Final Review 01

Operating Systems
Wenbo Shen

Acknowledgement: Some slides are based on Yajin Zhou@ZJU slides.

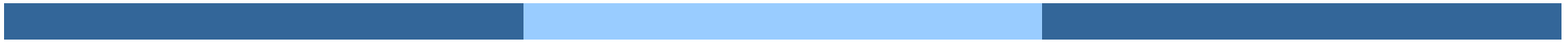
Summary

- Computer architecture
- OS introduction
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

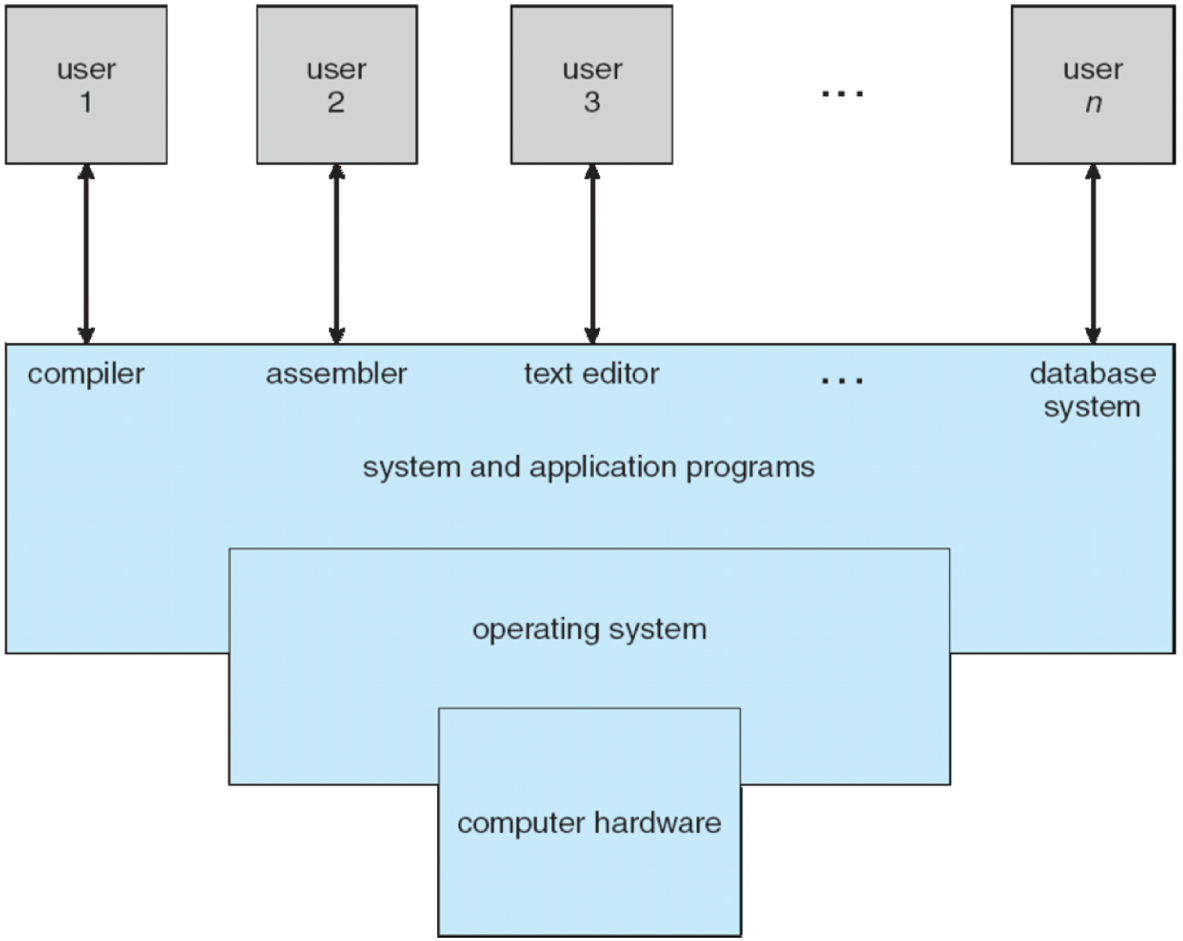
Summary

- Memory – segmentation
- Memory – paging
- Virtual memory
- Virtual memory – Linux
- Mass storage
- IO
- FS interface
- FS implementation
- FS in practice

01: Introduction



Four Components of a Computer System



Interrupts and Traps

- Interrupt transfers control to the interrupt service routine
  - **interrupt vector:** a table containing addresses of all the service routines
  - incoming interrupts are disabled while serving another interrupt to prevent a lost interrupt
  - **interrupt handler** must save the (interrupted) execution states
- A **trap** is a **software-generated interrupt**, caused either by an error or a user request
  - an **interrupt** is asynchronous; a **trap** is synchronous
  - e.g., **system call**, divided-by-zero exception, general protection exception…
    - Int 0x80
- Operating systems are usually **interrupt-driven**

Interrupt Handling

- Operating system preserves the execution state of the CPU
  - save registers and the program counter (PC)
- OS determines which device caused the interrupt
  - **polling**
  - **vectored** interrupt system
- OS handles the interrupt by calling the device's driver
- OS restores the CPU execution to the saved state

I/O: from System Call to Devices, and Back

- A program uses a **system call** to access system resources
  - e.g., files, network
- Operating system converts it to device access and issues I/O requests
  - I/O requests are sent to the device driver, then to the controller
  - e.g., read disk blocks, send/receive packets…
- OS puts the program to wait (**synchronous I/O**) or returns to it without waiting (**asynchronous I/O**)
  - OS may switches to another program when the requester is waiting
- I/O completes and the controller **interrupts** the OS
- OS processes the I/O, and then wakes up the program (synchronous I/O) or send its a signal (asynchronous I/O)

Operating System Operations: Multiprogramming

- **Multiprogramming** is necessary for efficiency
  - single user cannot keep CPU and I/O devices busy at all times
  - user's computing tasks are organized as jobs (code and data)
  - kernel schedules jobs (job scheduling) so CPU always has things to do
    - a subset of total jobs in system is kept in memory
    - when a job **has to wait** (e.g., for I/O), kernel switches to another job
    - What are the problems here?

Operating System Operations: multitasking

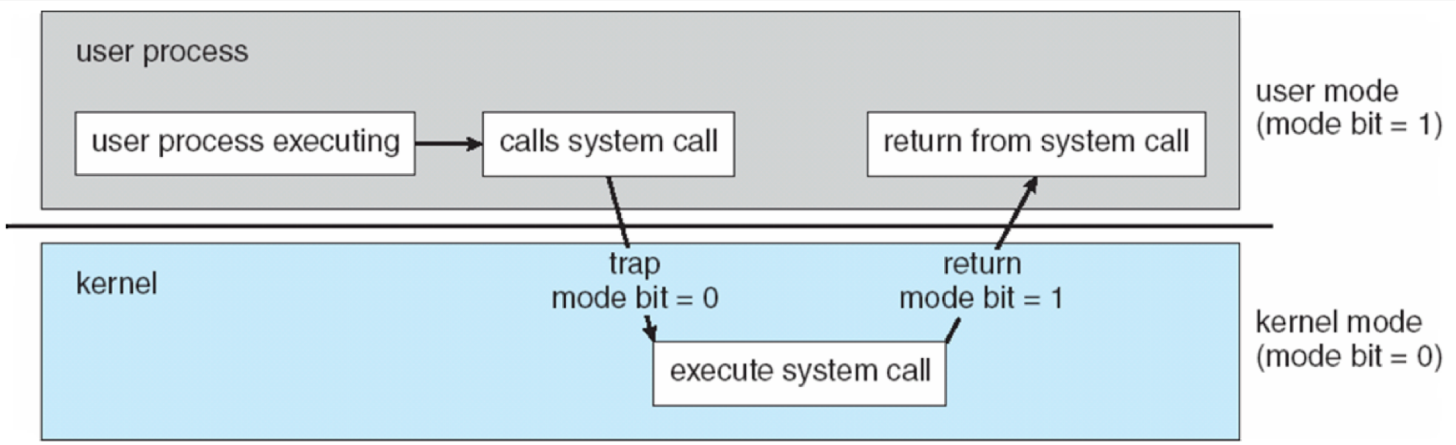
- **Timesharing** (multitasking) extends the multiprogramming
  - OS switches jobs **so frequently** that users can interact with each running job
  - response time should be < 1s
  - each user has at least one program executing in memory (**process**)
  - if several jobs ready to run at the same time (**CPU scheduling**)
  - It makes the programmer easier: virtual/physical memory

Dual-mode operation

- Operating system is usually interrupt-driven (why?)
  - Efficiency, regain control (timer interrupt)
- **Dual-mode operation** allows OS to protect itself and other system components
  - **user mode** and **kernel mode (or other names)**
  - a **mode** bit distinguishes when CPU is running user code or kernel code
  - some instructions designated as **privileged**, **only executable in kernel and cannot executed in user mode (and certain memory cannot be accessed in user mode!)**
  - **system call** changes mode to kernel, return from call resets it to user

