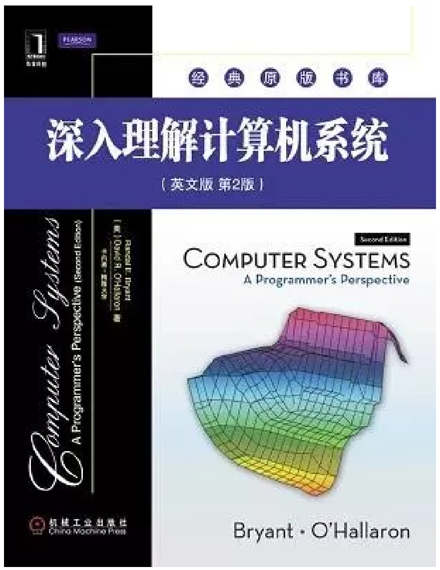
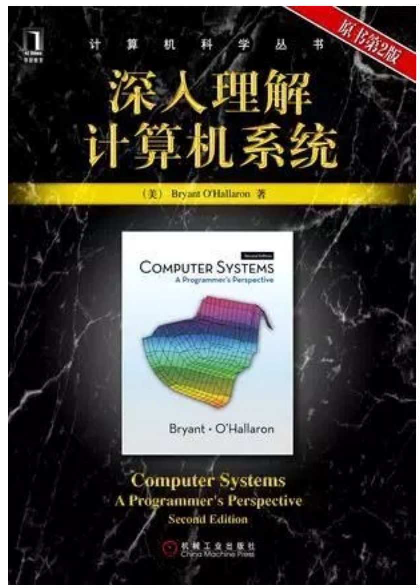
Processes

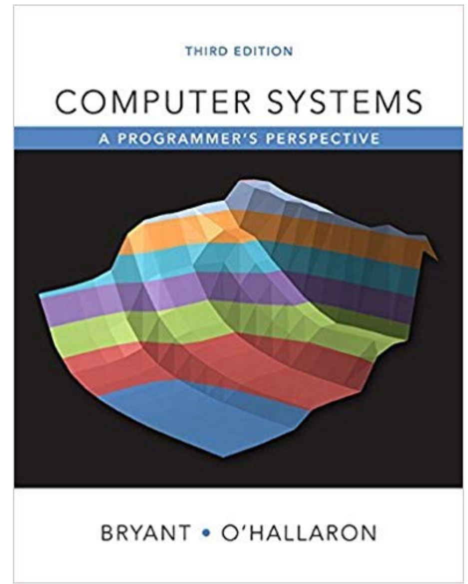
Operating Systems
Wenbo Shen

Some slides are from OSC and Henri Casanova@Hawaii

Recommendation







Processes

- Process Concept
- Process Control Block
- Process State
  - Process Creation
  - Process Termination
  - Process and Signal
- Process Scheduling

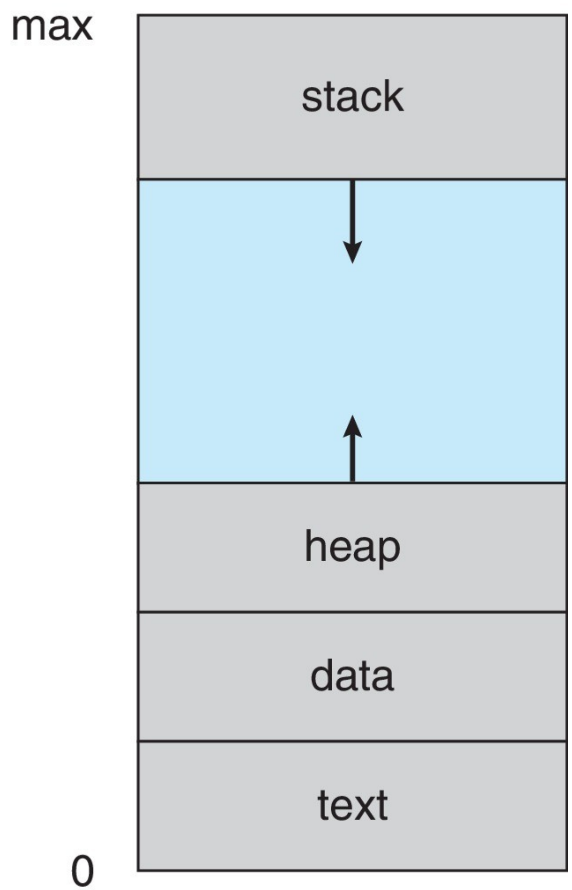
Process Concept

- How to use computer resources, such as CPU, memory
- A process is a program in execution (execution unit)
  - 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 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, user processes
- "job" and "process" are used interchangeably in OS texts

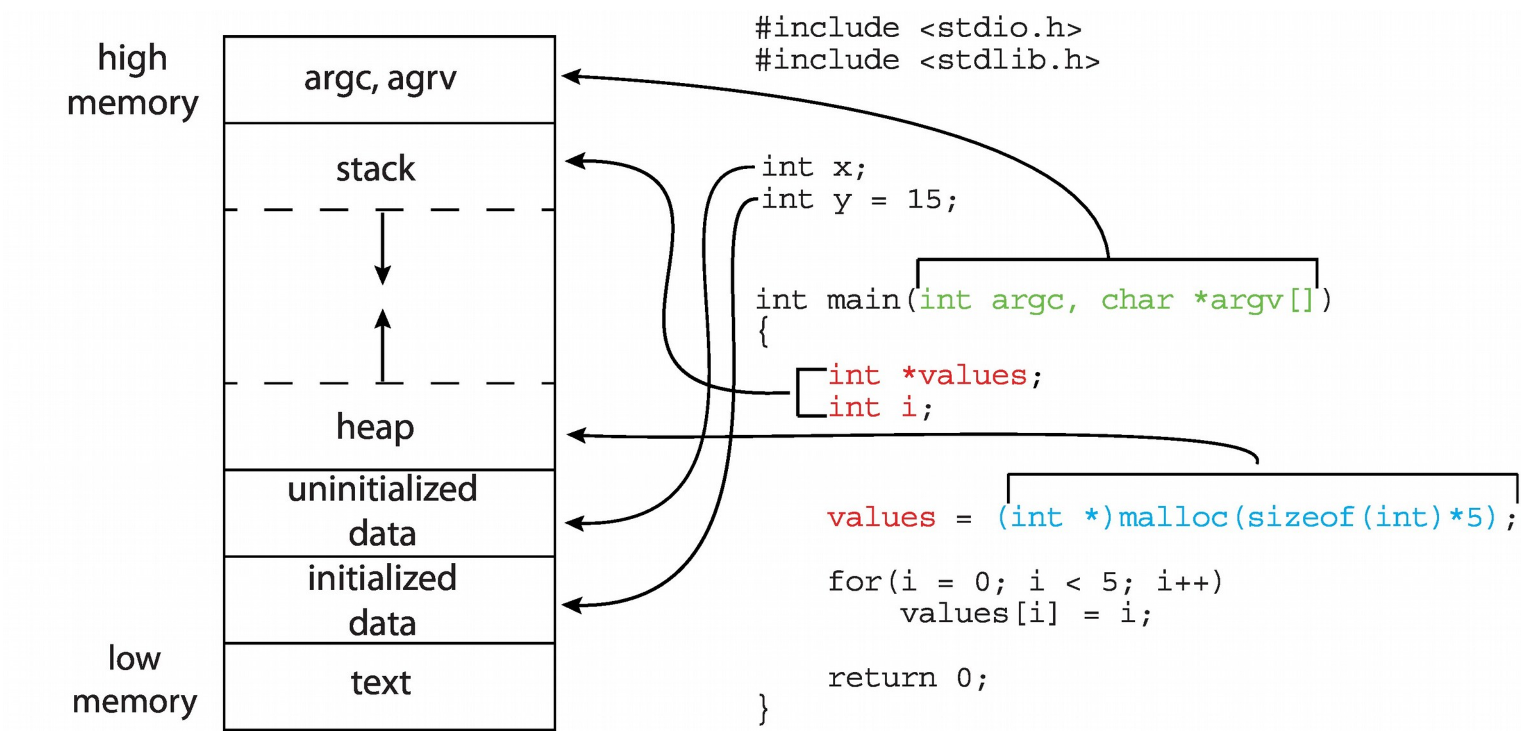
Process Concept

- 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



Memory Layout of a C Program



Process Address Space

```
7ffc75a5f000-7ffc75a80000 rw-p 00000000 00:00 0                          [stack]
7ffc75aa7000-7ffc75aaa000 r--p 00000000 00:00 0                          [vvar]
7ffc75aaa000-7ffc75aac000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
wenbo@parallels:~$ which cat
/bin/cat
wenbo@parallels:~$ file /bin/cat
/bin/cat: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l,
 for GNU/Linux 3.2.0, BuildID[sha1]=747e524bc20d33ce25ed4aea108e3025e5c3b78f, stripped
wenbo@parallels:~$ cat /proc/self/maps
55b793b79000-55b793b81000 r-xp 00000000 08:01 1048601                    /bin/cat
55b793d80000-55b793d81000 r--p 00007000 08:01 1048601                    /bin/cat
55b793d81000-55b793d82000 rw-p 00008000 08:01 1048601                    /bin/cat
55b794d33000-55b794d54000 rw-p 00000000 00:00 0                          [heap]
7f1974b90000-7f197555f000 r--p 00000000 08:01 662494                     /usr/lib/locale/locale-archive
7f197555f000-7f1975746000 r-xp 00000000 08:01 267596                     /lib/x86_64-linux-gnu/libc-2.27.so
7f1975746000-7f1975946000 ---p 001e7000 08:01 267596                     /lib/x86_64-linux-gnu/libc-2.27.so
7f1975946000-7f197594a000 r--p 001e7000 08:01 267596                     /lib/x86_64-linux-gnu/libc-2.27.so
7f197594a000-7f197594c000 rw-p 001eb000 08:01 267596                     /lib/x86_64-linux-gnu/libc-2.27.so
7f197594c000-7f1975950000 rw-p 00000000 00:00 0
7f1975950000-7f1975977000 r-xp 00000000 08:01 267568                     /lib/x86_64-linux-gnu/ld-2.27.so
7f1975b3c000-7f1975b60000 rw-p 00000000 00:00 0
7f1975b77000-7f1975b78000 r--p 00027000 08:01 267568                     /lib/x86_64-linux-gnu/ld-2.27.so
7f1975b78000-7f1975b79000 rw-p 00028000 08:01 267568                     /lib/x86_64-linux-gnu/ld-2.27.so
7f1975b79000-7f1975b7a000 rw-p 00000000 00:00 0
7ffc73010000-7ffc73031000 rw-p 00000000 00:00 0                          [stack]
7ffc73148000-7ffc7314b000 r--p 00000000 00:00 0                          [vvar]
7ffc7314b000-7ffc7314d000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
wenbo@parallels:~$
```

