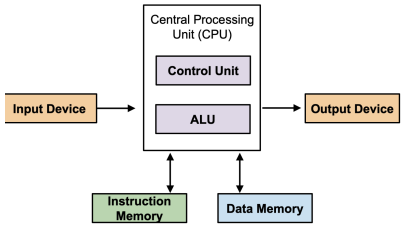
CS2106
AY22/23 Sem 2
github.com/securespider

01. Introduction

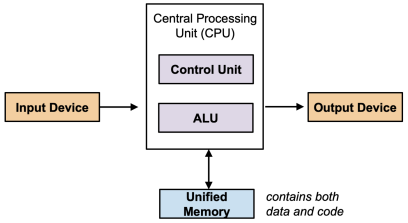
OS - Program that acts as an intermediary between user and hardware

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

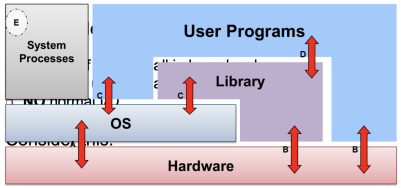
OS vs User Program

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



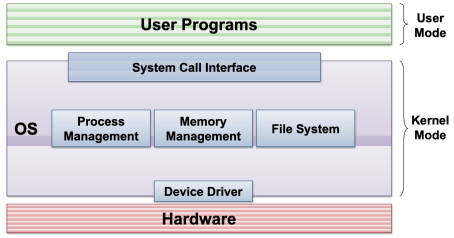
Why OS dont occupy entire hardware layer

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

OS structure

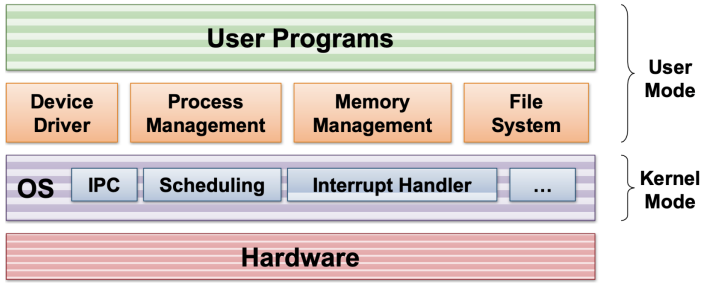
Monolithic OS

- 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

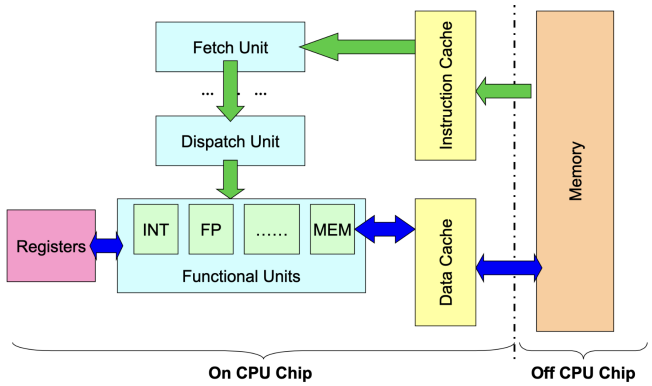


02. Process abstraction

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

03. Process abstraction in Unix

Process information

Pid

Process state Running, sleeping, stopped, **zombie** - Process that has stopped but resources not cleared

Parent pid

Cumulative CPU time For scheduling

fork()

Process creation

Package unistd.h and sys/types.h

return PID of newly created process(parent) and 0 (child process)

Behaviour

- Creates a child process
  - **Copy** data of parent (Independent memory space)
  - Sane code, same address space
  - Differs by pid, ppid and fork() return value

Implementation

Clone the parent process

1. Create address space of child process
2. allocate new pid to child and pass to parent
3. Create kernel process data structures
4. Copy kernel environment of parent process
5. Initialize child process context (pid, ppid, cpu_time = 0)
6. Copy memory regions from parent
   - Very expensive operation
   - Code, data, stack
7. Acquire shared resources
8. Initialize hardware context for child process (copy parent registers)

Problem: Memcopy is very expensive operation
Solution: Copy on write

- Only duplicate a memory location when it is written to

exec(@param)

Replace current executing process image

- code and data replaced
- PID intact

Format

param char *path, char *arg0...

