

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 and the recommended text books. 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.

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 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 user, 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 known as firmware. This bootstrap program is responsible for initialising all aspects of the system, loading the operating system kernel, 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 system preserves the state of the CPU by storing registers and the program counter. It then determines which type of interrupt occurred,

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.

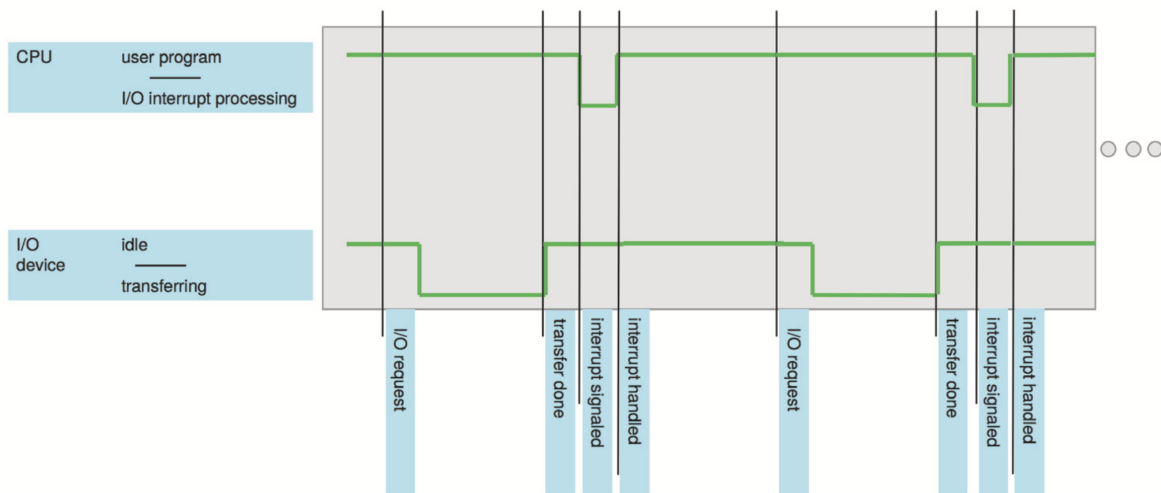


Figure: Interrupt timeline for a single program doing output.

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** contains 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 chunk 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-divided 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

Storage 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 kernel.

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 than 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 survives 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 kernel 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 kernel mode. System calls are used to change the mode to kernel, 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()`

```
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()`

```
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

```
// 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

```
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

```
#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

```
#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

```

#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"
}

```

Example 4: Function pointers

```
#include <stdio.h>

void function_a(int num) {
    printf("Function A: %d\n", num);
}

void function_b(int num) {
    printf("Function B: %d\n", num);
}

void caller(void (*function) (int)) {
    function(1);
    function(2);
    function(3);
}

int main(void) {
    caller(function_a);
    caller(function_b);

    return 0;
}
```


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>		
<code>ssize_t</code>	<code>read</code>	<code>(int fd, void *buf, size_t count)</code>
return value	function name	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`.

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 system 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.

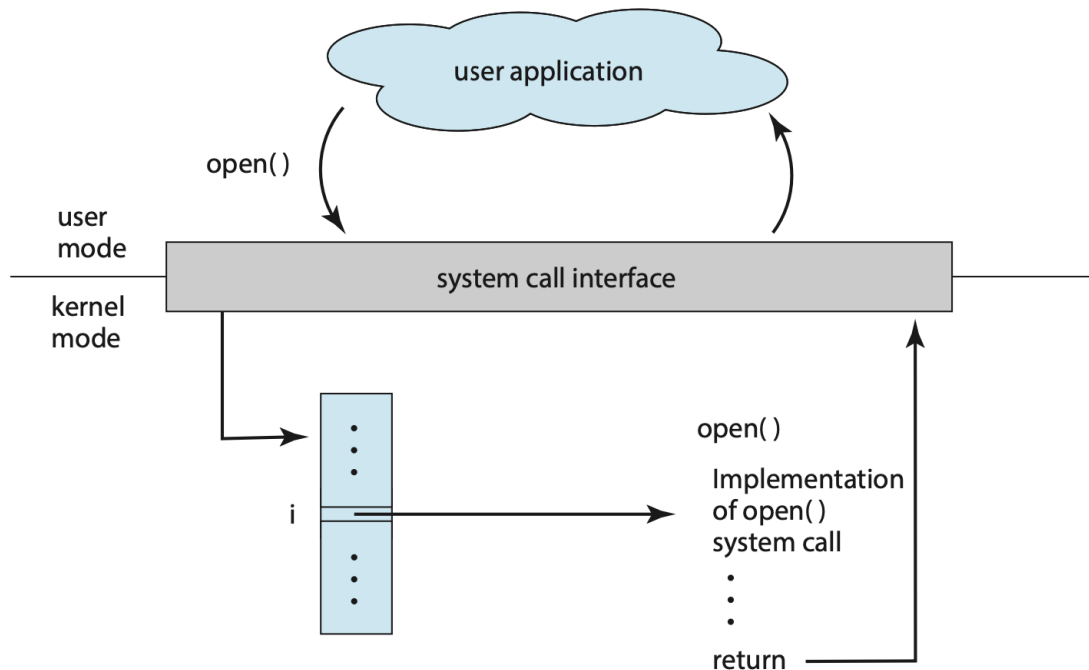


Figure: The handling of a user application invoking the `open()` system call.

