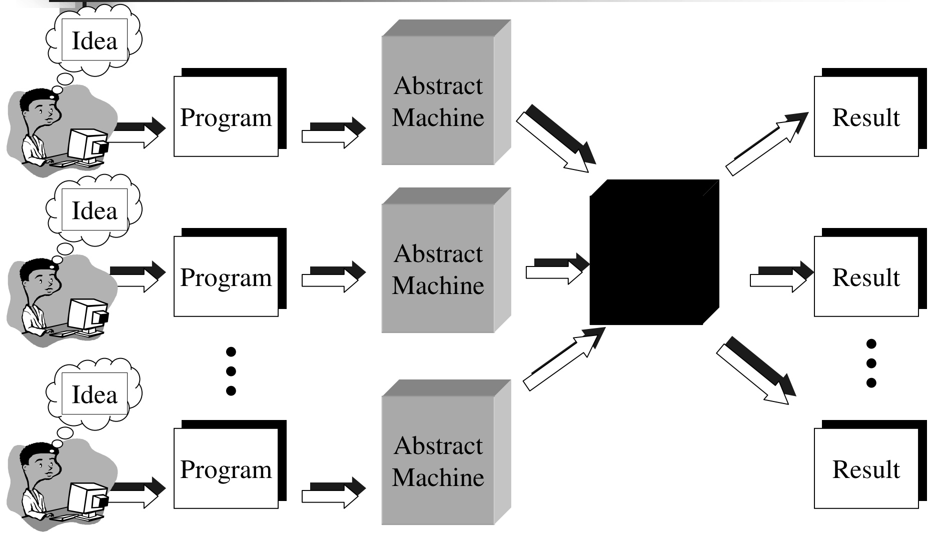
Chapter 2: Using the OS

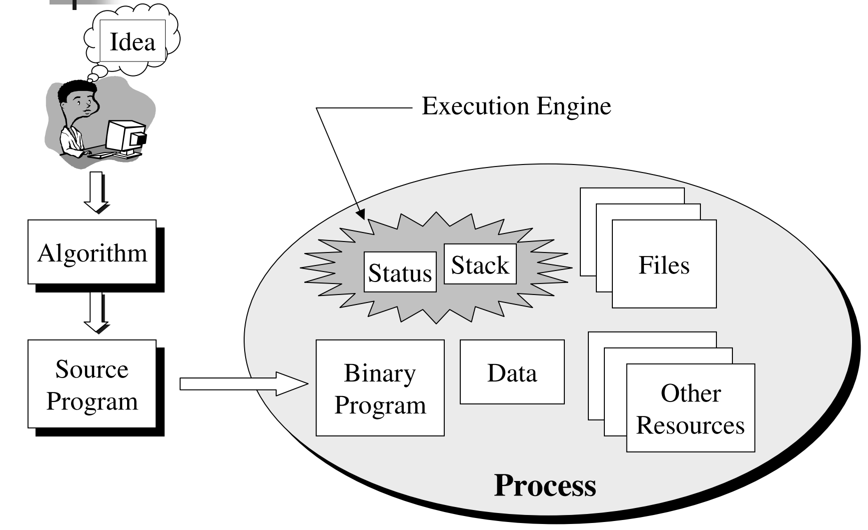
Basic abstraction



Abstract Machine Entities

- ***Process***: A sequential program in execution
- ***Resource***: Any abstract resource that a process can request, and which may can cause the process to be blocked if the resource is unavailable.
- ***File***: A special case of a resource. A linearly-addressed sequence of bytes. "A byte stream."

