Operating Systems — Introduction and Processes

ECE_3TC31_TP/INF107

Stefano Zacchiroli

2024

3TC31 — Where are we in our Trip

3TC31: “from the logic gate to the operating system”

- Part 1: *logic gate* → *processor*
- Part 2: *processor* → *system programs* (C programming language)
- Part 3: *system programs* → *operating system* ⇐ we are here

Goals of Part 3:

- provide an overview of what **Operating Systems (OS)** *do*,
- *how* OS work internally and how to implement one.

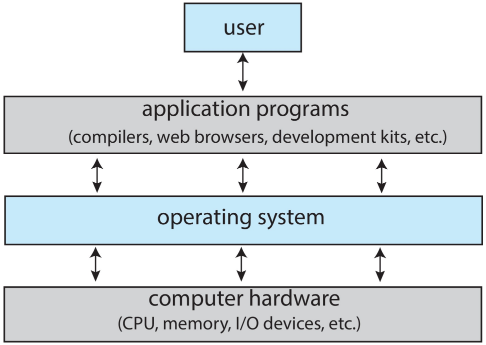
What Operating Systems Do

Warm-up Quiz

- Q: what’s an operating system (in your own words)?
- A: ⟨*your answer here*⟩

What is an Operating System? (Intuition)

- A **program** (= software) that acts as an **intermediary** between a *user* of a computer and the computer *hardware*


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

- Operating system **goals**:
  - *Execute user programs* and make solving user problems easier
  - Make the computer system *convenient to use*
  - Use the computer hardware in an *efficient* manner

What Operating Systems Do

- Depends on the **point of view**
  - **Users** want convenience, ease of use and good performance
  - Don't care about resource utilization
- But shared computer such as mainframe or minicomputer must *keep all users happy*
  - Operating system is a **resource allocator** and control program *making efficient use of hardware* and managing execution of user programs
- **Resources are scarce** in many contexts for different reasons
  - Servers: many users, need to share resources between them
  - Mobile devices: optimize for battery life
  - Embedded devices: limited hardware

Operating systems arbiter the allocation of scarce resources[1] to demanding users, in the best possible way.[2]

[1] Hardware resources for the most part, but also software resources.

[2] For some precise measure of "best".

What is an Operating System? (Definitions)

- No universally accepted definition

- *"Everything a software vendor ships when you order an OS"* is a good approximation
  - But varies wildly
- *"The one program running at all times on the computer"* is the **kernel**, *part* of the OS
- Everything else is either:
  - A **system program**[3] (ships with the OS, but is not part of the kernel), or
  - An **application program**, all programs not associated with the OS, but that rely on it for execution

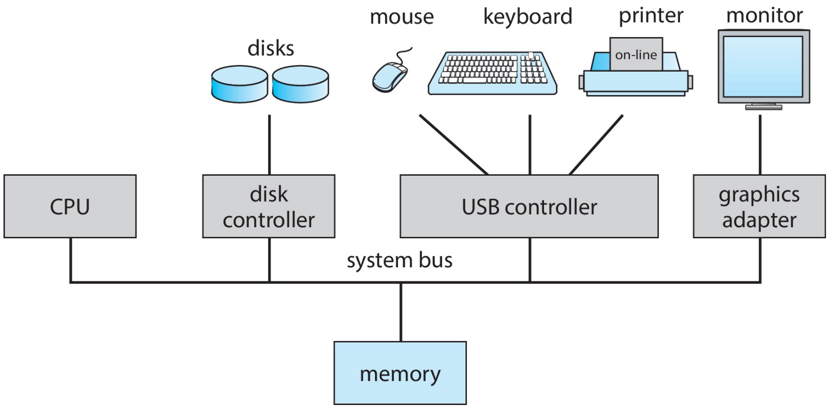
- Today's OSes for general purpose and mobile computing also include **middleware** — a set of software frameworks that provide additional services to application developers such as databases, multimedia, graphics

[3] cf. 3TC31, part 2

Basics of Computer System Structure

Computer System Organization and the Bus

- One or more **CPUs** and **device controllers** connect through a common **system bus** providing access to a shared main memory
- **Concurrent execution** of CPUs and devices, who are competing for memory cycles

