Processes

ICS332

Operating Systems

Definition

- A process is a program in execution
  - program: passive entity (bytes stored on disk as part of an executable file)
  - becomes a process when it's loaded in memory
- Multiple processes can be associated to the same program
  - on a multi-user node (aka shared server) each user may start an instance of the same application (e.g., a text editor, the Shell)
  - A user can often start multiple instances of the same program
- A running system consists of multiple processes
  - OS processes: Processes started by the OS to do "system things"
    - Not everything's in the kernel after all (e.g., ssh daemon)
  - User processes
    - Execute user code, with the possibility of executing kernel code by going to kernel mode through system calls
- "job" and "process" are used interchangeably in OS texts

Definition

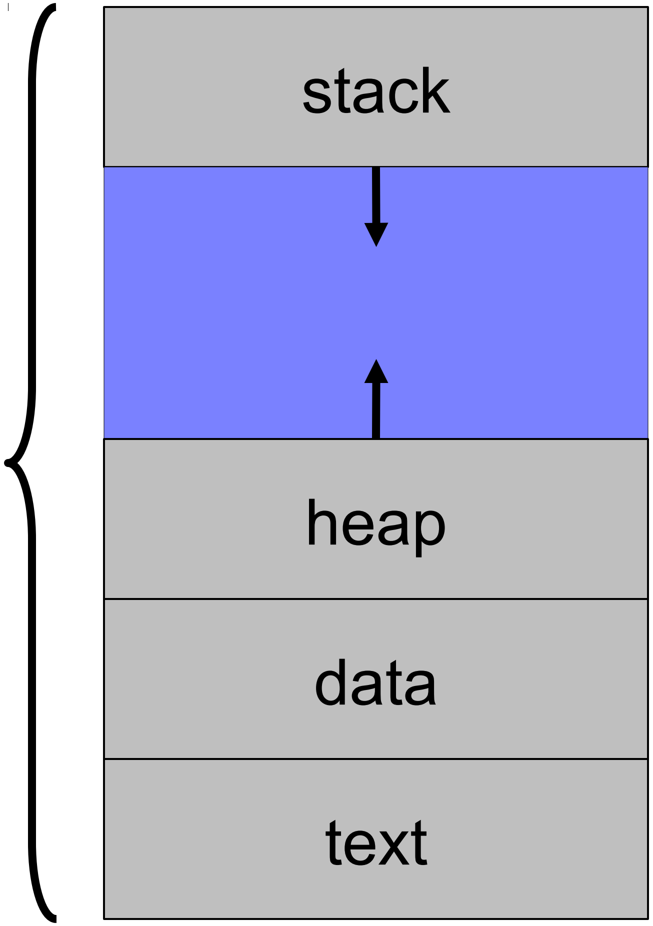
- What is in a process?
- Other way to think about it: what needs to be in memory/registers to fully define the state of a running program?

Definition

- Process =
  - code (also called the text)
    - initially stored on disk in an executable file
  - program counter
    - points to the next instruction to execute (i.e., an address in the code)
  - content of the processor’s registers
  - a runtime stack
  - a data section

    global variables (.bss (uninitialized static variables) and .data (initialized global variables and static local variables) in x86 assembly)
  - a heap
    - for dynamically allocated memory (malloc, new, etc.)

Process Address Space



bounded by
a max size

“Review”: The Stack

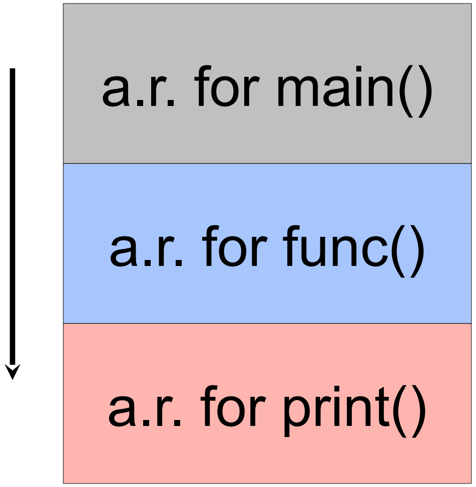
- The runtime stack is
  - A stack on which items can be pushed or popped
  - The items are called activation records
  - The stack is how we manage to have programs place successive function/method calls
  - The management of the stack is done entirely on your behalf by the compiler
    - Unless you took ICS312, in which case you saw how to manage the stack by hand (fun?)
- An activation record contains all the “bookkeeping” necessary for placing and returning from a function/method call

“Review”: Activation Record

- Any function needs to have some “state” so that it can run
  - The address of the instruction that should be executed once the function returns: the return address
  - Parameters passed to it by whatever function called it
  - Local variables
  - The value that it will return
- Before calling a function, the caller needs to also save the state of its registers
- All the above goes on the stack as part of activation records, which grows downward

Sample Runtime Stack

- main() calls func(), which calls print()



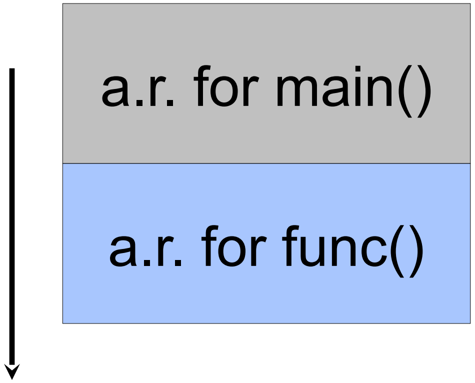
a.r. for main()

a.r. for func()

a.r. for print()

Sample Runtime Stack

- print() returns

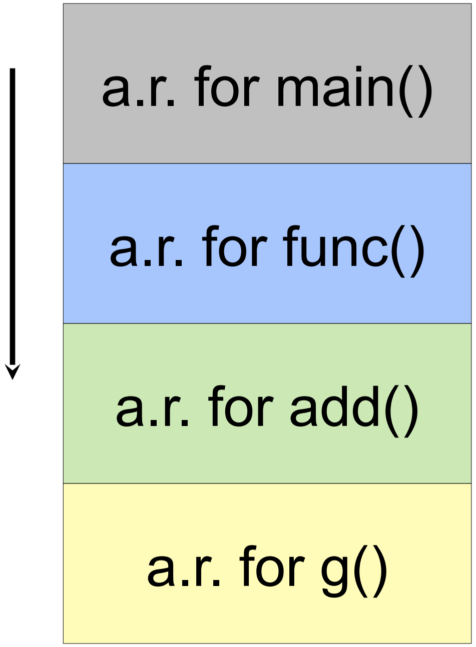


a.r. for main()

a.r. for func()

Sample Runtime Stack

- func() calls add(), which calls g()



a.r. for main()

