Operating Systems: Processes

CSC-4320/6320 –Summer 2014

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 is loaded into memory
- Multiple processes can be associated to the same program
  - On a shared server each user may start an instance of the same application (e.g., a text editor, the Shell)
- 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
  - “jobs” and “processes” are used interchangeably in OS texts

Definition

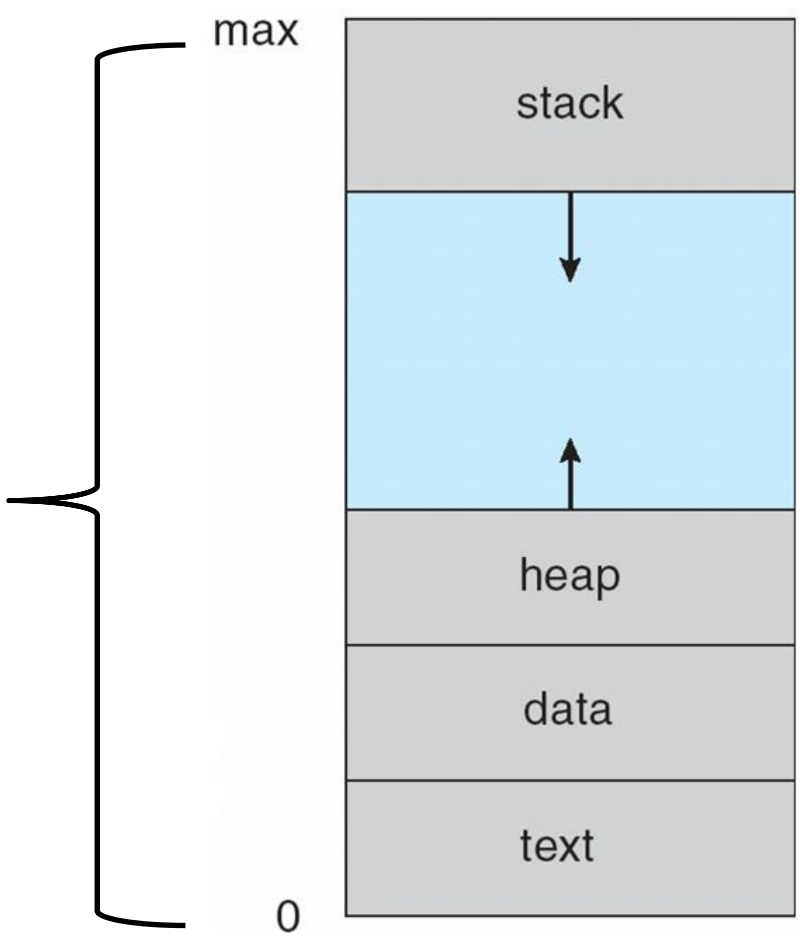
- What do you think is in a process?
- Other way to think about it: what needs to be in memory/registers to fully define the state of the running program?

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 and .data in x86 assembly)
  - A heap
    - For dynamically allocated memory (malloc, new, etc.)

Process Address Space

Bounded by
A max size



“Review”: The Stack

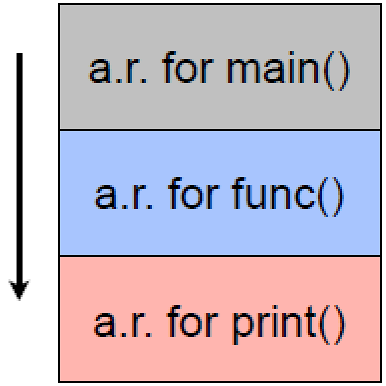
- The runtime stack is
  - A stack on which items can be pushed or popped
  - The items are called activation records
  - The stack is how we manage to have programs place successive function/method calls
  - The management of the stack is done entirely on your behalf by the compiler
    - Unless you are programming in assembly, loads of fuunnn.
- 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 the activation records, which grow downward

