Applications, Address Spaces, and Processes

Separating Units of Computation

Understanding Processes

- A Computer System has lots of nasty details in it.
  - **CPU, Memory, Stack, Address space, Privileged execution, Exceptions, System calls, I/O Control & Interrupts**
- A PROCESS is the operating system's ABSTRACTION for all of this junk.
- Programs execute using the PROCESS abstraction.
- Basis for
  - **memory management**
  - **sharing & isolation**
  - **concurrency**
  - **scheduling**

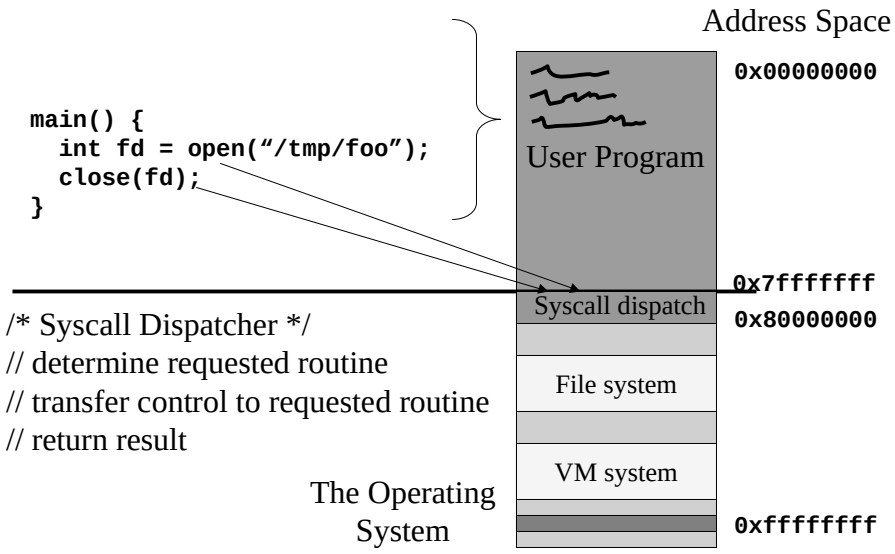
Definitions

- User mode
  - when the system is executing with the privileged bit off
- Kernel mode
  - when the system is executing with the privileged bit on
- Address space
  - the range of addresses available for a program to use
- Legal address space
  - the range of addresses that a program can use right now

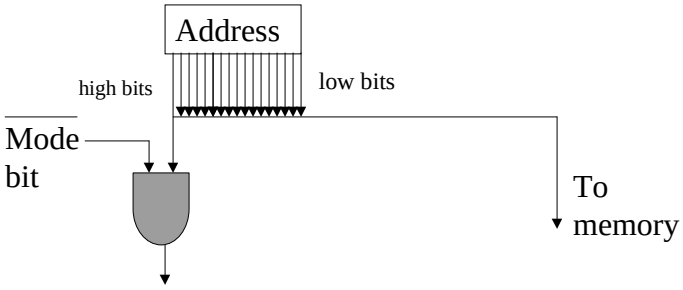
User and Kernel Memory

- 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 for the kernel
- But, when the mode bit is off, the user program can only see its own memory
  - the kernel's address space is OFF LIMITS
  - what happens if the user tries?
- Good for the OS, and good for the user program

