CAB403 Study Guide | 2023 Semester 1

Timothy Chappell | Notes for CAB403 at the Queensland University of Technology

Unit Description

Disclaimer

Everything written here is based off the QUT course content. However, there are at times parts of text that are taken from the QUT slides and most of the examples are directly from the course slides (these are referenced when done). This content is designed only for those currently studying an IT degree at QUT, do not share these resources with anyone outside of this community.

If any member of the QUT staff or a representative of such finds any issue with these guides please contact me at jeynesbrook@gmail.com and I will take these down without an argument. The last thing I want to do is cause any issues or damages to the QUT name or QUT resources. I am simply just trying to help students out with presenting the content in an easy to digest manor.

Week 1

Operating Systems

What is an Operating System

An operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. It's acts as a resource allocator managing all resources and decides between conflicting requests for efficient and fair resource use. An OS also controls the execution of programs to prevent errors and improper use of the computer.

The operating system is responsible for:

- Executing programs
- Make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer System Structure

Computer systems can be divided into four main components

1. **Hardware**: These items provide basic computing resources, i.e. CPU, memory, I/O devices.
2. **Operating system**: Controls and coordinates the use of hardware among various applications and users.
3. **Application programs**: These items define the ways in which the system resources are used to solve the computing problems of the use, i.e. word processors, compilers, web browsers, database systems, video games.
4. **Users**: People, machines, or other computers.

Computer Startup

A bootstrap program is loaded at power-up or reboot. This program is typically stored in ROM or EPROM and is generally know as firmware. This bootstrap program is responsible for initialising all aspects of the system, loading the operating system kernal, and starting execution.

Computer System Organisation

- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type and has a local buffer.
- The CPU moves data from/to the main memory to/from local buffers.
- I/O is from the device to the local buffer of a particular controller.
- The device controller informs the CPU that it has finished its operation by causing an interrupt.

Common Functions of Interrupts

Operating systems are interrupt driven. Interrupts transfer control to the interrupt service routine. This generally happens through the interrupt vector which contains the addresses of all the service routines. The interrupt architecture must save the address of the interrupted instruction.

A trap or exception is a software-generated interrupt caused by either an error or a user request.

Interrupt Handling

The operating systems preserves the state of the CPU by storing registers and the program counter. It then determines which type of interrupt occured, polling or vectored interrupt system.

Once determined what caused the interrupt, separate segments of code determine what action should be taken for each type of interrupt.

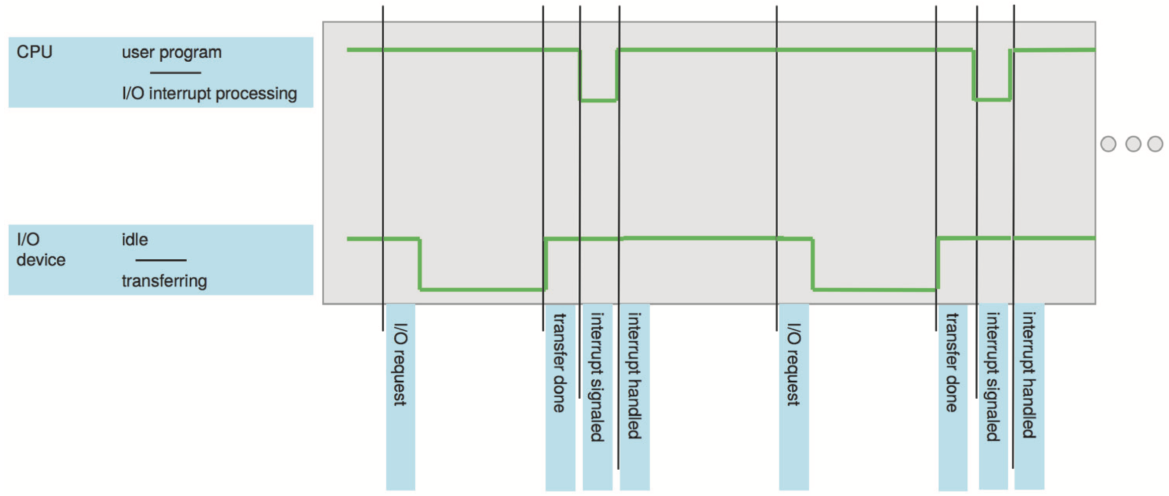


Figure: Interrupt timeline for a single program doing output.