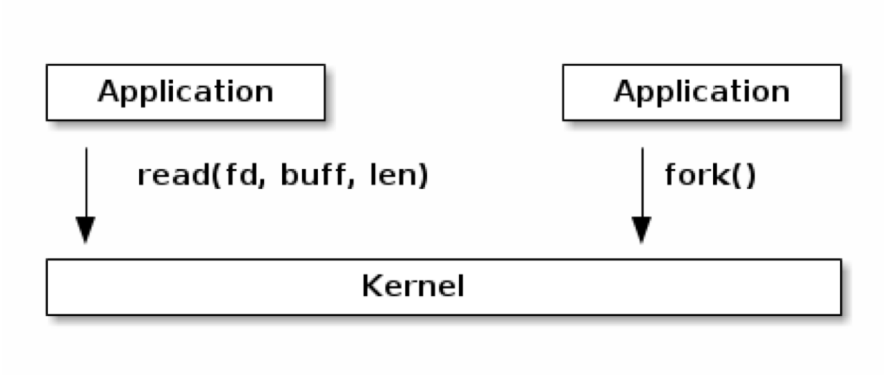
Transition between Modes

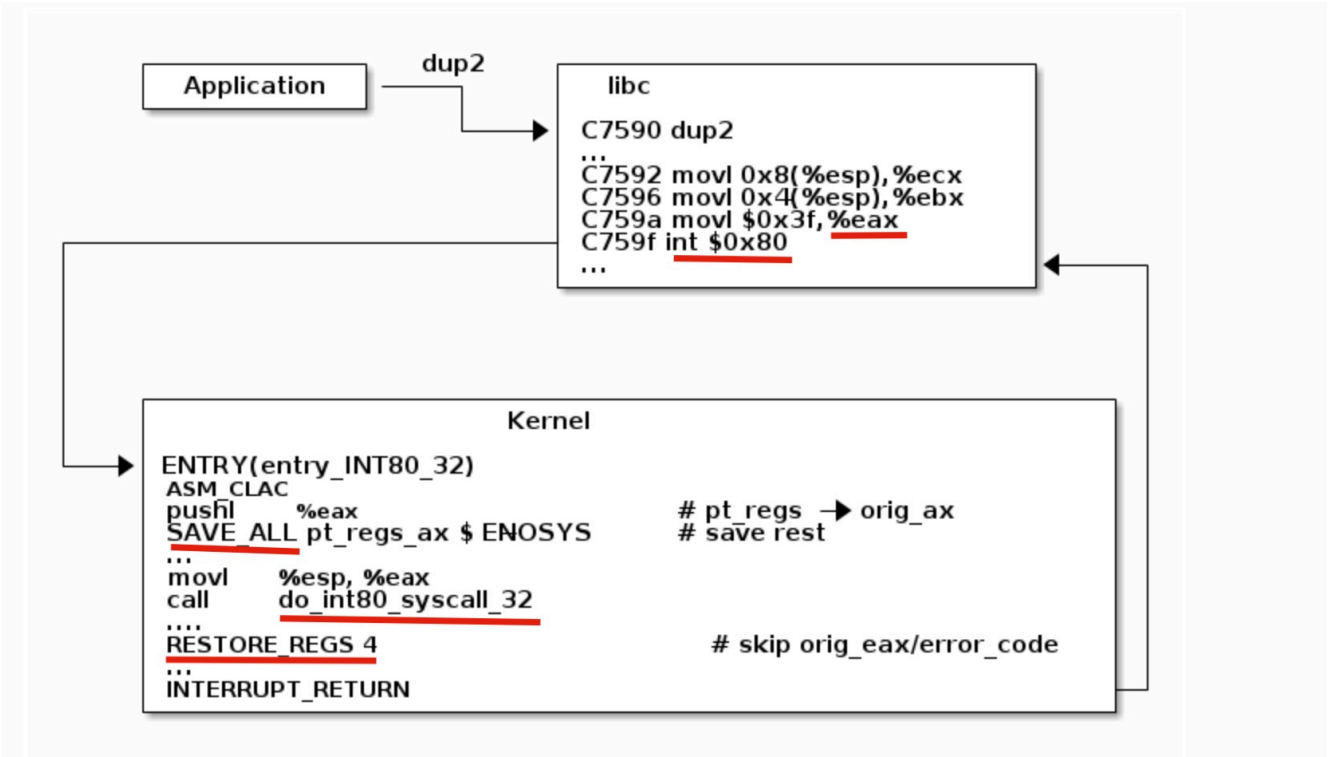
- **System calls, exceptions, interrupts** cause transitions between kernel/user modes



In user mode, certain instructions cannot be executed

How System Call Is Handled





How System Call Is Handled

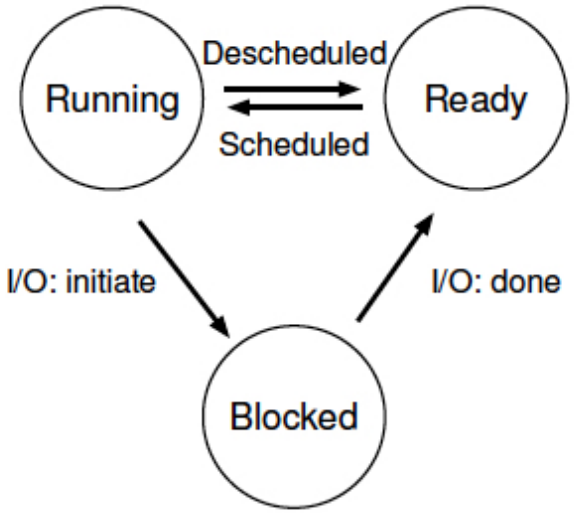
- System call number and parameters:
  - on 32bit x86 architecture, the system call identifier is stored in the EAX register, while parameters in registers EBX, ECX, EDX, ESI, EDI, EBP
- Steps:
  - The application is setting up the system call number (EAX for x86) and parameters and it issues a trap instruction (int 0x80 for x86)
  - **The execution mode switches** from user to kernel; the CPU switches to a kernel stack; the user stack and the return address to user space is saved on the **kernel stack**
  - The kernel entry point saves registers on the kernel stack (SAVE_ALL)
  - Get system call number
  - The system call dispatcher identifies the system call function using the system call number
  - Execute the system call function and saves the return value (to EAX for x86)
  - The user space registers/stack are restored and execution is switched back to user (e.g. calling IRET)
  - The user space application resumes

Resource Management: Process Management

- A process is **a program in execution**
  - program is a ***passive*** entity, process is an ***active*** entity
  - a system has many processes running concurrently
- Process needs resources to accomplish its task
  - OS reclaims all reusable resources upon process termination
  - e.g., CPU, memory, I/O, files, initialization data

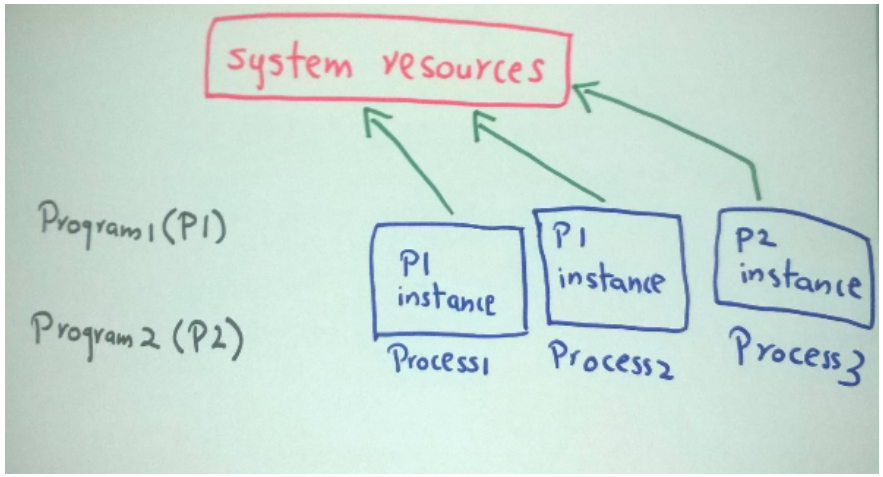
Process Management Activities

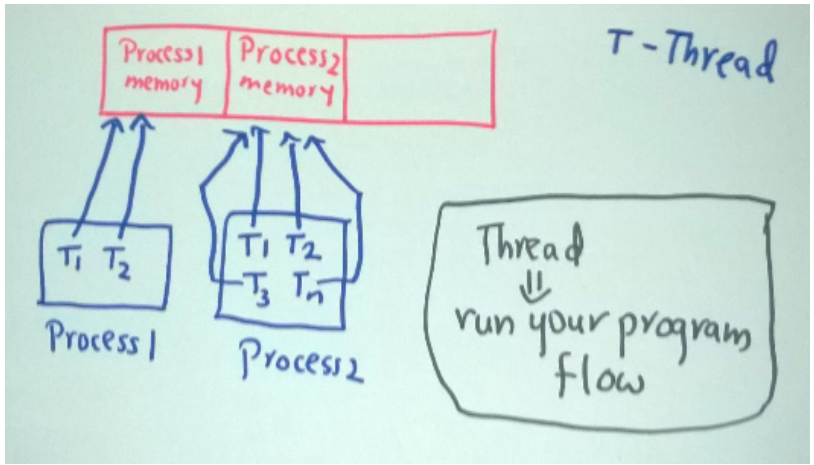
- Process creation and termination
- Processes suspension and resumption
- Process synchronization primitives
- Process communication primitives
- Deadlock handling



From Process to Thread

- Single-threaded process has one program counter
  - **program counter** specifies *location of next instruction to execute*
  - processor executes instructions sequentially, one at a time, until completion
- Multi-threaded process has **one program counter per thread**
- **Process is the unit of resource allocation and protection, not thread!**