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.

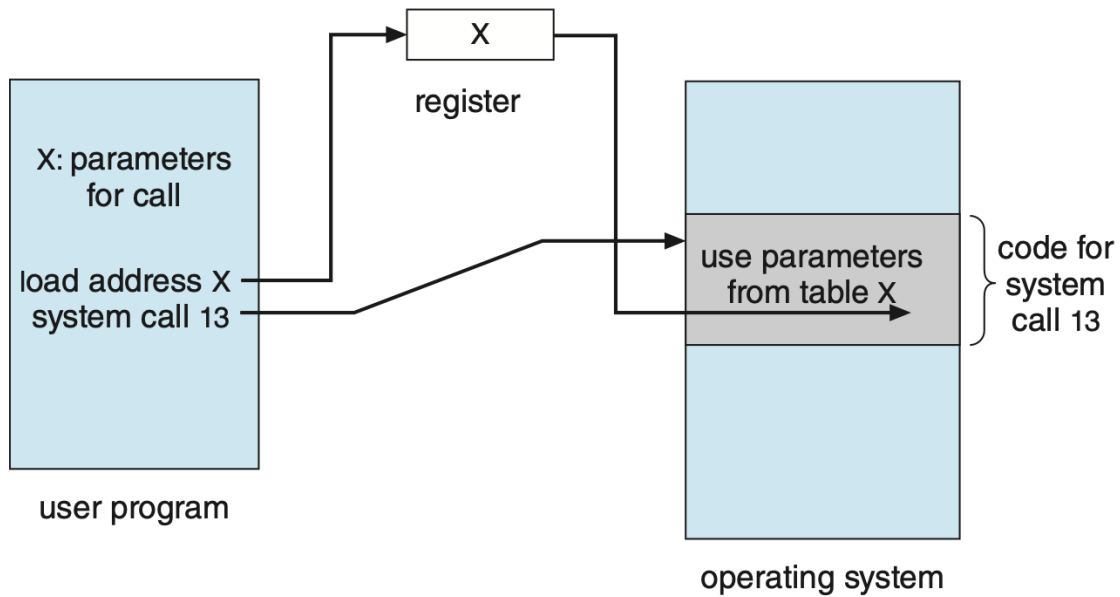


Figure: Passing of parameters as a table.

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.

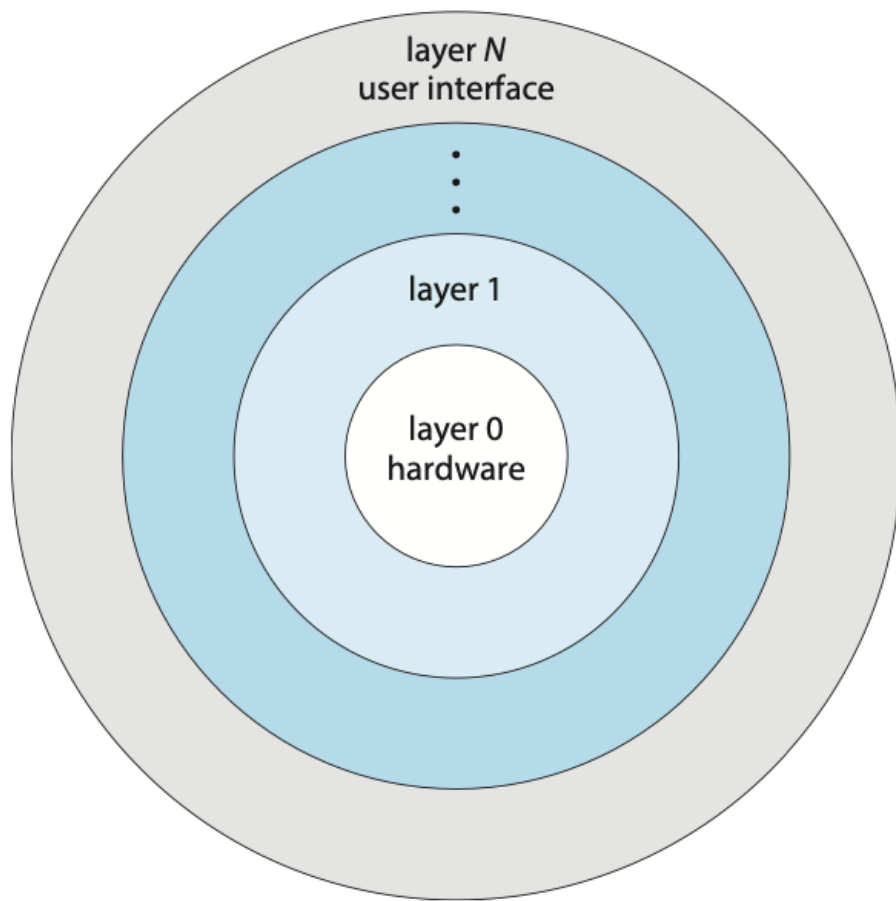


Figure: A layered operating system.

