CS2106 Intro Op. Systems Notes

AY23/24 Sem 2, github.com/gerteck

1. Introduction

Course objectives: Introduces basic concepts in operating systems. Focusing on OS structure and architecture, process management, memory management, file management and OS protection mechanism. At the end of the course, identify and understand major functionalities of modern operating systems and extend and apply the knowledge in future related courses.

Supplementary Text: Modern Operating System (5th Edition), by Andrew S. Tanenbaum, Pearson, 2023.

Learning Outcomes

- Understand how an **OS manages computational resources for multiple users and applications, and the impact on application performance**
- Appreciate the **abstractions and interfaces provided by OS**
- Write **multi-process / thread programs** and avoid common pitfalls such as **deadlocks, starvation and race conditions.**
- Write system programs that utilizes **POSIX** syscall for process, memory and I/O management.
- Self-learn and explore advanced OS topics.
- Understand important design principles in complex systems.

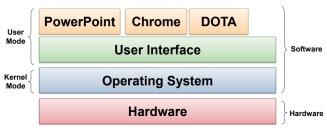
Areas to focus on: Try to understand how things are running in parallel, since we naturally think sequentially. Secondly, how we can manage memory and how they combine and interact (in strange ways), synchronization.

Operating System OS

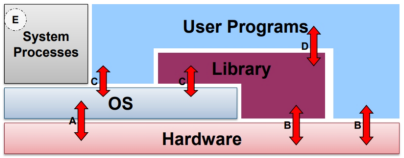
An OS is a program that acts as an intermediary between a computer user and the computer hardware. Motivation for OS:

- Manage resources and coordination. (Resource Allocator: Process synchronization, resource sharing)
- Simplify programming (Abstraction of hardware / hardware virtualization, convenient services)
- Enforce usage policies
- Security and protection
- User Program Portability (across different hardware)
- Efficiency (Optimize for particular usage and hardware).

Kernel Mode: Complete access to all hardware resources.
User Mode: Limited / Controlled access to hardware resources.



Generic OS Components



- **A**: OS executing machine instructions
- **B**: normal machine instructions executed (program/library code)
- **C**: calling OS using ***system call interface***
- **D**: user program calls library code
- **E**: system processes
  - Provide high level services, usually part of OS

- OS is known as the **kernel**.
  Program that deals with hardware issues, provide system call interface and special code for interrupt handlers, device drivers.
- Kernel code is different from normal programs:
  No use of system call in kernal code, can't use normal libraries, no normal I/O (must do I/O itself).
- **Implementing OS**: Historically in assembly/machine, now in HLLs (C, C++). Heavily hardware achitecture dependent. Challenges include complexity, debugging, codebase size.

OS Structures

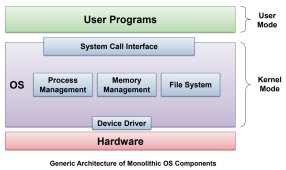
Monolithic OS: One Big program.

- Well understood, good performance, but highly coupled components (everything running in kernel mode) and usually devolved into very complicated internal structure.

Microkernel OS:

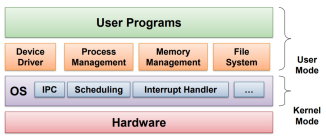
- Kernel is very small and clean, only providing basic and essential facilities.
- Inter-Process Communication (**IPC**), Address space management, Thread management etc.
- Higher level services are built on top of basic facilities, run as server process *outside* of OS, use IPC to communicate.
- Kernel is more robust and extendible, better isolation and protection between kernel and high level services. But, lower performance. (Latency)

Monolithic Kernel Illustration



Generic Architecture of Monolithic OS Components

Microkernel Components



Generic Architecture of Microkernel OS Components

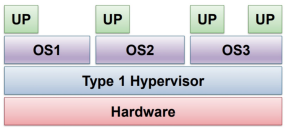
Other OS Structure

- **Layered Systems**: Generalization of monolithic system, organize components into hierarchy of layers. Lowest is hardware, highest is user interface.
- **Client-Server Model**: Variation of microkernel. Two classes of processes: Client p. request service from server process, server process built on top of microkernel. Client & Server process can be on separate machine.

