Discussion Week 5

TA: Kyle Dewey

Overview

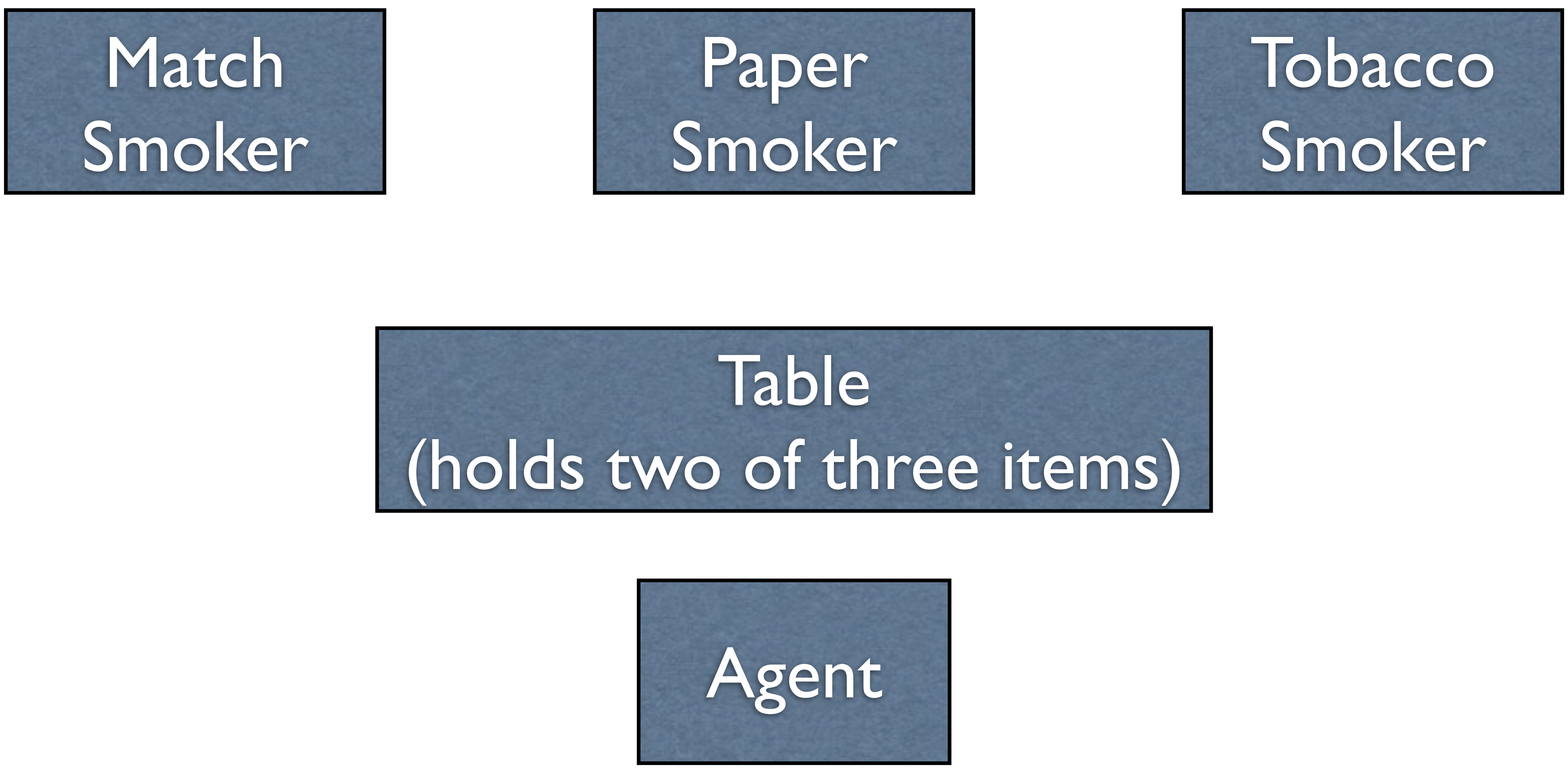
- HW 3.10 and 6.2 review
- Binary formats
- System call execution in NACHOS
- Memory management in NACHOS
- I/O in NACHOS

Homework 3.10

- "Identify the values of `pid` at lines A, B, C, and D. Assume that the actual pids of the parent and child are 2600 and 2603, respectively."