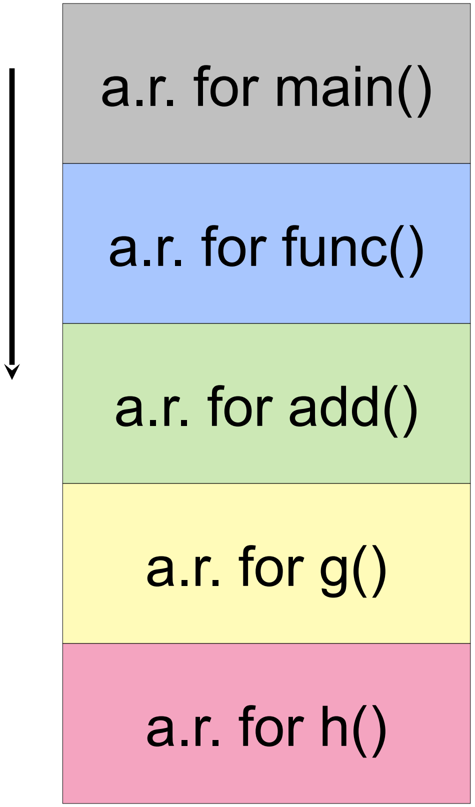
a.r. for func()

a.r. for add()

a.r. for g()

Sample Runtime Stack

- g() calls h()



a.r. for main()

a.r. for func()

a.r. for add()

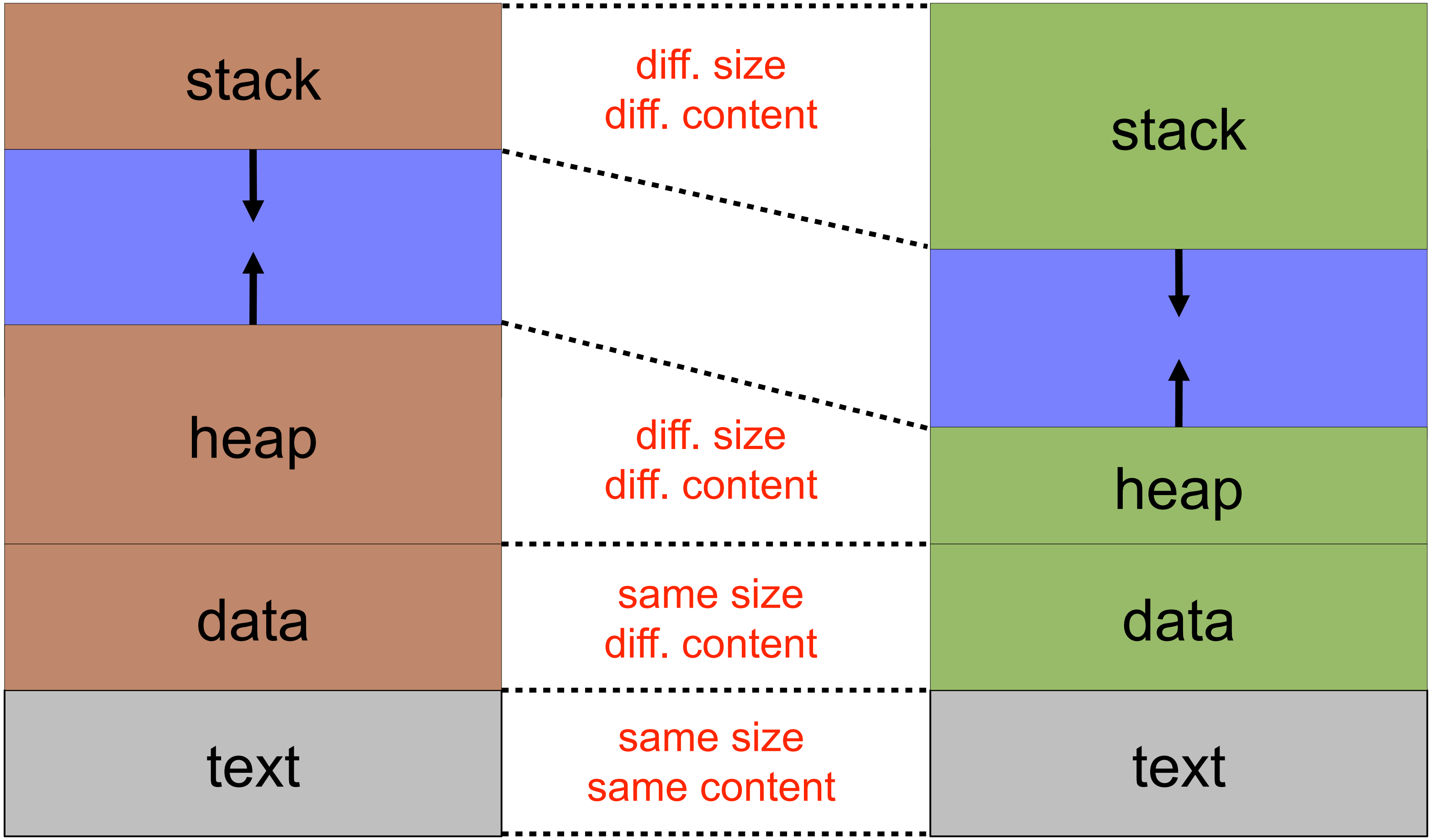
a.r. for g()

a.r. for h()

Runtime Stack Growth

- The mechanics for pushing/popping are more complex than one may think and pretty interesting (take ICS312)
- The longer the call sequence, the larger the stack
  - Especially with recursive calls!!
- The stack can get too large
  - Hits some system-specified limit
  - Hits the heap
- The famous “runtime stack overflow” error
  - Causes a trap, that will trigger the Kernel to terminate your process with that error
  - Typically due to infinite recursion

