电子科技大学
格拉斯哥学院
Glasgow College, UESTC

UESTC4019: Real-Time Computer Systems and Architecture

Lecture 11

Overview of an Operating System (OS)

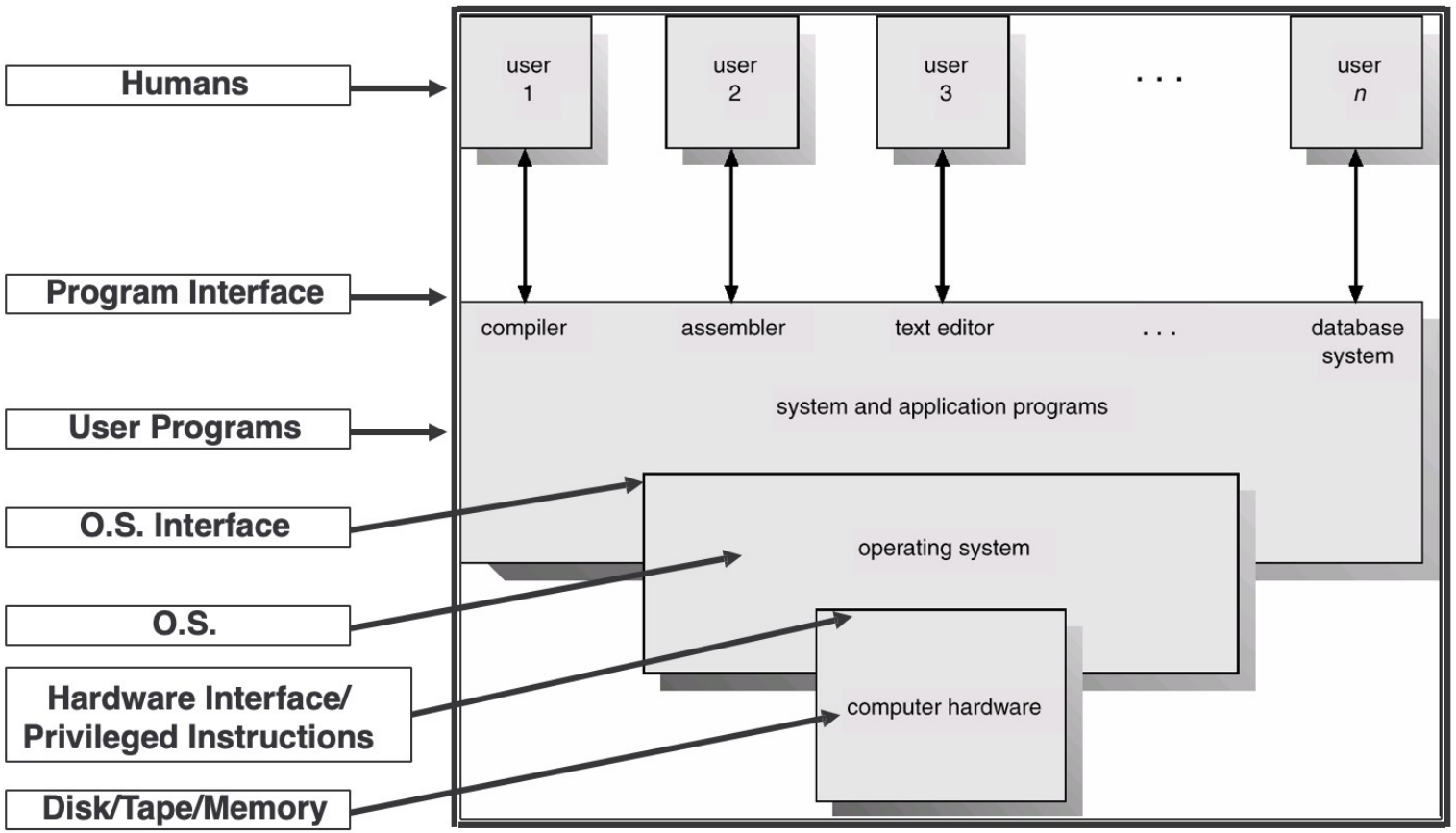
What is OS? (1 of 2)

- An operating system (OS) is the program that manages all the other programs in a computer
- The other programs are called applications or application programs
- The application programs make use of the operating system by making requests for services through a defined application program interface (API)
- Users can interact directly with the operating system through a user interface such as a command line or a graphical user interface (GUI)

What is OS? (2 of 2)

- OS provides an interface between users and hardware - an environment "architecture”
- Allows convenient usage; hides the tedious stuff
- Allows efficient usage; parallel activity, avoids wasted cycles
- Provides information protection
- Gives each user a slice of the resources
- Acts as a control program

Operating Systems Overview



OS Services (1 of 2)

- An operating system performs these services for applications:
  - In a multitasking operating system where multiple programs can be running at the same time, the operating system determines which applications should run in what order and how much time should be allowed for each application before giving another application a turn
  - It manages the sharing of internal memory among multiple applications
  - It handles input and output to and from attached hardware devices, such as hard disks, printers, and dial-up ports.

