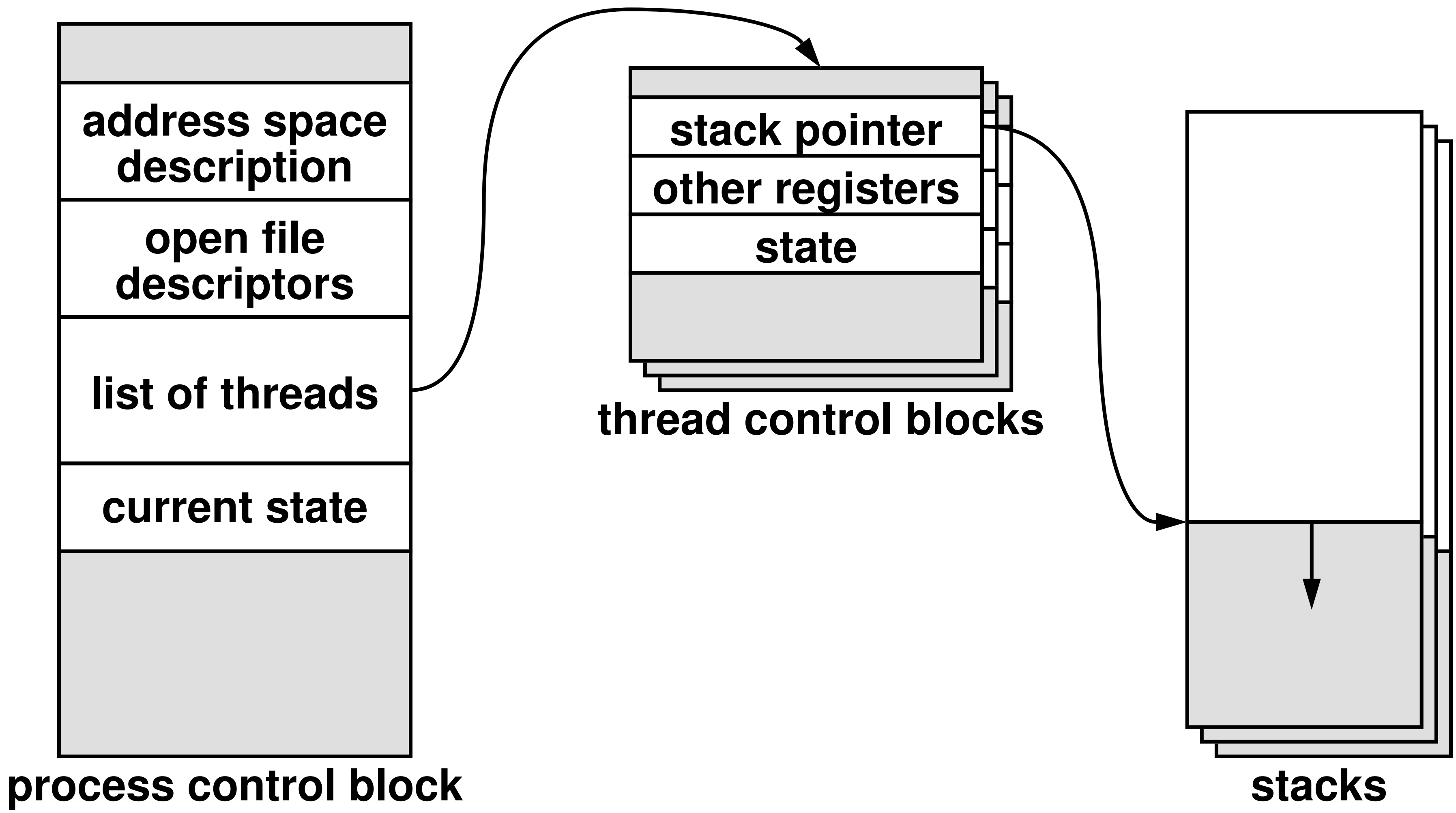
Housekeeping (Lecture 12 - 10/7/2013)

- **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
- **Any system issue, please get it resolved NOW**
  - come to office hours to get help
- **We did pretty well with kernel group forming**
  - there is only one student in each section who does not have partners
  - I'm still waiting for the student to let me know if I should let the class know who they are so you can contact him/her directly
  - even if your team already has 4 students, you can add this student to your team!
- **Post questions about the kernel assignments to class Google Group**
  - extra credit for posting good responses

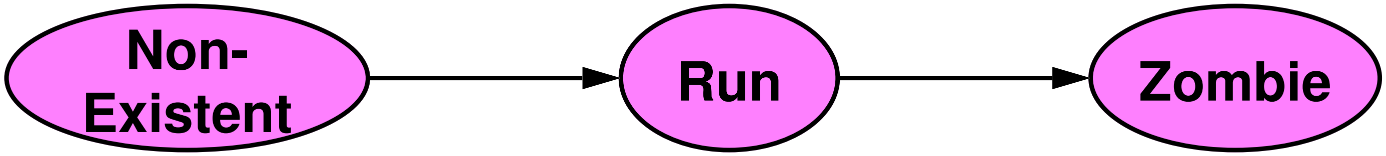
Processes and Threads



Note: all these are relevant to your Kernel Assignment 1

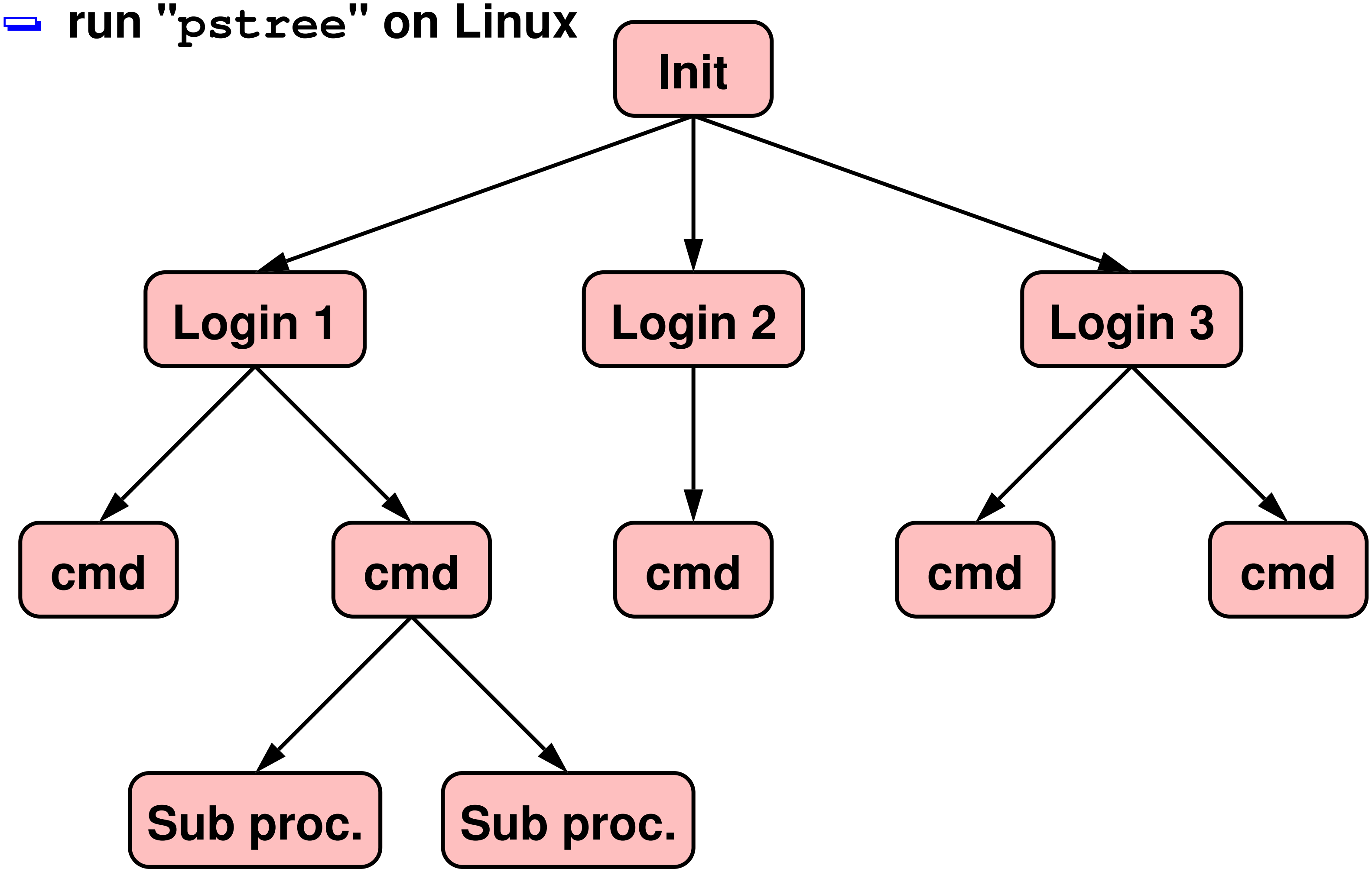
Process Life Cycle

- Pretty simple

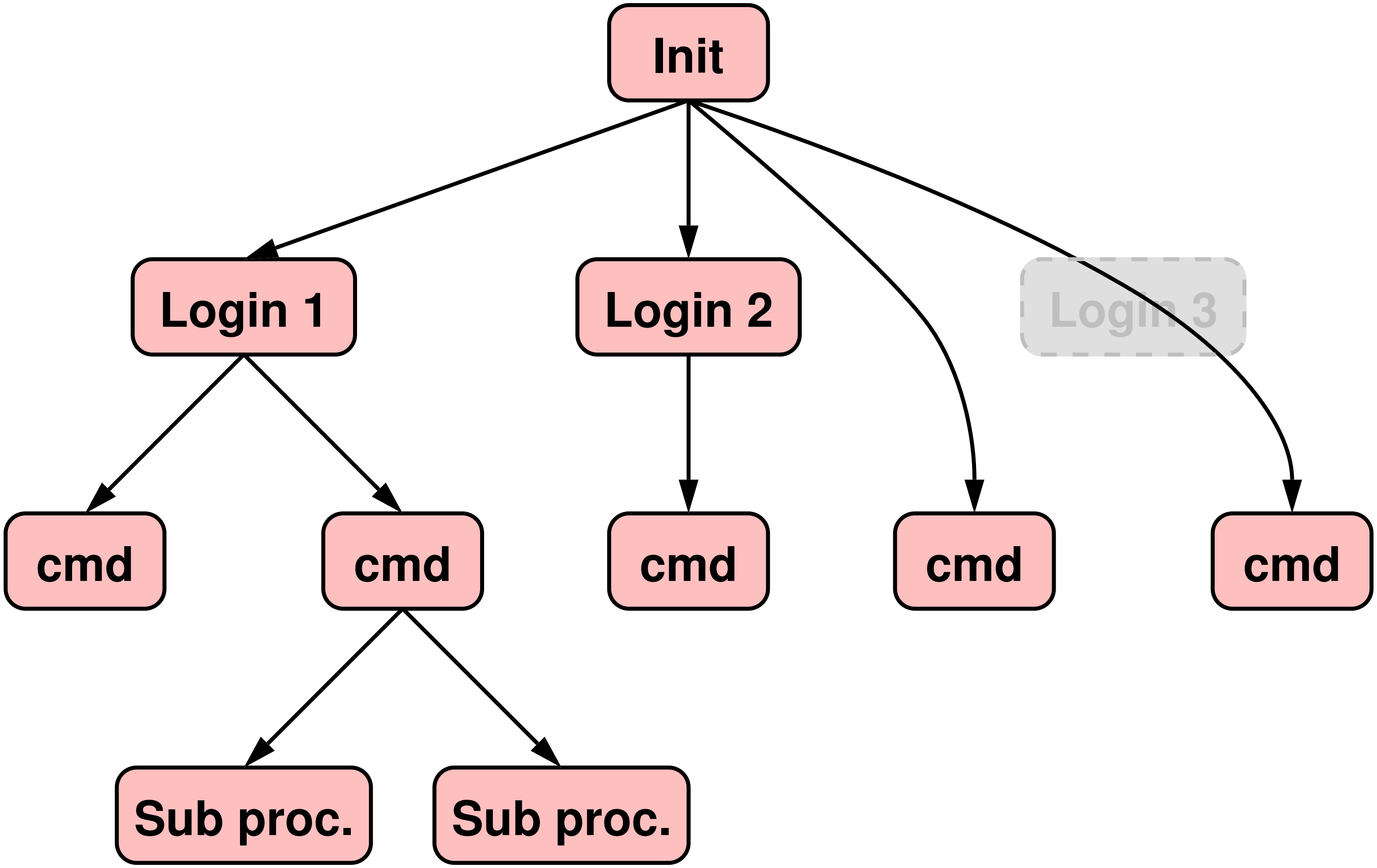


Process Relationships (1)

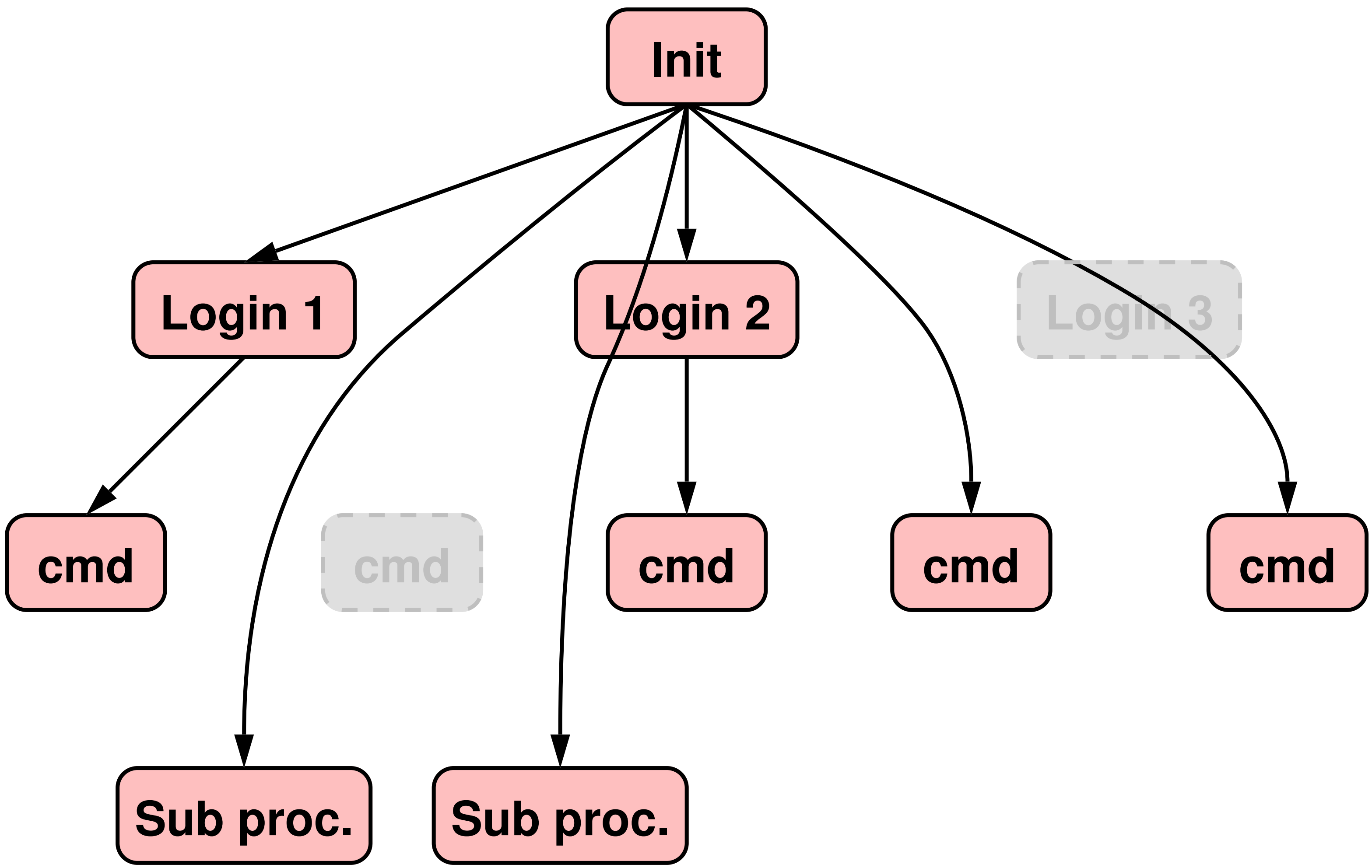
- Process hierarchy
  - run "`pstree`" on Linux