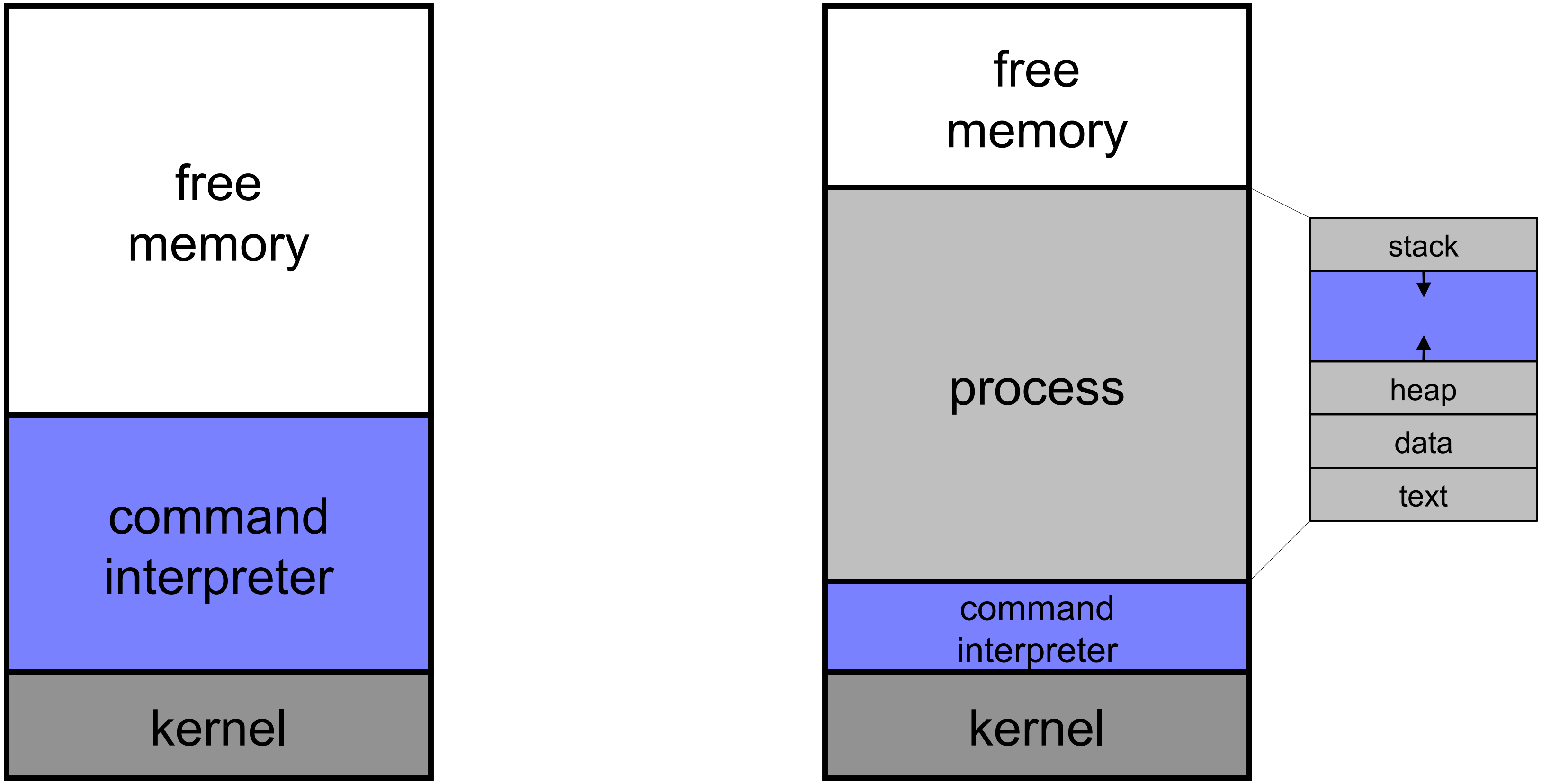
2 Processes for 1 Program



Single- and Multi-Tasking

- OSes used to be single-tasking: only one process can be in memory at a time
- MS-DOS is the best known example
  - A command interpreter is loaded upon boot
  - When a program needs to execute, no new process is created
  - Instead the program's code is loaded in memory by the command interpreter, which overwrites part of itself with it!
    - Memory used to be very scarce
  - The instruction pointer is set to the 1st instruction of the program
  - The small left-over portion of the interpreter resumes after the program terminates and produces an exit code
  - This small portion re-loads the full code of the interpreter from disk back into memory
  - The full interpreter resumes and provides the user with his/her program's exit code

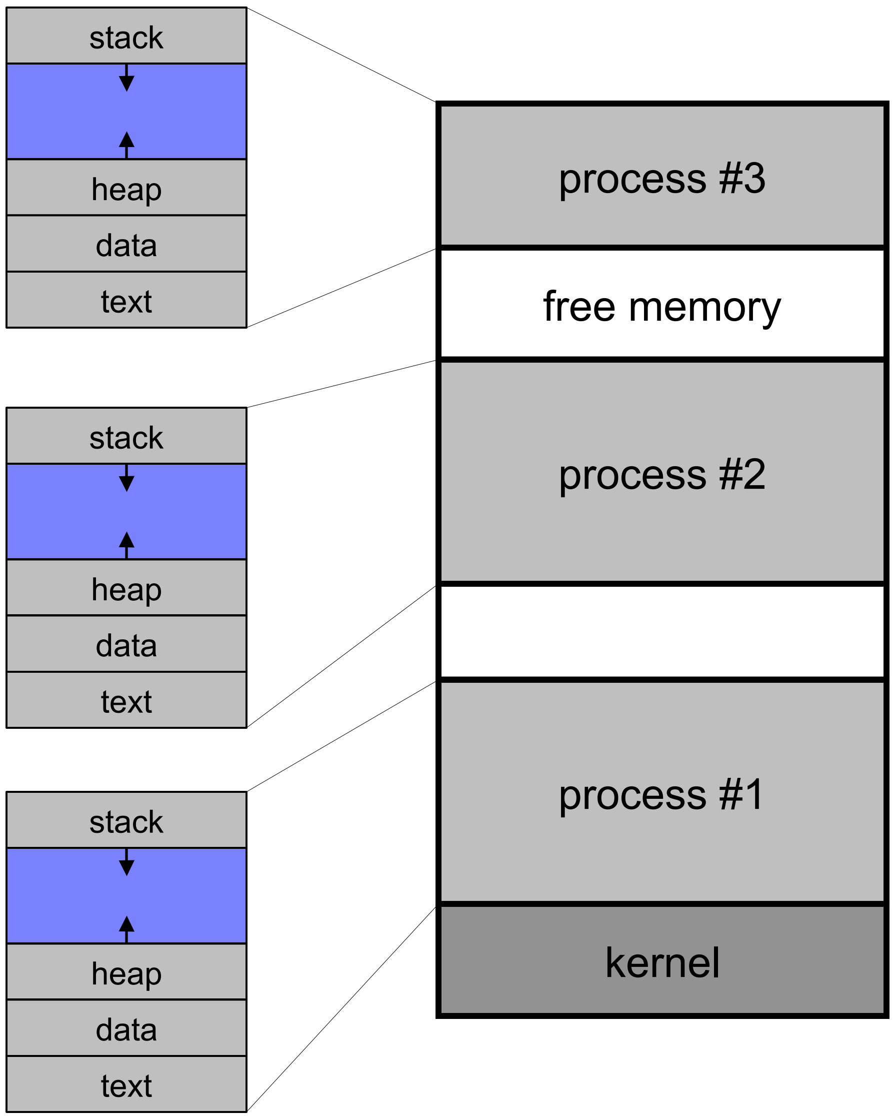
Single-Tasking with MS-DOS



idle
full-fledge command-interpreter

running a program
small command-interpreter left

Multi-Tasking (Multi-Programming)