I/O Structure

There are two ways I/O is usually structured:

1. After I/O starts, control returns to the user program only upon I/O completion.
    - Wait instructions idle the CPU until the next interrupt.
    - At most, one I/O request is outstanding at a time. This means no simultaneous I/O processing.
2. After I/O starts, control returns to the user program without waiting for I/O completion.
    - **System call**: Request to the OS to allow users to wait for I/O completion.
    - A **device-status table** containes entries for each I/O device indicating its type, address, and state.
    - The OS indexes into the I/O device table to determine the device status and to modify a table entry to include an interrupt.

Storage Definitions and Notation Review

The basic unit of computer storage is a bit. A bit contains one of two values, 0 and 1. A byte is 8 bits, and on most computers is the smallest convenient chunck of storage.

- A kilobyte, or KB, is $1,024$ bytes
- A megabyte, or MB, is $1,024^2$ bytes
- A gigabyte, or GB, is $1,024^3$ bytes
- A terabyte, or TB, is $1,024^4$ bytes
- A petabyte, or PB, is $1,024^5$ bytes

Direct Memory Access Structure

This method is used for high-speed I/O devices able to transmit information at close to memory speeds. Device controllers transfer blocks of data from buffer storage directly to main memory without CPU intervention. This means only one interrupt is generated per block rather than the one interrupt per byte.

Storage Structure

- **Main memory**: Only large storage media that the CPU can access directly.
  - Random access
  - Typically volatile
- **Secondary storage**: An extension of main memory that provides large non-volatile storage capacity.
- **Magnetic discs**: Rigid metal or glass platters covered with magnetic recording material. The disk surface is logically divided into tracks which are sub-diveded into sectors. The disk controller determines the logical interaction between the device and the computer.
- **Solid-state disks**: Achieves faster speeds than magnetic disks and non-volatile storage capacity through various technologies.

Storage Hierarchy

Storages systems are organised into a hierarchy:

- Speeds
- Cost
- Volatility.

There is a device driver for each device controller used to manage I/O. They provide uniform interfaces between controllers and the kernal.

Caching

Caching allows information to be copied into a faster storage system. The main memory can be viewed as a cache for the secondary storage.

Faster storage (cache) is checked first to determine if the information is there:

- If so, information is used directly from the cache
- If not, data is copied to the cache and used there

The cache is usually smaller and more expensive that the storage being cached. This means cache management is an important design problem.

Computer-System Architecture

Most systems use a single general-purpose processor. However, most systems have special-purpose processors as well.

Multi-processor systems, also known as parallel systems or tightly-coupled systems, usually come in two types; Asymmetric Multi-processing or Symmetric Multi-processor. Multi-processor systems have a few advantages over a single general-purpose processor:

- Increase throughput

- Economy of scale
- Increased reliability, i.e. graceful degradation or fault tolerance

Clustered Systems

Clustered systems are like Multi-processor systems, they have multiple systems working together.

- These systems typically share storage via a storage-area network (SAN).
- Provide a high-availability service which survices failures:
  - Asymmetric clustering have one machine in hot-standby mode.
  - Symmetric clustering have multiple nodes running applications, monitoring each other.
- Some clusters are for high-performance computing (HPC). Applications running on these clusters must be written to use parallelisation.
- Some have a distributed lock manager (DLM) to avoid conflicting operations.

Operating System Structure

Multi-programming organises jobs (code and data) so the CPU always has one to execute. This is needed for efficiency as a single user cannot keep a CPU and I/O devices busy at all times. Multi-programming works by keeping a subset of total jobs in the system, in memory. One job is selected and run via job scheduling. When it has to wait (for I/O for example), the OS will switch to another job.

Timesharing is a logical extension in which the CPU switches jobs so frequently that users can interact with each job while it is running.

- The response time should be less than one second.
- Each user has at least one program executing in memory (process).
- 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.

- If several jobs are ready to run at the same time, the CPU scheduler handles which to run.

Operating-System Operations

Dual-mode operations (user mode and kernal mode) allow the OS to protect itself and other system components. A mode bit provided by the hardware provides the ability to distinguish when a system is running user code or kernel code. Some instructions are designated as privileged and are only executable in kernal mode. System calls are used to change the mode to kernal, a return from call resets the mode back to user.

