Operating Systems and Systems Programming I (*CPS1012*) Short Notes

Aiden Cachia

Contents

1 Introduction

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.

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.

Computer System Structure

- Hardware.
- Operating System.
- Application programs.
- Users.

1.1 Operating System Services

- User Interface (CLI, GUI, Touch-screen, batch)
- Program Execution
- I/O Operations
- File-system manipulation
- Communications (On Device, Network)
- Error Detection
- Resource Allocation
- Protection and Security

Command Line Interface

Simple Interface where command is fetched from user, it is exectued and the output is displayed.

Graphical User Interface

More user friendly design, making use of;

- Mouse, Keyboard and monitor
- Icons representing files, programs, actions etc ...
- Menus

1.2 System Calls

Ways to interact with the Operating System. They are typically written in a high level language (C or C++) and are used to request a service from the Operating System.

Implementation of System Calls

- A table of all is usualy stored in the System-call interface.
- The System-call interface invokes the system-call in OS kernel and returns the status of the system-call and any return values.

System Call Parameters

- Registers
- Block/table
- Stack

Types of System Calls

Process Control

- Create process, terminate process
- End, abort
- Load, execute
- Get process attributes, set process attributes
- Wait for time
- Wait for event, signal event
- Allocate and free memory
- Dump memory
- Debugging
- Locking

File Management

- Create file, delete file
- Open, close
- Read, write, reposition
- Get file attributes, set file attributes

Device Management

- Request device, release device

- Read, write, reposition
- Get device attributes, set device attributes
- Logically attach or detach devices

Information Maintenance

- Get or set
  - time or date
  - system data
  - process, file or device attributes

Communications

- Create, delete communication connection
- Send, receive messages
- Shared memory
- Transfer status information
- Attach or detach remote devices

1.3 System Services

Helps in the development of programs. Some are just system-call interfaces, others more complex.

Types of System Services

- File management

  Create, delete, copy, rename, print, dump, list and generally manipulate files and directories.
- Status information
  - Basic information (date, time, amount of available memory, disk space, number of users).
  - Detailed logging and Debugging information.
  - Registry to store and retrieve configuration information.
  - File modification
  - Program loading and execution
  - Communications
- Programming language support (Compilers, assemblers, debuggers and interpreters sometimes provided)
- Program loading and execution
- Communications
- Background services

  - Launch at boot
  - Provides facilities like disk checking, process scheduling, error logging, printing.
  - Run in user context not kernel context
  - Known as *services*, *subsystems* or *daemons*
- Application programs
  - Unrelated to system
  - Run by users
  - Not part of OS
  - Launched by CLI, Mouse click, finger poke

Linkers and Loaders

A Linker is used to link multiple object files (compiled from source code) into a single executable file.

A Loader is used to load the executable file into memory and start the execution of the program.

Relocation is the process of adjusting the addresses in the object file to match the addresses in memory.

Dynamic linked libraries (DLLs) are libraries that are linked at runtime, rather than at compile time, so they could be used by multiple programs that would require the same version of the library.

Why Applications are Operating System Specific

System calls and file formats are different from one operating system to another, among other things. Because of this applications are not portable between operating systems without modification.

Applications can be made portable:

- By using an interpreted language such as Python or Ruby;
- By using a language which include a VM such as Java;
- By using compiling a program which a User standard language (such as C), then recompiling it in target OSes.

An Application Binary Interface (ABI) is a specification defining requirements for an application on a specific hardware platform.

1.4 Operating System Design

There is no correct way to design an Operating System, but there are some common design principles.

An OS should have User goals (ease of use, convience, etc ...) and System goals (efficiency, security, etc ...).

Policy and Mechanism

Policy is what will be done, *Mechanism* is how it will be done.

They should be seperate, as it would allow for maximum flexibility if policy decisions are to be chanced later.

Implementation

Early OSes were written in Assembly, but now they only have the base of the OS writen in Assembly, and the rest written in high level languages such as C or C++.

Emulation allows for OS to run on non-native hardware.

The Kernel

The Kernel is the core of the OS, which is used to manage system resources and provides essential services for all other parts of the OS and user-level programs.

Handles tasks such as:

- Process management;
- Memory management;
- Scheduling;
- Interfacing with hardware devices.

Operating System Structure

- Simple Structure - MS-DOS
- More complex - UNIX
- Layered - an abstraction
- Microkernel - Mach

Monolithic - OG Unix

Structure

- System Programs
- The kernel, which contains everything between the system-call interface and the hardware, providing File system, CPU scheduling, memory management, etc …

Layered

Like an Onion, the Core being Hardware (Layer 0), and the outer being the User Interface (Layer N). Each layer only has access to the layers below it.