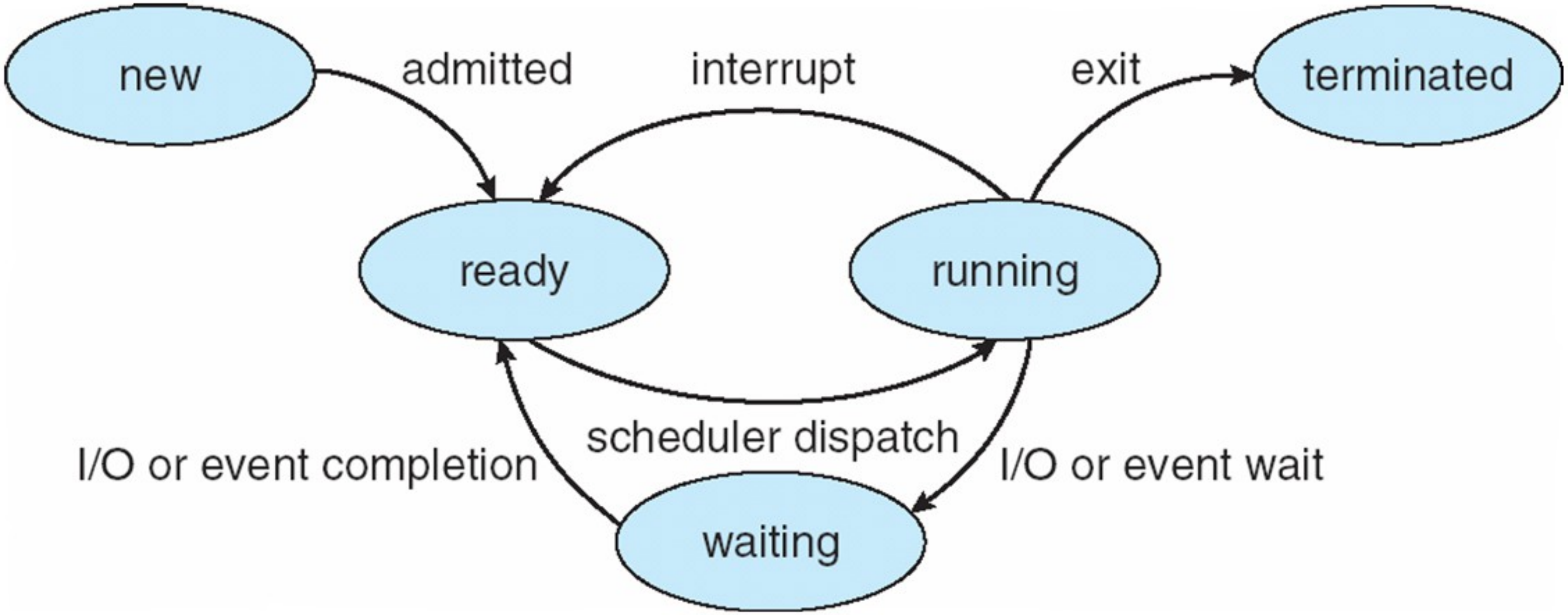
- Modern OSes support multi-tasking: multiple processes can co-exist in memory
- To start a new program, the OS simply creates a new process (via a system-call called fork() on a UNIX system)

Kernel Stack?

- Within the kernel, the code calls a series of functions
- Important: the kernel has a fixed-size stack
  - It is not very large (e.g., 4KB to 16KB) → *ulimit -s*
- When writing kernel code, there is no such thing as allocating tons of temporary variables, or calling tons of nested functions each with tons of arguments
  - That's a luxury only allowed in user space
- There are many such differences between user-space development and kernel-space development
- Example of another difference: when writing kernel code, one doesn't have access to the standard C library!
  - Chicken-and-egg problem
  - Would be inefficient anyway
- So the kernel re-implements some useful functions
  - e.g., printk() replaces printf() and is implemented in the kernel source
- And yes, the Linux kernel is written in C

Process State

- As a process executes, it may be in various states
- These states are defined by the OS, but most OSes use (at least) the states below

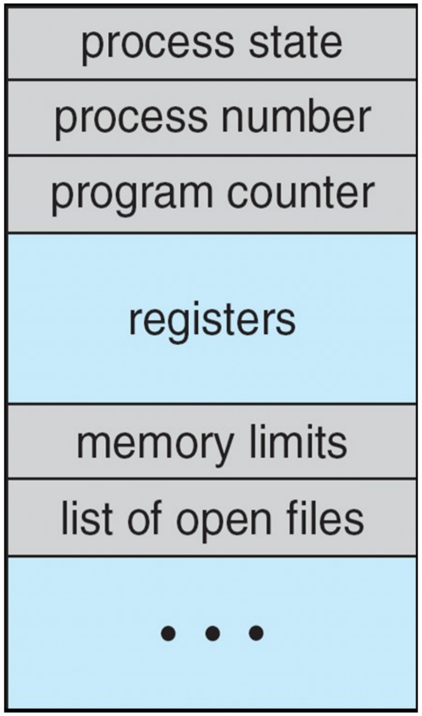


Process Control Block

- The OS keeps track of processes in a data structure, the process control block (PCB), which contains:
  - Process state
  - Process ID (aka PID)
  - Program counter and CPU registers contents
    - when saved, allow a process to be restarted later
  - CPU-scheduling info
    - priority, queue, ... (see future lecture “Scheduling”)
  - Memory-management info
    - base and limit registers, page table, ... (see future lectures “Main Memory” and “Virtual Memory”)
  - Accounting info
    - amount of resources used so far, ...
  - I/O status info
    - list of I/O devices allocated to the process, open files, ...

Process Control Block

- Figure from the book



- The reality is of course a bit messier
  - include/linux/sched.h *(look up “task_struct {”)*
  - See page 110 in the textbook

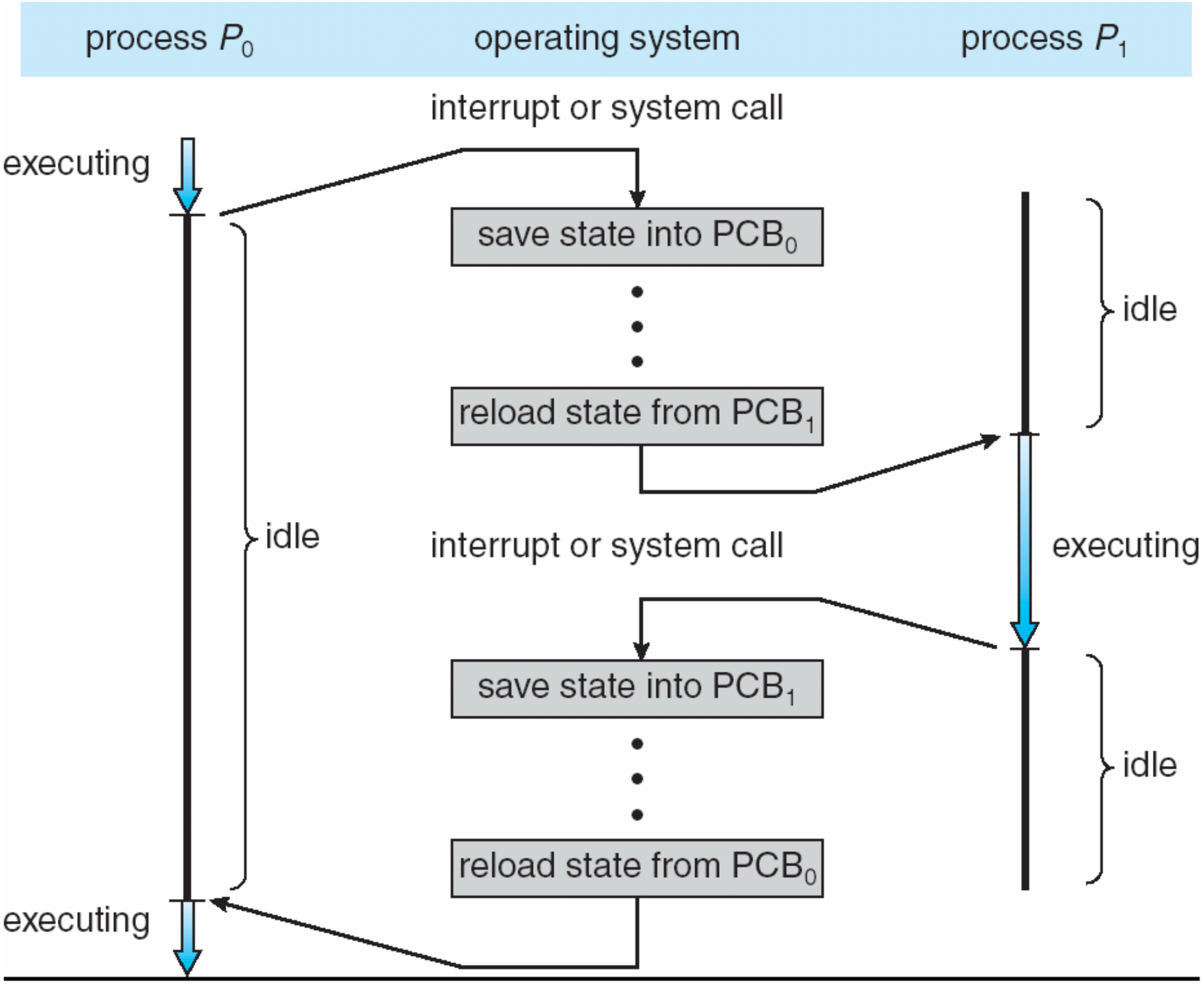
The Kernel’s “Process Table”