OS Services (2 of 2)

- An operating system performs these services for applications:
  - It sends messages to each application or interactive user (or to a system operator) about the status of operation and any errors that may have occurred.
  - It can offload the management of what are called *batch* jobs (for example, printing) so that the initiating application is freed from this work.
  - On computers that can provide parallel processing, an operating system can manage how to divide the program so that it runs on more than one processor at a time.

How did the OS come into existence (1 of 2)

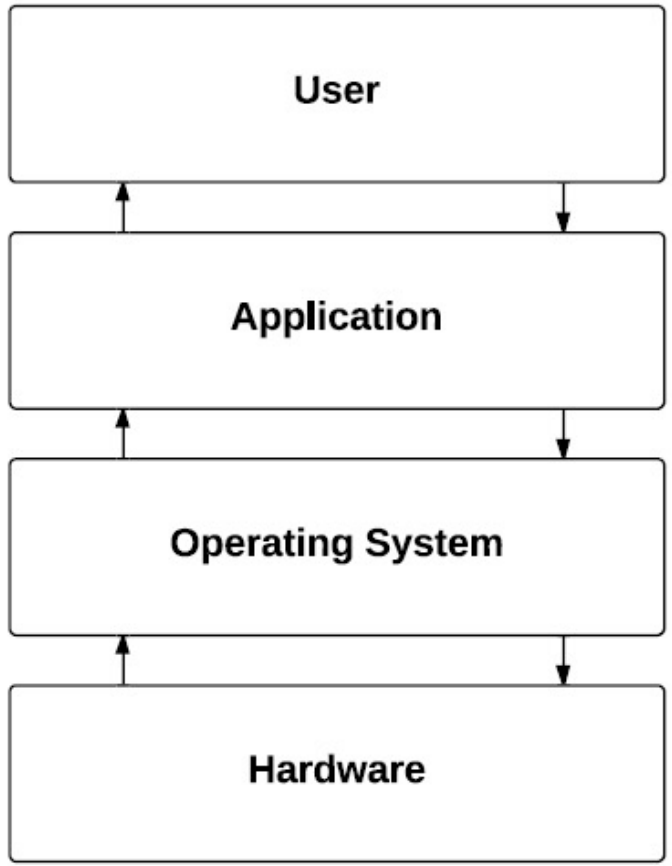
- No OS
  - Manually transferred programs by using switches
  - Simple interface like blinking lights on a panel
  - Sign-up sheets as a scheduling tool
  - Single user had to set up everything
- Then an operator
  - Programs written in punched card
  - Operator will handle the task submitted by users and return the result to them
  - Standard card libraries
- Early OS: batch system
  - Monitor, permanently resident in the memory
  - Handle a batch of jobs without user interaction
  - Problematic

How did the OS come into existence (2 of 2)

- Multiprogramming
  - Multiprogramming shortens the time that the CPU is blocked by I/O, thus, better utilisation
  - OSs have to be more sophisticated in terms of handling memory for various tasks and schedule the next task once the CPU is blocked by I/O
- Time sharing system
  - Users would like to interact with the computer as well
  - Interactive terminal access: time slices for users
- OS and Personal Computer

