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 Space

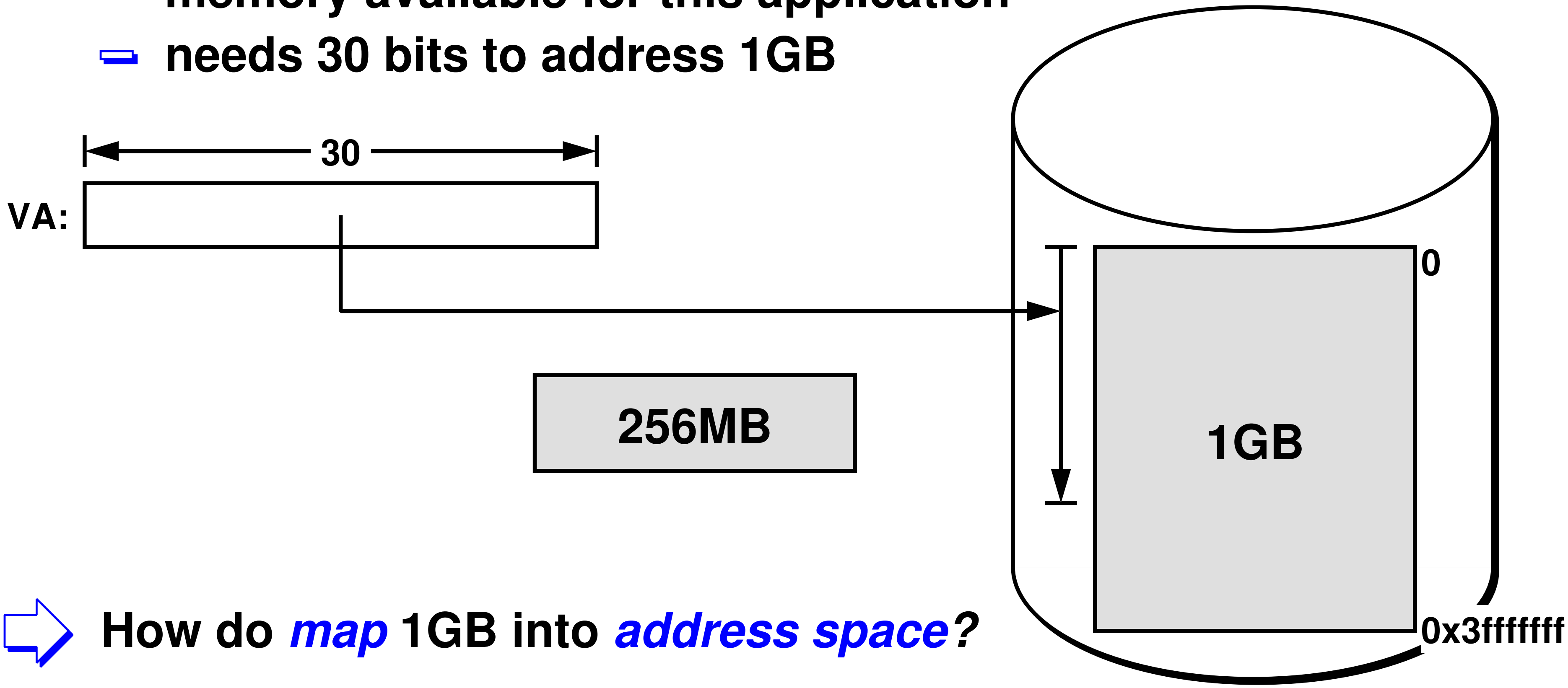
- Where to store data?
  - *primary storage*, i.e., physical memory
    - directly addressable
  - *secondary storage*, i.e., disk-based storage
- What would it take to support the idea of virtual memory, i.e., application's "view" of memory?
- An application only works with "virtual memory" (as far 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
- *Virtual Memory* ties everyting together!

