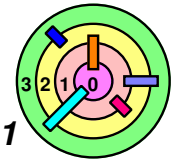
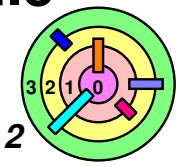
4.1 A Simple System (Monolithic Kernel)

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

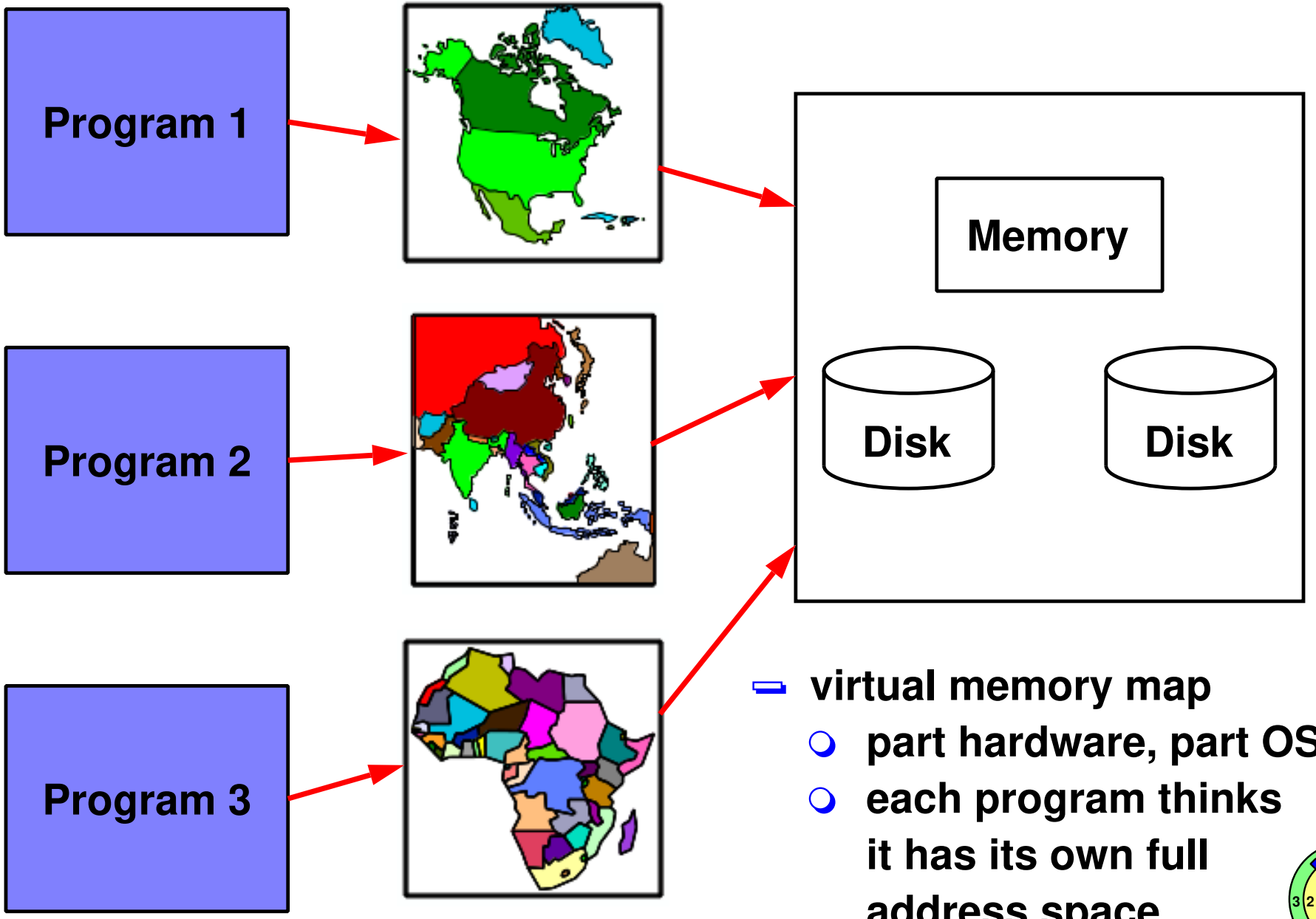


Storage Management

- 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

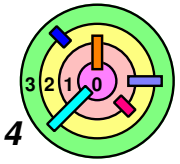


- virtual memory map
  - part hardware, part OS
  - each program thinks it has its own full address space