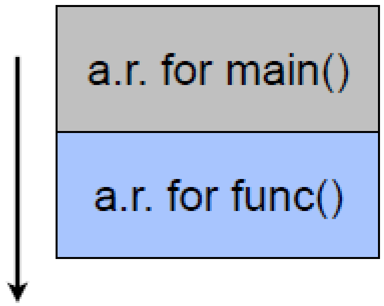
Sample Runtime Stack

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



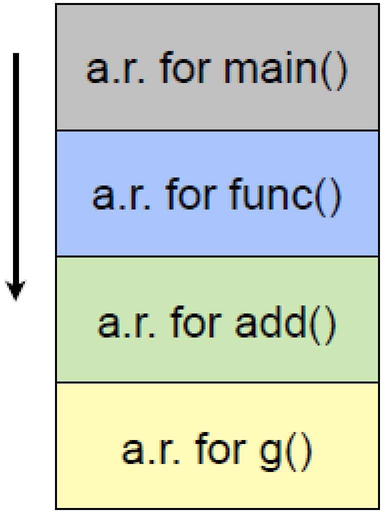
Sample Runtime Stack

- print() returns



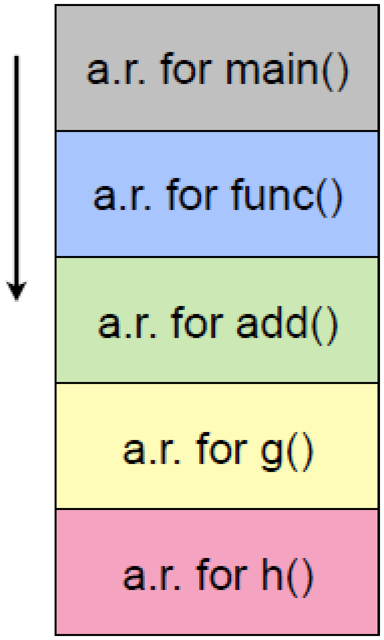
Sample Runtime Stack

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



Sample Runtime Stack

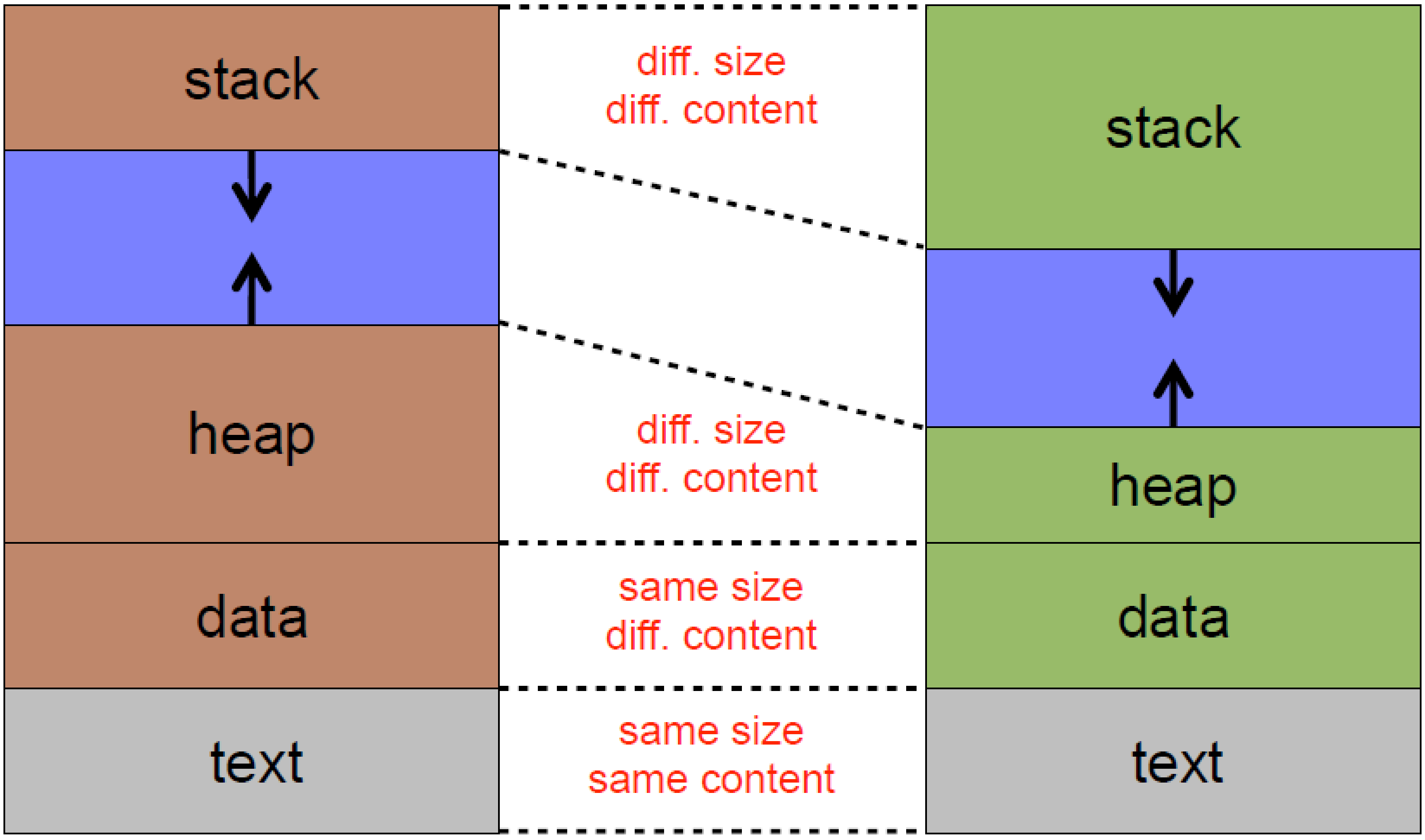
- g() calls h()



Runtime Stack Growth

- The mechanics for pushing/popping are more complex than one may think and pretty interesting (this should have been covered in System Level Programming)
- 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 infamous “runtime stack overflow” error
  - Causes a trap, that will trigger the Kernel to terminate your process with that error
  - Typically due to infinite recursion (but none of us have ever caused that to happen)