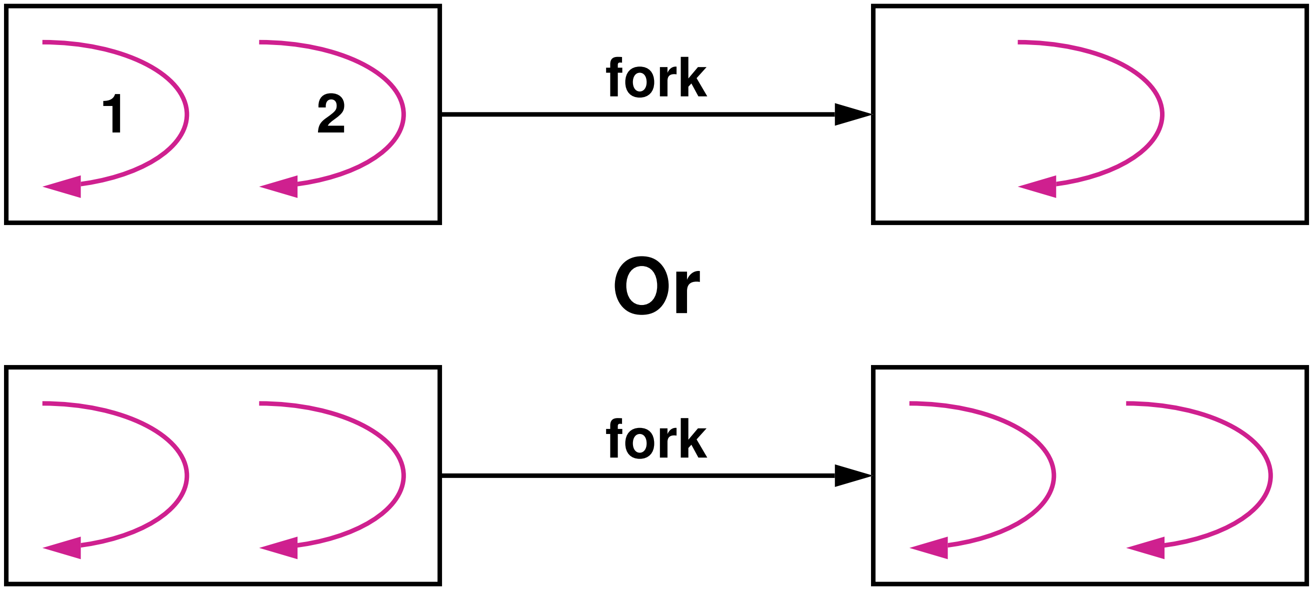
Process Relationships (2)



Process Relationships (3)

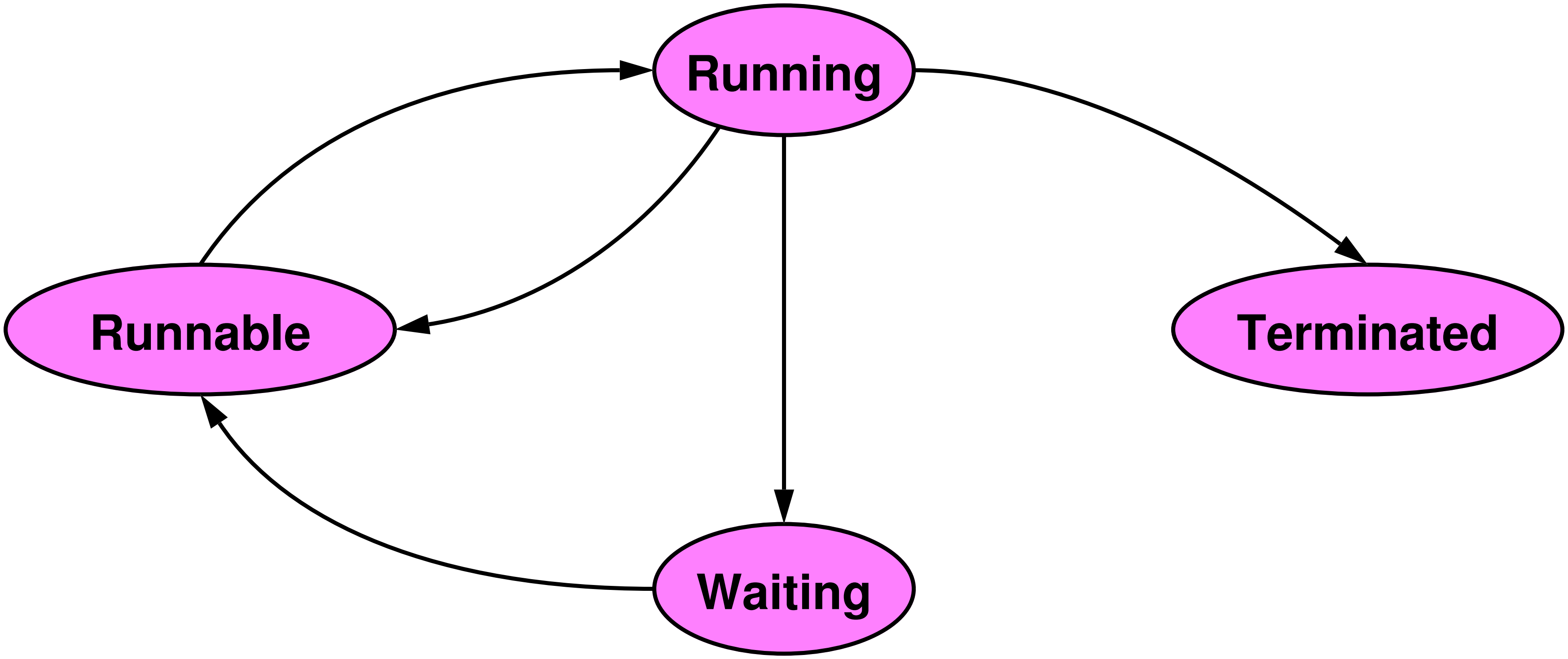


Fork and Threads



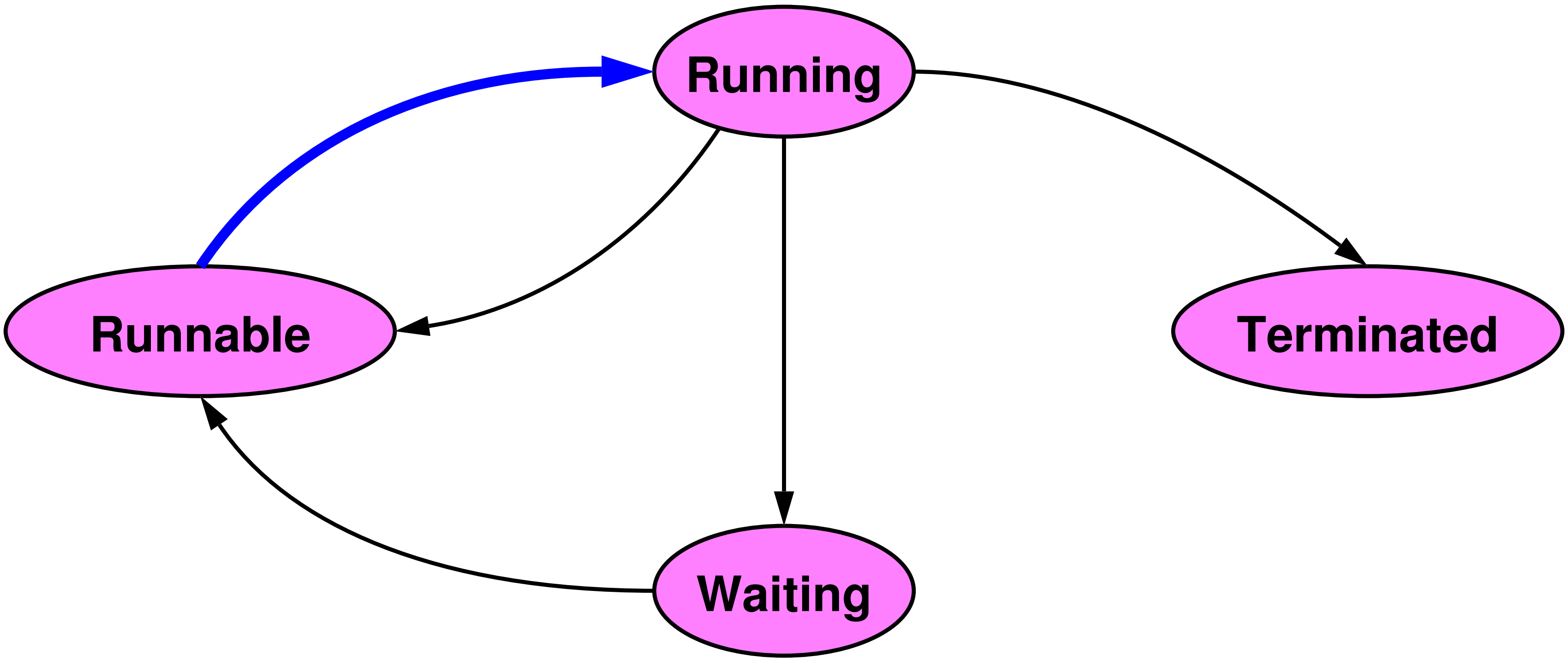
- Solaris uses the 2nd approach
  - expensive to fork a process
- Problem with 1st approach
  - thread 1 called `fork()` and thread 2 has a mutex locked
    - who will unlock the mutex?
  - POSIX solution is to provide a way to unlock all mutex before `fork()`

Thread Life Cycle



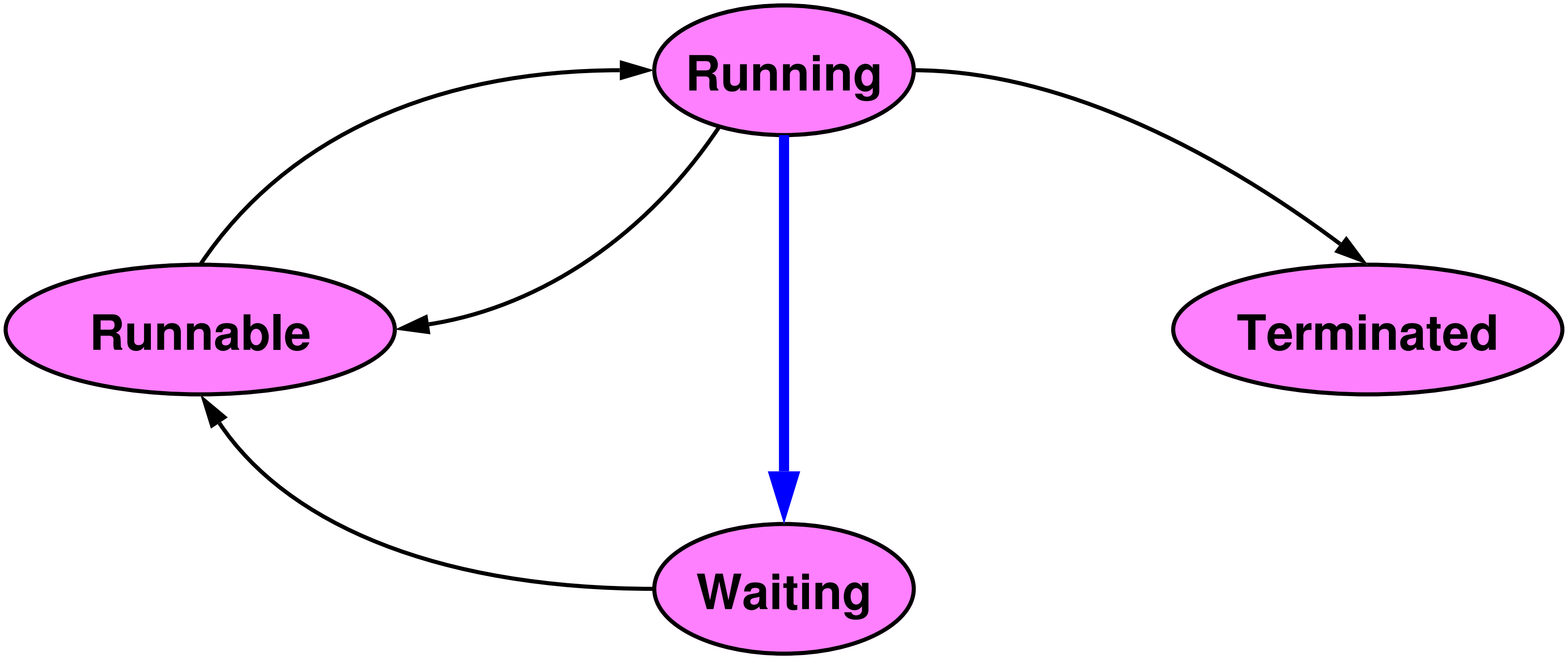
- a thread starts in the *runnable* state

Thread Life Cycle



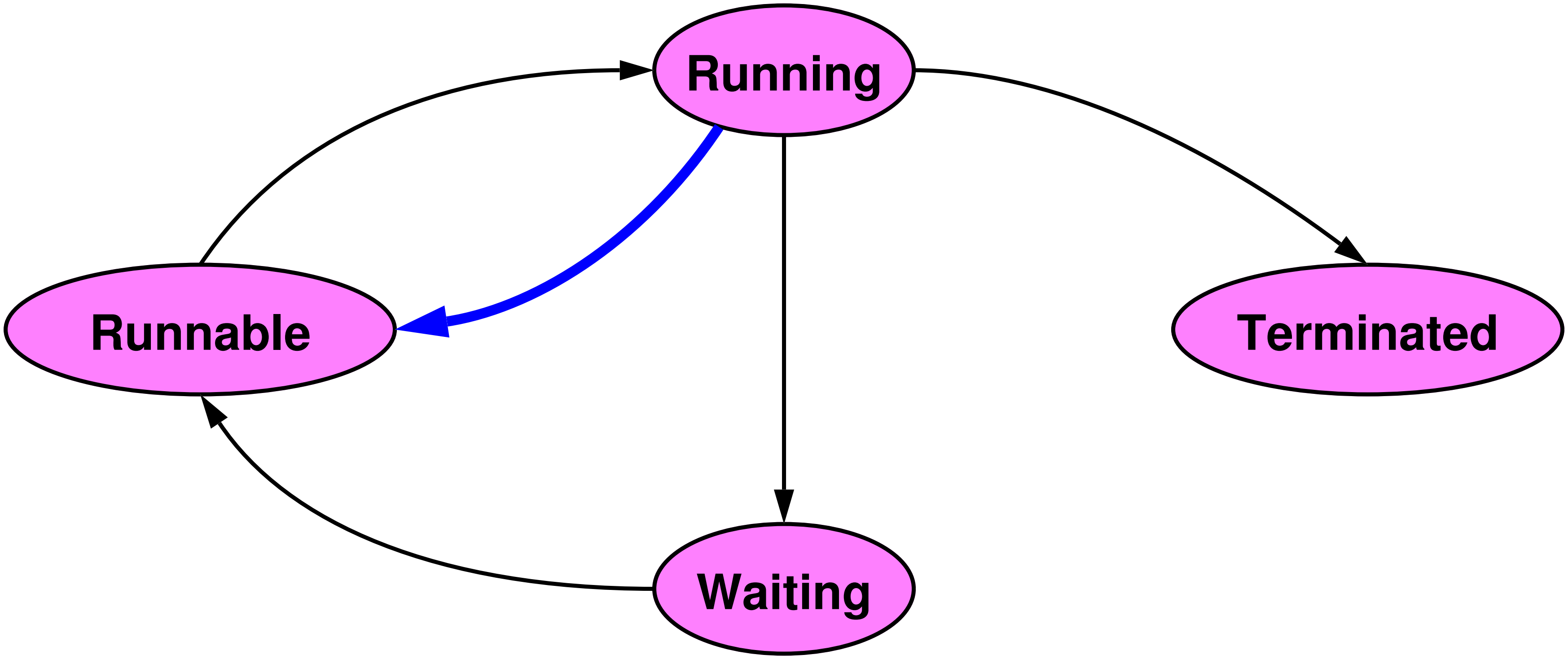
- a thread starts in the *runnable* state
- the *scheduler* switches a thread's state from runnable to running

