Operating System

fl_334

Contents

1 Intro to OS **2**

1.1 Computer System Components 2

1.2 Functions of Operating Systems 2

1.3 OS History 2

1.3.1 The First Computers 2

1.3.2 Mainframes: Batch Systems 3

1.3.3 Mainframes: Multiprogramming 3

1.3.4 Mainframes: Timesharing 4

1.3.5 Desktop Operating Systems (1980s) 4

1.3.6 Parallel Operating Systems 4

1.3.7 Distributed Operating System 5

1.3.8 Real-Time Operating System 5

2 OS Structure **5**

2.1 OS System Components 5

2.1.1 Process Manager 5

2.1.2 Main-Memory Management 6

2.1.3 File Management 6

2.1.4 I/O System Management 6

2.1.5 Secondary-Storage Management . . . 6

2.1.6 Networking . . . 6

2.1.7 Protection System . . . 6

2.1.8 Command-Interpreter System . . . 7

2.1.9 Kernel . . . 7

2.2 Operating System Interface . . . 8

2.3 Monolithic Architecture . . . 8

2.4 Layered Structure . . . 8

2.4.1 First Layers-based OS . . . 8

2.4.2 Advantages . . . 10

2.4.3 Layered Architecture . . . 10

2.4.4 Problems . . . 11

2.5 MircroKernel Architecture . . . 11

2.5.1 Architecture . . . 11

2.5.2 Benefits . . . 11

2.5.3 Disadvantages . . . 12

2.6 Modules . . . 12

2.7 OS & Hardware Features . . . 13

2.7.1 Protected Instructions . . . 13

2.7.2 Dual Mode Operation . . . 14

2.7.3 Crossing Protection Boundries . . . 14

2.7.4 Exceptions . . . 15

2.7.5 Memory Protection . . . 16

2.7.6 I/O Control . . . 16

2.7.7 CPU Protection . . . 16

3 Processes & Threads 18

3.1 Processes 18

3.1.1 Program 18

3.1.2 Process in memory 18

3.1.3 Process in Operating System 18

3.1.4 Process States 19

3.1.5 Process Control Block(PCB) 19

3.1.6 Process Switching 20

3.1.7 Overhead in Context Switch 20

3.1.8 Process Creation 21

3.1.9 Process Termination 21

3.1.10 Child Processes 22

3.1.11 Interprocess Communication(IPC) 22

3.1.12 Issues 23

3.2 Thread 23

4 Process Synchronisation 24

4.1 Concurrent Execution 24

4.2 Atomic Operations 25

4.3 Atomic Instructions 25

4.4 Mutual Exclusion & Critical Section 25

4.4.1 Important Definitions 25

4.5 Locking and Mutual Exclusion 25

4.5.1 Requirements for True Solution for CS 27

4.5.2 Desirable Properties of a ME Mechanism 27

4.5.3 Implementations 28

4.6 Semaphores 28

4.6.1 Mutual Exclusion using Semaphores 29

4.6.2 Counting Semaphores 29

4.6.3 Binary Semaphores (Mutex) 29

1 Intro to OS

1.1 Computer System Components

- **Hardware**
  - Basic Computing Resources(CPU, memory...)
- **Operating Systems**
  - Control and Coordination: Manage hardware usage for diverse user applications.
- **Application Programs**
  - Resource Utilization: Utilize system resources to solve problems or complete tasks.
- **Users**
  - User Types: People, machines, or other computers interacting with the computer system.

1.2 Functions of Operating Systems

- **Interface Between User and Hardware**
  - Mediating User-Hardware Interaction
- **Control Interactions Between Users and Programs**
  - Managing User-Program Interactions
- **Provides a Controlled and Efficient Environment**
  - Efficiency and Control
- **Provides Mechanisms and Policies**
  - Resource Management

1.3 OS History

1.3.1 The First Computers

- Early machines (1940s to mid-1950s) had no Operating System.
- The user interacted directly with the hardware.
  - Initial interfaces: console of switches (input) & lights (output).
  - Later interfaces: punched cards, printers, etc.