Storage Management

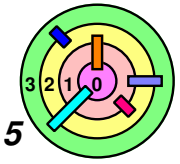
- Two ways for an application to access secondary storage
  - using *sequential I/O:* `open()`, `read()`, `write()`, `close()`
  - using *block I/O:* `open()`, `mmap()`, `close()`

- 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



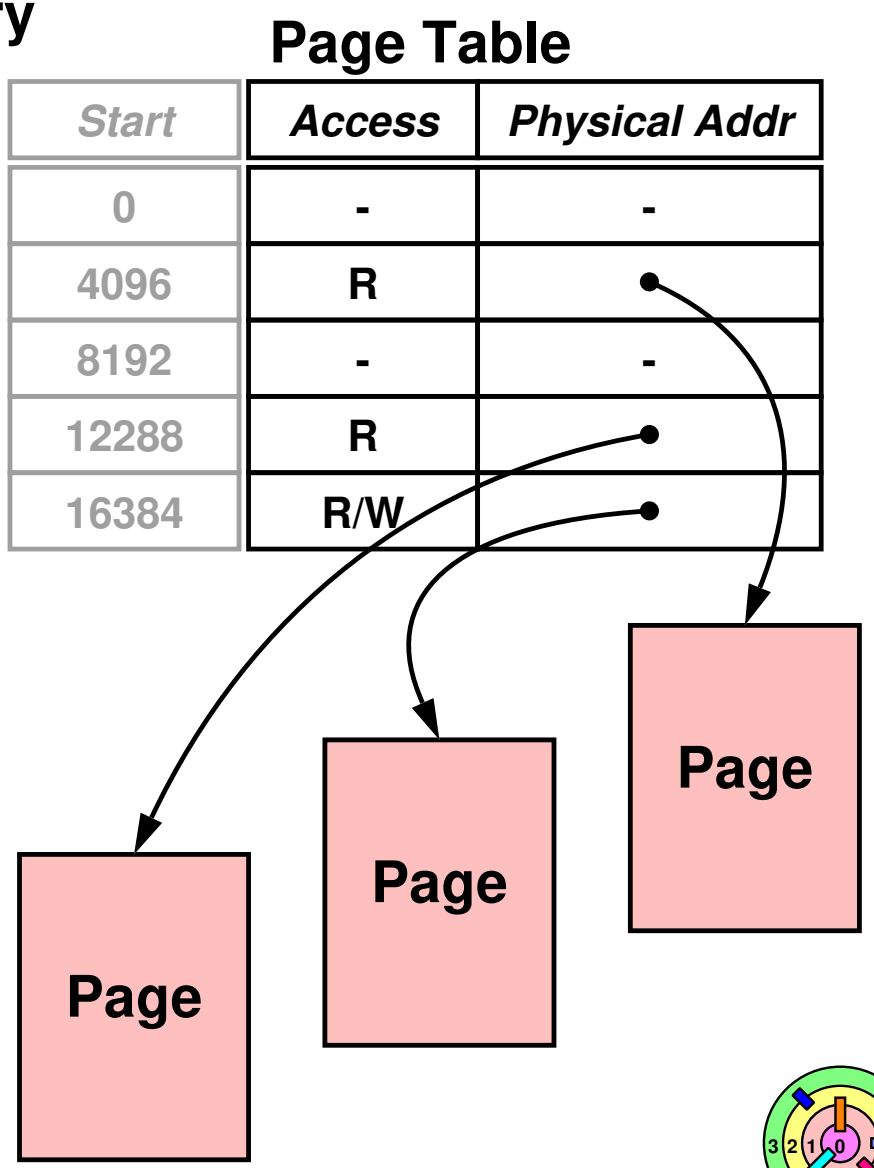
Memory Management Concerns

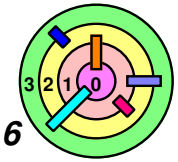
- In reality, the OS is too slow since *every* virtual address needs to be resolved
  - some of the virtual memory mechanisms must be built into the *hardware*
    - in some cases, the hardware is given the complete *"map"* (i.e., mapping from virtual to physical address)
    - in other cases, only a 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*