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	

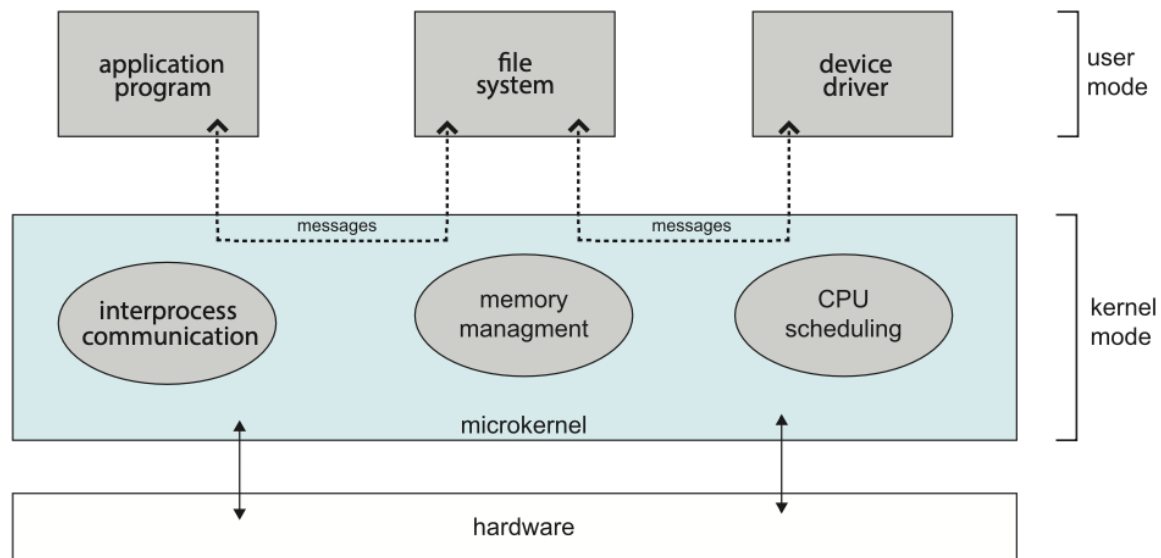


Figure: Architecture of a typical microkernel.

Hybrid System

Most modern operating systems don't use a single model but 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

```
#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

```

#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

```
#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 or by making use of a pointer.

Example 1: Strings in practice

```
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


```

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.

```
int counter = 0;

int increment(void) {
    counter++;

    return counter;
}
```

Example 2: Static variables.