Homework 6.2

- The Cigarette-Smokers Problem



Problem Specifics

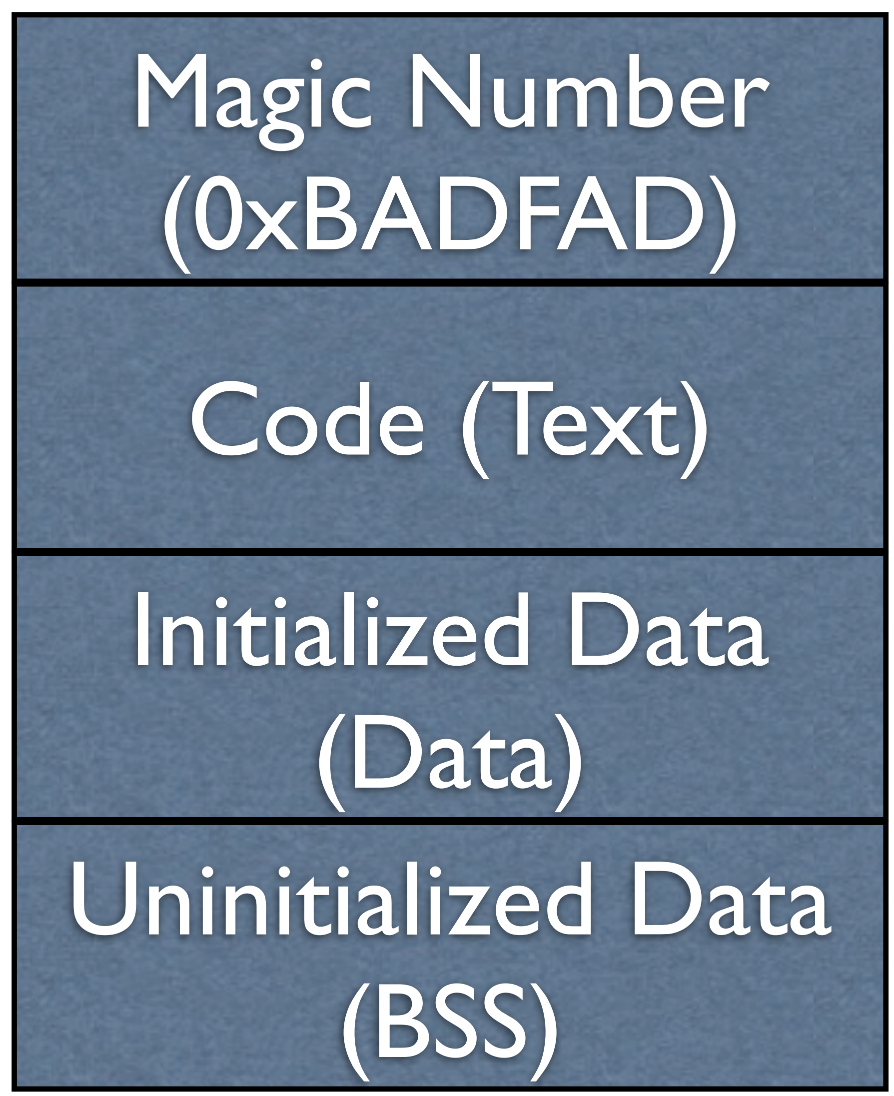
- Agent places two items
- Smoker with remaining item grabs the two and smokes
- The process repeats

Java Implementation

Binary Formats

NOFF

- NACHOS Object File Format



Why Bother?