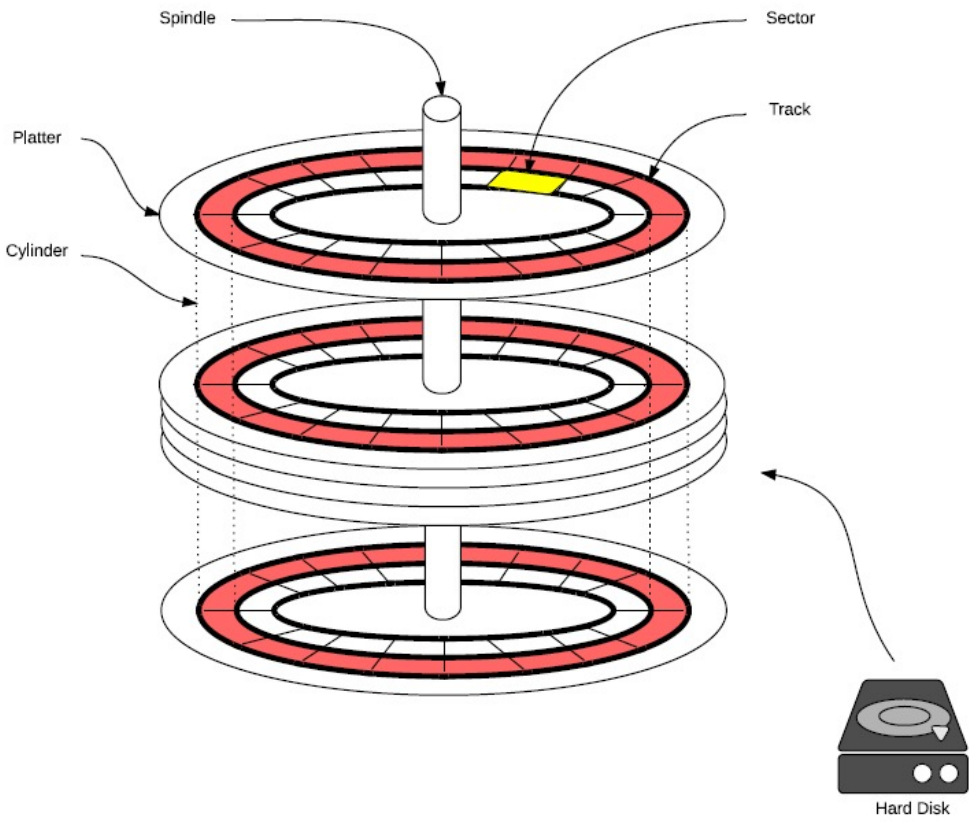
OS Does the Plumbing

- OS provides an abstraction of the messy hardware
  - Hardware is detailed and specific
  - Manipulating hardware requires not only programming knowledge, but also understanding of the hardware
  - User/Programmer does not have to care about the "pipes", simply turn on and off the "tap"
  - More productive



An Example: Writing to Disk (1 of 2)

- Data from memory to the device buffer
  - Reading/writing from/to a disk is usually slower than reading/writing from/to memory
  - *Read(DataMemAddr, Size, DeviceId);*
- Move the read/write head to the right area, identify the platter, track
  - How does a disk work?
  - *Locate(DeviceId, PlatterId, TrackId);*
- Finally write to the track cluster
  - *Write(DeviceId, Cluster);*



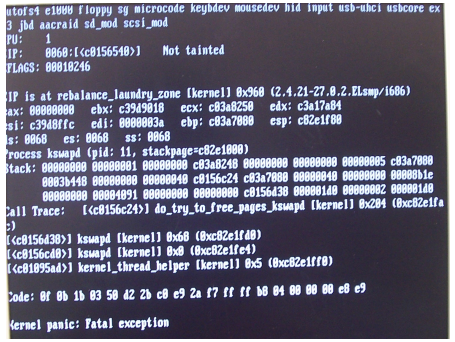
An Example: Writing to Disk (2 of 2)

- OS offers a simplified *Write* function that encapsulates the aforementioned commands
- Also, a logical address instead of the physical address is provided:
  - *Write(DataMemAddr, Size, DeviceId, LogAddr);*
- An even more abstracted function is common
  - Treat memory and the disk as the same file storage
  - A unified file identification *FieId*
  - And then use a library such as “C stdio”
  - Write an integer variable, *datum*, onto the disk at an implicit offset from the beginning of the file
  - *fprintf(FileId, “%d”, datum);*
  - “Everything is a file” philosophy (Unix)

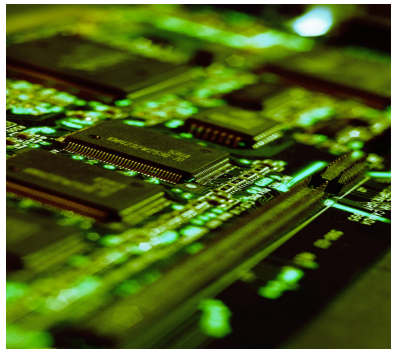
More examples

- User's view: copy, paste, send, save…
- Application developer's view: malloc(), fork(), open()…
- OS programmer's view: read-disk, start-printer, track-mouse…