- The Kernel keeps around all the PCB in its memory, in a data structure often called the Process Table
- Because Kernel size must be bounded, the Process Table size is also bounded
  - Based on a configuration parameter of the kernel, but you can’t set it to infinity
- Therefore the Process Table can fill up!
- If you keep creating processes that don’t terminate, eventually you won’t be able to create new processes
  - And your system will be in trouble
- It’s very easy to write code that does this
  - Called a “fork bomb” (see upcoming slides)

Disclaimer for what Follows

- In all that follows we assume a single-CPU system
- The book talks about threads, and talks about schedulers and other things in Chapter 3
  - The author tends to keep giving preview of future chapters
  - I chose to not give too many previews
  - You may skip that content in the book until a future lecture
    - as mentioned in the reading assignment on the web site
- Important: with the above assumptions, only one process is executed by the CPU at a time
  - Multiple processes may be loaded in memory
  - But only one is in the "Running" state
  - All others are, e.g., in the "Ready" state
- The OS gives the CPU to a process for a limited amount of time, then gives it to another process, and so on

Switching between Processes

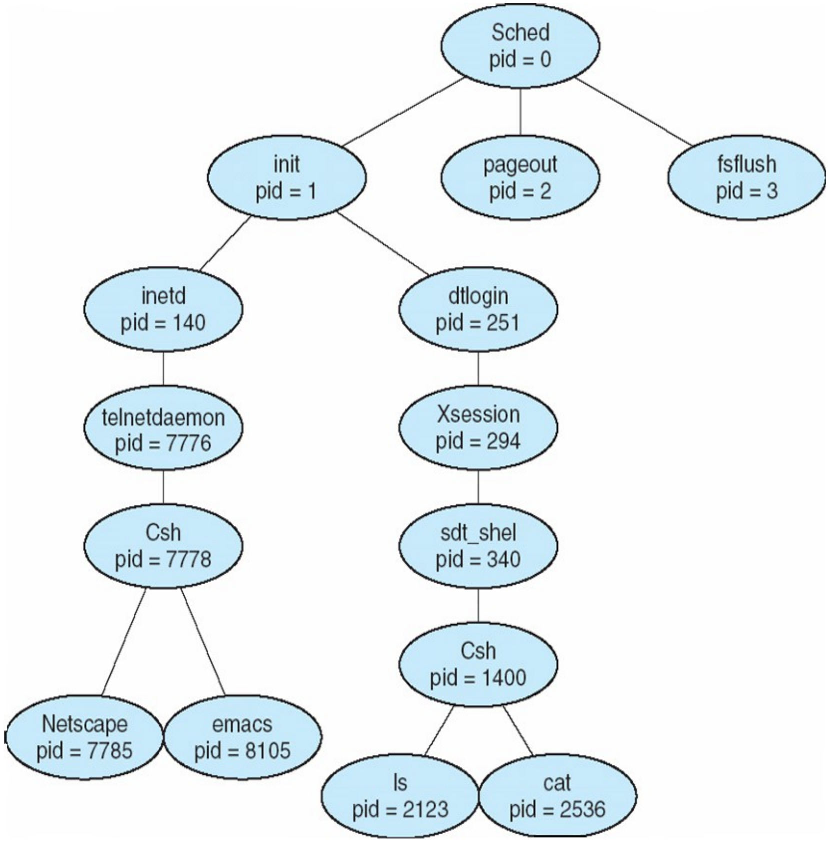


Switching between Processes

- This switching is called context switching
  - The context is the state of the running process
- Context-switching time is pure overhead
  - While it happens processes do not do useful work
- Therefore it should be fast
  - No more than a few microseconds, and hopefully less
- The hardware can help
  - e.g., save all registers in a single instruction
  - e.g., multiple register sets
    - Switching between register sets is done with a simple instruction
    - If more processes than register sets, then revert to the usual save/restore
- Context switching is the *mechanism*. The *policy* is called scheduling
  - See future lecture

Process Creation

- A process may create new processes, in which case it becomes a parent
- We obtain a tree of processes
- Each process has a pid
  - ppid refers to the parent’s pid
- Example tree, on Solaris



- ps axlw on a Mac OSX system gives the “tree” (ps faux / ps --forest -eaf)

Process Creation

- The child may inherit/share some of the resources of its parent, or may have entirely new ones
    - Many variants are possible and we'll look at what Linux does
- A parent can also pass input to a child
- Upon creation of a child, the parent can either
    - continue execution, or
    - wait for the child's completion
- The child could be either
    - a clone of the parent (i.e., have a copy of the address space), or
    - be an entirely new program
- Let's look at process creation in UNIX / Linux
- You can read the corresponding man pages
    - "man 2 *command*" or "man 3 *command*"

The fork() System Call

- fork() creates a new process
- The child is a copy of the parent, but...
    - It has a different pid (and thus ppid)
    - Its resource utilization (so far) is set to 0
- fork() returns the child’s pid to the parent, and 0 to the child
    - Each process can find its own pid with the getpid() call, and its ppid with the getppid() call
- Both processes continue execution after the call to fork()

fork() Example

fork_example1.c

```
pid = fork();
if (pid < 0) {
    fprintf(stdout,”Error: can’t fork()\n”);
    perror(“fork()”);
} else if (pid != 0) {
    fprintf(stdout,”I am the parent and my child has pid %d\n”,pid);
    while (1);
} else {
    fprintf(stdout,”I am the child, and my pid is %d\n”, getpid());
    while (1) ;
}
```