Figures: https://medium.com/coderscorner/threads-vs-processes-a0b199f56c2d

Resource Management: Memory Management

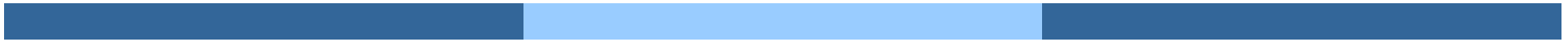
- Memory is the main storage directly accessible to CPU
  - data needs to be kept in memory before and after processing
  - all instructions should be in memory in order to execute
- Memory management determines what is in memory to **optimize CPU utilization** and **response time, provides a virtual view of memory for programmer**
- Memory management activities:
  - keeping track of which parts of memory are being used and by whom
  - deciding which processes and data to move into and out of memory
  - allocating and deallocating memory space as needed

Resource Management: File Systems

- OS provides a uniform, logical view of data storage
  - **file** is a logical storage unit that abstracts physical properties
    - files are usually organized into **directories**
    - **access control** determines who can access the file
- File system management activities:
  - creating and deleting files and directories
  - primitives to manipulate files and directories
  - mapping files onto secondary storage
  - backup files onto stable (non-volatile) storage media

A special FS: /proc file system

02: Operating System Services & Structures



Operating System Services (User/Programmer-Visible)

- **User interface**
  - most operating systems have a user interface (UI).
  - e.g., command-Line (CLI), graphics user interface (GUI), or batch
- **Program execution: from program to process**
  - load and execute an program in the memory
  - end execution, either normally or abnormally
- **I/O operations**
  - a running program may require I/O such as file or I/O device
- **File-system manipulation**
  - r/w, create, delete files/directories
  - search or list files and directories
  - permission management

```
work@hackvm:~/temp$ ls -l
total 21256
drwxrwxr-x  6 work work     4096 Jan  6 02:39 angr-utils
drwxrwxr-x  7 work work     4096 Jan  6 02:39 bingraphvis
-rwxrwxr-x  1 work work     8704 Jan  7 00:48 t
-rw-rw-r--  1 work work      115 Jan  7 00:48 t.c
drwxrwxr-x 28 work work     4096 Jan  6 03:05 valgrind-3.14.0
-rw-rw-r--  1 work work 16602858 Oct 10 03:42 valgrind-3.14.0.tar.bz2
drwxrwxr-x 11 work work     4096 Jan  6 23:27 z3-z3-4.8.4
-rw-r--r--  1 work work  5126062 Jan  6 23:16 z3-z3-4.8.4.zip
```

Operating System Services (User-Visible)

- **Communications**
  - processes exchange information, on the same system or over a network
  - via shared memory or through message passing
- **Error detection**
  - OS needs to be constantly aware of possible errors
  - errors in CPU, memory, I/O devices, programs
  - it should take appropriate actions to ensure correctness and consistency

Operating System Services (System View)

- **Resource allocation**
  - allocate resources for multiple users or multiple jobs running concurrently
  - many types of resources: CPU, memory, file, I/O devices
- **Accounting/Logging**
  - to keep track of which users use how much and what kinds of resources
- **Protection and security**
  - protection provides a mechanism to control access to system resources
    - access control: control access to resources
    - isolation: processes should not interfere with each other
  - security authenticates users and prevent invalid access to I/O devices
    - a chain is only as strong as its weakest link
  - protection is the **mechanism**, security towards the **policy**

```
top - 01:25:31 up 14:16,  3 users,  load average: 0.00, 0.00, 0.00
Tasks:  98 total,   1 running,  97 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.0 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1003772 total,   217452 free,    69248 used,   717072 buff/cache
KiB Swap:  1046524 total,  1046012 free,      512 used.   749832 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26713 root      20   0       0      0      0 S   0.3  0.0   0:00.01 kworker/u6+
    1 root      20   0  119600   5076   3336 S   0.0  0.5   0:03.39 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kthreadd
    4 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:+
    6 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 mm_percpu_+
    7 root      20   0       0      0      0 S   0.0  0.0   0:00.04 ksoftirqd/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:01.02 rcu_sched
    9 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 watchdog/0
   12 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/0
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
   14 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 watchdog/1
   15 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/1
   16 root      20   0       0      0      0 S   0.0  0.0   0:00.11 ksoftirqd/1
   18 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:+
   19 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   20 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
```

User Interface

- GUI
- Shell
  - How to get a return value from in bash: $?
  - Operations: eq: for numeric one, == for string. "$?" -eq "1"
  - How to chain multiple commands: cmd1;cmd2;cmd3
  - echo, touch, cat

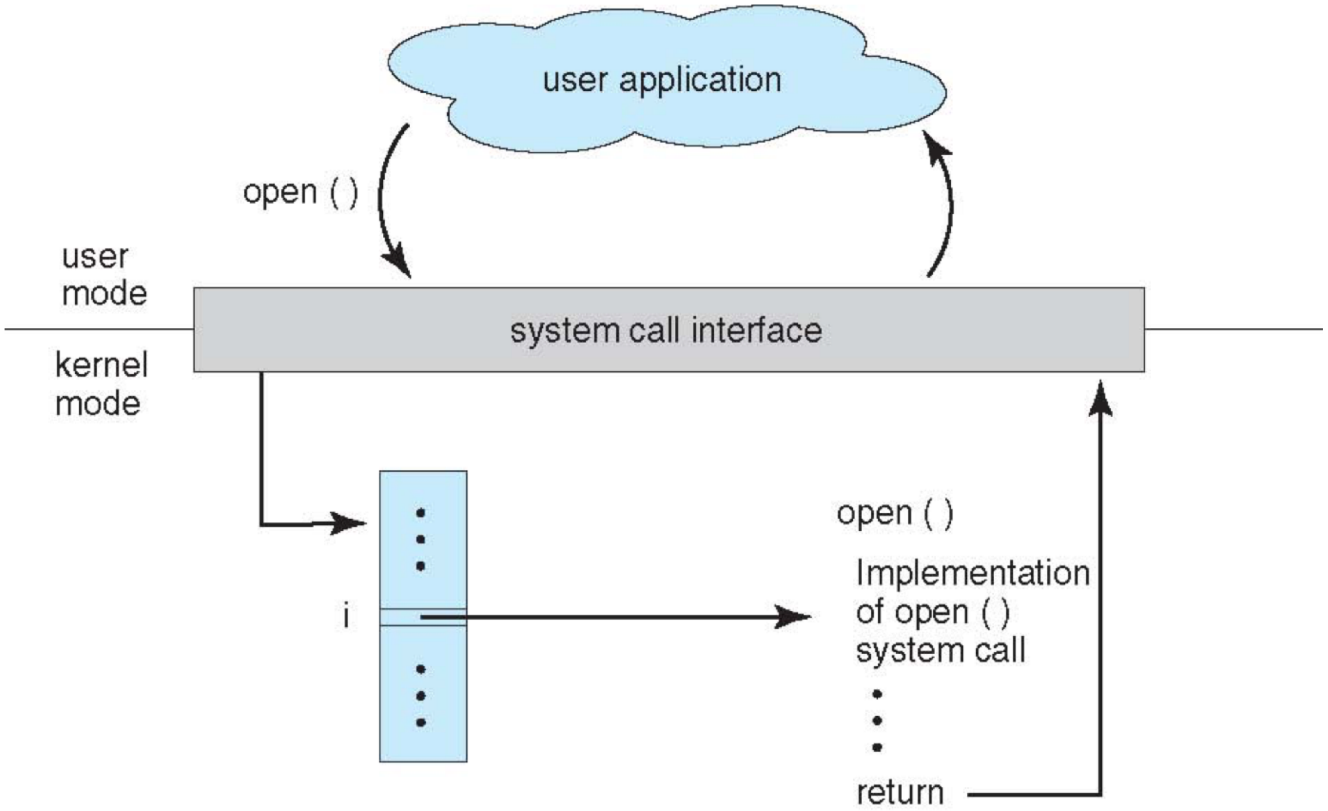
System Calls

- System call is **a programming interface to access the OS services**
  - Typically written in a high-level language (C or C++)
  - Certain low level tasks are in assembly languages

Application Programming Interface

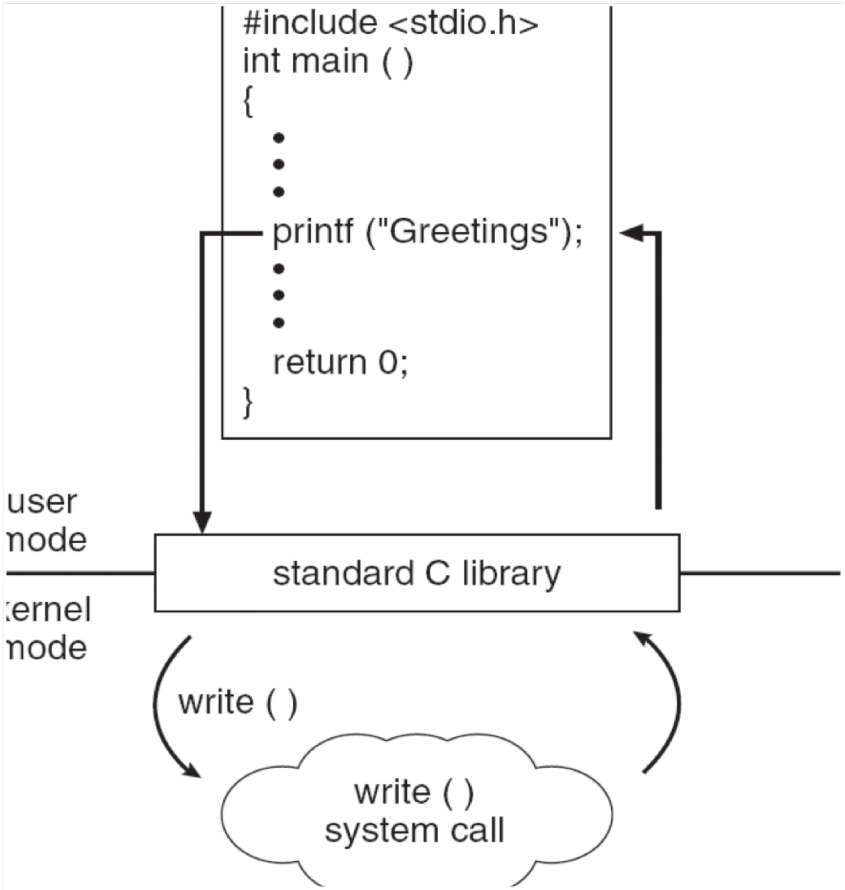
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
  - three most common APIs:
    - **Win32** API for Windows
    - **POSIX** API for POSIX-based systems (UNIX/Linux, Mac OS X)
    - **Java** API for the Java virtual machine (JVM)
    - why use APIs rather than system calls?
      - portability