Most CPUs also support multi-mode operations, i.e. virtual machine manages (VMM) mode for guest VMs.

Input and Output

`printf()`

`printf()` is an output function included in `stdio.h`. It outputs a character stream to the standard output file, also known as `stdout`, which is normally connected to the screen.

It takes 1 or more arguments with the first being called the control string.

Format specifications can be used to interpolate values within the string. A format specification is a string that begins with `%` and ends with a conversion character. In the above example, the format specifications `%s` and `%d` were used. Characters in the control string that are not part of a format specification are placed directly in the output stream; characters in the control string that are format specifications are replaced with the value of the corresponding argument.

Example 1: Output with `printf()`

```c
printf("name: %s, age: %d\n", "John", 24); // "name: John, age: 24"
```

`scanf()`

`scanf()` is an input function included in `stdio.h`. It reads a series of characters from the standard input file, also known as `stdin`, which is normally connected to the keyboard.

It takes 1 or more arguments with the first being called the control string.

Example 2: Reading input with `scanf()`

```c
char a, b, c, s[100];
int n;
double x;

scanf("%c%c%c%d%s%lf", &a, &b, &c, &n, n, &x);
```

Relevant Links

- cppreference - printf
- cppreference - scanf

Pointers

A pointer is a variable used to store a memory address. They can be used to access memory and manipulate an address.

Example 1: Various ways of declaring a pointer

```c
// type *variable;

int *a;
int *b = 0;
int *c = NULL;
int *d = (int *) 1307;

int e = 3;
int *f = &e; // `f` is a pointer to the memory address of `e`
```

Example 2: Dereferencing pointers

```c
int a = 3;
int *b = &a;

printf("Values: %d == %d\nAddresses: %p == %p\n", *b, a, b, &a);
```

Relevant Links

- cppreference - pointer

Functions

A function construct in C is used to write code that solves a (small) problem. A procedural C program is made up of one or more functions, one of them being `main()`. A C program will always begin execution with `main()`.

Function parameters can be passed into a function in one of two ways; pass by value and pass by reference. When a parameter is passed in via value, the data for the parameters are copied. This means any changes to said variables within the function will not affect the original values passed in. Pass by reference on the other hand passes in the memory address of each variable into the function. This means that changes to the variables within the function will affect the original variables.

Example 1: Function control

```c
#include <stdio.h>

void prn_message(const int k);

int main(void) {
    int n;

    printf("There is a message for you.\n");
    printf("How many times do you want to see it?\n");

    scanf("%d", &n);

    prn_message(n);

    return 0;
}

void prn_message(const int k) {
    printf("Here is the message:\n");

    for (size_t i = 0; i < k; i++) {
        printf("Have a nice day!\n");
    }
}
```

Example 2: Pass by values

```c
#include <stdio.h>

void swapx(int a, int b);

int main(void) {
    int a = 10;
    int b = 20;

    // Pass by value
    swapx(a, b);

    printf("within caller - a: %d, b: %b\n", a, b); // "within
caller - a: 10, b: 20"

    return 0;
}

void swapx(int a, int b) {
    int temp;

    temp = a;
    a = b;
    b = temp;

    printf("within function - a: %d, b: %b\n", a, b); // "within
function - a: 20, b: 10"
}
```

Example 3: Pass by value

```c
#include <stdio.h>

void swapx(int *a, int *b);

int main(void) {
    int a = 10;
    int b = 20;

    // Pass by reference
    swapx(&a, &b);

    printf("within caller - a: %d, b: %b\n", a, b); // "within
caller - a: 20, b: 10"

    return 0;
}

void swapx(int *a, int *b) {
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;

    printf("within function - a: %d, b: %b\n", *a, *b); // "within
function - a: 20, b: 10"
}
```

Week 2

Operating System Structures

Operating System Services

Operating systems provide an environment for execution of programs and services to programs and users.

