CSE 451 Winter 2013

Section 7

Anton Osobov
aosobov@cs

Material adapted from previous offerings of CSE 451

Reminders

- Quiz tomorrow (2/22)
- Project 4 is up
  - Due Wednesday, 3/13
  - Group project

Topics for Today

- Project 3 Recap
- Virtual Address Spaces
- Project 4

Project 3 Recap

- How was performance?
  - Async vs. Sync?
  - Sync # of threads?
  - Async # of calls?
  - Buffer size?

Project 3 Recap

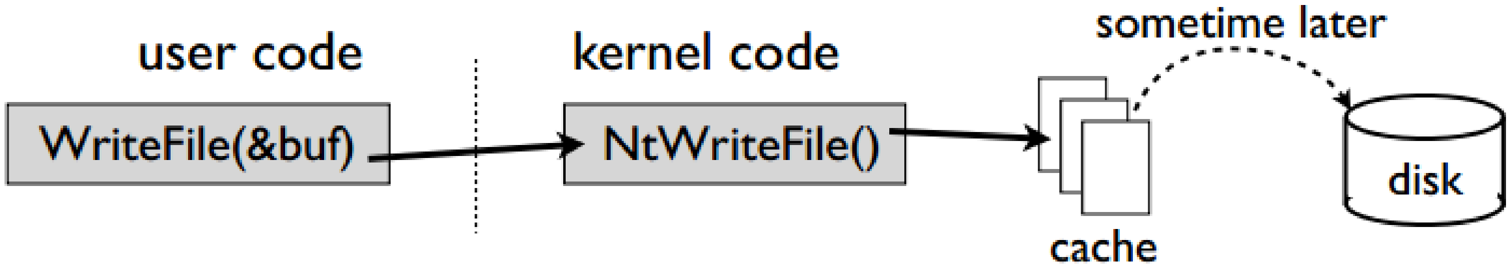
- Calls to disk are all sequential access!
  - Seems like concurrency won't help much…

Project 3 Recap

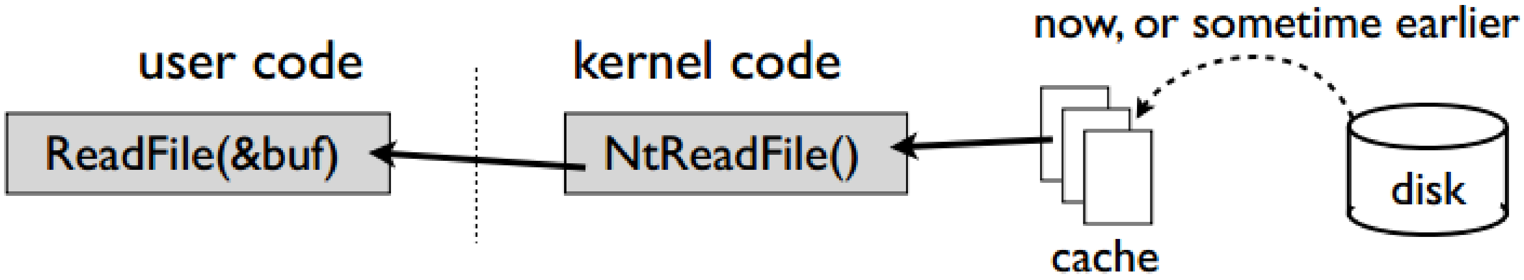
- Calls to disk are all sequential access!
  – Seems like concurrency won't help much…
- Disk Caching!
  – Optimizations to keep pages in memory

Project 3 Recap

- Write caching



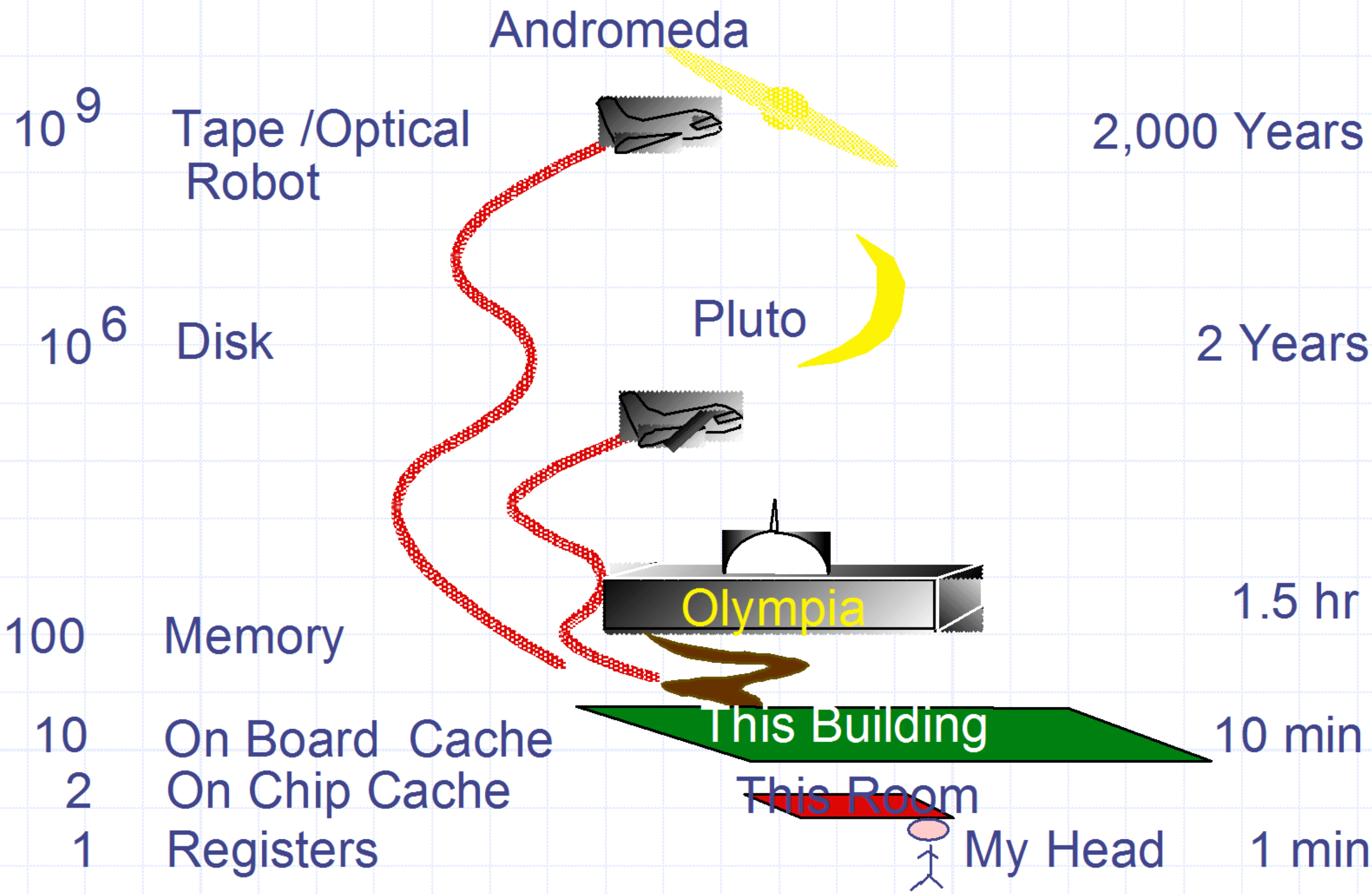
- Read caching



Project 3 Recap

- Disk scheduler can minimize amount of I/O between memory and disk
- Delay write to disk as long as possible
- Reads **must** be immediate
  - If a write occurs on a file, a read on the same file must fetch from disk