API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



System Call Parameter Passing

- Parameters are required besides the **system call number**
  - exact type and amount of information vary according to OS and call
- Three general methods to pass parameters to the OS
  - **Register**:
    - pass the parameters in registers
    - simple, but there may be more parameters than registers
  - **Block**:
    - parameters stored in a memory block (or table)
    - address of the block passed as a parameter in a register
    - taken by Linux and Solaris
  - **Stack**:
    - parameters placed, or pushed, onto the stack by the program
    - popped off the stack by the operating system
  - Block and stack methods don't limit number of parameters being passed

How System Call Is Handled

- System call number and parameters:
  - on 32bit x86 architecture, the system call identifier is stored in the EAX register, while parameters in registers EBX, ECX, EDX, ESI, EDI, EBP
- Steps:
  - The application is setting up the system call number (EAX for x86) and parameters and it issues a trap instruction (int 0x80 for x86)
  - **The execution mode switches** from user to kernel; the CPU switches to a kernel stack; the user stack and the return address to user space is saved on the **kernel stack**
  - The kernel entry point saves registers on the kernel stack (SAVE_ALL)
  - Get system call number
  - The system call dispatcher identifies the system call function using the system call number
  - Execute the system call function and saves the return value (to EAX for x86)
  - The user space registers/stack are restored and execution is switched back to user (e.g. calling IRET)
  - The user space application resumes

Operating System Design and Implementation

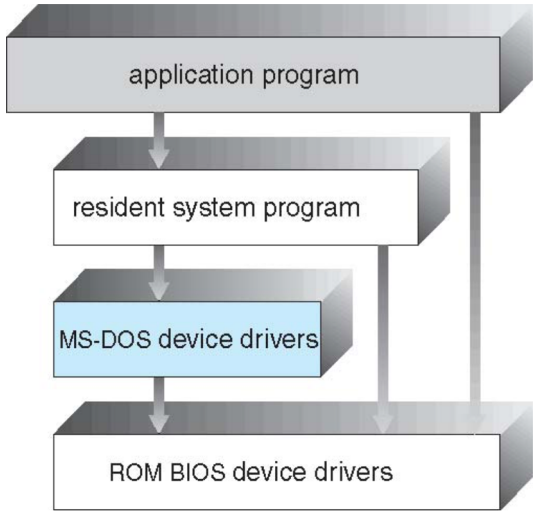
- Much variation
  - Early OSes in **assembly language**
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in **assembly**
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to port to other hardware
  - But slower

Operating System Structure

- Many structures:
  - **simple structure - MS-DOS**
  - **more complex -- UNIX**
  - **layered structure - an abstraction**
  - **microkernel system structure - L4**
  - **hybrid: Mach, Minix**
  - research system: exokernel

Simple Structure: MS-DOS

- No structure at all!: (1981~1994)
  - written to provide the most functionality in the least space
- A typical example: MS-DOS
  - Has some structures:
    - its interfaces and levels of functionality are not well separated
  - the kernel is not divided into modules

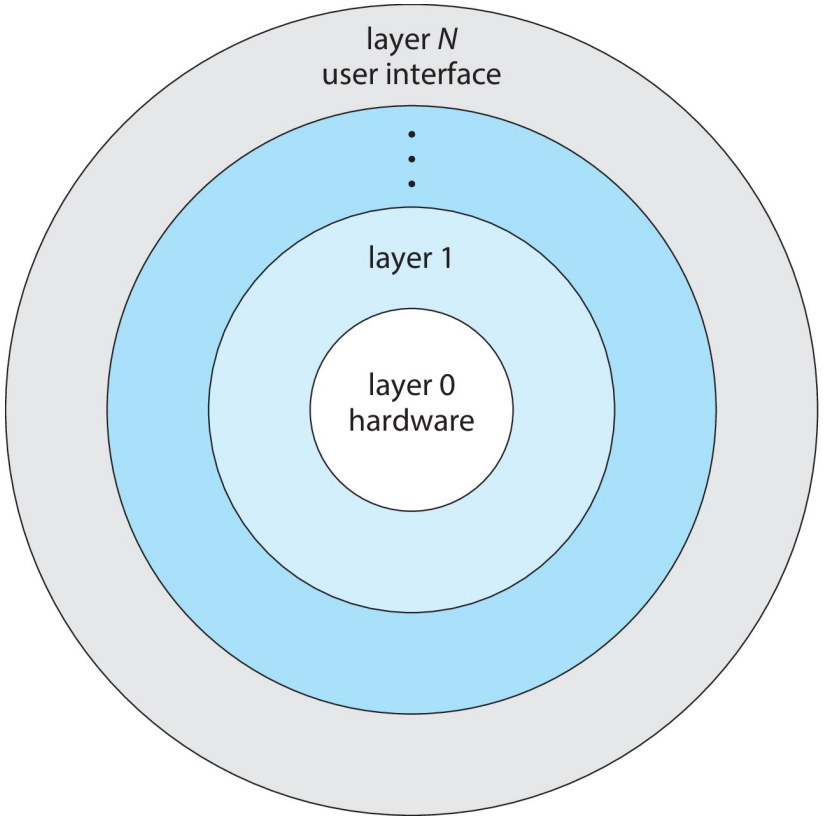


Monolithic Structure – Original UNIX

- Limited by hardware functionality, the original UNIX had limited structure
- UNIX OS consists of two separable layers
  - systems programs
  - the kernel: everything below the system-call interface and above physical hardware
    - a large number of functions for one level: file systems, CPU scheduling, memory management …

Layered Approach

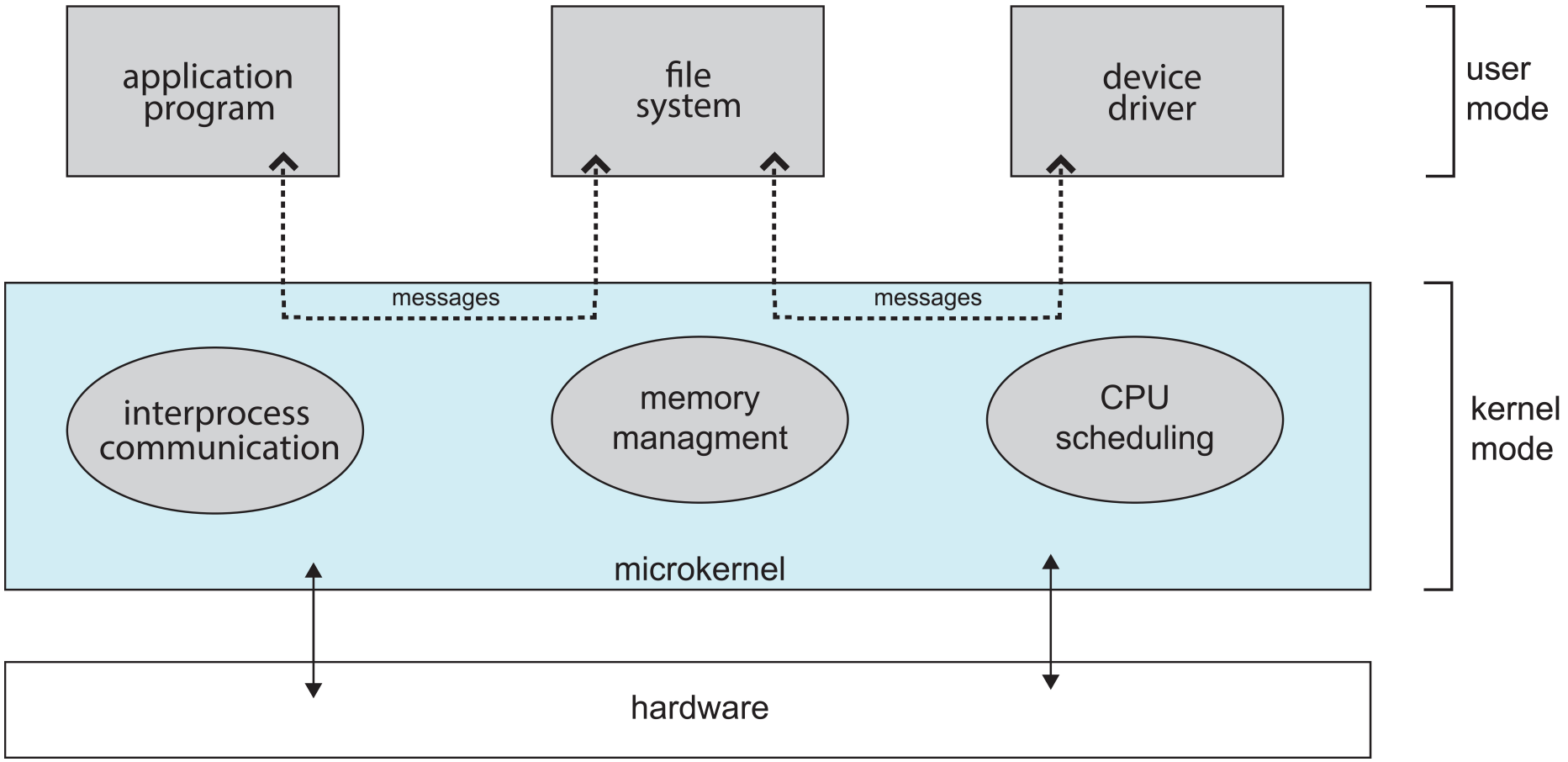
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Crossing layers introduces overhead