- **Issues**
  - Long setup time for a program to run.
  - Users accessed the system one at a time.
  - Scheduling made by hand.
  - No sharing of libraries, drivers, and other resources.

1.3.2 Mainframes: Batch Systems

- The earliest Operating Systems were used in mainframes (1950s).
- These OSs were batch systems, which aimed to eliminate the manual setup of programs to be run and provided reusable code to access hardware (i.e., drivers). The Operating System was stored in main memory (referred to as a monitor).
- One job (program) was loaded at a time from a punched card/tape reader into the remaining memory, and job control instructions told the Operating System what to do.
- These simple OSs were code to which one linked one's program (loaded as a whole into main memory) to be run, essentially functioning as a run-time library.
- **Issues**
  - Input/output (I/O) operations were very slow.
  - No computations were done while performing I/O.
  - This decreased CPU usage.

1.3.3 Mainframes: Multiprogramming

- Idea: Expand memory to hold two or more programs and switch among all of them (multitasking or multiprogramming).
- Multiprogramming systems became possible with the advent of the first integrated circuits (IC) in the early 1960s, aiming to increase processor utilization and optimize throughput (i.e., jobs completed per unit of time).
- The degree of multiprogramming refers to the number of jobs that can be managed at once by the OS.
- Multiprogramming (aka multitasking) is the central theme of modern OSs, where multiple runnable jobs are loaded in memory simultaneously, allowing overlap of I/O operations of one job with the computations of another.
- This approach benefits from I/O devices that can operate asynchronously, such as interrupts and direct memory access (DMA).

1.3.4 Mainframes: Timesharing

- Initially, multiprogramming was still batch-based:
  - Turnaround time could be long for any particular job.
  - No interactivity.
- The idea was to have multiple users simultaneously using terminals, with the OS interleaving the execution of each user program in short quanta of computation.
- **Timesharing Systems**
  - Based on time slicing (a.k.a. time multiplexing), where each user feels like using the shared computer as if it were their own.
  - The challenge was to optimize response time.
  - Timesharing systems allow users to view, edit, debug, and run their programs interactively.

1.3.5 Desktop Operating Systems (1980s)

- Very Large Scale of Integration (VLSI) circuits made it cheaper to manufacture complex hardware.
- Hardware became cheaper.
- Easier to have one computer per user than share a mainframe.
- Usability was facilitated by the introduction of graphical user interfaces (GUI).
- The idea was to maximize user convenience and responsiveness, focusing on the user experience, apart from CPU and I/O considerations, similar to multiprogrammed and timesharing systems.

1.3.6 Parallel Operating Systems

The concept behind Parallel Operating Systems is to efficiently run and manage parallel applications on tightly coupled parallel computers (multiprocessors). These operating systems provide support for parallel applications composed of several time-consuming but separable subtasks.

Key features of Parallel Operating Systems include:

- Providing primitives for assigning (scheduling) parallel subtasks to different processors.
- Offering primitives for dividing a task into parallel subtasks, if possible.
- Supporting efficient communication between parallel activities.
- Enabling synchronization of activities to coordinate data sharing.

1.3.7 Distributed Operating System

The concept behind Distributed Operating Systems is to have a common operating system shared by a network of loosely coupled independent computers. These systems facilitate the sharing of resources located in different places, both hardware and software, and appear to their users like an ordinary centralized operating system. Key features include:

- Supporting communication between parts of a job or between different jobs across the network.
- Allowing for some level of parallelism, although speed is not the primary goal.

1.3.8 Real-Time Operating System

Real-Time Operating Systems are designed to guarantee a response to physical events within a fixed interval of time. They are commonly used for specialized applications such as subway systems, flight control, factories, and power stations. Key characteristics of real-time operating systems include:

- Scheduling all activities to meet critical requirements and performing operations within predetermined timeframes.
- The distinction between ”Hard real-time” for critical systems and ”Soft real-time” implemented by all modern PC operating systems to run multimedia applications.

