

Processes

CS 416: Operating Systems Design

Department of Computer Science

Rutgers University

<http://www.cs.rutgers.edu/~vinodg/teaching/416/>

Von Neuman Model

Both text (program) and data reside in memory

Execution cycle

Fetch instruction

Decode instruction

Execute instruction

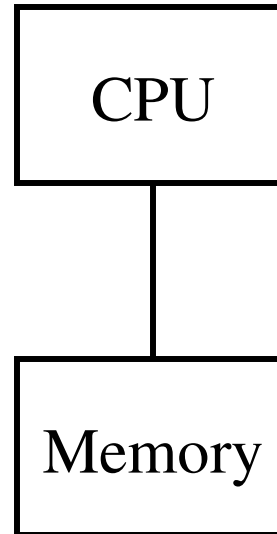
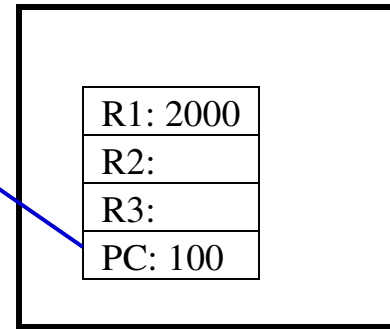


Image of Executing Program

100 load R1, R2
104 add R1, 4, R1
108 load R1, R3
112 add R2, R3, R3
...
2000 4
2004 8



CPU

Memory

How Do We Write Programs Now?

```
public class foo {  
  
    static private int yv = 0;  
    static private int nv = 0;  
  
    public static void main() {  
        foo foo_obj = new foo;  
        foo_obj->cheat();  
    }  
  
    public cheat() {  
        int tyv = yv;  
        yv = yv + 1;  
        if (tyv < 10) {  
            cheat();  
        }  
    }  
}
```

How to map a program like
this to a Von Neuman
machine?

Where to keep yv, nv?

What about foo_obj and tyv?

How to do foo->cheat()?

Global Variables

Dealing with “global” variables like *yv* and *nv* is easy

Let's just allocate some space in memory for them

This is done by the compiler at *compile time*

A reference to *yv* is then just an access to *yv*'s location in memory

Suppose *yv* is stored at location 2000

Then, $yv = yv + 1$ might be compiled to something like

```
loadi    2000, R1
load     R1, R2
add      R2, 1, R2
store    R1, R2
```

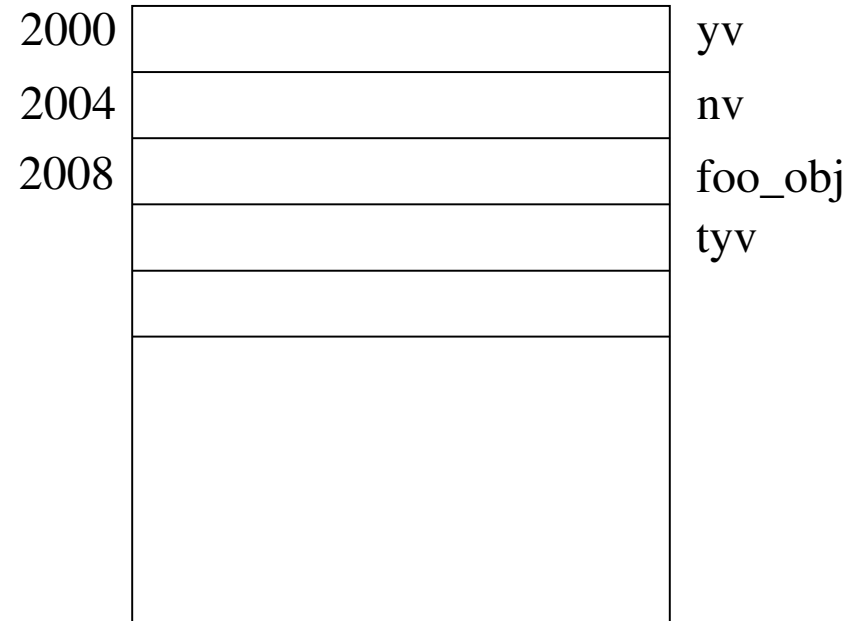
Local Variables

What about `foo_obj` defined in `main()` and `tyv` defined in `cheat()`?

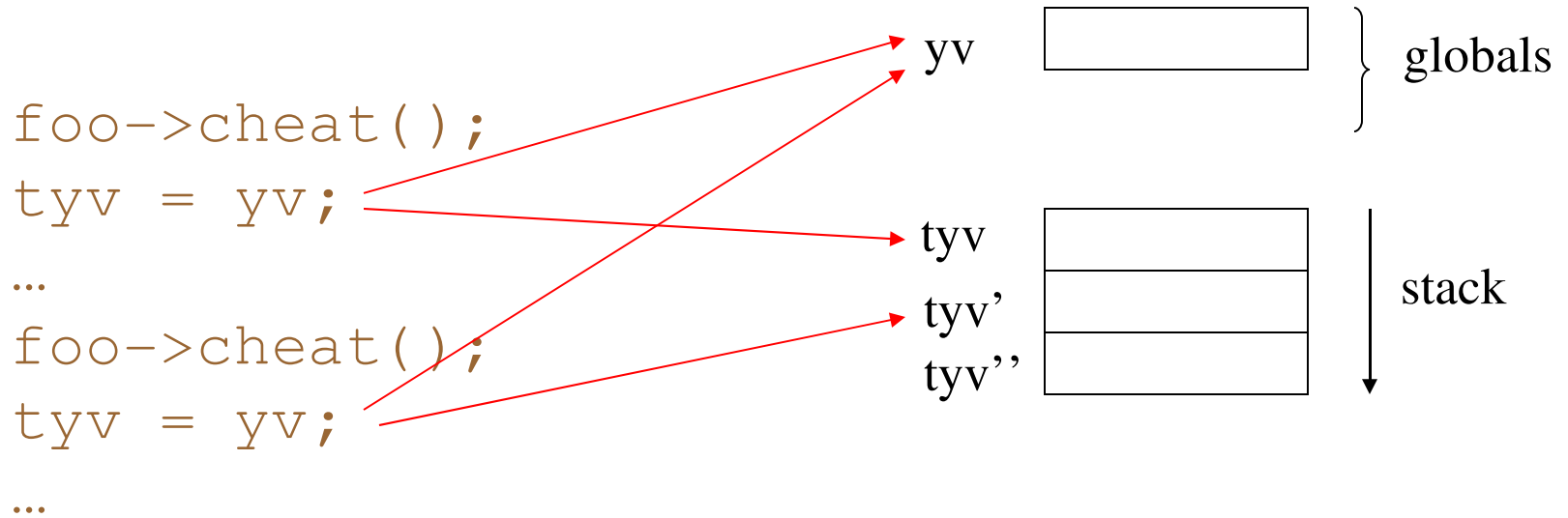
1st option you might think of is just to allocate some space in memory for these variables as well (as shown to the right)

What is the problem with this approach?

How can we deal with this problem?