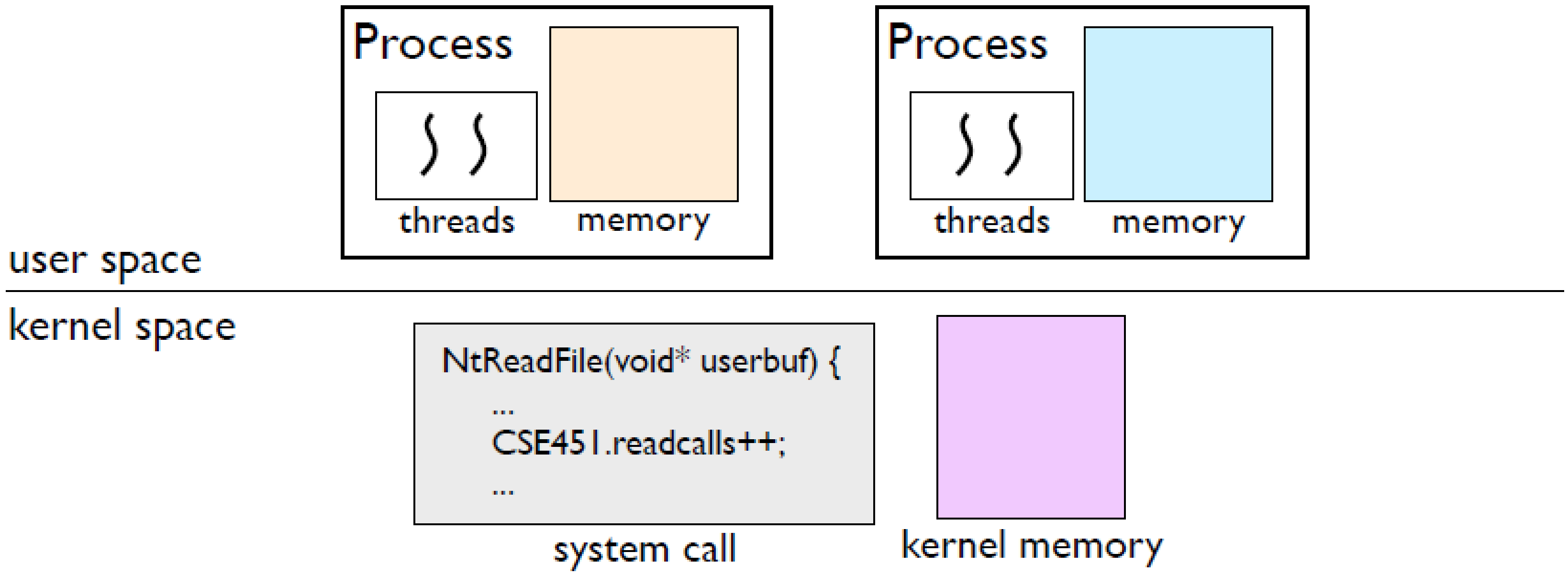
Storage Latency: How Far Away is the Data?



Topics for Today

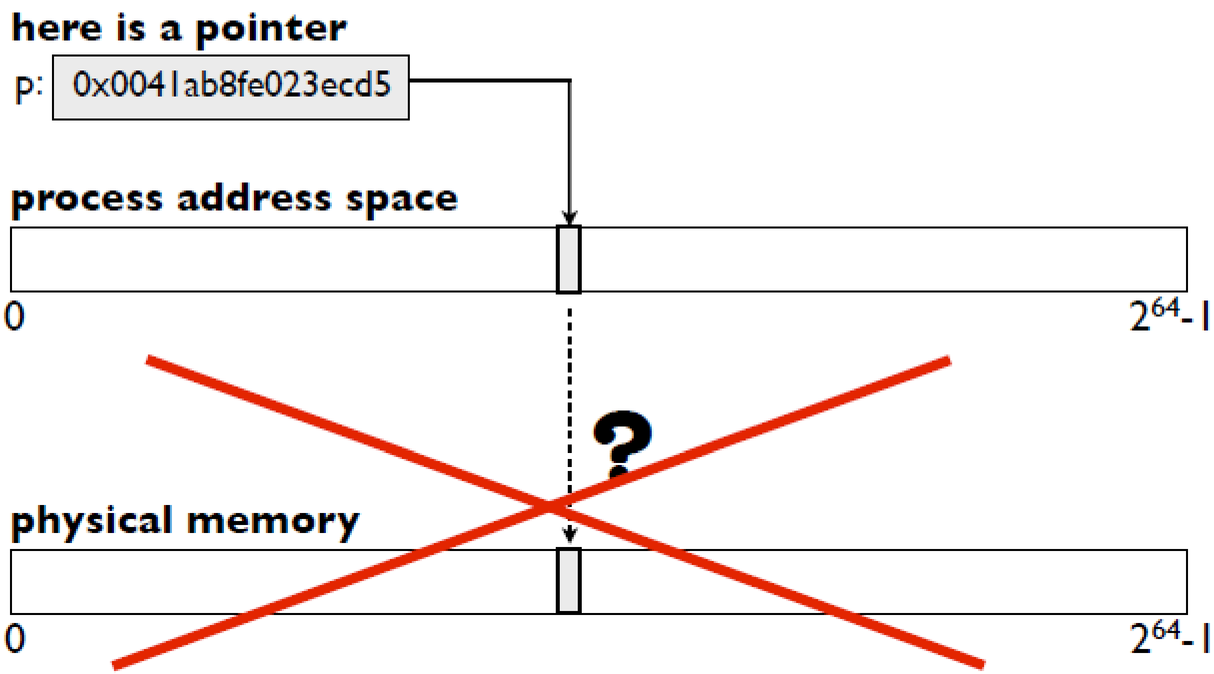
- ~~Project 3 Recap~~
- Virtual Address Spaces
- Project 4