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

Storage Space

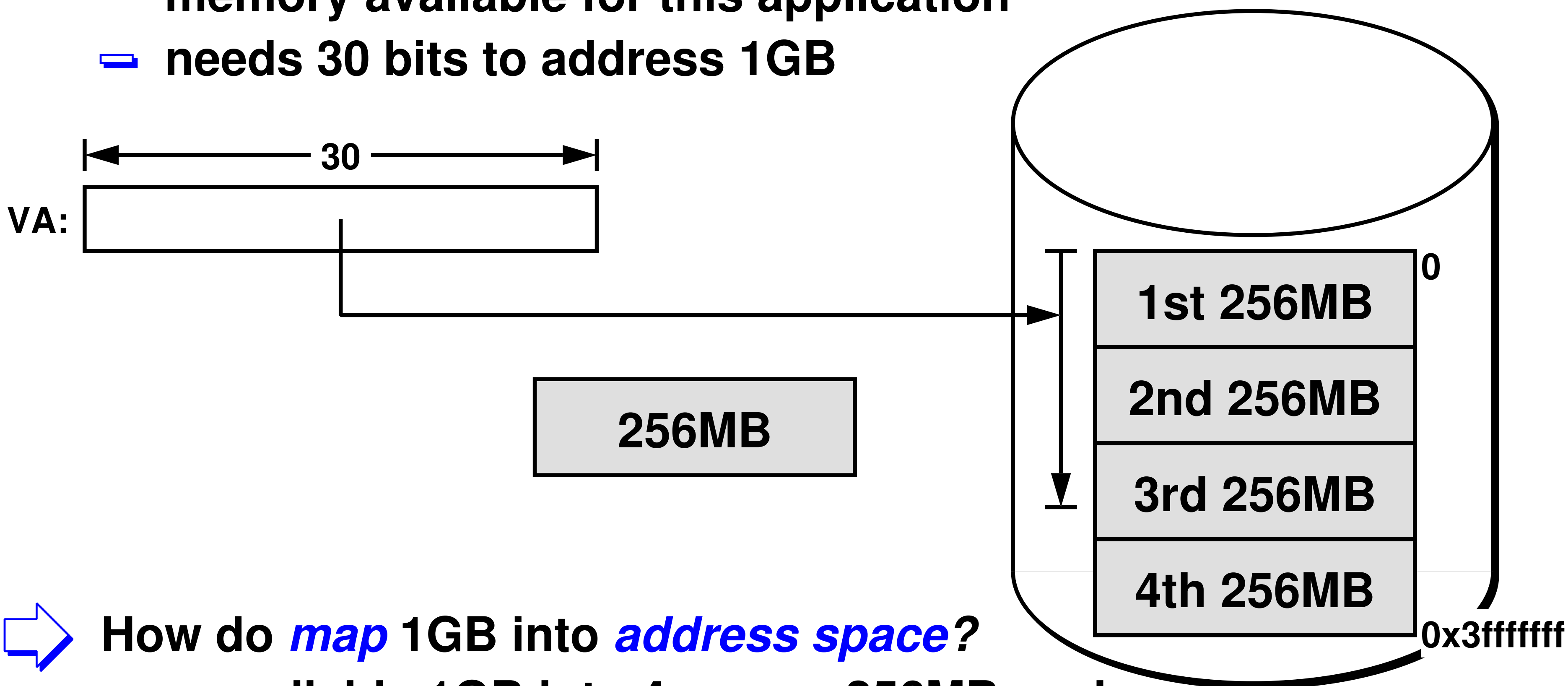
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



- How do *map* 1GB into *address space*?

Storage Space

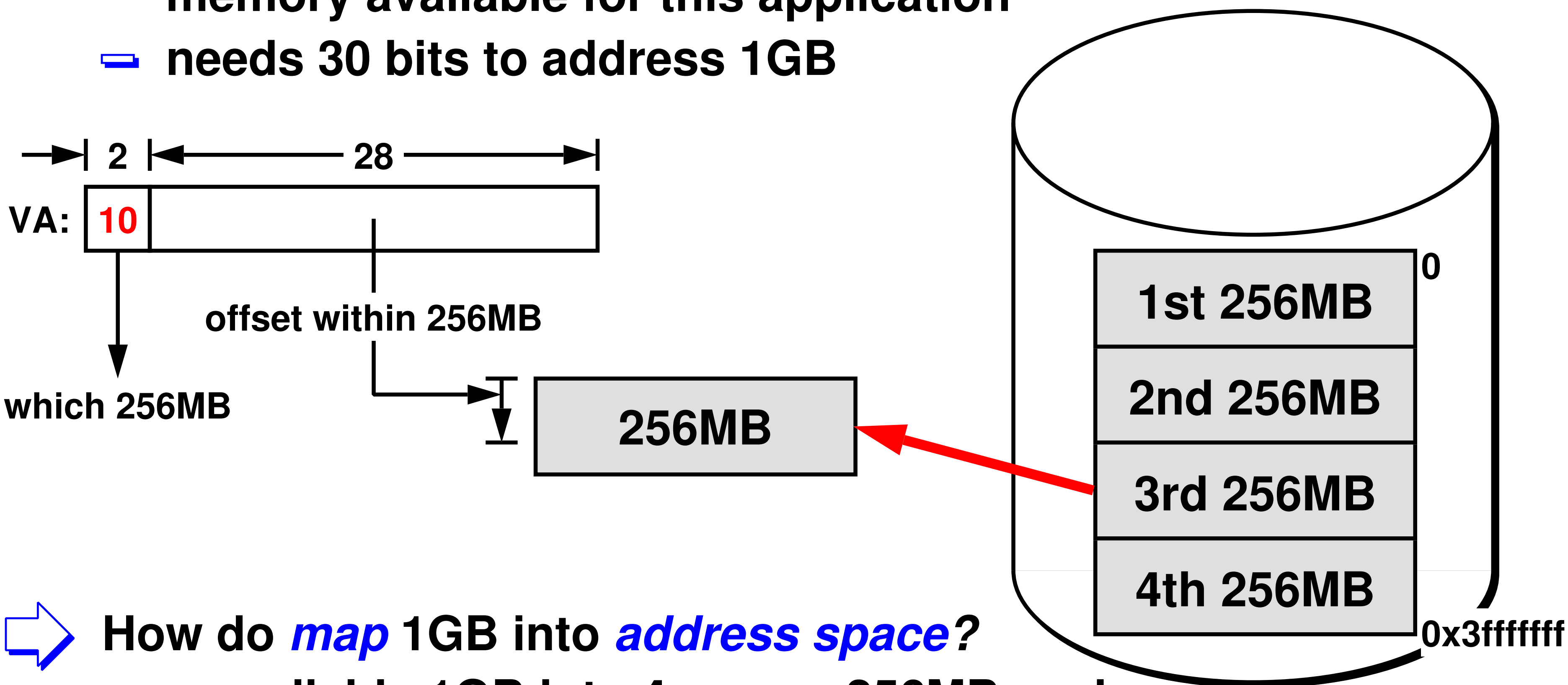
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