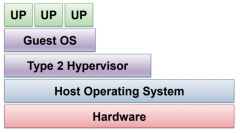
Virtual Machines

- **Motivation**: OS assumes total control of hardware, making it hard to run several OS on same hardware at same time. OS is also hard to debug / monitor, hard to observe working of OS, test potentially destructive implementation.
- **Virtual Machine**: Software emulation of hardware.
- **Virtualization of underlying hardware**: Illusion of complete hardware to level above. (Memory, CPU etc.) Normal OS can then run on top of virtual machine. Aka **Hypervisor**.
  - **Type 1 Hypervisor**: Provides individual virtual machines to guests OSes (e.g. IBM VM/370)
  - **Type 2 Hypervisor**: Runs in host OS, Guest OS runs inside Virtual Machine, (e.g. VMware)

Type 1 Hypervisor



Type 2 Hypervisor



- Upcoming Topics -

OS Process Management: As OS (to maximise efficiency hardware resources), to be able to switch from running one program to the other (share hardware, e.g. CPU), requires information regarding execution of A stored, and A's information replaced with B's information to run. (E.g. the registers in CPU replaced)

- **2. Process Abstraction**: Info describing executing program
- **3. Process Scheduling**: Deciding which process gets to execute
- **4. Inter-Process Communication**: Passing information between processes (tough)
- **5. Threads + Synchronization**: Alternative to Process (Light-weight process aka Thread)
- **6. Memory Management**
- **7. Disjoint Memory Management**
- **8. File System Management**
- **9. File System Implementation**

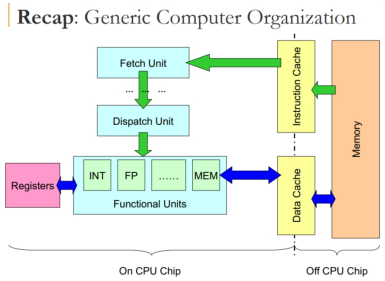
2. Process Abstraction

To switch programs, requires information of both programs. Hence, we need abstraction to describe running program, aka **process**.

- **(Process / Task / Job)** is a dynamic abstraction for executing program.
- It is info required to describe a *running program*:
  - Memory Context (Code/Text, Data, Stack, Heap),
  - Hardware Context (Register/PC, Stack/Frame Pointer),
  - OS Context (Process Properties (PID, State), Resources Used).

Computer Organization (Recap)

- **Components**: Memory, Cache, Fetch Unit (Loads instruction, location indicated by special register **PC**.)
- **Functional Units** (Carry out instruction execution, dedicated to diff. instr. type) (CS2100 looked at INT func. unit)
- Registers (Internal storage, fastest access speed).
  - **GPR**: General Purpose Register, accessible by user program / compiler.
  - **Special Registers**: PC, Stack/Frame Pointer, PSW etc.
- **Binary Executable File**: file in machine language (built by compiler) for specific processor:
  - Executable (binary) consists two major components: Instr. (Text) & Data
  - When under execution, more info: Memory, Hardware, OS context.



Memory Context for Function Call (Stack Memory)

Memory Context Challenges of Functional Calls:

- Control Flow Issues: Need to jump to function body, resume after, need to store PC of caller.
- Data Storage Issues: Need to pass params to function, capture return result, may need declare local variables.
- Require region of memory dynamically used by function invocations.

Hence, portion of memory space used as **stack memory** that stores executing function using **stack frame**, which includes usage of *Stack Pointer, Frame Pointer*.

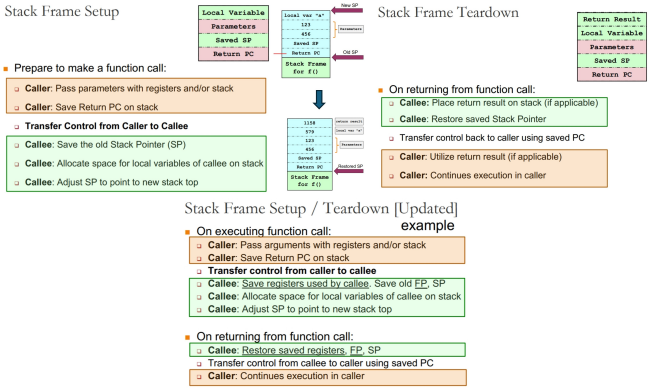
Stack Memory Region

- **Memory region to store information function invocation.**
- **Stack Frame**: Describes information of function invocation.
- Stack frame added on top when function is invoked, stack "grows", removed from top when function call ends, stack "shrinks".
- Stack Frame contains return PC address of caller, arguments for function, storage for local variables, add on.... (AKSJKFGJHKLJHD)
- **Stack Pointer**: Indicates top of stack region (first unused memory location). Usually indicated in specialized register.

