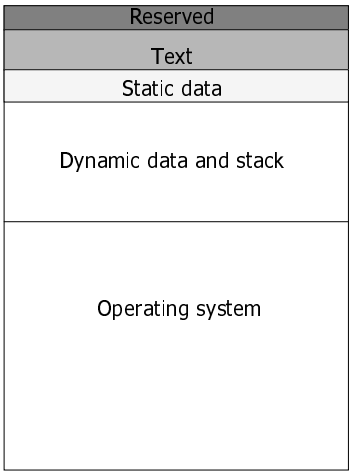
Lecture Topics: 11/3

- Address spaces
- Memory protection
- Creating a process
  - NT style
  - Unix style

Address Spaces

- An *address space* is the range of addresses that a process may use
- The *legal address space* is the range of addresses that a process may use *right now*

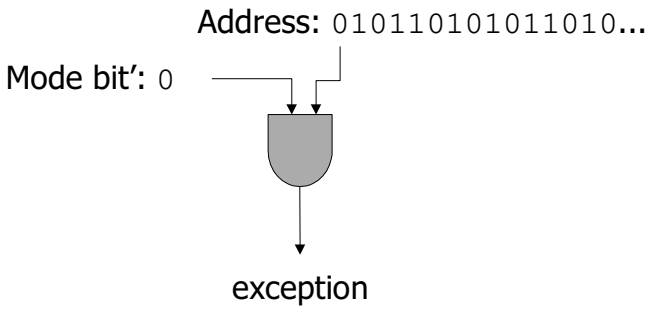


User and Kernel Memory

- When the mode bit is set to privileged, the kernel can see all of memory
  - user program, arguments, etc.
  - user memory is like a big data structure to the kernel
- When the mode bit is off, the user program can only see its own memory
  - the kernel's part of the address space is off limits

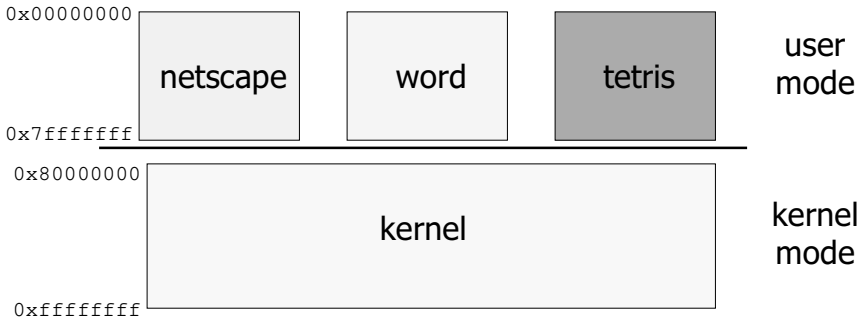
Enforcing Kernel Memory

- To enforce this policy, the hardware must check *every reference* to make sure it's OK



Multiple Address Spaces

- Nearly all modern operating systems support the abstraction of multiple address spaces



Processes and Address Spaces

- Each process has its own address space
- Process A's memory references (loads and stores) are interpreted within process A's address space
- Can Process A load a word from Process B's address space?
- Can Process A load a word from kernel memory?

Memory Protection

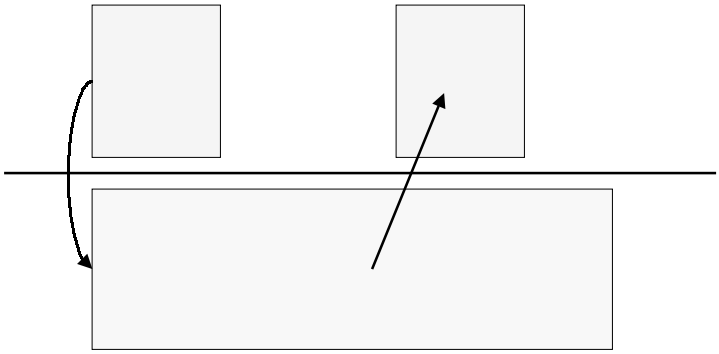
- We mean two things by memory protection:
  - user programs can't access kernel memory
  - user programs can't access each other's memory
- In the first case, we're using hardware support
- In the second case, we're using *naming*

Process Creation

- We'll see two approaches
- The NT way: the directed approach
  - The parent process issues a series of system calls, setting up the child
- The Unix way: fork/exec
  - The parent clones itself, and the child alters itself to do its task