OS/User Protection



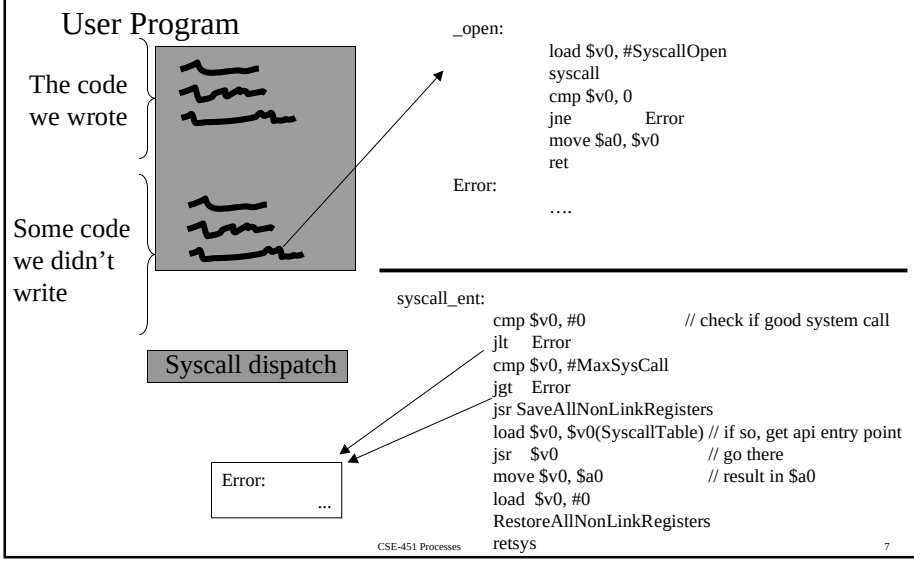
Privileged Memory Protection



Mode A	High Bit B	Fault? C
0	0	0
0	1	1
1	0	0
1	1	0

$C = \overline{A}B$

Inside the User Program



What Happens on Syscall?

- Automatic
  - Hardware MODE bit flips (go from nonpriv to priv)
  - Minimal save and restore of context
    - SP <- Kernel Syscall SP
    - PC <- Kernel Syscall PC
    - *SP++ <- User SP
    - *SP++ <- User PC
  - What happens on retsys?

And then we pick it up...

- Sycall handler checks to make sure we're asking for a good service
- Control is transferred to the service
- Result is passed back

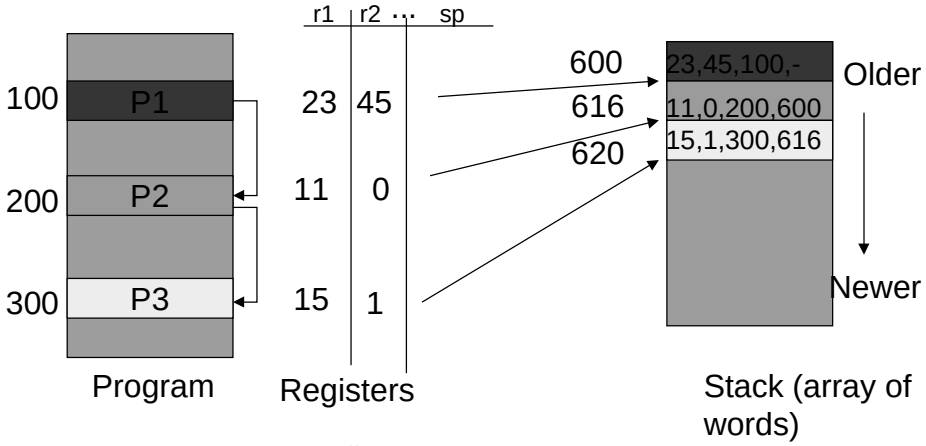
The Stack

- Push down record of execution state maintained as a series of frames

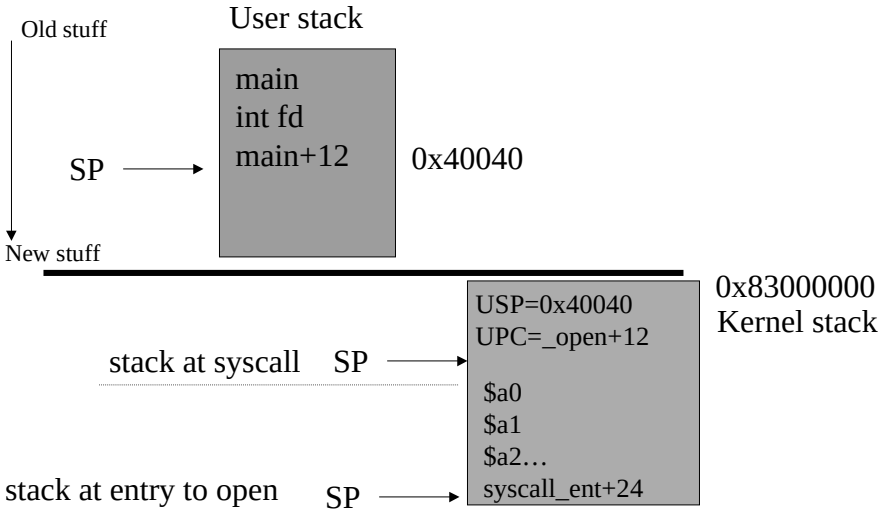
```
typedef struct StackFrame {
        int registers[NUM_REGS];
        int pc;
        StackFrame *prev;
};
```