The Stack

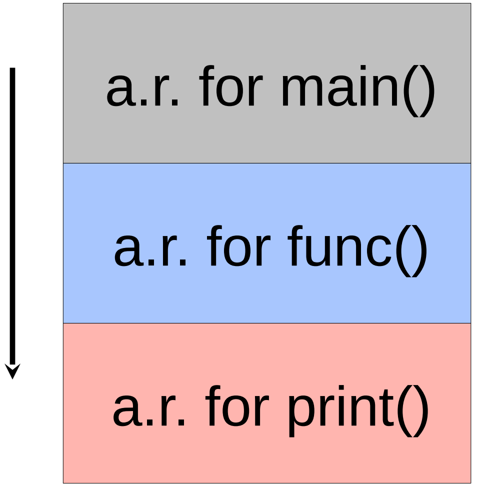
- The runtime stack is
  - A stack on which items can be pushed or popped
  - The items are called activation records
  - The stack is how we manage to have programs place successive function/method calls
  - The management of the stack is done entirely on your behalf by the compiler
- An activation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

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

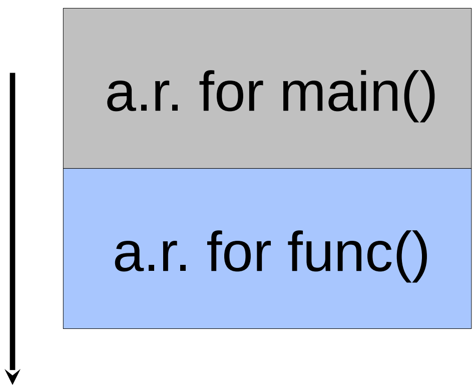
Simple Runtime Stack

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



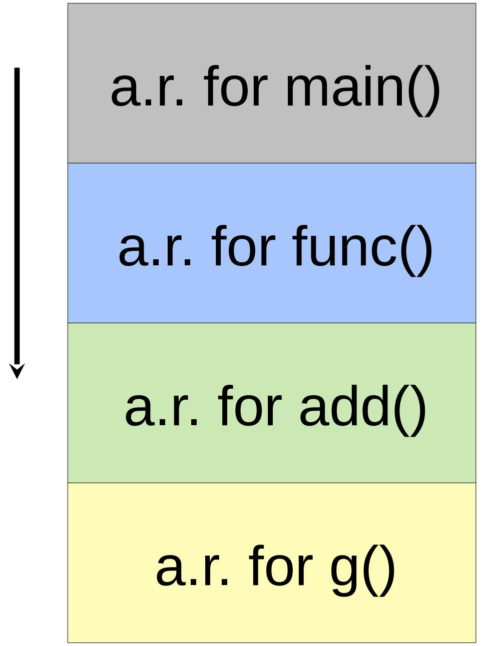
Simple Runtime Stack

- print() returns



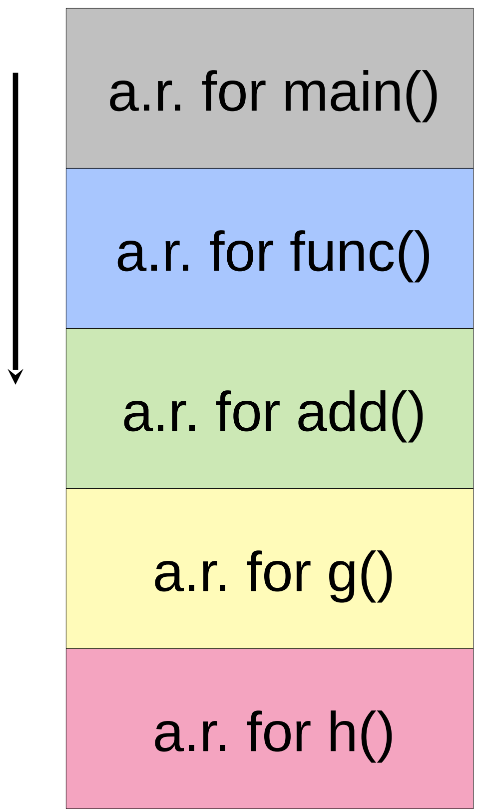
Simple Runtime Stack

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



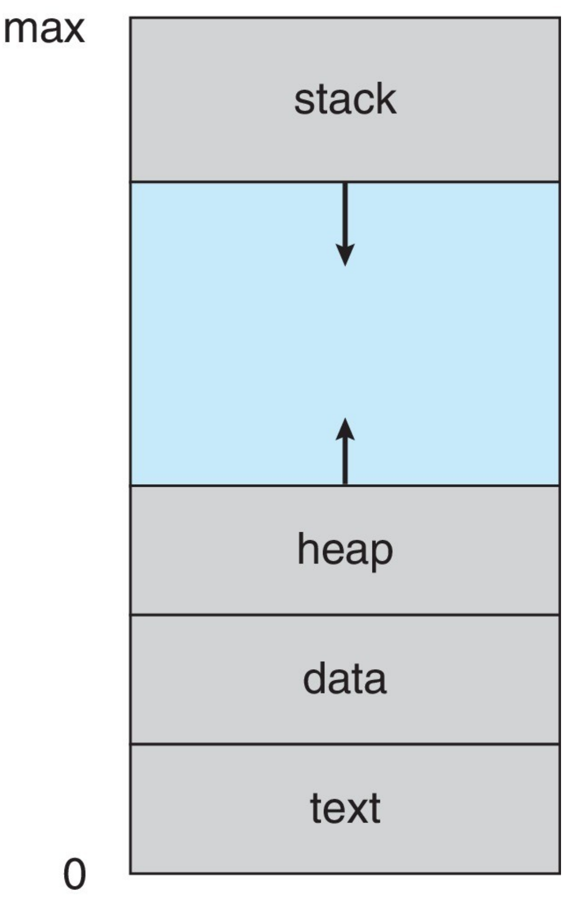
Simple Runtime Stack

- g() calls h()