Function Call Convention: Stack Frame Setup / Teardown

There are different ways to setup stack frame, known as function call convention, differences about (info stored in frame, which portion of stack fram prepared & cleared by caller / callee etc). Dependent on hardware & programming language.

Example Scheme:



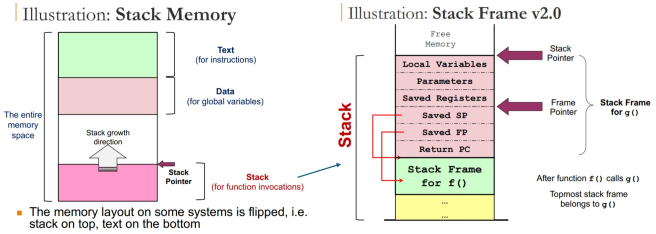
Stack Frame: Other Information

Frame Pointer

- To facilitate access of various stack frame items. As stack pointer hard to use as it can change, some processors provide dedicated register Frame Pointer.
- Frame Pointer points to fixed location in stack frame, other items accessed as displacement from frame pointer, usage of FP is platform dependent.

Saved Registers

- Since number of GPR limited, when GPR exhausted, use memory to temp. hold GPR values for reuse.
- **Known as Register Spilling**. Function can spill registers it intends to use before function starts, then restore registers at end of function.



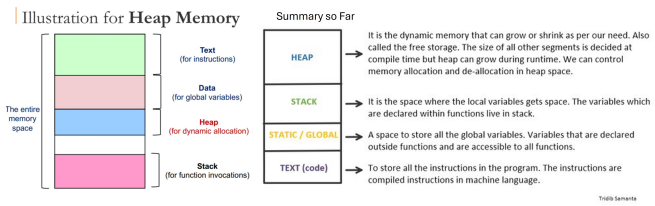
Memory Context for Dynamically Allocated Mem. (Heap)

Most programming languages allow dynamically allocated memory, i.e. acquire memory space during execution time.

- In C, `malloc()` function call, while C++ / Java: `new` keyword.
- Cannot place in *Data region*, as allocated at runtime, size not known during compilation.
- Cannot place in *Stack region*, as no definite deallocation time, cannot be freed by garbage collector.
- Solution: Set up separate **Heap Memory Region**

Heap Memory

- Managing heap memory trickier due to variable size, variable allocation / deallocation timing.
- Common situation where heap memory alloc/dealloc creating "holes" in memory. Free memory block squeezed between occupied memory blocks.
- Covered in memory management.



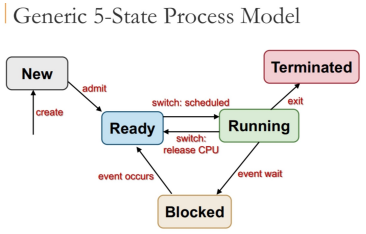
OS Context: Process ID, Process State

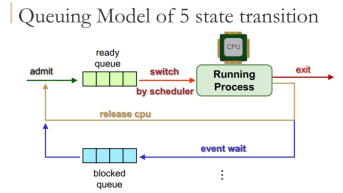
Process Identification:

- **Process ID**: To distinguish processes from each other (Just a number, unique among processes)
- PIDs are OS dependent as well, including if PIDs reused, if limits maximum no. of processes or any PIDs reserved.

Process State:

- Processes require a process state as indication of execution status. (Running / Not Running / Ready to Run etc.)
- **Process Model**: Set of states and transitions, describes behaviours of a process.
- **Global View of Process States**: Given n processes,
  - With 1 CPU, $\leq$ 1 process in running state, 1 transition at a time.
  - With m CPUs, $\leq$ m processes running state, possibly parallel transitions.
- Different processes may be in different states, each process may be in different part of its state diagram.
- **5-State Process Model**:





Process States for 5-Stage Model

- **New**:
  - New process created
  - May still be under initialization → not yet ready
- **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 State Transitions in 5-Stage Model

- **Create** (nil → New):
  - New process is created
- **Admit** (New → Ready):
  - Process ready to be scheduled for running
- **Switch** (Ready → Running):
  - Process selected to run
- **Switch** (Running → Ready):
  - Process gives up CPU voluntarily or *preempted* by scheduler
- **Event wait** (Running → Blocked):
  - Process requests event/resource/service which is not available/in progress
  - Example events:
    - System call, waiting for I/O, *(more later)*
- **Event occurs** (Blocked → Ready):
  - Event occurs → process can continue

End.