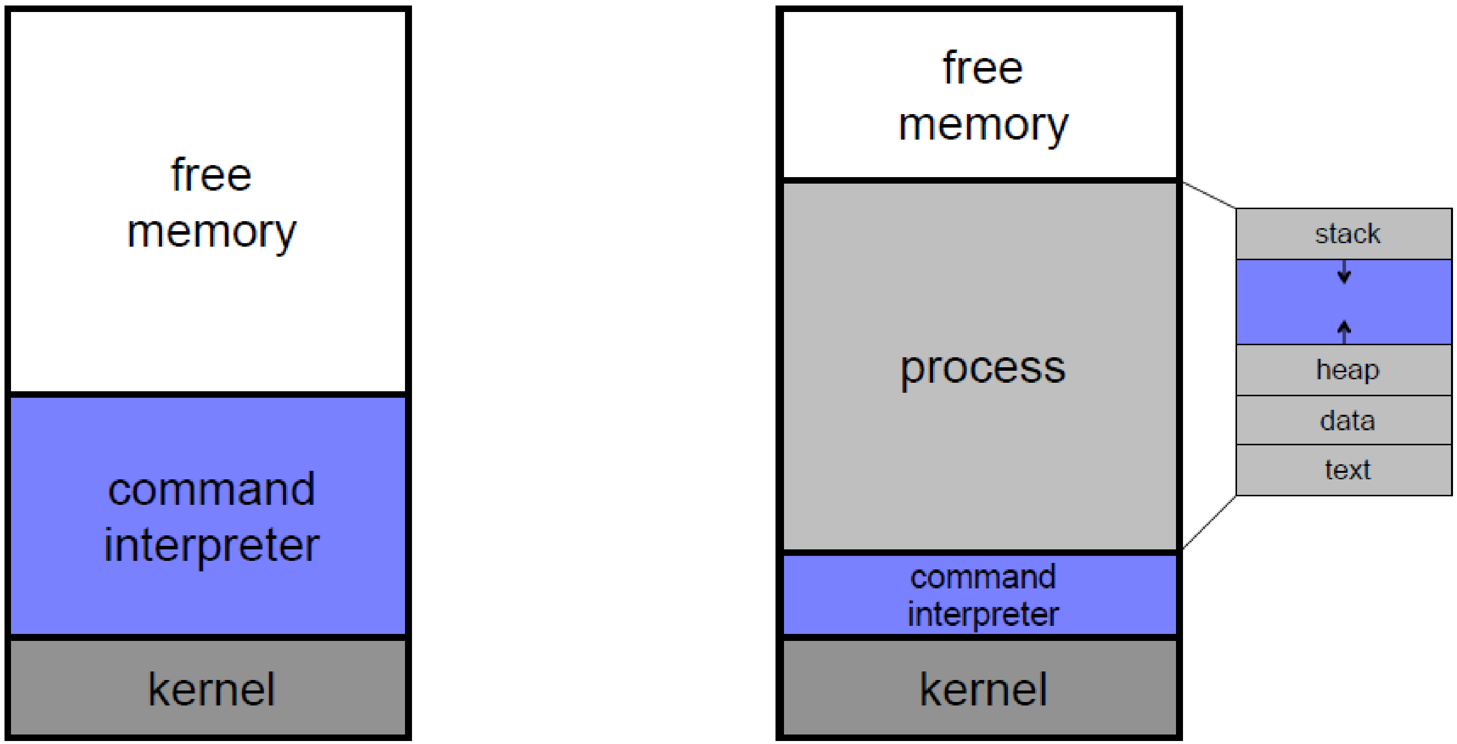
2 Processes for 1 Program



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

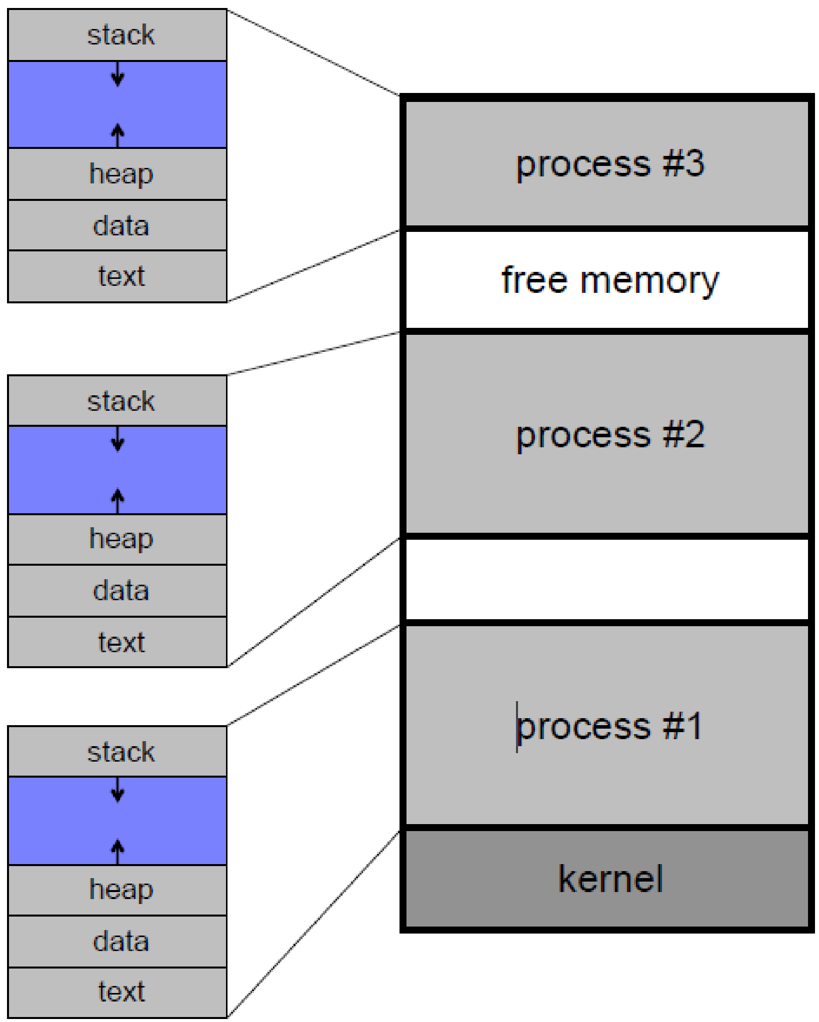
Single-Tasking with MS-DOS



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?

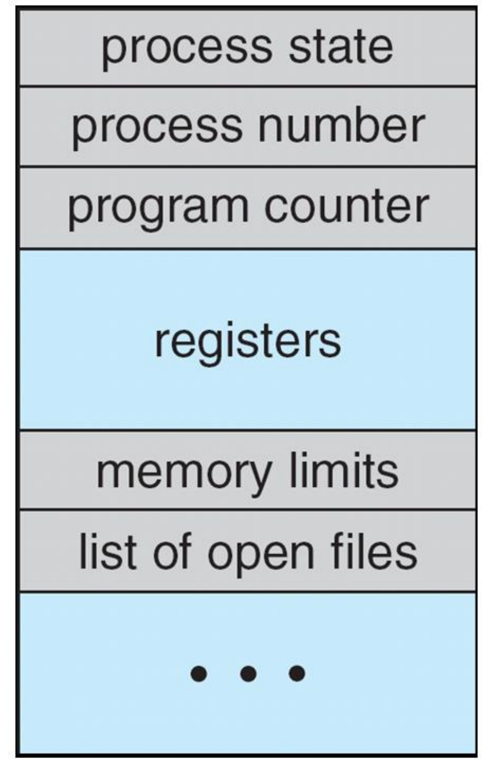
- Within the kernel, the code calls a series of functions
- Important: the kernel has a fixed-size stack
  - It is not very large (e.g. 4KB to 16KB)
- When writing kernel code, there is no such thing as allocating tons of temporary variables, or calling tons of nested functions each with tons of arguments
  - 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 does not have access to the standard C library!
  - 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 Control Block

- The OS keeps track of processes in a data structure, the process control block (PCB), which contains:
  - Process state
  - Process number (or Process ID: PID)
  - Per Thread
    - Program counter
    - CPU Registers
    - Stack
    - When saved, allow a process to be restarted later
  - CPU-scheduling info
    - Priority, queue, …(we will cover this soon)
  - Memory-management info
    - Base and limit registers, page table, …(covered later too)
  - Accounting info
    - Amount of resources used so far
  - I/O status info
    - List of I/O devices allocated to the process, open files, …
  - Child and Parent data

Process Control Block

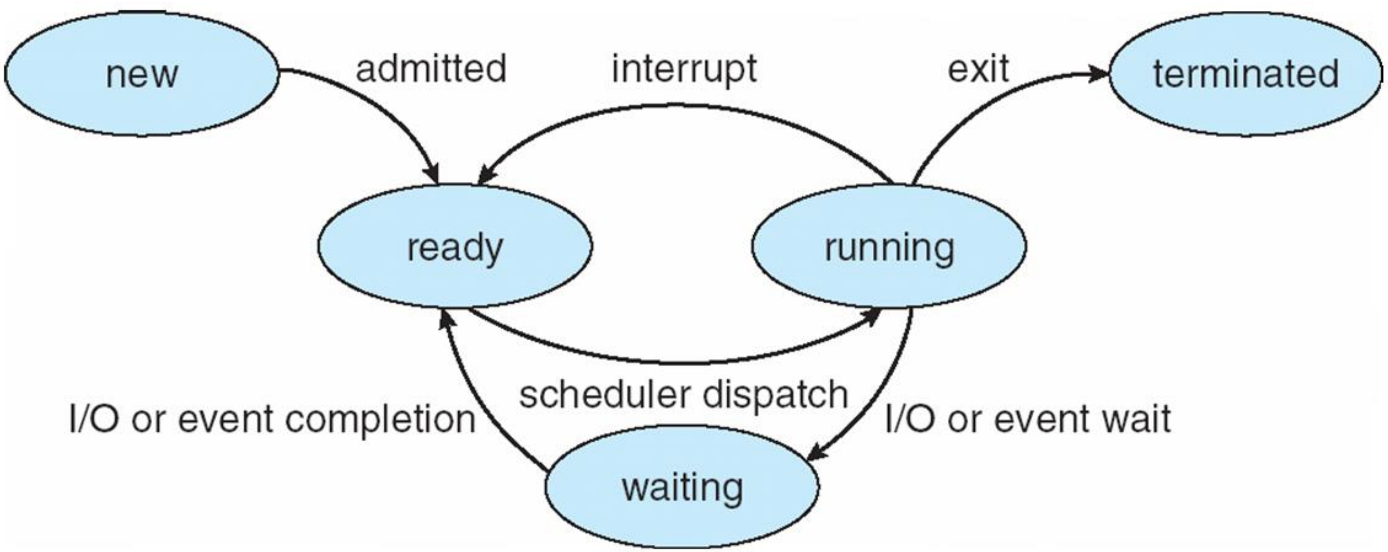
- Figure from book



- In reality, this is a bit messier
  - See page 110 in the text

Process States

- As a process executes, it changes state
  - New: The process is being created
  - Running: Instructions are being executed
  - Waiting: The process is waiting for some event to occur
  - Ready: The process is waiting to be assigned to a processor
  - Terminated: the process has finished execution



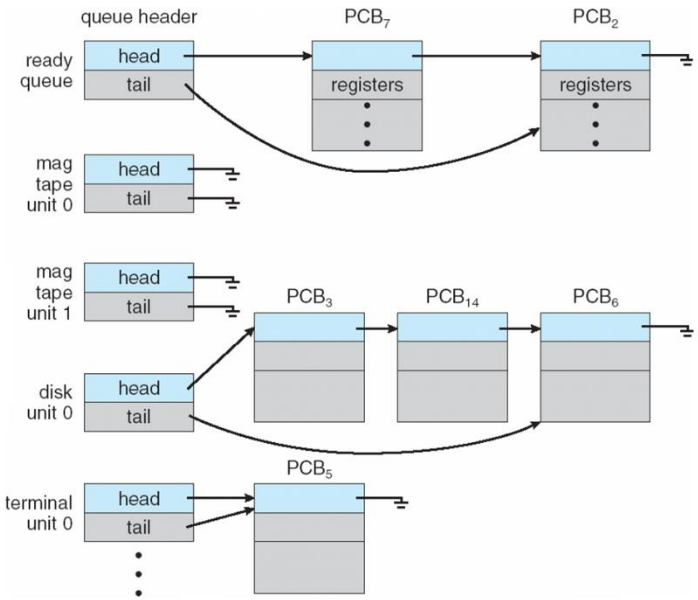
The Kernel’s “Process Table”