Thread Life Cycle



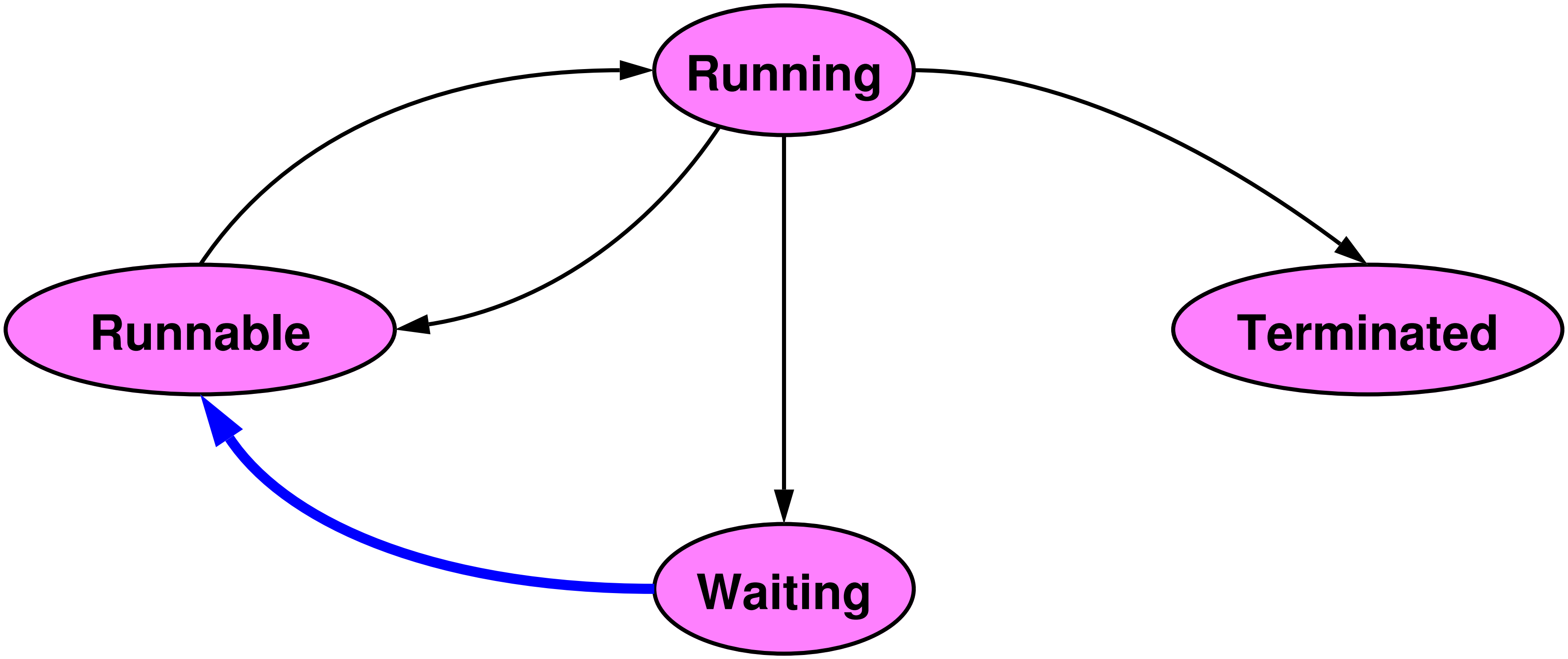
- a thread starts in the *runnable* state
- the *scheduler* switches a thread's state from runnable to running
- a thread goes from running to waiting when a *blocking call* is made

Thread Life Cycle



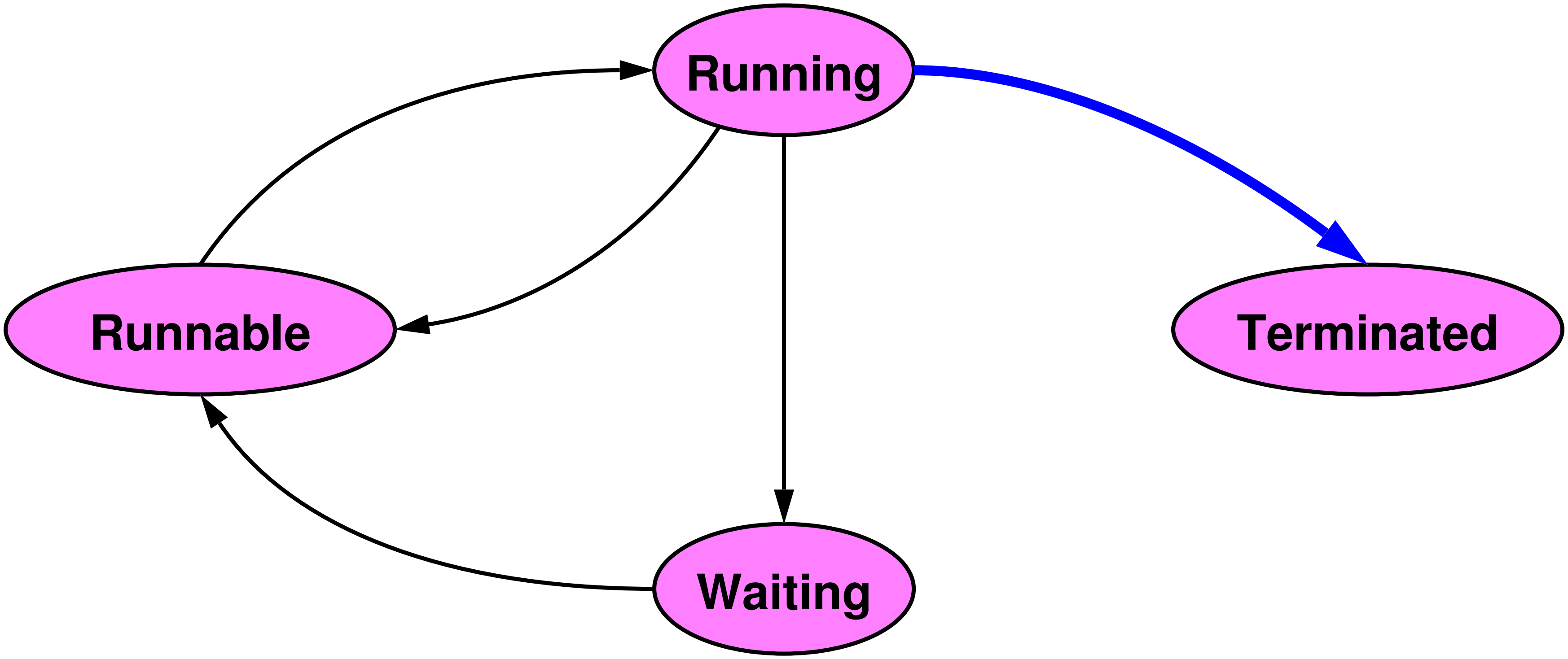
- a thread starts in the *runnable* state
- the *scheduler* switches a thread's state from runnable to running
- a thread goes from running to waiting when a *blocking call* is made
- the *scheduler* switches a thread's state from running to runnable when the thread used up its execution quantum

Thread Life Cycle



- a thread get *unblocked* by the action of another thread or by an interrupt handler

Thread Life Cycle



- a thread get *unblocked* by the action of another thread or by an interrupt handler
- in order for a thread to enter the terminated state, it has to be in the running state just before that
  - what if `pthread_cancel()` is invoked when the thread is not in the running state?

Thread Life Cycle

- Does `pthread_exit()` delete the thread (completely) that calls it?
  - no
- What's left in the thread after it calls `pthread_exit()`?
  - its thread control block
    - needs to keep thread ID and return code around
  - its stack
    - how can a thread *delete its own stack*? no way!

Thread Life Cycle

- Who is deleting the *thread control block* and freeing up the thread's *stack* space?

- If a thread is not detached
  - it can be taken care of in the `pthread_join()` code
    - the thread that calls `pthread_join()` does the clean up

- If a thread is detached (our simple OS does not suppor this)
  - can do this is one of two ways
    1) use a special *reaper thread*
       - basically doing `pthread_join()`
    2) queue these threads on a list and have other threads free them when it's convenient (e.g., when the scheduler schedule a thread to run)

4.1 A Simple System (Monolithic Kernel)

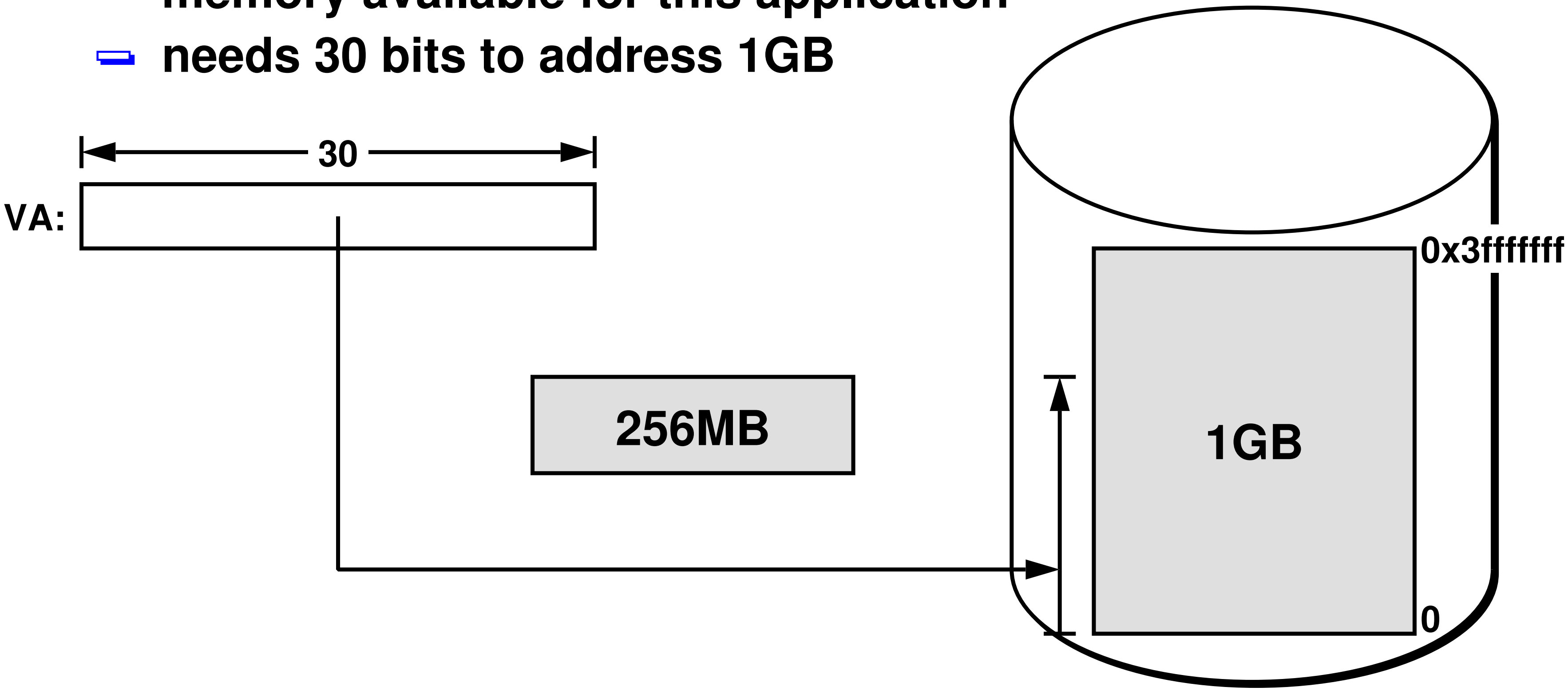
- A Framework for Devices
- Processes & Threads
- *Storage Management*
- Low-level Kernel (will come back to talk about this after Ch 7)

Storage Space

- Where to store data?
  - primary storage, i.e., physical memory
    - directly addressable
  - secondary storage, i.e., disk-based storage
- What would it take to support the idea of virtual memory, i.e., application's "view" of memory?
- An application only works with "virtual memory" (as far as an application is concerned, "virtual memory" is "real memory")
  - e.g., map a 1GB file into memory
    - this memory is *virtual memory*
  - can *allocate* 1GB of *virtual memory* while there's only 256MB of *physical memory*
  - the OS makes sure that real primary storage is available when necessary

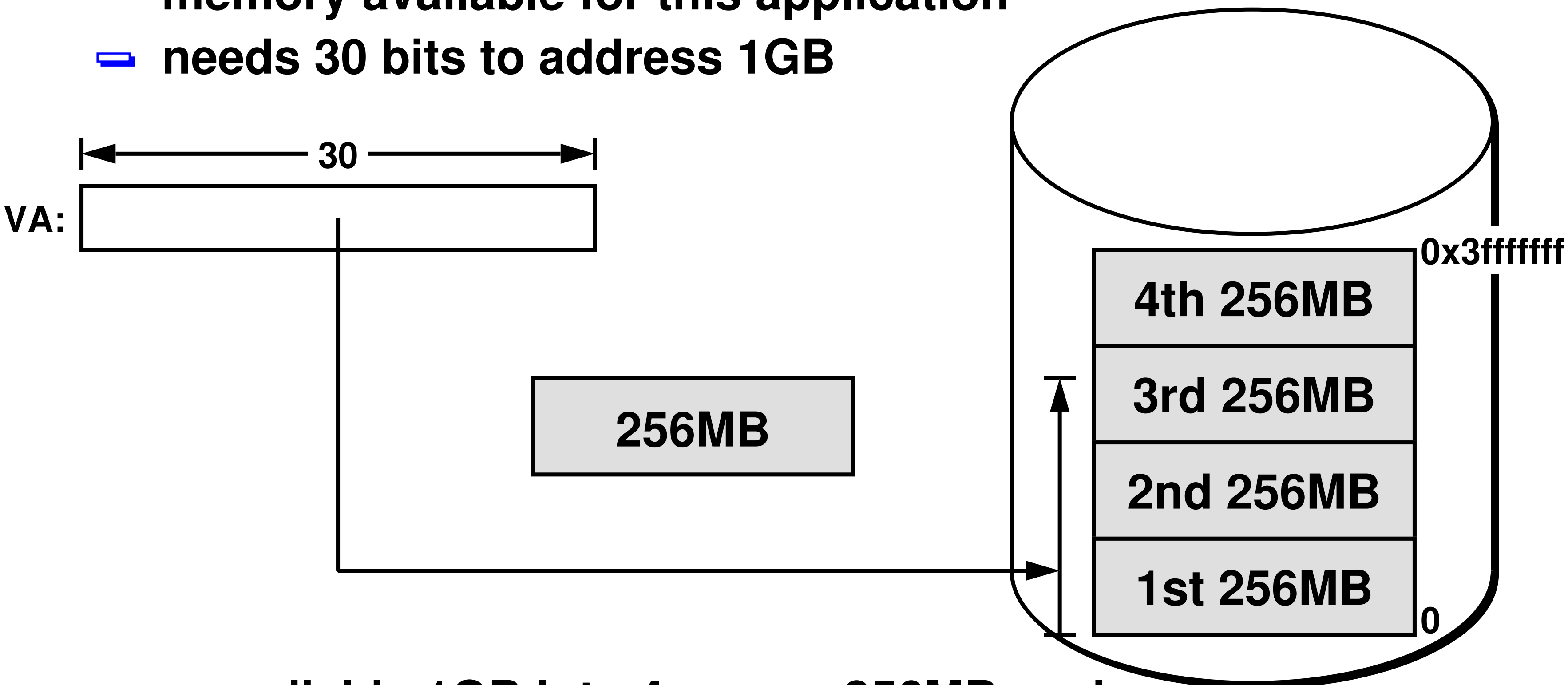
Storage Space

- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



Storage Space

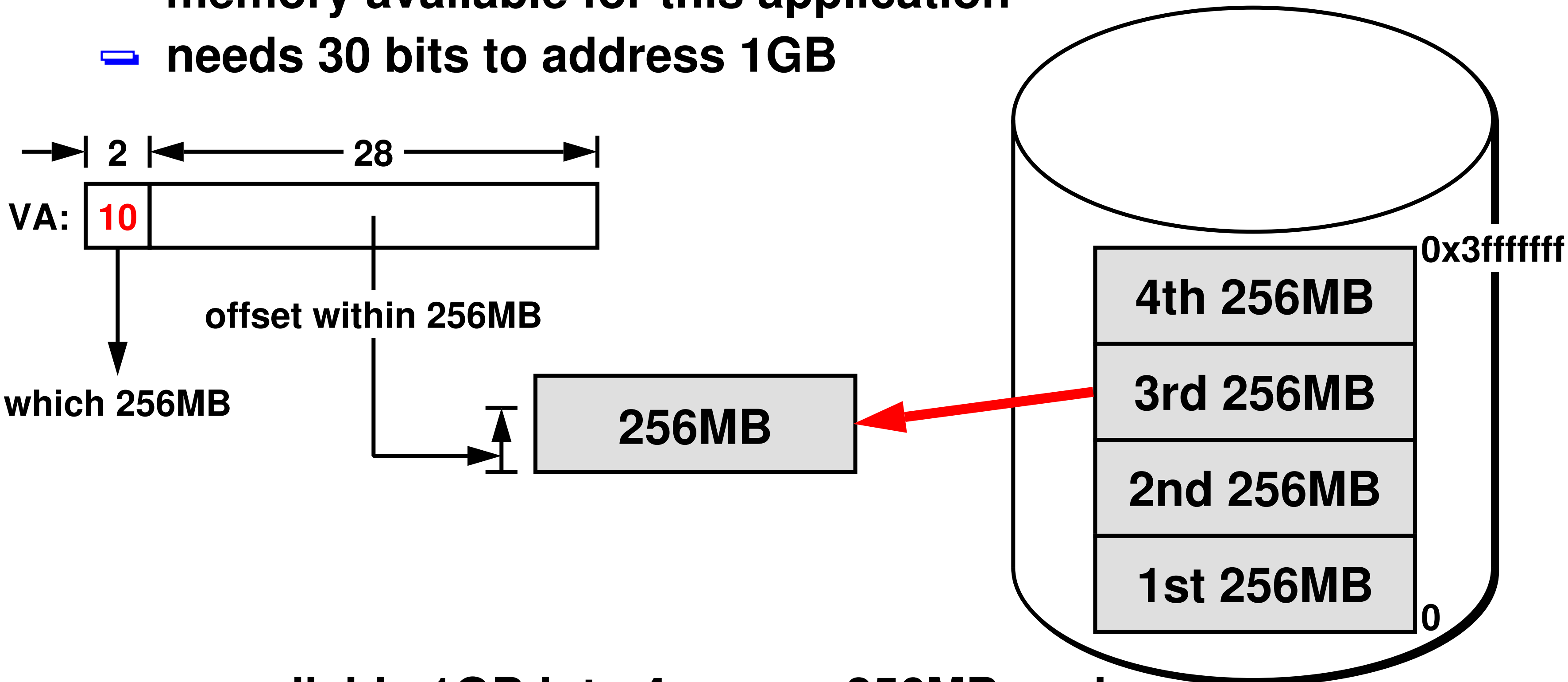
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