Microkernel

A microkernel is the near-minimum amount of software that can provide the mechanisms needed to implement an OS.

- Easier to extend a microkernel
- Easier to port the operating system to new architecture
- More reliable (less code is running in kernel mode)
- More secure

Modules

Most modern OSes maks use of Loadable Kernel Modules (LKMs) which can be loaded and unloaded at runtime, Each:

- uses Object-oriented approach;
- core component is seperate;
- talks to the other over known interfaces;
- is loadable as needed within the kernel.

Hybrid Systems

Most modern OSes are hybrids of the above systems inorder to address performance, security and usability needs.

Dual-mode Operation

Dual-mode operation allows OS to protect itself and other system components.

Mode bit is used to distinguish between user and kernel mode, which is provided by hardware.

2 Processes

A *process* is a program in execution.

Its execution must be sequential. No parallel executution of a single process.

A program becomes a process once it is **active** and **loaded into memory**.

One program can have several processes. Consider mulitple users executing the same program.

Process Composition

- *Text Section* - Program code
- *Currect Activity* - Program counter, CPU registers
- *Data Section* - Global variables

- *Stack* - Temporary data (function parameters, return addresses, etc ...)
- *Heap* - Dynamically allocated memory

Process State

- *New* - Process is being created
- *Running* - Instructions are being executed
- *Waiting* - Process is waiting for some event to occur
- *Ready* - Process is waiting to be assigned to a processor
- *Terminated* - Process has finished execution

Process Control Block

- *Process State* - New, Ready, Running, Waiting, Terminated
- *Program Counter* - Address of next instruction to be executed
- *CPU Registers* - contents of all process-centric registers
- *CPU Scheduling Information* - Priority, pointers to scheduling queues
- *Memory Management Information* - Page tables, segment tables (memory allocated to the process)
- *Accounting Information* - Amount of CPU and real time used, time limits, account numbers
- *I/O Status Information* - List of I/O devices allocated to the process, list of open files

Threading

A process is an instance of a program in execution. It can contain multiple threads, which are the smallest units of execution within the process. Threads within the same process share the same memory space and resources but can be scheduled and executed independently by the operating system's scheduler.

When a process creates multiple threads, those threads execute concurrently within the same process context, sharing data and resources of the parent process. This allows for parallel execution of tasks, improving the efficiency and performance of the application. Each thread operates independently, but they all belong to the same process and therefore share its resources and address space.

2.1 Process Scheduling

The process scheduler selects whoch process to run in a way to maimize CPU utilization and throughput.

The Scheduler would have a **ready** queue with all processes in the ready state and a **wait** queue with all processes in the waiting state. The Processes migrate between the two queues.

Context Switch

When the CPU switches to another process, the system must save the state of the old process and load the saved state for the next process via a context switch.

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 time.

The time is dependent on hardware support. Some hardware provides multiple sets of registers per CPU so multiple contexts can be loaded at once.

Process Creation

Process makes process which is his child, and those child create more child, making tree. Process have a unique process ID, and is managed by it.

The options for resource sharing are:

- Child Resources $=$ Parent Resources
- Child Resources $\subset$ Parent Resources
- Child Resources $\not\subseteq$ Parent Resources

The options for execution are:

- Parent and child execute concurrently
- Parent waits until child terminates

A child process is either a copy of the parent process or a new program.

`fork()` Creates a new process by duplicating the calling process.

`exec()` system call used after a `fork()` to replace the process' memory space with a new program

Parent process calls `wait()` waiting for the child to terminate

Process Termination

Process executes last statement and then asks the operating system to delete it using the `exit()` system call

Parent may terminate the execution of children processing using the `abort()` system call. Some reasons for doing so:

- Child has exceeded allocated resources
- Task assigned to child is not longer required
- The parent is exiting and the OS does not allow a child to continue if its parent terminates

Termination of a tree could be cascading or non-cascading.

- *Cascading* - Parent process terminates all children
- *Non-cascading* - Parent process terminates, children continue to be terminated by the OS as it relizes the parent is gone.

If no parent is waiting (did not invoke `wait()`) the process becomes a **zombie**. If the parent terminated without invoking `wait()`, the process becomes an **orphan**.