Microkernels

- Moves as much from the kernel into user space
  - Implemented as user space program
- **Mach** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure



Microkernel System Structure

- Microkernel moves as much from the kernel (e.g., file systems) into “user” space
- Communication between user modules uses **message passing**
- Benefits:
  - easier to extend a microkernel
  - easier to port the operating system to new architectures
  - more reliable (less code is running in kernel mode)
  - more secure
- Detriments:
  - performance overhead of user space to kernel space communication
- Examples: Minix, Mach, QNX, L4…

Exokernel: Motivation

- In traditional operating systems, only **privileged servers and the kernel** can manage system resources
- Un-trusted applications are required to interact with the hardware via some **abstraction model**
  - File systems for disk storage, virtual address spaces for memory, etc.
- **But application demands vary widely**!!
  - An interface designed to accommodate every application must anticipate all possible needs

03: Process

Process Concept

- Process is **a program in execution**, its execution must progress in sequential fashion
  - a program is static and passive, process is dynamic and active
  - **one program can be several processes** (*e.g.*, **multiple instances** of browser, or **even on instance** of the program)
  - process can be started via GUI or command line entry of its name
    - through system calls
- **Process is the basic unit for resource allocation and protection**
  - **Thread is unit of execution**

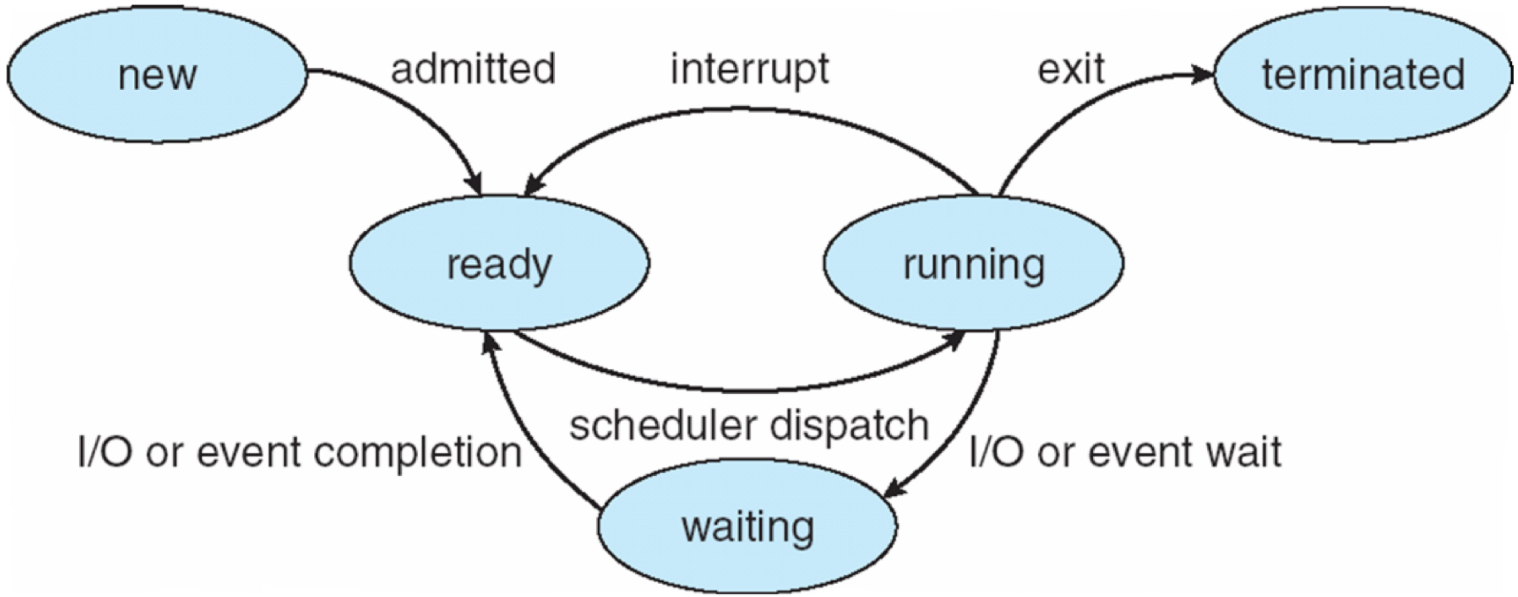
Process Concept

- A process has multiple parts:
  - the program **code**, also called **text section**
  - runtime **CPU states**, including program counter, registers, etc
  - various types of memory:
    - **stack**: temporary data
      - e.g., function parameters, local variables, and *return addresses*
    - **data** section: global variables
    - **heap**: memory dynamically allocated during runtime

Process State

- As a process executes, it changes state
  - **new**: the process is being created
  - **running**: instructions are being executed
  - **waiting/blocking**: 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

Diagram of Process State



Process Control Block (PCB)

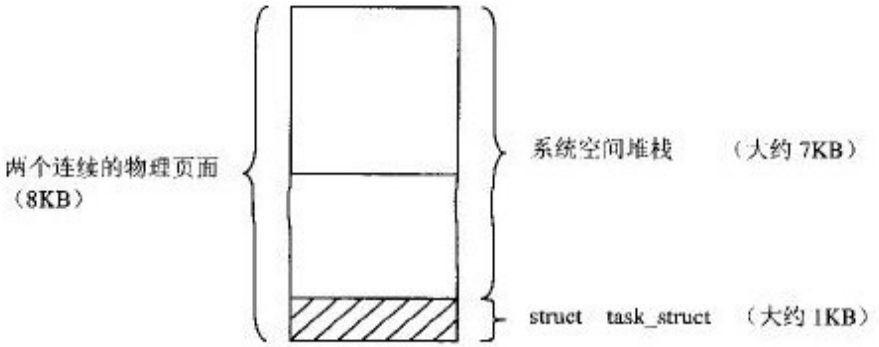
- In the kernel, each process is associated with a **process control block**
  - process number (pid)
  - **process state**
  - **program counter (PC)**
  - CPU registers
  - CPU scheduling information
  - memory-management data
  - accounting data
  - I/O status

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * 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. Only
         * scheduling-critical items should be added above here.
         */
        randomized_struct_fields_start