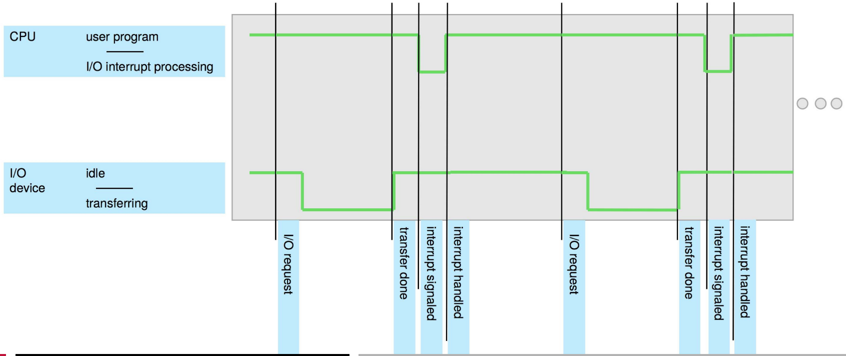


Device Controllers and Interrupts

- Each device controller is in charge of a particular device type
- Each device controller has a **local buffer**
- Each device controller type has an operating system **device driver** (= software) to manage it
- CPU moves data: main memory $\leftrightarrow$ local buffers of controllers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an **interrupt**

Interrupts

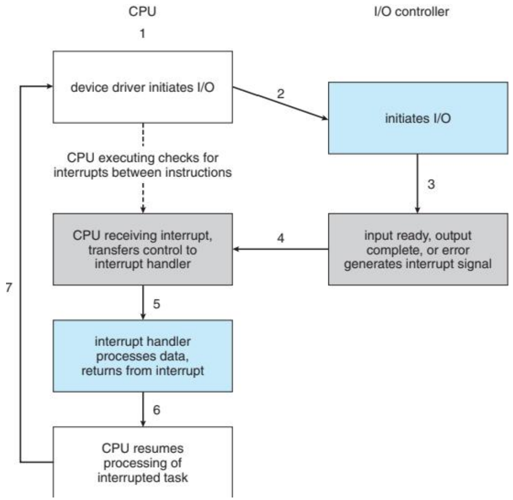
- Interrupt transfers control to the **interrupt service routine** generally, through the interrupt vector, which contains the addresses of all the service routines
- Interrupt architecture must **save the address of the interrupted instruction** (to return to it later)
- A **trap or exception** is a software-generated interrupt caused either by an error or a user request
  - They are handled the same way than I/O interrupt
- Modern operating systems are mostly *interrupt-driven*



Interrupts (cont.)

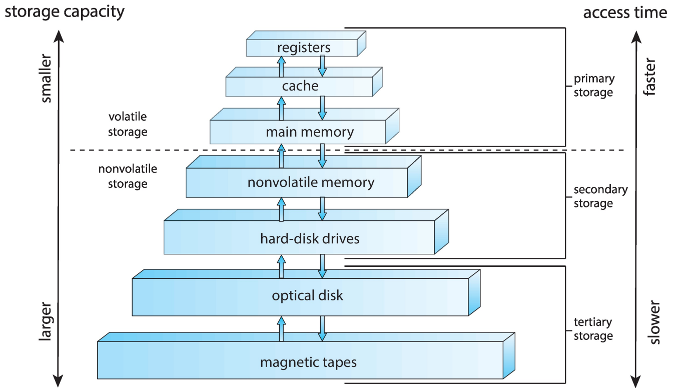
- Upon receiving an interrupt, the OS preserves the **state of the CPU** by storing the *registers and the program counter (PC)*
- Determines which type of interrupt has occurred
- Separate segments of code determine what action should be taken for each type of interrupt

A typical I/O scenario hence corresponds to the workflow shown on the right.

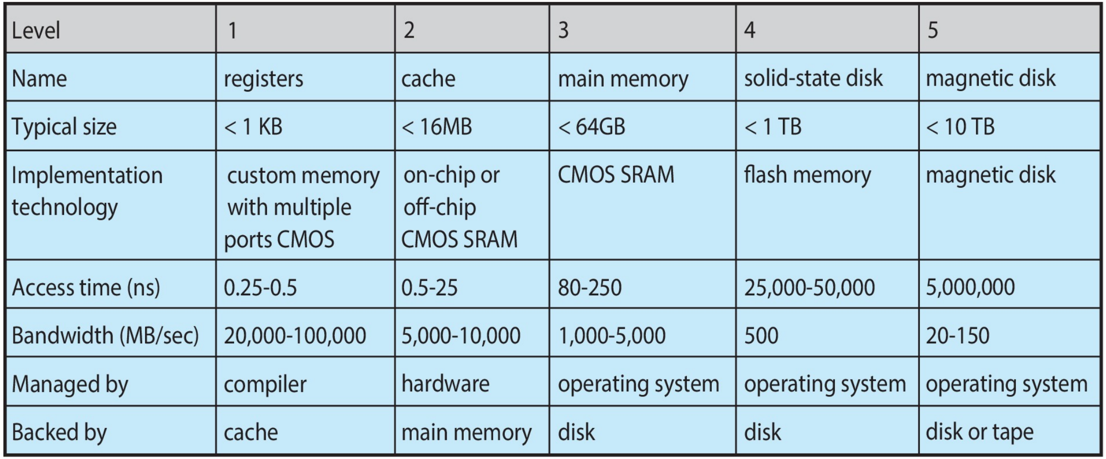

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Storage Hierarchy

- Storage is organized in a **hierarchy** with varying: speed, cost, volatility
  - **Main memory**: only large storage that *CPU can access directly*
  - **Secondary storage**: large nonvolatile storage capacity. Main types: hard disk drives (*HDD*), non-volatile memory (*NVM*)
  - **Tertiary storage**: even larger and slower (e.g., for backup purposes)