- How do *map* 1GB into *address space*?
  - e.g., divide 1GB into 4 *pages*, 256MB each

Storage Space

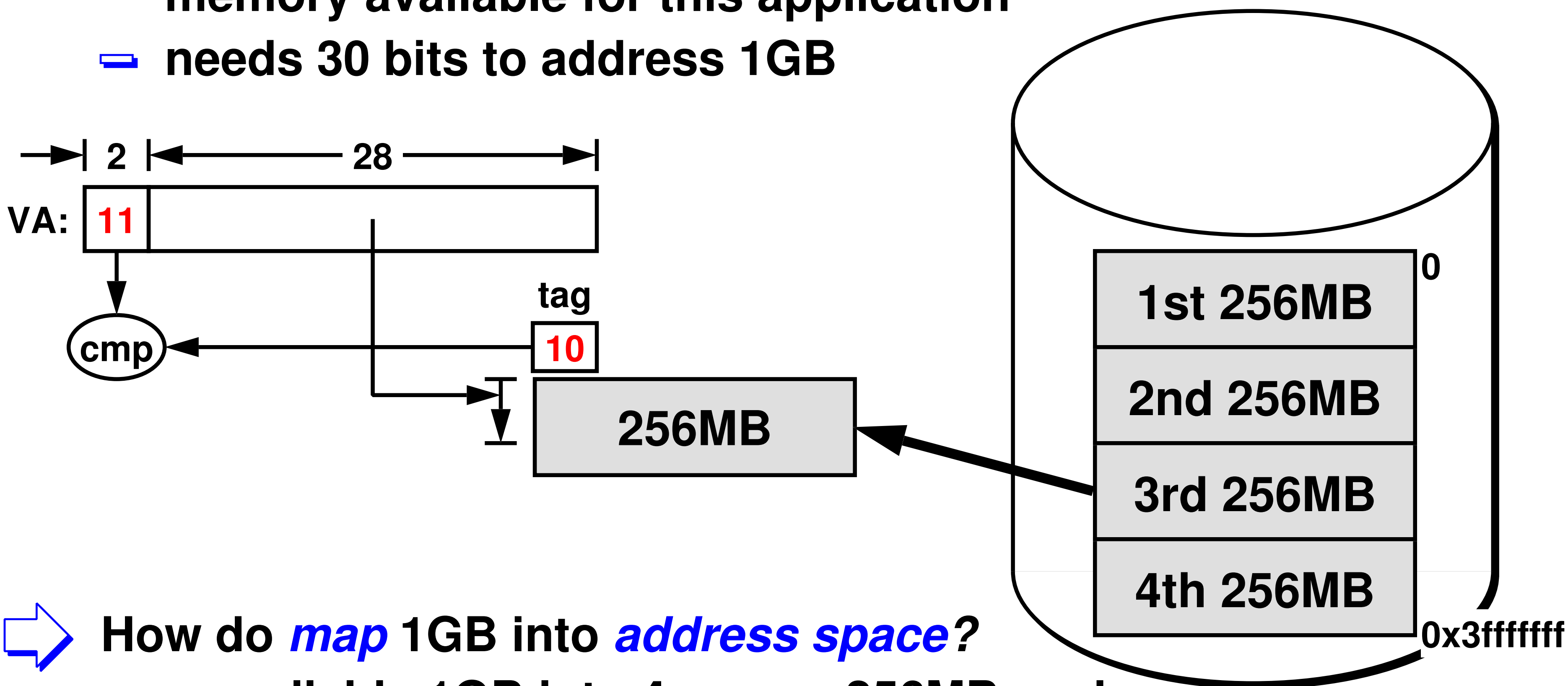
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



- How do *map* 1GB into *address space*?
  - 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