Local Variable



Allocate a new memory location to tyv every time cheat() is called at run-time

Convention is to allocate storage in a stack (often called the control stack)

Pop stack when returning from a method: storage is no longer needed

Code for allocating/deallocating space on the stack is generated by compiler at compile time

What About “new” Objects?

```
foo foo_obj = new foo;
```

foo_obj is really a pointer to a foo object

As just explained, a memory location is allocated for foo_obj from the stack whenever main() is invoked

Where does the object created by the “new foo” actually live?

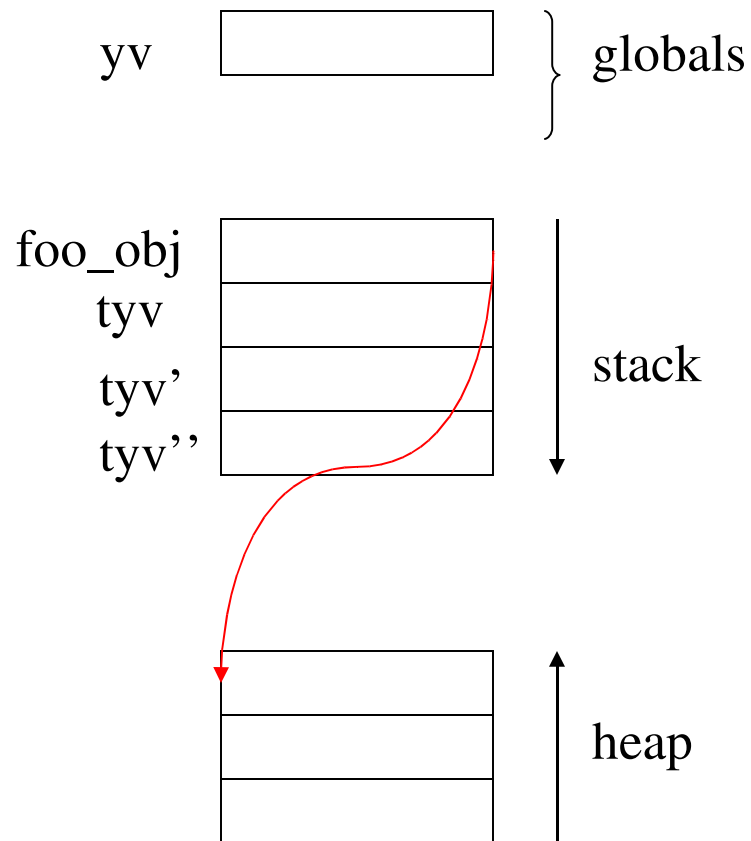
Is the stack an appropriate place to keep this object?

Why not?

Memory Image

Suppose we have executed the following:

```
yv = 0
nv = 0
main()
foo_obj = new foo
foo->cheat()
tyv = yv
yv = yv + 1
foo->cheat()
tyv = yv
yv = yv + 1
foo->cheat()
tyv = yv
yv = yv + 1
```



Data Access

How to find data allocated dynamically on stack?

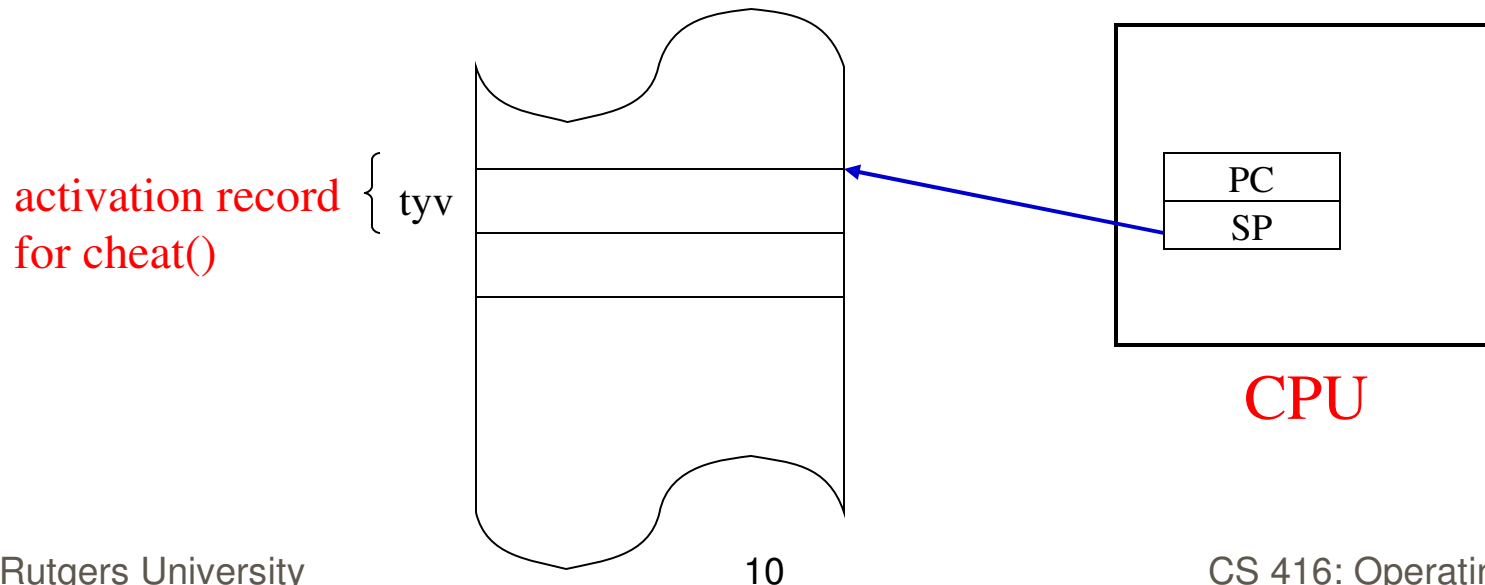
By convention, designate one register as the stack pointer

Stack pointer always points to current **activation record**

Stack pointer is set at entry to a method

Code for setting stack pointer is generated by compiler

Local variables and parameters are referenced as offsets from sp



Data Access

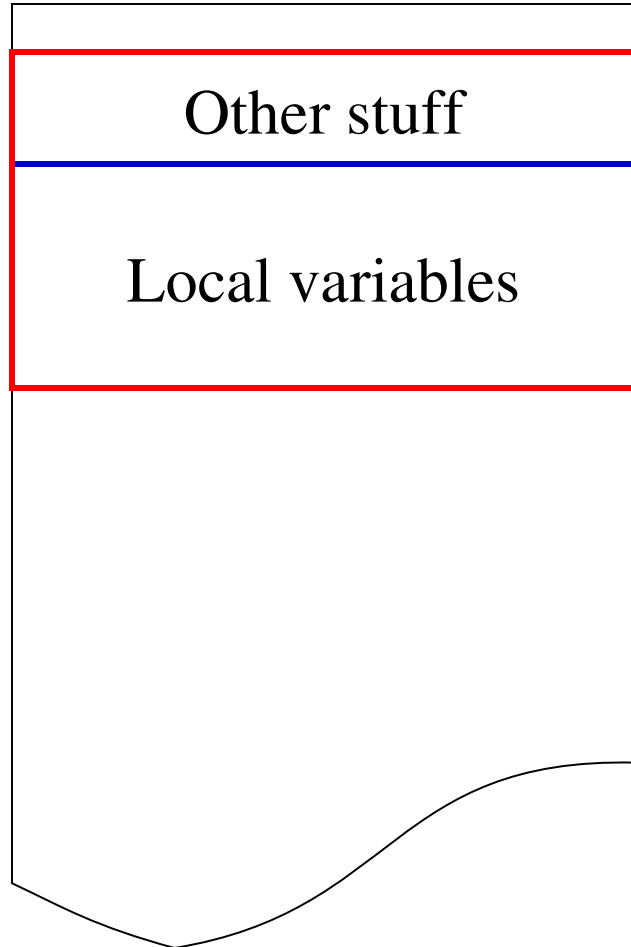
The statement

$tyv = tyv + 1$

Would then translate into something like

addi	0, sp, R1	# tyv is the only variable so offset is 0
load	R1, R2	
addi	1, R2	
store	R1, R2	