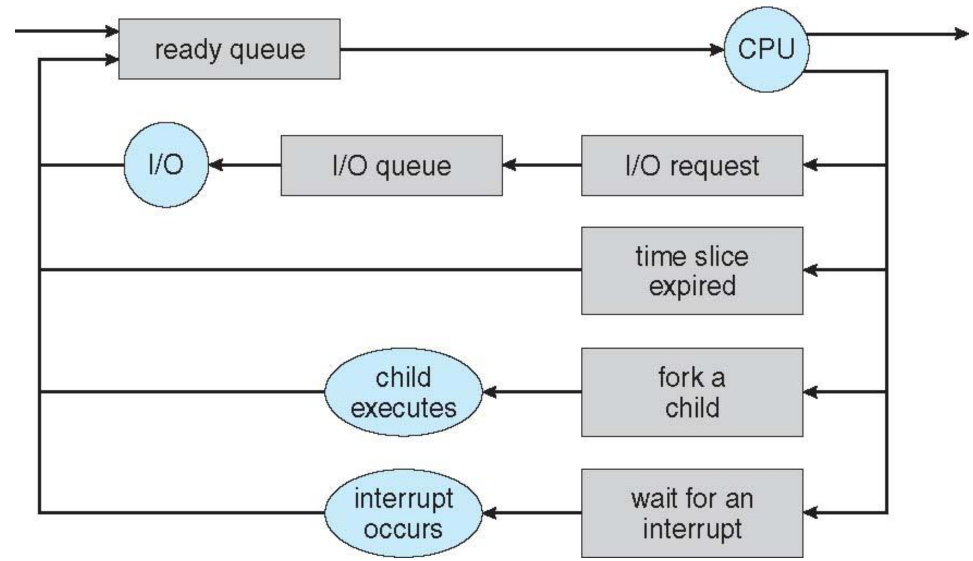
- The Kernel keeps around all the PCBs in its memory, in a data structure often called the Process Table
- Because Kernel size must be bounded, the Process Table size is also bounded
  - Based on a configuration parameter of the kernel, but you can’t set it to infinity (It is also set at kernel compile time so requires a recompile)
- 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 is very easy to write code that does this
  - Called a “fork bomb”

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 previews of future chapters
  - I will try to keep away from giving too many
- 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, in the “Ready” state (or “Waiting” etc.)
- The OS gives the CPU t a process for a limited amount of time, then gives it to another process, and so on

Process Management

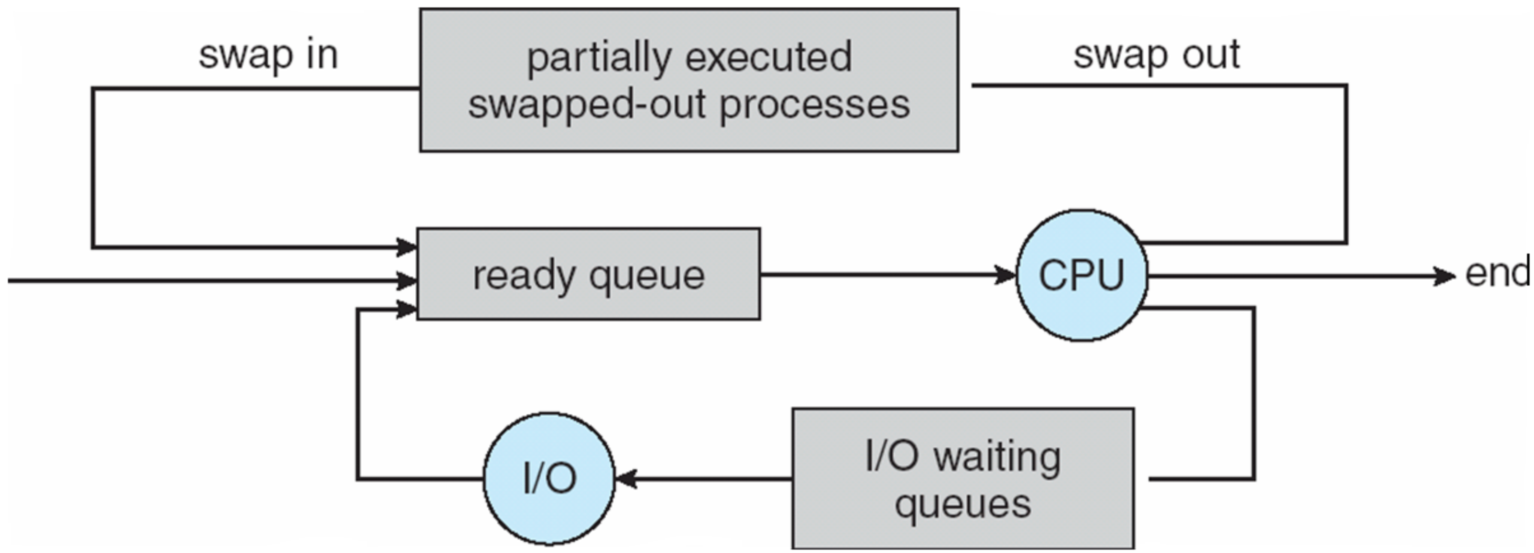




Goal: Mix of I/O bound (filling I/O queues) and CPU bound (filling ready queue)

- Job queue—All system processes
- Ready queue—All processes in memory and ready execute
- Device queue—All processes waiting for an I/O request completes
- Process migrate between the various queues depending on their state

Process Schedulers

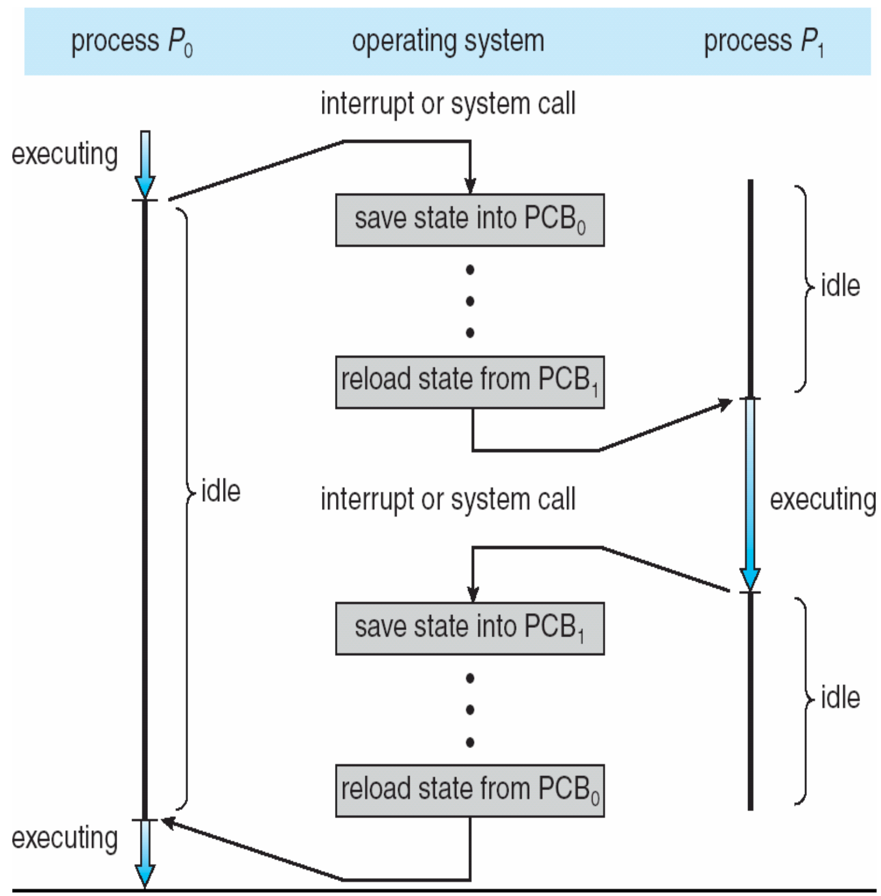


- Long-term (What multiprogramming degree?)
  - Slow; runs infrequently in the background, when processes terminate
- Medium-term (Which to swap?)
  - Efficient; runs roughly every second, when a process needs memory
- Short-Term (which to dispatch next?)
  - Must be fast; runs often (i.e., every 100 ms), after every interrupt