User applications – service request –



OS – hardware instructions –

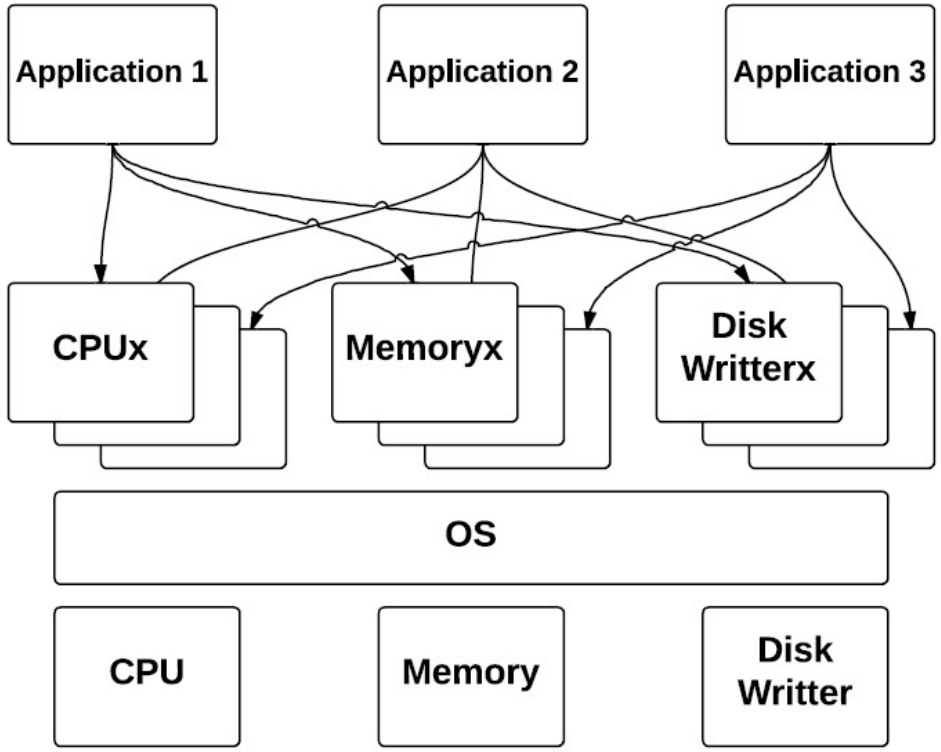


hardware

- "All problems in computer science can be solved by another level of abstraction"
- OS provides an abstract programming environment in which specific applications can be further developed to leverage the hardware

OS Resource Management

- OS also plays the role of resource manager
  - Process management
  - Memory management
- Always limited hardware resources
- How to leverage and optimise the use of hardware
- Time-multiplexing and space-multiplexing
- Control the execution of user applications and I/O devices to avoid errors and misuses



OS Components

OS Components

- Process management
  - How to run a program?
  - How to allocate resources?
- Memory management
  - Memory allocation
  - Protection
  - Virtual memory
- File systems
  - Secondary storage
- I/O
  - Device Drivers
- Network
- Security
- GUI

Process

- Operating system deals with various kind of activities or sub programs are called processes
  - User applications and system applications
  - Abstracted as process - all the execution context
  - An instance of a running program – possible to have multiple processes running the same program at the same time
  - Each process has independent memory space
  - Creation and destruction the processes
  - Schedules the processes depending on their deadline, priority, sequence etc
  - Maintain communication of processes
  - Maintain concurrency of processes

Memory

- How to organize processes' memory
  - Programs are stored in memory as well as data
  - Multiprogramming supports
  - Sharing data between processes
  - Pages
  - Virtual memory – use the secondary memory, RAM as a cache

Files and I/O

- File – an abstraction of a bulk of information
  - Secondary storage
  - Standard operations such as create, delete, copy and paste
  - Advanced, searching, backup and so on

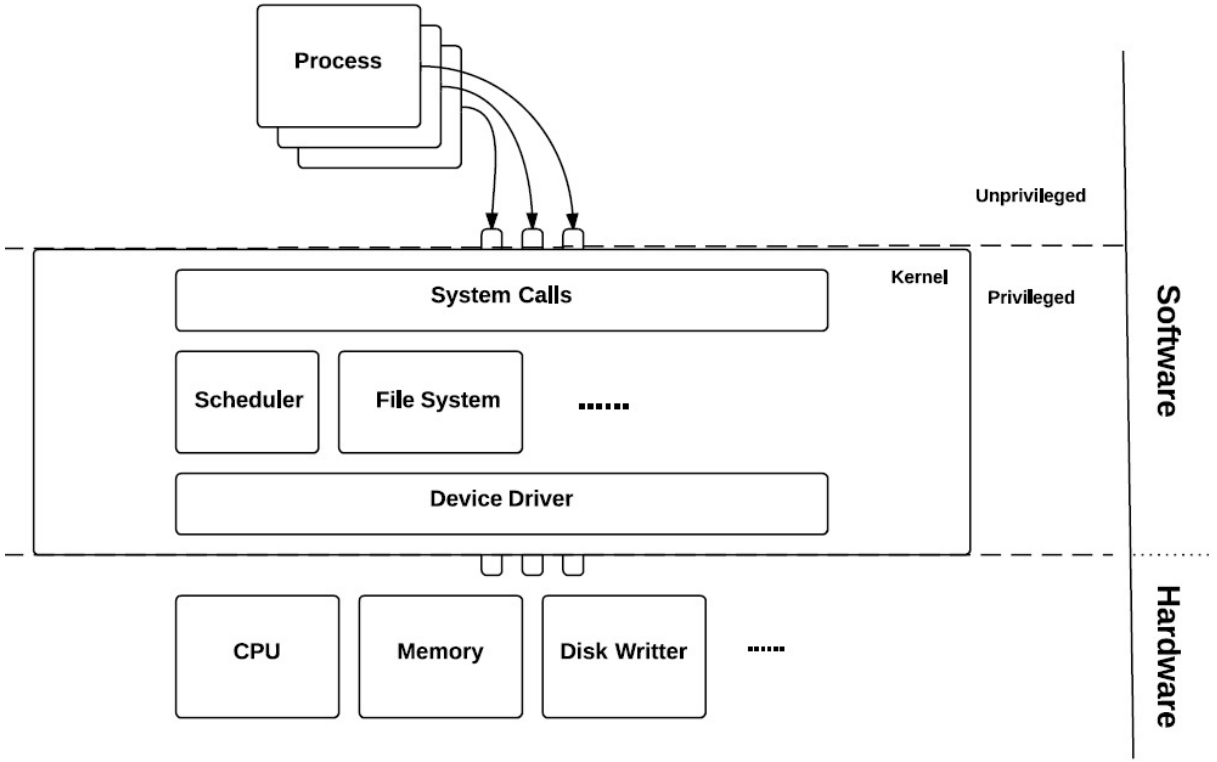
- I/O
  - As shown in the previous example
  - Requires device-specific knowledge
  - Device drivers and standard interface