```
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()`

```

#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](#)

Week 3

Processes

An operating system executes a variety of programmes either via:

- Batch systems (jobs)
- or Time-shared systems (user programs or tasks)

A process, sometimes referred to as a job, is simply a program in execution. The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers.

A process is made up of multiple parts:

- **Text section:** The executable code
- **Data section:** Global variables
- **Heap section:** Memory that is dynamically allocated during program run time
- **Stack section:** Temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

It's important to note that a program itself is not a process but rather a passive entity. In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

As a process executes, it changes state. A process may be in one of the following states:

- **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.

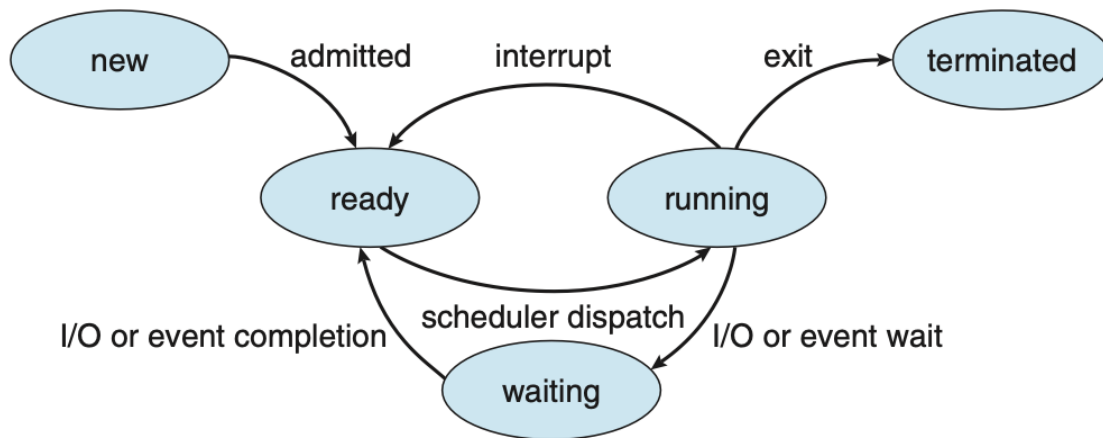


Figure: Diagram of process state.

Process Control Block (PCB)

Each process is represented in the OS by a process control block, also known as a task control block. It contains information associated with a specific process such as:

- **Process state:** The state of the process.
- **Program counter:** The address of the next instruction to be executed for this process.
- **CPU registers:** The contents of all process-centric registers. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- **CPU scheduling information:** Information about process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, etc..

- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, etc..

Threads

In a single-threaded model, only a single thread of instructions can be executed. This means only a single task can be completed at any given time. For example, in a word document, the user cannot simultaneously type in characters and run the spell checker.

In most modern operating systems however, the use of multiple threads allows more than one task to be performed at any given moment. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker.

In a multi-threaded system, the PCB is expanded to include information for each thread.

Process Scheduling

The objective of multi-programming is to have some process running at all times so as to maximize CPU utilization. A process scheduler is used to determine which process should be executed. The number of processes currently in memory is known as the degree of multiprogramming

When a process enters the system, it's put into a **ready queue** where it then waits to be executed. When a process is allocated a CPU core for execution it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event. Any process waiting for an event to occur gets placed into a **wait queue**.

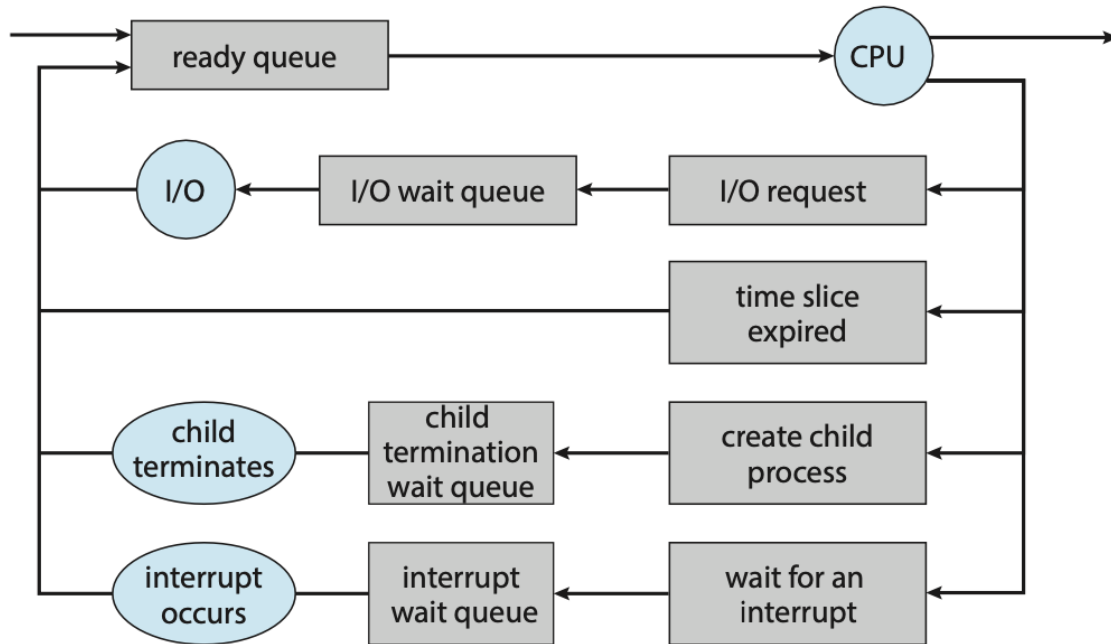


Figure: Queueing-diagram representation of process scheduling.

Most processes can be described as either:

- **I/O bound:** A I/O bound process that spends more of its time doing I/O operations.
- **CPU bound:** Spends more of its time doing more calculations with infrequent I/O requests.

Context Switch

Interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. These operations happen frequently so it's important to ensure that when returning to the process, no information was lost.

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the

context of the old process in its PCB and loads the saved context of the new process scheduled to run.

The time between a context switch is considered as overhead as no useful work is done while switching. The more complex the OS and PCB, the longer it takes to context switch.

Process Creation

During execution, a process may need to create more processes. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Processes are identified by their process identifier (PID).

When a process is created, it will generally require some amount of resources to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. The return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Once forked, it's typical for `exec()` to be called on one of the two processes. The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts its execution.

For example, this code forks a new process and, using `execvp()`, a version of the `exec()` system call, overlays the process address space with the UNIX command `/bin/ls` (used to get a directory listing).

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

int main() {
    pid_t pid;
    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork failed\n");

        return 1;
    } else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    } else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);

        printf("Child complete\n");
    }

    return 0;
}
```

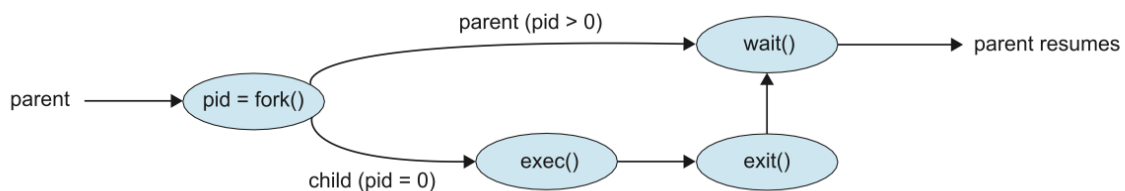


Figure: Process creation using the fork() system call.

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the `wait()` system call).

A parent may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;  
int status;  
  
pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status.