Context Switches

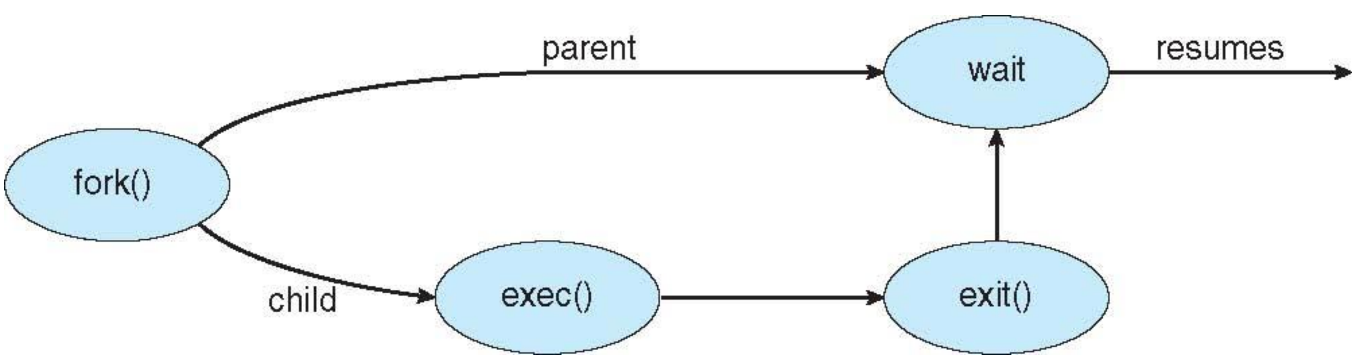
Taking control from one process and giving it to another

- Save the old process state and load the saved new process state
- Context-switch time
  - Pure overhead, no useful work done
  - Cost dependent on hardware support
    - e.g., save all registers in a single instruction
    - e.g., multiple register sets
- Time slicing gives CPU cycles in a round-robin manner
- Context switching is the *mechanism.* The *policy* is called scheduling

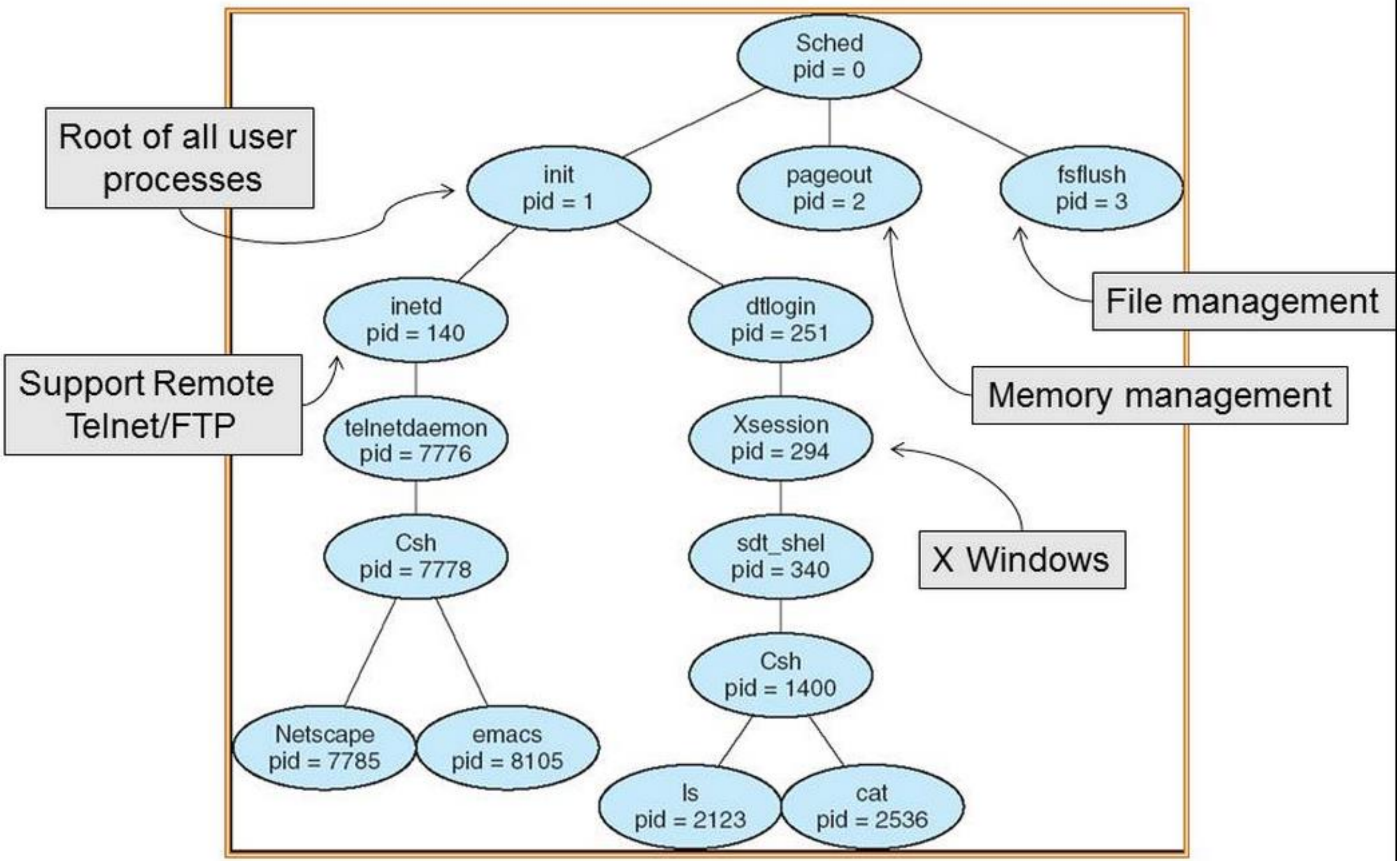


Process Creation

- Parent spawns children, children span others
- Resource sharing options
  - Parent and children share all resources
  - Children share some parent resources
  - No sharing
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space options
  - Child is a duplicate of the parent
  - Child space has a program loaded into it



A Solaris process spawning tree



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

```
pid=fork();
if(pid<){
        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, “Iam 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

fork() and Memory

- What does the following code print?