There are many operating system services that provide functions that are helpful to the user such as:

- **User interface**: Almost all operating systems have a user interface. This can be in the form of a graphical user interface (GUI) or a command-line (CLI).
- **Program execution**: The system must be able to load a program into memory and run that program, end execution, either normally or abnormally.
- **I/O operations**: A running program may require I/O, which may involve a file or an I/O device.
- **File-system manipulation**: The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, manage permissions, and more.
- **Communication**: Processors may exchange information, on the same computer or between computers over a network.
- **Error detection**: OS needs to be constantly aware of possible errors:
    - May occur in the CPU and memory hardware, in I/O devices, in user programs, and more.
    - For each type of error, the OS should take the appropriate action to ensure correct and consistent computing.
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.

Another set of OS functions exist for ensuring the efficient operation of the system itself via resource sharing.

- **Resource allocation**: When multiple users or multiple jobs are running concurrently, resources must be allocated to each of them.
- **Accounting**: To keep track of which users use how much and what kinds of resources.
- **Protection and security**: The owners of information stored in a multi-user or networked computer system may want to control use of that information. Concurrent processes should not interfere with each other.
    - Protection involves ensuring that all access to system resources is controlled.
    - Security of the system from outsiders requires user authentication. This also extends to defending external I/O devices from invalid access attempts.
    - If a system is to be protected and secure, pre-cautions must be instituted throughout it. A chain is only as strong as its weakest link.

System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally written in higher-level languages such as C and C++. These system calls however, are mostly accessed by programs via a high-level application programming interface (API) rather than direct system call use.

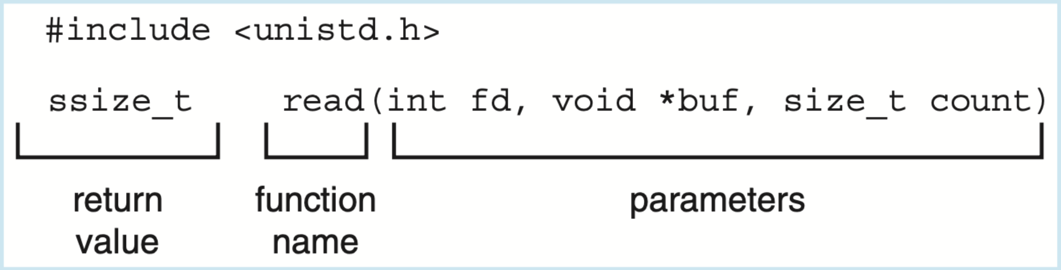
The three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems, and JAVA API for the Java virtual machine (JVM)