  - e.g., divide 1GB into 4 *pages*, 256MB each

Storage Space

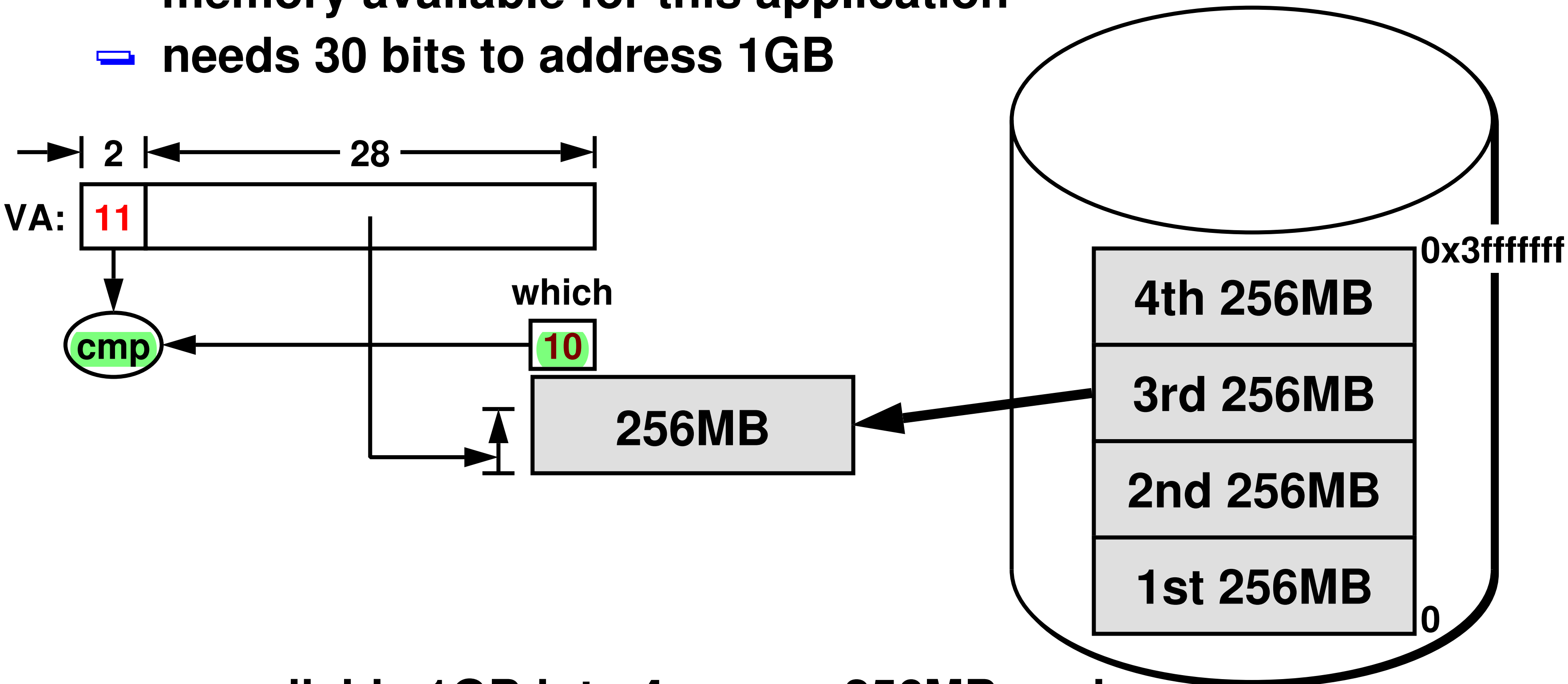
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



  - e.g., divide 1GB into 4 *pages*, 256MB each
  - the *first 2 bits* in the *virtual address* tell you which *page*
  - the rest of the bits give you the *offset* within the *page*

Storage Space

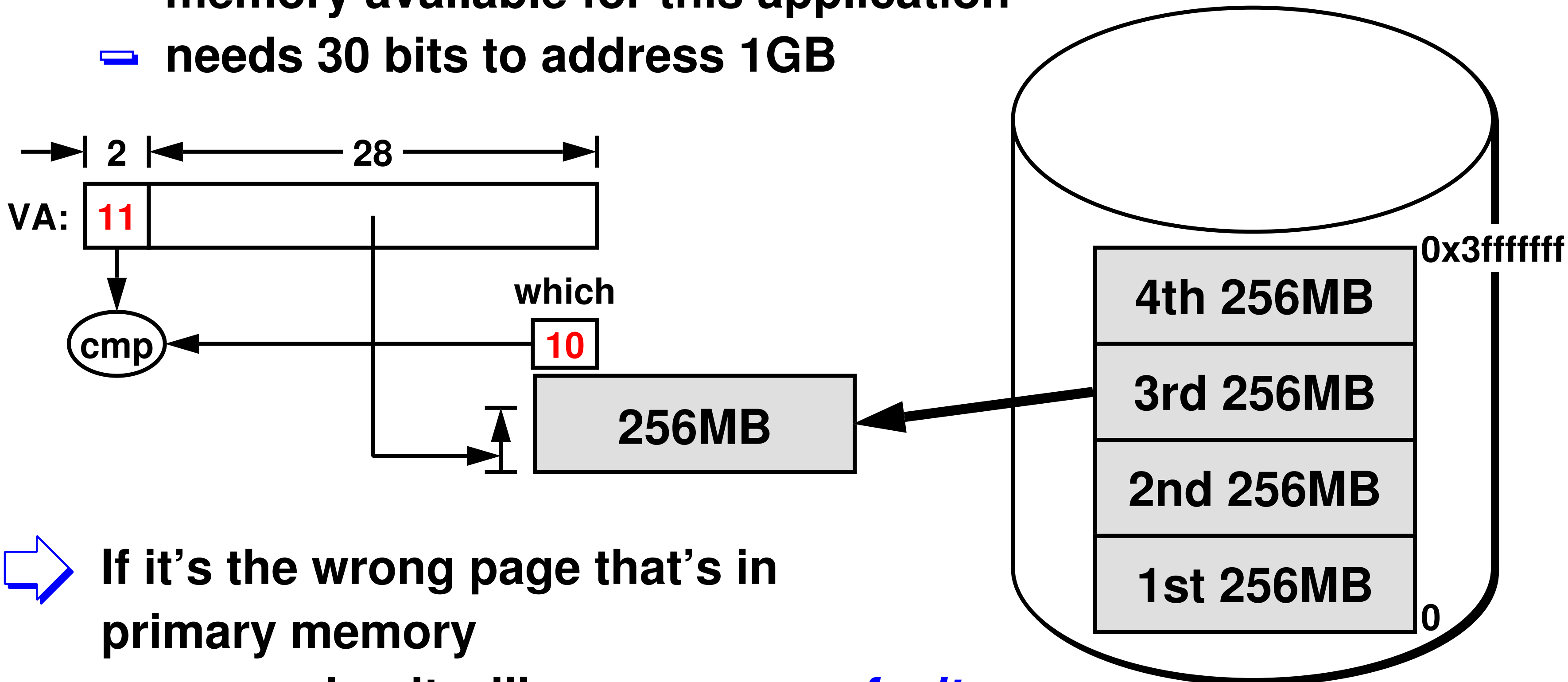
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



  - e.g., divide 1GB into 4 *pages*, 256MB each
  - the *first 2 bits* in the *virtual address* tell you which *page*
  - the rest of the bits give you the *offset* within the *page*
  - check to see if the right page is in *physical* memory

Storage Space

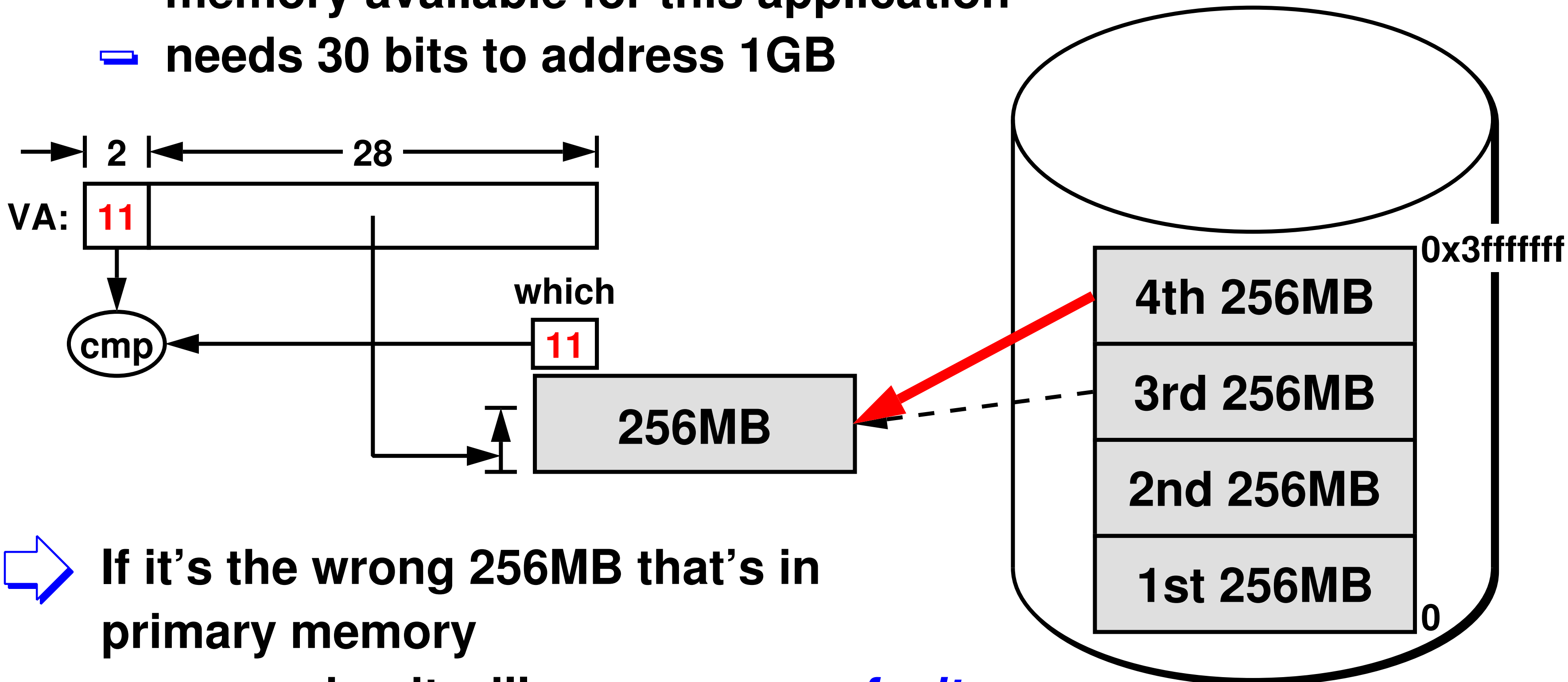
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



- If it's the wrong page that's in primary memory
  - accessing it will cause a *page fault*
  - during a page fault, OS brings the right page into real memory
  - then the thread is allow to proceed with accessing the memory

Storage Space

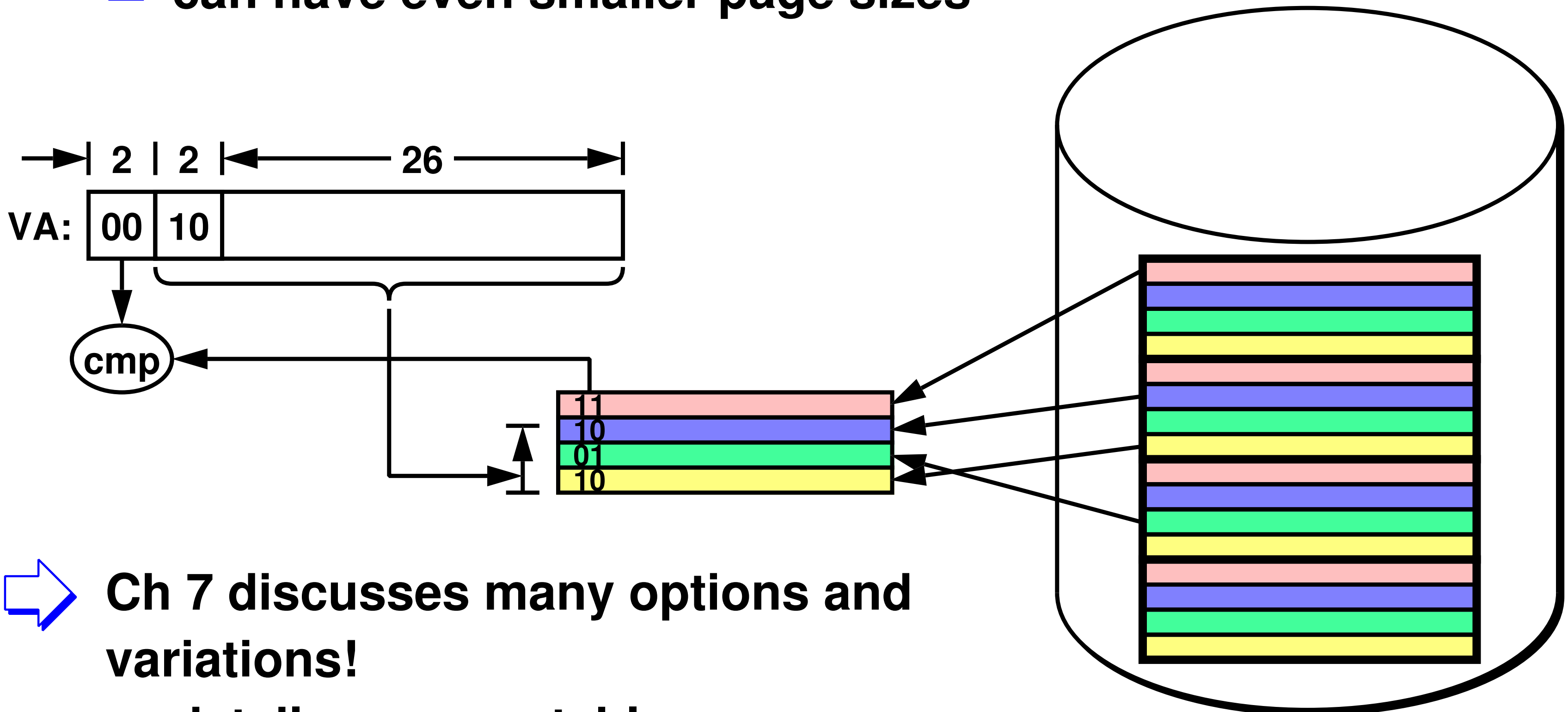
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



- If it's the wrong 256MB that's in primary memory
  - accessing it will cause a *page fault*
  - this "simple" approach has *really poor performance*
    - why just use 2 leading bits? different organizations?

Storage Space

- A more complicated scheme with a smaller page size
  - compare to determine if there is a *hit* or not
  - can have even smaller page sizes



- Ch 7 discusses many options and variations!
  - details on page tables,
    translation look-aside buffers, etc.