- You should _always_ check error codes (as above for fork())
  - in fact, even for fprintf, although that’s considered overkill
  - I don’t do it here for the sake of brevity (see sources on the Web site)

fork() and Memory

- What does the following code print?

fork_example2.c

```
int a = 12;
if (pid = fork()) { // PARENT
      sleep(10);  // ask the OS to put me in Waiting
      fprintf(stdout,”a = %d\n”,a);
      while (1);
} else {  // CHILD
      a += 3;
      while (1);
}
```

fork() and Memory

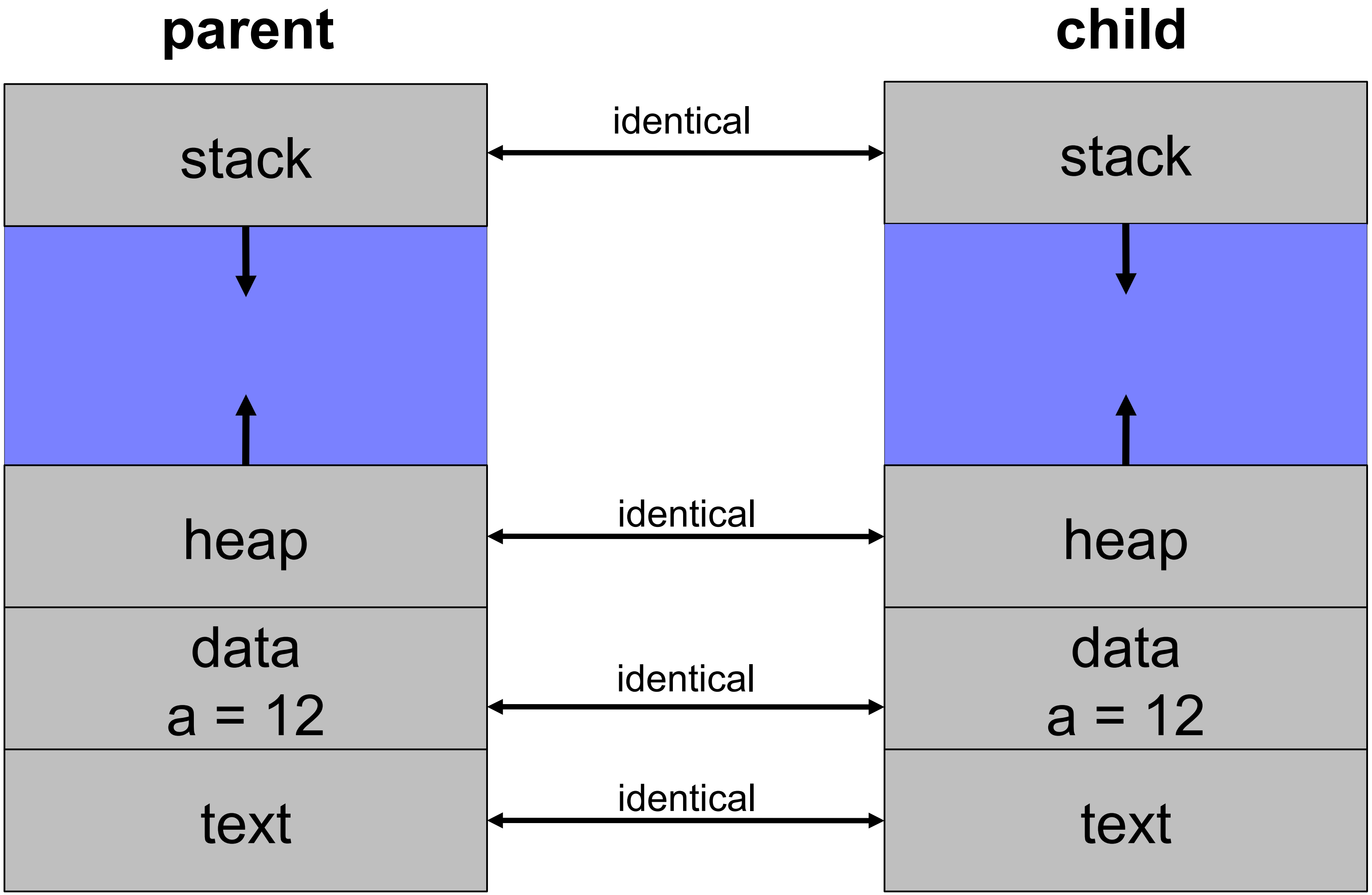
- What does the following code print?

fork_example2.c

```
int a = 12;
pid = fork();
if (pid != 0) {
      sleep(10);  // ask the OS to put me in Waiting
      fprintf(stdout,”a = %d\n”,a);
      while (1);
} else {
      a += 3;
      while (1);
}
```

