



Processes

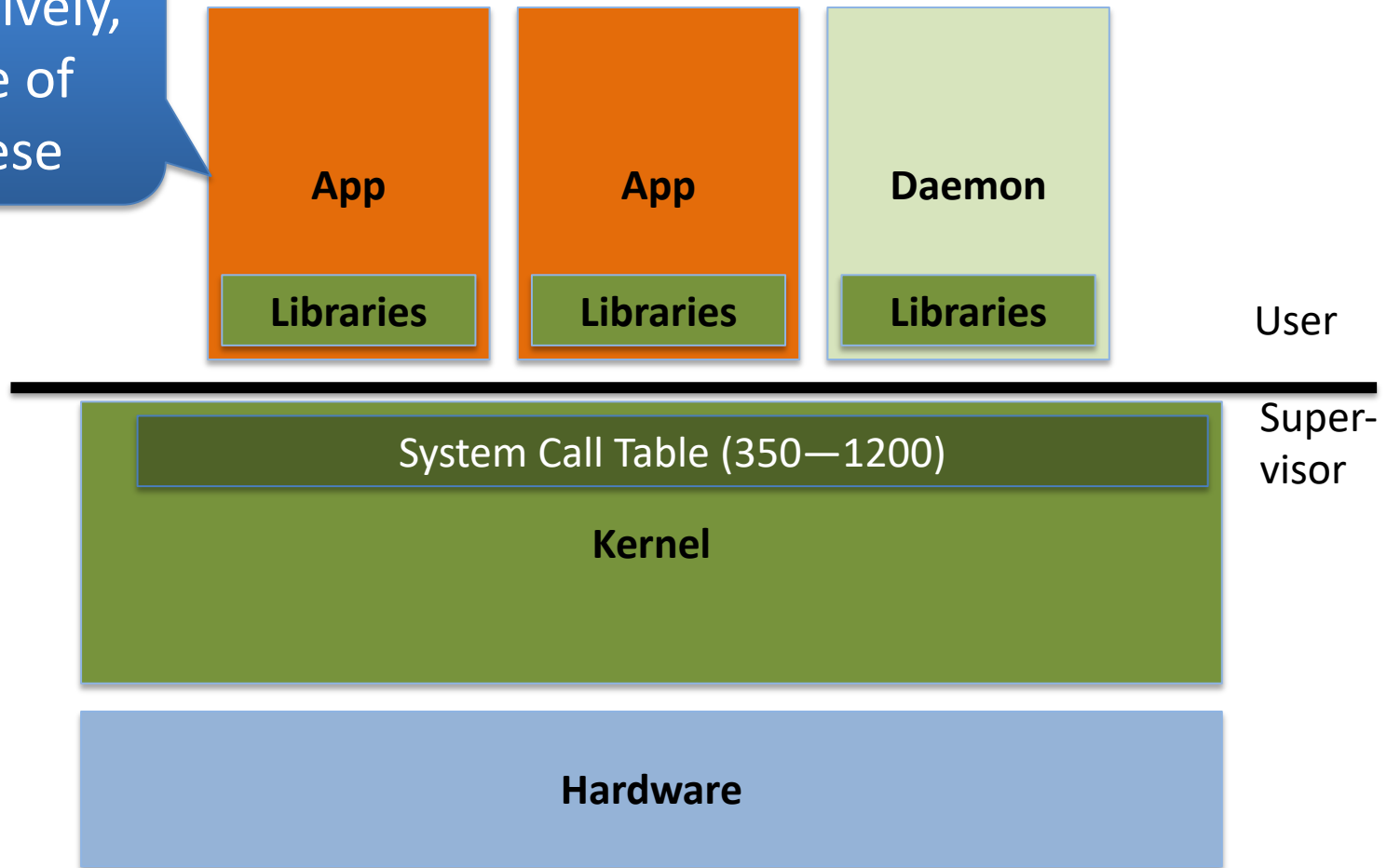
Don Porter

Portions courtesy Emmett Witchel



What is a process?

Intuitively,
one of
these





What is a process?