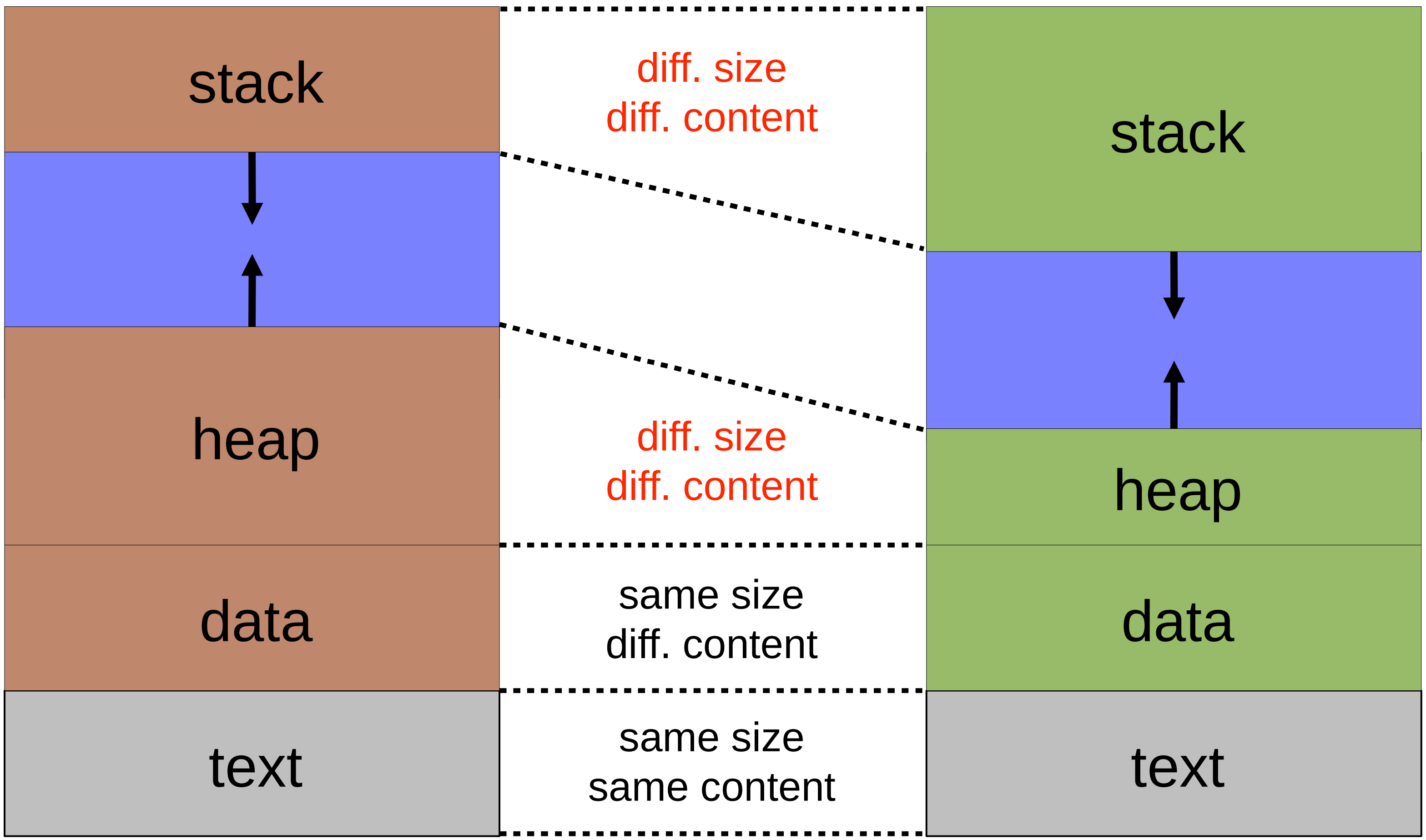


Runtime Stack Growth

- The mechanics for pushing/popping are more complex than one may think and pretty interesting
- 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

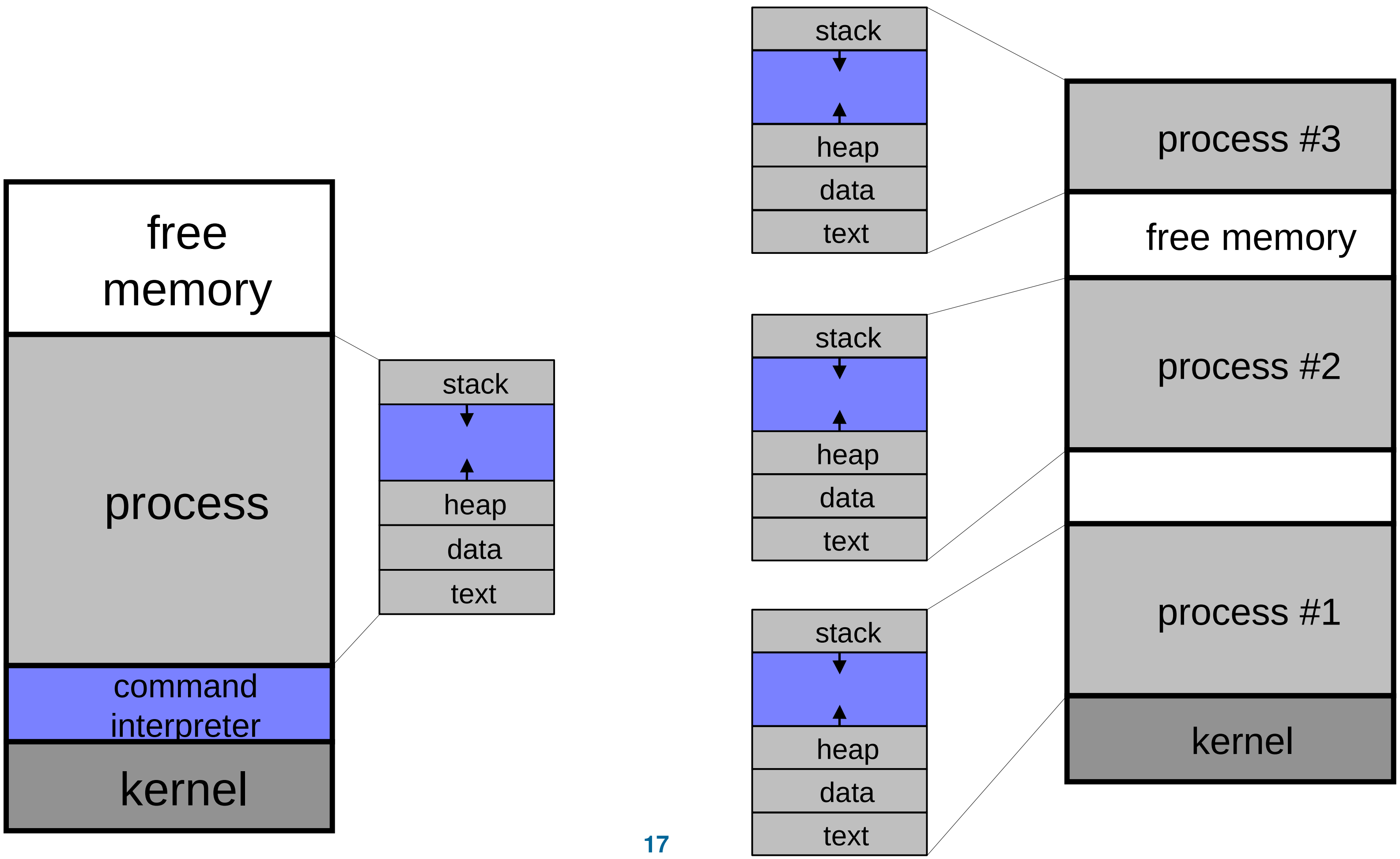


2 processes for the same program



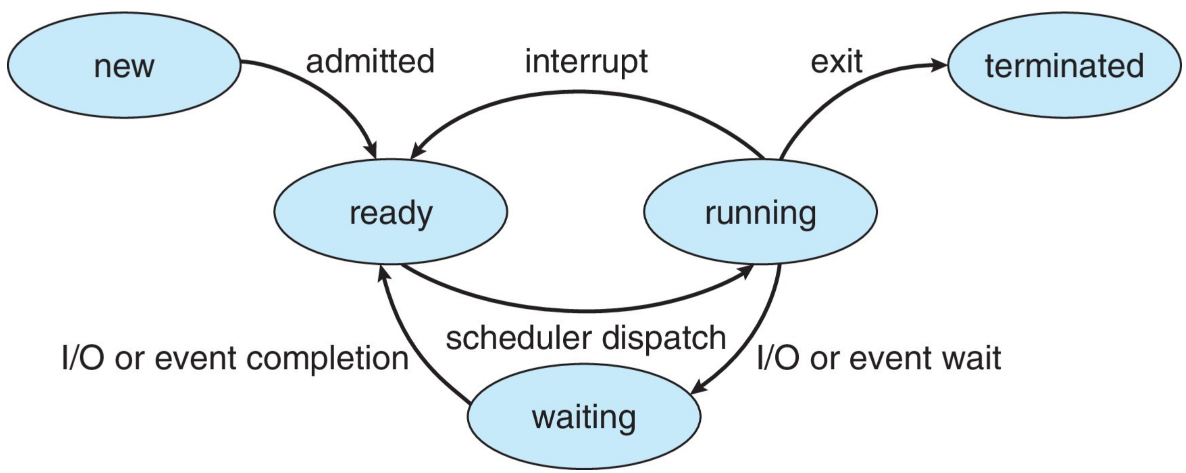
Single- and Multi-Tasking

- OSes used to be single-tasking
- Modern OSes support multi-tasking
  - To start a new program, the OS simply creates a new process (via a system-call called fork() on a UNIX system)



Process State

- 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

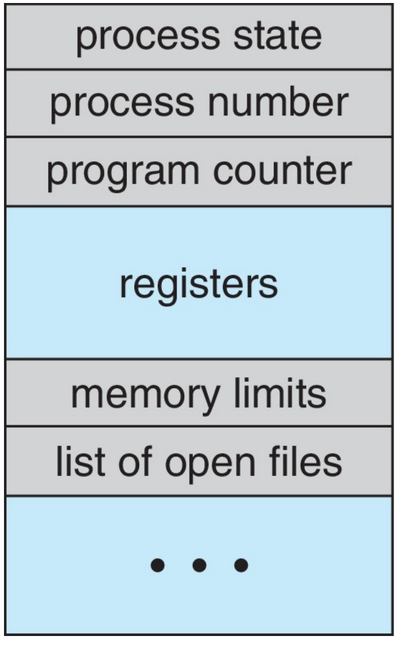


Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Process Representation in Linux

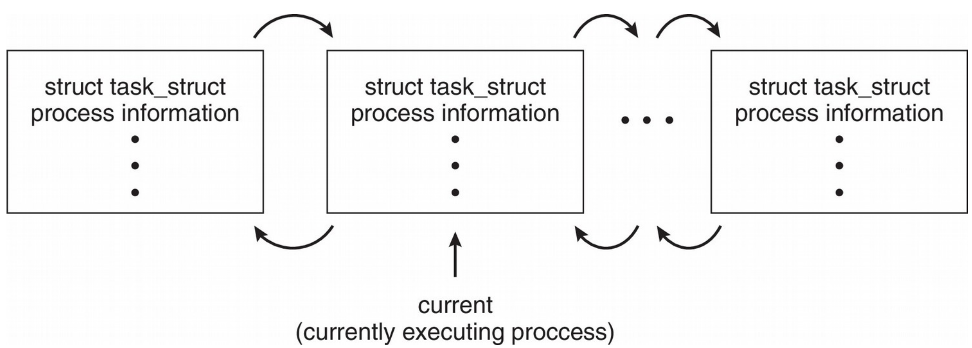
Represented by the C structure `task_struct` (https://elixir.bootlin.com/linux/v5.3.1/source/include/linux/sched.h#L637)

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	/* -1 unrunnable, 0 runnable, >0 stopped: */
	volatile long			state;

