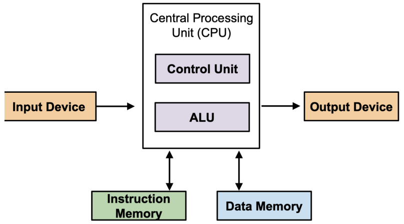
CS2106
AY22/23 Sem 2
github.com/securespider

01. Introduction

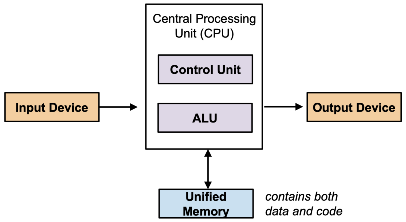
OS - Program that acts as an intermediary between user and hardware

Different architectures

Harvard architecture



Von Neumann architecture



Difference Separate vs common storage pathway for code and data

Why do we need OS?

Mainframe

Old analog "computers" using physical cards for programming

Improvements

- Problem: Batch processing inefficient
- Solution: Multiprogramming
  - Loading multiple jobs that runs while other jobs using I/O
  - Overlapping computation with I/O
- Problem: Only one user
- Solution: Time sharing OS
  - Multiple concurrent users using terminals
  - User job scheduling
  - Memory management
  - **Hardware virtualization** - Each program executes as if it had all resources

Motivation

1. Abstraction
   - Hide low level details and present common, high-level functionality to users
2. Resource allocation
   - Allow concurrent usage of resource and execute programs simultaneously
   - Arbitrate conflicting request fairly and efficiently
3. Control programs
   - Restrict resource allocation
   - Security, protection and error prevention
   - Ensure proper use of device

Advantage

- Portable and flexible
- Use computer resources efficiently

Disadvantage

- Significant overhead

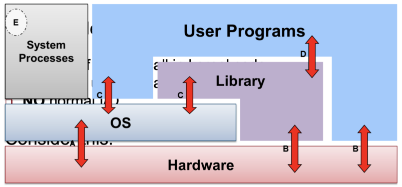
OS vs User Program

Similarities

- Both softwares

Difference

- OS runs in **kernel mode** - Access to all hardware resources
- User programs run in **User mode** - Limited access
- User programs use syscalls to communicate with OS for hardware processes



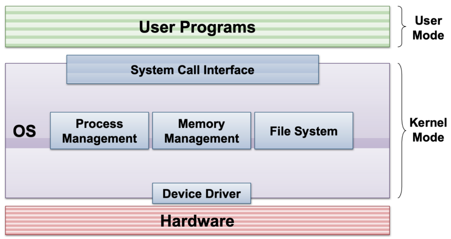
Why OS dont occupy entire hardware layer

- Slow to have all operations pass through intermediary
- User programs can have direct interaction with hardware (eg. Arithmetic) during low risk operations

OS structure

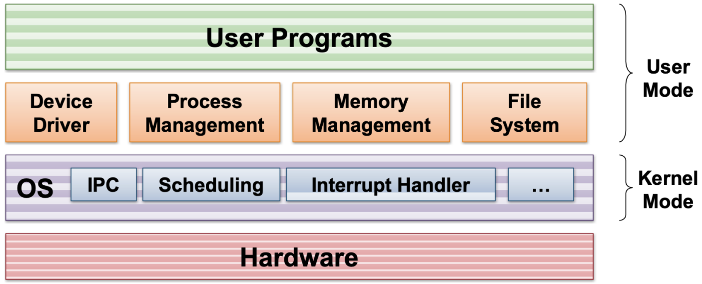
Monolithic OS

- One big kernel program
- Well understood and has good performance
- Highly **coupled** - internal structure interconnected that unintentionally affect each other



Microkernel

- Small clean
- Basic and essential facilities
- IPC communication OR run external programs outside OS
- Robust and more **modular** - Extendible and maintainable
- Better isolation btw kernel and services
- Lower performance

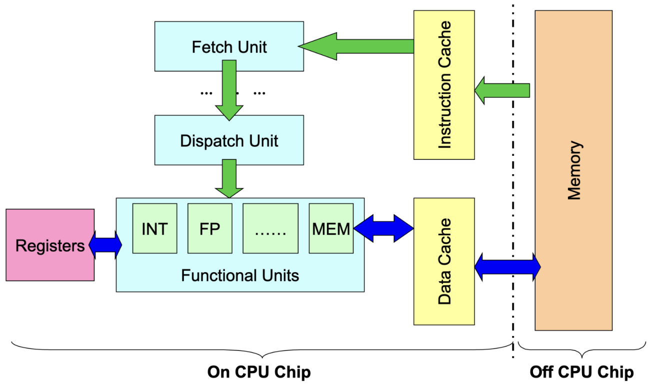


02. Process abstraction

Motivation

- Allow concurrent usage of hardware
- Multiple programs sharing the same processors/IO