The Stack

- Push down record of execution state



Understanding the Stack

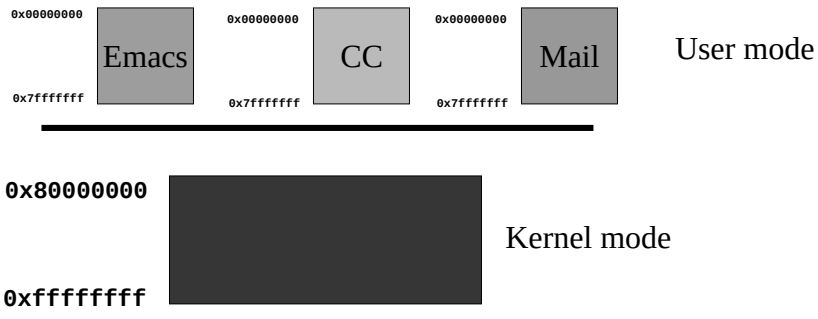


Concepts So Far

- User programs operate out of a different portion of the address space than the kernel
- There is a context switch that occurs every time we enter the kernel
- Inside the kernel we have expanded privileges
- A combination of hardware and software is responsible for this behavior

Multiple Address Spaces

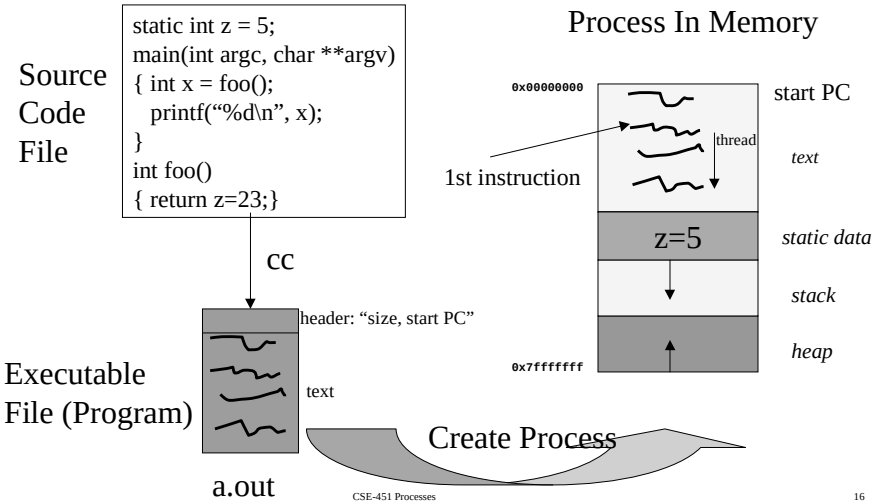
- Nearly all operating systems support the abstraction of multiple address spaces