Algorithms, Programs, and Processes

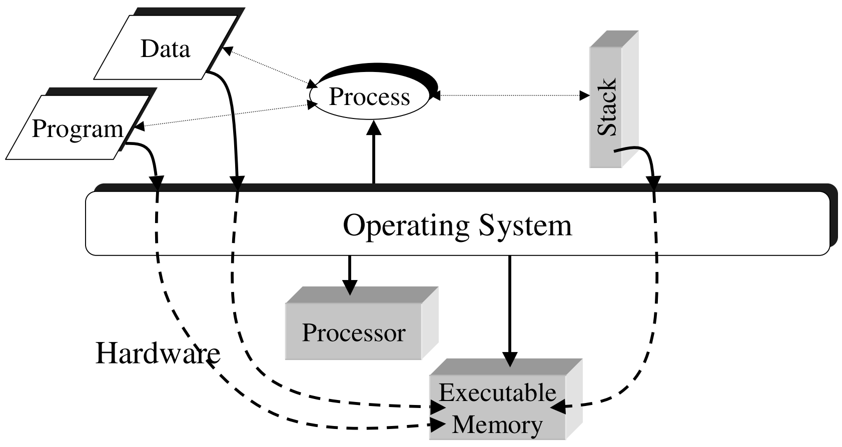


Classic Process

- OS implements {abstract machine} – one per task
- *Multiprogramming* enables N programs to be space-muxed in executable memory, and time-muxed across the physical machine processor.
- Result: Have an environment in which there can be multiple programs in execution *concurrently*,* each as a processes

* Concurrently: Programs appear to execute simultaneously

Process Abstraction

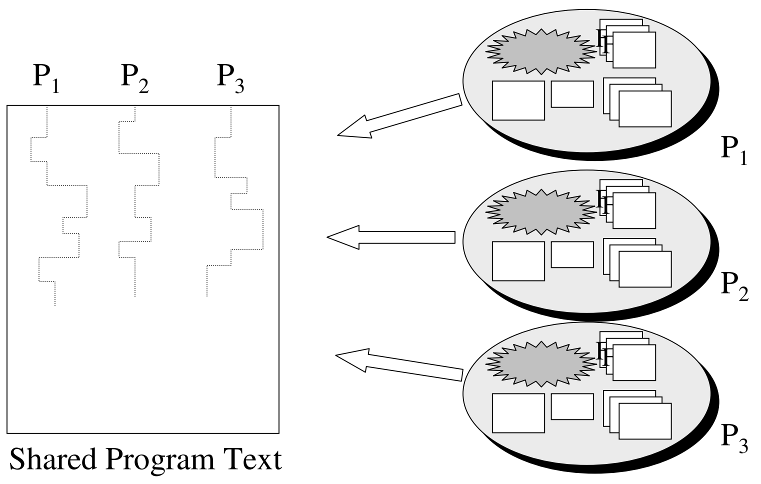


Example

```
int main() {
  int a;
  cin >> a;
  switch (a) {
     case 1: do_fun1(); break;
     case 2: do_fun2(); break;
     case 3: do_fun3(); break;
  }
}
```

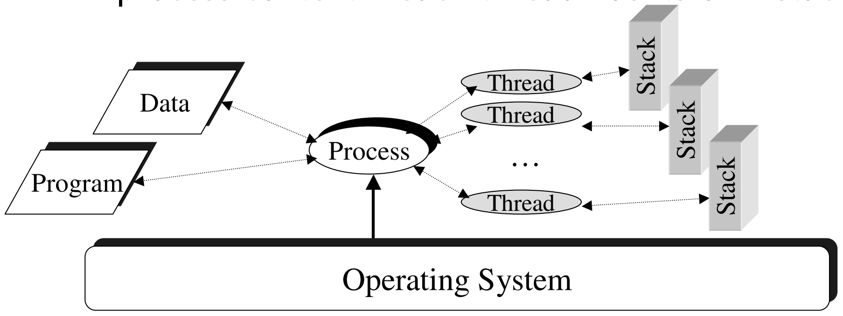
What happens if three users on an UNIX machine simultaneously run this program with different values of `a`?

Processes Sharing a Program

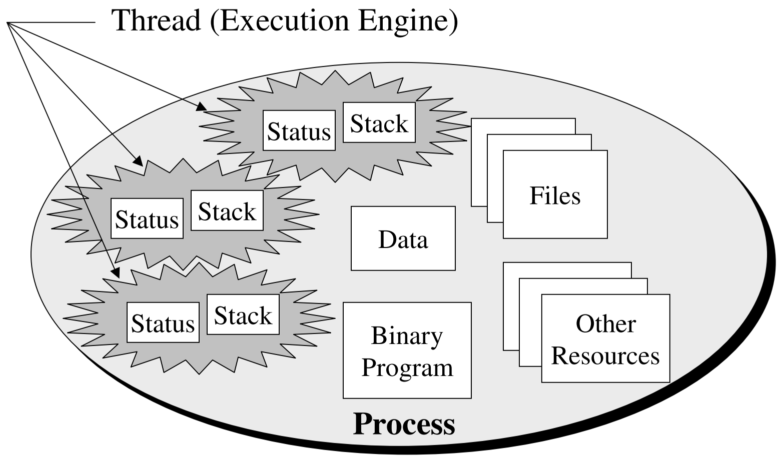


Modern Process & Thread

- Divide classic process:
  - *Process* is an infrastructure in which execution takes place – address space + resources
  - *Thread* is a program in execution within a process context – each thread has its own stack