Activation Record



We have only talked about allocation of local variables on the stack

The activation record is also used to store:

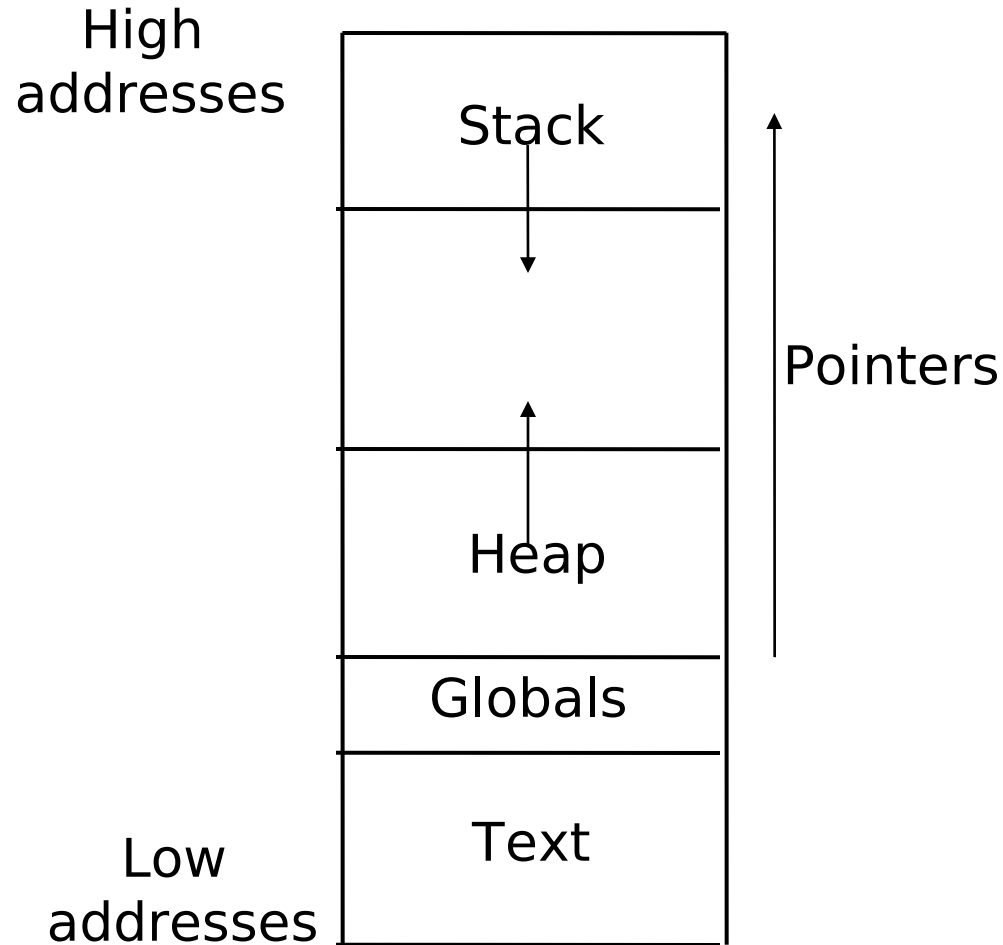
- Parameters

- The beginning of the previous activation record

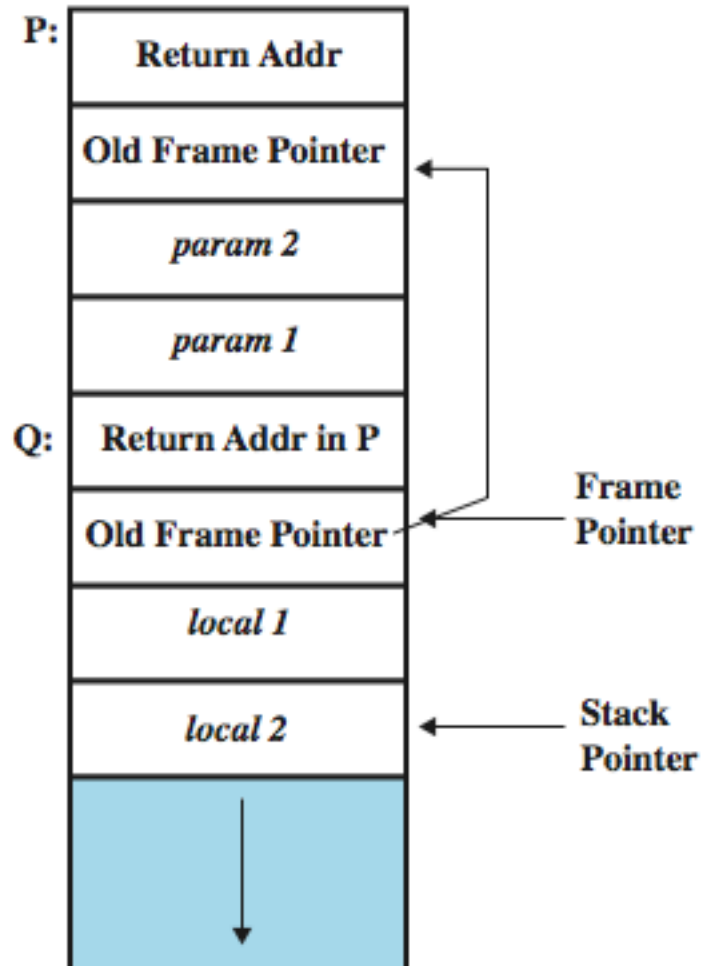
- The return address

- ...