- A process is a **program during execution**.
 - Program = static file (image)
 - Process = executing program = program + execution state.
- A process is the basic unit of execution in an operating system
 - Each process has a number, its process identifier (pid).
- Different processes may run different instances of the same program
 - E.g., my javac and your javac process both run the Java compiler
- At a minimum, process execution requires following resources:
 - Memory to contain the program code and data
 - A set of CPU registers to support execution



Program to process

- We write a program in e.g., Java.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

```
void X (int b) {  
    if(b == 1) {  
...  
int main() {  
    int a = 2;  
    X(a);  
}
```



Process in memory

What you wrote:

```
void X (int b) {  
    if(b == 1) {  
...  
int main() {  
    int a = 2;  
    X(a);  
}
```

Data

What is in memory:

main; a = 2
X; b = 2

Stack



Heap

```
void X (int b) {  
    if(b == 1) {  
...  
int main() {  
    int a = 2;  
    X(a);  
}
```

Code



Where do processes come from?

- When I type './a.out', the binary runs, right?
 - Really only true for static binaries (more on this later)
- In reality a **loader** sets up the program
 - Usually a user-level program
 - Can also be in-kernel, or split between both



Where do processes come from?

- In order to run a program, the loader:
 - reads and interprets the executable file
 - sets up the process's memory to contain the code & data from executable
 - pushes “argc”, “argv” on the stack
 - sets the CPU registers properly & calls “_start()”
- Program starts running at _start()

```
_start(args) {  
    initialize_java();  
    ret = main(args);  
    exit(ret)  
}
```

“process” is now running; no longer think of “program”
- When main() returns, OS calls “exit()” which destroys the process and returns all resources

What bookkeeping does the OS need for processes?



Keeping track of a process

- A process has code.
 - OS must track program counter (code location).
- A process has a stack.
 - OS must track stack pointer.
- OS stores state of processes' computation in a process control block (PCB).
 - E.g., each process has an identifier (process identifier, or PID)
- Data (program instructions, stack & heap) resides in memory, metadata is in PCB (which is a kernel data structure in memory)



Context Switching

- The OS periodically switches execution from one process to another
- Called a **context switch**, because the OS saves one execution context and loads another

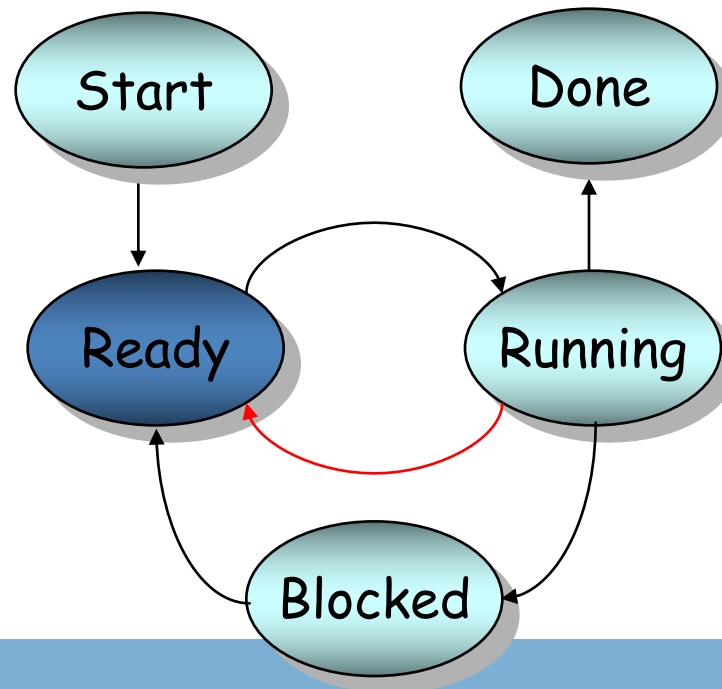


What causes context switches?

- Waiting for I/O (disk, network, etc.)
 - Might as well use the CPU for something useful
 - Called a blocked state
- Timer interrupt (preemptive multitasking)
 - Even if a process is busy, we need to be fair to other programs
- Voluntary yielding (cooperative multitasking)
- A few others
 - Synchronization, IPC, etc.