Computer organisation



Memory

- Storage for instruction and data
- Managed by the OS
- Normally accessed via load/store instructions

Cache

- Fast and invisible to software
- Duplicate part of the memory for faster access
- Usually split into instruction and data cache

Fetch

- Load instructions from memory
- Location indicated by **Program Counter**

Functional units

- Carry out instruction execution
- Dedicated to specific instruction type

Registers

- Internal storage for fastest access speed

Information needed

- Memory context
  - Code
  - Data
- Hardware context
  - Register
  - PC value
  - Frame Pointer
- OS context
  - Process properties
  - Resources used
  - Files

Function calls

Separation of text and data

Suppose a function f() calls g()

- f is caller and g is callee

Steps of control flow

1. Setup parameters
2. Trf ctrl to callee
3. Setup local var
4. Store any results
5. Return ctrl to caller

Issues

Control Flow

- Need to jump to functional body when callee called
- Need to resume to next instruction in caller after done

Data storage

- Need to pass parameters to function
- Need to capture return result
- May have local variables

Additional

- May lead to overriding of data in caller by callee (interference)
- Calling g() multiple times may lead to insufficient space and overriding

Stack memory

Memory to store function invocation **Stack Pointer** - Indicates the first free location in the stack region
Frame Pointer - Points to the frame and is used for traversing around the stack easily

- On executing function call:
  - **Caller**: Pass arguments with registers and/or stack
  - **Caller**: Save Return PC on stack
  - **Transfer control from caller to callee**
  - **Callee**: Save registers used by callee. Save old FP, SP
  - **Callee**: Allocate space for local variables of callee on stack
  - **Callee**: Adjust SP to point to new stack top
- On returning from function call:
  - **Callee**: Restore saved registers, FP, SP
  - Transfer control from callee to caller using saved PC
  - **Caller**: Continues execution in caller

Information needed for function invocation - Stack frame

- Return address of caller
- Arguments for the function
- Local variables
- Stack and frame pointer of caller
- GPR values (register spilling)

Callee stack frame will be on top of the caller

Dynamic memory (Heap)

Memory that the program/user specifies manually (eg. malloc, new)
Problems:

- Allocated only at runtime
  - Size not known at program compilation time
  - Cannot specify a region in data
- No definite dellocation timing
  - Must be freed explicitly by the program
  - Cannot place in stack region

Solution:
Add a region "Heap for dynamic allocation
Problems with heap memory:

- Generation of holes in between data due to variable deallocation timing

OS context

Process identification

Features:

- Distinguish processes from each other (Unique)
- Communicated to the hardware

Process state

- Denotes whether a process is running or not (Running vs waiting vs not running)

Generic 5-State Process Model

- **New**:
  - New process created
  - May still be under initialization ➔ not yet schedulable
- **Ready**:
  - process is waiting to run
- **Running**:
  - Process being executed on CPU
- **Blocked**:
  - Process waiting (sleeping) for event
  - Cannot execute until event is available
- **Terminated**:
  - Process has finished execution, may require OS cleanup

Process control block - Table representing all processes

Exceptions and interrupts

Exceptions

- Synchronous (due to program execution)
- Machine level instructions arise errors
- Exception handler executed automatically in software

Interrupts

- Asynchronous (Can happen anytime)
- External events that cause execution to fail (hardware related errors)
- Program execution suspended and **interrupt handler** executed automatically

Instruction execution

1. Read byte from PC and decode instruction
2. Read 2 bytes to get the address/operands
3. Perform ALU operations
4. Store result into destination
5. Check if any interruptions

Interrupts can happen at anytime, and will remain pending until step 5 where it is handled

Interruption handling

1. Push PC and status register into hardware
2. Disable interrupts
3. Read **Interrupt Vector Table** - Table where the OS stores address of all interrupt handlers
4. Switch to kernel mode
5. Set PC to handler address and execute the instructions

- OS populates the IVT table with address of interrupt routines
- Hardware reads IVT to locate the handler

System calls

Application Program Interface - Provides way of calling facilities/services in kernel
Instructions can only be done in kernel mode

Method

- Library version with the same name and same arguments
- User friendly library version
- using the function $long\ syscall(long\ number);$

Mechanism

1. User invoke library call
2. Place call number in the designated location
3. Library call executes a special instruction (**TRAP/syscall**) to change user to kernel mode
4. (in kernel) syscall handler is determined (by a **dispatcher**)
5. syscall handler is executed
6. Syscall handler ends and control returned to the library call
7. Return to user mode and continue normal function mechanism