Virtual Address Spaces

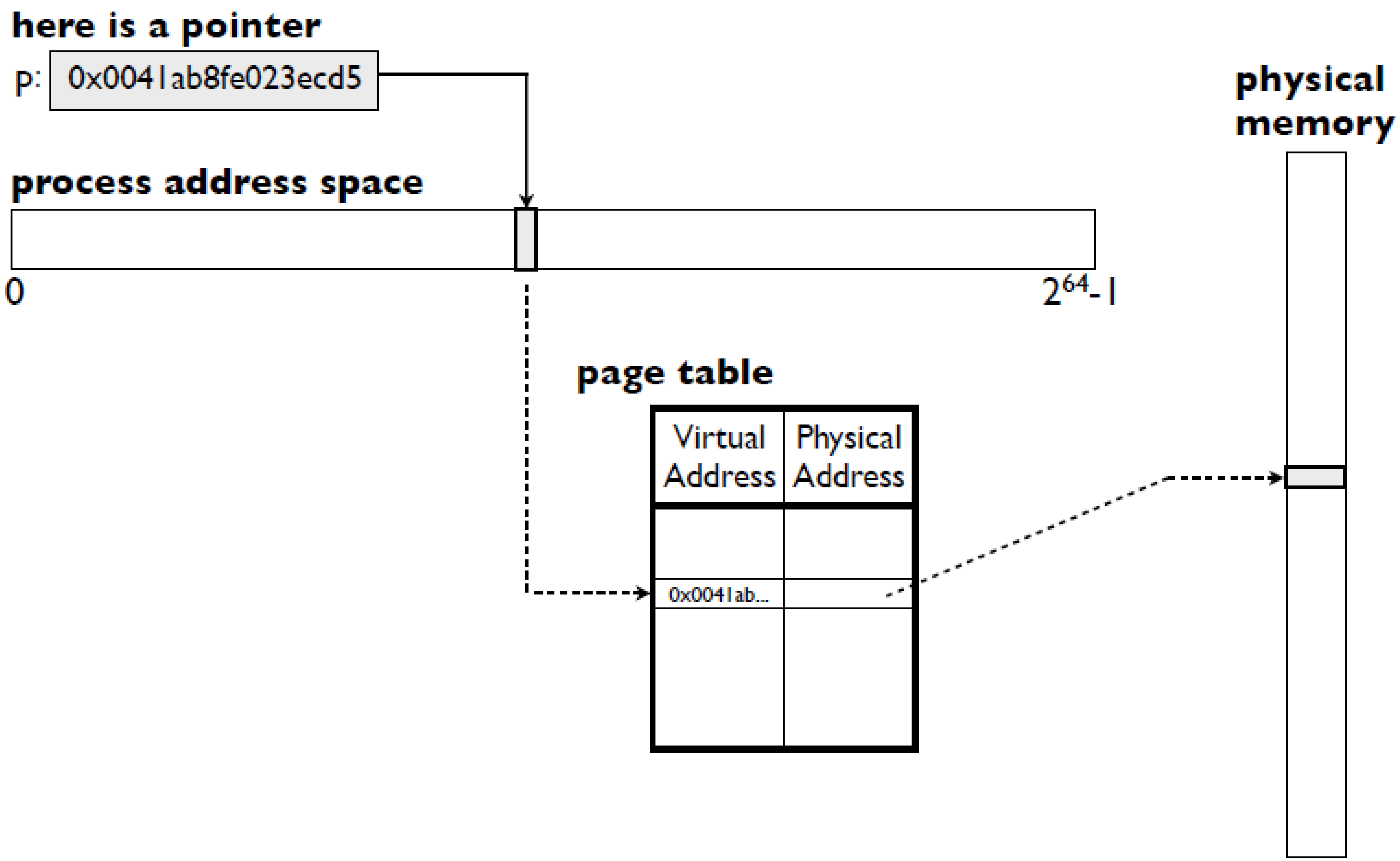


- Wait, if pointers are just numbers ...
  - how does each process get a private memory space?
  - how does the kernel get a private memory space?
  - how does the kernel access process memory?

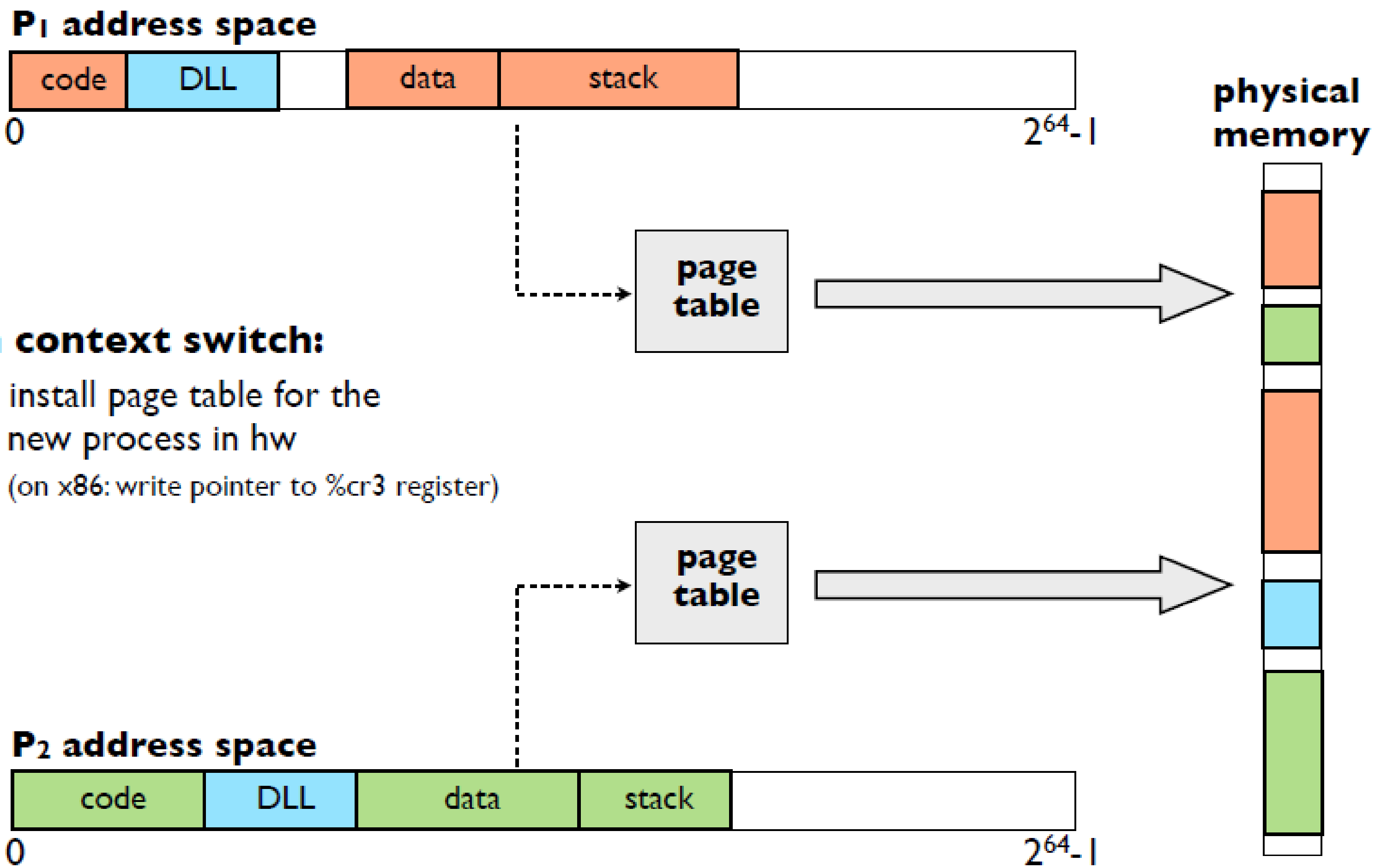
Virtual Address Spaces



Virtual Address Spaces



Virtual Address Spaces



On context switch:

- install page table for the new process in hw

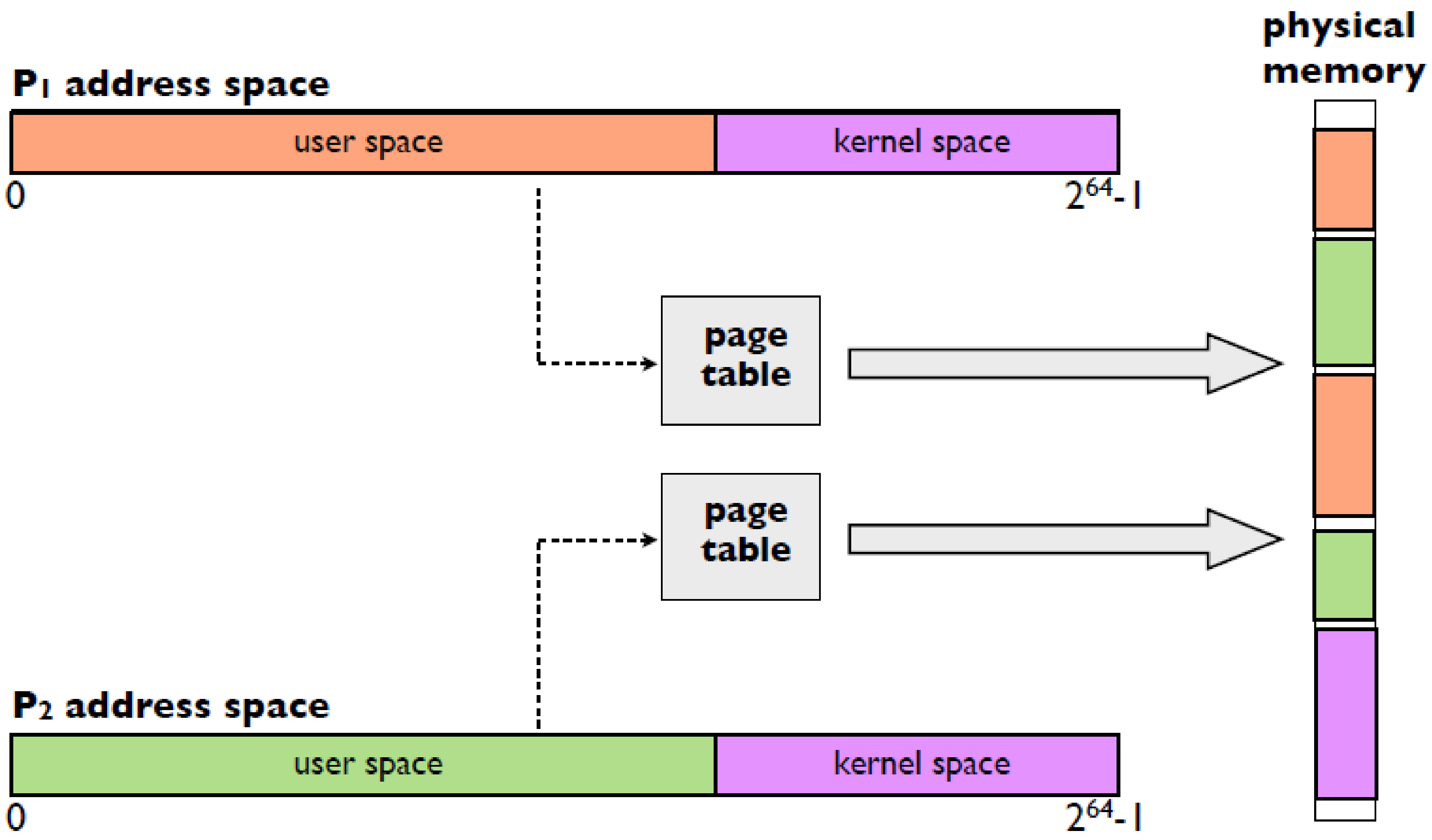
  (on x86: write pointer to %cr3 register)

Virtual Address Spaces

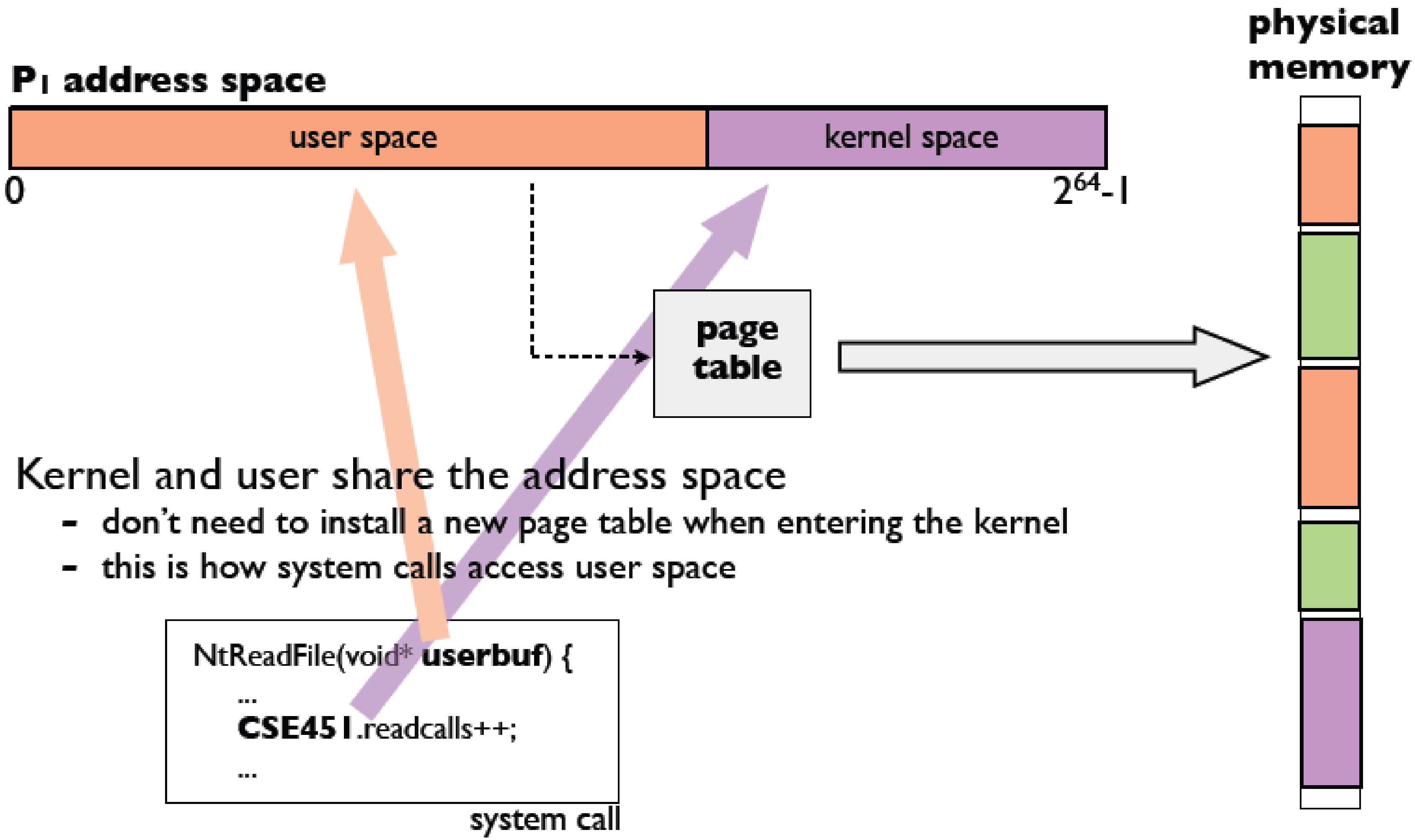
- Great, that explains how processes are isolated
- What about the kernel?
  - how does the kernel get a private memory space?
  - how does the kernel access user memory?

```
NtReadFile(void* userbuf) {
    ...
    CSE451.readcalls++;
    ...
```