	/*
	 * This begins the randomizable portion of task_struct.
	 * scheduling-critical items should be added above here.
	 */
	randomized_struct_fields_start

	void				*stack;
	refcount_t			usage;
```

Process Representation in Linux

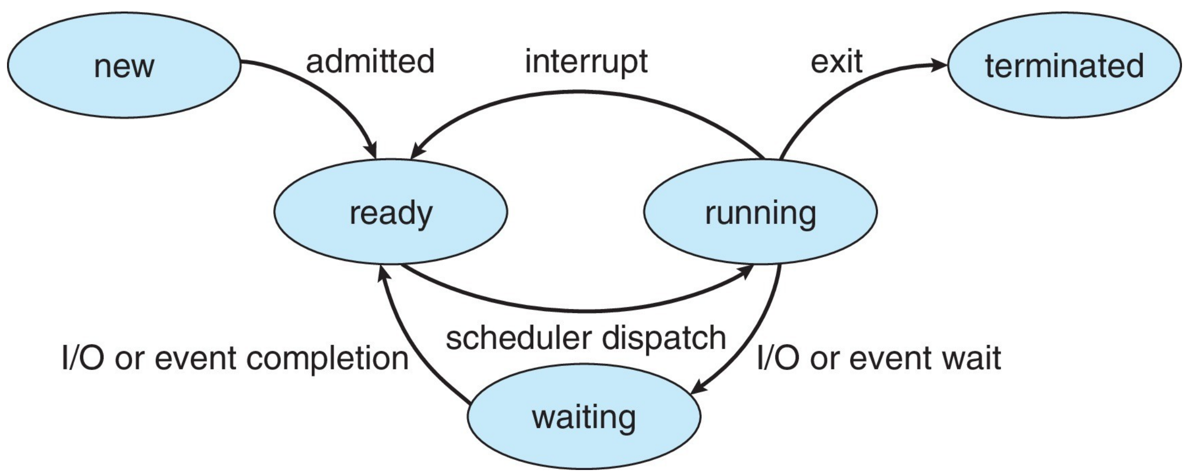
Represented by the C structure `task_struct` (https://elixir.bootlin.com/linux/v5.3.1/source/include/linux/sched.h#L637)

```
pid t_pid;          /* process identifier */
long state;             /* state of the process */
unsigned int time_slice   /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;  /* address space of this process */
```



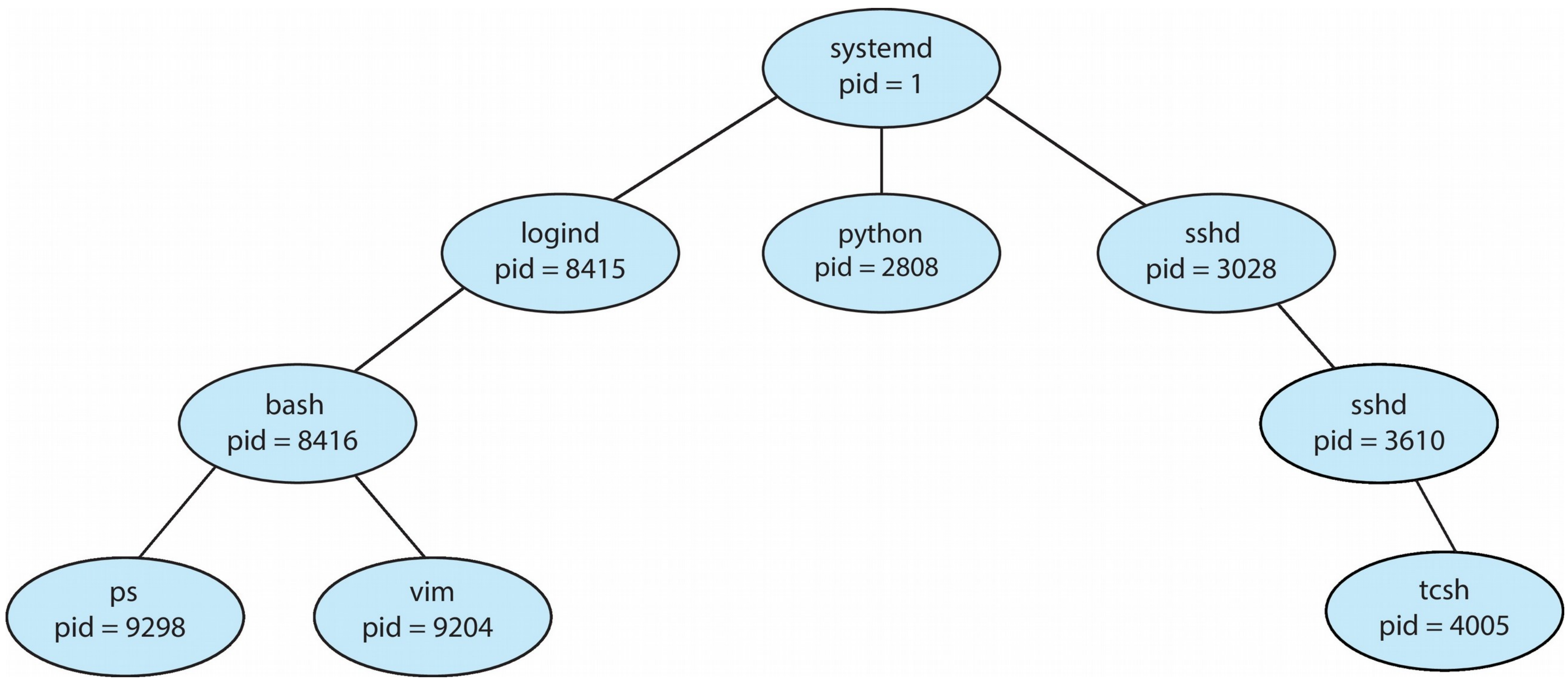
Process State

- 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



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



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 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 < 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

fork() and Memory

- What does the following code print?

```
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);
 }
```

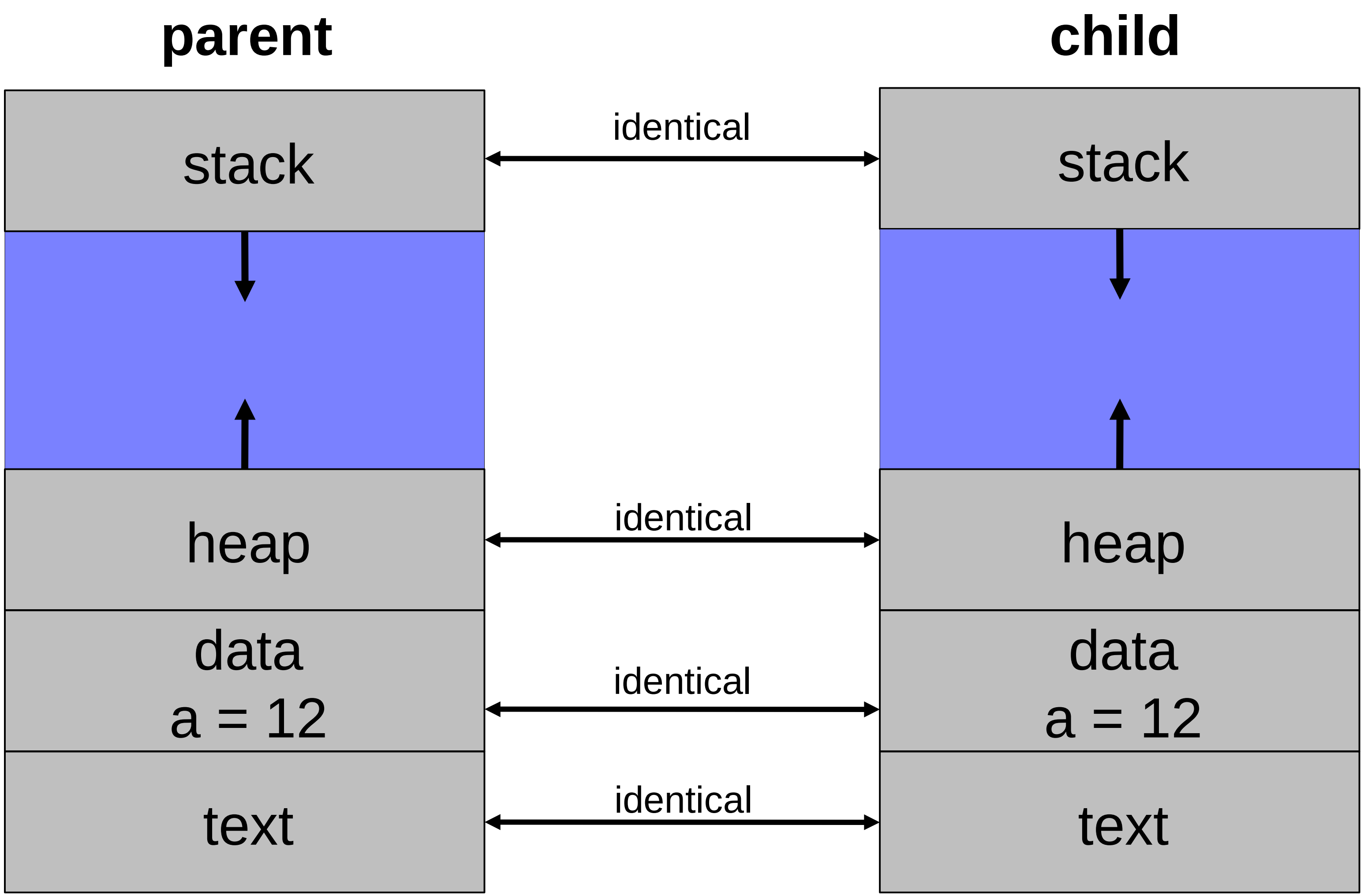
fork() and Memory

- What does the following code print?