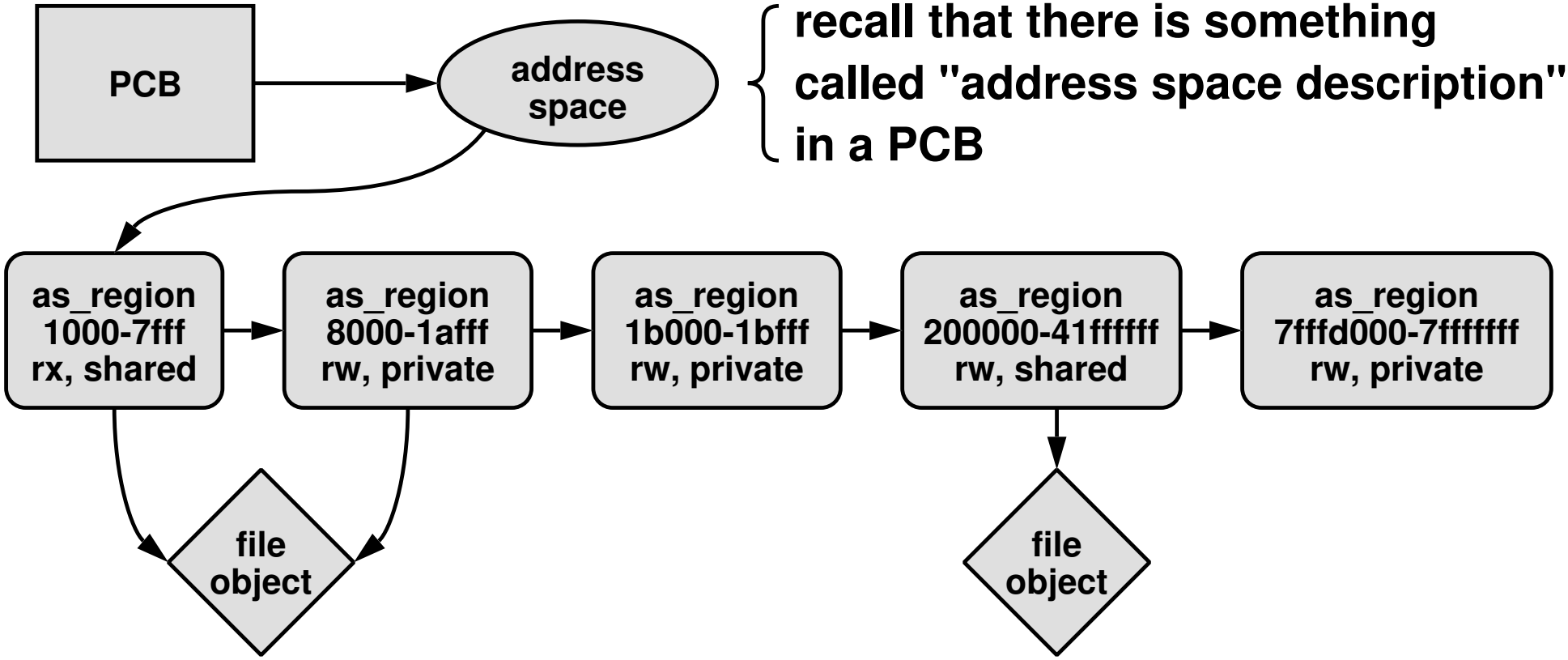
Memory Management Concerns

- 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*

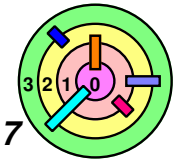




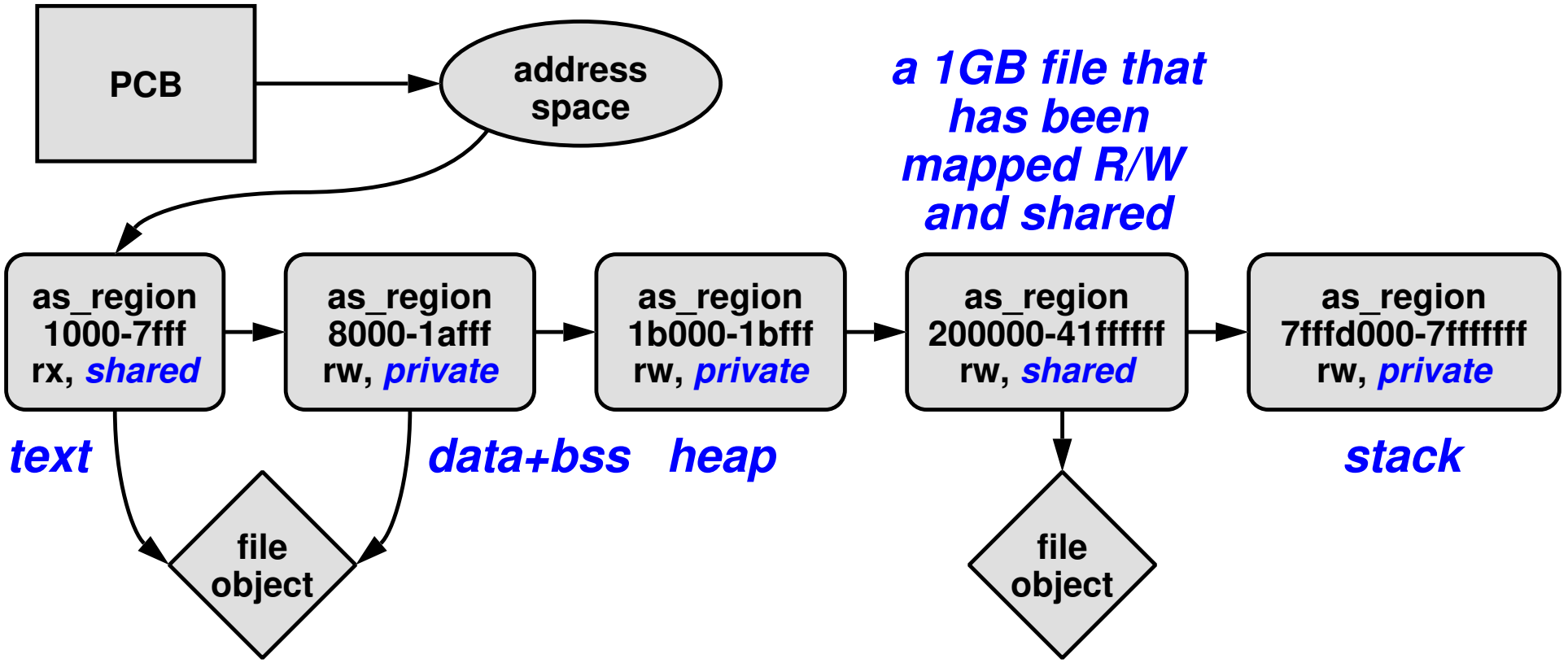
User Address Space Representation



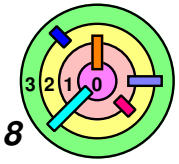
- `as_region` (address space region data structure) contains:
  - *start address*, *length*, *access permissions*, *shared* or *private*
  - if mapped to a file, pointer to the corresponding *file object*
- This is related to Kernel Assignment 3 where you need to create and manage *address spaces* / *virtual memory maps*



User Address Space Representation

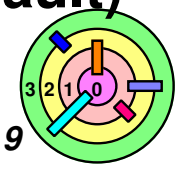


- In this example, text and data map portions of the same file
  - *text* is marked read-execute and *shared*
  - *data+bss* is marked read-write and *private* to mean that changes will be private, i.e., will not affect other processes exec'ed from the same file



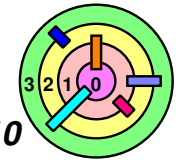
How OS Makes Virtual Memory Work?

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



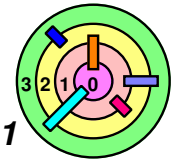
Storage Management

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



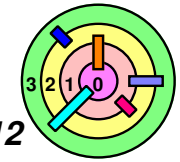
How Is The Primary Storage "Resource" Managed?

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



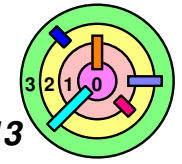
In Reality, Have To Deal With Mapped Files

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



In Reality, Have To Deal With Mapped Files

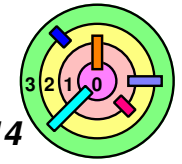
- It's difficult to be *fair*
  - it's difficult to even define what *fair* means
- We will discuss some solutions in Ch 7
  - for now, we use the following solution
    - give each participant (processes, file subsystem, etc.) a minimum amount of storage
    - leave some additional storage available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