2 OS Structure

2.1 OS System Components

2.1.1 Process Manager

- Process: Program in execution
- How to run and terminate a process
- Suspend/pause and resume a process
- Process synchronisation
- Inter-process communication
- Handling deadlocks
- Keep track what is running in the system
- Run the processes efficiently

2.1.2 Main-Memory Management

- Track what part of memory is being used and by which process
- Decide where to lead a process in the system
- Allocate and free memory as per required

2.1.3 File Management

- Create and delete files and directories
- Basic file and directory manipulation like read/write etc.
- Mapping the files into secondary storage, like a hard disk

2.1.4 I/O System Management

- A device driver interface
- Drivers for specific hardware
- A memory management component implementing buffering, caching and spooling

2.1.5 Secondary-Storage Management

- Free space management
- Storage allocation
- Disk scheduling

2.1.6 Networking

- Implement network stack
- Provide services to connect the OS to the network

2.1.7 Protection System

- Memory protection
- Device protection
- Memory protection...

2.1.8 Command-Interpreter System

- Provide an interface to the user so that they can use the services of the OS
- Can be a command line shell
- Can be a GUI

2.1.9 Kernel

Software containing the core OS components; it may typically include:

- Memory Manager
  - Provides efficient memory allocation and deallocation of memory
- I/O Manager
  - Handles input and output requests from and to hardware devices(through device drivers)
- Inter-process communication (IPC) manager
  - Provides communication between different processes(programs in execution)
- Process Manager (scheduler)
  -Handles what is executed when and where (if more than one CPU)

A OS kernel may consist of many more components

- System service routines
- File System (FS) manager
- Error handling systems
- Accounting systems
- System programs
- And many more

2.2 Operating System Interface

- Original OS interfaces were very simple and called **Command Line Interface(CLI)** or **Command Interpreter(CI)** - The user types a command and the CI executes it - Later systems used a more user friendly Graphical User Interface(GUI) - Based on Desktop idea

- Usually mouse, keyboard, and monitor
- WIMP(windows, icons, menus, pointing)
- Icons represent files, programs, actions, etc

- Used on almost all systems today

2.3 Monolithic Architecture

- every OS component is contained in the kernel
- any component can directly communicate with any other(by means of function calls)
- due to this they tend to be highly efficient(performance)
- Example : MS DOS
  - Written to provide the most functionality in the least space
  - Not divided into modules
  - Interfaces and levels of functionality are not well separated
- Problems
  1. Unstructured – Hard to understand, modify and maintain
  2. All kernel code runs with unrestricted access to the system
     – Susceptible to damage from errant or malicious code

2.4 Layered Structure

Component are grouped into layers that perform similar functions

2.4.1 First Layers-based OS

- Layering was first used in Dijkstra's The OS(1968)
- Each layer "sees" a logical machine provided by the lower layers
  - Layer 4(user space) sees virtual I/O devices

Operating System/img/Example_system.jpg

Operating System/img/MS_DOS.jpg

  - Layer 3 sees virtual console
  - Layer 2 sees virtual memory
  - Layer 1 sees virtual processors

- Based on a static set of cooperating systems

- Each process can be tested and verified independently

2.4.2 Advantages

1. Give the system structure and consistency for designing the system as a number of modules

2. Allows easier debugging, modification and reuse

2.4.3 Layered Architecture

Each layer communicates only with layers immediately above and below it

- Each layer is a virtual machine to the layer above

- A higher layer provides a higher-level virtual machine

Layer 4	User Programs
Layer 3	I/O Management
Layer 2	Console Device(commands), IPC
Layer 1	Memory Management
Layer 0	CPU Scheduling (multiprogramming)
	Hardware

2.4.4 Problems

1. Appropriate definition of layers is difficult
    - A layer is implemented using only those operations provided by lower-level layers
    - A real system is often more complex than the strict hierarchy required by layering
2. Not flexible
    - The secondary memory(disk) driver woould normally placed **above** the CPU scheduler because an I/O wait may trigger CPU rescheduling operation
    - However, in a large system the CPU scheduler may need more memory than can fit in memory: parts of the memory can be swapped to disk(virtual memory), and then the secondary memory driver should be below the CPU scheduler
3. Performance Issues
    - Layer Crossing: Processes' requests might pass through many layers before completion
    - System throughout can be lower than in monolithic kernels
4. Still susceptible to malicious/errant code
    - If all layers have unrestricted access to the system
    - Can only be avoided through hardware