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>

ssize_t    read(int fd, void *buf, size_t count)
```

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.

Typically, a number is associated with each system call. The system-call interface maintains a table indexed according to these numbers. The system call interface invokes the intended system call in the OS kernel and returns a status of the systema call and any return values. The caller needs to know nothing about how the system call is implemented, it just needs to obey the API and understand what the OS will do as a result call.

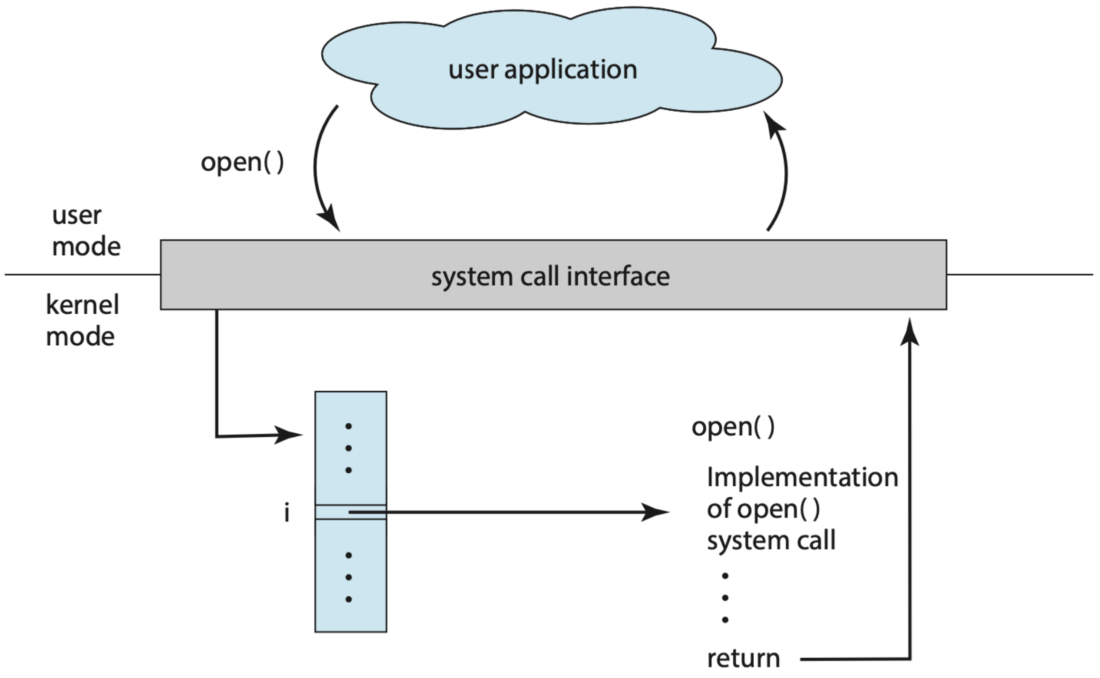


Figure: The handling of a user application invoking the `open()` system call.

There are many types of system calls:

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

Often, more information is required than simply the identity of the system call. There are three general methods used to pass parameters to the OS:

1. Pass parameters into registers. This won't always work however as there may be more parameters than registers.
2. Store parameters in a block, or table, in memory, and pass the address of the block as a parameter in a register.
3. Parameters are placed, or pushed, onto the stack by the program and popped off the stack by the operating system. This method does not limit the number length of the parameters being passed.

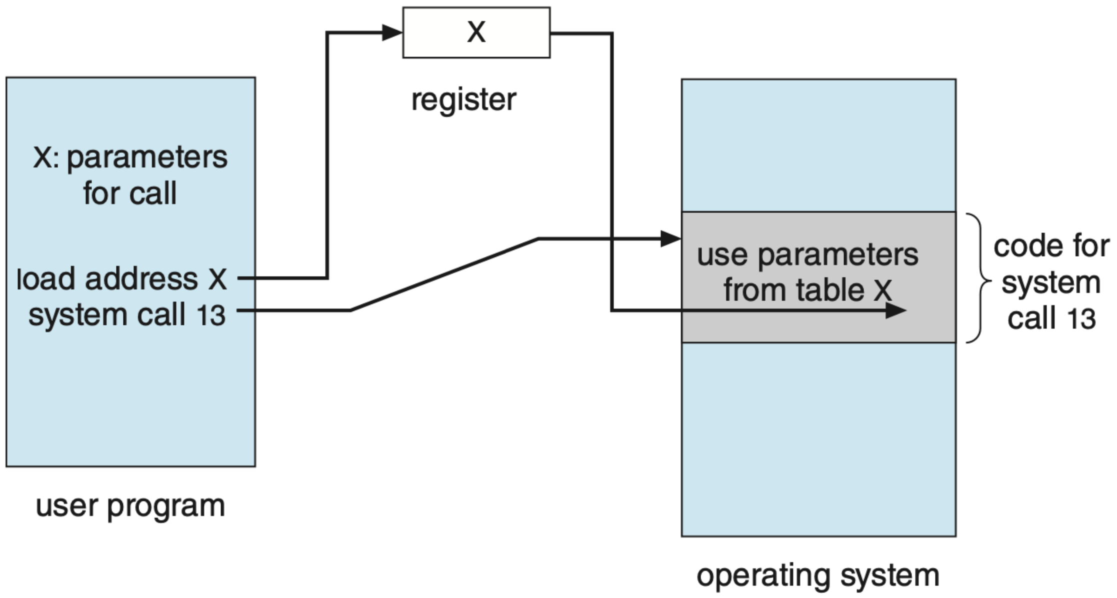


Figure: Passing of parameters as a table.

System Programs

System programs provide a convenient environment for program development and execution. They can be generally divided into:

- File manipulation
- Status information sometimes stored in a file modification
- Programming language support
- Program loading and execution
- Communications
- Background services
- Application programs

UNIX

UNIX is limited by hardware functionality. The original UNIX operating system had limited structing. The UNIX OS consists of two separable parts:

1. Systems programs
2. The kernel:
    - Consists of everything below the system-call interface and above the physical hardware.
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions.