- Note that last term MUST be **NULL** indicating end of argument list

header unistd.h

exit(status)

return Does not return anything

param int status - returned to the wait call

- Most system resource used by process are released on exit
- return from main() implicitly calls exit(0)
- Basic processes are not releasable
  - Pid and status
  - Process accounting info

wait(&status)

Parent child synchronisation

param &status - address to put return value

header sys/types.h and sys/wait.h

return pid of terminated process

- Call is blocking - suspend operation until at least one child terminates
- Cleans up remainder of child system resources (PID, status)
- waitpid - used for waiting for specific child process

Orphan and zombie process

Zombie - Process that has exited but parent did not call wait
Orphan - Child process whose parent has been terminated

- Parenthood will be propagated up to init which may use wait() to clean up

04. Inter Process Communication

Mechanism

- Shared memory
- Message passing
  - Pipes
  - Signal

Shared memory

Communication through read/write to shared memory
Advantages

Efficient Only require OS to setup shared region once

Ease of use Simple reads and write to memory

- Implicit communication
- Can store any type of information

Disadvantages

Limited to single machines Less efficient over different system

Requires synchronization Might have data races without synchronisation

Race condition - System behaviour is dependent on the context/interleaving of process → unpredictable outcome Steps of usage

1. Create/locate a shared memory region M
2. Attach M to process memory space
3. Read from/write to M
4. Detach M from memory space after use
5. Destroy M
   - Only one process need to do this
   - Can only destroy if M is not attached to any process

Message Passing

Explicit communication through exchange of message

Naming Have to identify the parties in the communication

Synchronisation Behaviour of sending/receiving operations

- Messages have to be stored in kernel memory space
- All sending/receiving operations have to be done through syscalls

Direct communication

Sender/receiver explicitly name parties in communication

- One link per pair of communicating processes
- Need to know identity of other party

Indirect communication

Messages sent to message storage (**Mailbox/Port**)

- Can be shared among a number of processes

Synchronisation behaviour

Blocking

- send and receive is blocked until a message has received/sent (other party is ready)

Non-blocking (asynchronous)

- execute immediately and sends information to somebody OR returns empty handed

Typically, receive is synchronous and send is asynchronous
BUT send just buffers the message and only sends the message when the receiver is receiving (no loss)
Message buffers

- Under OS control
- Decouples sender and receiver - less sensitive to variation in execution
- Mailbox capacity declaration in advance

Rendezvous

- Synchronous message passing
- Sender is blocked until receiver sends matching receive
- No buffering needed

Pros

- Applicable beyond a single machine
- **Portable** - Easily implemented on many platforms and processing environments
- Easier synchronisation
  - Implicit via send/receive behaviour
  - Communication and synchronisation is done simultaneously

Cons

Inefficient Usually requires OS intervention every operation

Difficult to use Requires information packing into specific format

Unix Pipes

3 different communication channels to user

1. stdin - standard in bounded to keyboard input
2. stderr - used for error messages
3. stdout - linked to screen

"|" symbol for linking input/output channels of each process to each other
Create communication channel with 2 ends (one reading, one writing)

- Producer-consumer relationship
- Like anonymous file reading information FIFO

Synchronisation

- Circular bounded byte buffer with implicit synchronization
- Writer wait when buffer is full
- Reader wait when buffer is empty
- Unidirectional (one write one read exclusively) vs bidirectional (any end for reading and writing)

pipe(@param)

param array of file descriptors

return 0 for success, != for errors

05.Synchronisation

06.Synchronisation primitives

07.Threads

Motivation

- Processes are expensive
  - Process creation via fork() duplicates memory space and process context
  - Requires context switching and saving/restoring process information repeatedly
- Hard to communicate with each other
  - Independent memory space (IPC communication only)
- Need easy way to execute some instructions simultaneously (more threads of control to same process)

Features

- **Multithreaded process** - Single process with multiple threads
- Threads share the same **Memory context**(Text, data, heap) and **OS context**(PID)
- Contains unique information
  - Identification (thread id)
  - Registers (GPR and special)
  - Stack

Benefits

Economy Less resources to manage

Resource sharing

Responsiveness

Scalability Can take advantage of multiple CPUs

Problems

- Syscall concurrency
  - Parallel execution of threads may result in parallel syscalls
- Process behaviour
  - Unable to impact process operations

Thread implementation

1. User thread
   - Implemented as a user library - More flexible and configurable
   - Runs on any OS
   - Kernel unaware of threads in process (scheduling at process level)
2. Kernel thread
   - Implemented in the OS through syscalls (slower and more resource intensive)
   - Scheduling among threads instead of via process
   - Use threads in kernel operations
   - Less flexible
     - Used by all multithreaded programs so kernel must cater to all
     - Balance btw many and less features
3. Hybrid thread
   - OS schedules on kernel threads
   - User threads can be bound to kernel threads
   - Lead to greater flexibility

Posix Threads (pthreads)

pthread_create

param tid, attributes, &function, function_params

return 0 success, != 0 fail

pthread_exit

Automatically called after function finishes

param exit_value

behaviour return value of function will be the "exit value"

pthread_join

param tid, **&status** - Exit value returned by target pthread

08.Synchronization classics