Virtual Address Spaces

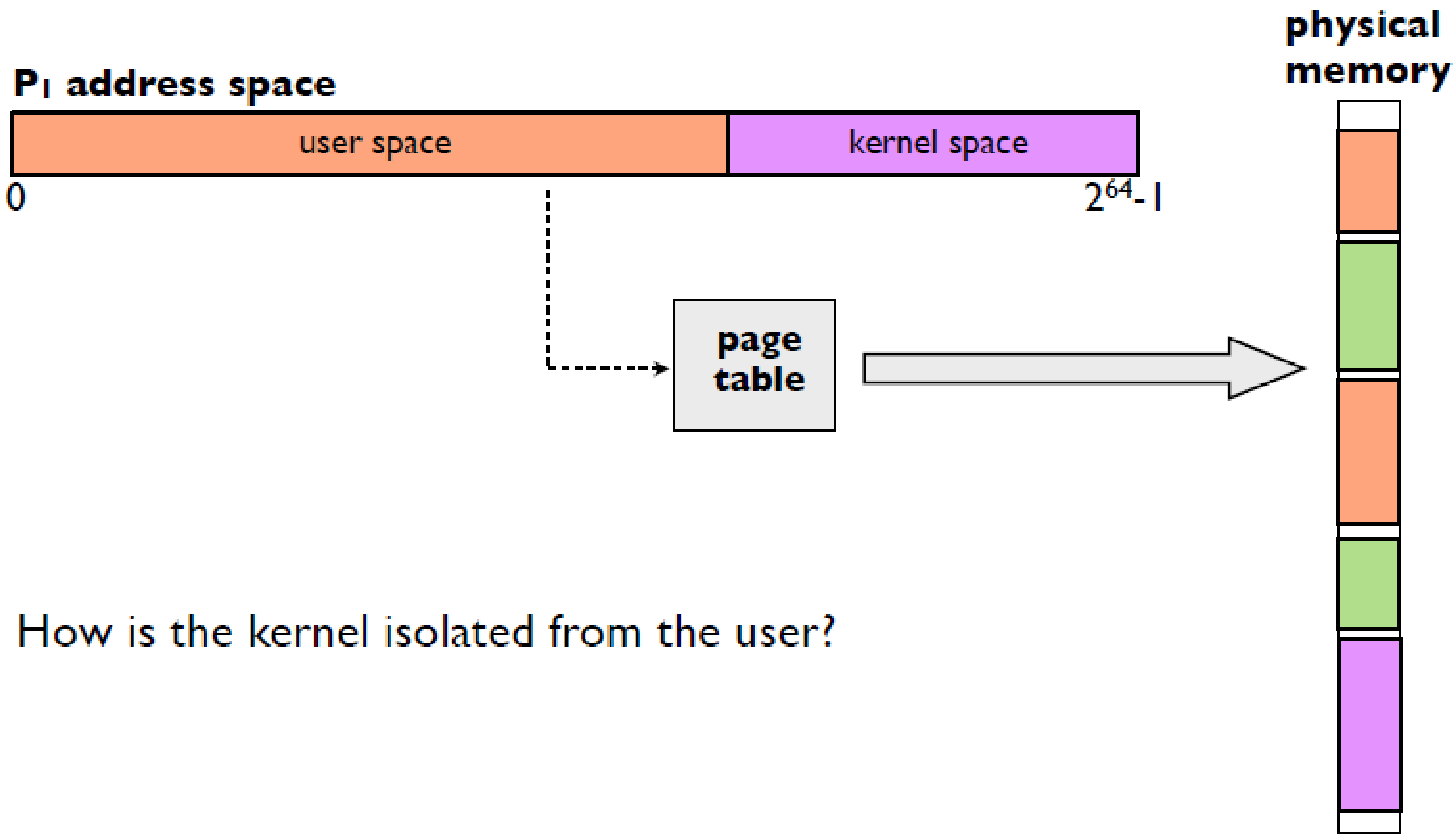


Virtual Address Spaces



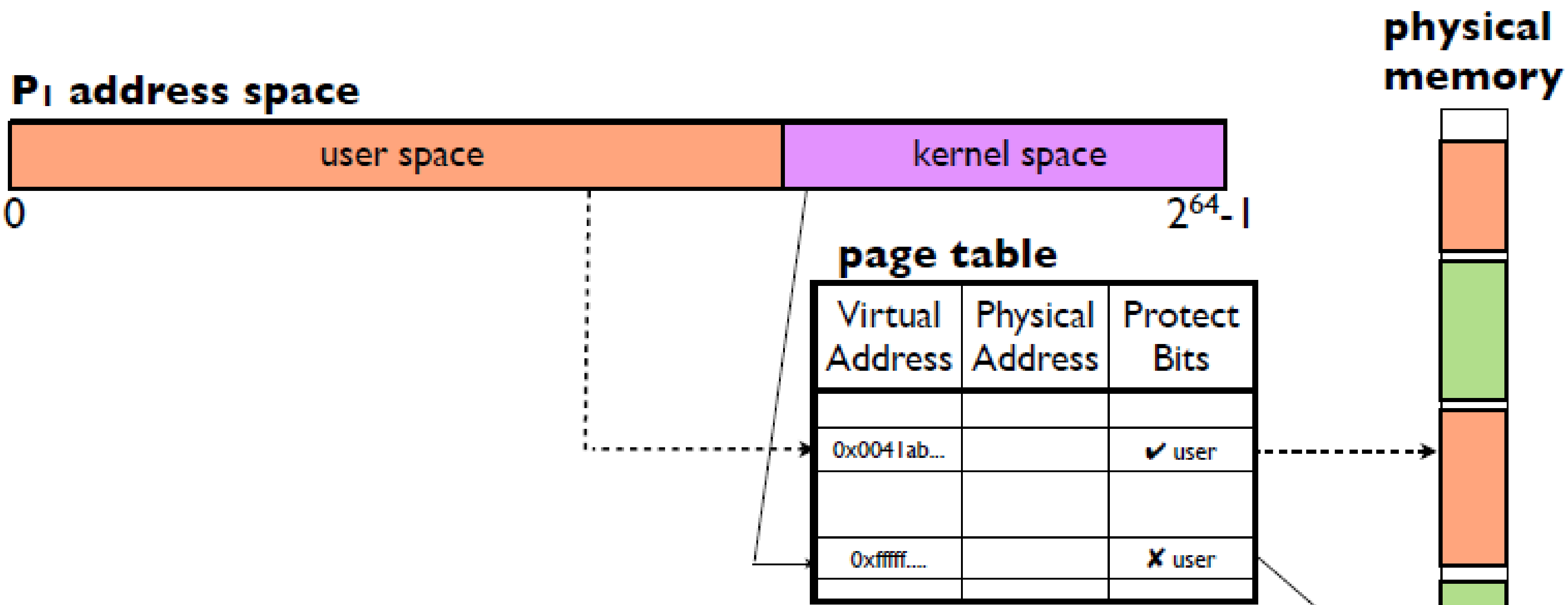
- Kernel and user share the address space
  - don't need to install a new page table when entering the kernel
  - this is how system calls access user space

Virtual Address Spaces



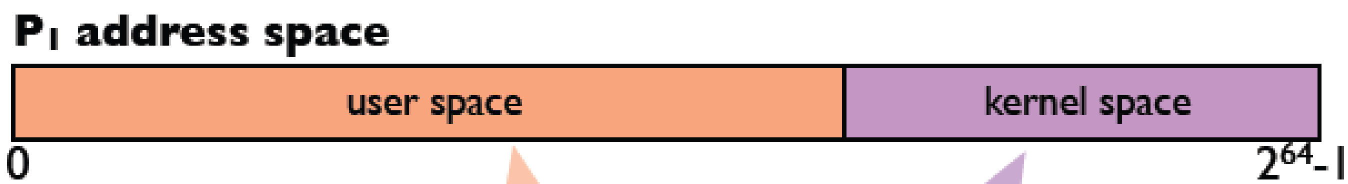
- How is the kernel isolated from the user?

Virtual Address Spaces



- How is the kernel isolated from the user?
  - set protection bits in the page table:
    hw triggers a page fault if user code tries to access kernel memory

Virtual Address Spaces

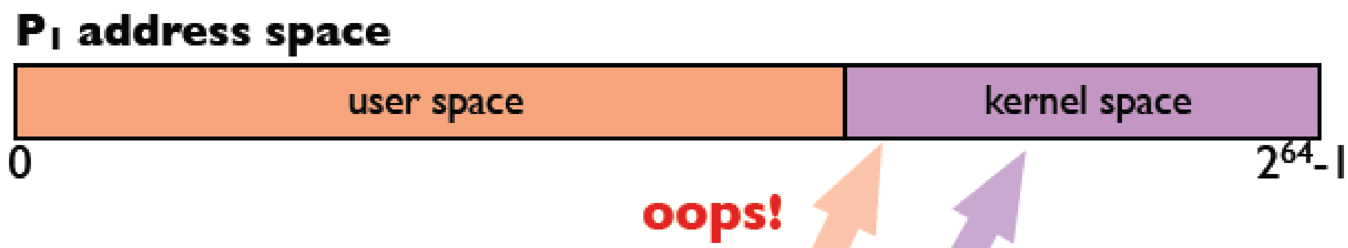


- So user and kernel share the address space. Great! What could possibly go wrong?

```
NtReadFile(void* userbuf,
             int   userlen)
{
    ...
    memcpy( userbuf,
            FileData,
            FileDataSize );
```

- What if **userbuf** is invalid?
  - e.g. NULL or
    - points at an unmapped page
- The kernel will segfault!

Virtual Address Spaces



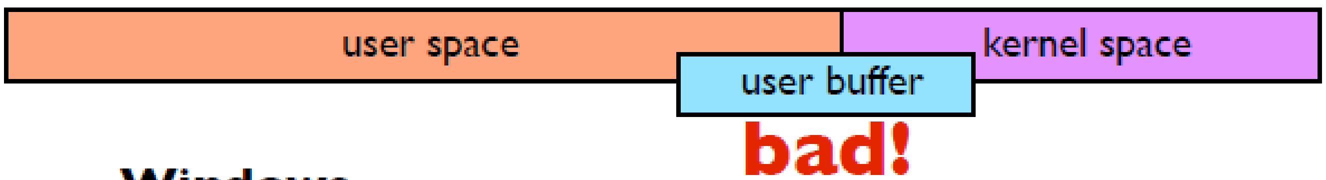
- So user and kernel share the address space. Great! What could possibly go wrong?

```
NtReadFile(void* userbuf,
              int    userlen)
{
    ...
    memcpy( userbuf,
              FileData,
              FileDataSize );
```

- What if **userbuf** points into kernel space?
  e.g. malicious user code "guesses" a pointer value
- The kernel data structures will be corrupted!

Always validate user pointers in the kernel

- Check that user pointers point at **user** memory, not kernel memory



Windows

```
ProbeForWrite( userbuf, length );
```

Linux

```
access_ok( userbuf, length );
```

- Guard kernel code that accesses user pointers against segfaults

Windows

```
try {
   memcpy( userbuf,
                FileData,
                FileDataLen );
} except {
   x = GetExceptionCode();
   ...  // oops, handle segfault
}
```

Linux

```
copy_to_user( userbuf,
                     FileData,
                     FileDataLen );
   // copy_to_user deals with
   // a segfault if it happens
```

Always validate user pointers in the kernel

- An example from Project 2:

```
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    // copy event buffer to user space
    memcpy( info->buffer, CseEventBuffer, info->bufferSize );
    ....
}
```

Always validate user pointers in the kernel

- An example from Project 2.
  Added a fix. Is this enough? What could go wrong?

```
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    ProbeForWrite( info->buffer, info->bufferSize );
    try {
        memcpy( info->buffer, CseEventBuffer, info->bufferSize );
    } except {
        ....
}
```

Always validate user pointers in the kernel

- An example from Project 2.
  Added a fix. Is this enough? What could go wrong?