A Process with Multiple Threads



More on Processes

- Abstraction of *processor* resource
  - Programmer sees an *abstract machine environment* with spectrum of resources and a set of resource addresses (most of the addresses are memory addresses)
  - User view is that its program is the only one in execution
  - OS perspective is that it runs one program with its resources for a while, then switches to a different process (*context switching*)
- OS maintains
  - A *process descriptor* data structure to implement the process abstraction
    - Identity, owner, things it owns/accesses, etc.
    - Tangible element of a process
  - Resource descriptors for each resource

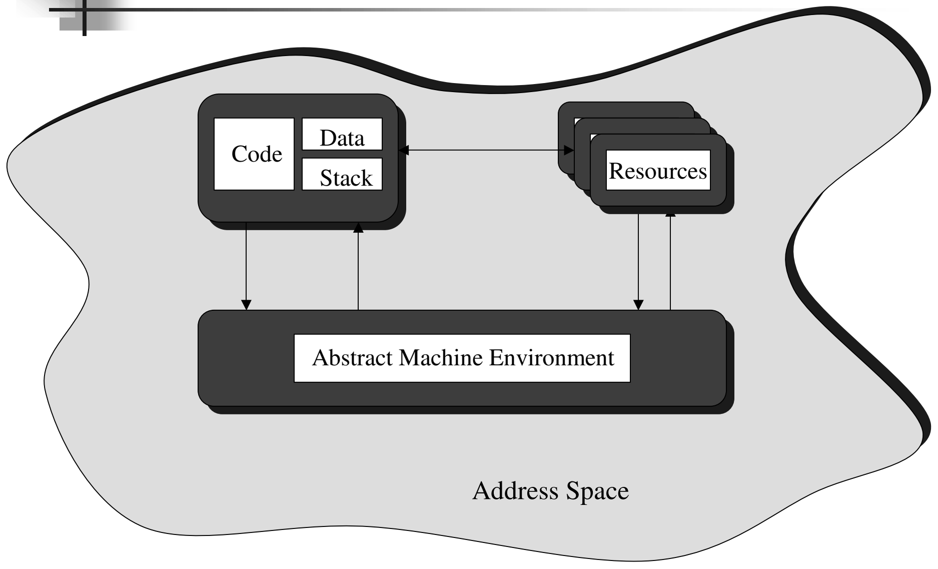
Address Space

- Process must be able to reference every resource in its abstract machine
- Assign each unit of resource an address
  - Most addresses are for memory locations
  - Abstract device registers
  - Mechanisms to manipulate resources
- Addresses used by one process are inaccessible to other processes
- Say that each process has its own *address space*

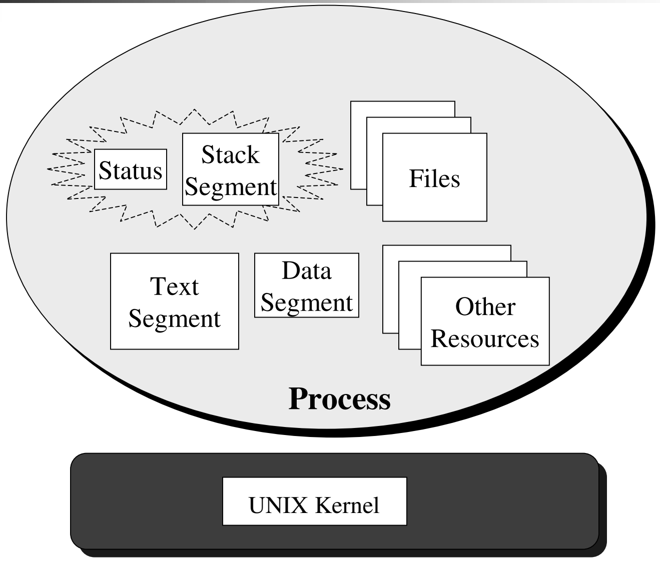
Shared Address Space

- Classic processes sharing program ⇒ shared address space support
- Thread model simplifies the problem
  - All threads in a process implicitly use that process's address space , but no "unrelated threads" have access to the address space
  - Now trivial for threads to share a program and data
    - If you want sharing, encode your work as threads in a process
    - If you do not want sharing, place threads in separate processes