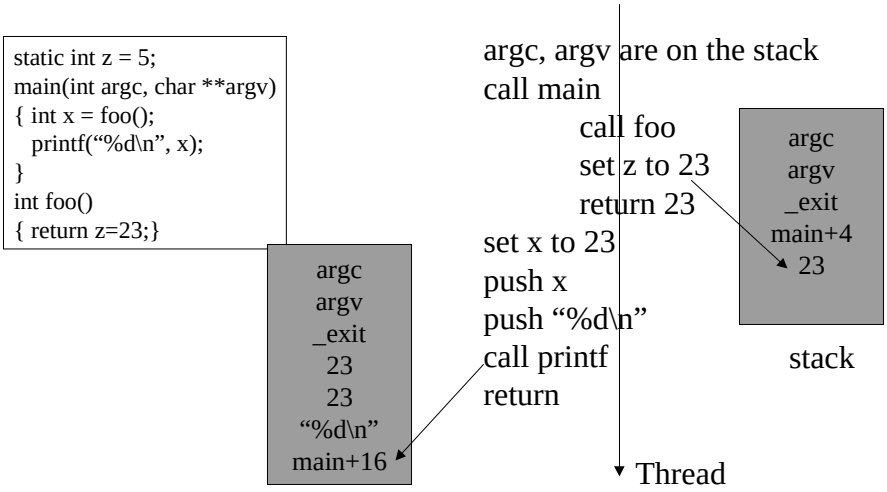
A Process

- Each address space contains a process
  - a bunch of text & data
  - a "thread" in execution
- A thread represents the flow of control that is active inside a program
  - deterministic change of state prescribed by the current state and the PC (which is actually part of the current state)

A Process is a Program in Execution



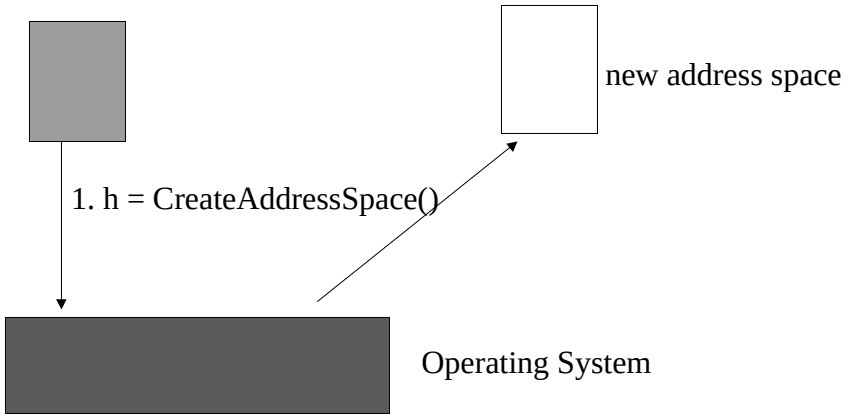
The Thread Of Control



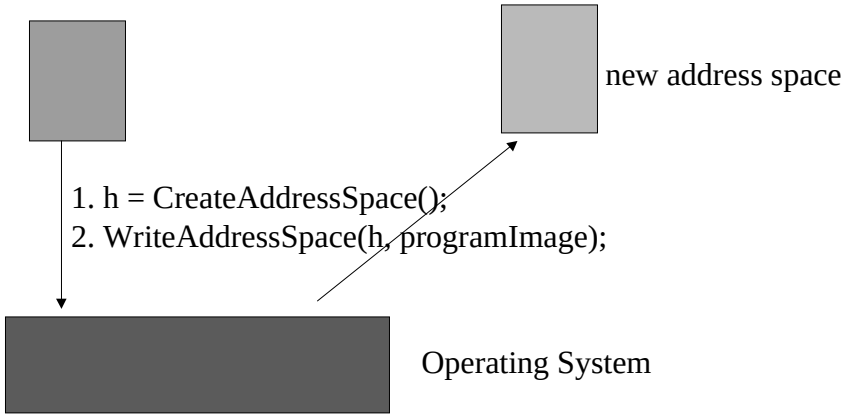
Where do Processes Come From?

- A process is an address space with some stuff in it and a thread of control
- All operating systems have facilities for creating new processes
- Some of them (eg, NT) are quite simple:
  - CreateAddressSpace, WriteAddressSpace, CreateThreadInAddressSpace, StartThread
- Others (eg, UNIX) are more subtle, but quite elegant

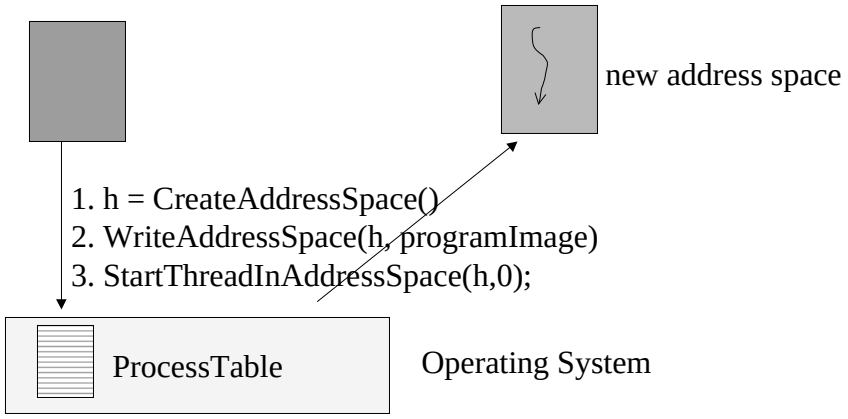
Process Creation: The Directed Approach



Process Creation: The Directed Approach (2)