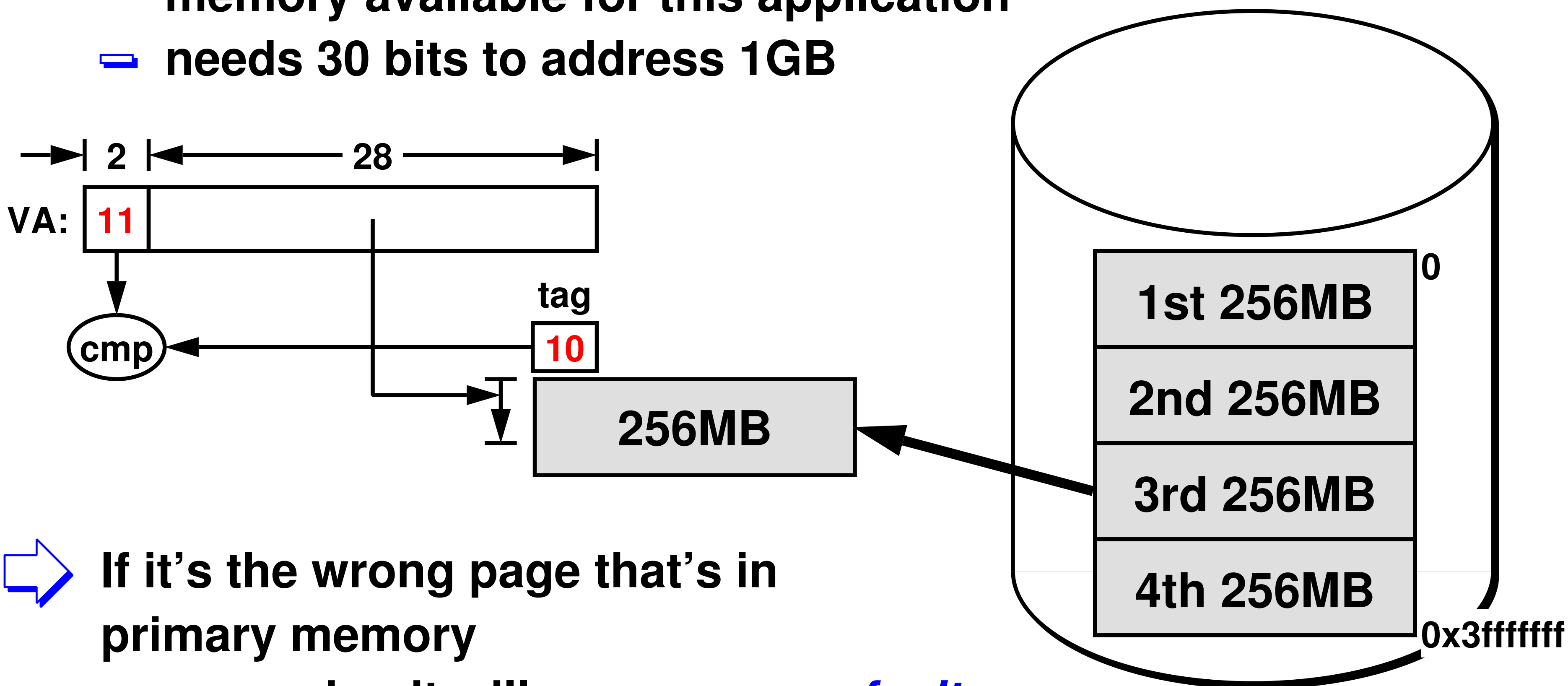
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



- How do *map* 1GB into *address space*?
  - 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