If a child process is terminated but the parent has not called `wait()`, the process is known as a zombie process. If a parent is terminated before calling `wait()`, the process is known as an orphan.

Interprocess Communication

Processes within a system may be independent or cooperating. A process is cooperating if it can affect or be affected by the other processes executing in the system.

There are a variety of reasons for providing an environment that allows process cooperation:

- Information sharing
- Computational speedup
- Modularity
- Convenience

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data. There are two fundamental models of interprocess communication: shared memory and message passing.

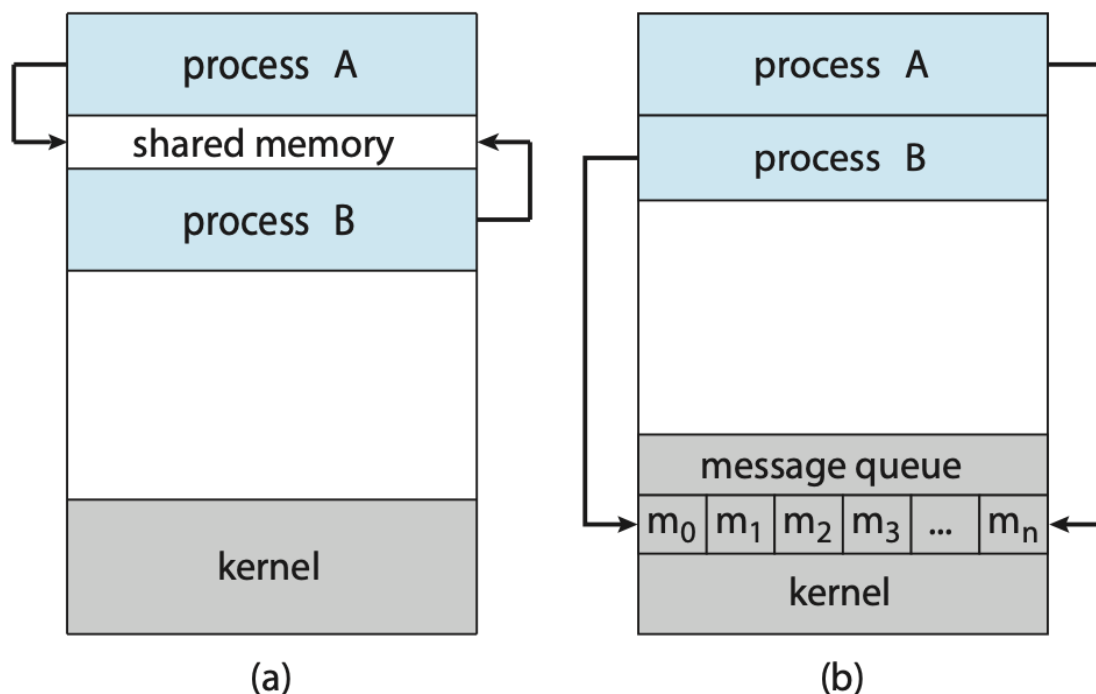


Figure: Communications models. (a) Shared memory. (b) Message passing.

In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information

by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Producer-Consumer Problem

The Producer-Consumer problem is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message Passing

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

A message-passing facility provides at least two operations:

1. `send(message)`
2. `receive(message)`

Before two processes can communicate, they first need to establish a communication link.

This could be via physical hardware:

- Shared memory.
- Hardware bus.

or logical:

- Direct or indirect communication.
- Synchronous or asynchronous communication.
- Automatic or explicit buffering.

Direct Communication

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.

- `send(P, message)` - send a message to process P.
- `receive(Q, message)` - receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically.
- The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification.

- `send(A, message)` — Send a message to mailbox A.
- `receive(A, message)` — Receive a message from mailbox A.

The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mail box.
- Send and receive messages through the mailbox.
- Delete a mail box.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Now suppose that processes P_1 , P_2 , and P_3 all share mailbox A. Process P_1 sends a message to A, while both P_2 and P_3 execute a `receive()` from A. Which process will receive the message sent by P_3 ? The answer depends on which of the following methods we choose:

- Allow a link to be associated with at most two processes
- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronisation

Communication between processes takes place through calls to `send()` and `receive()` primitives. Message passing may be either blocking or nonblocking also known as synchronous and asynchronous.

- **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send:** The sending process sends the message and resumes operation.
- **Blocking receive:** The receiver blocks until a message is available.

- **Nonblocking receive:** The receiver retrieves either a valid message or a null.

Different combinations of `send()` and `receive()` are possible. When both `send()` and `receive()` are blocking, we have a rendezvous between the sender and the receiver.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. These queues can be implemented in three ways:

1. **Zero capacity:** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
2. **Bounded capacity:** The queue has finite length n ; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
3. **Unbounded capacity:** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Week 4

Threads

A thread is a basic unit of CPU utilisation. A thread consists of:

- A thread ID
- A program counter PC
- A register set
- A stack

A thread shares its code section, data section, and other operating-system resources with other threads within the same process. A traditional process usually consists of a single thread of control, this is called a single-threaded process. A process with multiple threads of control can therefore perform more than one task at any given moment, this is called a multi-threaded process.