Process Creation: The Directed Approach (3)



Tying it Together with a User Level Routine

```
spawn(char *programName);
{
        AddressSpace a;
        char *programImage;
        int fd = open(programName);
        programImage = (char*)malloc(filesize(fd));
        read(fd, programImage, filesize(fd));
        a = CreateAddressSpace();
        WriteAddressSpace(a, programImage, filesize(fd));
        StartThreadInAddressSpace(a, 0);
}
```

What's the significance of this being a user level routine?

Processes Under UNIX

- In Unix, the fork() system call is the only way to create a new process
  - attractive when new and old process share a lot (the original Unix model)
- int fork() does many things at once:
  - creates a new address space (called the child)
  - copies the parent's address space into the child's
  - starts a new thread of control in the child's address space
  - parent and child are equivalent -- almost
    - in parent, fork() returns a non-zero integer
    - in child, fork() returns a zero.
    - difference allows parent and child to distinguish
- int fork() returns TWICE!

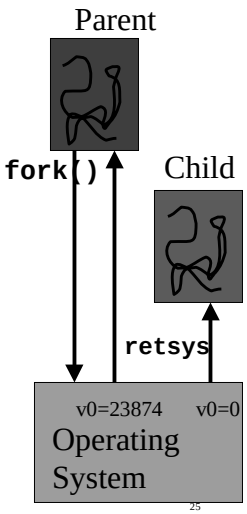
Example

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   int cpid = fork(); // cpid is a handle.
   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);
   }
}
```

What does this program print?

Bizarre But Real

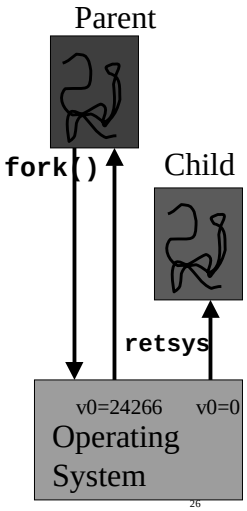
```
lace:tmp<15> cc a.c
lace:tmp<16> ./a.out foobar
The child of foobar is 23874
My child is 23874
```



Even More Bizarre

```
lace:tmp<15> cc a.c
lace:tmp<16> ./a.out foobar
The child of foobar is 23874
My child is 23874
lace:tmp<17> ./a.out foobar
My child is 24266
The child of foobar is 24266
lace:tmp<18>
```

Why do we get a different answer??



Fork is half the story

- Fork() gets us a new address space, but not one that's all that different.
  - parent and child share EVERYTHING
    - memory, operating system state
- int exec(char *programName) completes the picture
  - throws away the contents of the calling address space
  - replaces it with the program named by programName
  - starts executing at header.startPC

Starting a new program

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   char *progName = argv[2];

   int cpid = fork();
   if (cpid == 0) {
      printf("The child of %s is %d\n", myName, getpid());
      execl(progName,  // executable name
            progName, 0); // null terminated argv
      printf("OH NO. THEY LIED TO ME!!!\n");
 } else {
      printf("My child is %d\n", cpid);
      exit(0);
   }
}
```

HW 1

- Write a simple UNIX program to simulate the UNIX shell in a “read/fork/exec” loop
  - don’t bother with path searches. All commands can be fully qualified

```
CSE451Shell% /bin/cat /etc/motd
DEC OSF/1 V3.2 (Rev. 214); Thu Feb 22 08:48:40 PST 1996
DEC OSF/1 V3.2 Worksystem Software (Rev. 214)

This is an AFS fileserver.  Please run long running jobs (hours) or memory
intensive jobs elsewhere.

CSE451Shell% /bin/date
Sun Apr  5 22:51:50 PDT 1998
```