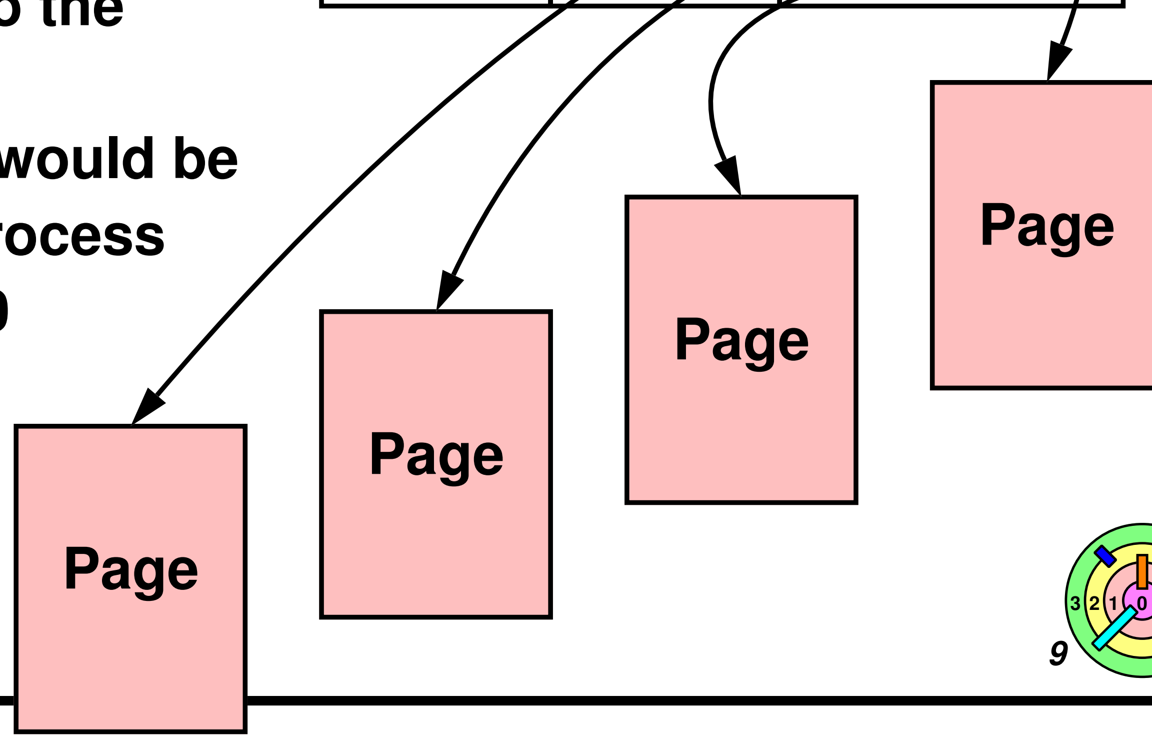
Segmentation Fault

- 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
  - e.g., virtual address 0