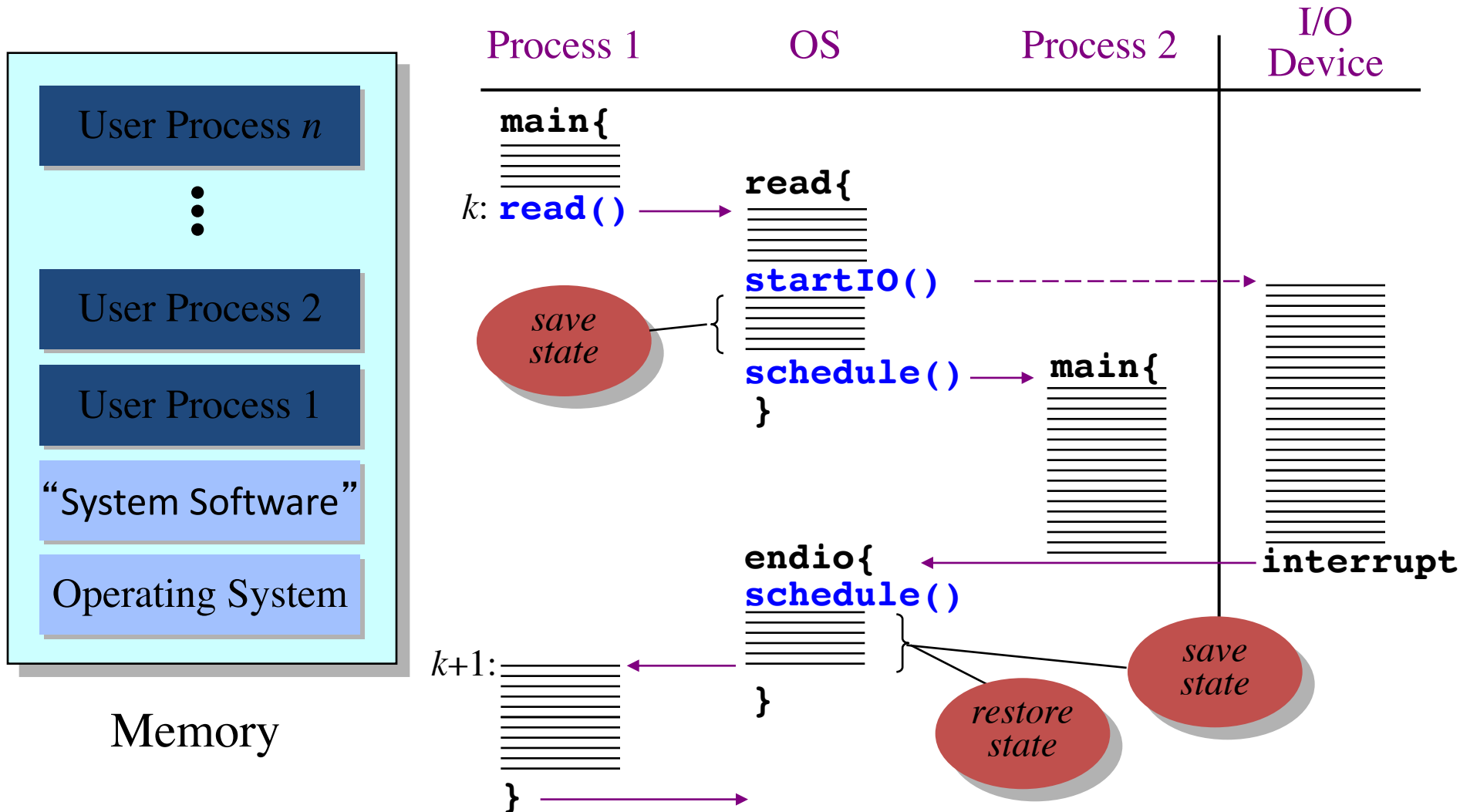
Process life cycle

- Processes are always either:
 - *Executing*
 - *Waiting to execute*, or
 - *Blocked waiting for an event* to occur





Process contexts





When a process is waiting for I/O, what is its state?