2.2 Interprocess Communication

Processes may be *independent* or *cooperating*. Reasons for cooperating processes are **Information Sharing**, **Computation Speedup**, **Modularity** and **Convenience**. For processes to cooperate, they must be able to communicate with each other, thus *Interprocess Communication* (IPC) is needed.

Producer-Consumer Problem

A producer process produces information that is consumed by a consumer process.

There are two types of variations to this problem:

- *Unbounded Buffer* - The buffer can hold an infinite number of items.
- *Bounded Buffer* - The buffer can hold a finite number of items. If producer tries to add to a full buffer, it must wait until a consumer removes an item. If a consumer tries to remove from an empty buffer, it must wait until a producer adds an item.

Shared Memory

Shared Memory which is used by two or more processes to communicate and share data is to be managed **NOT** by the OS but by the processes themselves. The biggest issue is to provide a way to synchronize their actions when accessing the shared memory. This is an implementation when using Bounded Buffers.

Shared Data

```
#define BUFFER_SIZE 10
typedef struct {
    .   .   .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer

```
item next_produced;

while(true) {
    // Produce an item in next_produced
    while (((in + 1) % BUFFER_SIZE) == out)
        ; // Do nothing

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
item next_consumed;

while(true) {
    while(in == out)
        ; // Do nothing

    next_consumed = buffer[out]
    out = (out + 2) % BUFFER_SIZE;

    // Consume the item in next_consumed
}
```

Filling the Buffer

A `counter` could be used in order to keep track of the number of items in the buffer. The producer would increment the `counter` after adding an item, and the consumer would decrement the `counter` after removing an item. Initially, the `counter` would be set to 0.

Producer with Counter

```
while(true) {
    // Produce an item in_next produced

    while(counter == BUFFER_SIZE)
        ; // Do nothing

    buffer[in] = next_produced
    in = (in + 1) % BUFFER_SIZE
    counter++;
}
```

Consumer with Counter

```
while(true) {

    while(counter == 0)
        ; // Do nothing

    next_produced = buffer[out]
    out = (out + 1) % BUFFER_SIZE
    counter--;

    // Consume the item in next consumer
}
```

Race Condition

A *Race Condition* is an undesirable situation in which the outcome of a program depends on the relative timing of uncontrollable events.

Increment and decrement could be worked out internally as following:

Increment

```
register1 = counter;
register1 = register1 + 1;
counter = register1;
```

Decrement

```
register2 = counter;
register2 = register2 - 1;
counter = register2;
```

Using this raw implementation without locking, the following could happen:

- **Producer** sets `register1` to `counter` **register1 = 5**
- **Producer** sets `register1` to `register1 + 1` **register1 = 6**
- **Consumer** sets `register2` to `counter` **register2 = 5**
- **Consumer** sets `register2` to `register2 - 1` **register2 = 4**
- **Producer** sets `counter` to `register1` **counter = 6**
- **Consumer** sets `counter` to `register2` **counter = 4**

The `counter` should have been 5, but due to the Race Condition, it is 4 (which is **incorrect**).

Message Passing

Message passing is a form of IPC that allows processes to communicate and synchronize by sending and receiving messages.

The *IPC* facility provides two operations:

- `send(message)` - Message is sent to another process
- `receive(message)` - Message is received from another process

Rules to be followed:

- A communication link must be established between them
- Messages must be exchanged via `send`/ `receive`

Implementation issues:

- How are links established?
- Can a link be associated with more than two processes?

- How many links can there be between every pair of communicating processes
- What is the capacity of a link
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional

Communication comes in two types; *Physical* and *Logical*.

Physical

- Shared memory
- Hardware bus
- Network

Logical

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

Direct Communication

Processes must name each other explicitly:

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

Properties:

- Link is established automatically
- A link is associated with exactly two processes
- Between each pair of processes, there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

Indirect Communication

Messages are sent to and received from mailboxes (also known as ports). A mailbox can be owned by a process or the OS. Processes can communicate only if they share a mailbox. Each mailbox has a unique ID.

Properties

- Link is established only if processes share a mailbox
- A link can be associated with many processes
- Each pair of communicating processes may share several communication links
- Links may be unidirectional or bi-directional

Operations

- Create a new mailbox
- Send and recieve messages through mailbox
  - `send(A, message)` - Send a message to mailbox A
  - `receive(A, message)` - Receive a message from mailbox A
- Delete a mailbox

Mailbox sharing

- P_1, P_2 and P_3 share mailbox A
- P_1 sends; P_2 and P_3 receive

Who gets the message?

This can be solved by one of the following:

- 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.

Synchronization

Message passing may be either *blocking* (synchronous) or *non-blocking* (asynchronous).

Blocking

- *Blocking send* - Send is blocked until the message is received
- *Blocking receive* - Receive is blocked until a message is available

Non-blocking

- *Non-blocking send* - Send sends the message and continues
- *Non-blocking receive* - Receive receives a valid message or `null`

Different combinations of blocking and non-blocking operations are possible.

Producer/Consumer Problem with Message Passing

Producer