- A ***page fault*** is a ***segmentation fault***

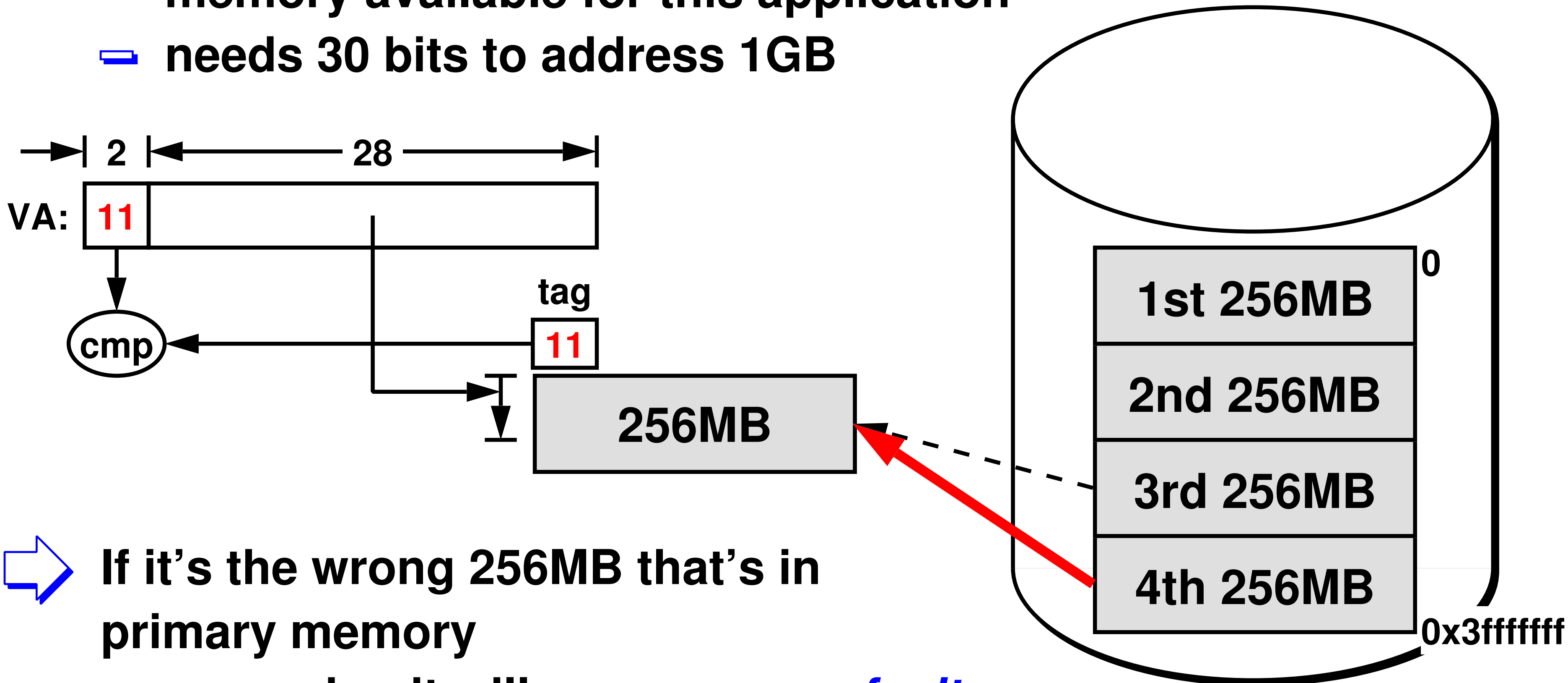
Page Table

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



Storage Space

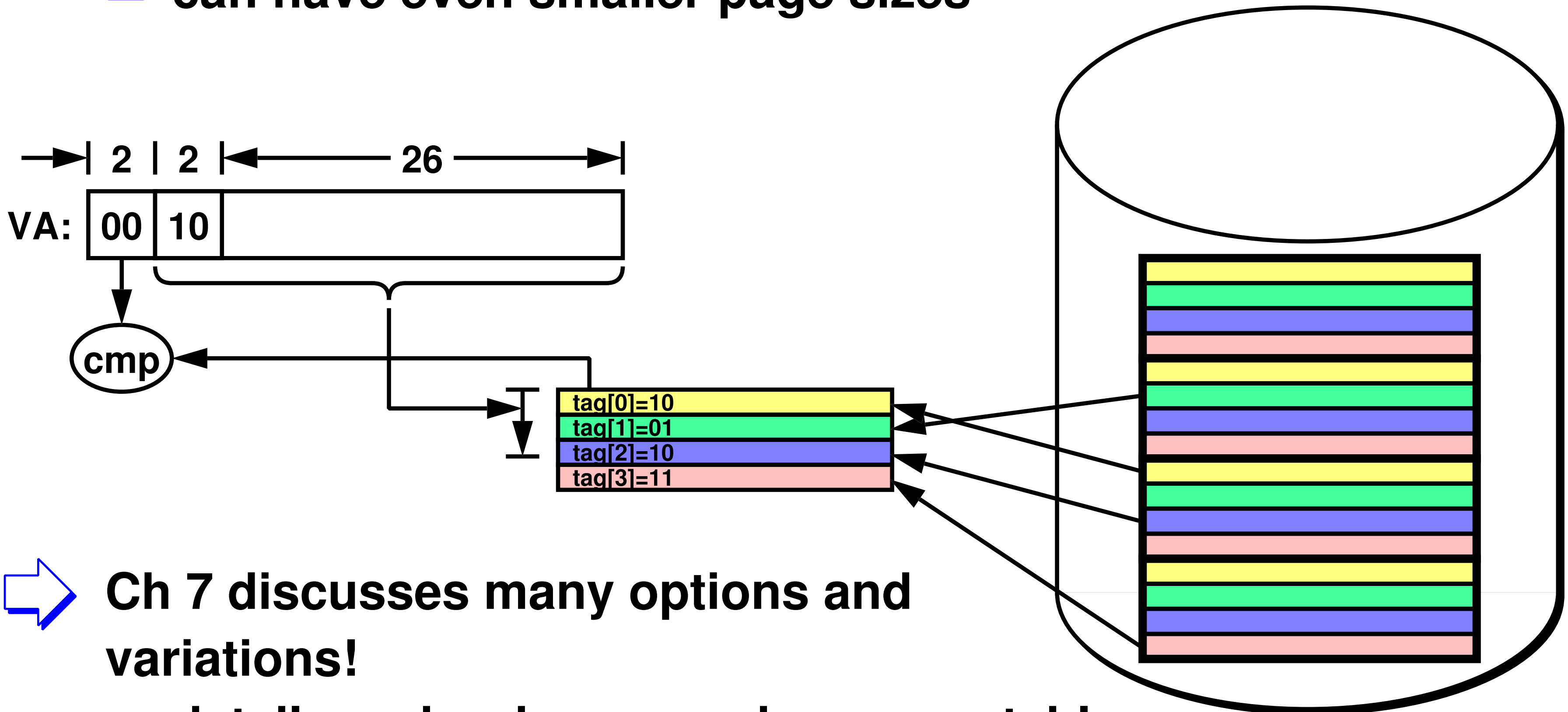
- A simple example of virtual memory
  - application needs 1GB but there is only 256MB of physical memory available for this application
  - needs 30 bits to address 1GB



- 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*
    - takes too long to copy 256MB
    - 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