Process & Address Space



UNIX Processes



UNIX Processes

- Each process has its own address space
  - Subdivided into text, data, & stack segment
  - `a.out` file describes the address space
- OS kernel creates *descriptor* to manage process
- *Process identifier* (PID): User handle for the process (descriptor)
- Try "`ps`" and "`ps -aux`" (read man page)

Creating/Destroying Processes

- UNIX `fork()` creates a process
  - Creates a new address space
  - Copies text, data, & stack into new adress space
  - Provides child with access to open files
- UNIX `wait()` allows a parent to wait for a child to terminate
- UNIX execα() allows a child to run a new program

Creating a UNIX Process

```
int pidValue;
 ...
pidValue = fork();       /* Creates a child process */
if(pidValue == 0) {
  /* pidValue is 0 for child, nonzero for parent */
  /* The child executes this code concurrently with parent */
    childsPlay(…);      /* A procedure linked into a.out */
    exit(0);
}
/* The parent executes this code concurrently with child */
parentsWork(..);
wait(…);
 ...
```

Child Executes a different Program

```
int pid;
 ...
/* Set up the argv array for the child */
 ...
/* Create the child */
if((pid = fork()) == 0) {
  /* The child executes its own absolute program */
    execve(childProgram.out, argv, 0);
  /* Only return from an execve call if it fails */
    printf("Error in the exec … terminating the child …");
    exit(0);
}
 ...
wait(…);    /* Parent waits for child to terminate */
 ...
```

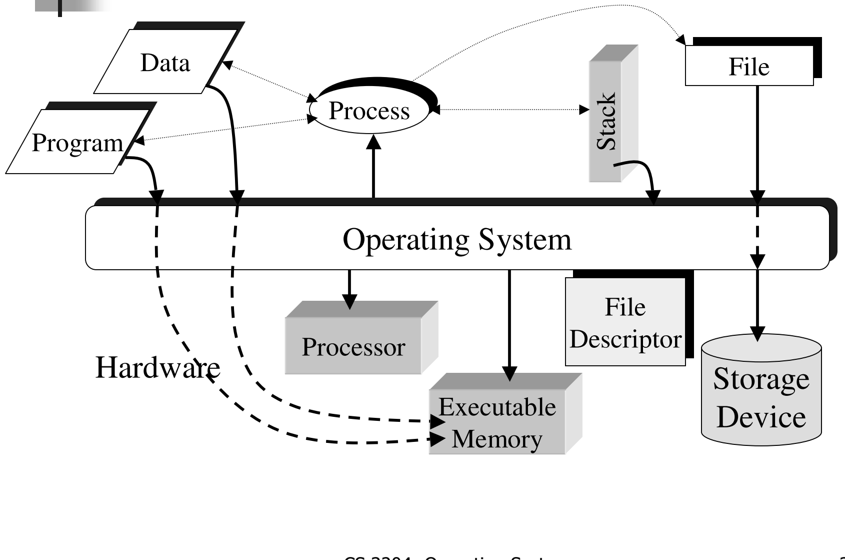
Example: Parent

```
#include        <sys/wait.h>

#define NULL    0

int main (void)
{
    if (fork() == 0){   /* This is the child process */
        execve("child",NULL,NULL);
        exit(0);        /* Should never get here, terminate */
    }
/* Parent code here */
    printf("Process[%d]: Parent in execution ...\n", getpid());
    sleep(2);
    if(wait(NULL) > 0) /* Child terminating */
        printf("Process[%d]: Parent detects terminating child \n",
                            getpid());
    printf("Process[%d]: Parent terminating ...\n", getpid());
}
```

Example: Child

```
int main (void)
{
/* The child process's new program
   This program replaces the parent's program */

    printf("Process[%d]: child in execution ...\n", getpid());
    sleep(1);
    printf("Process[%d]: child terminating ...\n", getpid());
}
```