2.5 MircroKernel Architecture

2.5.1 Architecture

- No agreement about minimal set of services inside the microkernel
- At least: minimal process and memory mangament capabilities, plus inter-process communications
- Services such as networking and file system tend to run nonprivileged at the user process level

2.5.2 Benefits

- Modularity
- Promote uniform interfaces
- Distributed systems support
Modules communicate through the microkernel, even through a network
- Reliability
- Scalability
- Portability
- Easy to extent and customise

2.5.3 Disadvantages

System performance can be worse than in monolithic kernels,
especially if kernel minimisation is taken too far

`Operating System/img/microkernel.jpg`

2.6 Modules

Many modern operating systems implement loadable kernel modules

1. Uses object-oriented approach
2. Each core component is separate
3. Each talks to the others over known interfaces
4. Each is loadable as needed within the kernel

Overall, similar to layers but with more flexibility (Linux, Solaris...)

Operating System/img/Solaris_Modular_Approach.jpg

2.7 OS & Hardware Features

2.7.1 Protected Instructions

- Typically users are not allowed to
  1. Directly access I/O (disk, printer, ...)
  2. Directly manage memory
  3. Execute CPU halt instructions
- These operations are always handled through privileged instructions or memory mapping

2.7.2 Dual Mode Operation

- The implementation of protected instructions required some type of hardware mechanism
- The HW must support -at least two operation modes
  - **kernel mode:** access to all the CPU instruction set // also called monitor/system/privileged mode
  - **user mode:** access restricted to a subset of the instruction set
- The mode is indicated by a **status bit(mode bit)** in a protected processor register
  - OS programs & protected instruction executed in kernel mode
  - User programs executed in user mode
- Examples of protection in older and newer systems
  - **MS-DOS(based on Intel 8088):**no protection modes
  - **Windows 2000/XP, OS/2, Linux (based on Interl x86 systems):** protection modes

2.7.3 Crossing Protection Boundries

- User-mode programs cannot execute privileged instructions but they still need kernel=mode services (I/O operations, memory management, etc)
- To execute a privileged instruction, a user must call an OS procedure: **system call**
- A system call causes a **trap**, which jumps to the trap handler in the kernel
- When call the trap handler
  - Uses call parameters to determine which system routine to run
  - Saves caller's state: Program counter(PC), mode bit, ...
- After this the hardware mut
  - Implement caller's parameters verification(e.g. memory pointers should only be allowed within user's section)
  - Return to user-mode when trap system call finished
- The trap is treated by the hardware as a **software-initiated interrupt**

Operating System/img/open_system_call.jpg

2.7.4 Exceptions

- **Exception(hardware-initiated interrupt):** basically the same as a trap
- Automatically triggered by an error or a particular situation rather than on purpose(like in a system call)
- Transfer control to a handler within the OS
  System status can be saved on exceptions(memory dump) so faulty processes can be later debugged
- Decrease performance
  Exception conditions could be detected by inserting extra instructions in the code, but at a high performance cost
- Typical Exceptions
  1. memory access out of user space
  2. overflow, underflow
  3. trace traps(debugging)
  4. illegal use of privileged instructions
  5. virtual memory(paging): page faults, write to read-only page

2.7.5 Memory Protection

- A memory protection mechanism must protect
  - user programs from each other
  - OS(kernel) from user programs
- Simplest scheme is to use **base and limit registers**
  - Based and limit registers are loaded by the OS before starting the execution of any program in user mode
  - base $\leq$ address $\leq$ base + limit; otherwise exception raised