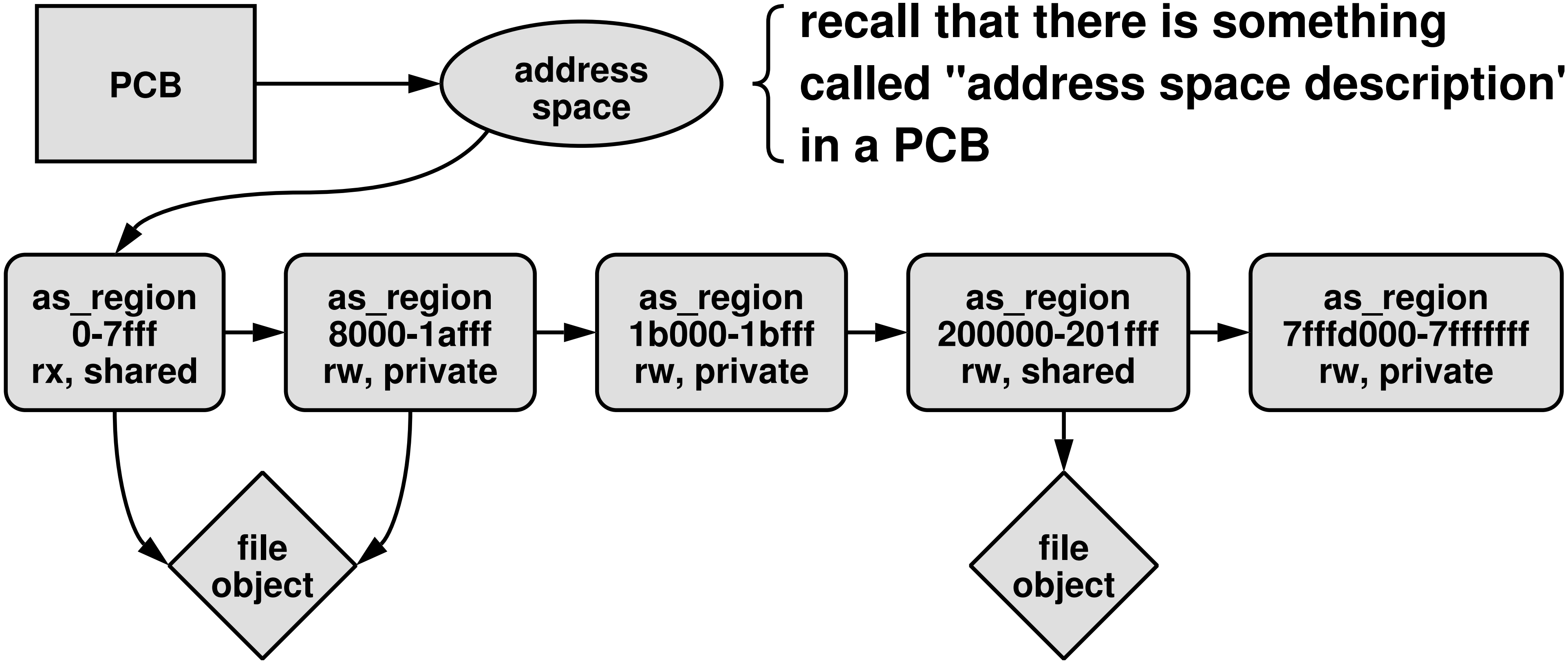


- Ch 7 discusses many options and variations!
  - details on hardware, such as page tables, translation look-aside buffers, etc.
  - details on OS software, such as how to implement *memory map*, *copy-on-write*, etc.