        void                            *stack;
        atomic_t                        usage;
        /* Per task flags (PF_*), defined further below: */
        unsigned int                    flags;
        unsigned int                    ptrace;
```

- Linux's PCB is defined in struct task_struct: https://elixir.bootlin.com/linux/v5.5-rc3/source/include/linux/sched.h#L629

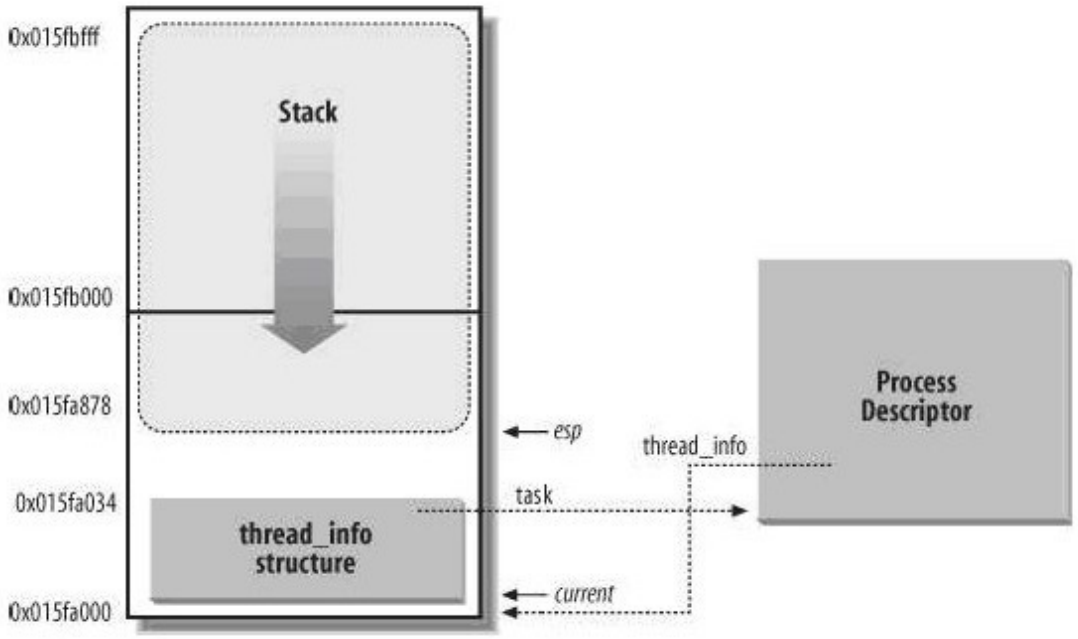
Kernel stack

- 2.4



进程系统堆栈示意图

- 2.6



Threads

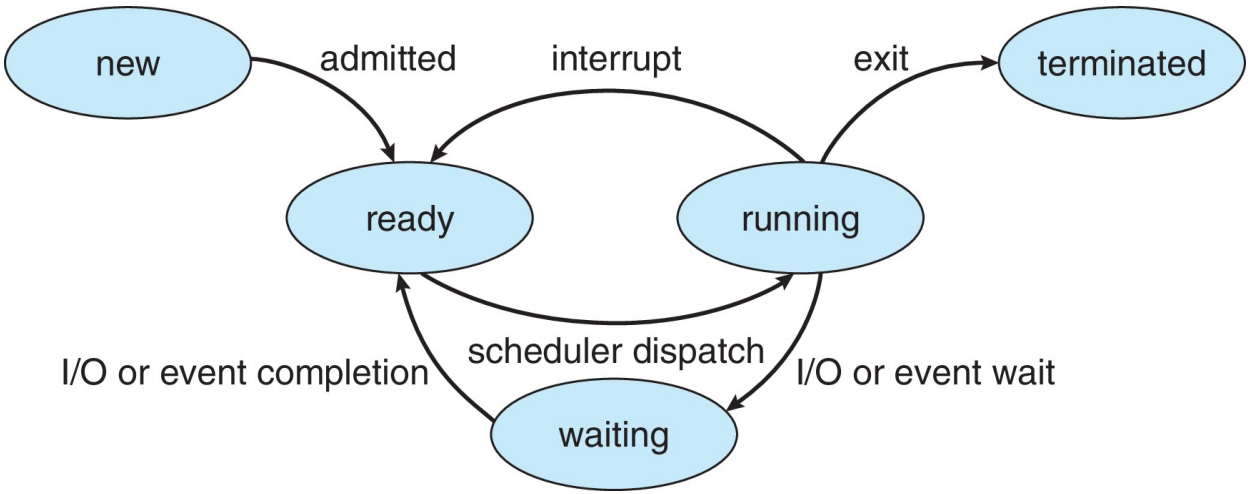
- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - **Multiple locations** can execute at once
    - Multiple threads of control -> threads
- Must then have storage for thread details, multiple program counters in PCB

Context Switch

- **Context switch**: the kernel switches to another process for execution
  - save the state of the old process
  - load the saved state for the new process
- **Context-switch is overhead**; CPU does no useful work while switching
  - the more complex the OS and the PCB, longer the context switch
- Context-switch time depends on hardware support
  - some hardware provides multiple sets of registers per CPU: multiple contexts loaded at once

Process State

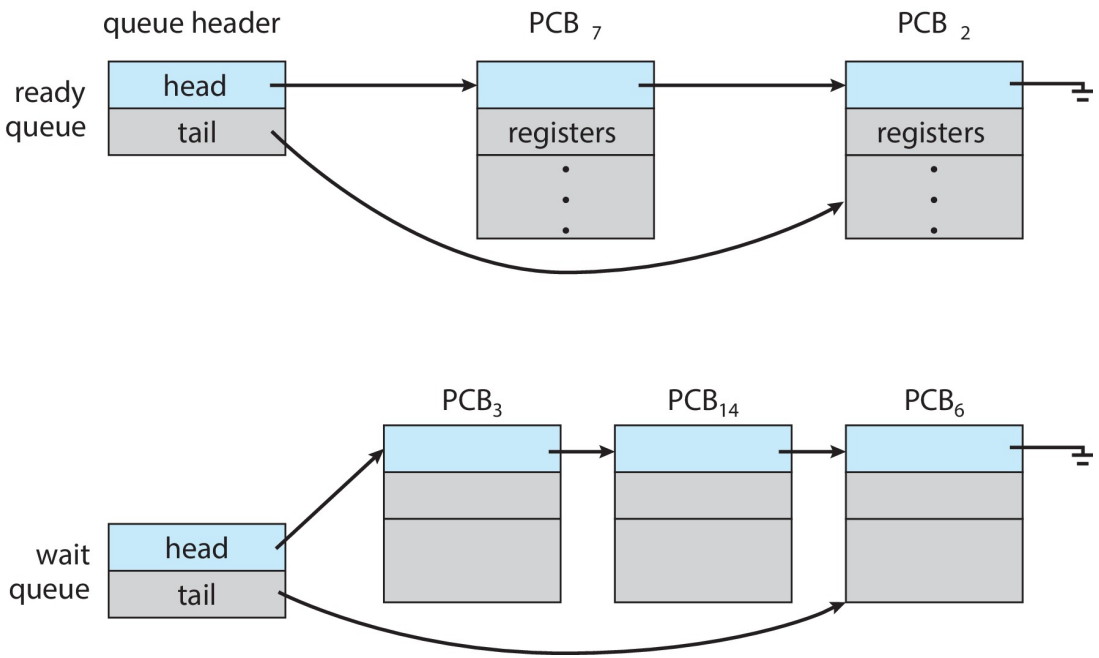
- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process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

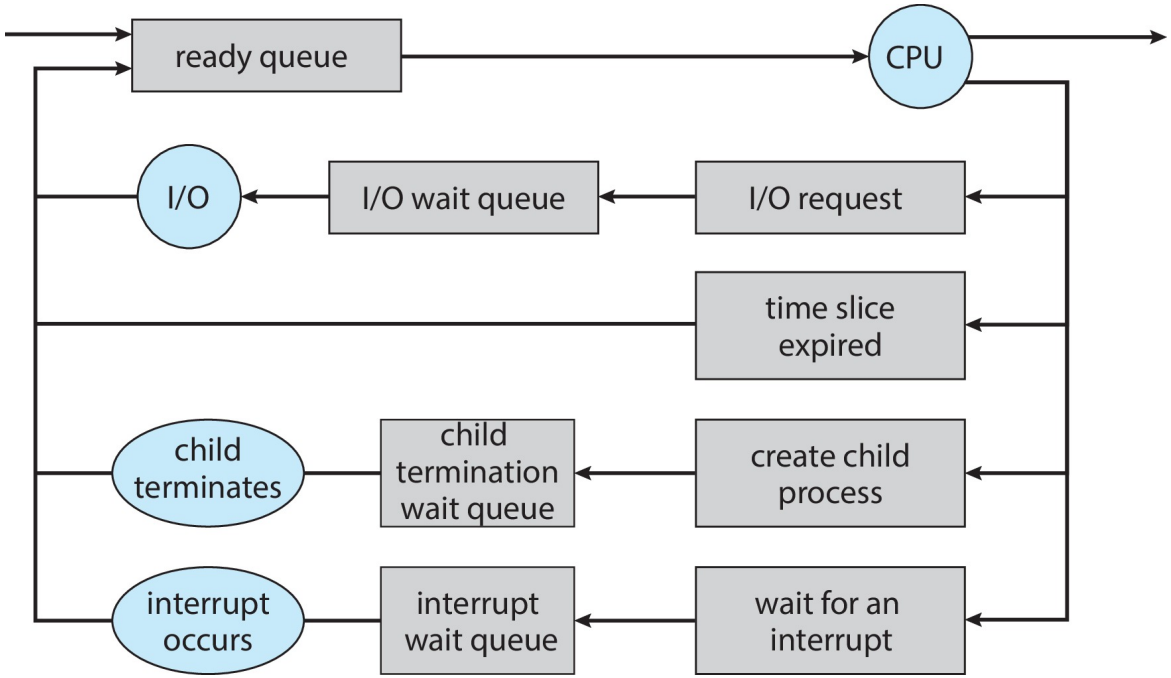
Ready and Wait Queues



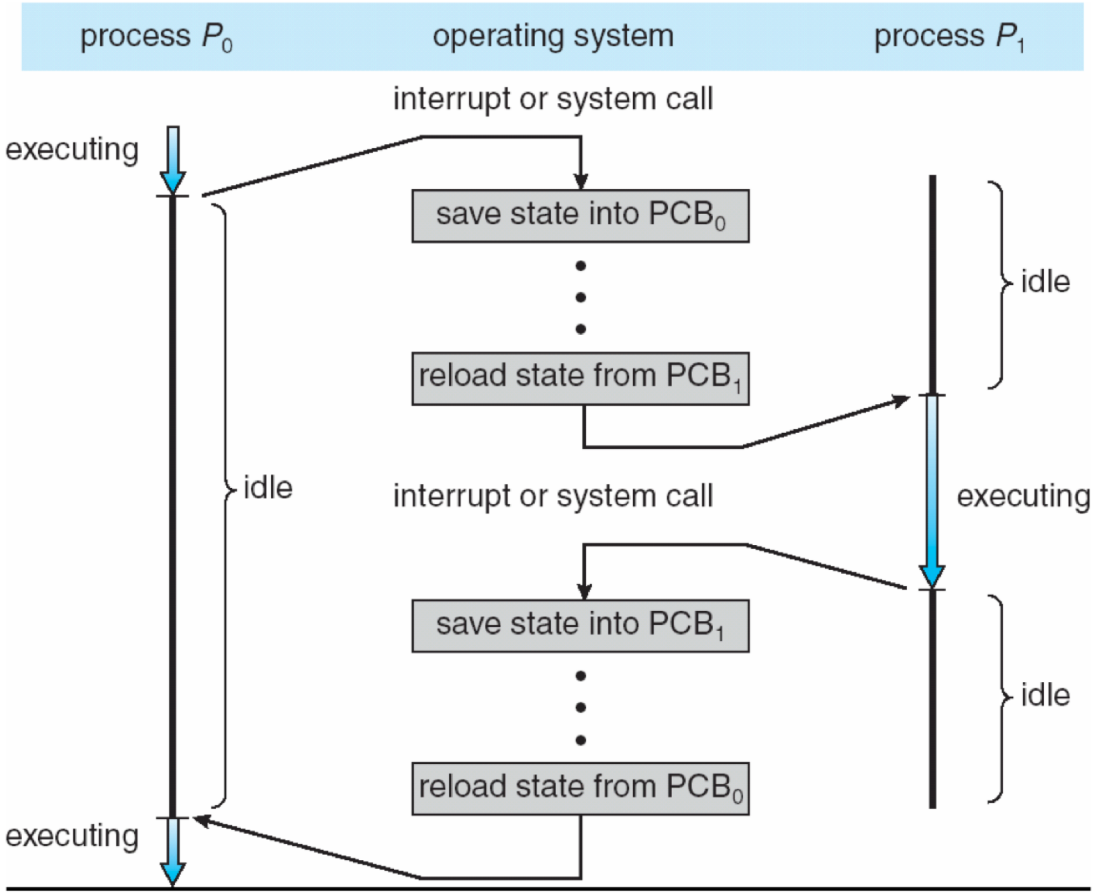