Operating System Structure

- Different components interact with each other
- Not so straightforward as to how to organize all the components
- A challenging software engineering problem
  - Reliability
  - Backwards compatibility
  - Extensibility
  - Portability

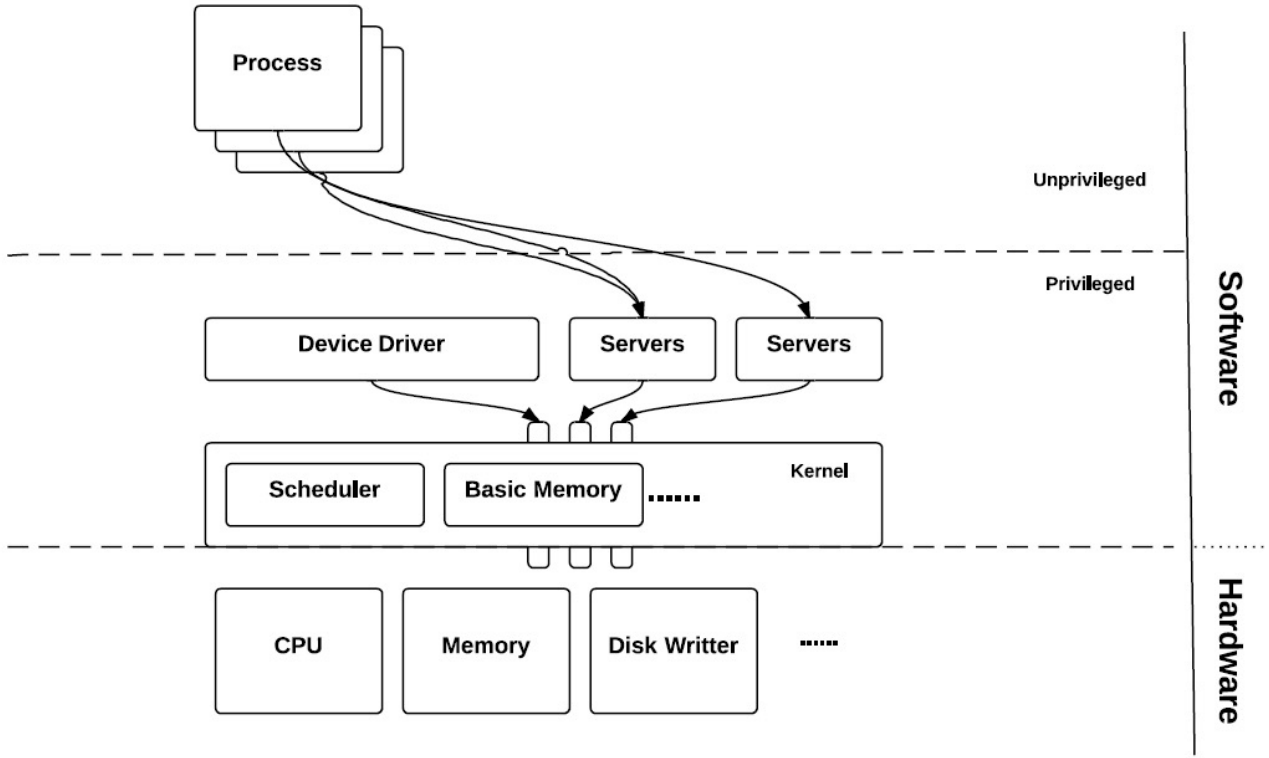
Monolithic

- Traditional OS structure
  - Single program includes all kernel code and offers all OS services
  - System calls
  - Fast and efficient
  - Less portable and difficult to maintain
  - Minor bugs can crash the entire system!
  - Unix and Unix-like OSs
  - Linux with loadable modules



Microkernel

- Minimal Kernel
  - Kernel design is simplified (privilege modes only)
  - User-space servers (may be privileged but usually unprivileged)
  - Rapid development, unit testing, easy maintenance
  - Huge memory footprint
  - Frequent context switching and inter-process communication
  - Not easy to implement