Process memory layout



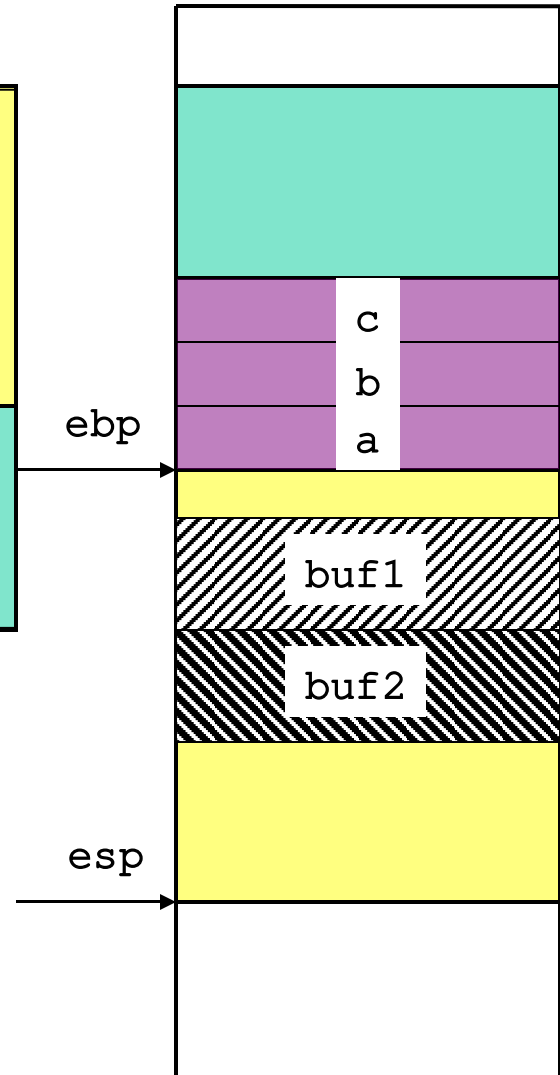
Function Calls and Stack Frames



The stack

```
void function(int a, int b, int c ) {
    char buf1[5];
    char buf2[10];
    ...
}

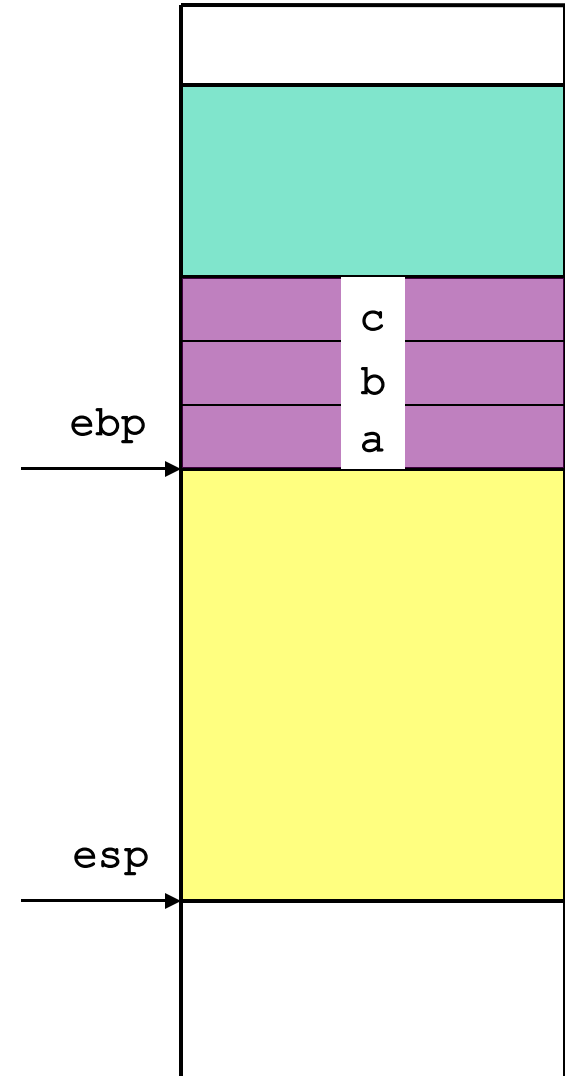
void main() {
    function(1, 2, 3);
}
```



The stack

```
void main() {  
    function(1, 2, 3);  
}
```

```
pushl $3  
pushl $2  
pushl $1  
call function
```

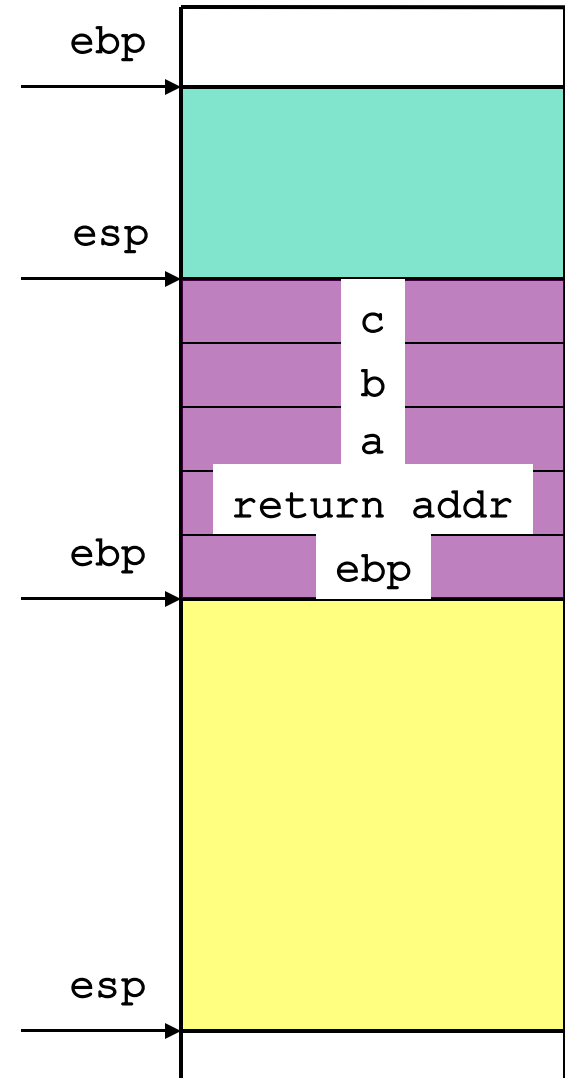


A function call

```
void main() {  
    function(1, 2, 3);  
}
```

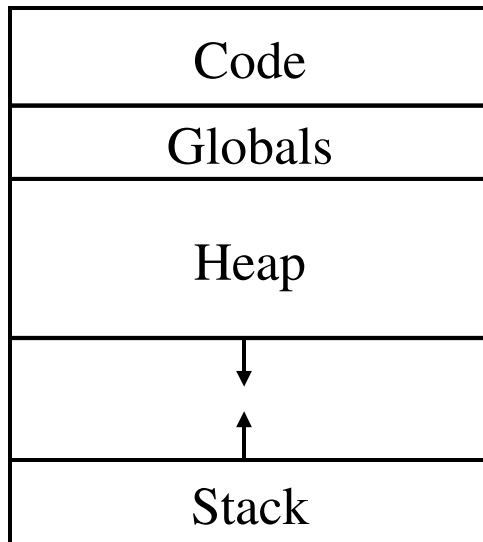
```
pushl $3  
pushl $2  
pushl $1  
call function
```

```
pushl %ebp  
movl %esp, %ebp  
subl $20, %esp
```



Run Time Storage Organization

Low addresses



High addresses

Virtual Memory

Each variable must be assigned a storage class

Global (static) variables

Allocated in globals region at compile-time

Method local variables and parameters

Allocate dynamically on stack

Dynamically created objects (using new)

Allocate from heap

Objects live beyond invocation of a method

Garbage collected when no longer “live”

Why Did We Talk About All That Stuff?