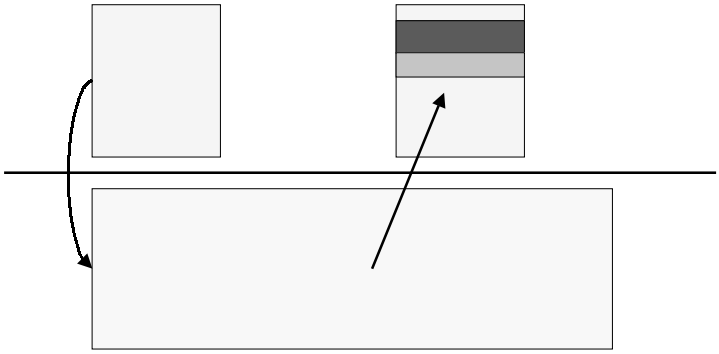
Directed Process Creation 1/3

- Step 1: Ask the OS to make a new address space



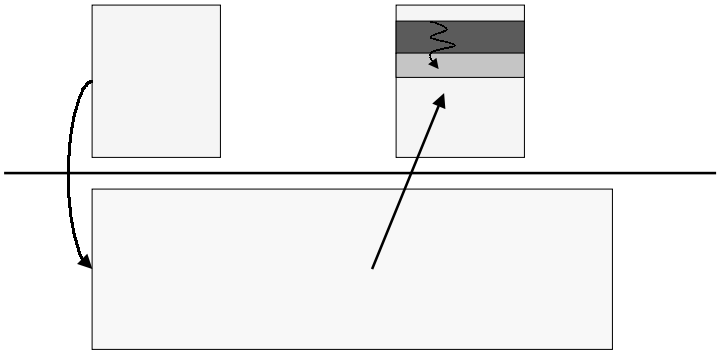
Directed Process Creation 2/3

- Step 2: Ask the OS to write the program image into the new address space



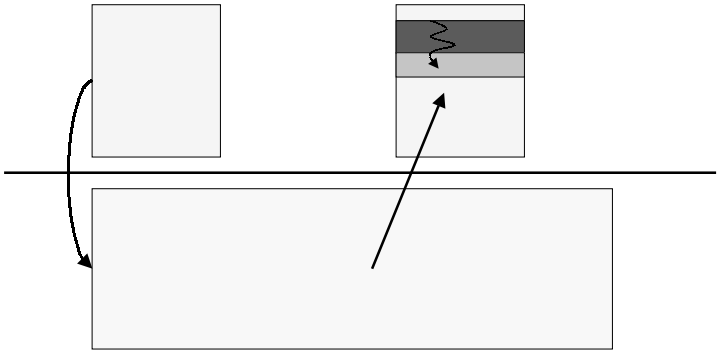
Directed Process Creation 3/3

- Step 3: Ask the OS to start the new process running at some procedure



Unix Process Creation

- Only one step: call `fork()`
- How does the OS know what program to run and where to start it?



Unix fork()

- Fork() is the Unix system call that creates a new process, but it doesn't take any arguments
- Instead, it makes an *exact duplicate* of the parent process. The only differences:
  - The PID (process ID number)
  - The return value of fork() itself

An Example Using fork()

```
main(int argc, char *argv[]) {
  char *myName = argv[1];
  int cpid = fork();
  if(cpid == 0) {
    printf("The child of %s is %d\n",
            myName, getpid());
    exit(0);
  } else {
    printf("My child is %d\n", cpid);
    exit(0);
}
```

Bizarre But Real

```
sequoia> gcc –o forkexample forkexample.c
sequoia> ./forkexample george
The child of george is 23874
My child is 23874
```

Even More Bizarre

```
sequoia> ./forkexample george
The child of george is 23874
My child is 23874
sequoia> ./forkexample george
My child is 24266
The child of george is 24266
```

Why do we get a different answer?

What good is fork()?

- By itself, not very useful
  - You might want to run an entirely different program some day!
- Another system call, exec(), completes the picture.
- Exec():
  - Replaces the program image of the parent with a new image
  - Starts executing at the beginning

Why is Unix Like That?

- The Unix assumption is that the child will have a lot in common with the parent
  - it will read the same files
  - the username of the owner is the same
  - it uses some of the same data, etc.
- The Unix approach starts with the parent and alters to suit
- The NT approach starts from scratch