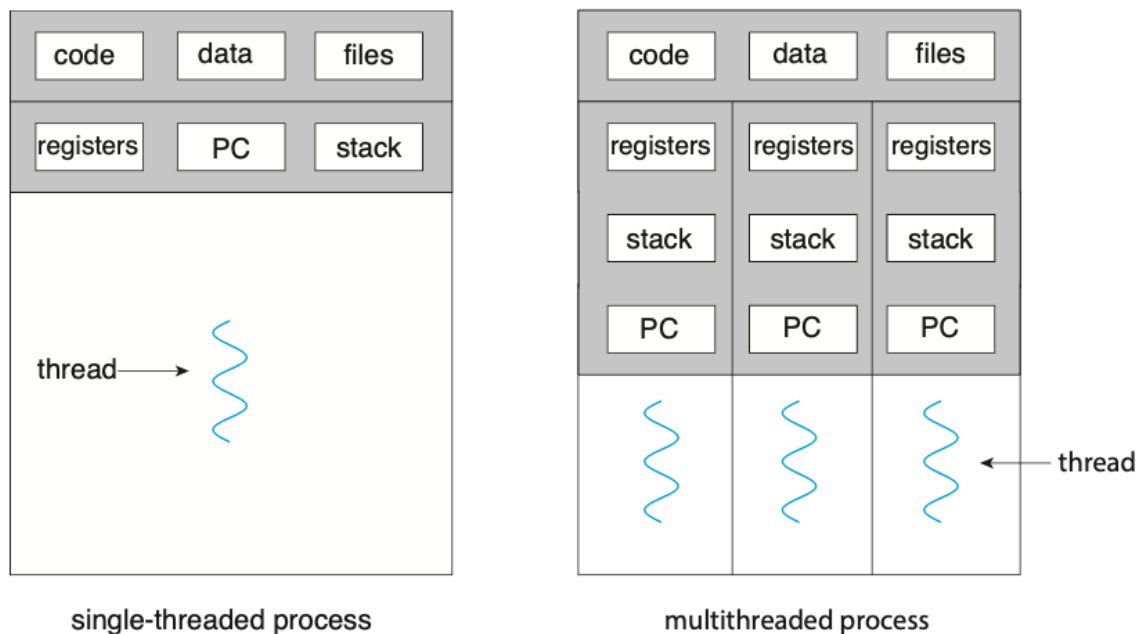


Figure: Single-threaded and multithreaded processes.

Most programs that run on modern computers and mobile devices are multithreaded. For example, A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user,

and a third thread for performing spelling and grammar checking in the background.

In certain situations, a single application may be required to perform several tasks at any one time. For example, a web server needs to accept many client requests concurrently. A solution to this is to have the server run a single process that accepts requests. When a request is received, a new process is created to service the request. Before threads became popular, this was the most common way to handle such situation.

The problem with this however is that processes are expensive to create. If the new process will perform the same tasks as the existing process, why incur the overhead of creating another. If a web server is multithreaded, the server will create a separate thread that listens for client requests. When a new request comes in, the server will create a new thread to service the request and resume listening for more requests.

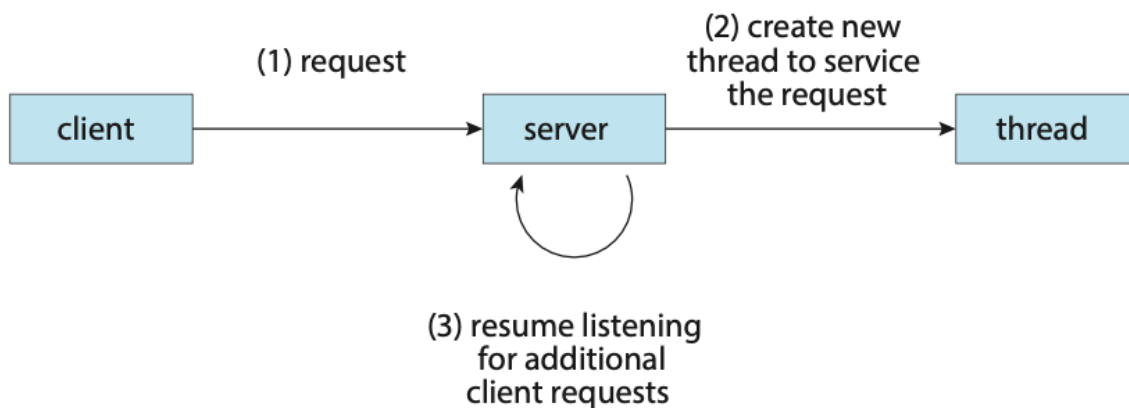


Figure: Multithreaded server architecture.

Most operating system kernels are also typically multithreaded. During system boot time on Linux systems, several kernel threads are created to handle tasks such as managing devices, memory management, and interrupt handling.

There are many benefits to using a multithreaded programming approach:

1. **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.