Operating System Structure

There are a few ways to organise an operating system.

Layered

The operating system is divided into a number of layers, each built on top of the lower layers. The bottom layer (layer 0), is the hardware; the highest is the user interface.

Due to the modularity, layers are selected such that each uses functions and services of only lower-level layers.

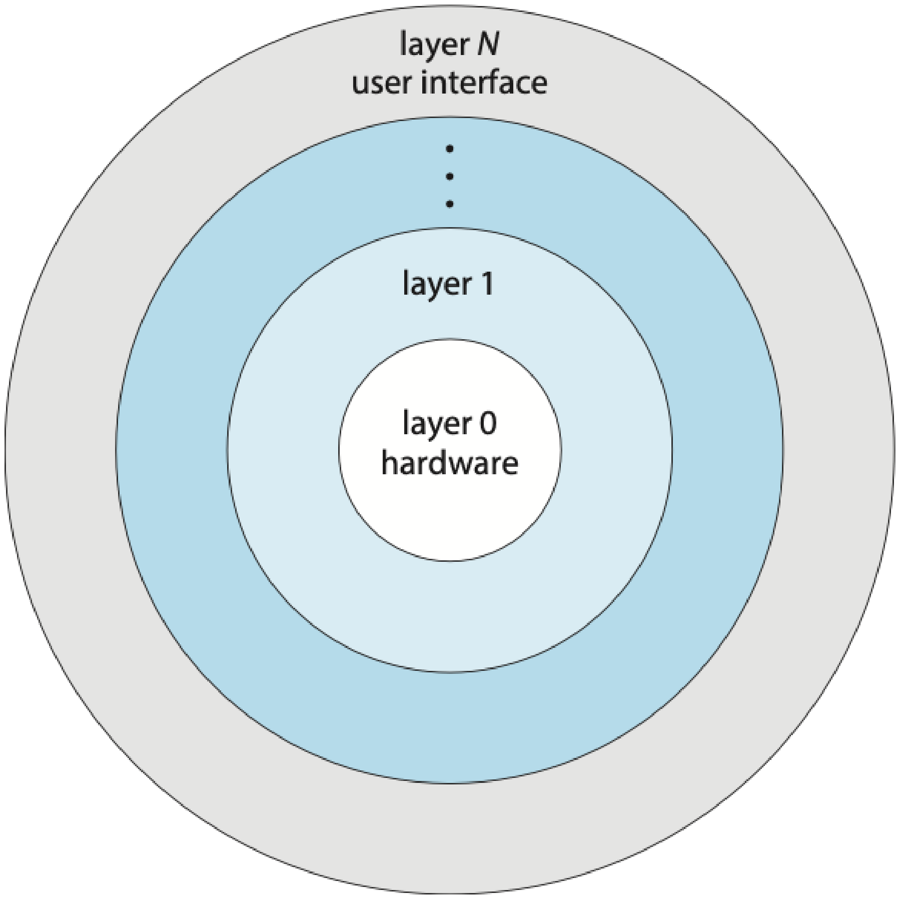


Figure: A layered operating system.

Microkernel System

In this organisation method, as much as possible is moved from the kernel into user space. An example OS that uses a microkernel is Mach, which parts of the MacOSX kernel (Darwin) is based upon. Communication takes place between user modules via message passing.

Advantages	Disadvantages
Easier to extend a microkernel	Performance overhead of user space to kernel space communication
Easier to port the operating system to new architectures	

Advantages	Disadvantages
More reliable (less code is running in kernel mode)	
More secure	

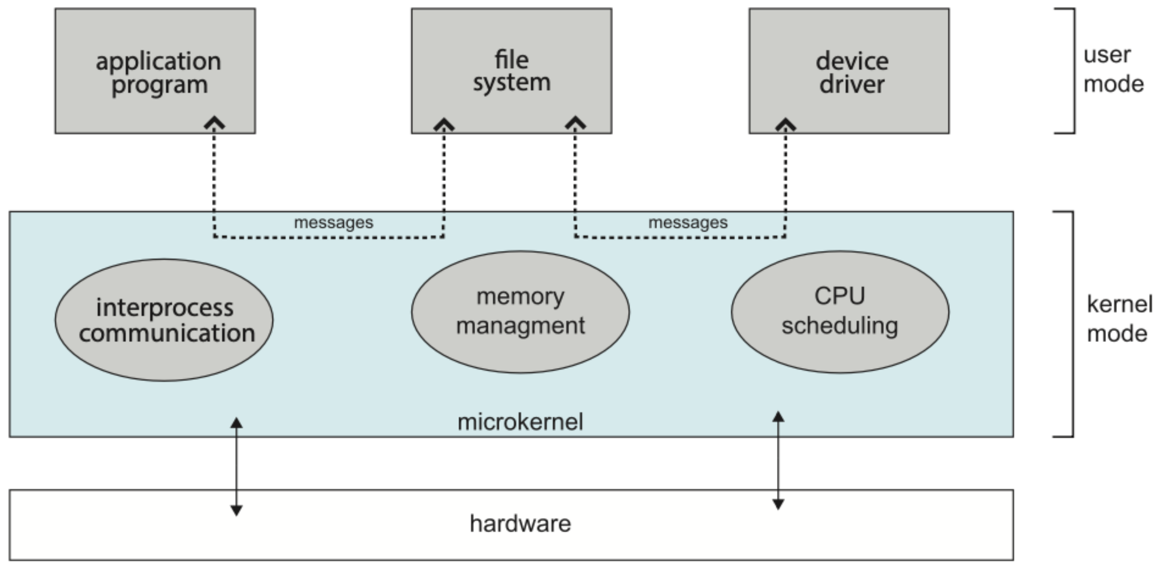


Figure: Architecture of a typical microkernel.

Hybrid System

Most modern operating systems don't use a single model but a use concepts from a variety. Hybrid systems combine multiple approaches to address performance, security, and usability needs.

For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance. However, it's also modular, so that new functionality can be dynamically added to the kernel.

Modules

Most modern operating systems implement loadable kernel modules (LKMs). Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time

Each core component is separate, can talk to others via known interfaces, and is loadable as needed within the kernel.

Arrays

An array is a contiguous sequence of data items of the same type. An array name is an address, or constant pointer value, to the first element in said array.

Aggregate operations on an array are not valid in C, this means that you cannot assign an array to another array. To copy an array you must either copy it component-wise (typically via a loop) or via the `memcpy()` function in `string.h`.

Example 1: Arrays in practice