```
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);
 }
```

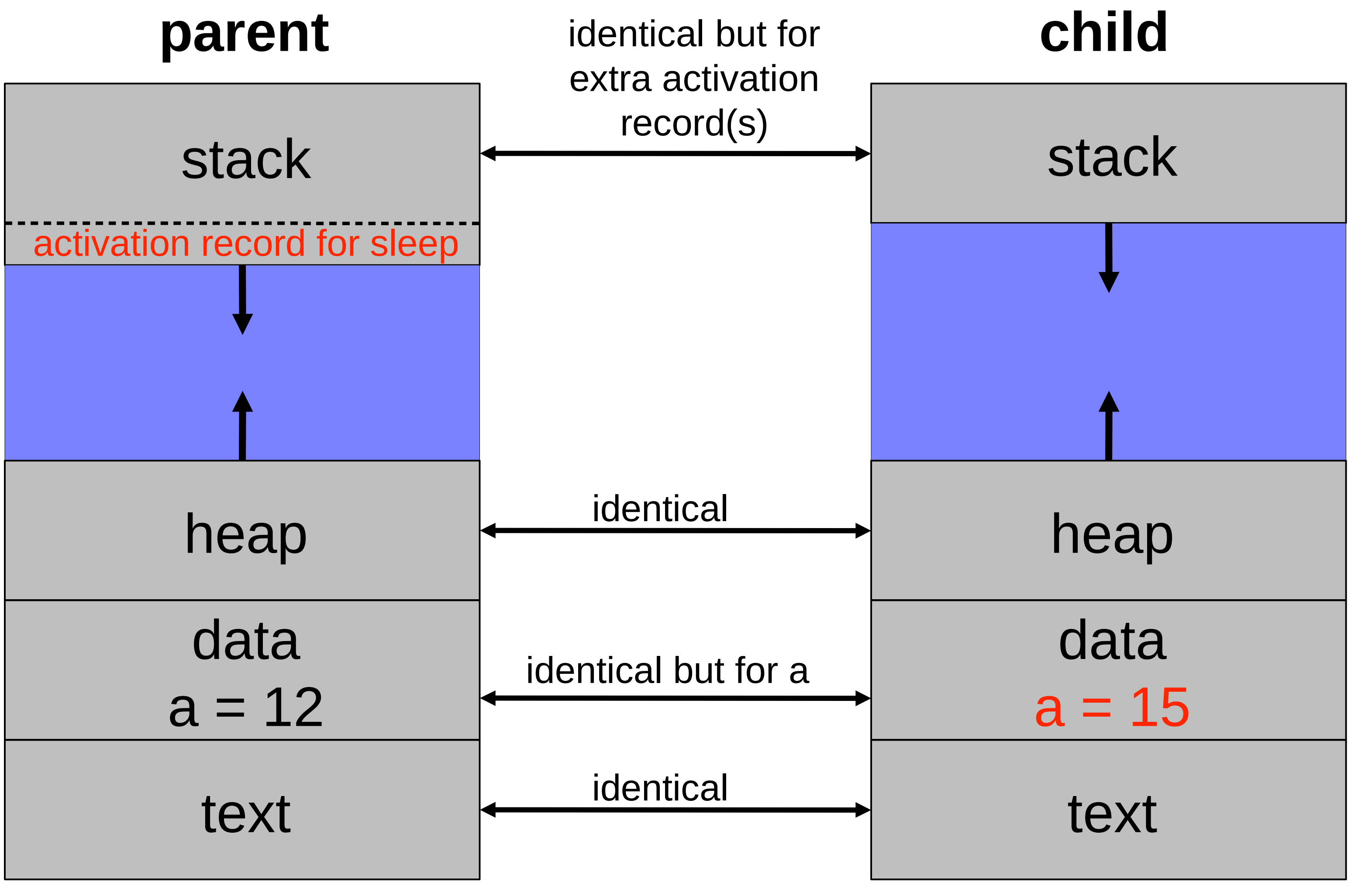
Answer: 12

fork() and Memory



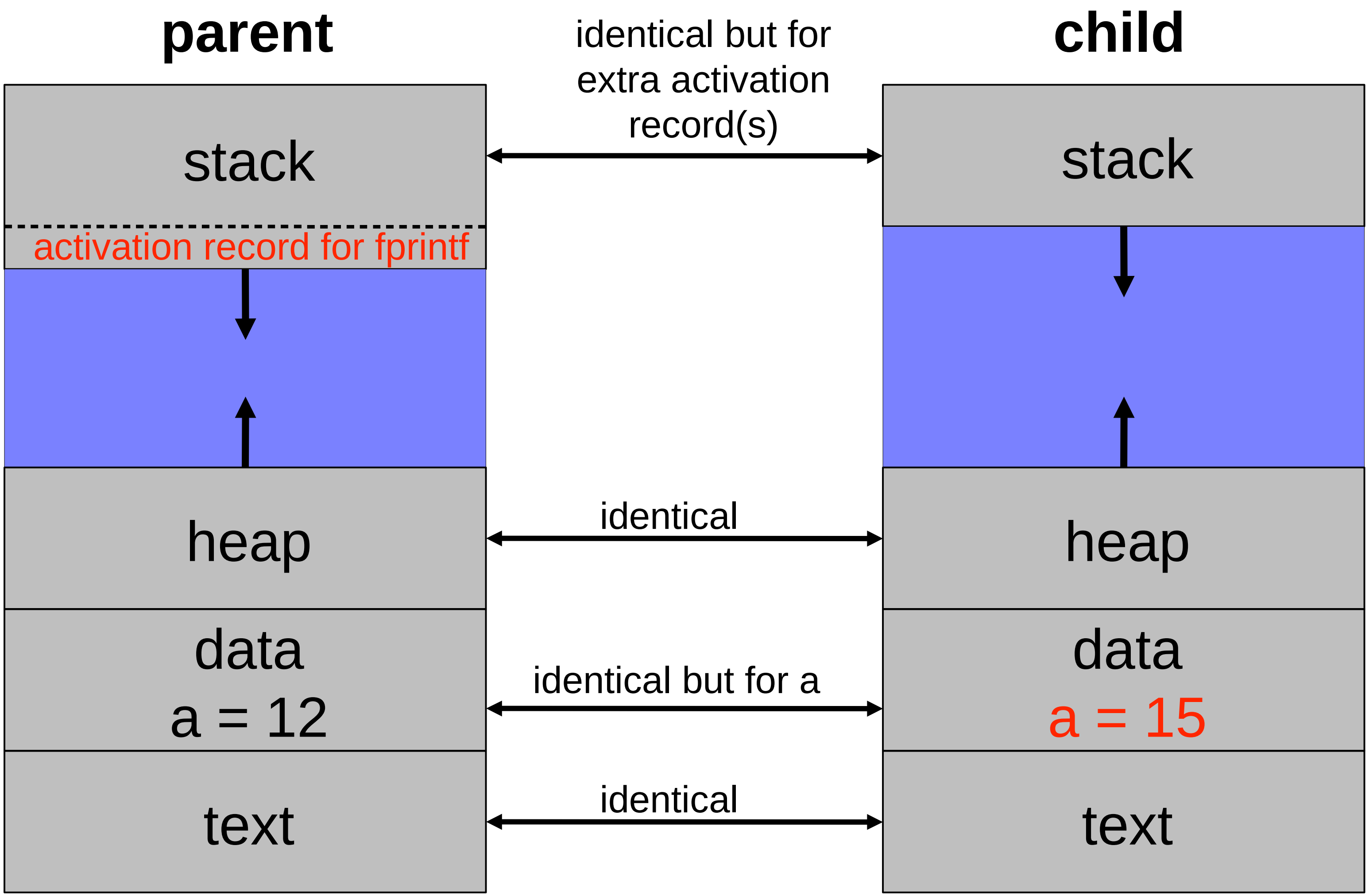
State of both processes right after fork() completes

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() and Memory

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

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

fork() and Memory

- How many 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?

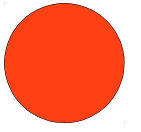
```
int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    }
    fork ();
}
```

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

Popular Homework Question

- How many processes does this C program create?

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

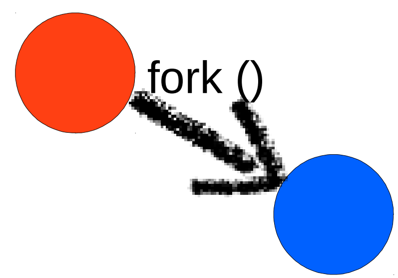


original process right when main begins

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 ();
}
```

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

Popular Homework Question

- How many processes does this C program create?