Storage Characteristics



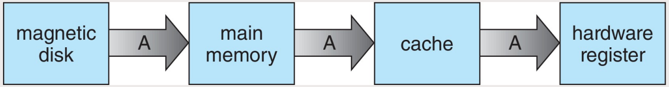
Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Caching

- Important principle, performed at many levels in a computer (hardware, OS, software)
- Information in use **copied from slower to faster storage** temporarily
- Faster storage **(cache) checked first** to determine if information is there
  - If it is (*cache “hit”*), information used directly from the cache (fast)
  - If not (*cache “miss”*), data copied to cache (slow) and used from there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

Example

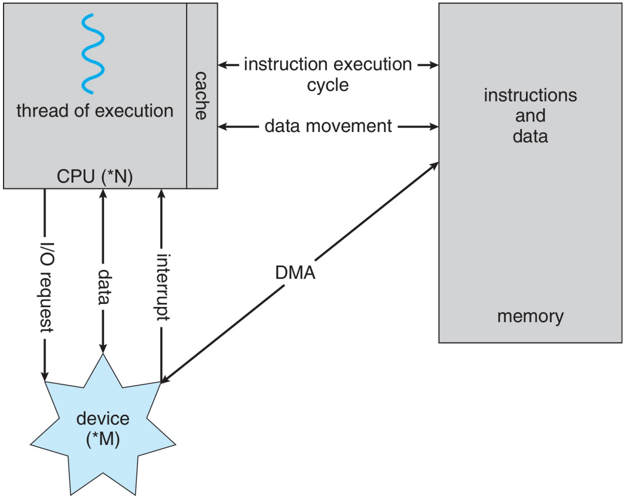
The path of an integer x from disk to register, where the CPU can actually do something with it:



How a Modern Computer Works

Modern general-purpose computers:

- Implement the **von Neumann architecture**, where memory contains both data and instructions, interpreted one way or another by the CPU
  - if the PC points to it → it's an instruction
- Allow devices to read/write memory directly (**Direct Memory Access**, or DMA) to reduce bus contention



Multiprocessor Systems

- Most systems use a single general-purpose processor
  - Plus several special-purpose processors, e.g., in device controllers
- **Multiprocessors** systems are growing in use and importance
  - Also known as parallel, tightly-coupled systems
  - Advantages:
    1. Increased throughput
    2. Economy of scale
    3. Increased reliability (e.g., fault tolerance)
- Two types:
  - **Asymmetric multiprocessing** – each processor is assigned a special task
  - **Symmetric multiprocessing** – each processor can perform any task

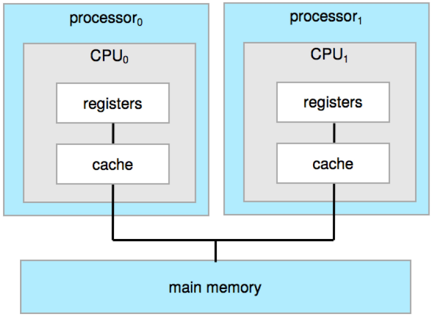


Figure: a symmetric multiprocessing architecture

Multicore Systems

- Each physical processor *chip* (sometime confused with the term “CPU”) can host one or more units capable of executing CPU instructions at a time, called **core**
- A chip containing more than one core is called **multicore**
- Can mix and match multiprocessor and multicore in the same system
  - E.g., current high-end laptop: 1 processor, 14 cores
  - E.g., current high-end server: 4 processors, 24 cores each
- Note: only with more than one core (no matter if on the same chip or different ones) there can be parallelism, i.e., more than one CPU instructions executed *at the same time*

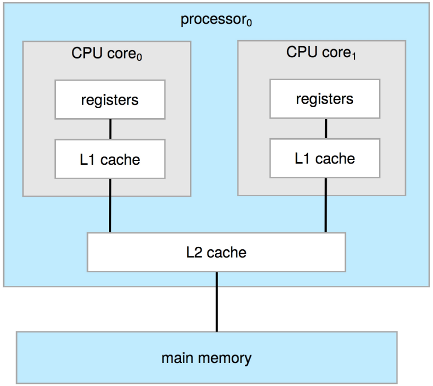


Figure: a single-processor, dual-core (= two cores) architecture

Multiprogramming and Multitasking

Multiprogramming (batch systems)