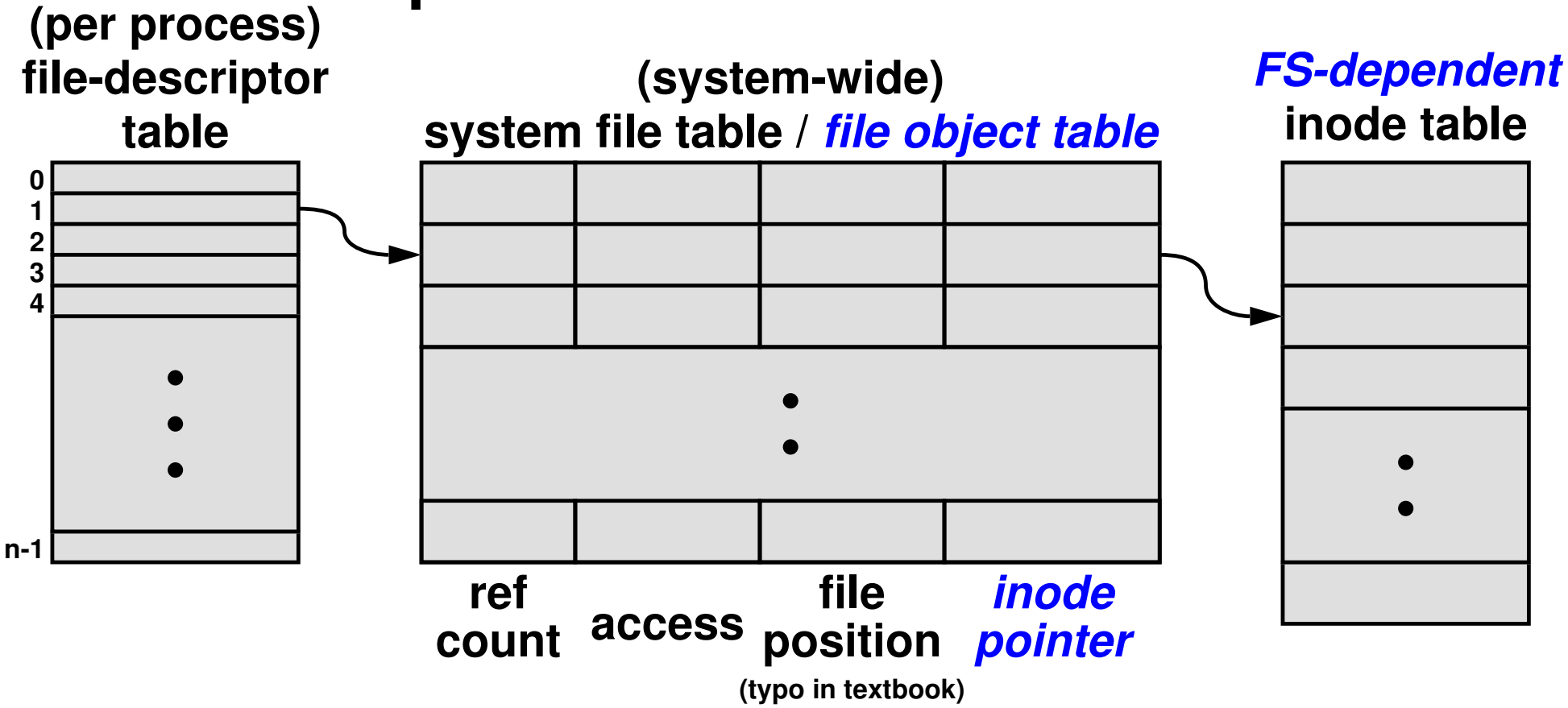
How Are Objects Managed In Secondary Storage?

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

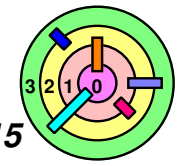
- The file system is usually divided into two parts
  - *file system independent*
    - supports the "file abstraction"
    - on Unix, this is called the *"virtual file system (VFS)"*
      - Kernel Assignment 2
    - on Windows, this is called the *"I/O manager"*
  - *file system dependent*
    - on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



Open-File Data Structures



- 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?

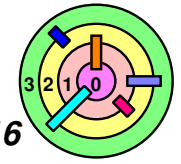


File Object

- 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);
  ...
};
```

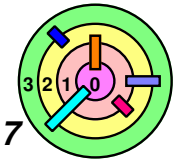
- But wait ...
  - what's this about C++?
    - real operating systems are written in C ...
    - checkout the `DRIVERS` kernel documentation (we skipped this `weenix` assignment)
      - similar trick (polymorphism) used in VFS



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