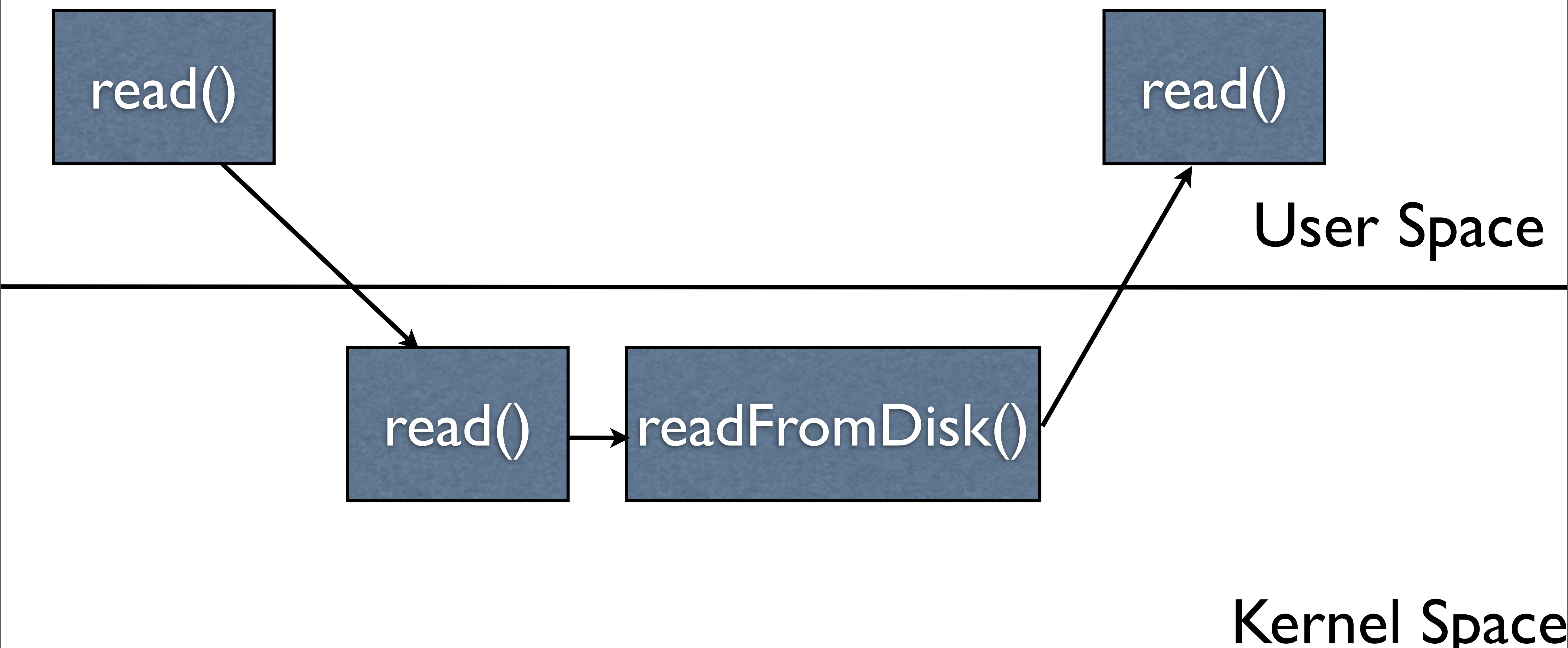
- CPU sees only a stream of instructions
- All gets loaded into memory anyway

Advantage

- Tells OS roughly how portions will be used
- Optimizations possible
  - Share (reentrant) code and constant data
  - Prevent execution of non-code regions

System Calls Revisited

System Call Execution



User -> Kernel

- Problem: kernel space and user space are enforced by hardware
- Hardware must be informed of jump