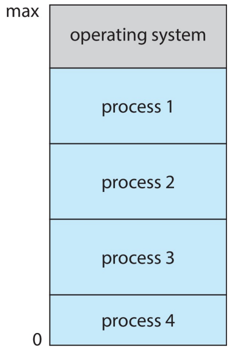
- A single user cannot always keep CPU and I/O devices busy
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute
- A subset of total jobs in system is kept in memory
- One job selected and run via **job scheduling**
- When job has to wait (for I/O for example), OS switches to another job

Multitasking (timesharing)

A logical extension of multiprogramming — the CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive computing**

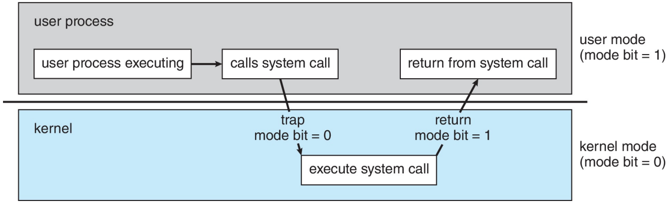
- Response time should be short (<< 1 second)
- Each user has at least one program executing in memory → **process**
- If several jobs ready to run at the same time → CPU scheduling
- If processes don't fit in memory, swapping moves them in and out to run
- Virtual memory allows execution of processes not completely in memory

Memory layout: code+data of OS and all executing programs is in memory.



Dual-mode Operation

- Dual-mode operation allows the OS to *protect itself and other system components*
  - **User mode** and **kernel mode**
- **Mode bit** provided by *hardware*
  - Provides ability to distinguish when system is running **user code** or **kernel code**
  - When user code is running → mode bit is “user”
  - When kernel code is executing → mode bit is “kernel”
- Some **instructions are designated as “privileged”** and only executable in kernel mode
- How do we guarantee that user code does not set the mode bit to “kernel”?
  - User code can requests system services only by invoking **system calls** (more on this later)
  - System calls change mode to kernel, return from call resets it to user automatically



Operating System Responsibilities

Operating System Responsibilities

An operating system has several **responsibilities**, which we briefly present in the following. We will expand upon most of them later in the course of 3TC31, so we will *skim through them quickly* for now.

Several of OS responsibilities belong to the general area of **managing resources** that executing programs need:

- CPU, memory, file-system, I/O

Other OS responsibilities are more general and **cross-cutting**, such as:

- Protection and security
- Virtualization

Process Management

- A **process** is *a program in execution* (more on this later). It is a unit of work within the system.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources

OS activities for process management

- Creating and terminating processes
- Suspending and resuming processes
- Providing mechanisms for:
  - Process synchronization
  - Process communication
  - Deadlock handling (more on this later)

Memory Management

- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory too
- Memory management determines what is in memory and when

OS activities for memory management

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

File-system Management

OS provides uniform, logical view of information storage:

- Abstracts physical properties to logical storage unit: file
- Each medium is controlled by device (i.e., disk drive, tape drive)
- Files usually organized into directories
- Access control to determine who can access what

OS activities for file-system management

- 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

I/O Management

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices

Protection and Security

- **Protection:** any mechanism for controlling access of processes or users to resources defined by the OS
- **Security:** defense of the system against internal and external attacks
  - Huge range, including: denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** (controlled) allows user to change to effective ID with more rights

Virtualization

- Allows operating systems to run applications within other OSes
  - Vast and growing industry
- **Emulation** used when source CPU type different from target type (i.e., PowerPC to Intel x86)
  - Generally slowest method
  - When computer language not compiled to native code — **Interpretation**
- **Virtualization** — OS natively compiled for CPU, running **guest OS** also natively compiled
  - E.g., VMware running WinXP guests, each running applications, all on native WinXP **host OS**
  - **VMM** (Virtual Machine Manager, part of the OS) provides virtualization services

Operating System Services

Operating System Services

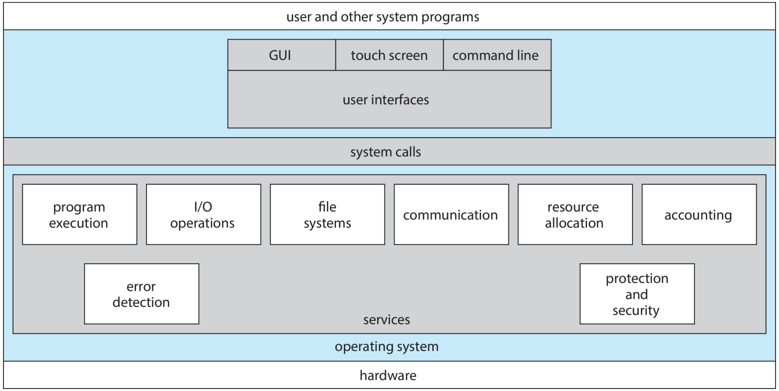
Operating systems provide a number of **services** to users and running programs