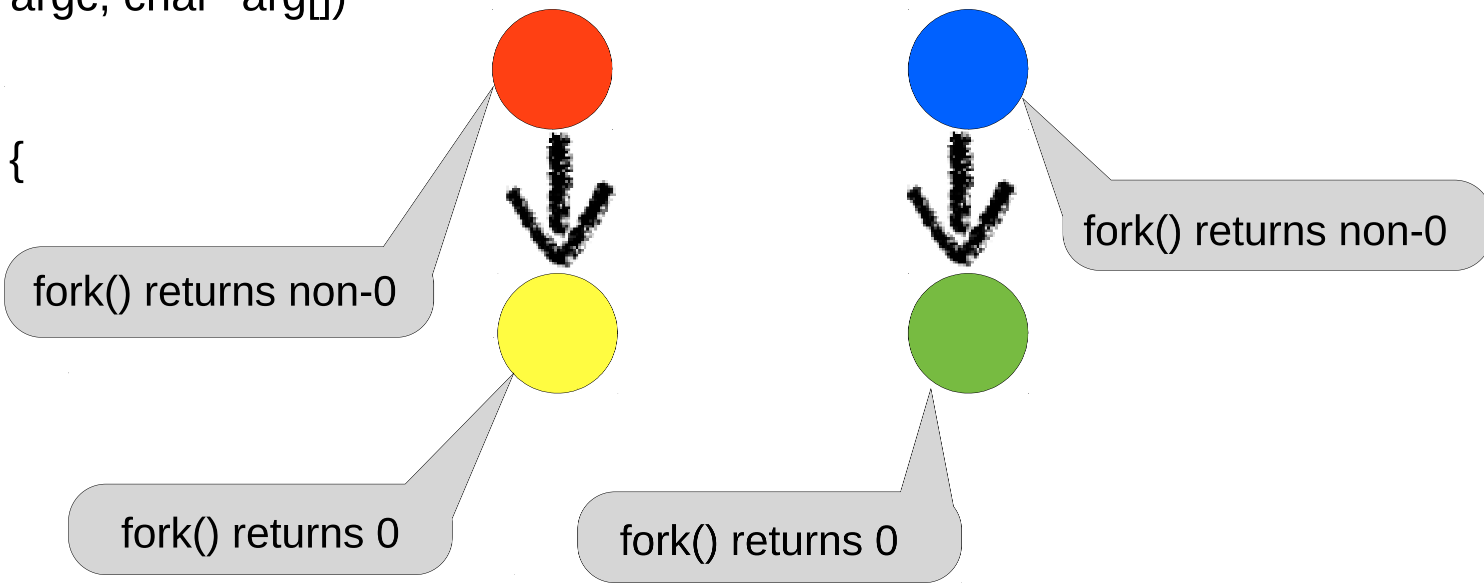
```c
int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    }
    fork ();
}
```

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

Popular Homework Question

- How many processes does this C program create?

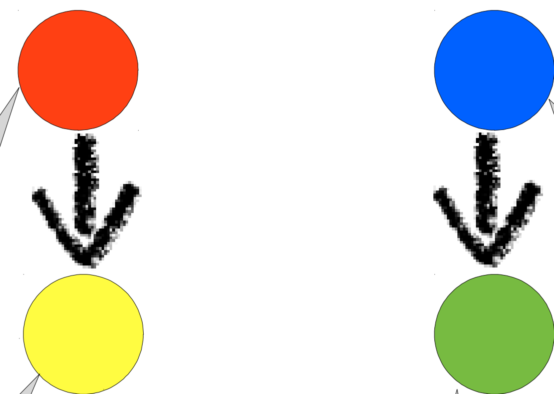
```
int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    }
    fork ();
}
```



Popular Homework Question

- How many processes does this C program create?

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

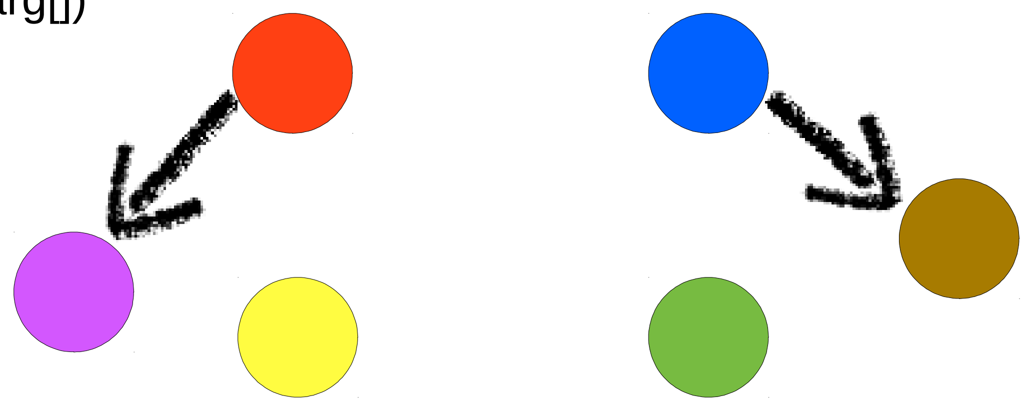


yellow and green: don’t go into the if clause
red and blue: go!!

Popular Homework Question

- How many processes does this C program create?

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

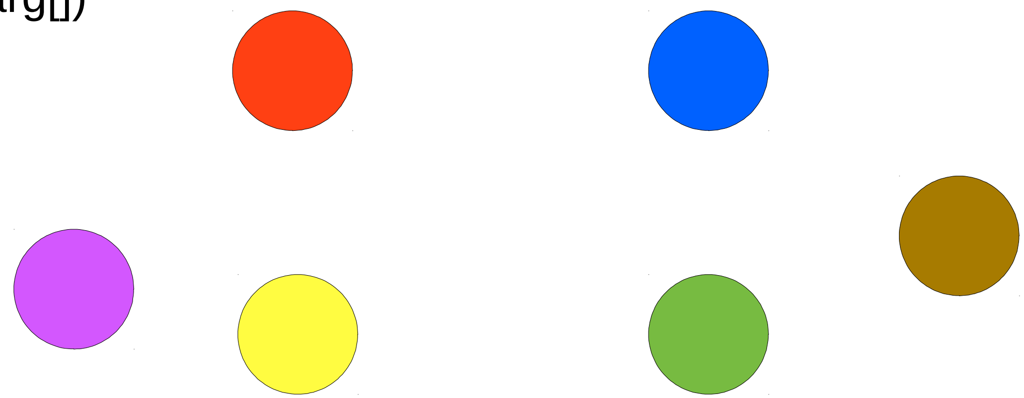


red and blue each creates a new child process (purple an brown)

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 from 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

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** for the child to terminate

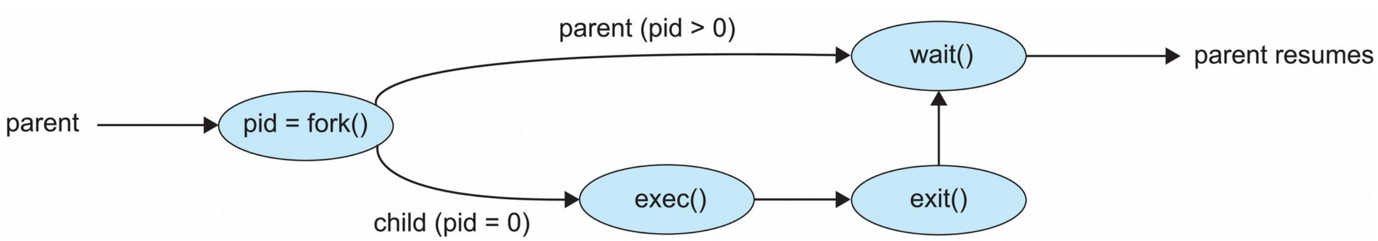
The exec*() Family of Syscalls

- The “exec” system call replaces the process image by that of a specific program
  - see “man 3 exec” to see all the versions
  - Standard names of such functions in C are **execl**, **execle**, **execlp**, **execv**, **execve**, and **execvp**, but not “exec” itself. The Linux kernel has one corresponding system call named “execve”, whereas all aforementioned functions are user-space wrappers around it. (https://en.wikipedia.org/wiki/Exec_(system_call))
  - Try man 2 execve
- 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

The exec*() Family of Syscalls

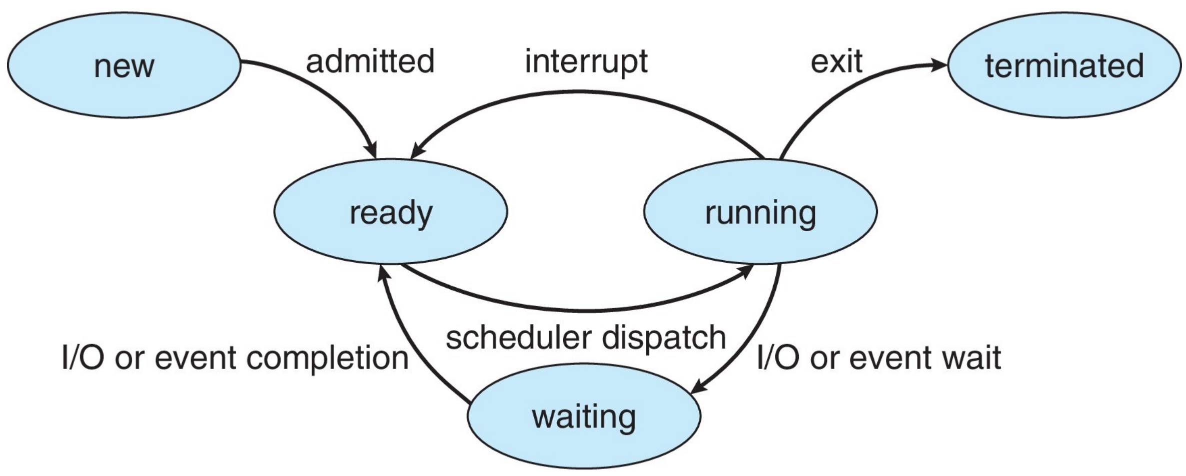
- The “exec” system call replaces the process image by that of a specific program