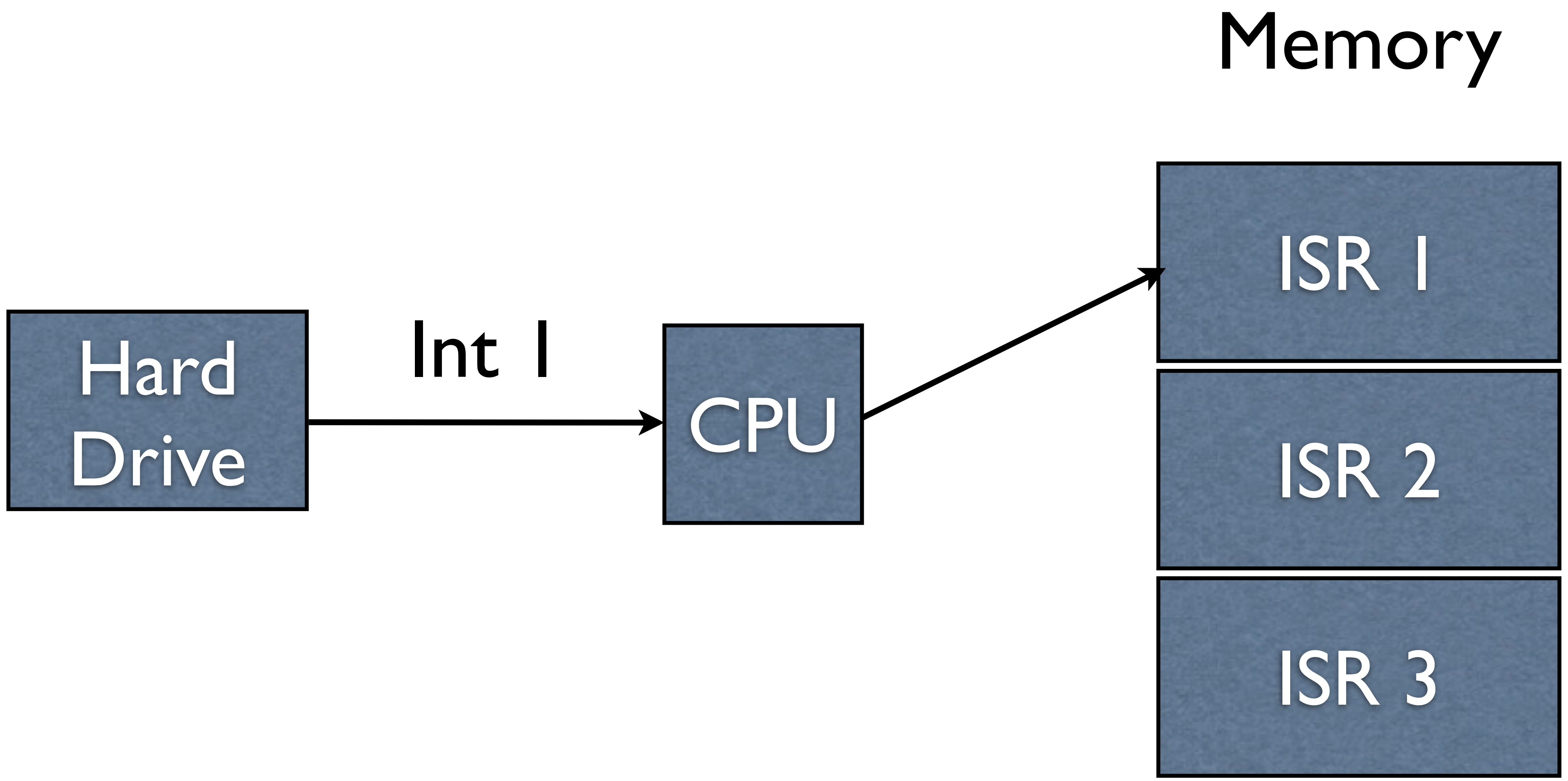
Solution?

- Instruction to specify the level
  - By necessity, it is privileged
  - Need kernel space to tell the system we're in kernel space - catch 22

Existing Machinery

- Interrupts are serviced by the kernel
  - Generated from other devices, often I/O
  - Preempt all else and enter the kernel
- The routines that service interrupts are called “interrupt service routines” - ISRs

Interrupts



Using Interrupts

- Trigger a "software interrupt"
  - Kernel mode entered synchronously
  - Parameters can be passed in registers, in a specific memory location, etc.
- Note that the actual mechanism and lingo is hardware dependent