1. Ready
2. Running
3. Blocked 
4. Zombie
5. Exited



CPU Scheduling

- Problem of choosing which process to run next
 - And for how long until the next process runs
- Why bother?
 - Improve performance: amortize context switching costs
 - Improve user experience: e.g., low latency keystrokes
 - Priorities: favor “important” work over background work
 - Fairness

We will cover techniques later



When does scheduling happen?

- When a process blocks
- When a device interrupts the CPU to indicate an event occurred (possibly un-blocking a process)
- When a process yields the CPU
- **Preemptive scheduling:** Setting a timer to interrupt the CPU after some time
 - Places an upper bound on how long a CPU-bound process can run without giving another process a turn
- **Non-preemptive scheduling:** Processes must explicitly yield the CPU



Scheduling processes

- OS uses PCBs to represent a process
- Every resource is represented with a queue
- OS puts PCB on an appropriate queue.
 - Ready to run queue.
 - Blocked for IO queue (Queue per device).
 - Zombie queue.
- When CPU becomes available, choose from ready to run queue
- When an event occurs, remove waiting process from blocked queue, move to ready queue.



Why use multiple processes in one app?

Consider a Web server:

- get network message (URL) from client
- fetch URL data from disk
- compose response
- send response

How well does this web server perform?

With many incoming requests?

That access data all over the disk?

A single process cannot overlap CPU and I/O



Why use multiple processes in one app?

Consider a Web server

get network message (URL) from client

create child process, send it URL

Child

fetch URL data from disk

compose response

send response

- ◆ Now the child can block on I/O, parent keeps working
 - ◆ Different children can block on reading different files
- ◆ How does server know if child succeeded or failed?



Orderly termination: `exit()`

- After the program finishes execution, it calls `exit()`
- This system call:
 - takes the “result” of the program as an argument
 - closes all open files, connections, etc.
 - deallocates memory
 - deallocates most of the OS structures supporting the process
 - checks if parent is alive:
 - ❖ If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the `zombie/defunct` state
 - ❖ If not, it deallocates all data structures, the process is dead
- Process termination is the ultimate garbage collection

Web server ex: Child uses exit code for success/failure



The wait() system call

- Child returns a value to parent via `exit()`
- The parent receives this value with `wait()`
- Specifically, `wait()`:
 - Blocks the parent until child finishes (need a wait queue)
 - When a child calls `exit()`, the OS unblocks the parent and returns the value passed by `exit()` as a result of the `wait()` call (along with the pid of the child)
 - If there are no children alive, `wait()` returns immediately

Zombies!!!

- A parent can wait indefinitely to call `wait()`
- The OS to store the exit code for a finished child until the parent calls `wait()`
- Hack: Keep PCB for dead processes around until:
 - Parent calls `wait()`, or
 - Parent `exit()`s (don't need to `wait()` on grandkids)
- And that is a zombie (done state)
 - Will not be scheduled again





Where do processes come from? (redux)

- Parent/child model
- An existing program has to spawn a new one
 - Most OSes have a special ‘init’ program that launches system services, logon daemons, etc.
 - When you log in (via a terminal or ssh), the login program spawns your shell



Approach 1: Windows CreateProcess

- In Windows, when you create a new process, you specify the program
 - And can optionally allow the child to inherit some resources (e.g., an open file handle)



Approach 2: Unix fork/exec()

- In Unix, a parent makes a **copy** of itself using fork()
 - Child inherits everything, runs same program
 - Only difference is the return value from fork()
 - Child gets 0; parent gets child pid
- A separate exec() system call loads a new program
 - Like getting a brain transplant
- Some programs, like our web server example, fork() clones (without calling exec()).
 - Common case is probably fork+exec



Program loading: exec()

- The `exec()` call allows a process to “load” a different program and start execution at `main` (actually `_start`).
- It allows a process to specify the number of arguments (`argc`) and the string argument array (`argv`).
- If the call is successful
 - it is the same process ...
 - but it runs a different program !!
- Code, stack & heap is overwritten
 - Sometimes memory mapped files are preserved.
- **Exec does not return!**



fork() + exec() example

In the parent process:

main()