Operating System Structure

- Monolithic has better performance in general
- Microkernels have better modularity and extensibility
  - Price for switching between modes is high

- Modern OSs (most commercial) adopt a hybrid approach
  - The kernel is kept as small as possible
  - But most servers are in the privileged kernel space

  - Windows NT
  - XNU (OS X)

Types of Operating Systems (1 of 3)

- Network operating system
  - OS for computer networks
  - Allows and facilitates file sharing and hardware access
  - For local area networks (commonly seen in enterprise environments)
  - More features than the single computer OS: more communication
  - Routers OS (Cisco IOS)

  - Peer-to-peer
    - Master and slave network
    - Structure depends on the topology of the network

Types of Operating Systems (2 of 3)

- Distributed Operating Systems
  - Each node carries a “Horcrux ” – microkernel plus service components that coordinates with other nodes
  - Work collectively to fulfil all functions of an OS
  - Single node has full access to all system resources
    - Complicated scheduling and parallelism
    - User may not be aware of which specific node is executing the program or where the physical location of the file is – all automatically handled by the OS

Types of Operating Systems (3 of 3)

- RTOS
  - Real-time operating system dedicated to meeting specific timing constraints
  - Two types: hard real-time (ensures the critical tasks are to be completed on time) and soft real-time (if the deadline is not met, it is still worth finishing the task)
  - Industrial applications: robots, aircraft control …
  - Key design requirements:
    - Predictability and determinism
    - Speed is practically important. Usually achieved via a simplified OS design and sometimes traded off for predictability and determinism
    - Responsiveness and user control: do the right thing fast enough and priorities can be dynamically adjusted by users
    - Fail-safety: sometimes simply shutting down everything may not be a good option
  - Demands advanced scheduling and memory allocation

Embedded Operating Systems

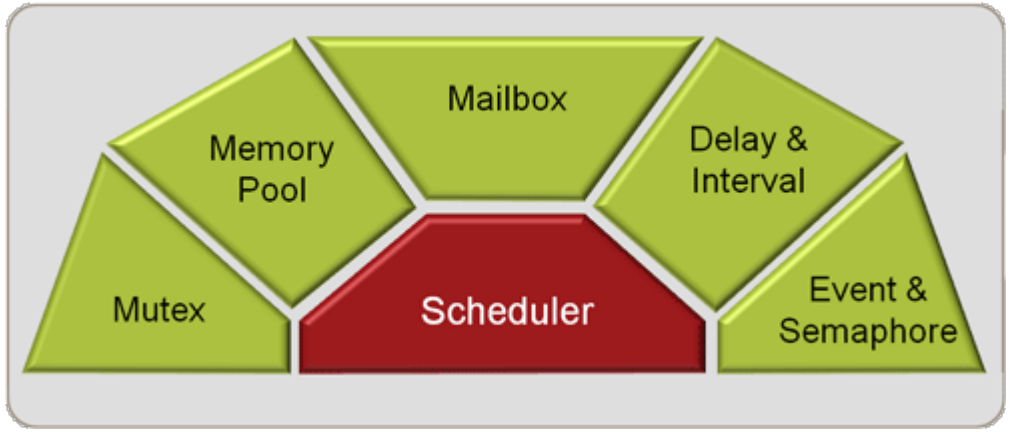
- RTOS and EOS are not exactly the same thing, but most EOSs are RTOSs, and aim at meeting the same timing constrains, therefore interchangeable in this course
- Predictability and determinism again
  - Major scheduling algorithms based on predicting the upper bound of the execution time
  - Interrupts
- “Real-Time”
  - Unified understanding of the deadline
  - Precise time services: TAI, UTC
- Fast, everything sits on the real-time kernel, even device driver
- What might not be important?
  - GUI?
  - Security? Depends on the application

RTOS on Embedded System vs. Super Loop

- Super-Loop is straightforward to implement and fits the computational model of ES
  - Depends on lengthy interrupt service routine (ISR)
  - Needs to keep the synchronization between ISRs
  - Poor predictability (nested ISRs) and extensibility
  - Change of the ISR or the Super-Loop ripples through entire system
- RTOS: all computation requests are encapsulated into tasks and scheduled based on the demand
  - Better program flow and event response
  - (Illusionary) multitasking
  - Concise ISRs thus deterministic
  - Better communication
  - Better resource management