Services for users:

- User interfaces: CLI, GUI, touch screen
- Program execution

Services for running programs:

- I/O, file-system ops.
- Communication between programs (locally or via the network)
- Resource allocation, error detection
- Accounting, protection, security



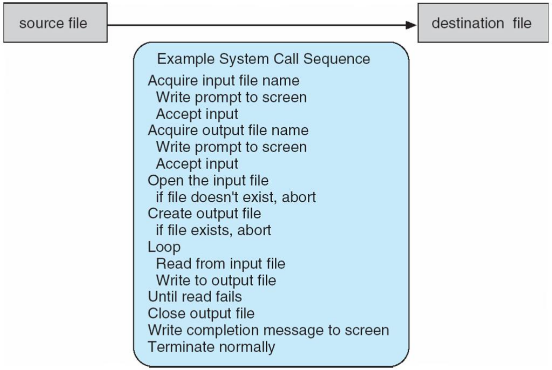
System Calls

Running programs request OS services by invoking **system calls** (or *syscalls*, for short).

- Programming interface to the services provided by the OS
- Typically written in a system-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** implemented by system libraries (e.g., the **C standard library**, or `libc`) rather than direct syscall invocation
- Common high-level APIs for syscalls:
  - Win32 API for Windows
  - POSIX API for UNIX systems (including Linux and Mac OS)
  - (subset of) Java API for the Java Virtual Machine (JVM)

System Calls — Example

Consider a program that interactively asks the user interactively for two file names and copies the content of one file to the other. How many system call (invocations) are involved in such a task?

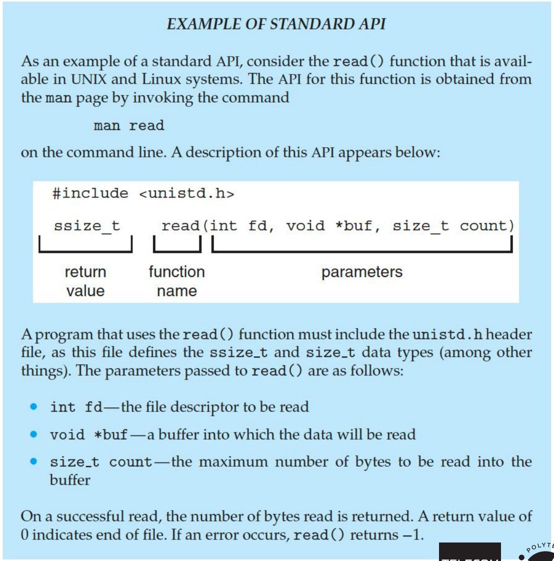


Try it out for yourself by running `strace cp input_file output_file` in a terminal.

Bottom line: *a lot* of what running programs do is invoking OS services.

System Calls — Program, libc, OS — Example (1)

- `read` is a standard POSIX system call that requests the service of reading content from a file into a memory buffer of the requesting program (e.g., a byte array)
  - (You will learn about `read` details later.)
- The `read` *system call* can be invoked by C programs by calling the `read` *function* implemented in the libc
- `read` is a *blocking* system call; calling program suspends its execution, waiting for completion



