Vocabulary: **Mode bit** It is a bit that represents or switches to user or kernel mode **System Program** a program associated with the operating system but not necessarily part of the kernel **System Call** the primary interface between processes and the operating system, providing a means to invoke services made availability by the operating system; a software-triggered interrupt allowing a process to request a kernel service. **Middleware** set of software frameworks that provide additional services to application developers **Bus** a communication system; within a computer, a bus connects various components, such as the CPU and I/O devices, allowing them to transfer data and commands. **Device Driver** an operating system component that provides uniform access to various devices and manages I/O to those devices. **Hardware Interrupt** a hardware mechanism that enables a device to notify the CPU that it needs attention **Interrupt Request Line** the hardware connection to the CPU on which interrupts are signaled. **Interrupt-handler Routine** called when an interrupt signals is received. **Interrupt Vector** an operating-system data structure indexed by interrupt address and pointing to the interrupt handlers; a kernel memory data structure that holds the addresses of the interrupt service routines for the various devices. **Interrupt-controller Hardware** computer hardware components for interrupt management. **Maskable Interrupt Line** for interrupts that can be delayed or blocked (e.g., when the kernel. **Non-maskable Interrupt Line** for interrupts that cannot be delayed or blocked (e.g., an unrecoverable memory error) **Interrupt Chaining** A mechanism by which each element in an interrupt vector points to the head of a list of interrupt handlers, which are checked individually until one is found to service the interrupt request. **Interrupt Priority Level** a prioritization of interrupts to indicate handling order. **RAM, Dynamic RAM, ROM, | Electrically Erasable Programmable ROM (EEPROM)** storage that is infrequently written to and is nonvolatile. **Bootstrap Program** the program that allows the computer to start running by initializing hardware and loading the kernel. **System Daemons** a program that waits in the background to be activated by a stimulus. **Signal** a software-generated interrupt that notifies a process that an event has occurred **Exception (trap)** a software-generated interrupt caused by an error or by a specific request from a user program that an operating-system service be performed by executing a special operation called a **System Call** request for operating system service. **Dual mode** (operation) allows OS to protect itself & other system components (i.e. **user mode** and **kernel mode**). **Asymmetric multiprocessing** Each processor is assigned one specific task. A *boss processor* schedules and allocate work to the *work processors* **Symmetric multiprocessing** Most common, each processor performs all tasks, they work as *peers*. **Clustered systems** another type of multiprocessor systems, composed of two or more individual systems, or nodes, joined together. **Privileged Instructions** term to define some of the machine instructions that may cause harm. Used to manage interrupts, and only executed in *kernel mode* **Program Counter** specifies the current instruction to execute **Timer** can be set to interrupt the computer after a specified period. **Process** is the unit of work in a system. **File** logical storage unit. **Tertiary storage** magnetic tapes, CD, DVD, **WORM** Write-once, read-many-times. **Process Management Activities** (OS Activities include) creating/deleting both user and system processes, suspending and resuming processes, providing mechanisms for process synchronization, providing mechanisms for process communication, providing mechanisms for deadlock handling **Memory Management** *determines what is the memory & optimizing CPU utilization & computer response to users* OS Activities include keeping track of which parts of memory is being used & by whom, deciding which processes & data to move in/out of memory, (de)allocating memory space (as needed). **File-System Management** *files organized into directories & access control determine who can access them* (OS Activities include) creating/deleting files/directories, primitives to manipulate files/directories, mapping files onto secondary storage, backup files onto stable (non-volatile) storage media **Mass-Storage Management** (OS Activities) (un)mounting, free-space management, storage allocation, disk scheduling, partitioning, protection **Caching** information in use is copied from slower to faster storage temporarily. the cache is smaller than the storage being cached. **Cache management** Careful selection of the cache size and of a replacement policy can result in greatly increased performance **I/O Subsystem** consist of several components a memory-management component that includes buffering, caching, and spooling; a general device-driver interface; drivers for specific hardware devices. **Protection** any mechanism for controlling the access of processes or users to the resources defined by a computer system. **Security** defend the system from external and internal attacks. **Mobile computing** refers to computing on handheld smartphone and tablet computers. **Virtualization** is a technology that allows operating systems to run as applications within other operating systems.

Compile to Assembly:

```
gcc -std=c17 -Wall -pedantic-errors -g -O0 -S filename.c
```

`-Wall` - all warnings
`-g` - debug mode; include symbol table
`-O0` - optimization level 0; no optimizations
`-S` - compile to assembly (alternatively, use `-c` to compile directly to machine code)
`-c` - compile without linking

Assemble to Machine Code:

```
as -o filename.o filename.s
```

`-o filename.o` output to filename.o

Link into Executable:

```
-$ gcc -o filename filename.o other.o
```

Load and Execute:

```
$./filename
```

- **Anatomy of a process**

As processes enter the system, put into the **ready queue**, where they are ready & waiting to execute on CPU.
(generally stored as a linked list)
processes waiting for an event to occur placed in a **wait queue**.