```
struct list_head {
        struct list_head *next, *prev;
};
```

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

Representation of Process Scheduling



Context Switch



Process Creation

- Parent process creates children processes, which, in turn create other processes, forming **a tree of processes**
  - process identified and managed via a process identifier (pid)
- Design choices:
  - three possible levels of **resource sharing**: all, subset, none
  - parent and children's **address spaces**
    - child duplicates parent address space (*e.g.*, Linux)
    - child has a new program loaded into it (*e.g.*, Windows)
  - **execution** of parent and children
    - parent and children execute concurrently
    - parent waits until children terminate

Process Creation

- UNIX/Linux system calls for process creation
  - **fork** creates a new process
  - **exec** overwrites the process' address space with a new program
  - **wait** waits for the child(ren) to terminate

C Program Forking Separate Process

```c
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
    pid_t  pid;
    pid = fork();                       /* fork another process */
    if (pid < 0) {                      /* error occurred while forking */
        fprintf(stderr, "Fork Failed");
        return -1;
    } else if (pid == 0) {              /* child process */
        execlp("/bin/ls", "ls", NULL);
    } else {                            /* parent process */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

fork() and Memory

- How many times does this code print "hello"?

```
pid1 = fork();
printf("hello\n");
pid2 = fork();
printf("hello\n");
```

Answer: 6 times

Popular Homework Question

- How many processes does this C program create?

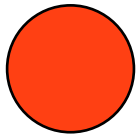
```
int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    }
    fork ();
}
```