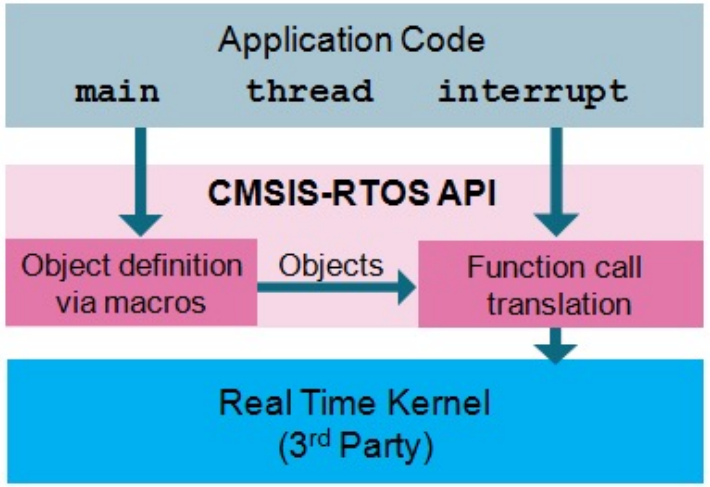
RTOS for this Course

- Keil RTX
  - Support ARM Cortex-M cores
  - Well-rounded RTOS for ES
  - Scheduler/ Mutex/ Event/ Semaphore/ Mailbox…



RTX Structure

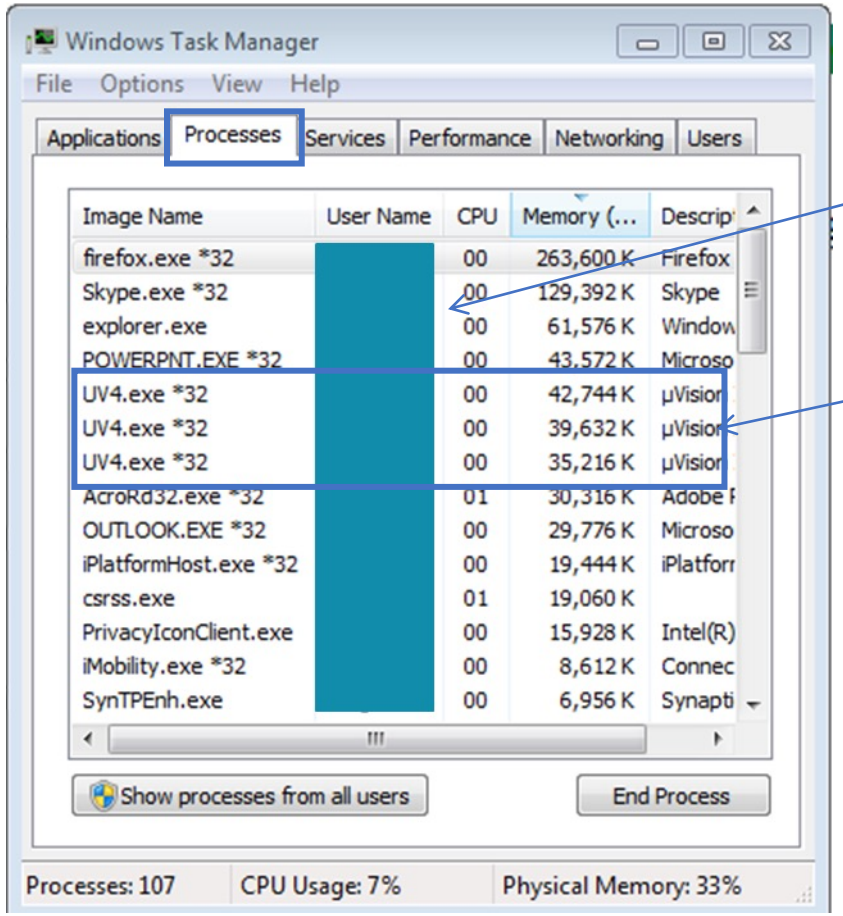
- CMSIS-RTOS API
  - Generic RTOS interface
  - CMSIS RTOS for ST is based on Keil RTX
  - Utilise some Cortex-M instructions