Memory Management Concerns

- ***Mapping*** **virtual addresses to real ones**
- **Determining which addresses are *valid*, i.e., refer to allocated memory, and which are not**
- **Keeping track of which real objects, if any, are mapped into each range of virtual addresses**
- **Deciding what should to keep in primary storage (RAM) and what to fetch from elsewhere**

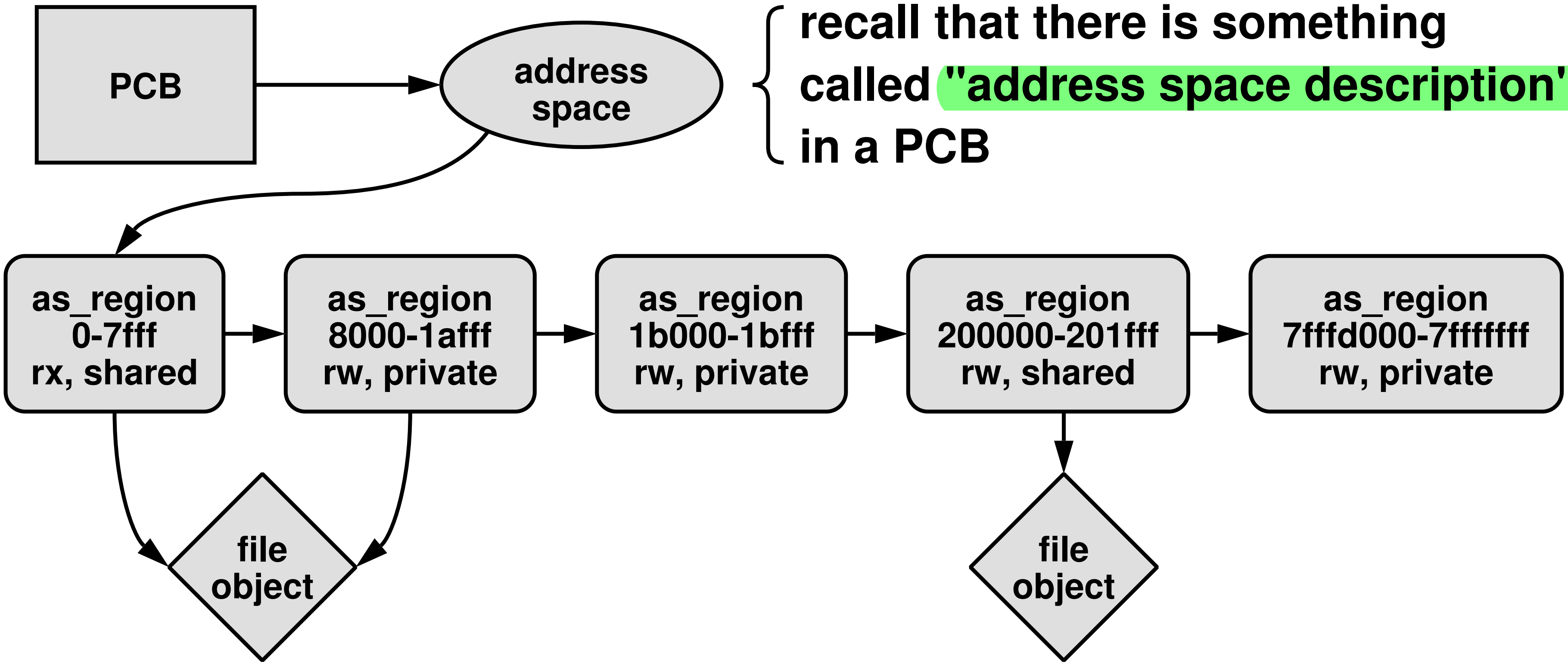
Segmentation Fault

- 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 invalid virtual address will cause a segmentation fault (the OS will deliver SIGSEG to the process)
    - the default action would be to terminate the process

Hardware Memory Map

- 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 some 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
      - often referred as the *memory map*, or *mmap*

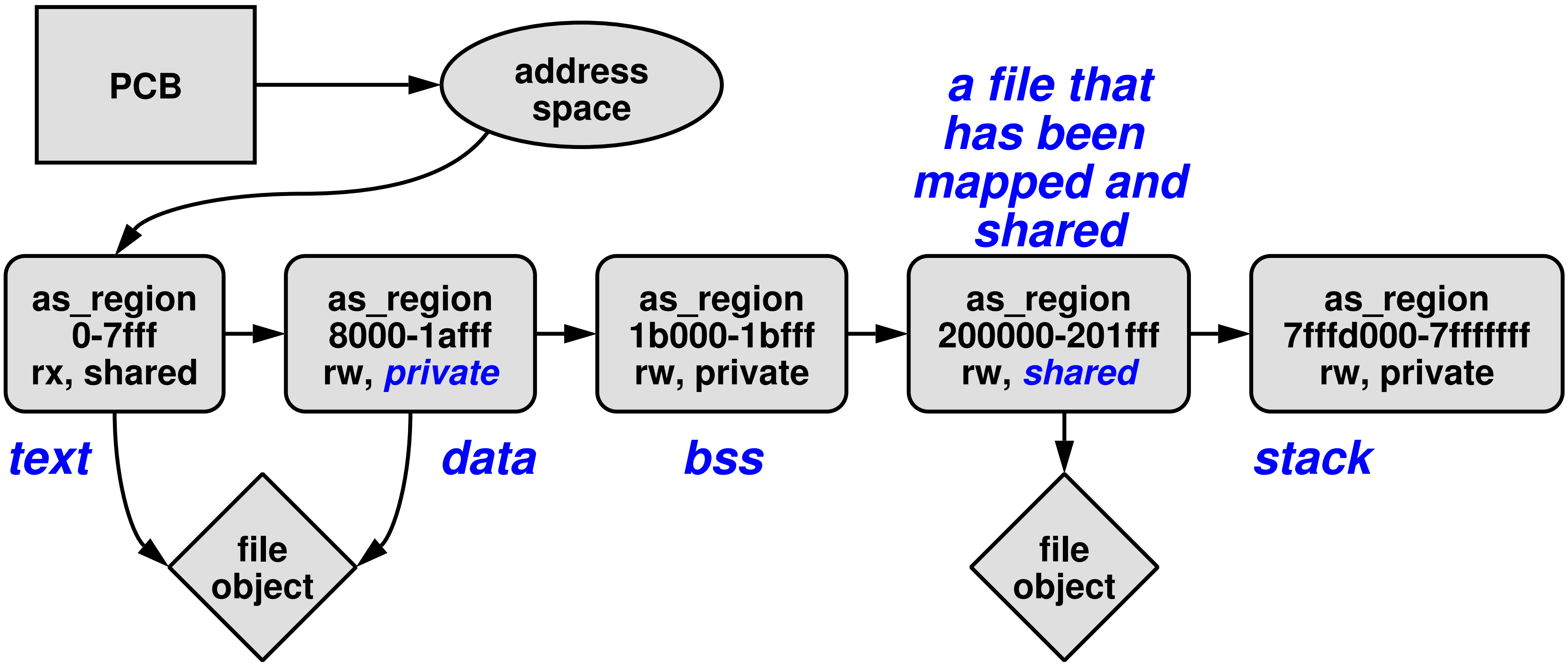
Address Space Representation



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* / *memory maps*

Address Space Representation



- In this example, text and data map portions of the same file
  - *text* is marked read-execute and *shared*
  - data 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 memory 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 memory or if the translation is not in the map
  - a *fault* is occurred and the OS is invoked
    - OS checks the `as_region` 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
      - details in Ch 7
- Two issues need further discussion
  - how is the *primary storage* managed?
  - how are these objects managed in *secondary storage*?

How Is The Primary Storage Managed?

- Who needs primary memory?
  - application processes
  - terminal-handling subsystem
  - communication subsystem
  - I/O subsystem
- They *compete* for available memory
  - it's difficult to be "fair" (what does it even mean?)
- If primary memory is managed poorly
  - one subsystem can use up all the available memory
    - then other subsystem won't get to run
    - this many lead to OS crash when a subsystem runs out of memory
- If there are no mapped files, the solution can be simple
  - equally divide the primary memory among the participants
    - this way, they won't compete

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

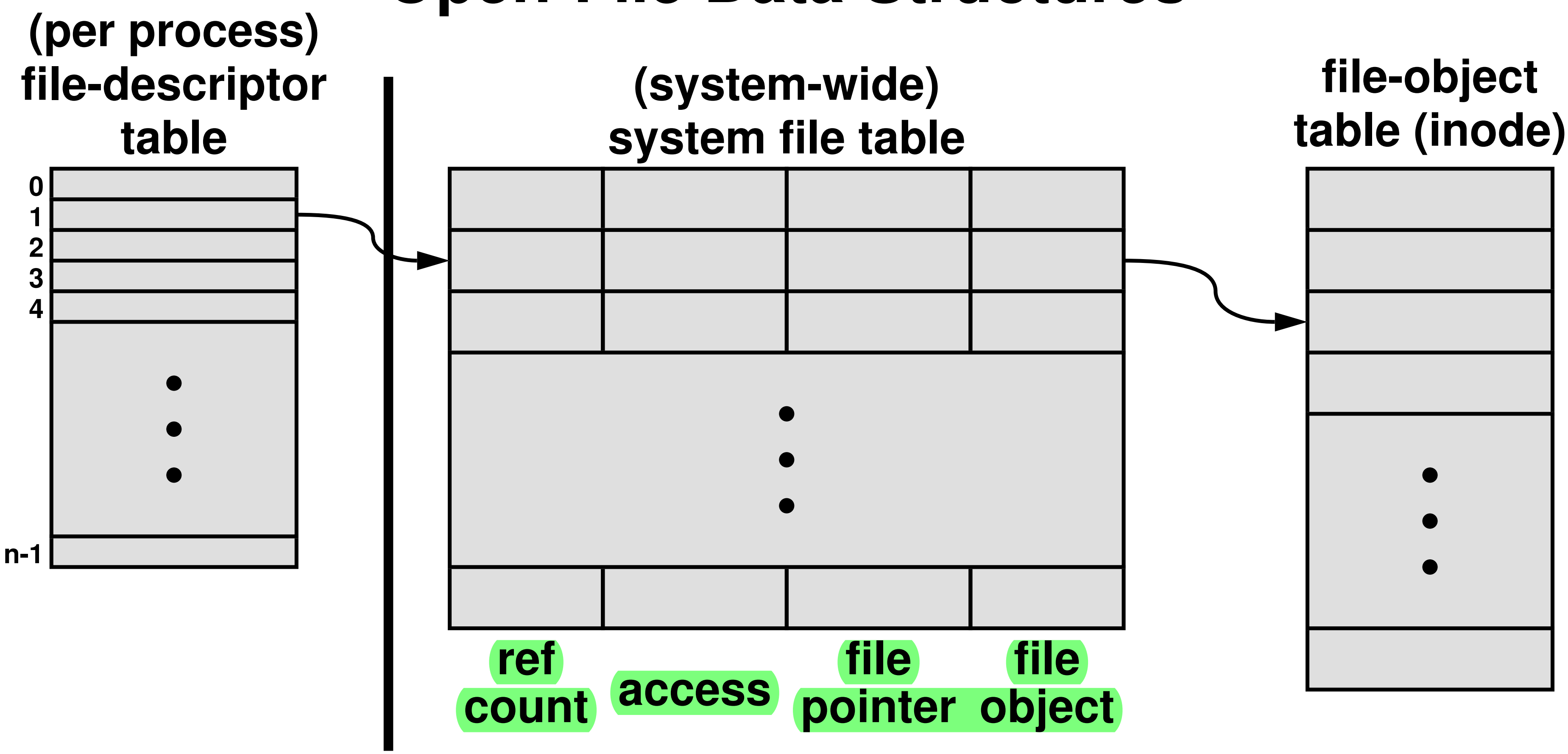
- It's difficult to be *fair*
  - it's difficult to even define what *fair* means
- We will discuss some solutions in Ch 7
  - for now, we use the following solution
    - give each participant (processes, file subsystem, etc.) a minimum amount of storage
    - leave some additional storage available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

How Are Objects Managed In Secondary Storage?

- The *file system* is used to manage objects in secondary storage

- The file system is usually divided into two parts
  - *file system independent*
    - supports the "file abstraction"
    - on Windows, this is called the *"I/O manager"*
    - on Unix, this is called the *"virtual file system (VFS)"*
      - Kernel Assignment 2
  - *file system dependent*
    - on Windows, this is called the "file system"
    - on Unix, this is called the "actual file system"

Open-File Data Structures



- Each process has its own file-descriptor table
  - system file table and file-object table belongs to the kernel

- The *file object* forms the boundary between *VFS* and the actual file system (i.e., will point to device-dependent stuff)

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;
  ...
  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)

File Object in C

```
typedef struct {
  unsigned short refcount;
  struct file_ops *file_op;
  /* function pointers */
} FileObject;
```

- A file object uses an *array of function pointers*
  - this is how C implements *C++ polymorphism*
  - one 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 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

- ***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**

- **A simple** ***hash function*** **is used to locate file blocks in the cache**
  - **keyed by** ***inode number***

Ch 5: Processor Management

Bill Cheng

http://merlot.usc.edu/cs402-f13

Processor Management

- Threads *Implementation*
  - lock/mutex implementation on multiprocessors
- Interrupts
- Scheduling
- Linux/Windows Scheduler

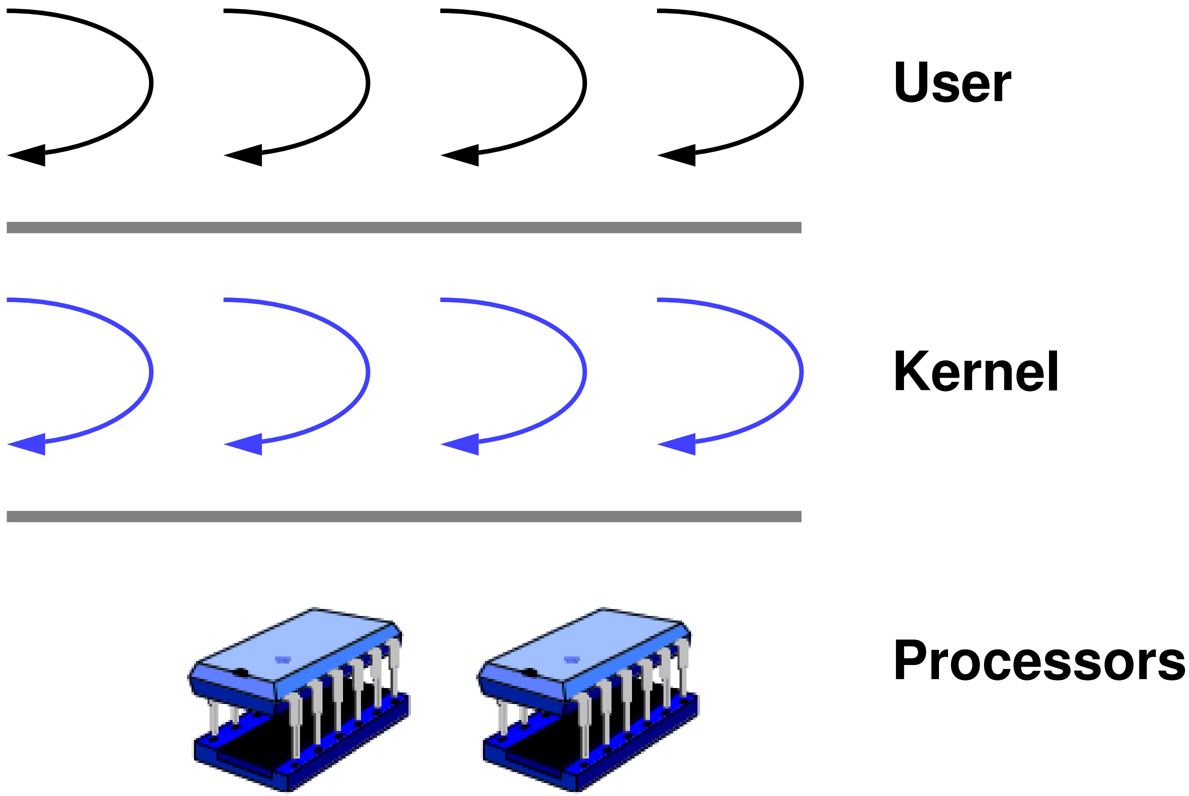
5.1 Threads Implementations

- *Strategies*
- A Simple Thread Implementation
- Multiple Processors

Threads Implementation

- The ultimate goal of the OS is to support user-level applications
  - we will discuss various strategies for supporting threads
- Where are operations on threads implemented?
  - in the kernel?
  - or in user-level library?
- Approaches
  - one-level model (threads are implemented in the kernel)
    - variable-weight processes
  - two-level model (threads are implemented in user library)
    - $N \times 1$
    - $M \times N$
    - scheduler activations model

One-Level Model



One-Level Model

- The simplest and most direct approach is the *one-level model*
  - all aspects of the *thread implementation* are *in the kernel*
    - i.e., all thread routines (e.g., `pthread_mutex_lock`) called by user code are all system calls
  - each *user thread* is mapped one-to-one to a *kernel thread*

- If a thread calls `pthread_create()`
  - it's a system call, so it traps into the kernel
  - the kernel creates a thread control block
    - associate it with the process control block
  - the kernel creates a kernel and a user stack for this thread