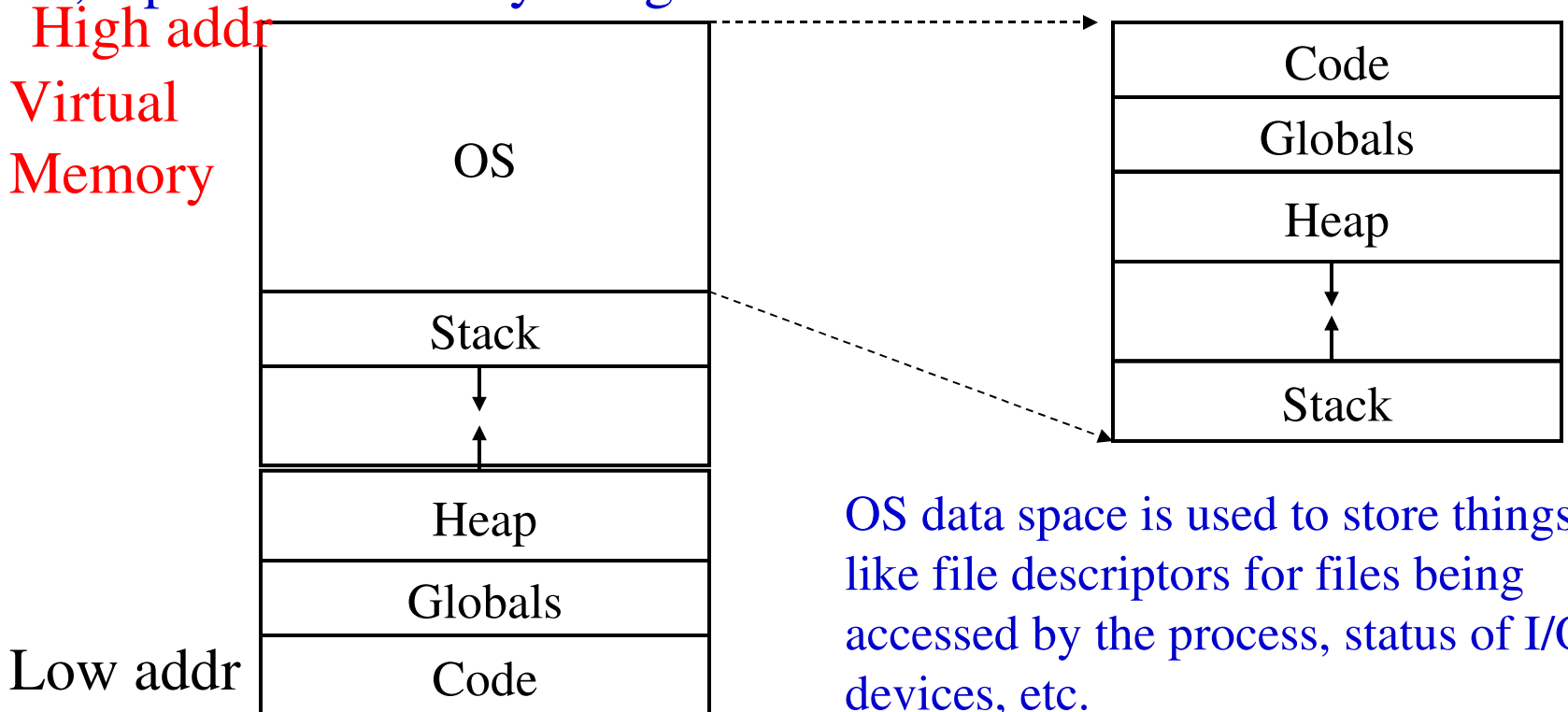
Process = system abstraction for the set of resources required for executing a program
= a running instance of a program
= memory image + registers' content (+ I/O state)

The stack + registers' content represent the *execution context* or *thread of control*

What About The OS?

Recall that one of the functions of an OS is to provide a virtual machine interface that makes programming the machine easier

So, a process memory image must also contain the OS



Copying data to/from the kernel/user

```
int x;
```

```
// Copy the value of p from user-space and set the value of x.
```

```
void sys_setint (int *p) { memcpy (&x, p, sizeof(x)); }
```

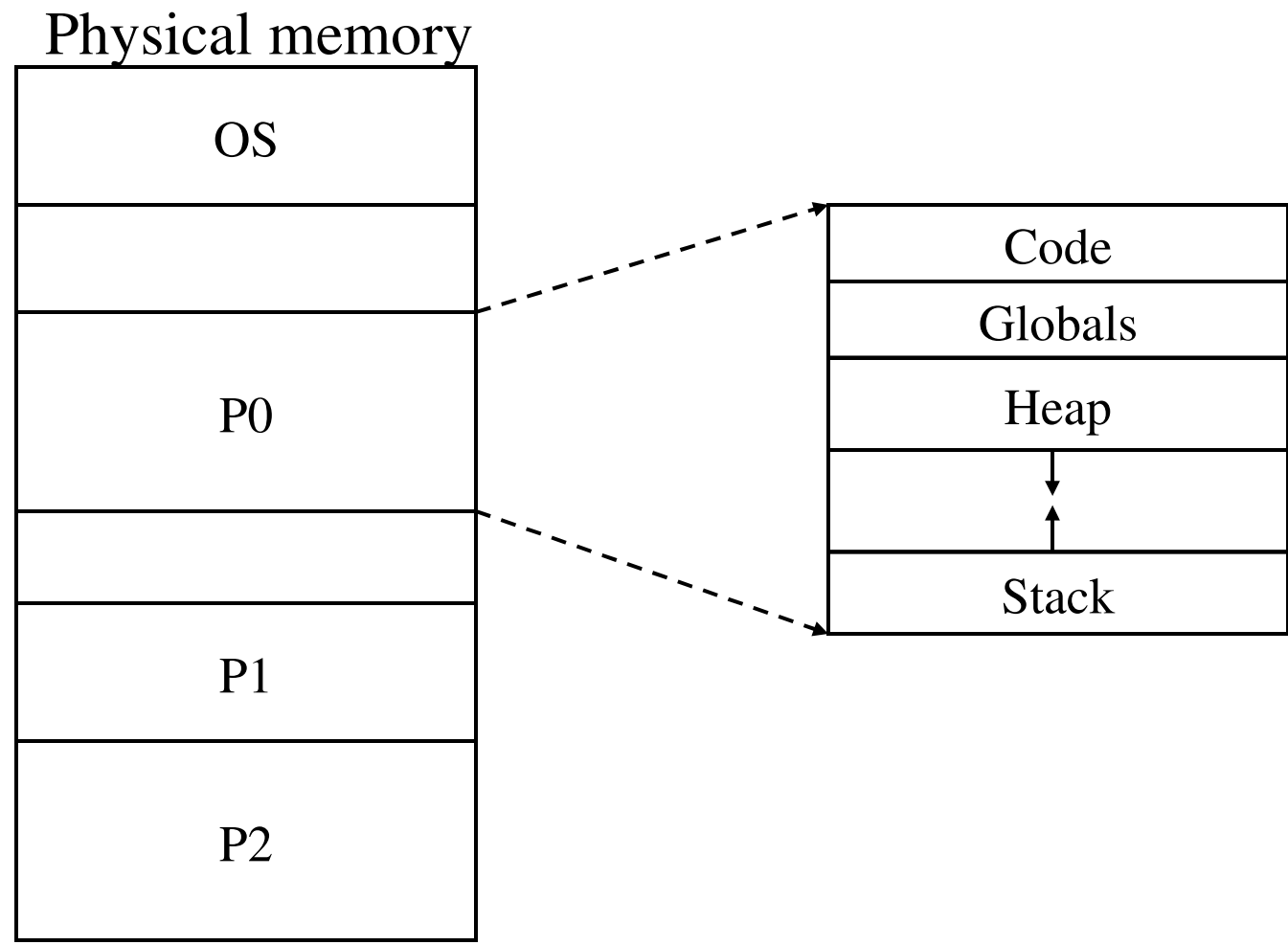
Wrong!

```
int x;
```

```
void sys_setint (int *p) { copy_from_user(&x, p, sizeof(x)); }
```

Correct!