...

```
int rv = fork();                // create a child
if(0 == rv) {                  // child continues here
    exec_status = exec("calc", argc, argv0, argv1, ...);
    printf("Something is horribly wrong\n");
    exit(exec_status);
} else {                       // parent continues
    printf("Shall I be mother?");
    ...
    child_status = wait(rv);
}
```

Exec should not
return



A shell forks and execs a calculator

```
int rv = fork();  
if(rv == 0) {  
    close(".history");  
    exec("/bin/calc");  
} else {  
    wait(rv);  
}
```

```
int rvc=maink();  
if(rvc== 0) {  
    close(".history");  
    exec="/bin/calc";  
} else {  
    wait(rv);  
}
```

USER

OS

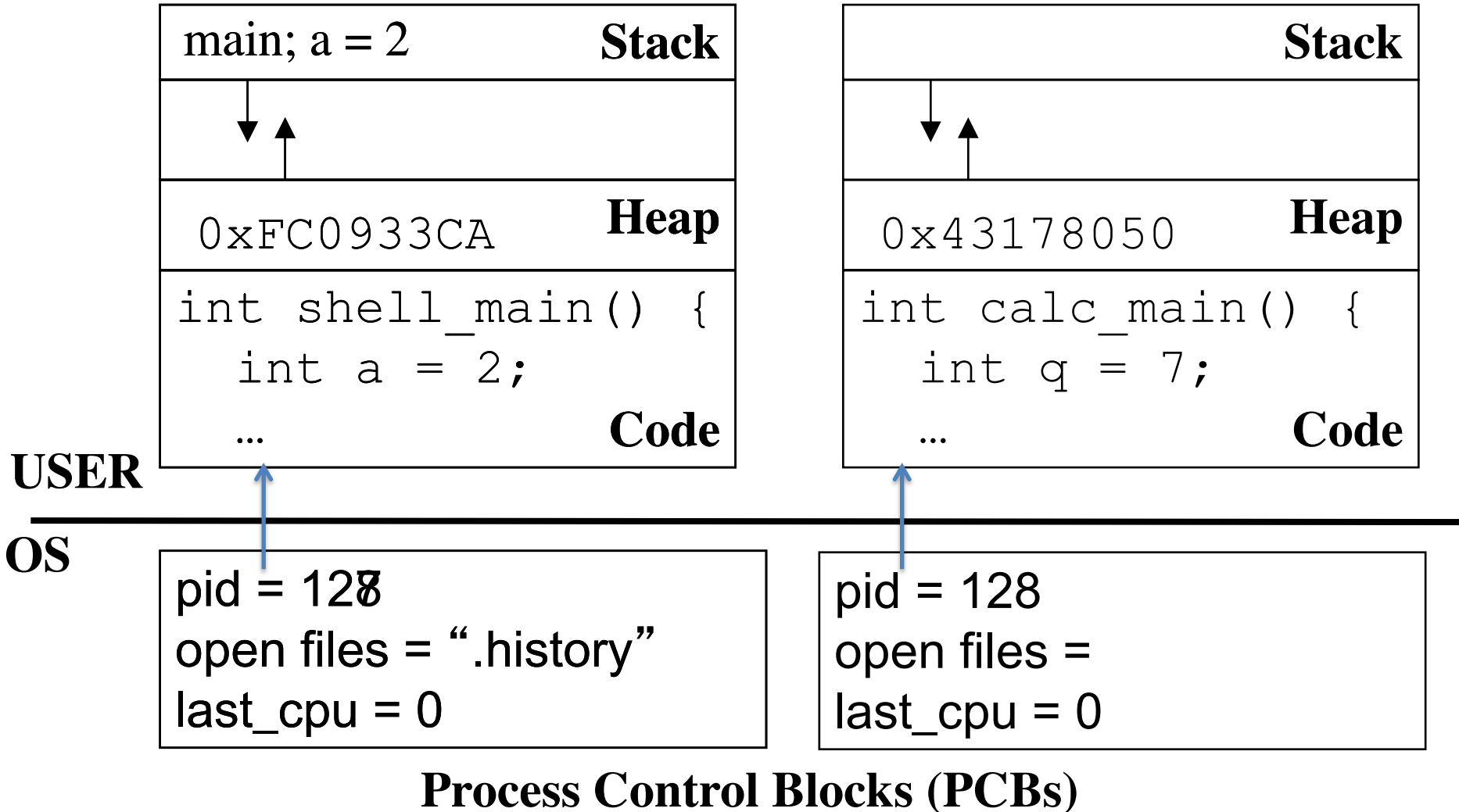
pid = 128
open files = ".history"
last_cpu = 0

pid = 128
open files =
last_cpu = 0

Process Control Blocks (PCBs)



A shell forks and then execs a calculator





Why separate fork & exec?