```
#include <stdio.h>

const int N = 5;

int main(void) {
    // Allocate space for a[0] to a[4]
    int a[N];
    int i;
    int sum = 0;

    // Fill the array
    for (i = 0; i < N; i++) {
        a[i] = 7 + i * i;
    }

    // Print the array
    for (i = 0; i < N; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }

    // Sum the elements
    for (i = 0; i < N; i++) {
        sum += a[i];
    }

    printf("\nsum = %d\n", sum);

    return 0;
}
```

Example 2: Arrays and Pointers

```c
#include <stdio.h>

const int N = 5;

int main(void) {
    int a[N];
    int sum;
    int *p;

    // The following two calls are the same
    p = a;
    p = &a[0];

    // The following two calls are the same
    p = a + 1;
    p = &a[1];


    // Version 1
    sum = 0;

    for (int i = 0; i < N; i++) {
        sum += a[i];
    }

    // Version 2
    sum = 0;

    for (int i = 0; i < N; i++) {
        sum += *(a + i);
    }
}
```

Example 3: Bubble Sort

```c
#include <stdio.h>

void swap(int *arr, int i, int j);
void bubble_sort(int *arr, int n);

void main(void) {
    int arr[] = { 5, 1, 4, 2, 8 };
    int N = sizeof(arr) / sizeof(int);

    bubble_sort(arr, N);

    for (int i = 0; i < N; i++) {
        printf("%d: %d\n", i, arr[i]);
    }

    return 0;
}

void swap(int *arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void bubble_sort(int *arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr, j, j + 1);
            }
        }
    }
}
```

Example 4: Copying an Array

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    // Copying an array component-wise
    int array_one[5] = { 1, 2, 3, 4, 5 };
    int array_two[5];

    for (int idx = 0; idx < 5; idx++) {
        array_two[idx] = array_one[idx];
    }

    // Copying an array via memcpy
    memcpy(array_two, array_one, sizeof(int) * 5);
}
```

Relevant Links

- cppreference - array
- cppreference - memcpy

Strings

A string is a one-dimensional array of type `char`. All strings must end with a null character `\0` which is a byte used to represent the end of a string.