What Happens When There is More Than One Running Process?



Process Control Block

Each process has per-process state maintained by the OS

Identification: process, parent process, user, group, etc.

Execution contexts: threads

Address space: virtual memory

I/O state: file handles (file system), communication endpoints (network), etc.

Accounting information

For each process, this state is maintained in a *process control block (PCB)*

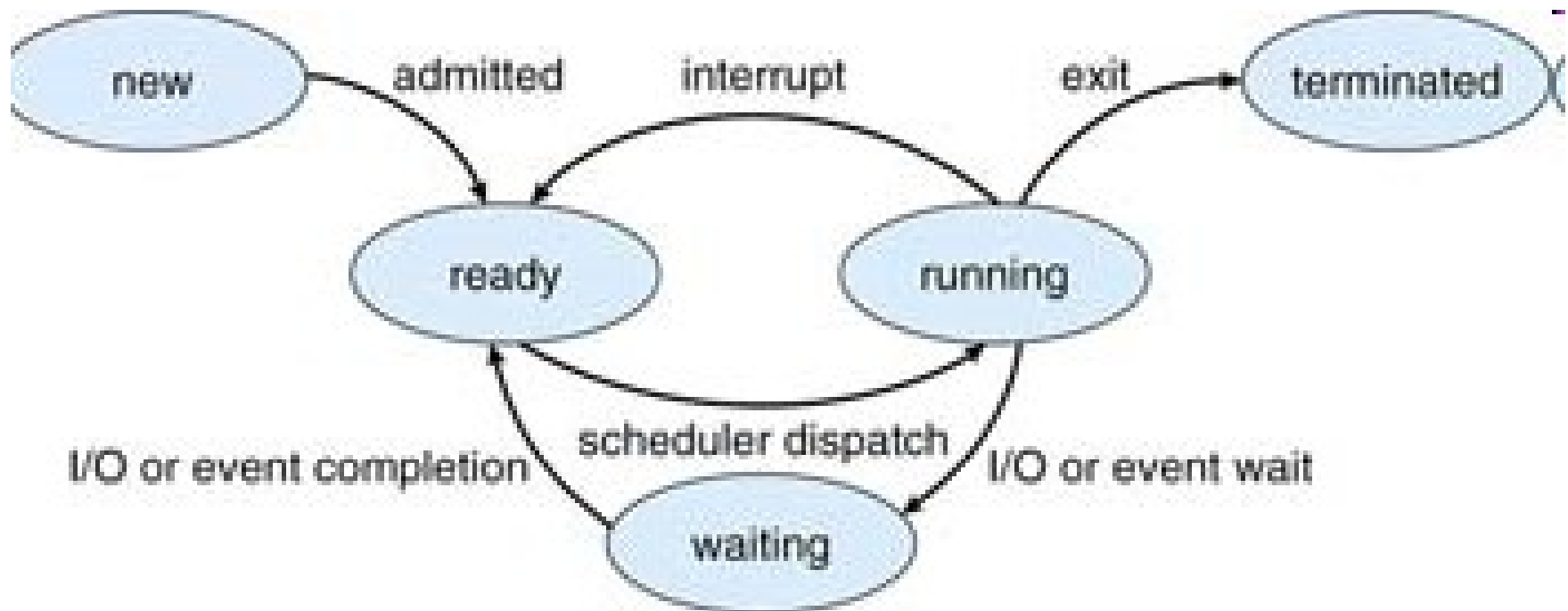
This is just data in the OS data space

Think of it as objects of a class

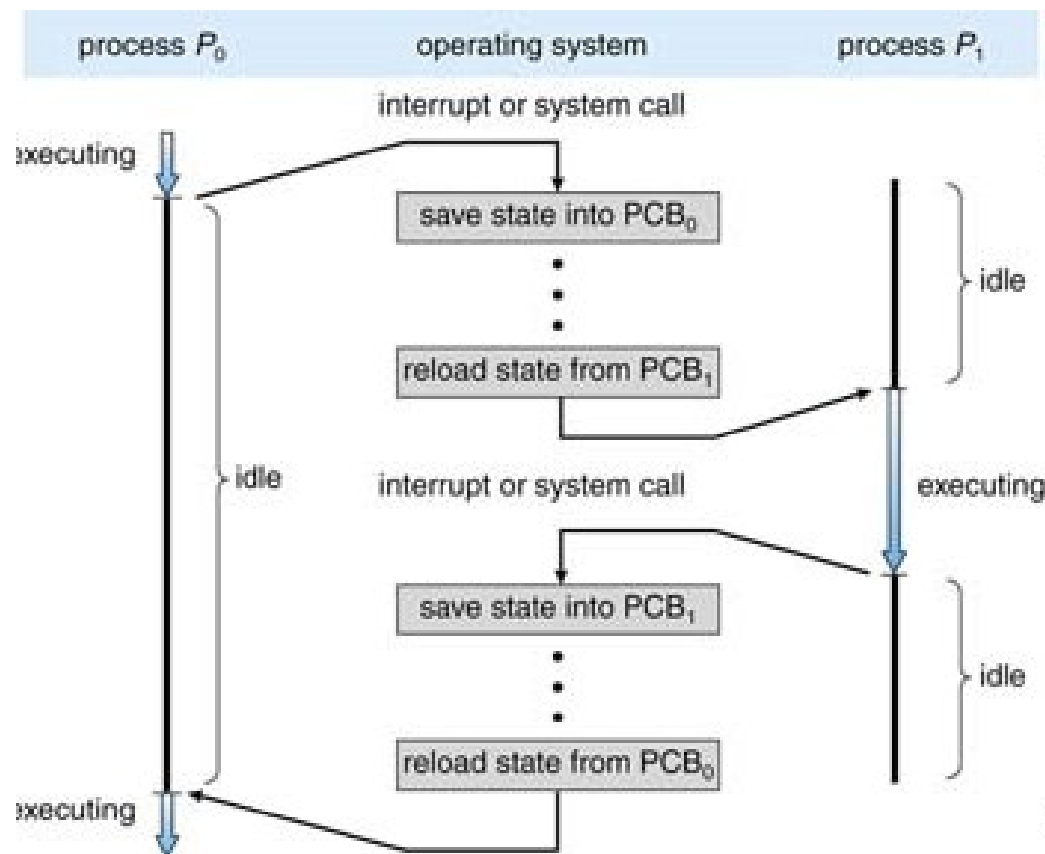
Process Control Block



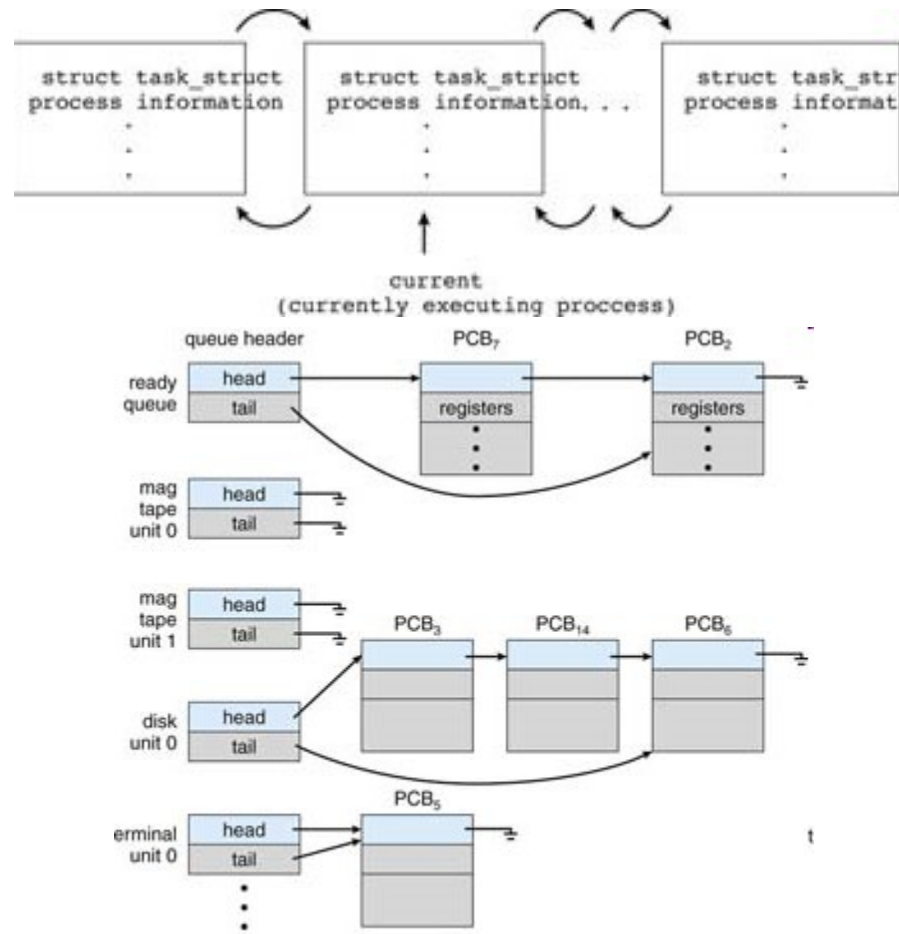
Process States



Switching between processes



PCB and queues in Linux



Process Creation

How to create a process? System call.

In UNIX, a process can create another process using the `fork()` system call

```
int pid = fork();      /* this is in C */
```