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>
```

$$\underbrace{\texttt{ssize\_t}}_{\text{return value}} \quad \underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

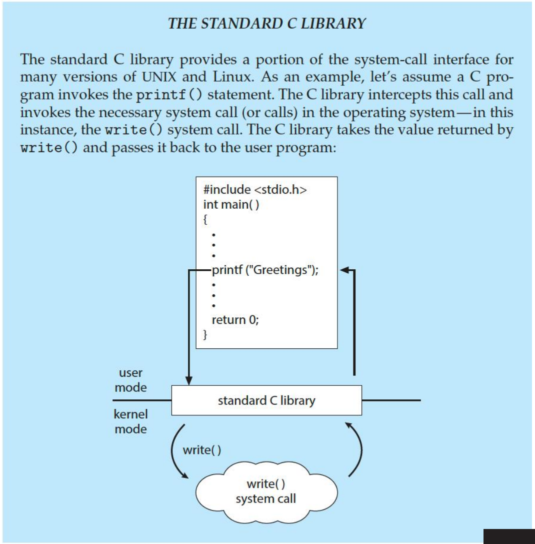
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Calls — Program, libc, OS — Example (2)

- Other, higher-level functions of the C standard library (and other libraries) are not 1-1 mappings to system calls, but call into system calls nonetheless
- For example, `printf` uses `write` (the complementary system call of `read`) to write formatted output to standard output (usually connected to your terminal)
- Note that user code (program and libc) executes in **user mode** whereas system call code executes in **kernel mode**

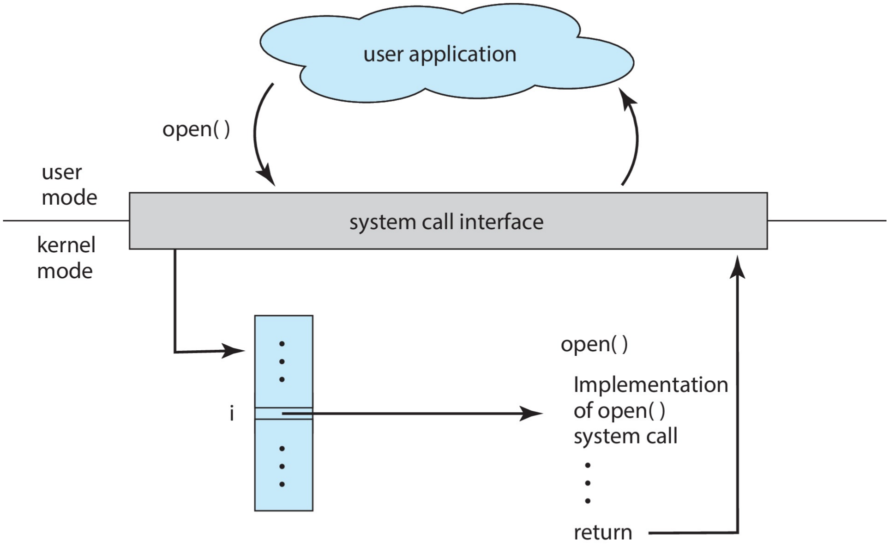


THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:

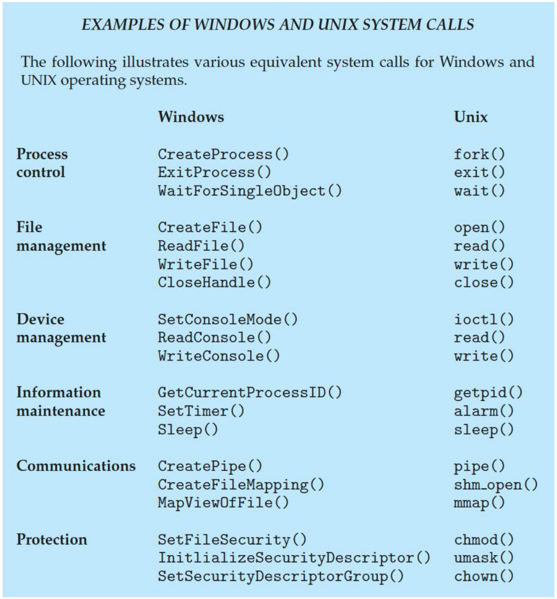
```
#include <stdio.h>
int main( )
{
  .
  .
  .
  printf ("Greetings");
  .
  .
  .
  return 0;
}
```

System Calls — User and Kernel Mode



Types of System Calls

- Many classes of system call exist, depending on the type of service requested
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications
  - Protection
- The sets of available system calls vary across OS
- You will learn about several (UNIX) system calls in the lab sessions of 3TC31



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Processes

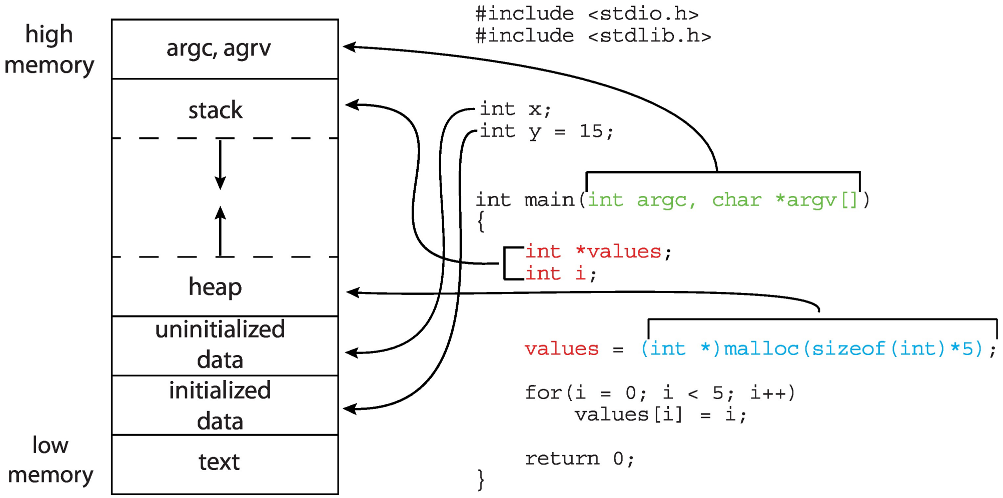
Process — Concept

- An operating system executes a variety of programs that run as processes
- A **process** is a *a program in execution*
- Process execution must progress in sequential fashion
- A process consists of multiple parts (already seen in Part 2 of 3TC31):
  - The **program code**, also called text section
  - Current activity including program counter and other processor **registers**
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Program vs Process — Intuition

- A program is a **passive entity** stored on disk (executable file)
- A process is an **active entity** in execution
- Program turns into a process when an executable file is loaded into memory for execution
- One program can correspond to several processes (e.g., multiple users executing same program)

Memory Layout of a C Program (Redux)



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

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

(Just a reminder, you’ve seen this before.)

Process States

As a process executes it changes **state:**

- **New:** being created
- **Running:** its instructions are being executed on a processor
- **Waiting:** waiting for some event to occur (cannot be executed, temporarily)
- **Ready:** waiting to be assigned to a processor
- **Terminated:** finished execution

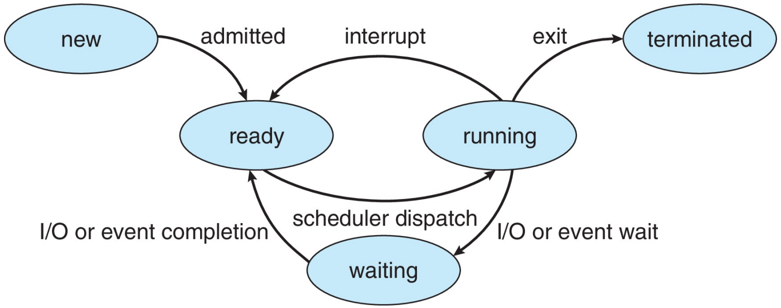
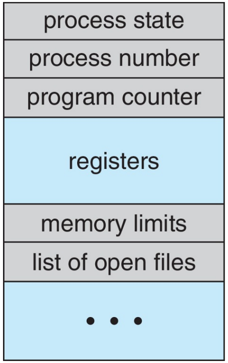


Figure: process state machine

Process Control Block (PCB)