```
int a=12;
if(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);
}
```

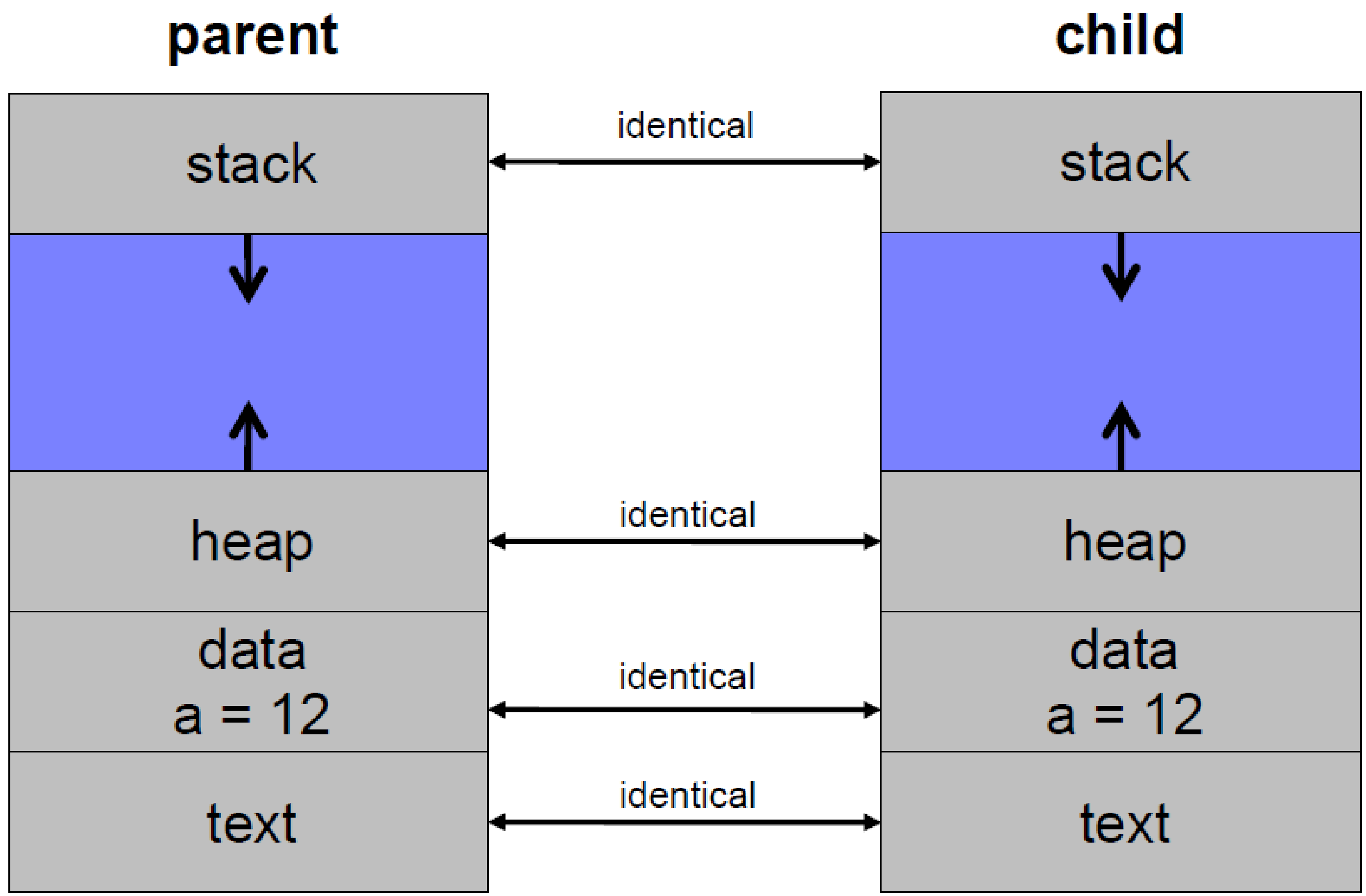
fork() and Memory

- What does the following code print?

```
int a=12;
if(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);
}
```

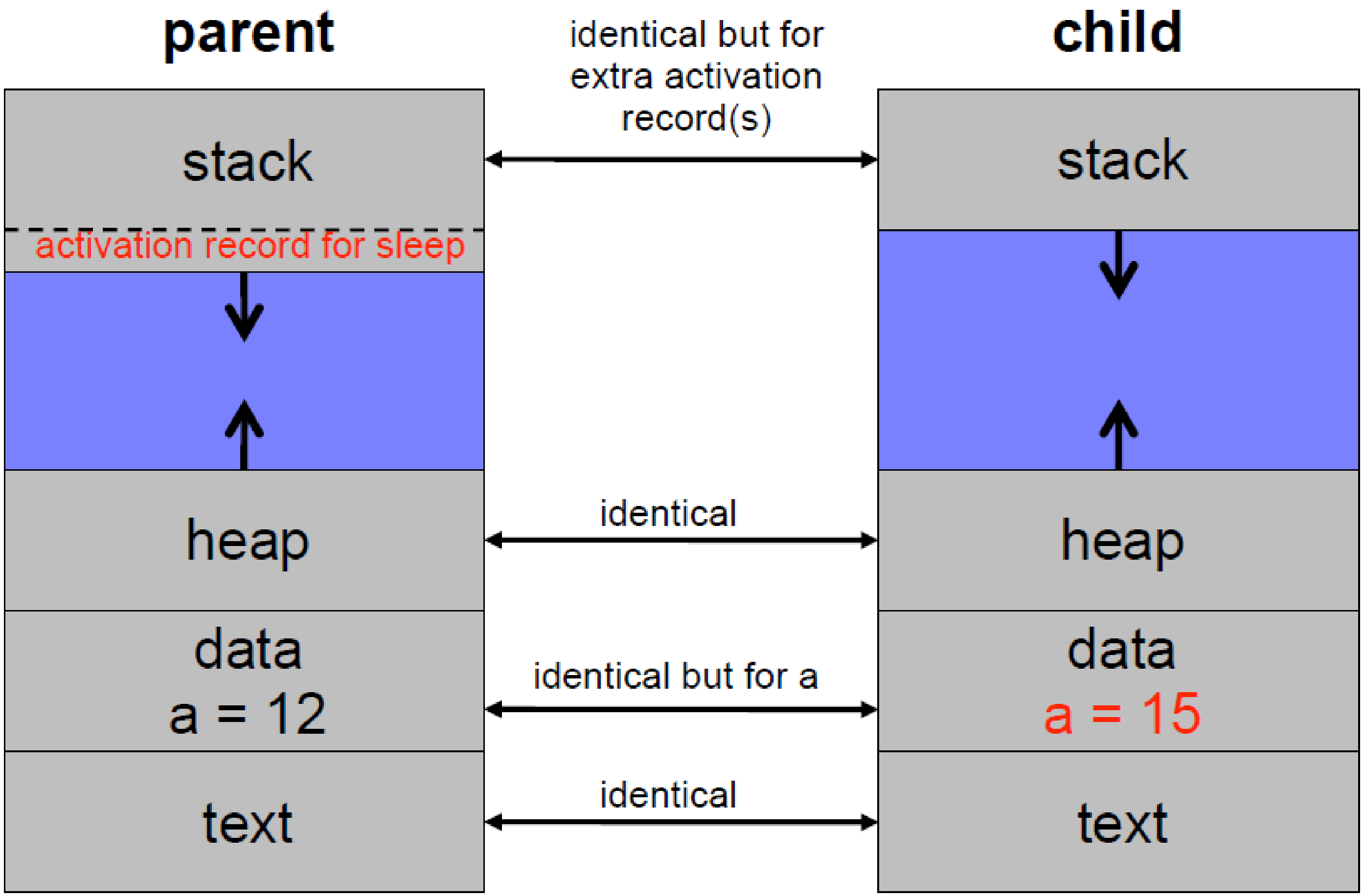
Answer: 12

fork() and Memory



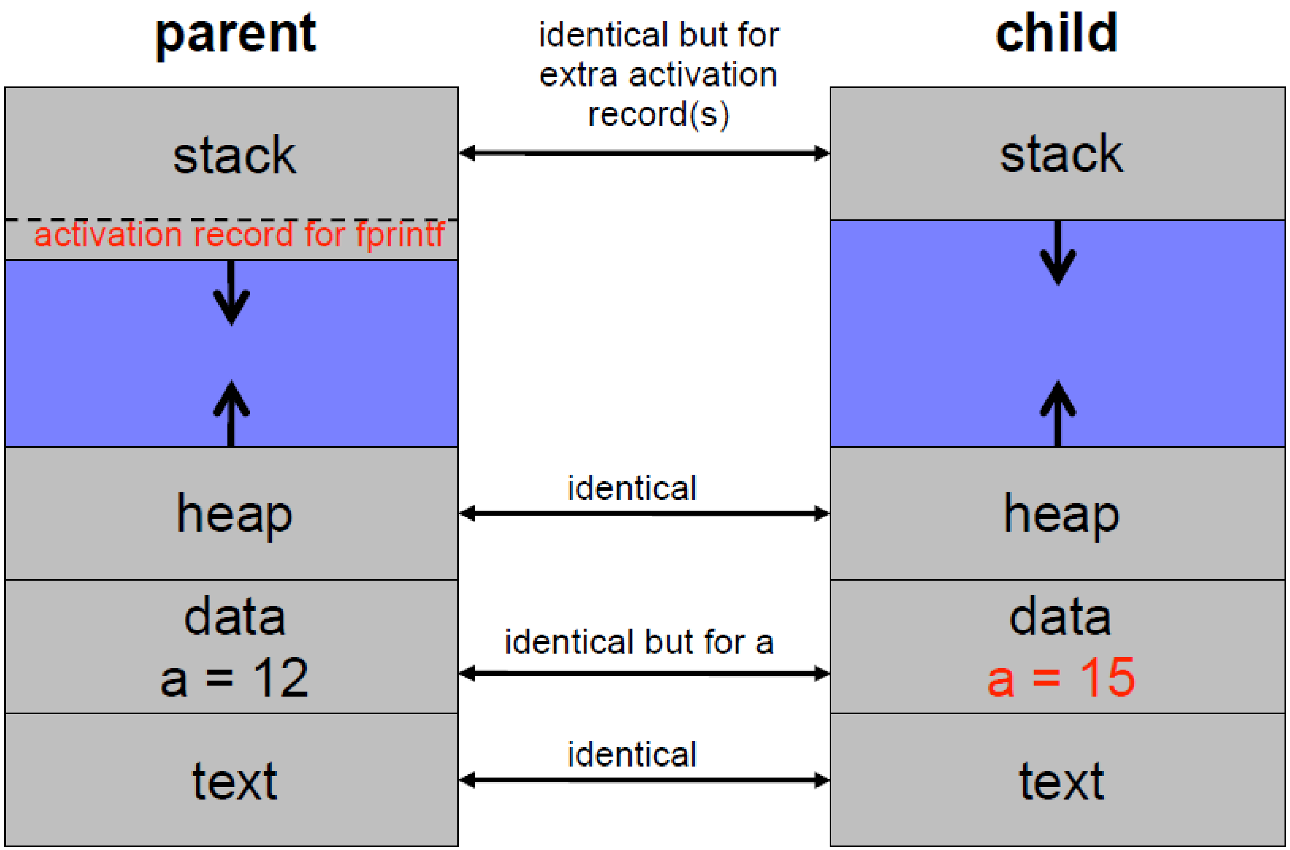
State of both processes right after fork() completes

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 amny times does this code print “hello”?

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

fork() can be confusing

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

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

Answer: 6 times

Popular Homework Question

- How many processes does this C program create?