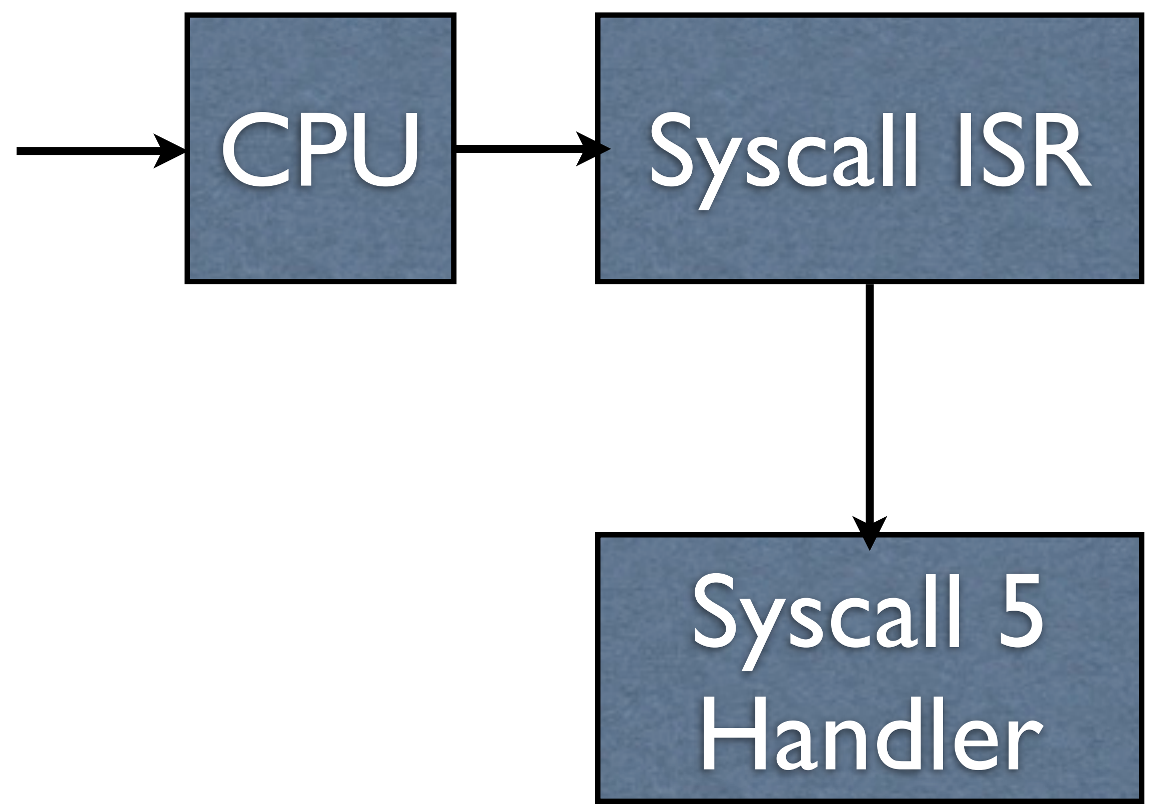
MIPS System Calls

- MIPS has the "syscall" instruction
- Processor throws a system call exception, triggering the OS' system call service routine
- By convention, the syscall ID is in `$v0`, and arguments are passed in `$a0` and `$a1`

MIPS System Calls

- Assume we want the system call with ID 5
- This call takes no arguments

```
addi $v0, $zero, 5
syscall
```



- code/userprog/exception.cc
- code/userprog/syscall.h
- code/test/start.s

Memory Management

Project #2 Memory

- Physical = virtual (until Project #3)
- Must using paging
- Need to allocate and free pages as requested