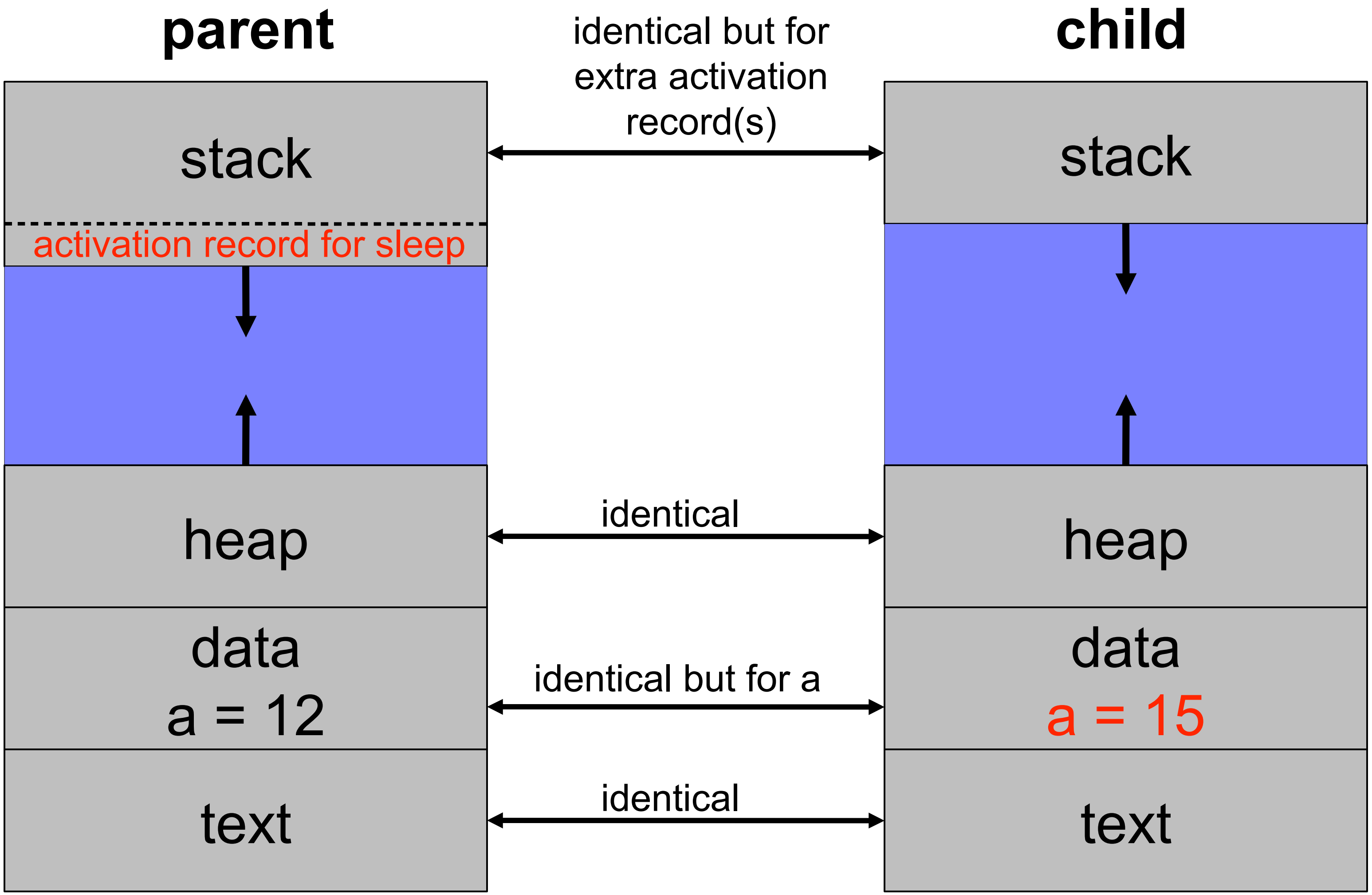
Answer: 12

fork() and Memory



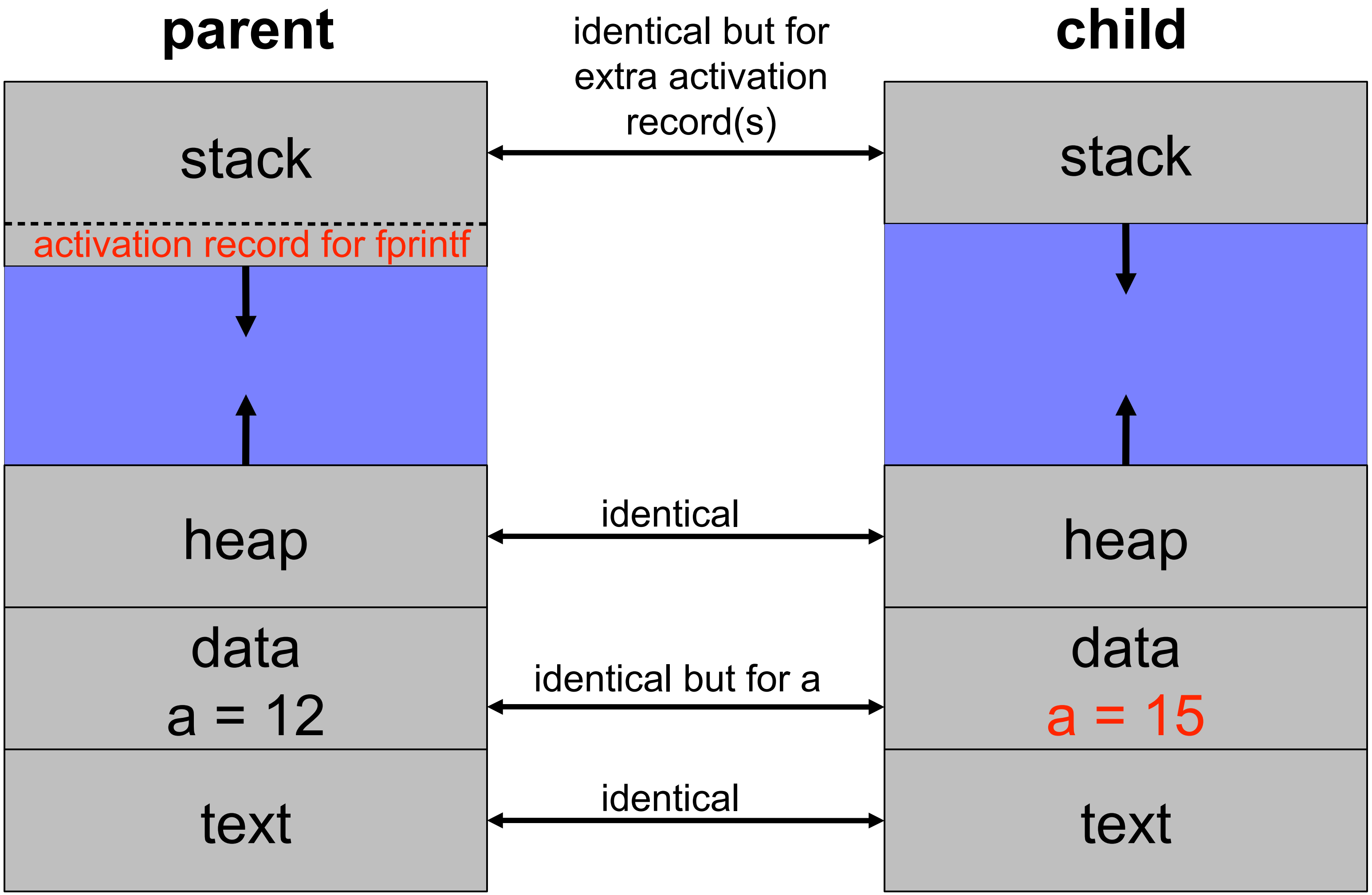
State of both processes right after fork() completes

fork() and Memory



State of both processes right **before** sleep returns

fork() and Memory



State of both processes right before fprintf returns (“12” gets printed)

fork() can be confusing

- How many times does this code print “hello”?

fork_example3.c

```
pid1 = fork();
fprintf(stdout,”hello\n”);
pid2 = fork();
fprintf(stdout,”hello\n”);
```

fork() can be confusing

- How many times does this code print “hello”?

```
pid1 = fork();
fprintf(stdout,”hello\n”);
pid2 = fork();
fprintf(stdout,”hello\n”);
```

fork_example3.c

Answer: 6 times

Fork bombs…

- C:

```
int main() {
    while (1) {  fork(); }
}
```

- Bash:

```
:(){ :|: & };:
```

- Limit the number of processes by user

```
ulimit -u <maximum number of processes>
```

The exec★() Family of Syscalls

- The “exec” system call replaces the process image by that of a specific program
  - see “man 3 exec” to see all the versions
- Essentially one can specify:
  - path for the executable
  - command-line arguments to be passed to the executable
  - possibly a set of environment variables
- An exec() call returns only if there was an error
- Example in the book: Figure 3.10
- Typical example (note the argv[0] value!!!)

```
if (fork() == 0) {  // runs ls
        char *const argv[] = {“ls”, “-l”,”/tmp/”,NULL};
        execv(“/bin/ls”, argv);
}
```

exec_example.c

Process Terminations

- A process terminates itself with the exit() system call
  - This call takes as argument an integer that is called the process' exit/return/error code
- All resources of a process are deallocated by the OS
  - physical and virtual memory, open files, I/O buffers, ...
- A process can cause the termination of another process
  - Using something called "signals" and the kill() system call

wait() and waitpid()