```
int main(int argc, char* arg[])
{
        fork();
        if(fork()){
                fork();
        }
        fork();
}
```

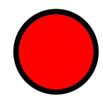
Typical C coding style: call fork() and if its return value is non-zero, do the if clause

- Let's see how to do this

Popular Homework Question

- How many processes does this C program create?

```
int main(int argc, char* arg[])
{
        fork();
        if(fork()){
                fork();
        }
        fork();
}
```

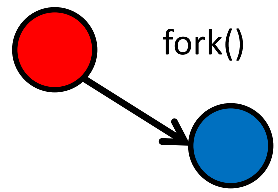


original process right when main begins

Popular Homework Question

- How many processes does this C program create?

```
int main(int argc, char* arg[])
{
        fork();
        if(fork()){
                fork();
        }
        fork();
}
```



Call to fork() creates a copy of the original process

Popular Homework Question

- How many processes does this C program create?

```
int main(int argc, char* arg[])
{
        fork();
        if(fork()){
                fork();
        }
        fork();
}
```

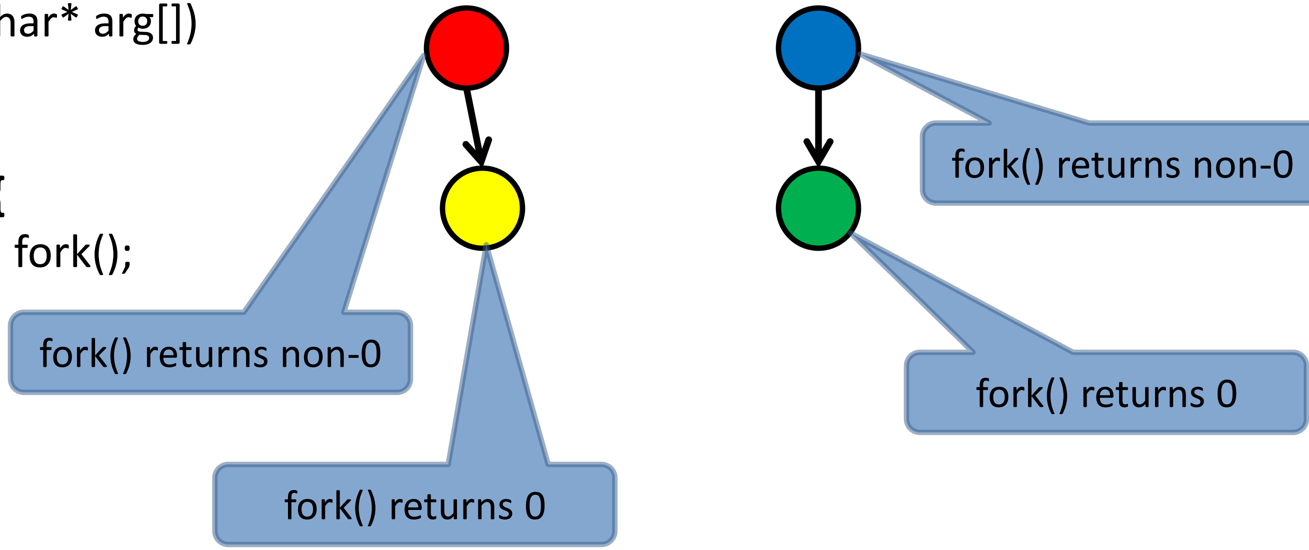
This code calls fork()

We now have two independent processes, each about to execute the same code

Popular Homework Question

- How many processes does this C program create?

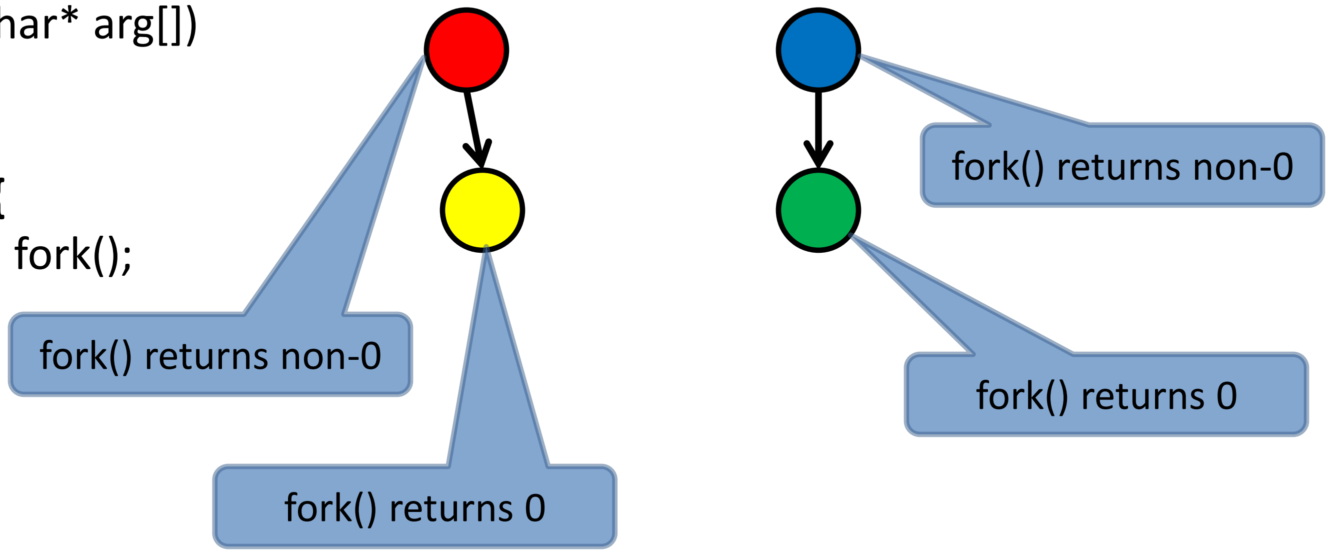
int main(int argc, char* arg[])
{
        ~~fork();~~
        if(fork()){
                fork();
        }
        fork();
}



Popular Homework Question

- How many processes does this C program create?

int main(int argc, char* arg[])
{
    ~~fork();~~
    if(fork()){
        fork();
    }
    fork();
}

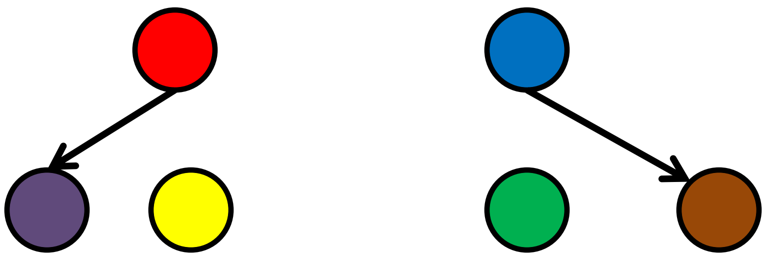


Yellow and green: do not go into the if clause, red and blue do

Popular Homework Question

- How many processes does this C program create?