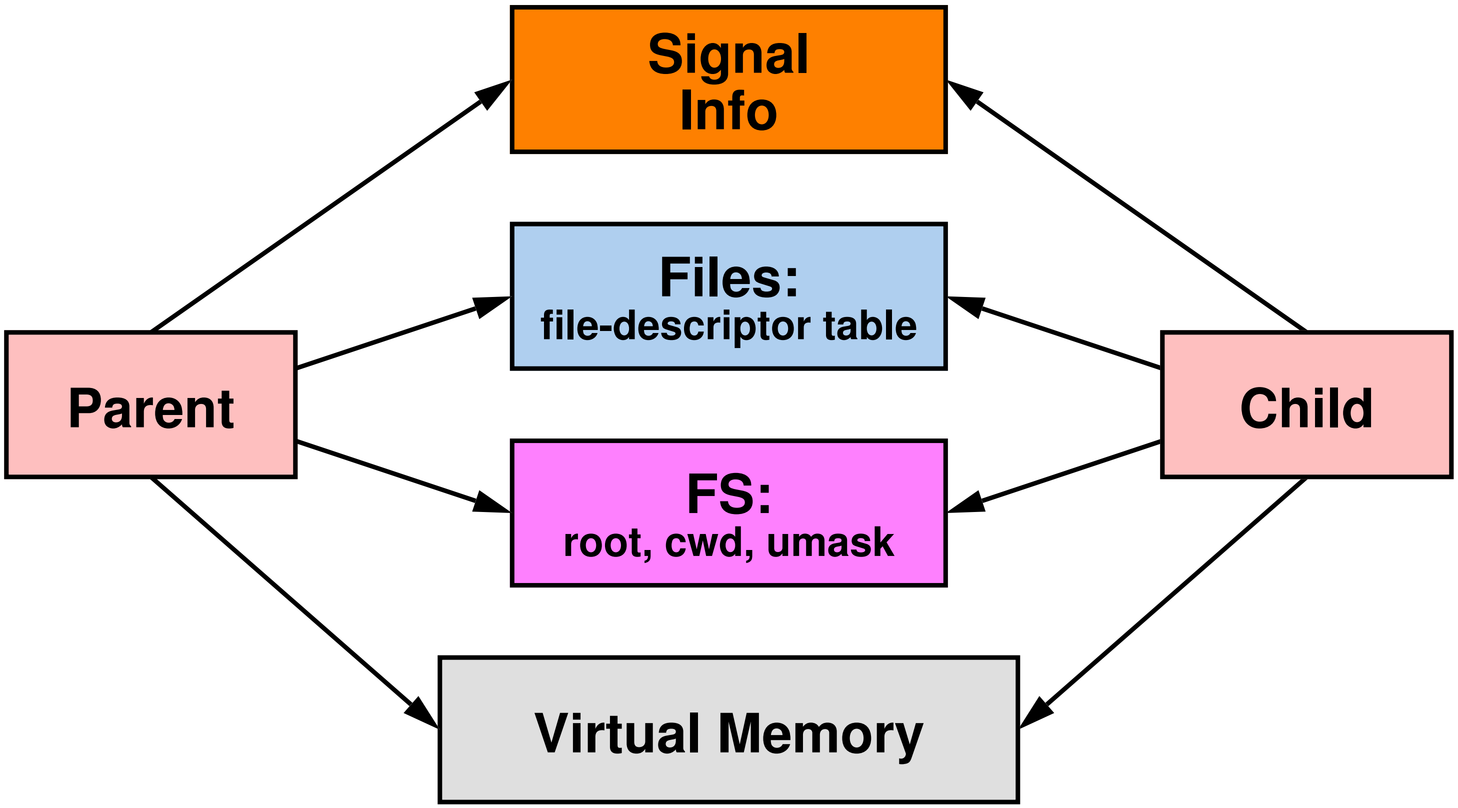
- What about `pthread_mutex_lock()`
  - why does it have to be done in the kernel?
  - it's not necessary to protect the threads from each other!
    - you definitely don't need the kernel to protect threads from each other

One-Level Model

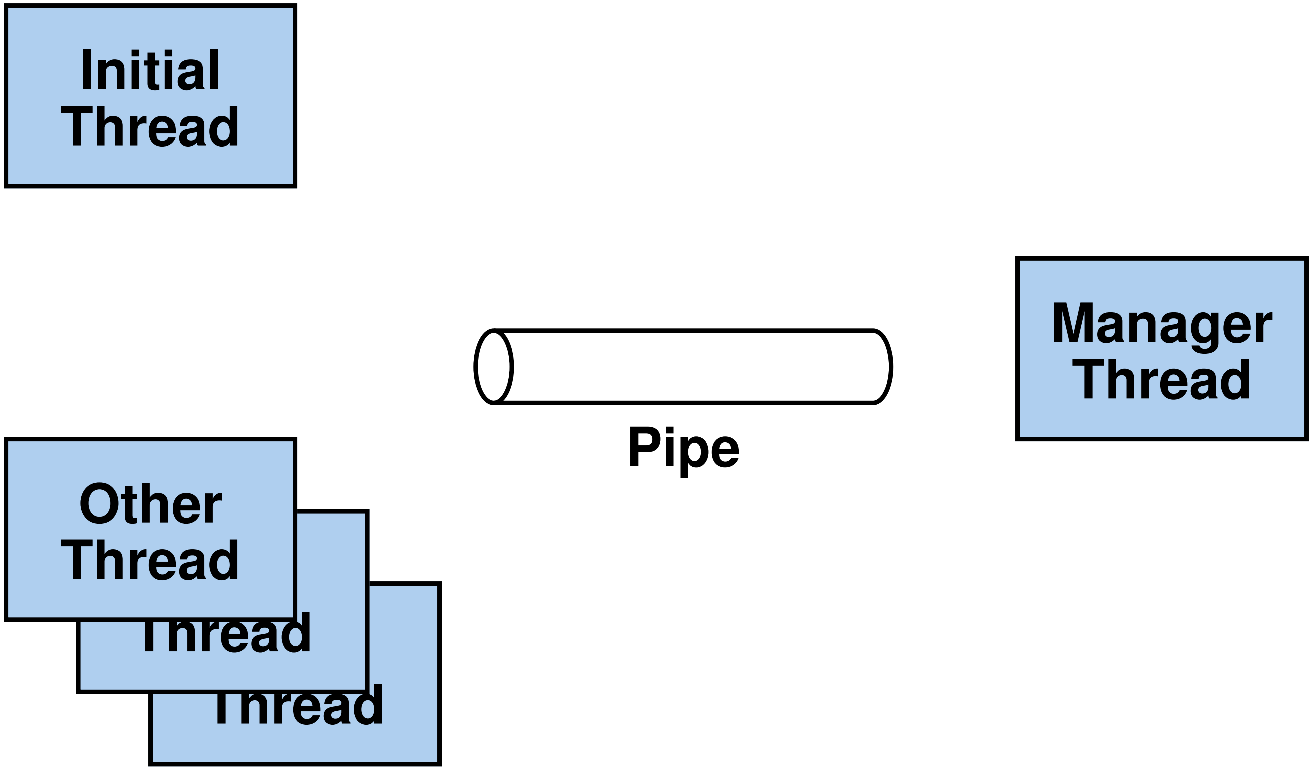
- **Problem: system calls are expensive**
  - **if `pthread_mutex_lock` finds the mutex available, it should return quickly (and lock the mutex)**
    - **if this can be done in user code, it can be 20 times faster (for the case where the mutex is available)**
    - **in Win32 threads, an equivalent of a mutex is represented in a user-level data structure**
      - **if such an object is not locked, it returns quickly**
      - **if such an object is locked, it makes a system call and blocks in the kernel**

Variable-Weight Processes

- Variant of one-level model
- Portions of parent process selectively *copied* into or *shared* with child process
- Children created using `clone()` system call



Linux Threads (pre 2.6)



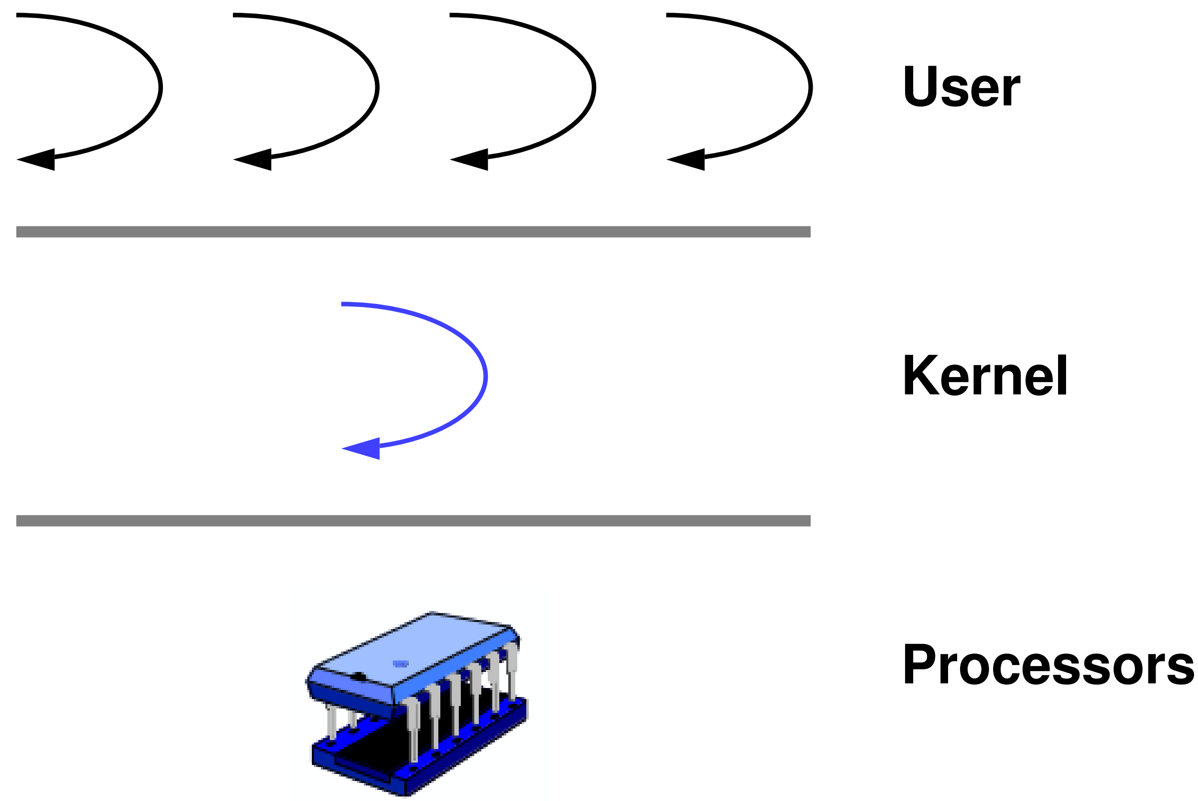
NPTL in Linux 2.6

- **Native POSIX-Threads Library**
  - **full POSIX-threads semantics on improved variable-weight processes**
  - **threads of a "process" form a *thread group***
    - **`getpid()` returns process ID of first thread in group**
    - **any thread in group can wait for any other to terminate**
    - **signals to process delivered by kernel to any thread in group**
  - **new kernel-supported synchronization construct: *futex* (fast mutex)**
    - **used to implement mutexes, semaphores, and condition variables**

Two-Level Model

- In the two-level model, a user-level library plays a major role
  - what an user-level application perceives as a thread is implemented within user-level library code
- Two versions
  - single kernel thread (per user process)
  - multiple kernel threads (per user process)

Two-Level Model - One Kernel Thread



- This is one of the earliest ways of implementing threads
  - threads are implemented entirely in the user level
    - thread control block, mutex in user space
    - thread stack allocated by user library code
  - mostly done on uniprocessors

Two-Level Model - One Kernel Thread

- Within a process, *user threads* are *multiplexed* not on the processor, but on a *kernel-supported thread*
  - the *OS multiplexes kernel threads* (or equivalently, processes) on the *processor*
- User thread creation
  - a stack and a thread control block is allocated
  - thread is put on a queue of runnable threads
    - wait for its turn to become the running thread
- Synchronization implementation
  - relative straightforward
  - e.g., mutex (one queue per mutex)
    - if a thread must block, it simply queues itself on a wait queue and calls context-switch routine to pass control to the first thread on the runnable queue

Two-Level Model - One Kernel Thread

- This is called the N-to-1 model
- Major advantage
  - no system calls (for thread-related APIs)!
- Major disadvantage
  - what if a thread makes a system call (for a non-thread-related API)?
    - it gets blocked in the kernel
    - no other user thread in the process can run

Coping ...

```
ssize_t read(int fd, void *buf, size_t count)
{
  ssize_t ret;
  while (1) {
    if ((ret = real_read(fd, buf, count)) == -1) {
      if (errno == EWOULDBLOCK) {
        sem_wait(&FileSemaphore[fd]);
        continue;
      }
    }
    break;
  }
  return(ret);
}
```

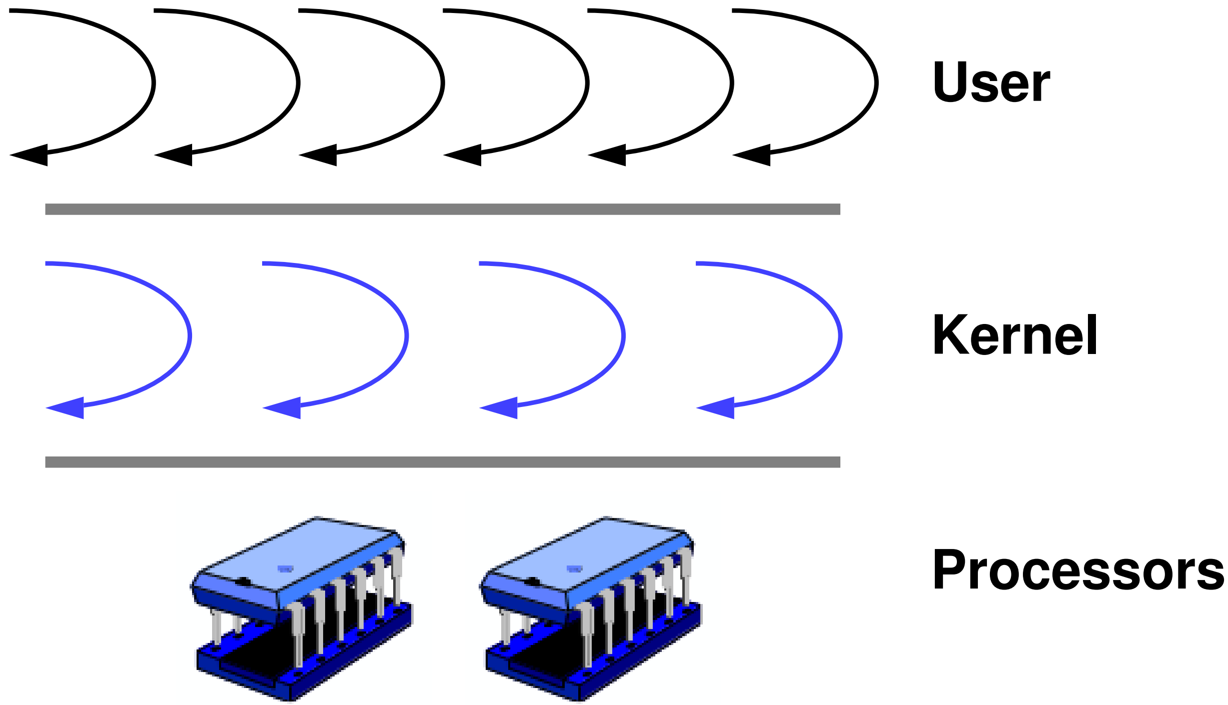
- **Solution is to have a non-blocking `read()` called `real_read()`**
  - `real_read()` either returns immediately with data in `buf`
  - or returns immediately with an error code in `errno`
    - `EWOULDBLOCK` means that a real `read()` would block, i.e., data is not ready to be read

Coping ...

```
ssize_t read(int fd, void *buf, size_t count)
{
  ssize_t ret;
  while (1) {
    if ((ret = real_read(fd, buf, count)) == -1) {
      if (errno == EWOULDBLOCK) {
        sem_wait(&FileSemaphore[fd]);
        continue;
      }
    }
    break;
  }
  return(ret);
}
```

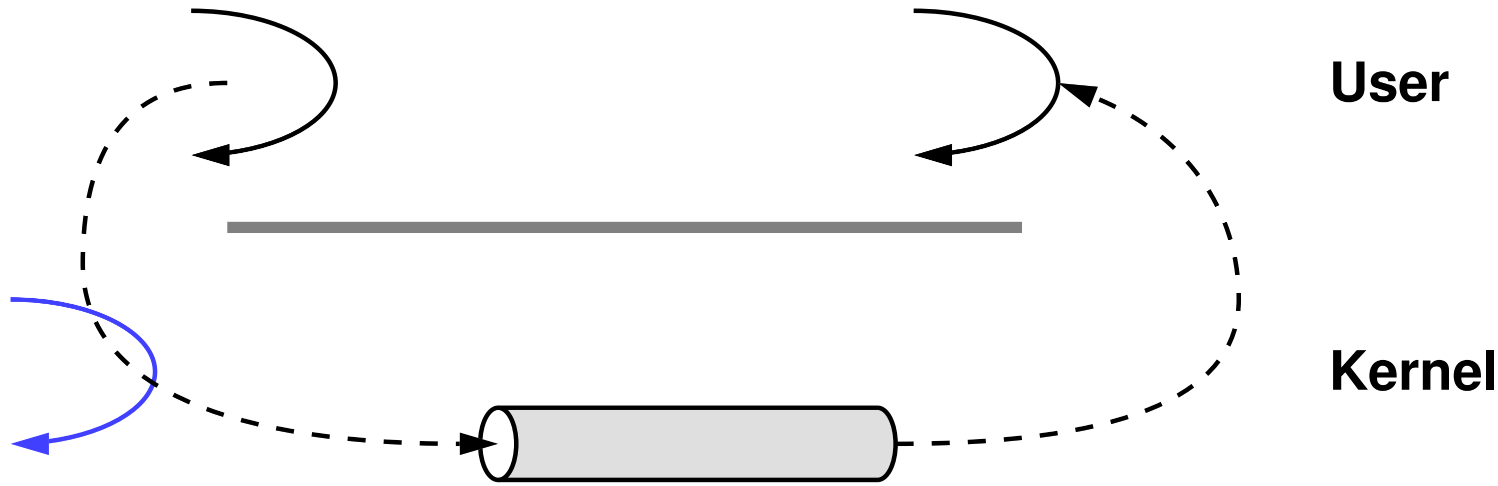
- One semaphore for each open file
  - perhaps a signal handler will invoke `sem_post()` to when data is ready to be read
- Major drawback
  - only works for some I/O objects - not a general solution