2.7.6 I/O Control

- All I/O instructions are privileged
  This is because a program could disrupt the whold system by issuing illegal I/O instructions
- Two situations must be considered:
  - I/O start: handled by system calls
  - I/O completion and I/O events: handled by interrupts
- Interrupts are the basis for **asynchronous** I/O
  - I/O devices have small processors that allow them to run autonomously (i.e. asynchronously with respect to the CPU)
  - I/O devices send interrupt signals when done with an operation;
    CPU switches to address corresponding to interrupt
    an interrupt vector table contains the list of kernel routine addresses that handle different events

2.7.7 CPU Protection

- A user program might get stuck into an infinite loop and never return control to the OS
- **Timer:**it generates an interrupt after a fixed or variable amount of execution time
- When an interrupt is generated by the Timer the OS may choose to treat the interrupt as a fatal error and stop program execution or allocate more execution time
- In time-sharing systems a time interrupt is periodically generated after a fixed period of time for scheduling a new program

Note 1. 1. An OS provides a number of services

2. They relate to managing

   (a) The hardware

   (b) The processes

   (c) The users

   (d) Communication between all of them

3. Multiple ways of structuring the kernel, the system programs and user programs

4. Each has advantags and disadvantages

OS Requirement	Hardware Feature
Dual kernel/user modes	Protected instructions
System calls	Trap instructions and vectors
Exceptions, signals	Interrupt vectors
Memory protection	Base and limit registers
I/O control	Interrupts
CPU protection, scheduling	Time (clock)

3 Processes & Threads

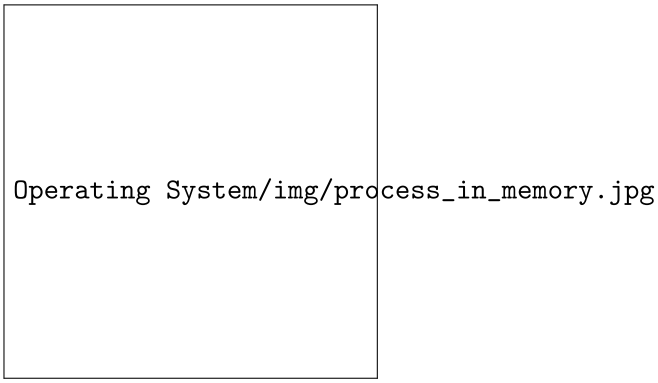
3.1 Processes

3.1.1 Program

- A program is a **passive** enitty stored on a disk
- A program becomes a process when it is **loaded into memory**
- One program can be many processes

3.1.2 Process in memory

- A process is a program in execution
- **text:** The program code
- Current activity including **program counter, processor registers**
- **Stack** containing temporary data
- **Data** section containing global variables
- **Heap** containing memory dynamically allocated during run time



3.1.3 Process in Operating System

- Each process runs in its **own address space**
- The **same** address in two different processes will be stored in two **different** locations in memory

3.1.4 Process States

- - **New:** The process is being created
- **Running:** Instructions a re 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

Operating System/img/process_states.jpg

3.1.5 Process Control Block(PCB)

- PCB contains information associated with each process (also called task control block)
- The OS keeps a either a system-wide or a per-user process table
- each entry contains a PID and a pointer giving the address of that process's PCB in memory

- Process state: running, waiting etc.
- Program Counter (PC): location of the next instruction to execute
- CPU registers - contents of all registers for this process
- CPU scheduling information - priorities, sheduling queue pointers
- Process number(PID)
- Memory-management information - memory allocated to the process
- Accounting information: CPU used, clock time elapsed since start, time limits
- I/O status information: I/O devices allocated to process, list of open files

Operating System/img/PCB.jpg

3.1.6 Process Switching

Using the information stored in the PCB the OS can easily save and load the state of a process

This allow processes to be easily switched; this is called a **context switch**

3.1.7 Overhead in Context Switch

- While a context switch is happening, the system is not doing any work
- The time taken is considered the **overhead** of the operation
- To minimise the amount of time wasted context switches must be fast (hardware dependent)
- Also we don't want to switch too much(too much overhead) or too little(not interactive enough)

Operating System/img/Process_Switching.jpg

3.1.8 Process Creation

Processes are created by two main events

- System boot
- Execution of process creation system call by another process

3.1.9 Process Termination

Processes are terminated in different conditions:

- **Voluntary:** normal exit, error exit
- **Involuntary:** fatal error, killed by another process

voluntary termination can **only** happen from the running state
Involuntary termination can happen from **any** state

3.1.10 Child Processes

- **Process Tree:** A hierarchical process structure is created in this way(any child process only has one parent)
- An **orphan process** is a computer process whose parent has stopped but the process remains running
- Normally these processes are adopted by another parent process
- If this does not happen, they become a **zombie** process
- They should then be removed or reaped from the system
- A failure to reap a zombie process is normally due to a bug in the operating system
- Example(Unix)
  - A child process is created using the fork() system call
    - To the parent the fork system return the PID of the new child process
    - To the child it will return zero
  - The child receives almost everything from its parents
  - In essence the parent address & PCB are copied
  - The child process must have a new PID, and will have different pointers for its parent/child processes
  - Because the PCB is copied the child process begins execution after the fork() instruction

3.1.11 Interprocess Communication(IPC)

- Independent Processes
  - Those that can neither affect nor be affected by the rest of the system
  - Two independent processes cannot share system state or data
  - e.g.: processes running on different non-networked computers
  - **Deterministic** behaviour: only the input state determines the results and the results are **reproducible**
  - Can be stopped and restarted without any problems
- Cooperative Processes
  - Share something
  - Two processes are cooperative if the execution of one of them may affect the execution of the other
  - e.g.: processes that share a single file system
  - **Nondeterministic** behaviour : many factors may determine the result and results may be difficult to reproduce
  - Make testing and debugging difficult
  - Subject to **race conditions**
  - Result may depend on the sequence or timing of events in other processes

3.1.12 Issues

- Not efficient
  - Creation of a new process is costly
  - all the process structures must be allocated upon creation
- don't directly share memory
  - Each process runs in its own address space
  - But parallel and concurrent processes often want to manipulate the same data
  - Most communications go through the OS: **slow**

3.2 Thread

- The idea is that there is more than one active entity(thread of control) within a single process
- Concurrency in some existing OS
  - MS-DOS: one address space, one thread
  - Unix(originally): multiple address spaces, one thread per address space
  - OSX, Solaris, Windows 10: multiple address spaces, multiple threads per address space (multi-threading)
- Threads & Processes
  - **Process:** defines the address space and general process attributes
  - **Thread:** defines a single sequential execution stream within a process
  - All threads belonging to a process share almost everthing in the process:
    - Address space (code and data)
    - Global variables
    - Privileges
    - Open files
    - Timers
    - Signals
    - Semaphores
    - Accounting information
  - Threads however **do not share**
    - Register set, in partucular
      - Program counter(PC)
      - Stack pointer(SP)
      - Interrrupt vectors
    - Stack
    - State
    - Child threads

  - Cheap to create(no need to allocate PCB, new address space)
  - Communicate with each other efficiently through the process global variables or through common memory, using simple primitives
  - Facilitate concurrency, and therefore are useful even on uniprocessor systems
  - Threads can be created statically or dynamically (by a process or by another thread)
  - If a thread needs a service provided by the OS (system call) it acts on behalf of the process it belongs to

- Thread Implementations
  - In user space(many-to-one model)
    - The kernel schedules **processes** (does not implement or know about threads)
    - **per process thread table:** process decides which of its threads to run when it is running
    - Advantages
      - A single-threaded OS can emulate multi-threading
      - thread scheduling controlled by run-time library, no system call overheads
      - portability: a OS independent user-space threads library is possible
    - Disadvantages
      - If a thread blocks, all other threads belonging to the same process are blocked too because the OS only schedules processes
  - In kernel space(kernel threads, one-to-one model)
    - The kernel schedules threads
    - In this case threads are the smallest units of scheduling
    - **System-wide thread table** (similar to a process table)
    - Advantages
      - Individual management of threads
      - Better interactivity
    - Disadvantages
      - Scheduling and synchronisation operations always invoke the kernel, which increases overheads
      - Less portable
  - Hybrid(many-to-many model)

4 Process Synchronisation

4.1 Concurrent Execution

- Concurrent Execution is when two processes are executing at the same time
- Each cooperative processes may affect the execution of the other process
  - by changing the resources available
  - by changing the value of memory

4.2 Atomic Operations

Note 2. References and assignment are all **atomic** in the CPU

All read and write operations happen as a single step.