```
int main(int argc, char* arg[])
{
        fork();
        if(fork()){
                fork();
        }
        fork();
}
```

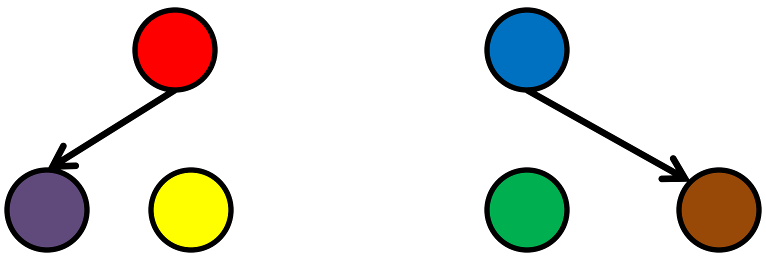


Red and blue each create a new child process (purple and brownish)

Popular Homework Question

- How many processes does this C program create?

```
int main(int argc, char* arg[])
{
        ~~fork();~~
        if(fork()){
                fork();
        }
        fork();
}
```



All processes execute the last call to fork().
Red, purple, blue, and brown after they exit the if clause.
Yellow and green after they skip the if clause.
We have 6 processes calling fork(), each creating a new process.
So, we have a total of 12 processes at the end, one of which was the original process

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

Process Terminations

- A process terminates itself with the exit() system call
  - This call takes an integer argument 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, etc.
- 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
- Lets look at wait_example1 and wait_example2
- 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 signal"
- Signals happen for various reasons
  - ctl+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

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

void handler(int sig){
        fprintf(stdout, “I do what I want!\n”);
        return;
}

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

Zombies

- 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”
- See zombie_example
- A zombie lingers until:
  - Its parent has called wait() for the child, or
  - Its parent dies
- It is bad practice to leave zombies just laying around unnecessarily

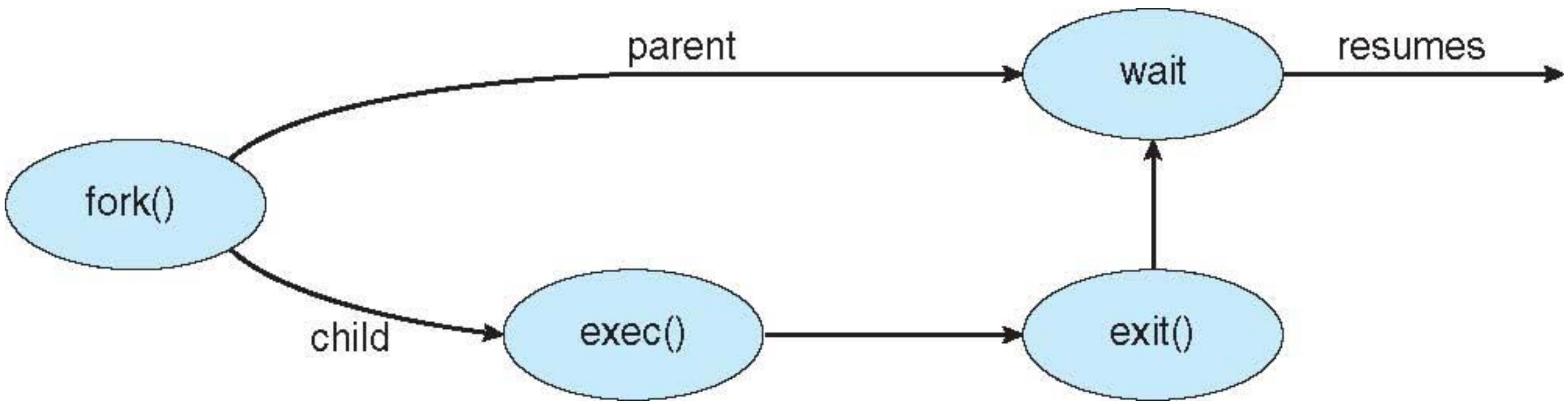
Getting rid of zombies

- 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

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 Linux system
  - launched on 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 example)
- 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
  - orphan_example1
  - orphan_example2

In a Nutshell



What about Windows?

- See example in Figure 3.11
- In Windows, the CreateProcess() call combines fork() and exec()
- In Win32 fashion, calls have many arguments
- There is an equivalent to wait(): WaitForSingleObject()
- TerminateProcess() is like kill()
- So, overall, it allows the same capabilities (which shouldn’t be surprising), but with a different flavor
  - Developers can be really opinionated about this

Processes in Java

- What about Java processes?
- The JVM doesn’t implement a Process abstraction, meaning that there is no notion of running multiple processes within the JVM (threads are supported which we will talk about later)
  - Partly because supporting several independent address spaces in the JVM is a pain
- It 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 provide 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

- Don’t forget about your Homework!!
- More will be posted on Friday, we don’t slow down any because this is a shortened semester!