The full status of a process is captured by its **Process Control Block** (PCB), which contains:

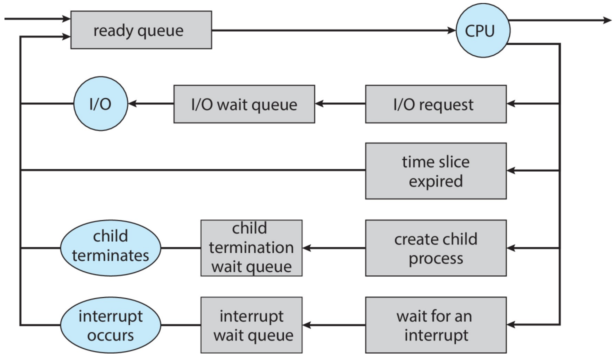
- Process state — running, waiting, etc.
- Program counter — location of instruction to next execute
- CPU registers — contents of all process-centric registers
- CPU scheduling information — priorities, scheduling queue pointers
- Memory-management information — memory allocated to the process
- Accounting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



All process PCBs are maintained by the OS using dedicated data structures.

Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal: maximize CPU use; Implementation: quickly switch processes on/off CPU core(s)
- Maintains **scheduling queues** of processes
  - **Ready queue:** set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues:** (plural!) set of processes waiting for an event (e.g., I/O, process termination, etc.)
  - Processes migrate among the various queues as they change state



Context Switch

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

- To execute a context switch, the OS must **save the state** (or "context") of the *old process* and load the **saved state** for the *new process*
- Full context of a process represented in the PCB
- Context-switch time is **pure overhead**; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch

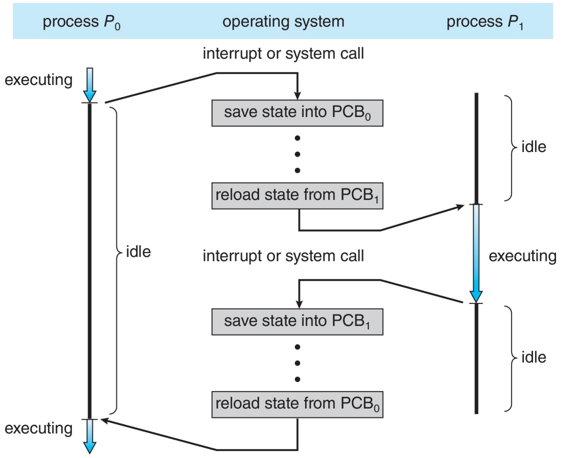


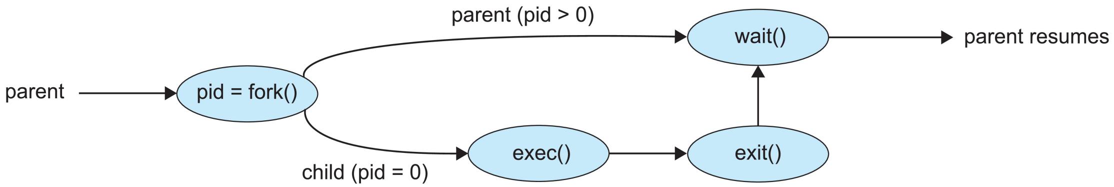
Figure: context switch timeline (from top to bottom)

Process Creation

- **Parent process create children processes**, which, in turn create other processes, forming a **tree of processes**
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options (depending on the OS):
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options (ditto):
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space options (ditto):
  - Child duplicate parent's address space
  - Child has a (new) program loaded into it

Process Creation on UNIX — Example

- `fork()` system call creates new process
  - Child shares some parent's resources (e.g., open files)
  - Parent and child execute concurrently
  - Child duplicates parent's address space
- (optional) `exec()` system call used after a fork() to replace the process address space with a new program
- (optional, for coordination) Parent process calls `wait()` system call to wait for the child to terminate



(More on this in the upcoming 3TC31 lab session.)

Process Creation on UNIX — Example (cont.)

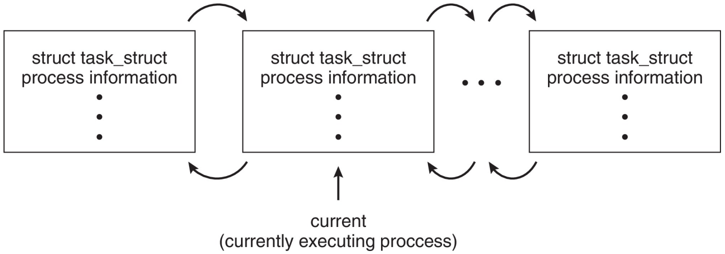
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t pid;
    pid = fork();  /* create a child process */
    if (pid < 0) {  /* fork() syscall failed */
        fprintf(stderr, "E: Fork failed.\n");
        return 1;
    } else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    } else { /* parent process */
        wait(NULL);  /* parent will wait for the child to complete */
        printf("I: Child completed.\n");
    }
    return 0;
}
```