- A parent can wait for a child to complete
- The wait() call
  - blocks until any child completes
  - returns the pid of the completed child and the child’s exit code
- The waitpid() call
  - blocks until a specific child completes
  - can be made non-blocking
- Let’s look at wait_example1.c and wait_example2.c on the Web site
- Read the man pages (“man waitpid”)

Processes and Signals

- A process can receive signals, i.e., software interrupts
  - It is an asynchronous event that the program must act upon, in some way
- Signals have many usages, including process synchronization
  - We'll see other, more powerful and flexible process synchronization tools
- The OS defines a number of signals, each with a name and a number, and some meaning
  - See /usr/include/sys/signal.h or "man 7 signal"
- Signals happen for various reasons
  - ^C on the command-line sends a SIGINT signal to the running command
  - A segmentation violation sends a SIGBUS signal to the running process
  - A process sends a SIGKILL signal to another

Manipulating Signals

- Each signal causes a default behavior in the process
  - e.g., a SIGINT signal causes the process to terminate
- But most signals can be either ignored or provided with a user-written handler to perform some action
  - Signals like SIGKILL and SIGSTOP cannot be ignored or handled by the user, for security reasons
- The signal() system call allows a process to specify what action to do on a signal:
  - signal(SIGINT, SIG_IGN); // ignore signal
  - signal(SIGINT, SIG_DFL); // set behavior to default
  - signal(SIGINT, my_handler);// customize behavior
    - handler is as: void my_handler(int sig) { ... }
- Let's look at a small example of a process that ignores SIGINT

Signal Example

signal_example.c

```
#include <signal.h>
#include <stdio.h>

void handler(int sig) {
    fprintf(stdout,”I don’t want to die!\n”);
    return;
}

main() {
    signal(SIGINT, handler);
    while(1); // infinite loop
}
```

They’re dead.. but alive!

- When a child process terminates, it remains as a **zombie** in an “undead” state (until it is “reaped” by the OS)
- Rationale: the child’s parent may still need to place a call to wait(), or a variant, to retrieve the child’s exit code
- The OS keeps zombies around for this purpose
  - They’re not really processes, they do not consume resources
  - They only consume a slot in the OS’s “process table”
    - Which may eventually fill up and cause fork() to fail
- Let's look at zombie_example.c on the Web site
- A zombie lingers on until:
  - its parent has called wait() for the child, or
  - its parent dies
- It is bad practice to leave zombies around unnecessarily

Getting rid of zombies

- When a child exits, a SIGCHLD signal is sent to the parent
- A typical way to avoid zombies altogether:
  - The parent associates a handler to SIGCHLD
  - The handler calls wait()
  - This way all children deaths are “acknowledged”
  - See nozombie_example.c on the Web site

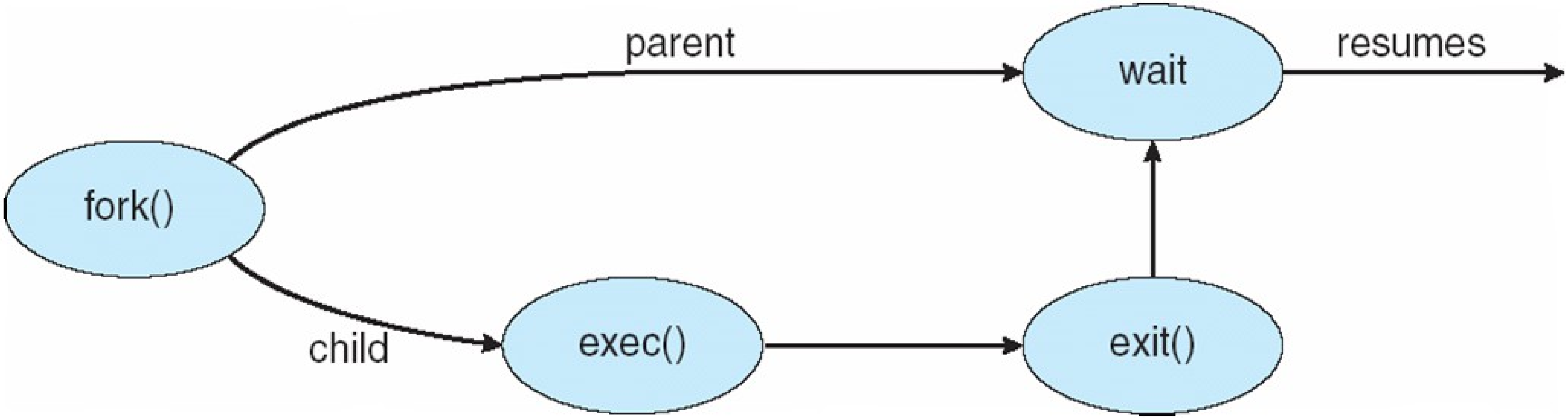
Orphans

- An orphan process is one whose parent has died
- In this case, the orphan is “adopted” by the process with pid 1
  - init on a Linux system / launchd on a Mac OS X system
- The process with pid 1 does handle child termination with a handler for SIGCHLD that calls wait (just like in the previous slide!)
- Therefore, an orphan never becomes a zombie
- “Trick” to fork a process that’s completely separate from the parent (with no future responsibilities): create a grandchild and “kill” its parent

```
if (!fork()) { // code of the child
        if (!fork()) {  // code of the grandchild, adopted by pid=1
    . . .
          exit(0);    // will be reaped by process pid=1
  }
        exit(0);      // will be reaped by the parent
} else {  // code of the parent
        wait(NULL);   // wait for the child to exit
}
```

orphan_example1.c
orphan_example2.c

In a Nutshell



What about Windows?

- See example in Figure 3.11
- In Windows, the CreateProcess() call combines fork() and exec()
  - Separation of fork() and exec() allows many clever “tricks” in UNIX, which are not possible in Windows
  - See also the spawn() functions family
- In Win32 fashion, calls have many arguments
- There is an equivalent to wait(): WaitForSingleObject()
- TerminateProcess() is like kill()

- So, overall, it allows for the same capabilities (which shouldn’t be surprising), but with a different flavor
  - Developers are really opinionated about this

Fork() with no exec() nowadays?

Nowadays because of threads fork() may seem useless without exec()

More about Threads in the lecture about them

… google-chrome vs firefox

Processes in Java

- In this course you'll write Java code
- What about Java and processes?
- The JVM doesn't implement a Process abstraction similar to C, meaning that there is no notion of running multiple processes **within** the JVM
  - Partly because supporting several independent address spaces in the JVM is a pain
- It's is however possible to create an "external process" that lives **outside** the JVM
  - Communication is via data streams
  - We'll see this in a future lecture

Conclusion

- Processes are running programs
- OSes provides a rich set of abstractions and system calls to deal with processes
  - Make sure you understand all the examples
  - Even better if you experiment yourself by compiling/playing with them
- In Java, one can only create external “OS” processes
  - Multiple independent execution entities in the JVM must be threads