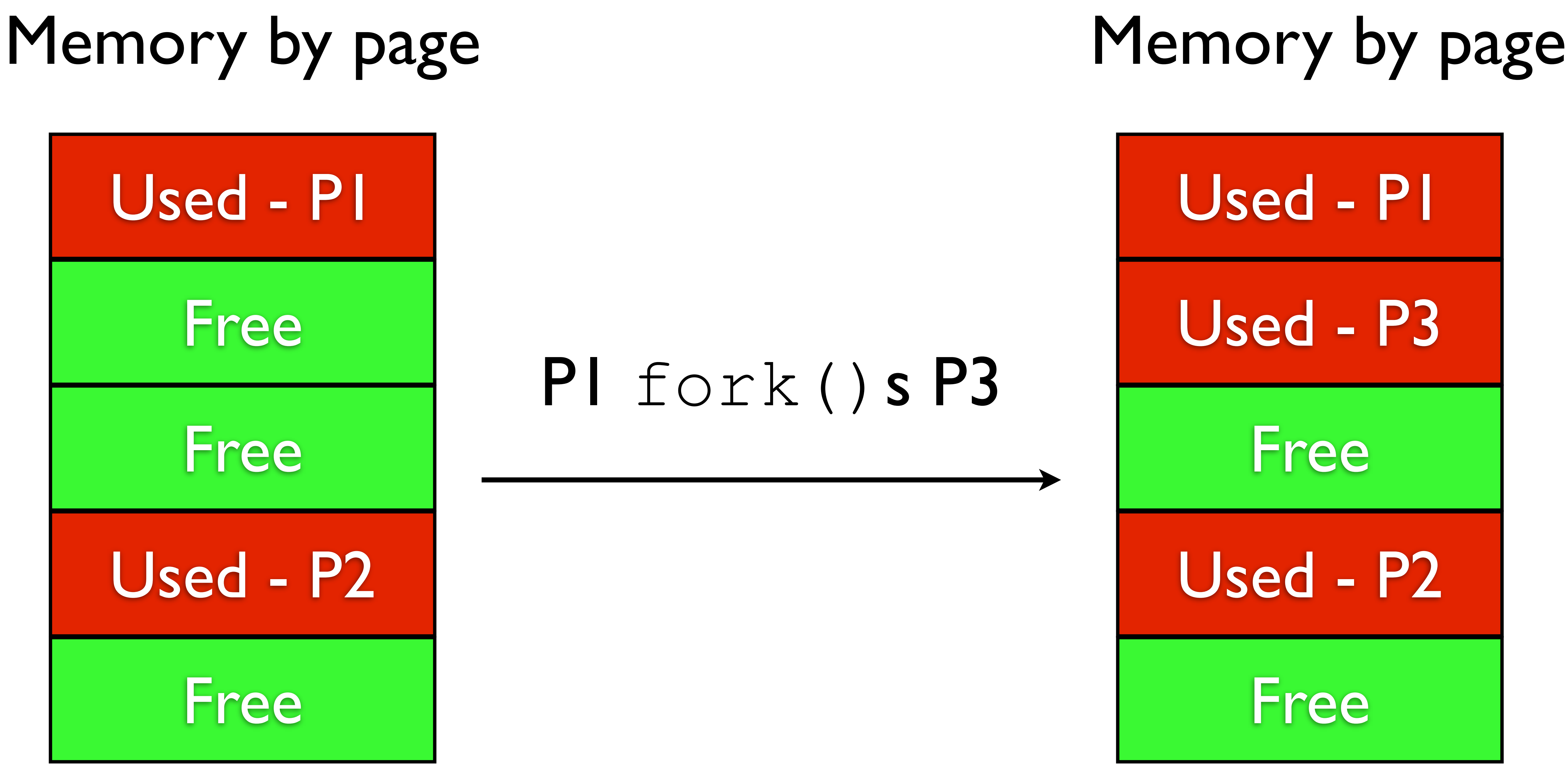
NACHOS Memory

- Does not have much
  - 128 byte pages
  - 32 pages total
  - 8 pages for each process' stack + data + code
- Simple bitmap is sufficient to record what is and is not used