Process Representation in Linux — Example

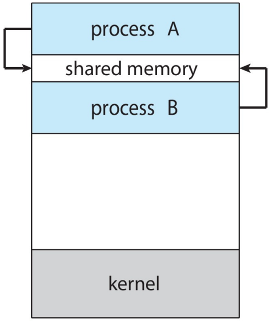
In the Linux kernel, the full status of a process (PCB) is captured in a `task_struct` structure (defined in `include/linux/sched.h`).

```
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 */
/* ... */
```

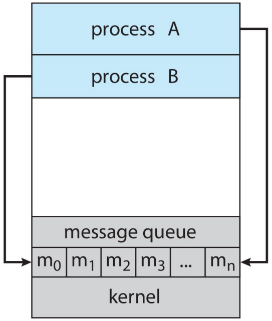


Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes: information sharing, computation speedup, modularity
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC: (a) **shared memory**, (b) **message passing**



(a)



(b)

Interprocess Communication — Examples

- IPC cannot happen without OS intervention
  - OS services must be requested either for each communication (often the case in message passing),
  - or initially to setup the communication mechanism (often the case for shared memory)
- Example of UNIX / POSIX IPC mechanisms:
  - Message passing: `pipe`, `mkfifo`, `mq_open`/`mq_send`/`mq_receive`/`mq_close`, `socket`
  - Shared memory: `mmap`, `shm_open`/`shm_unlink`

(More on some of these in later 3TC31 lectures and lab sessions.)

Reading List

You should study on books, not slides! Reading material for this lecture is:

- Silberschatz, Galvin, Gagne. Operating System Concepts, Tenth Edition:
  - Chapter 1: Introduction
  - Chapter 2: Operating-System Structures
  - Chapter 3: Processes

Credits:

- Some of the material in these slides is reused (with modifications) from the official slides of the book Operating System Concepts, Tenth Edition, as permitted by their copyright note.