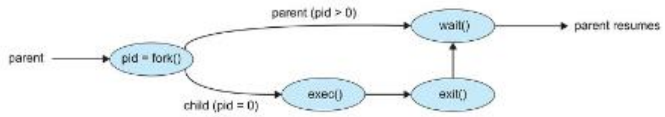
Process Termination

exit (3) - normal process termination
_exit(2) - immediate process immediately
atexit(3) - register function to be called at normal process termination
on_exit(3) - register a "fancy" function to be called at normal process termination
SIGCHILD

Process Creation

fork(2) - create a child process; separate address space than parent
vfork(2) - create a child process; same address space as parent wait (2) - wait for process to change state

waitpid(2) - suspend execution of the calling process unt one of its children terminates.



function calls and system calls -- both in C and x86_6 assembly. Three general methods are used to pass parameters to the operating system**.** The simplest appro is to pass the parameters in registers. In some cases, however, there may be more parameters than registers. these cases, the parameters are generally stored in a bl or table, in memory, and the address of the block is pass as a parameter in a register. Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the operating system. System calls can be grouped roughly into six major categories.

Process control: A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()).

File manipulation: We first need to be able to create() and delete() files. Once the file is created, we need to open() it and to use it. We may also read(), write(),or reposition()(rewindorskiptotheendofthefile,for example). Finally, we need to close() the file.

Device manipulation: A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process (request device, release device, get device attributes, set device attributes)

information maintenance: Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example a system call to return the current time() and date().(get system data, set system data)

Communications: There are two common models of interprocess communication: the message passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to transfer information**.** The source of the communication, known as the client, and the receiving daemon, known as a server, then exchange messages by using read message() and write message() system calls. The close connection() call terminates the communication. In the **shared-memory model**, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes. normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. **Protection:** provides a mechanism for controlling access to the resources provided by a computer system

-

To invoke a system call:

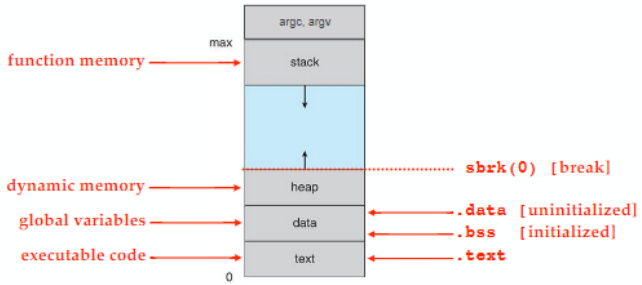
1. Set `%rax` to the system call number (see Full Table below);
2. Set `%rdi`, `%rsi`, `%rdx`, `%r8`, `%r10`, `%r9` (in order) according to the `glibc` prototype given in the manual; then
3. Perform `syscall` instruction.

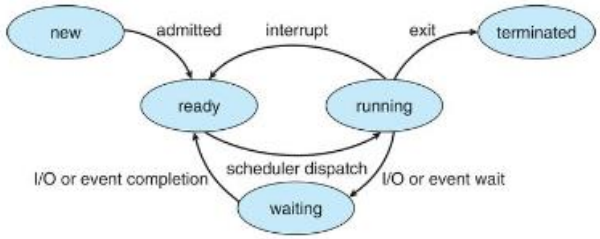
Instruction	Example	Pretend it's C	Notes
`mov`	`movq x, %rax`	`rax = x;`	
`inc`	`incq %rax`	`++rax;`	
`dec`	`decq %rax`	`--rax;`	
`add`	`addq %rbx, %rax`	`rax = rax + rbx;`	
`sub`	`subq %rbx, %rax`	`rax = rax - rbx;`	
`imul`	`mulq %rbx, %rax`	`rax = rax * rbx;`	
`and`	`andq %rbx, %rax`	`rax = rax & rbx;`	
`xor`	`xorq %rbx, %rax`	`rax = rax ^ rbx;`	
`shr`	`shrq $4, %rax`	`rax = rax >> 4;`	unsigned
`shl`	`shlq $5, %rax`	`rax = rax << 5;`	unsigned
`sar`	`shrq $4, %rax`	`rax = rax >> 4;`	signed
`sal`	`shlq $5, %rax`	`rax = rax << 5;`	signed
`imul`	`imul $0x10, %rax`	`rax = rax * 16;`	

You can use `call` and `ret` to transfer control between functions.

Instruction	Example	Pretend it's C	Notes
`call`	`callq foo`	`foo();`	automatic `pushq %rip`
`ret`	`retq`	`return;`	automatic `popq %rip;`

Process Memory





The order of the parameter registers is different for system calls!

Parameter	Function Call	System Call
1	`%rdi`	`%rdi`
2	`%rsi`	`%rsi`
3	`%rdx`	`%rdx`
4	`%rcx`	`%r8`
5	`%r8`	`%r10`
6	`%r9`	`%r9`