- Key issue: **Inheritance** of file descriptors, environment, etc.
 - Or, making the shell work
- Remember how the shell can do redirection?
 - `./warmup < testinput.txt`
 - File handle 0 (stdin) is opened to read testinput.txt
- The parent (shell) opens testinput.txt before fork()
 - The child (warmup) inherits this open file handle
 - Even after exec()



The convenience of separate fork/exec

- Decoupling fork and exec lets you do anything to the child's process environment without adding it to the CreateProcess API.

```
int rv = fork();                // create a child
if(0 == rv) {                  // child continues here
    // Do anything (unmap memory, close net connections...)
    exec("program", argc, argv0, argv1, ...);
}
```

- ◆ fork() creates a child process that inherits:
 - identical copy of all parent's variables & memory
 - identical copy of all parent's CPU registers (except one)
- ◆ Parent and child execute at the same point after **fork()** returns:
 - by convention, for the child, fork() returns 0
 - by convention, for the parent, fork() returns the pid of the child



The CreateProcess alternative

- Windows does allow you to create a process that is initially suspended
 - You can also change memory and handles of another process
 - And then unblock it
- Somewhat isomorphic
 - But a bit cumbersome
 - And prone to security issues (loading threads and libraries in another app!)



At what cost, fork()?

- Simple implementation of fork():
 - allocate memory for the child process
 - copy parent's memory and CPU registers to child's
 - *Expensive !!*
- In 99% of the time, we call exec() after calling fork()
 - the memory copying during fork() operation is useless
 - the child process will likely close the open files & connections
 - overhead is therefore high

Any ideas to improve this?



Pro tool: vfork

- If you know you are going to call `exec()` almost immediately:
 - Create a new PCB, stack, register state
 - But not a new copy of the full memory
- You can change OS state and call `exec` safely
- You cannot:
 - Return from the function that called `fork()`
 - Touch the heap
 - Probably other stuff
- Why does it improve performance? Avoids copies
- Unfortunate example of implementation influence on interface
 - Current Linux & BSD 4.4 have it for backwards compatibility



Copy-on-write fork (preview)

- Idea: write protect everything in memory after a `fork()`
 - Detect and copy only what you touch, until the `exec()`
 - After `exec()`, remove write protection from child memory
- Common case: `exec` quickly
 - Some overhead to setting copy-on-write, but cheaper than copying everything
- Uncommon case: fork never `execs`
 - Eventually copy everything
- We will see more about this later...



Process control

OS must include calls to enable special control of a process:

- Priority manipulation:
 - `nice()`, which specifies base process priority (initial priority)
 - In UNIX, process priority decays as the process consumes CPU
- Debugging support:
 - `ptrace()`, allows a process to be put under control of another process
 - The other process can set breakpoints, examine registers, etc.
- Alarms and time:
 - `Sleep` puts a process on a timer queue waiting for some number of seconds, supporting an alarm functionality



Tying it all together: The Unix shell

```
while(! EOF) {  
    read input  
    Parse and identify the program to execute  
    int rv = fork();                // create a child  
    if(rv == 0) {                  // child continues here  
        exec("program", argc, argv0, argv1, ...);  
    }  
    else {                          // parent continues here  
        ...  
    }  
}
```

- ◆ Translates <CTRL-C> to the kill() system call with SIGKILL
- ◆ Translates <CTRL-Z> to the kill() system call with SIGSTOP
- ◆ Allows input-output redirections, pipes, and a lot of other stuff that we will see later



Summary

- Understand what a process is
- The high-level idea of context switching and process states
- How a process is created
- Pros and cons of different creation APIs
 - Intuition of copy-on-write fork and vfork