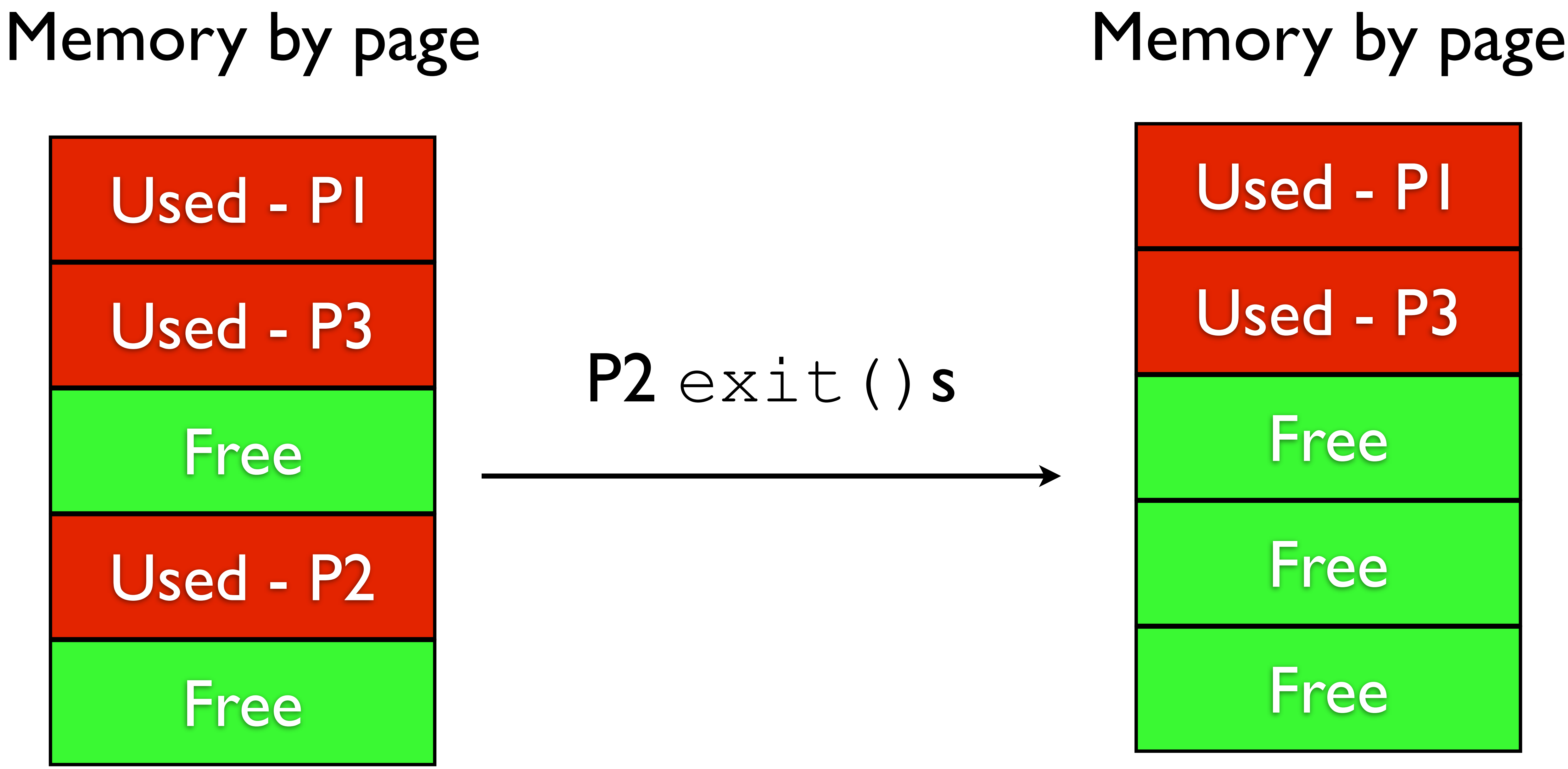
Contiguous Memory

- Since physical = virtual, served memory requests must be contiguous
  - I.e. if a process requests 5 pages, they must be contiguous
- *Could* do compaction, but this is a terrible idea

`Fork()` Example



`Exit()` Example



Getting Pages

- Memory is available through:
  - `machine->mainMemory`
  - Merely array of 1 byte characters
  - Need to split into pages on your own

Memory and Concurrency

- Multiple processes may request pages at the same time
- Only one may get any given page
- Synchronization primitives from Project #1 will have to be used
  - Make sure they work correctly!

I/O Syscalls

NACHOS Disk

- Do not need to worry about this until Project 3
- I/O syscalls for Project 2 utilize Linux's existing syscalls for file I/O directly

I/O Syscalls

- Actually implement `Read()` and `Write()`, NOT `readAt()` and `writeAt()`
- `readAt()` and `writeAt()`'s provided implementations are sufficient to implement `Read()` and `Write()`

Files and Concurrency

- Process A prints “Hello world!”
- Process B prints “Goodbye cruel world!”

```
Hello woGoodbye crld!
ruel world!
```

Files and Concurrency

- Determining what needs to be locked may be difficult
- May have separate things that need locking
  - May need multiple locks for distinct resources
  - Concurrent reads are OK, but not concurrent writes

Open File Semantics

- Semantics of `Fork()` are that child processes inherit open files
- `Read()` and `Write()` can only manipulate open files
- If a process will not close its files upon `Exit()`, then the OS must do so

Open Files

- Which files are opened must be recorded in the PCB
- This allows for all aforementioned behaviors
- Also allows for an offset for subsequent `Read()` and `Write()` requests

Console

- `Read()` and `Write()` may also manipulate the console
- Console is not opened or closed
- Constants specifying console usage are in `syscall.h`

Caveats

- The given code is really getting buggy
- Provided code is also getting really ugly

How-To Implement

- Project #2 has a step-by-step implementation guide at http://www.cs.ucsb.edu/~cs170/projects/homework_2guide.html
- Please read carefully