Two-Level Model: Multiple Kernel Threads



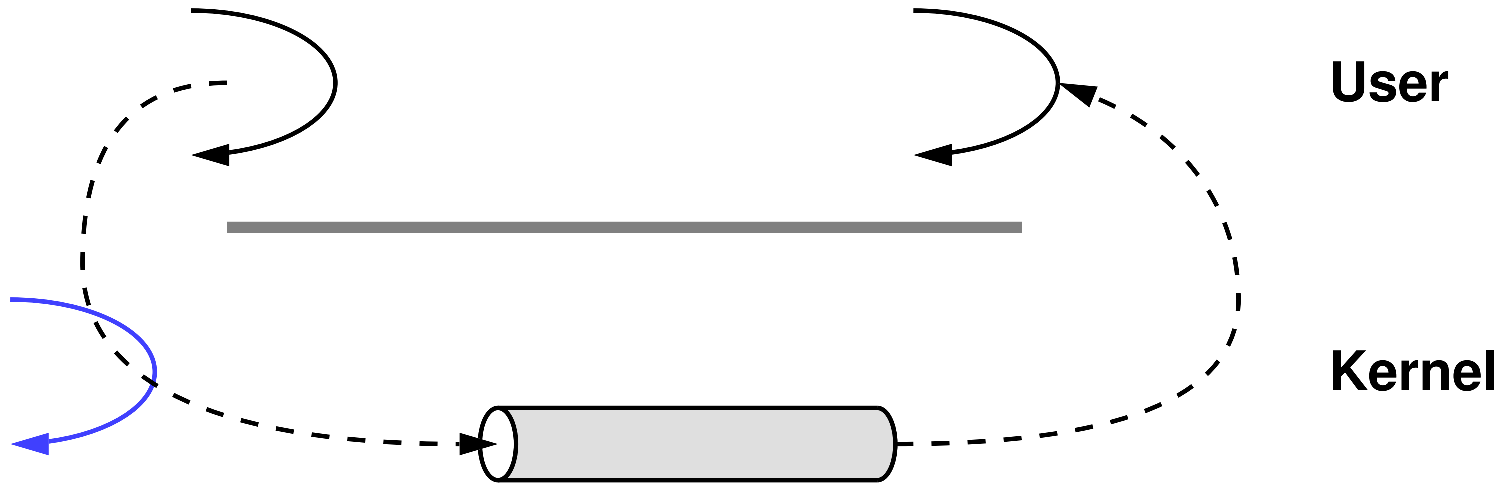
- This is called the M-to-N model
- Implementation is similar to the two-level model with a single kernel thread
  - no system calls (for thread-related APIs)
  - if we don't have enough kernel threads per user process, we end up having the same problem with the N-to-1 model

Deadlock



- Ex: two threads are communicating using a pipe (this is essentially a kernel implementation of the producer-consmer problem)
  - first user thread writes to a full pipe and get blocked in the kernel
    - first thread just happened to use the last kernel thread
    - 2nd thread wants to read the pipe to unblock the first thread, but cannot because no kernel thread left

Deadlock



- **Solaris solution: automatically create a new kernel thread**
  - **an obvious solution**

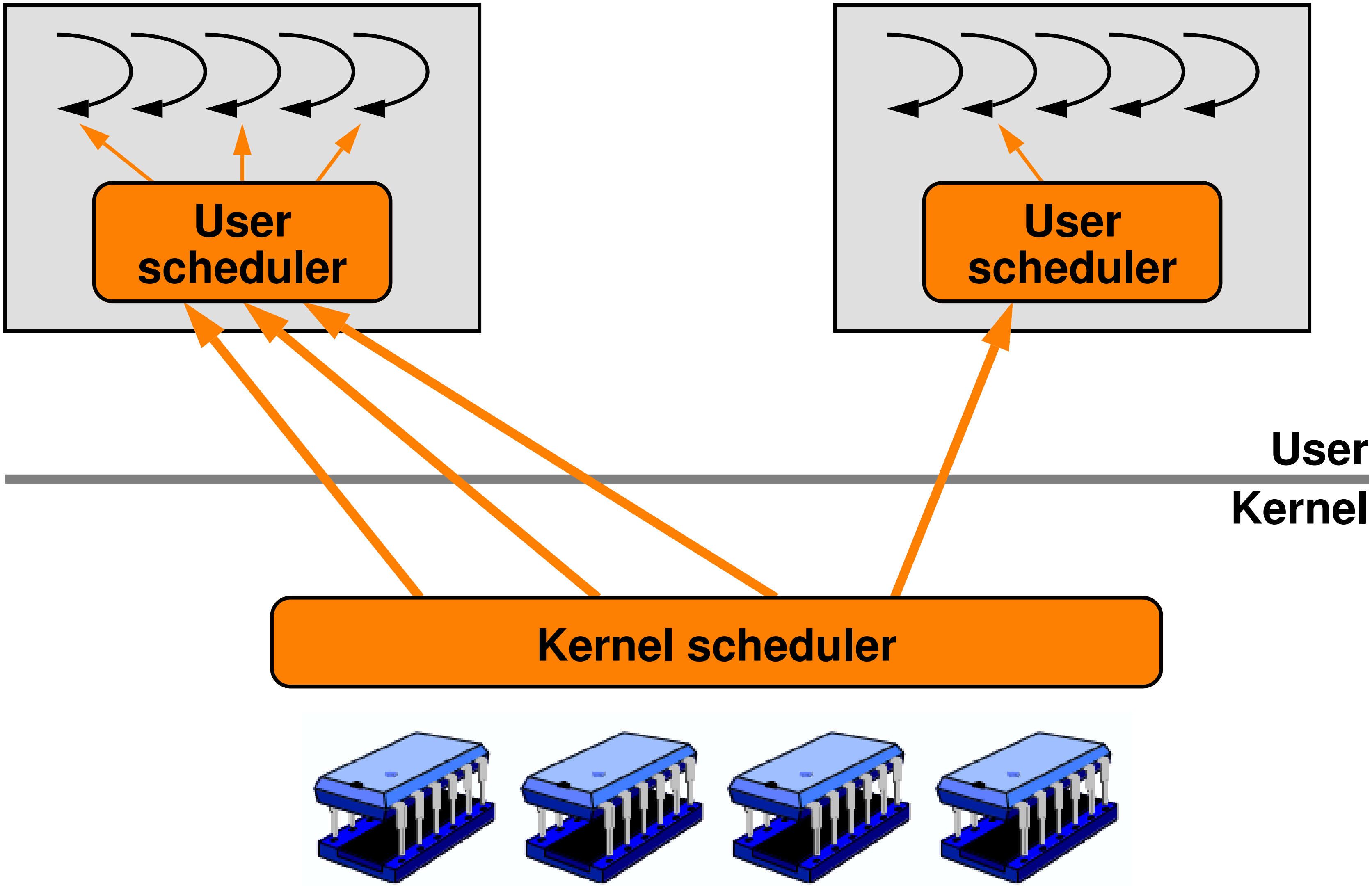
Recap - Problems

- 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
- 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

- 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

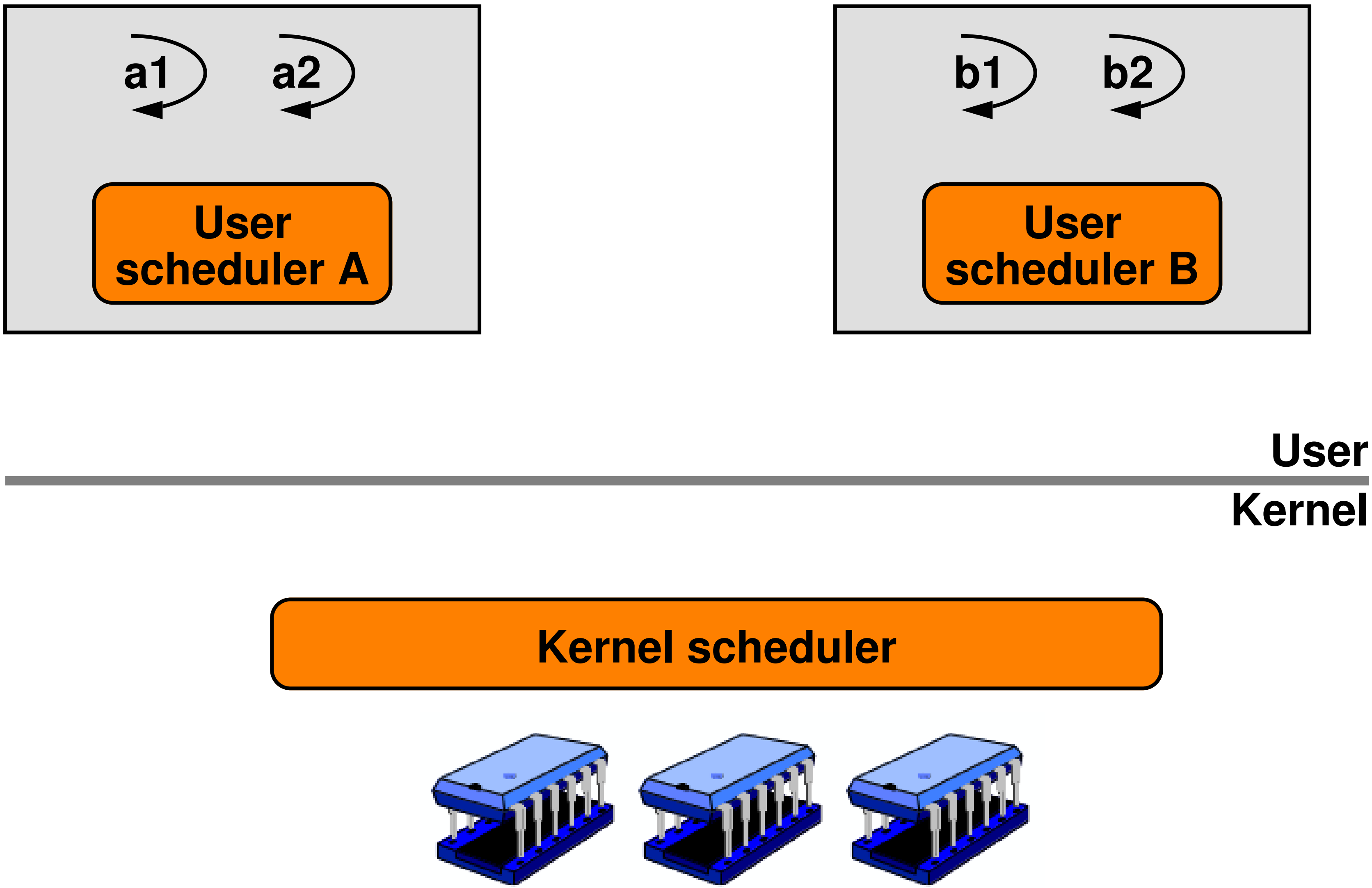


Scheduler Activations Model Example

- 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

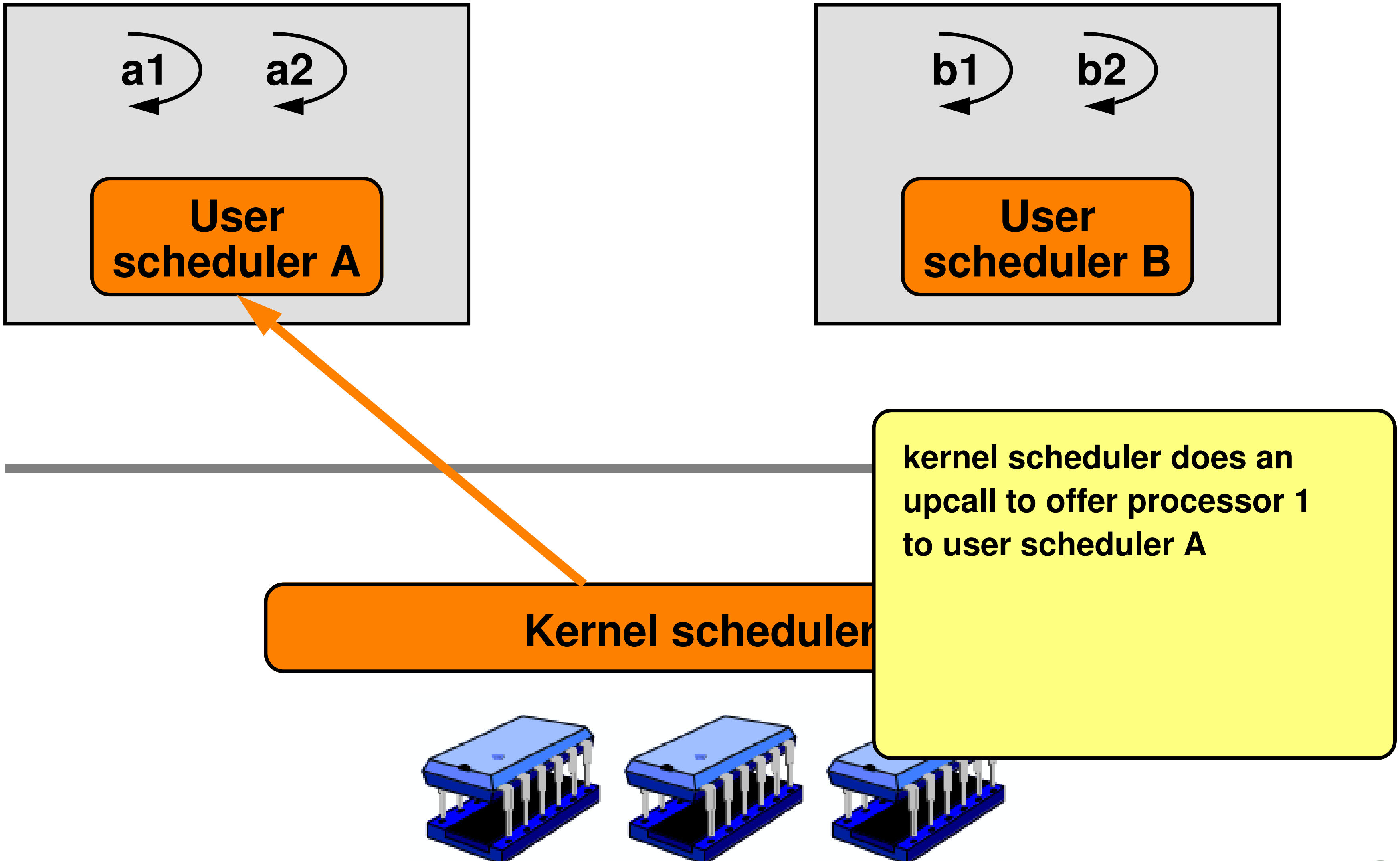
- Suppose the user program creates a new thread and parallelism is desired
  - code in user-level library notifies the kernel that it needs two processors
  - when a processor becomes available, the kernel creates a new kernel context
    - the kernel places an upcall to the user-level library, effectively giving it the processor
    - the user-level library code assigns this processor to the new thread

Scheduler Activations Model Example



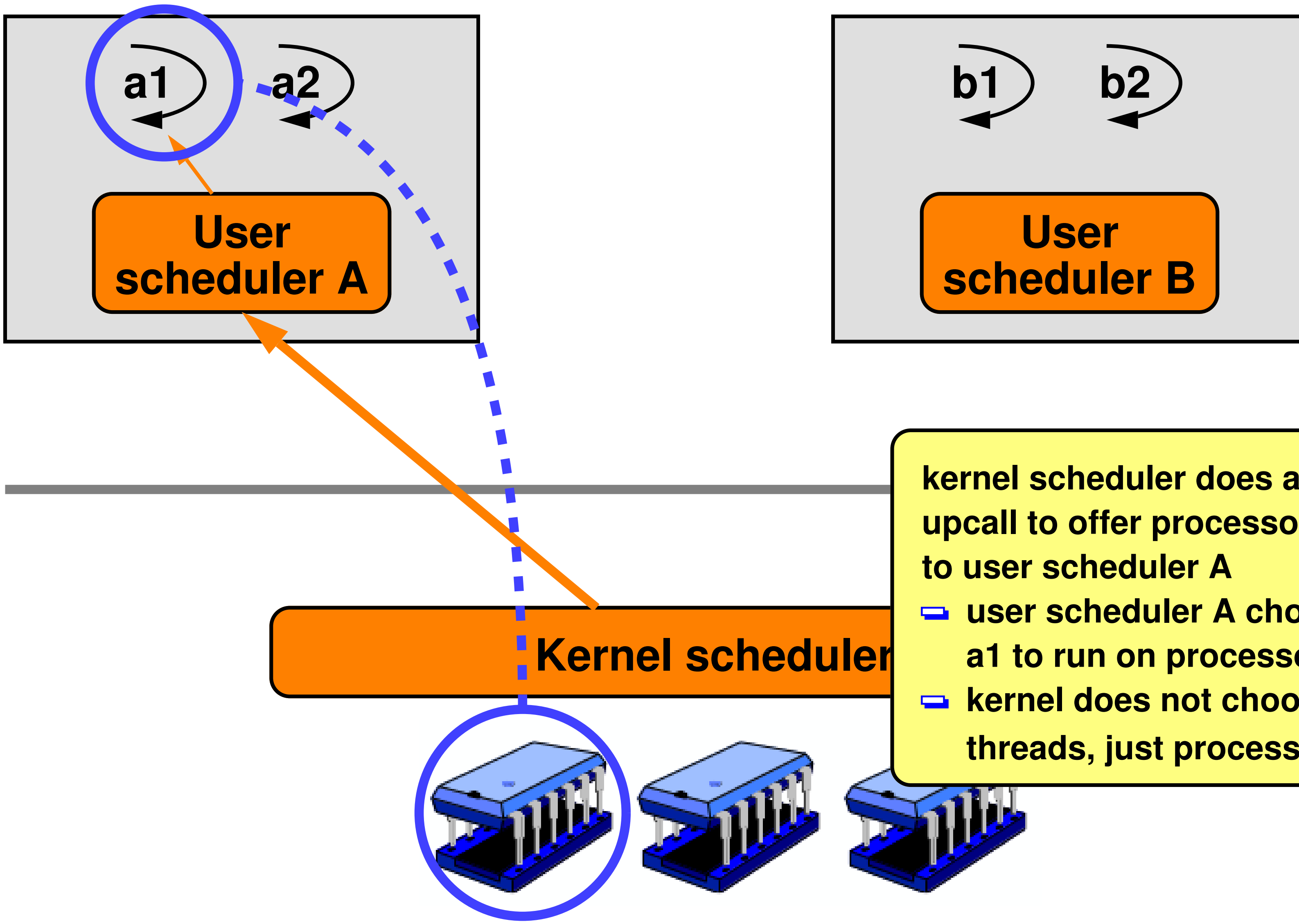
- Kernel scheduler does not schedule threads