What if another thread changes info->buffer after ProbeForWrite and before memcpy?

oops!

```
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    ProbeForWrite( info->buffer, info->bufferSize );
    try {
      memcpy( info->buffer, CseEventBuffer, info->bufferSize );
    } except {
      ....
}
```

Buggy user code example

Thread$_1$

```
NtQuerySystemInformation(info);
```

Thread$_2$

```
info->buffer = 0xfff....;
```

(a kernel address)

Always validate user pointers in the kernel

- An example from Project 2.
  The full fix:

capture

```
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    tmpBuffer = info->buffer;   // capture pointer
    tmpSize   = info->bufferSize;
    ....
    ProbeForWrite( tmpBuffer, tmpSize );
    try {
        memcpy( tmpBuffer, CseEventBuffer, tmpSize );
    } except {
        ....
}
```

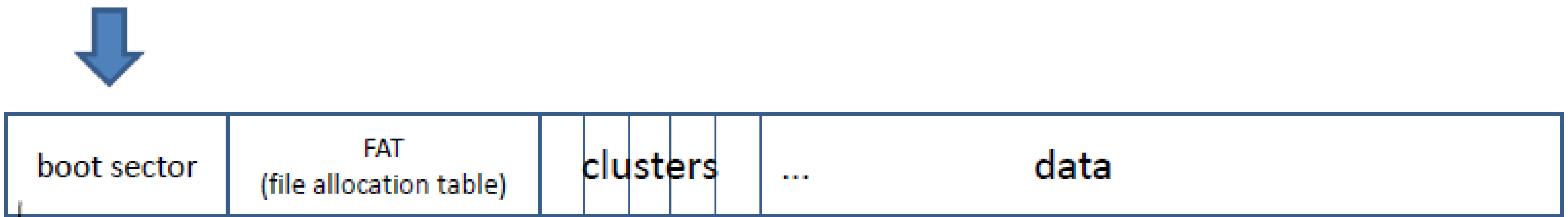
Topics for Today

- ~~Project 3 Recap~~
- ~~Virtual Address Spaces~~
- Project 4

Project 4

- Goals: Modify the FAT file system to
  - Make all directories sortable
  - Compact directories

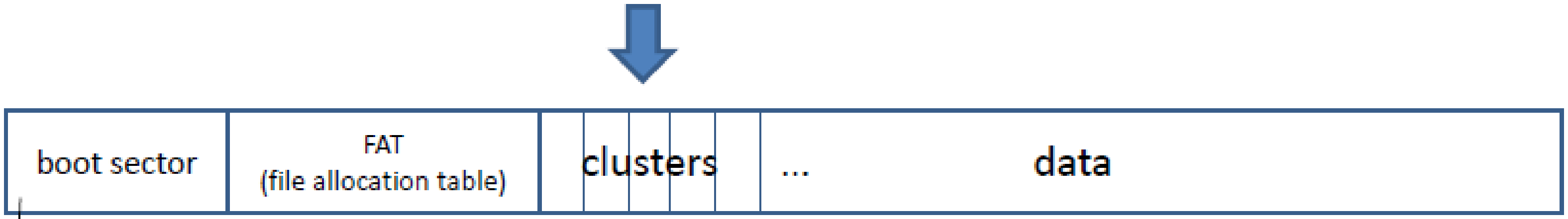
The FAT File System



Goal of FAT: store files and directories!

Size of FAT
Size of data area
Size of each cluster
Location of root dirent

The FAT File System



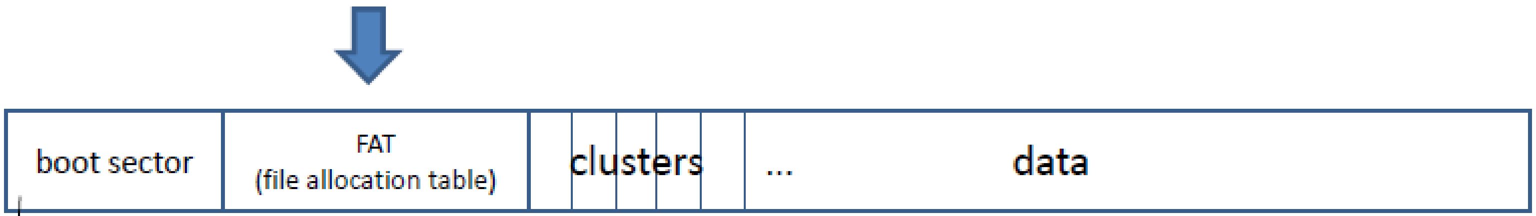
Size of FAT
Size of data area
Size of each cluster
Location of root dirent

Goal of FAT: store files and directories!

Each cluster either:

- Stores data for a file or…
- Stores lists of files in a directory (dirent)

The FAT File System



Size of FAT
Size of data area
Size of each cluster
Location of root dirent

Goal of FAT: store files and directories!

Each cluster either:

- Stores data for a file or...
- Stores lists of files in a directory (dirent)