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

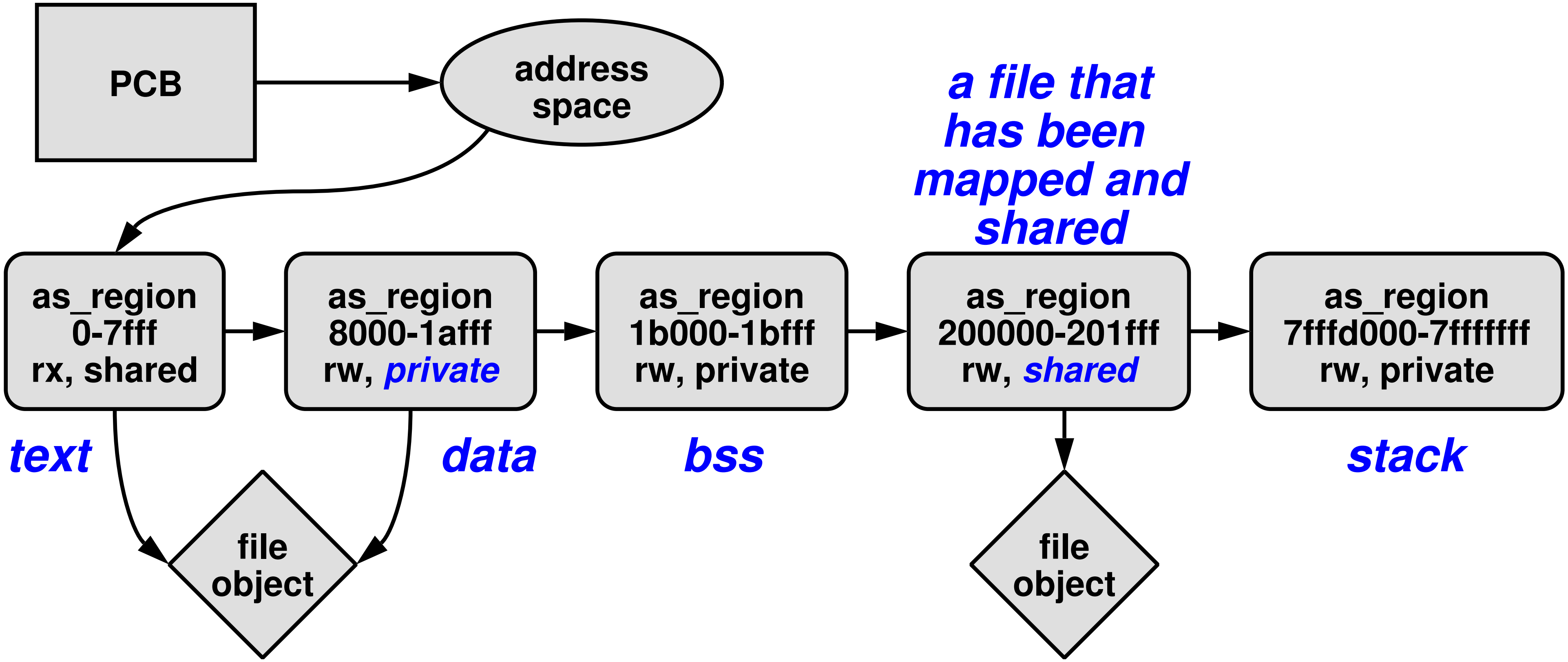


recall that there is something called "address space description" in a PCB

- `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 memory 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
      - 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!)

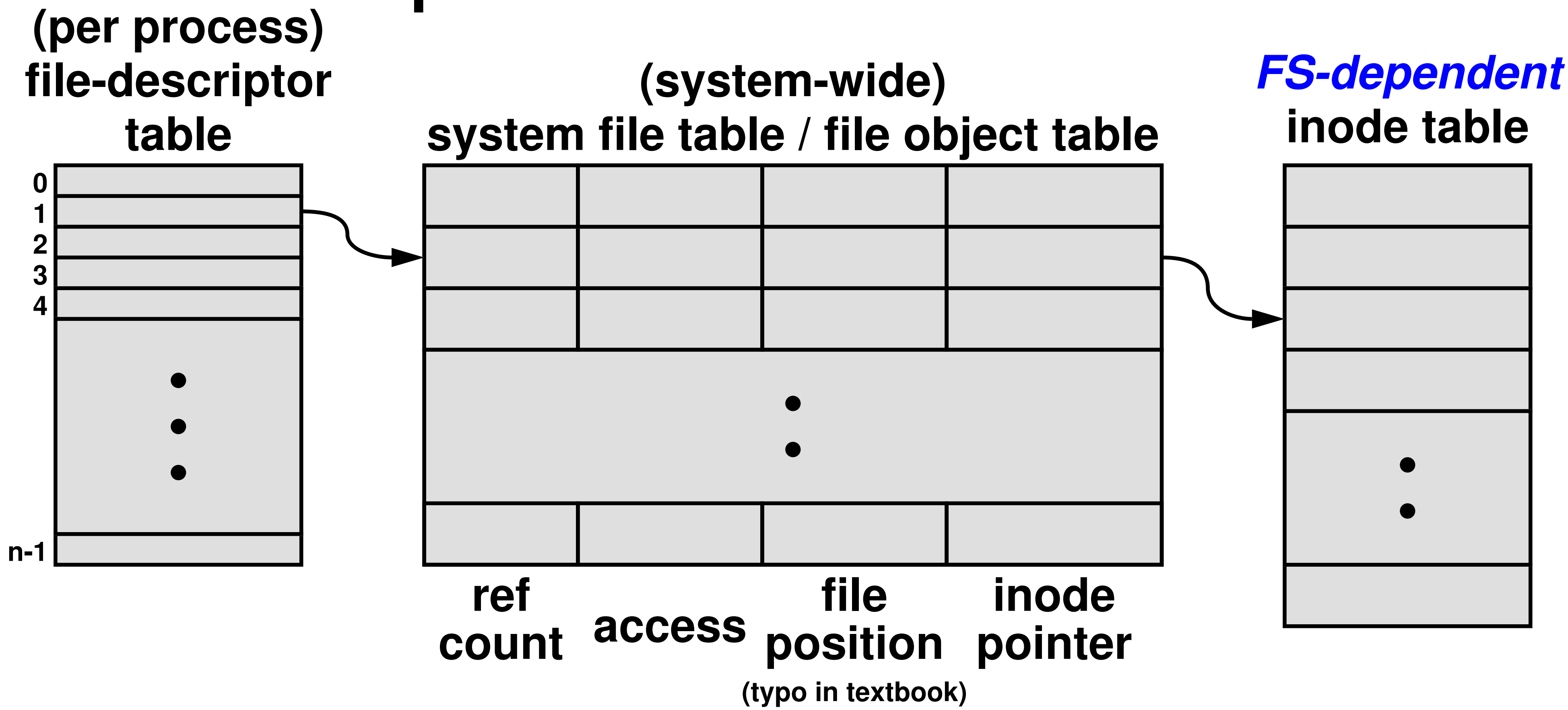
In Reality, Have To Deal With Mapped Files

- 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



- 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 actual file system (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;
  unsigned short 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)

File Object in C

```
typedef struct {
  unsigned short refcount;
  ...
  struct file_ops *file_op;
  /* function pointers (can use indirection) */
} 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***
- **More details in Ch 6**