```
message next_produced;

while(true) {
    // Produce an item in next_produced

    send(next_produced);
}
```

Consumer

```
message next_consumer;

while(true) {
    receive(next_consumed);

    // Consume item in next_consumed
}
```

Buffering

Messages can be buffered by using a queue of message attached to the link. Implementation in one of three ways:

1. *Zero Capacity* - No messages are queued on a link. Sender must wait for receiver (rendezvous).
2. *Bounded Capacity* - A finite number of *n* messages can be queued on a link. Sender must wait if link is full.
3. *Unbounded Capacity* - Messages are never lost. Sender never waits.

2.3 Threads

A *thread* is a basic unit of CPU utilization. It comprises a thread ID, program counter, register set, and stack. It shares with other threads belonging to the same process its code section, data section, and other operating system resources, such as open files and signals. Most modern applications are multithreaded.

Some example tasks that can be done with threads are:

- Updating the display
- Fetching data
- Answering a network request

It is less expensive to create a thread than a process. Threads can be created by the OS or by the process itself. Threads could also simply code and increase efficiency. Kernals are generally multithreaded.

Benefits of Threads

- *Responsiveness* - may allow continued execution if part of the process is blocked, especially important for user interfaces.
- *Resource Sharing* - Threads share resources of the process to which they belong, which is easier than shared memory or message passing.
- *Economy* - Creating a thread is cheaper than creating a process, as threads switching lower overhead than context switching.
- *Scalability* - Threads can be used to take advantage of multicore architectures.

Multicore Programming

Multicore systems put pressure on programmers as they must consider to following when writing code:

- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging

Parallelism implies a system can perform multiple tasks simultaneously. **Concurrency** implies a system can support multiple tasks making progress. Single processor/core, scheduler providing concurrency.

Types of Parallelism

- *Data Parallelism* - Distributing subsets of the same data across multiple cores and performing the same operation on each core.
- *Task Parallelism* - Distributing threads across multiple cores, each performing unique operations.

User Threads and Kernel Threads

User Threads are managed by the user-level thread libraries (such as POSIX Pthreads, Windows threads and Java threads) and not the OS. The OS knows nothing about the threads and manages them as if they were single-threaded processes.

Kernel Threads are supported by the OS and are visible to the OS. The OS schedules the threads and manages them as it would processes.

Multithreaded Models

- *Many-to-One*
  - Many user-level threads mapped to a single kernel thread.
  - If a thread makes a blocking system call, the entire process blocks.
  - Multiple threads may not run in parallel on multicore systems because only one may be in the kernel at a time
  - Few Systems use this model.
- *One-to-One*
  - Each user-level thread maps to a kernel thread.
  - Creating a user-level thread creates a kernel thread.
  - More concurrency than many-to-one.
  - The number of threads per process is sometimes restricted due to overhead.
  - Windows and Linux use this model.

- *Many-to-Many*
  - Allows many user-level threads to be mapped to many kernel threads.
  - Allows the OS to create a sufficient number of kernel threads.
  - Not very common

2.4 Threading issues

Does `fork()` duplicate only the calling thread or all threads? Some UNIXes have two versions of `fork()`.

`exec()` usually works as normal, it replaces the running process including all threads.

Thread Cancellation

This issue arises when one thread needs to terminate another before it has finished. There are two types of thread cancellation:

- *Asynchronous Cancellation* - One thread immediately terminates the target thread.
- *Deferred Cancellation* - The target thread periodically checks if it should be cancelled.

On Linux systems, thread cancellation is handled through signals.

Thread Local Storage

Each thread has its own copy of data. This is useful when a global variable is needed by all threads, but each thread needs its own copy.

Scheduler Activations

Many-to-many models require communication to maintain the appropriate number of kernel threads allocated for the application. Typically use an intermediate data structure between user and kernel threads - lightweight process (LWP)

- Appears to be a virtual processor on which a process can schedule user thread to run.
- Each LWP is attached to a kernel thread

Scheduler activations provide upcalls, a communication mechanism from the kernel to the upcall handler in the thread library. This communication allows an application to maintain the correct number kernel threads.