The File Abstraction



UNIX Files

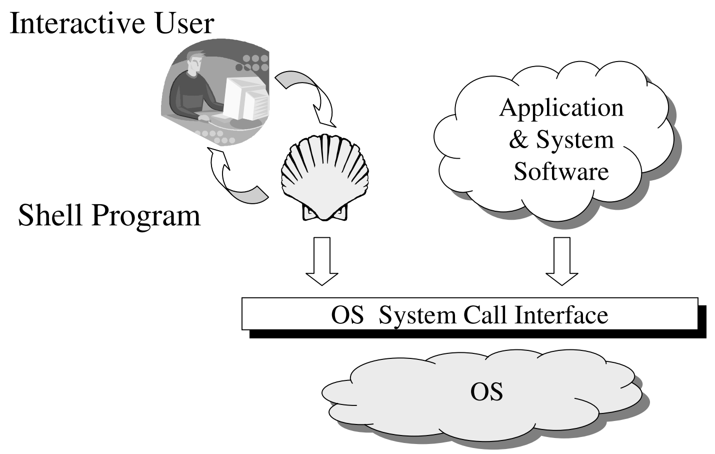
- UNIX and NT try to make every resource (except CPU and RAM) look like a file
- Then can use a common interface:

`open`	Specifies file name to be used
`close`	Release file descriptor
`read`	Input a block of information
`write`	Output a block of information
`lseek`	Position file for read/write
`ioctl`	Device-specific operations

UNIX File Example

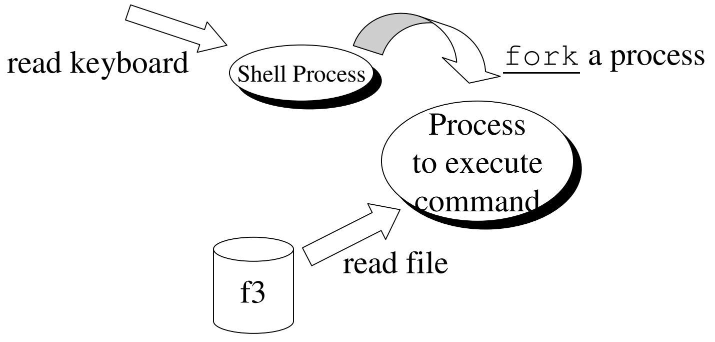
```
#include    <stdio.h>
#include    <fcntl.h>
int main() {
    int inFile, outFile;
    char *inFileName = “in_test”;
    char *outFileName = “out_test”;
    int len;
    char c;

    inFile = open(inFileName, O_RDONLY);
    outFile = open(outFileName, O_WRONLY);
/* Loop through the input file */
    while ((len = read(inFile, &c, 1)) > 0)
        write(outFile, &c, 1);
/* Close files and quite */
    close(inFile);
    close(outFile);
}
```

Shell Command Line Interpreter



The Shell Strategy

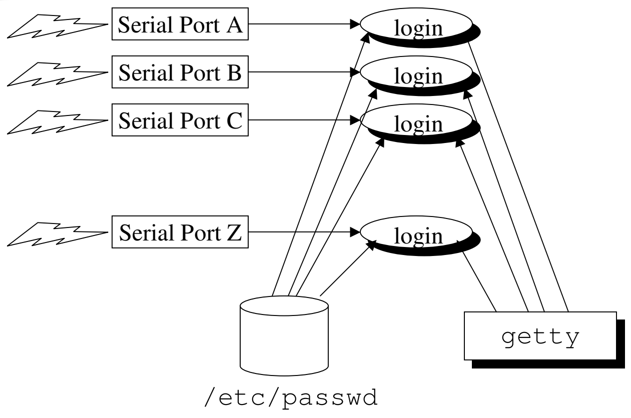
```
% grep first f3
```



Bootstrapping

- Computer starts, begins executing a *bootstrap program* -- *initial process*
- Loads OS from the disk (or other device)
- Initial process runs OS, creates other processes

Initializing a UNIX Machine



Objects

- A recent trend is to replace processes by objects
- Objects are autonomous
- Objects communicate with one another using messages
- Popular computing paradigm
- Too early to say how important it will be ...