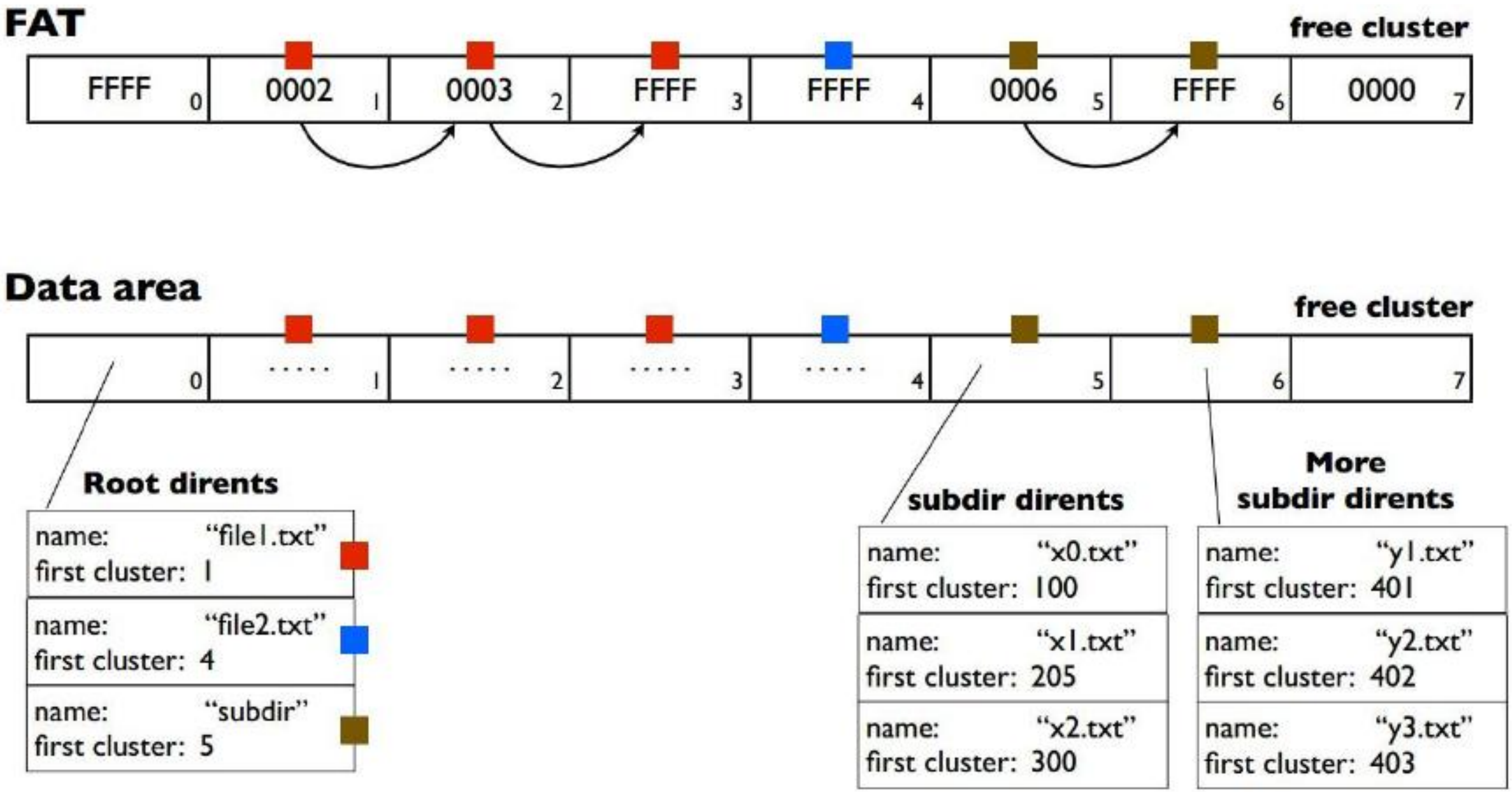
File Allocation Table

- Linked list of clusters
- As many entries as there are clusters

The FAT File System

- So, how do we get files?

The FAT File System



Project 4

- Goal: keep dirents sorted in each directory
  - Note: This means implementing your own sorting algorithm!

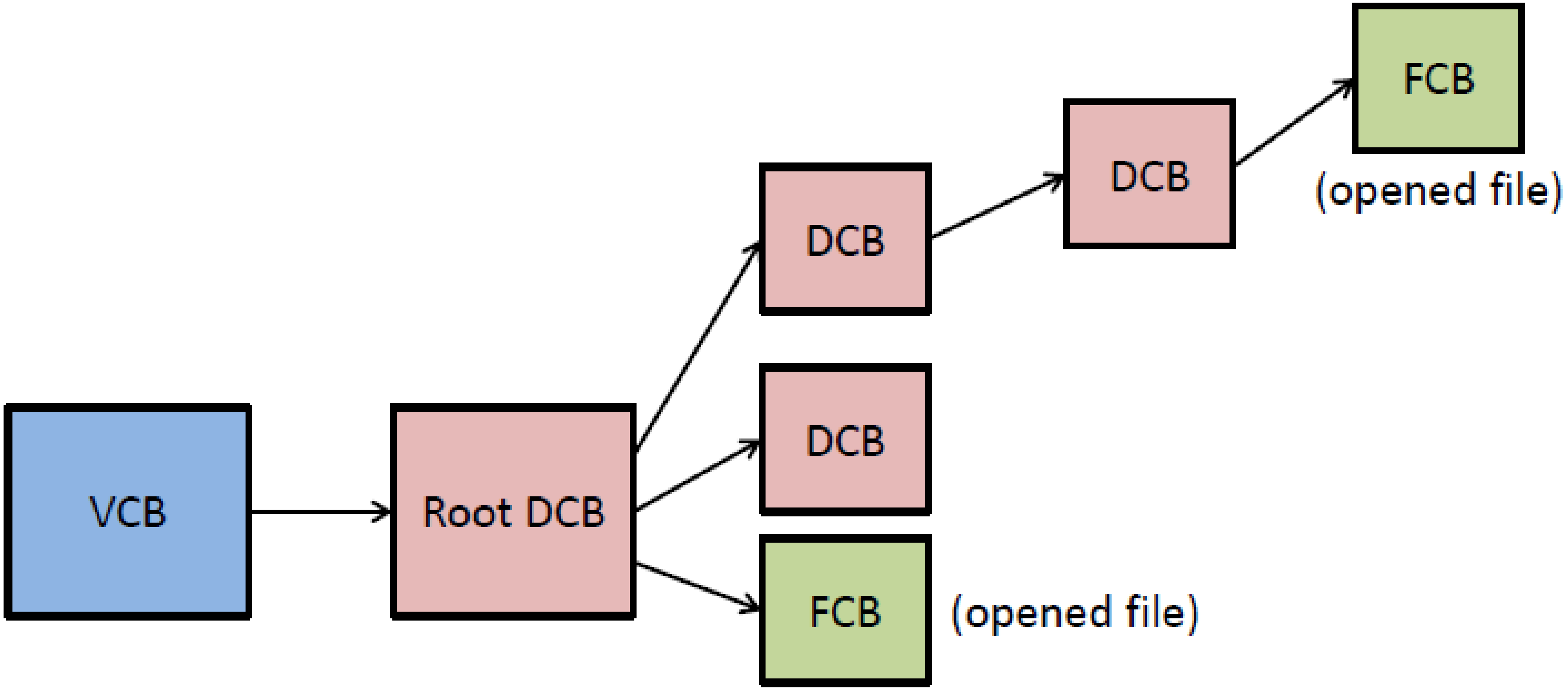
PACKED_DIRENT (from fat.h)

```
FileName:                          "file1.txt"
LastWriteTime:                     …
FirstClusterOfFile:                1
FileSize:                          4052
```

Project 4

- Kernel data structures: on-disk (fat.h)
  - PACKED_BOOT_SECTOR (boot info, etc – don't modify)
  - BIOS_PARAMETER_BLOCK (boot info, etc – don't modify)
  - PACKED_DIRENT(DIRENT struct)
- Kernel data structures: in-memory (fatstruct.h)
  - VCB (info about mounted volume)
  - FCB (cached files)
  - DCB (cached directories)

Project 4



Project 4

- Resort dirents when:
  - Creating a new file (name, extension, cluster number)
  - Closing a file (timestamp, size)
  - Re-sorting the entire dirent

Topics for Today

- ~~Project 3 Recap~~
- ~~Virtual Address Spaces~~
- ~~Project 4~~