The creating process is called the parent and the new process is called the child

The child process is created as a copy of the parent process (process image and process control structure) except for the identification and scheduling state

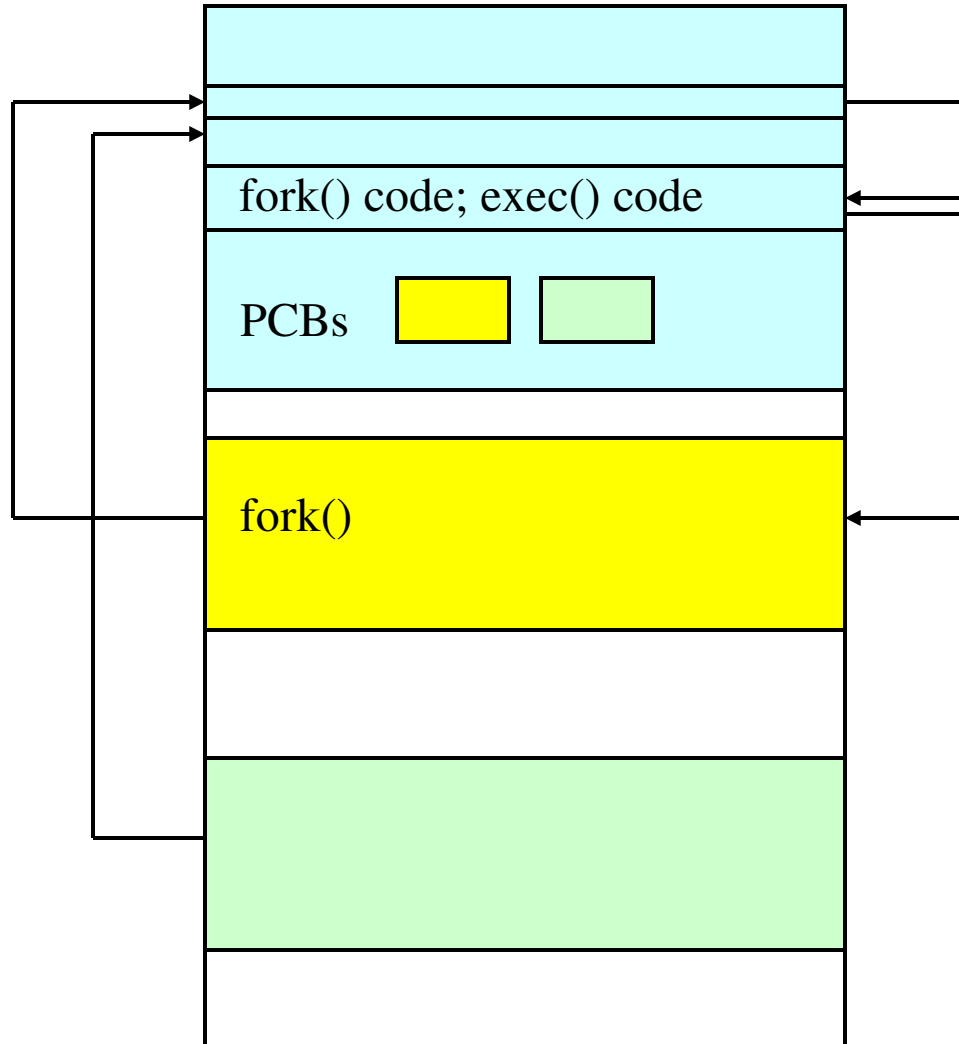
Parent and child processes run in two different address spaces

By default, there's no memory sharing

Process creation is expensive because of this copying

The `exec()` call is provided for the newly created process to run a different program than that of the parent

Process Creation



Example of Process Creation Using Fork

The UNIX shell is command-line interpreter whose basic purpose is for user to run applications on a UNIX system

cmd arg1 arg2 ... argn

```
while(TRUE) {
    get_command(command, parameters)

    if(fork() != 0) {    /* parent */
        wait(&status);
    } else {            /* child */
        exec(command, parameters)
    }
}
```