2. **Resource sharing:** Processes can share resources only through techniques such as shared memory and message passing. However, threads share the memory and the resources of the process to which they belong by default.
3. **Economy:** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
4. **Scalability:** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Multicore Programming

Due to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A trend in system design was to place multiple computing cores on a single processing chip where each core would then appear as a separate CPU to the operating system, such systems are referred to as multicore systems.

Imagine an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time due to the processing core only being capable of executing a single thread at a time.

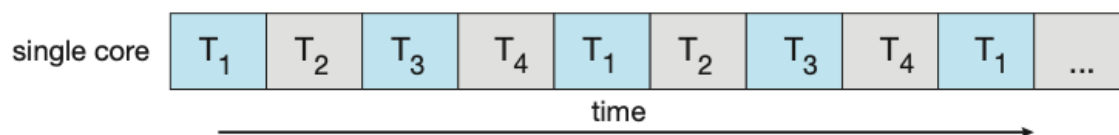


Figure: Concurrent execution on a single-core system.

On a system with multiple cores, concurrency means that some threads can run in parallel due to the system being capable of assigning a separate thread to each core.

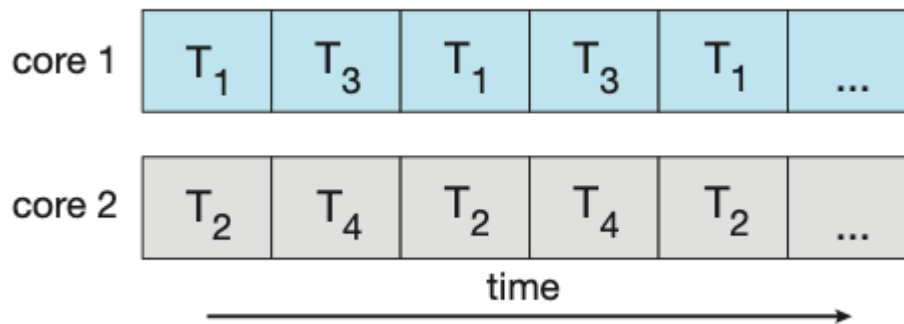


Figure: Parallel execution on a multicore system.

Types of Parallelism

There are two types of parallelism:

1. **Data parallelism:** Focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
2. **Task parallelism:** Involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

It's important to note that these two methods are not mutually exclusive and an application may use a hybrid method of both strategies.

Multithreading Models

Support for threads may be provided either at the user level (user threads) or by the kernel (kernel threads). User threads are supported above the kernel and are managed without kernel support. Kernel threads on the other hand are supported and managed directly by the operating system.

There are three common relationships between user threads and kernel threads.

1. **Many-to-One Model:** The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient. However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

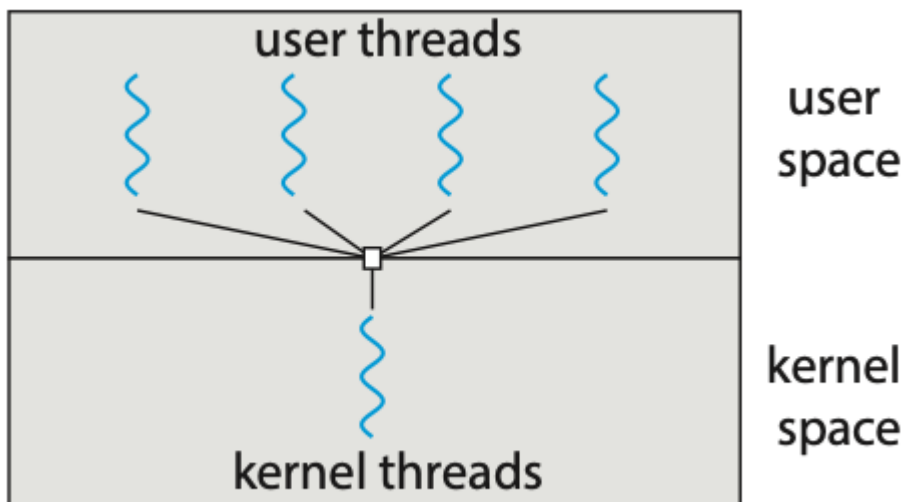


Figure: Many-to-one model.

2. **One-to-One Model:** The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system.

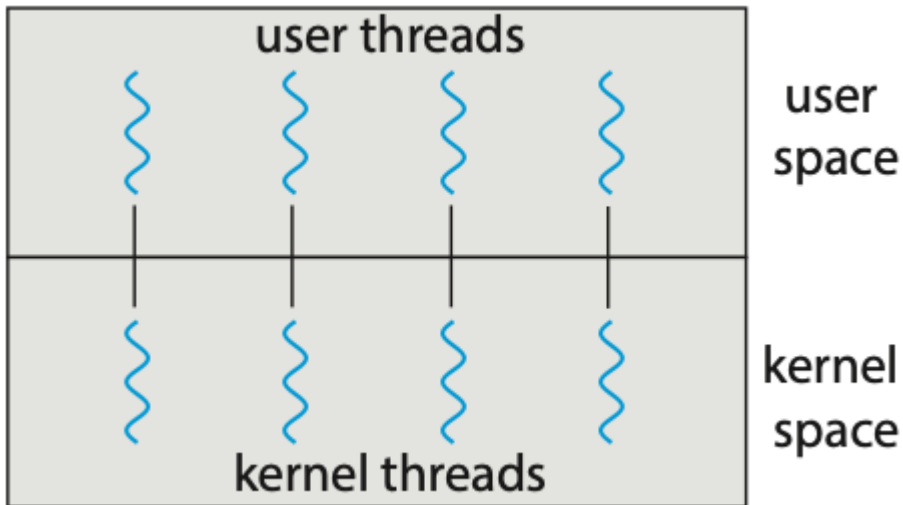


Figure: One-to-one model.