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



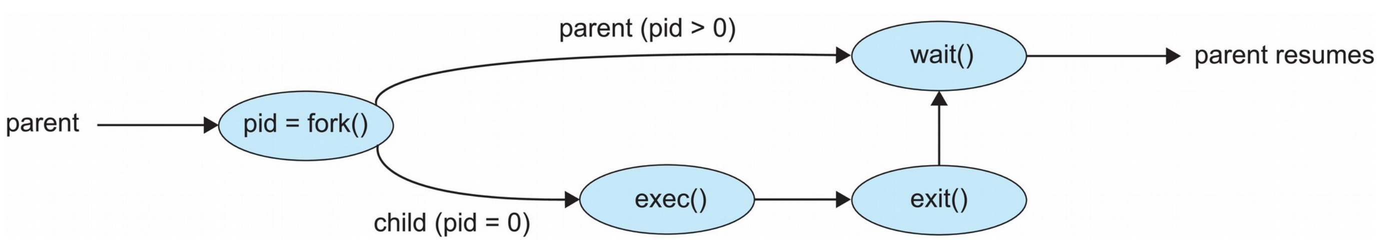
Process State

- 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



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 with WNOHANG options
  - Read the man pages ("man 2 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
  - ^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

Processes and Signals

- Kill command

```
wenbo@parallels:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Manipulating Signals

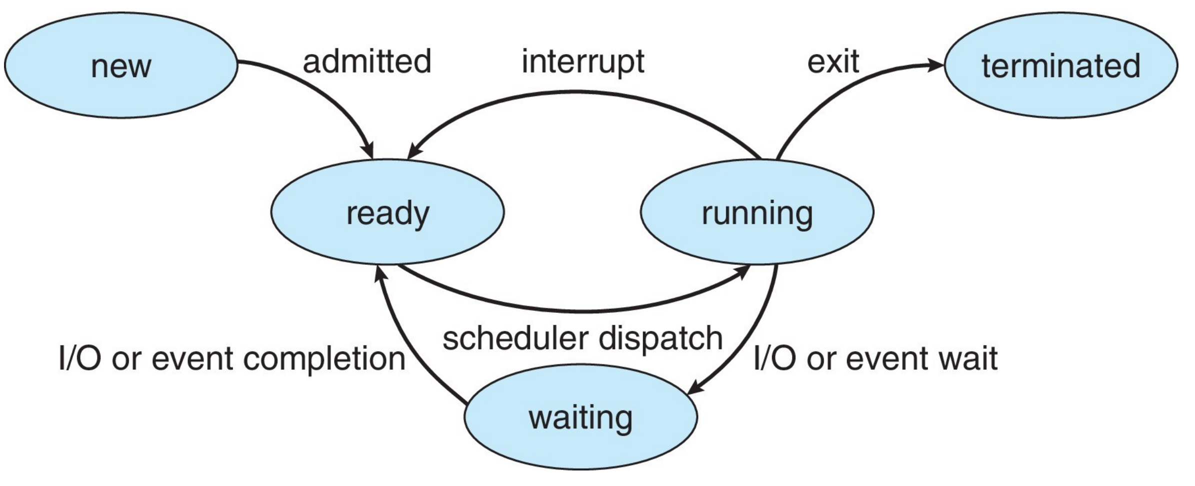
- Each signal causes a default behavior in the process
  - e.g., a SIGINT signal causes the process to terminate
- But most signals can be either ignored or provided with a user-written handler to perform some action
  - Signals like SIGKILL and SIGSTOP cannot be ignored or handled by the user, for security 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 don't want to die!\n");
    return;
}

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

Process State

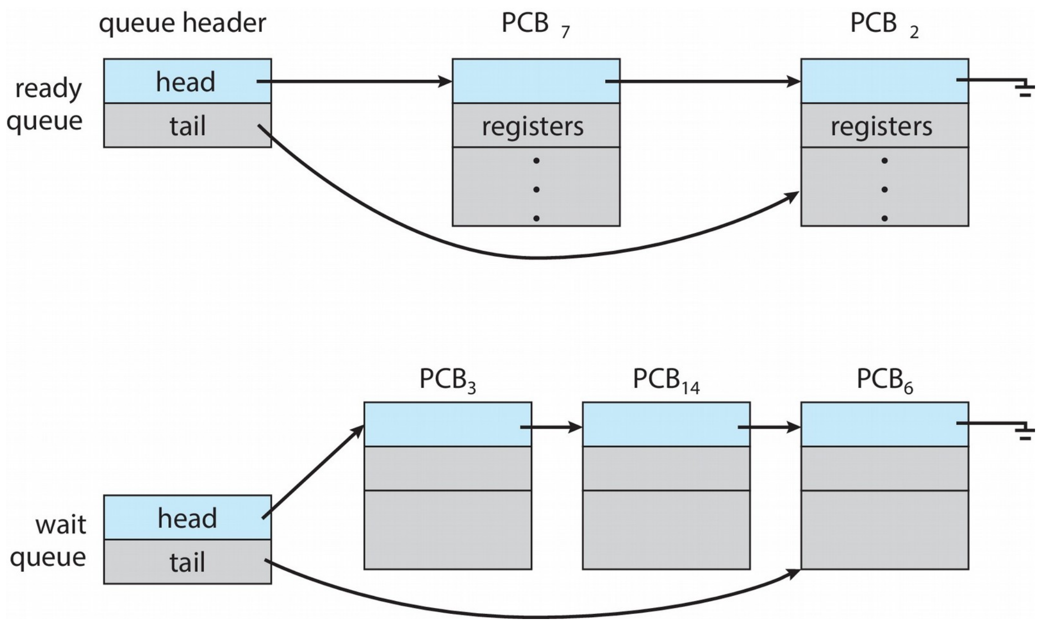
- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process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



Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU core
- **Process scheduler** selects among available processes for next execution on CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e. I/O)
  - Processes migrate among the various queues

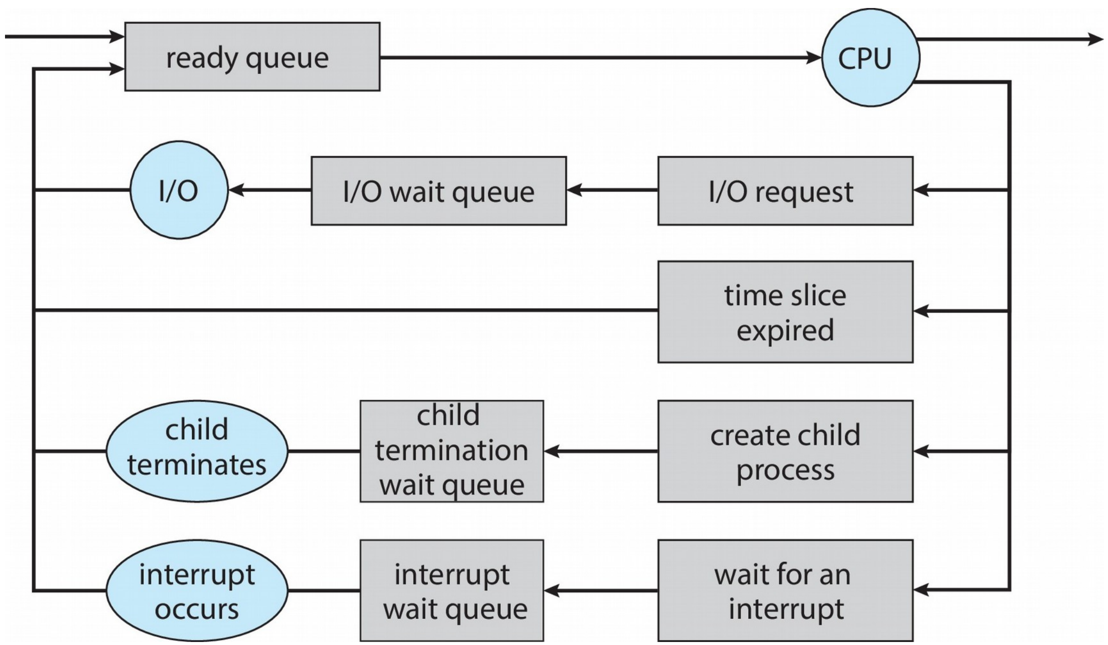
Ready and Wait Queues