Scheduler Activations Model Example



Kernel scheduler does not schedule threads

Scheduler Activations Model Example

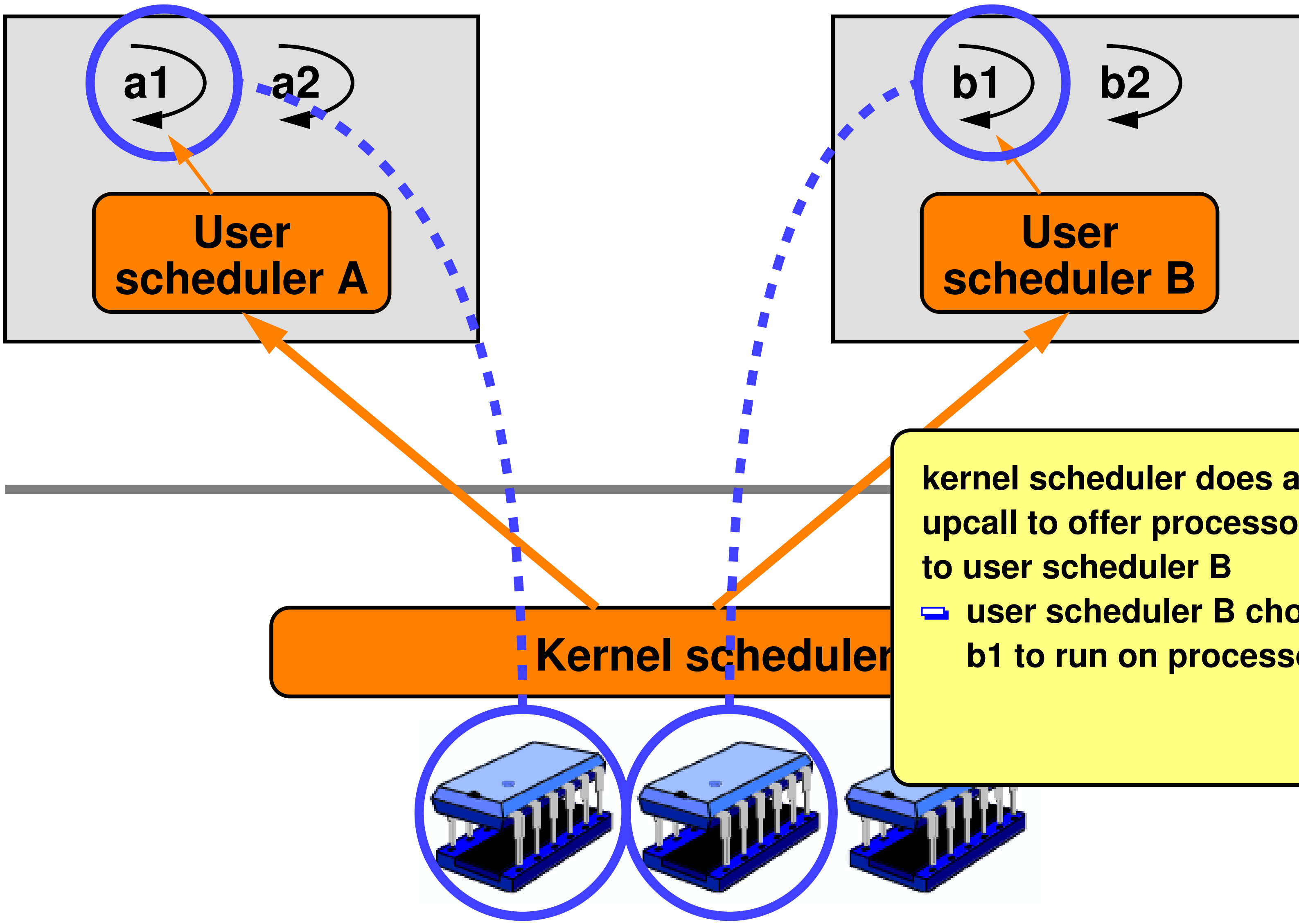


kernel scheduler does an upcall to offer processor 1 to user scheduler A

- 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

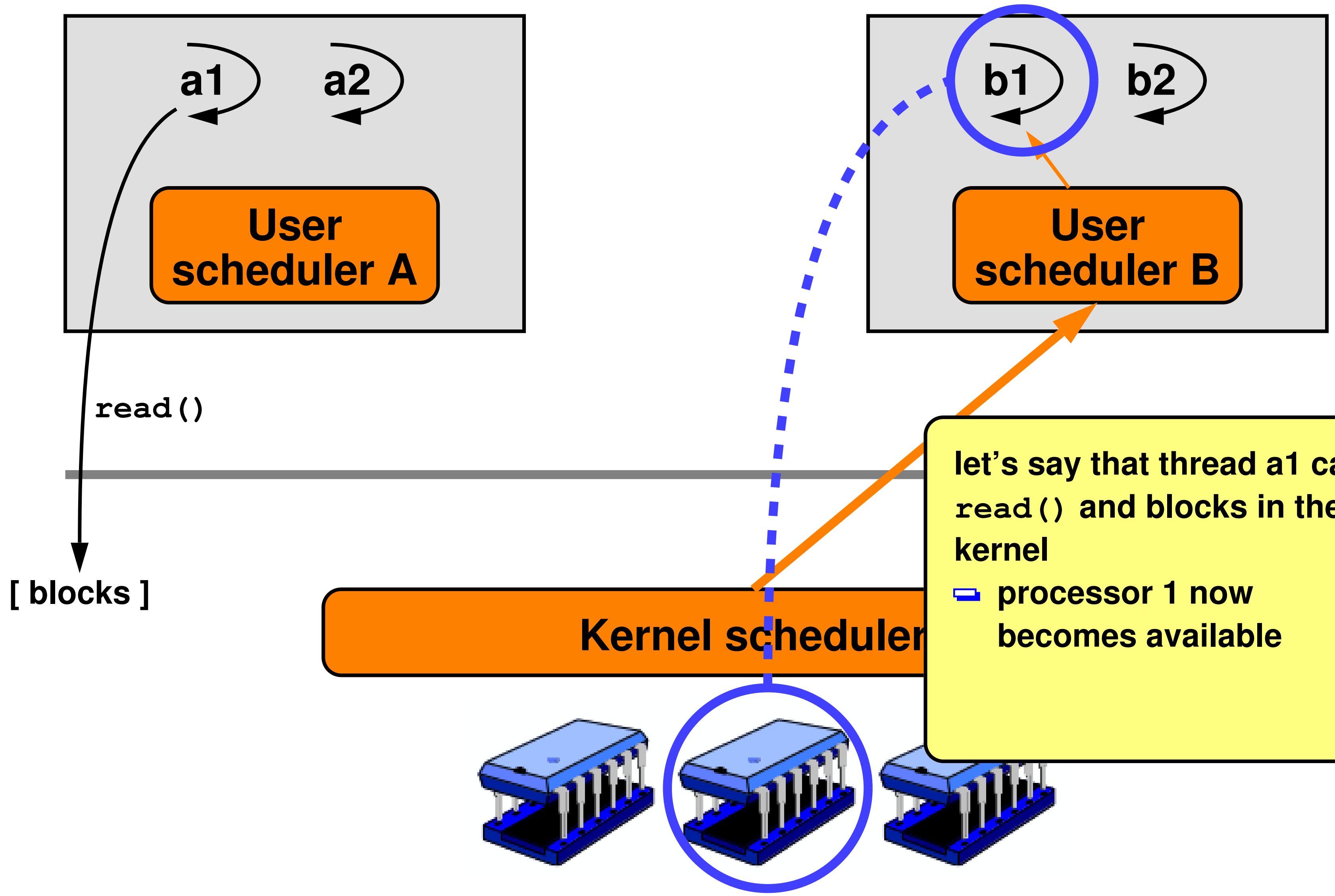


kernel scheduler does an upcall to offer processor 2 to user scheduler B

- user scheduler B chooses b1 to run on processor 2

Kernel scheduler does not schedule threads

Scheduler Activations Model Example

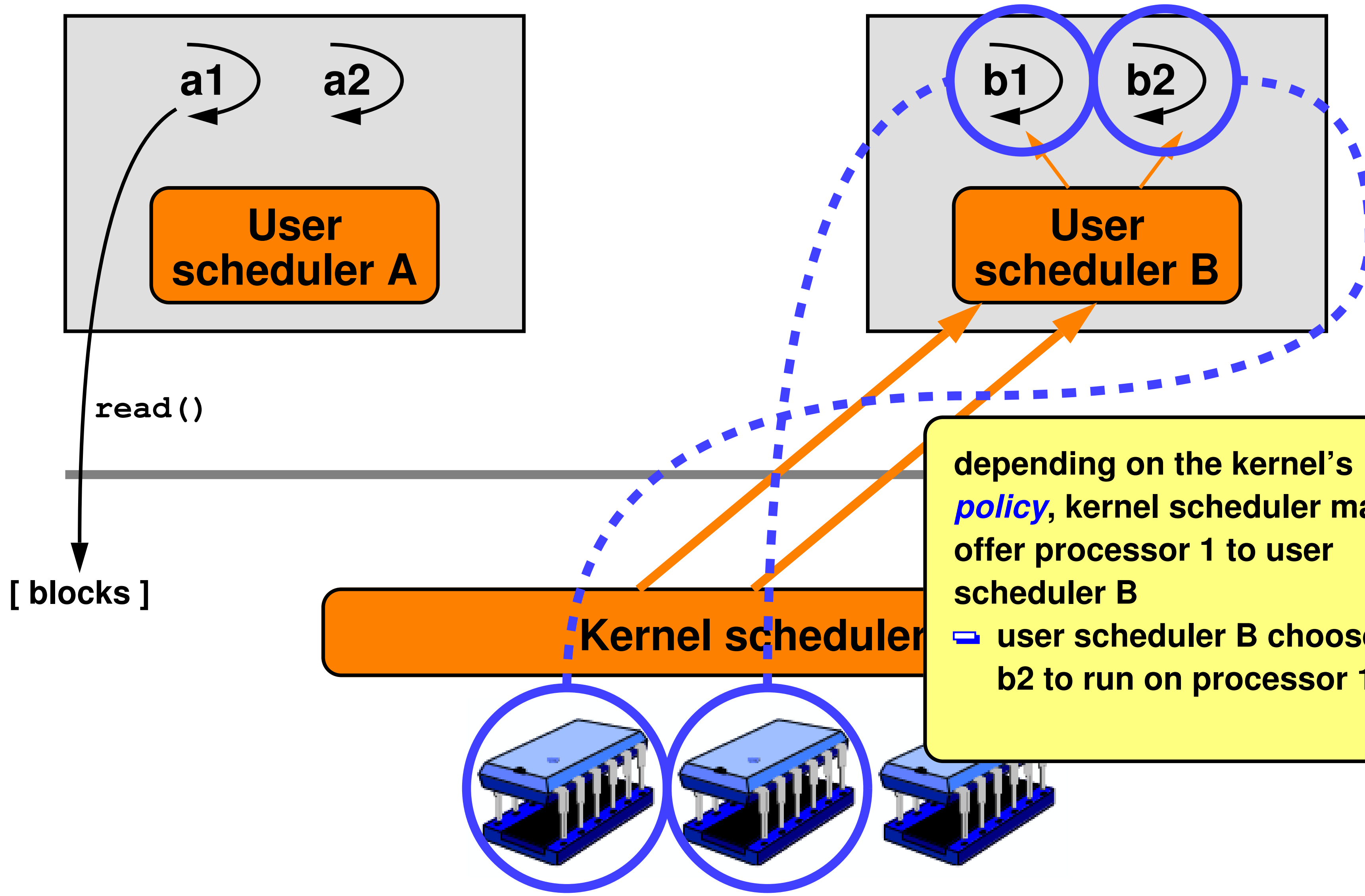


let's say that thread a1 calls `read()` and blocks in the kernel

- processor 1 now becomes available

➡ Kernel scheduler can have various scheduling policies

Scheduler Activations Model Example

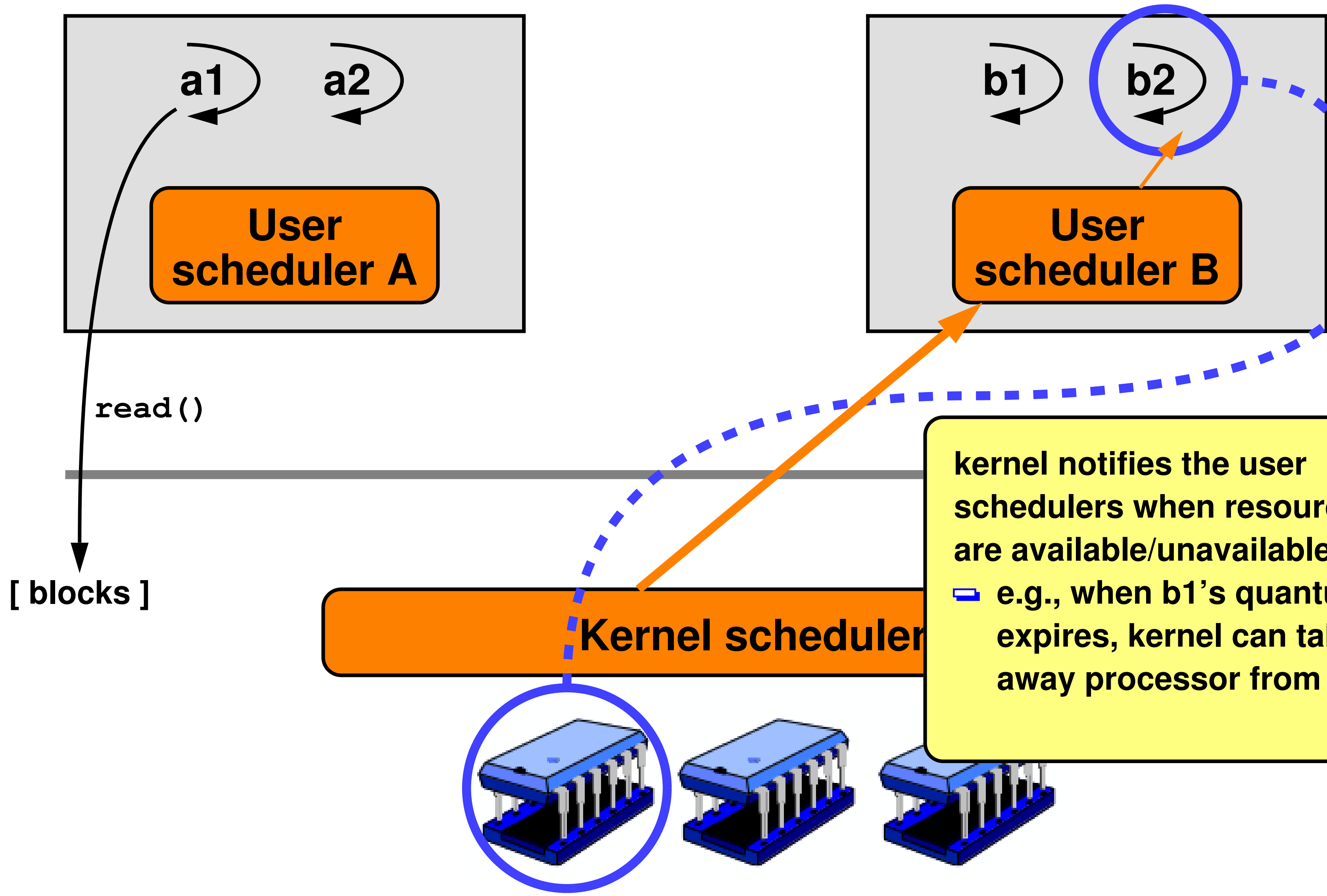


depending on the kernel's *policy*, kernel scheduler may offer processor 1 to user scheduler B

- user scheduler B chooses b2 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**