3. **Many-to-Many Model:** The many-to-many model (Figure 4.9) multiplexes many user-level threads to a smaller or equal number of kernel threads. Although the many-to-many model appears to be the most flexible of the models discussed, in practice it is difficult to implement.

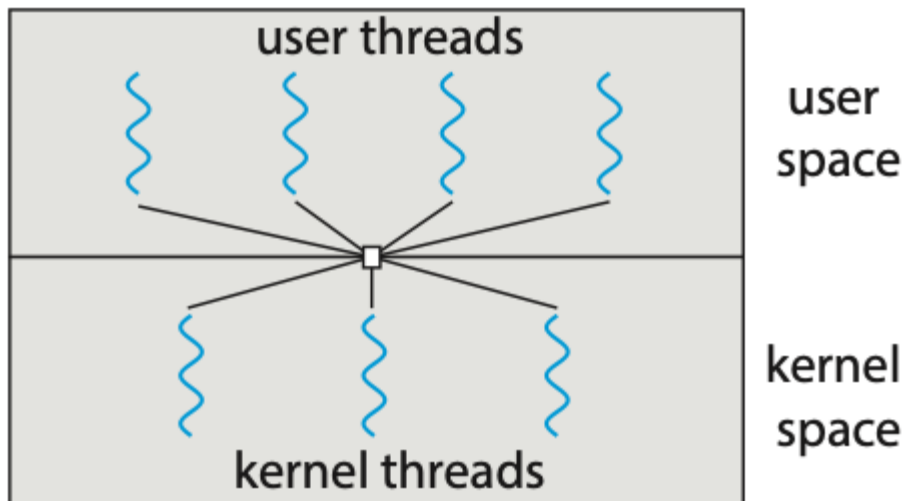


Figure: Many-to-many model.

Creating Threads

There are two general strategies for creating multiple threads:

1. **Asynchronous threading:** Once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another.
2. **Synchronous threading:** The parent thread creates one or more children and then must wait for all of its children to terminate before it resumes. Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed. Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution.

Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronisation. It's important to know that Pthreads is simply a specification for thread behaviour and not an implementation, that is left up to the operating-system designers.

Below is an example application using Pthreads to calculate the summation of a non-negative integer in a separate thread.

```

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

int sum; // The data shared among the threads.
void *runner(void *param); // The function called by each thread.

int main(int argc, char *argv[]) {
    pthread_t tid; // The thread identifier.
    pthread_attr_t attr; // Set of thread attributes.

    // Set the default attributes of the thread.
    pthread_attr_init(&attr);

    // Create the thread.
    pthread_create(&tid, &attr, runner, argv[1]);

    // Wait for the thread to finish executing.
    pthread_join(tid, NULL);

    printf("Sum: %d\n", sum);
}

void *runner(void *param) {
    int upper = atoi(param);
    int sum = 0;

    for (int i = 1; i <= upper; i++) {
        sum += 1;
    }

    pthread_exit(0);
}

```

This example program creates only a single thread. With the growing dominance of multicore systems, writing programs containing several threads has become increasingly common. A simple method for waiting on several threads using the `pthread_join()` function is to enclose the operation within a simple for loop.

```
#define NUM_THREADS 10

pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(workers[i], NULL);
}
```

Thread Pools

The idea behind a thread pool is to create a number of threads at start-up and place them into a pool where they sit and wait for work. In the context of a web server, when a request is received, rather than creating a new thread, it instead submits the request to the thread pool and resumes waiting for additional requests. Once the thread completes its service, it returns to the pool and awaits more work.

A thread pool has many benefits such as:

- Servicing a request within an existing thread is often faster than waiting to create a new thread.
- A thread pool limits the number of threads that exist at any one point. This ensures that the system does not get overwhelmed when creating more threads than it can handle.
- Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.

The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, amount of physical memory, and the expected number of concurrent client requests. More sophisticated thread pool architectures are able to dynamically adjust the number of threads in the pool based off usage patterns.

Fork Join

The fork-join method is one in which when the main parent thread creates one or more child threads and then waits for the children to terminate and join with it.

This synchronous model is often characterised as explicit thread creation, but it is also an excellent candidate for implicit threading. In the latter situation, threads are not constructed directly during the fork stage; rather, parallel tasks are designated. A library manages the number of threads that are created and is also responsible for assigning tasks to threads.

Threading Issues

- The `fork()` and `exec()` system calls
- Signal handling
- Thread cancellation
- Thread-local storage
- Scheduler activations