```
struct list_head {
    struct list_head *next, *prev;
};
```

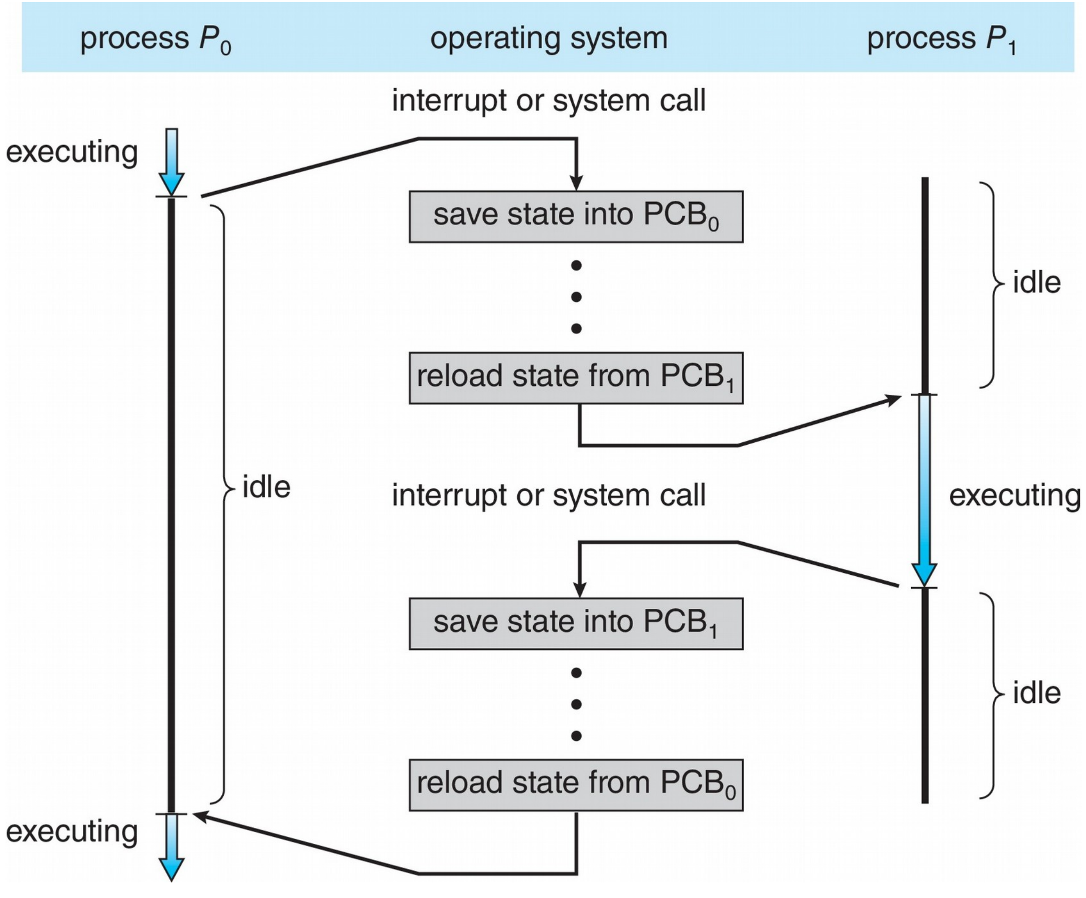
https://elixir.bootlin.com/linux/v5.2.8/source/include/linux/types.h#L181

Representation of Process Scheduling



CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



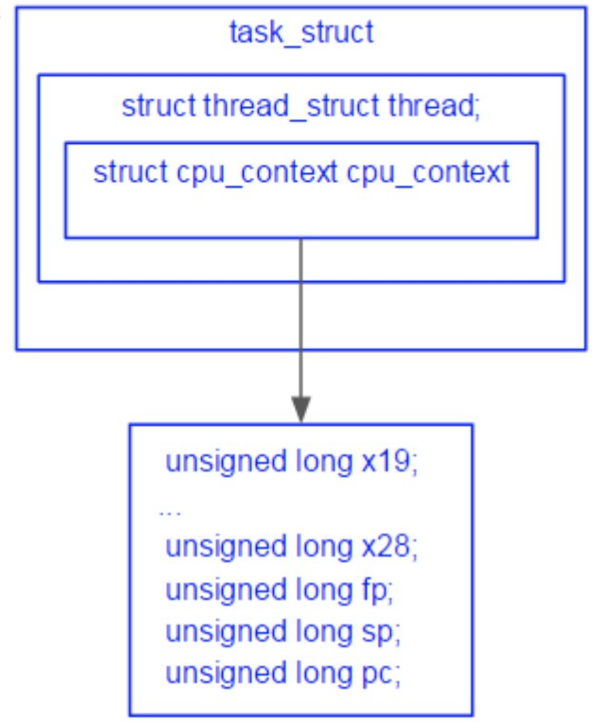
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ⇒ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ⇒ multiple contexts saved/loaded at once

Context Switch

```
extern struct task_struct *cpu_switch_to(struct task_struct *prev,
                                         struct task_struct *next);
```

```
ENTRY(cpu_switch_to)
        mov     x10, #THREAD_CPU_CONTEXT
        add     x8, x0, x10
        mov     x9, sp
        stp     x19, x20, [x8], #16
        stp     x21, x22, [x8], #16
        stp     x23, x24, [x8], #16
        stp     x25, x26, [x8], #16
        stp     x27, x28, [x8], #16
        stp     x29, x9, [x8], #16
        str     lr, [x8]
        add     x8, x1, x10
        ldp     x19, x20, [x8], #16
        ldp     x21, x22, [x8], #16
        ldp     x23, x24, [x8], #16
        ldp     x25, x26, [x8], #16
        ldp     x27, x28, [x8], #16
        ldp     x29, x9, [x8], #16
        ldr     lr, [x8]
        mov     sp, x9
        msr     sp_el0, x1
        ret
ENDPROC(cpu_switch_to)
```



Context Switch

```
extern struct task_struct *cpu_switch_to(struct task_struct *prev,
                                         struct task_struct *next);
```

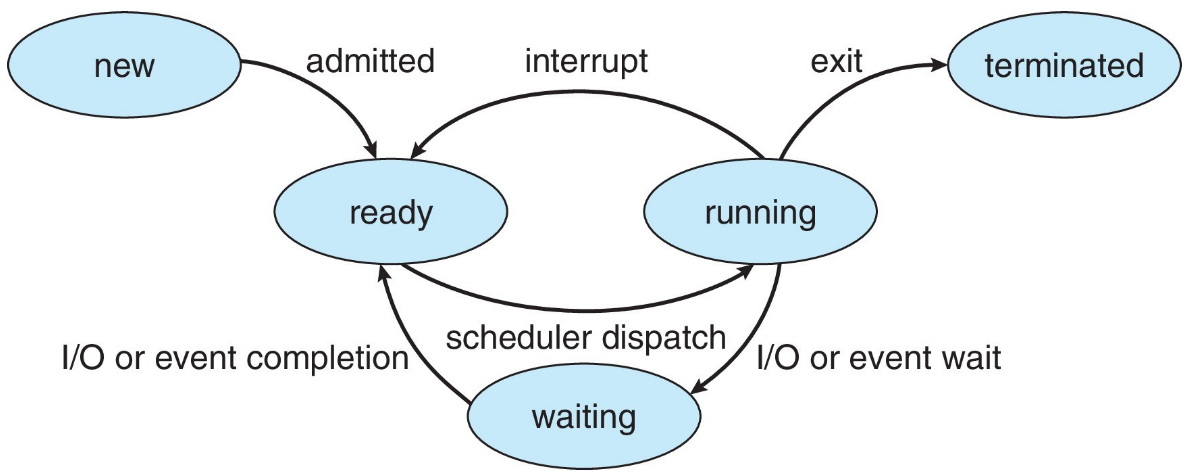
```
ENTRY(cpu_switch_to)
        mov     x10, #THREAD_CPU_CONTEXT
        add     x8, x0, x10
        mov     x9, sp
        stp     x19, x20, [x8], #16
        stp     x21, x22, [x8], #16
        stp     x23, x24, [x8], #16
        stp     x25, x26, [x8], #16
        stp     x27, x28, [x8], #16
        stp     x29, x9, [x8], #16
        str     lr, [x8]
        add     x8, x1, x10
        ldp     x19, x20, [x8], #16
        ldp     x21, x22, [x8], #16
        ldp     x23, x24, [x8], #16
        ldp     x25, x26, [x8], #16
        ldp     x27, x28, [x8], #16
        ldp     x29, x9, [x8], #16
        ldr     lr, [x8]
        mov     sp, x9
        msr     sp_el0, x1
        ret
ENDPROC(cpu_switch_to)
```

- Most important step is **STACK SWITCHING**

1) Move sp to general reg
2) Save old sp
3) Load new sp
4) Move new sp to sp reg

Process State

- 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



Code through

- Linux 0.11 (1991)
  - No list concept
  - Fixed PCB table, max 64
  - No ready queue
    - Go through PCB table directly to pick the next
  - No wait queue
    - Use a fixed array

Code through

- Linux 2.3.0
  - introduced list
  - Fixed PCB table, max 512
  - Has the unnamed ready queue
    - Implemented by list
    - task_struct -> prev_run, task_struct -> next_run
    - Go through list to pick the next
  - Has named wait queue
    - Such as motor_wait for flop disk
    - Sleep_on adds current to wait queue of motor_wait

Code through

- Linux 2.4.0
  - task_struct number can be dynamic
  - Has the named ready queue
    - Called runqueue_head, implemented by list
    - Links task_struct together, using task_struct->run_list
    - Go through list to pick the next
  - Has named wait queue
    - Such as motor_wait for flop disk
    - Sleep_on adds current to wait queue of motor_wait

Code through

- Linux 2.6.0
  - task_struct number can be dynamic
  - Has the named ready queue
    - Called runqueue_head, implemented by struct runqueue
    - Links task_struct together, using task_struct->run_list
    - Go through priority array to pick the next
  - Has named wait queue
    - Such as motor_wait for flop disk
    - Sleep_on adds current to wait queue of motor_wait

Latest kernel

- How about latest Linux 5.3.1
  - Tooooooooooooooooooo complex, complicated
  - Each schedule policy
    - Has dedicate data structures, such as rt_sched_class, cfs_sched_class
    - Using different runqueue
  - Runqueue
    - Can be list based
    - Can be (list + array) based
    - Can be tree based

Zombie - They’re dead.. but alive!

- When a child process terminates
  - Remains as a zombie in an “undead” state
  - Until it is “reaped” (garbage collected) by the OS
- Rationale:
  - The 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~~ CPU
  - They only consume a slot in memory
    - Which may eventually fill up and cause fork() to fail
- Let's look at zombie_example.c
  - *`ps xao pid,ppid,comm,state | grep a.out`*

Getting rid of zombies

- 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

- 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

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
  - Demo ： orphan_example1.c
- 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
  - Demo: orphan_example2.c

Takeaway

- Process Concept
  - Process vs Program
- Process Control Block
  - task_struct
- Process State
  - Five states, who has a queue
  - How to create and terminate a process
- Process Scheduling
  - Where are registers saved?
  - Switch steps

Homework 1 deadline is this Thursday （Sep 26）

Lab0 deadline next Tuesday （Oct 1）