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 ();
}
```

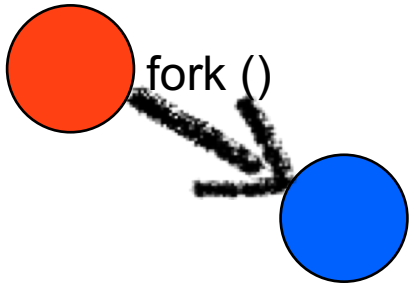


original process right when main begins

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?

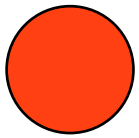
```
int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    }
    fork ();
}
```

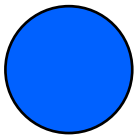
We now have two independent processes, each about to execute the same code

Popular Homework Question

- How many processes does this C program create?

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



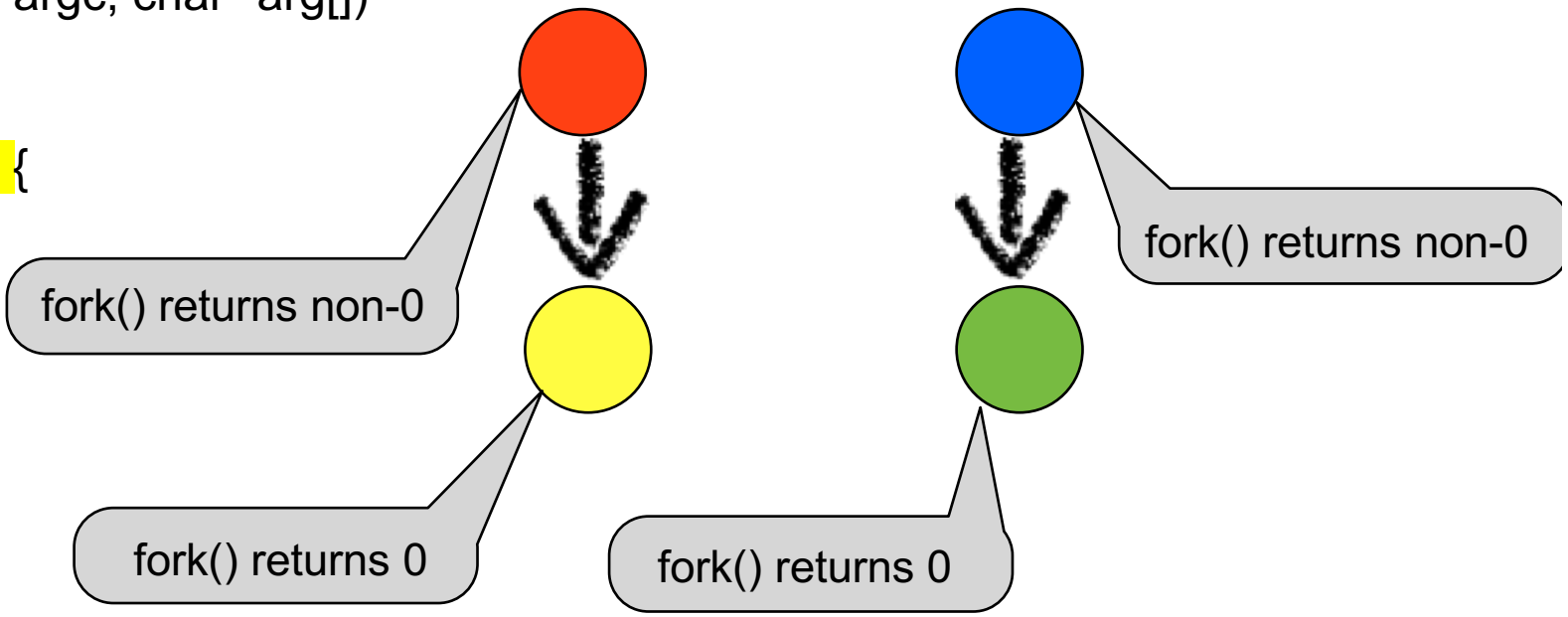


We now have two independent processes, each about to execute the same code
This code calls fork

Popular Homework Question

- How many processes does this C program create?

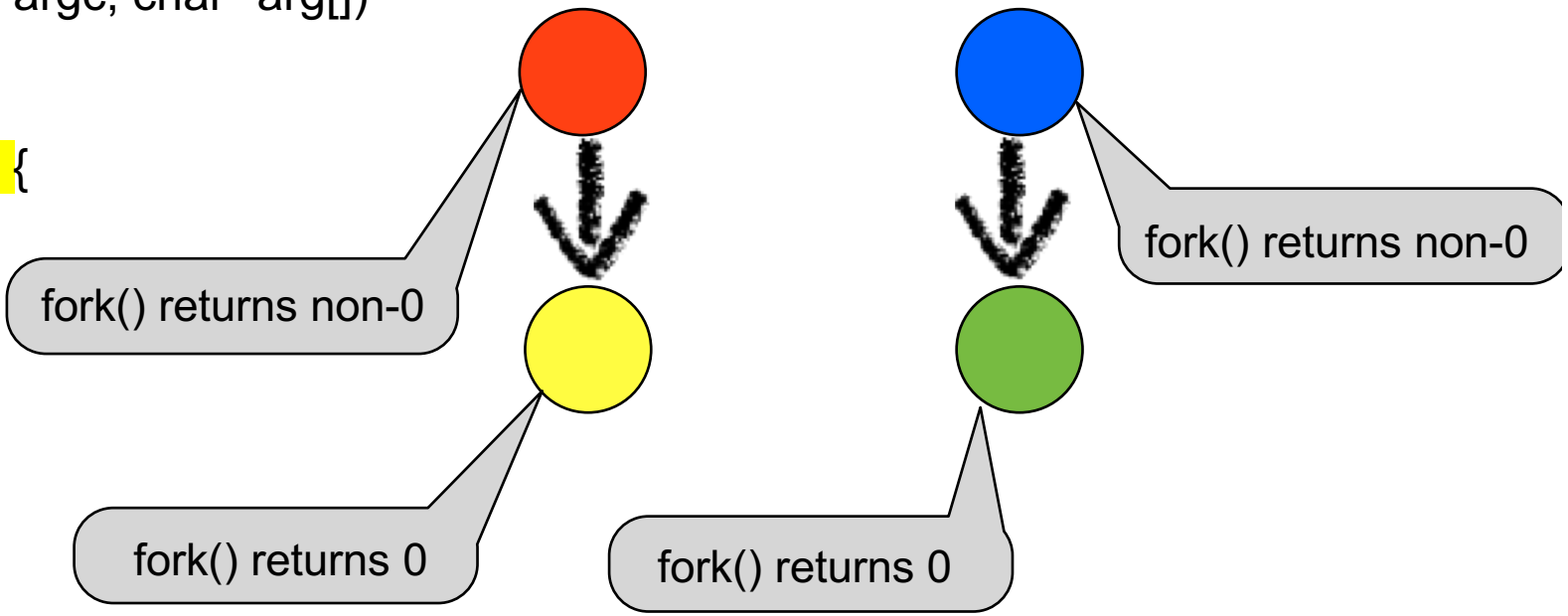
```
int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    }
    fork ();
}
```



Popular Homework Question

- How many processes does this C program create?

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

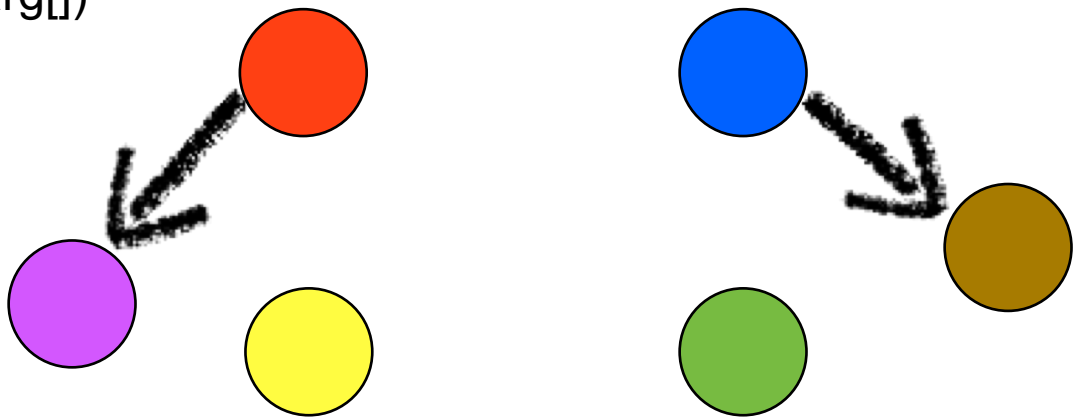


yellow and green: don't go into the if clause
red and blue: go!!

Popular Homework Question

- How many processes does this C program create?

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

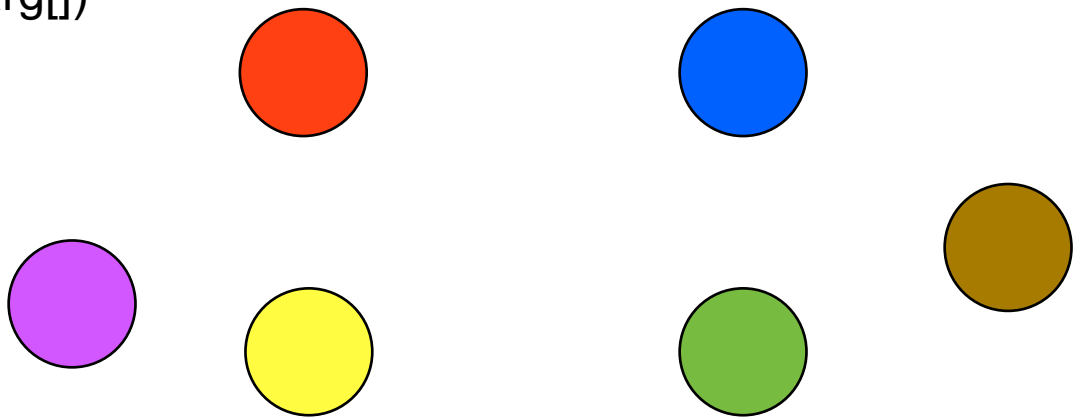


red and blue each creates a new child process (purple an brown)

Popular Homework Question

- How many processes does this C program create?

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

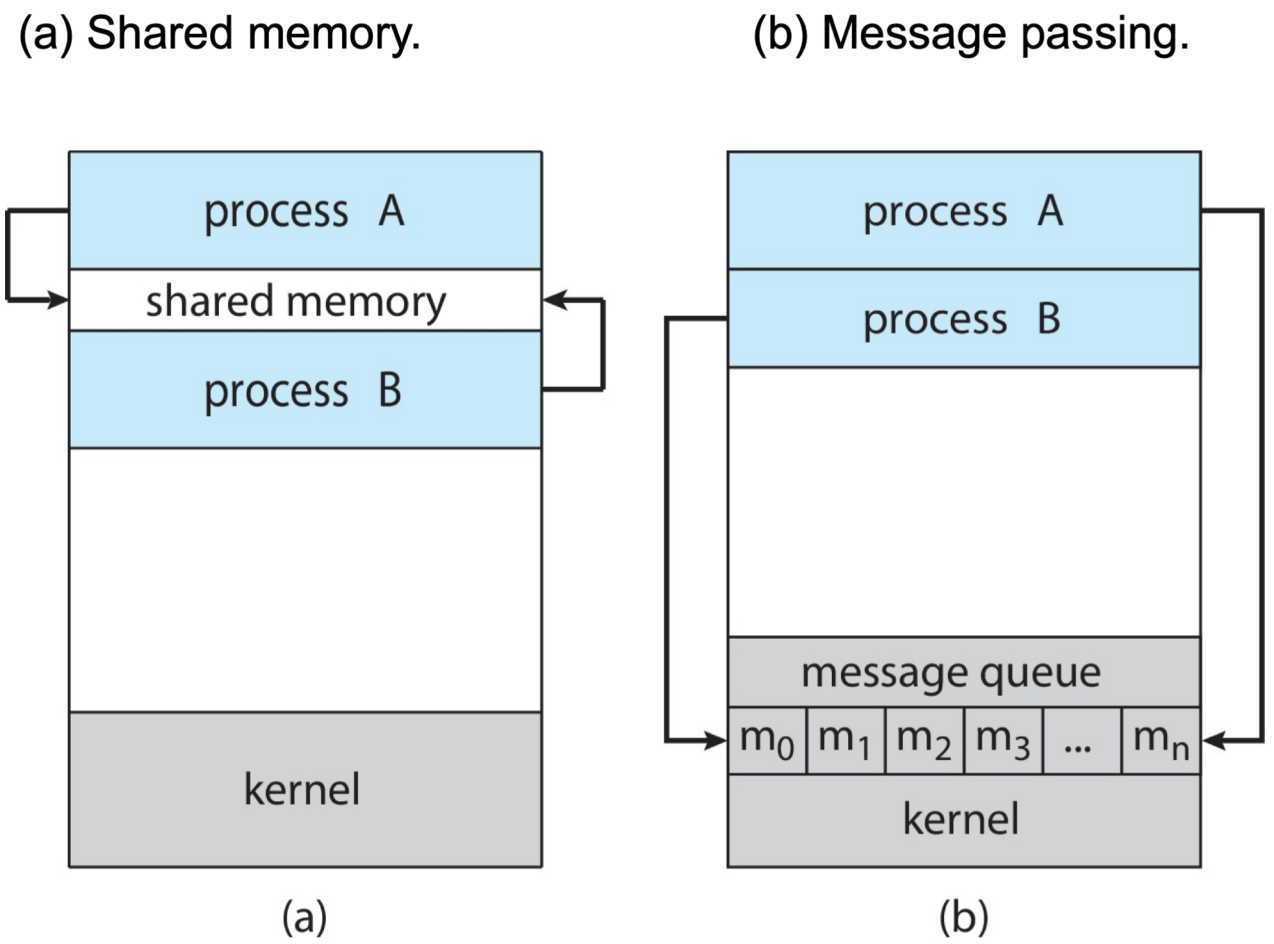


ALL processes execute the last call to fork()
red, purple, blue and brown after they exit from the if clause
yellow and green after they skip the if clause
We have 6 processes calling fork(), each creating a new process
So we have a total of **12 processes** at the end, one of which was the original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

Interprocess Communication

- Processes within a system may be independent or cooperating
  - **independent process**: process that cannot affect or be affected by the execution of another process
  - **cooperating process**: processes that can affect or be affected by other processes, including sharing data
    - reasons for cooperating processes: information sharing, computation speedup, modularity, convenience, Security
- Cooperating processes need **interprocess communication** (IPC)
- Two models of IPC
  - **Shared memory**
  - Message passing

Communications Models



Very fast!

POSIX Shared Memory

- POSIX Shared Memory
  - Process first creates shared memory segment
- shm_fd = **shm_open**(name, O CREAT | O RDWR, 0666);
  - Also used to open an existing segment
  - Set the size of the object: **ftruncate**(shm_fd, 4096);
  - Use **mmap**() to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by mmap().

A Simple Kernel Module

```c
/*
 *  hello-1.c - The simplest kernel module.
 */
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */

int init_module(void)
{
        printk(KERN_INFO "Hello world 1.\n");

        /*
         * A non 0 return means init_module failed; module can't be loaded.
         */
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_INFO "Goodbye world 1.\n");
}
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/printk.h>
#include <linux/sched.h>
#include <linux/sched/signal.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Yajin Zhou");
MODULE_DESCRIPTION("A simple example Linux module.");
MODULE_VERSION("0.01");


static int __init os_lkm_example_init(void) {
        struct task_struct *task;

        for_each_process(task)
                printk(KERN_INFO "%s [%d]\n", task->comm, task->pid);

    return 0;


}

static void __exit os_lkm_example_exit(void) {
        printk(KERN_INFO "Goodbye, World!\n");
}

module_init(os_lkm_example_init);
module_exit(os_lkm_example_exit);
```

insmod xxx.ko

Further reading: https://blog.sourcerer.io/writing-a-simple-linux-kernel-module-d9dc3762c234

Takeaway

- The whole slides are takeaways