Another example (From textbook)

```
Main() {  
    pid_t pid;  
    pid = fork();  
    if (pid < 0) { //error }  
    else if (pid == 0) { execlp("/bin/ls", "ls", NULL) }  
    else { wait(NULL); printf ("child complete") }  
}
```

Process Death (or Murder)

One process can wait for another process to finish using the `wait()` system call

- Can wait for a child to finish as shown in the example

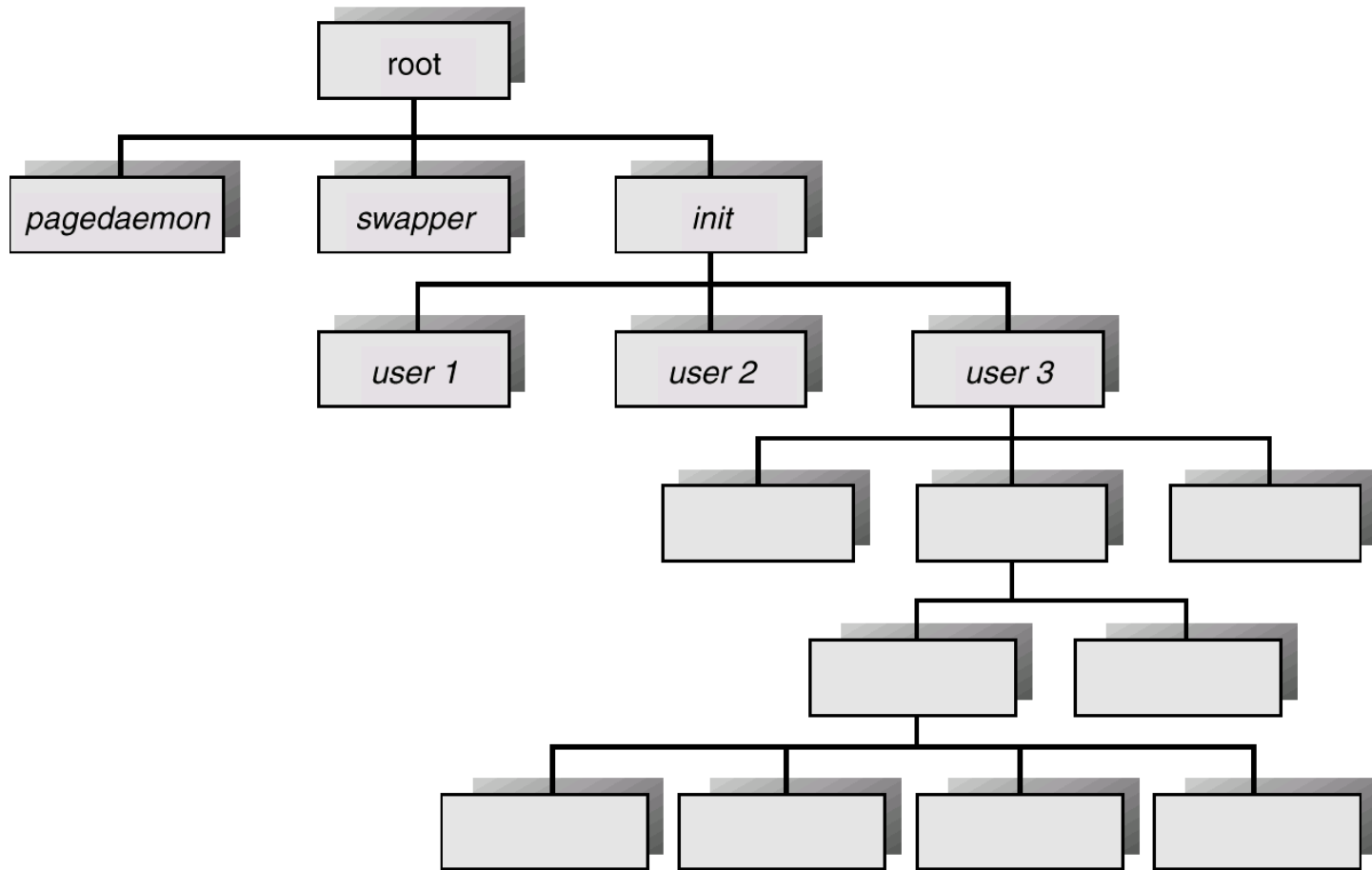
- Can also wait for an arbitrary process if know its PID

Can kill another process using the `kill()` system call

- What happens when `kill()` is invoked?

- What if the victim process doesn't want to die?

A Tree of Processes On A Typical UNIX System



Signals

User program can invoke OS services by using system calls

What if the program wants the OS to notify it *asynchronously* when some event occurs?

Signals

UNIX mechanism for OS to notify a user program when an event of interest occurs

Potentially interesting events are predefined: e.g., segmentation violation, message arrival, kill, etc.

When interested in “handling” a particular event (signal), a process indicates its interest to the OS and gives the OS a procedure that should be invoked in the upcall

How does a process indicate its interest in handling a signal?

Signals (Cont'd)

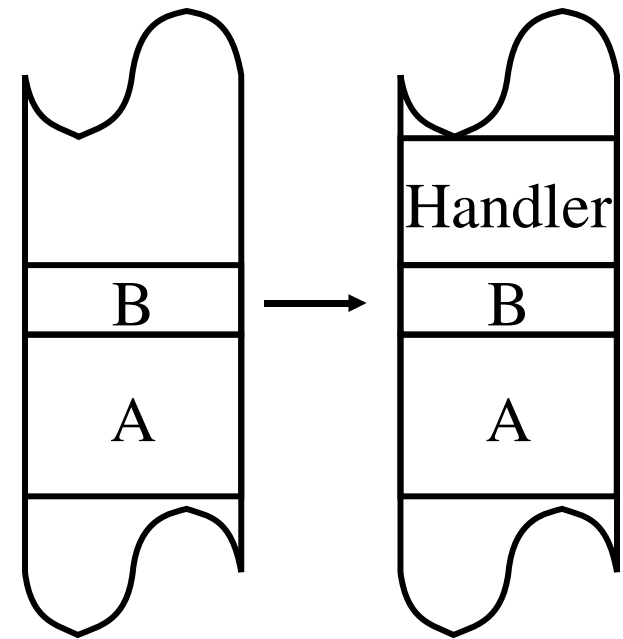
When an event of interest occurs:

The kernel handles the event first, then modifies the process's stack to look as if the process's code made a procedure call to the signal handler.

Puts an activation record on the user-level stack corresponding to the event handler

When the user process is scheduled next it executes the handler first

From the handler, the user process returns to where it was when the event occurred



Process: Summary

An “instantiation” of a program

System abstraction: the set of resources required for executing a program

- Execution context(s)

- Address space

- File handles, communication endpoints, etc.

Historically, all of the above “lumped” into a single abstraction

More recently, split into several abstractions

- Threads, address space, protection domain, etc.

OS process management:

- Supports user creation of processes and inter-process communication (IPC)

- Allocates resources to processes according to specific policies

- Interleaves the execution of multiple processes to increase system utilization

Next Time

Threads