A character in a string can be accessed either by an element in an array of by making use of a pointer.

Example 1: Strings in practice

```c
char *first = "john";
char last[6];

last[0] = 's';
last[1] = 'm';
last[2] = 'i';
last[3] = 't';
last[4] = 'h';
last[5] = '\0';

printf("Name: %s, len: %lu", first, strlen(first));
```

Relevant Links

- Wikipedia - Null-terminated string

Structures

Structures are named collections of data which are able to be of varying types.

Example 1: Structures in practice

```c
struct student {
    char *last_name;
    int student_id;
    char grade;
};

// By using `typedef` we can avoid prefixing the type with `struct`
typedef struct unit {
    char *code;
    char *name;
} unit;

void update_student(struct student *student);
void update_grade(unit *unit);

int main(void) {
    struct student s1 = {
        .last_name = "smith",
        .student_id = 119493029,
        .grade = 'B',
    };

    s1.grade = 'A';

    update_student(&s1);


    unit new_unit;

    new_unit.name = "Microprocessors and Digital Systems";

    update_unit(&new_unit);
}

void update_student(struct student *student) {
    // `->` shorthand for dereference of struct
    student->last_name = "doe";
    student->grade = 'C';
}

void update_unit(unit *unit) {
    // `->` shorthand for dereference of struct
    unit->code = "CAB403";
    unit->name = "Systems Programming";
}
```

Relevant Links

- cppreference - Struct declaration
- cppreference - typedef specifier

Dynamic Memory Management

Memory in a C program can be divided into four categories:

1. Code memory
2. Static data memory
3. Runtime stack memory
4. Heap memory

Code Memory

Code memory is used to store machine instructions. As a program runs, machine instructions are read from memory and executed.

Static Data Memory

Static data memory is used to store static data. There are two categories of static data: global and static variables.

Global variables are variables defined outside the scope of any function as can be seen in example 1. Static variables on the other hand are defined with the `static` modifier as seen in example 2.

Both global and static variables have one value attached to them; they are assigned memory once; and they are initialised before `main` begins execution and will continue to exist until the end of execution.

Example 1: Global variables.

```c
int counter = 0;

int increment(void) {
    counter++;

    return counter;
}
```

Example 2: Static variables.

```c
int increment(void) {
    // will be initialised once
    static int counter = 0;

    // increments every time the function is called
    counter++;

    return counter;
}
```

Runtime Stack Memory

Runtime stack memory is used by function calls and is FILO (First in, Last out). When a function is invoked, a block of memory is allocated by the runtime stack to store the information about the function call. This block of memory is termed as an *Activation Record*.

The information about the function call includes:

- Return address.
- Internal registers and other machine-specific information.
- Parameters.
- Local variables.

Heap Memory

Heap memory is memory that is allocated during the runtime of the program. On many systems, the heap is allocated in an opposite direction to the stack and grows towards the stack as more is allocated. On simple systems without memory protection, this can cause the heap and stack to collide if too much memory is allocated to either one.

To deal with this, C provides two functions in the standard library to handle dynamic memory allocation; `calloc()` (contiguous allocation) and `malloc()` (memory allocation).

`void *calloc(size_t n, size_t s)` returns a pointer to enough space in memory to store `n` objects, each of `s` bytes. The storage set aside is automatically initialised to zero.

`void *malloc(size_t s)` returns a pointer to a space of size `s` and leaves the memory uninitialised.

Example 3: `malloc()` and `calloc()`

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

int main() {
    int num_of_elements;
    int *ptr;
    int sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &num_of_elements);

    ptr = malloc(num_of_elements * sizeof(int));
    // or
    // ptr = calloc(num_of_elements, sizeof(int));

    if (ptr == NULL) {
        printf("[Error] - Memory was unable to be allocated.");

        exit(0);
    }

    printf("Enter elements: ");

    for (int i = 0; i < n; i++) {
        scanf("%d", ptr + i);

        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);

    free(ptr);

    return 0;
}
```

Relevant Links

- cppreference - malloc
- cppreference - calloc
- cppreference - realloc
- cppreference - free