An atomic operation cannot be interrupted to prevent illogical things happening

4.3 Atomic Instructions

This atomicity is provided by hardware

Higher-level constructs are not atomic in general

A higher-level construct is any sequence of two or more instructions

Note 3. Process synchronisation is all about making high-level constructs behave atomically

4.4 Mutual Exclusion & Critical Section

4.4.1 Important Definitions

Definition 1 (Synchronisation)**.** Ensuring proper cooperation among processes or threads by relying on atomic operations

Definition 2 (Mutual Exclusion(ME))**.** ENsuring that only one process at a time holds or modifies a shared resource

ME ensures atomicity

Definition 3 (Critical Section(CS))**.** A section of code in a program win which share resources are manipulated

Mutual exclusion in a critical section requires serialisation of process access to the critical section

4.5 Locking and Mutual Exclusion

Achieving mutual exclusion in a critical section always involve some sort of **locking mechanism**

Locking is where we prevent someone else for doing something with the shared resource

Locking involves three rules:

- Lock before entering a critical section

- Unlock when leaving a critical section
- Wait when trying to enter critical section if it is locked

4.5.1 Requirements for True Solution for CS

- Mutual exclusion
  1. One process at most inside the CS at any time
- Bounded waiting(no starvation)
  1. A process attempting to enter its CS will eventually do so
- Progress
  1. A process executing outside a CS cannot prevent another process from entering it
  2. If several processes are attempting to enter a CS at the same time the decision on which one goes in cannot be indefinitely postponed
  3. A process cannot stay inside its CS forever(or exit in there)

These conditions are necessary and sufficient for process synchronisation provided that basic operations are atomic

No assumptions are made about:

- number of processes
- relative speed of processes
- underlying hardware

4.5.2 Desirable Properties of a ME Mechanism

- Simple
- Systematic and easy to use
- Easy to maintain
- Efficient
  1. Does not use a lot of resources while waiting
  2. Overhead due to entering and leaving critical sections has to be small, at least smaller than the work inside it
  3. Scalable
     (a) It should work when many threads share the critical section

4.5.3 Implementations

1. Semaphores
   (a) Simple, but hard to program with
   (b) Very low level
2. Monitors
   (a) Higher level mechanism
   (b) Requires higher-level programming languages
3. Messages
   - Synchronisation without shared memory
   - Uses IPC messages instead

4.6 Semaphores

A semaphore is a protected integer variable S with an associated queue of waiting processes

Only two atomic operations $P()$ and $V()$ can be performed on S

Algorithm 1 Semaphore Operations

```
Semaphore $s \leftarrow$ initialValue                ▷ Initialize the semaphore

function P($s$)
    if $s > 0$ then                                  ▷ If the semaphore is greater than zero
        $s \leftarrow s - 1$                         ▷ Decrement the semaphore
    else
        wait                                         ▷ Wait for the semaphore to become positive
    end if
end function

function V($s$)
    $s \leftarrow s + 1$                             ▷ Increment the semaphore
    signal                                           ▷ Wake up waiting processes
end function
```

Initial value of S is **how many** processes can enter critical section at the same time

Processes in the queue are **blocked**

The operations are **atomic**, so only one process at a time can execute them

4.6.1 Mutual Exclusion using Semaphores

- Initialise S at 1
- To enter critical section we execute **P** on its semaphore
- When leaving the critical section we execute **V** on its semaphore

4.6.2 Counting Semaphores

- If we initialize $S > 1$, more than one process at a time can get into the CS.
- This means that there is no mutual exclusion.
- This is used in a particular type of semaphores called counting semaphores.

4.6.3 Binary Semaphores (Mutex)

- A semaphore with the initial value of 1 is also called a binary semaphore or mutex. It enforces mutual exclusion, allowing only one process at a time.