CMSIS-RTOS API Structure

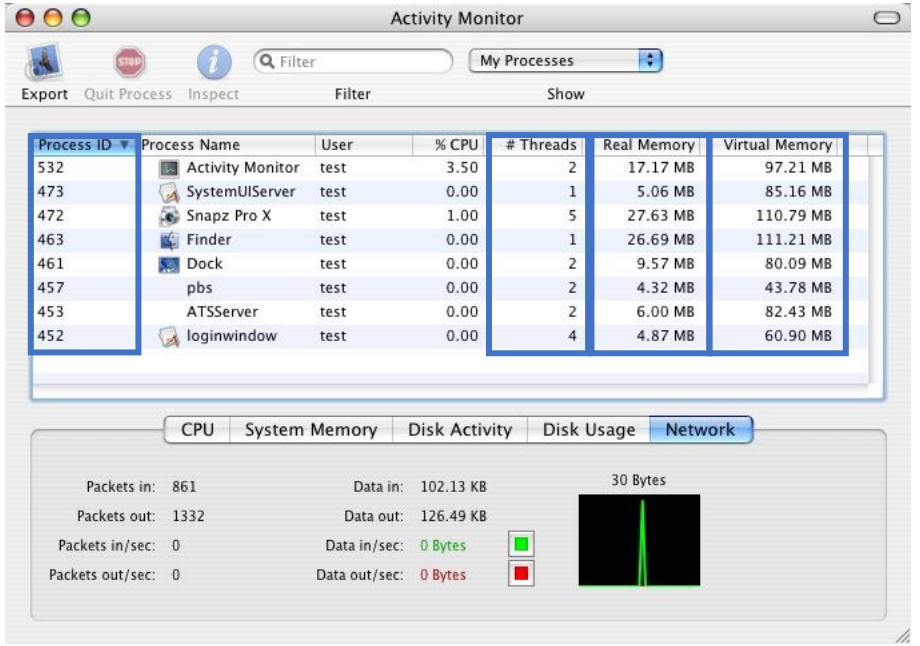
Processes

An Instance of a Running Program



Your user name or SYSTEM/LOCAL SERVICE/NETWORK SERVICE

Multiple instances of the µVision4
Independent memory for each process
"Memory(Private Working Set)"



Try to explore the task manager, activity monitor, or similar utility on your favourite OS

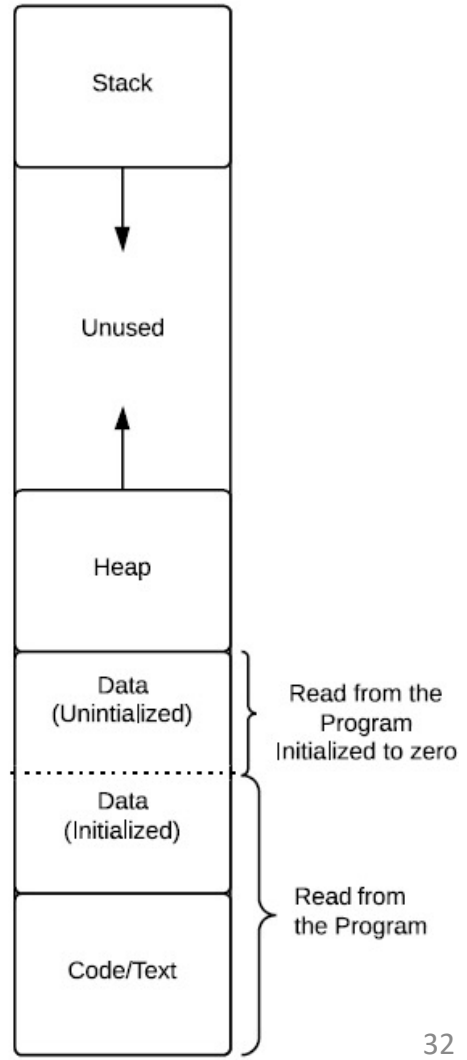
An Instance of a Running Program

- A process can have
  - A CPU time allocation (virtual CPU, the role of OS)
  - Memory (real or virtual)
  - Process ID
  - Threads

- Are they aware of the existence of other processes?
  - The operating system's role is to create an "illusion" that a process has all it needs to be executed
  - An abstraction of hardware resources
  - Each process sees one dedicated processor and one segment of memory (although they are often shared with others)
  - But they can be aware of each other – inter-process communication (IPC)

Memory Layout of an Executing Program

- Code or Text
  - Binary instructions to be executed
  - A clone of the program
  - Usually read-only
  - Program counter (PC), points to the next instruction
- Static Data
  - Global/Constant/Static variables - shared between threads
  - If not initialized by program, will be zero or null pointer
- Heap
  - malloc/free
- Stack
  - Used for procedure calls and return
  - Stack Pointer(SP)
    - FILO (First In, Last Out)



Process – The Abstraction

- Switching between executions means the operating system has to keep track of all the execution context
- Includes:
  - Memory State (code, data, heap and stack)
    - CPU state (PC, SP and other registers)
  - Also the OS state
- Hence the abstraction of the process
- Program usually refers to the instructions that are stored on disk
- Process is the program with execution context
- Some OSs may use the term “task”, particularly in an embedded system context. We will use both terms interchangeably for this course
- Thread: a lightweight process; a process may have multiple threads which share the same system resources - faster creation, termination, switching and communication

Process Control Block

- The Process Control Block (PCB) or Task Control Block (TCB) maintains all the relevant information for the process:
  - Process ID
  - Process state
  - PC, SP and other registers (stored)
  - Scheduling information (priority)
  - Memory management information
  - Accounting information
  - User information
    - Inter-Process Communication (IPC)
  - Other information
- The ID points to the entry in the process table where the pointer to the PCB is stored

Process Creation

- By initialization, by request of a process, or by request of a user
- Unique ID for the process
  - init_task_pid() in Linux
- Allocate memory for the PCB and other control structures (kernel) and user memory
- Initialize the PCB and memory management
- Link the PCB in the queue (see later)

Process Termination

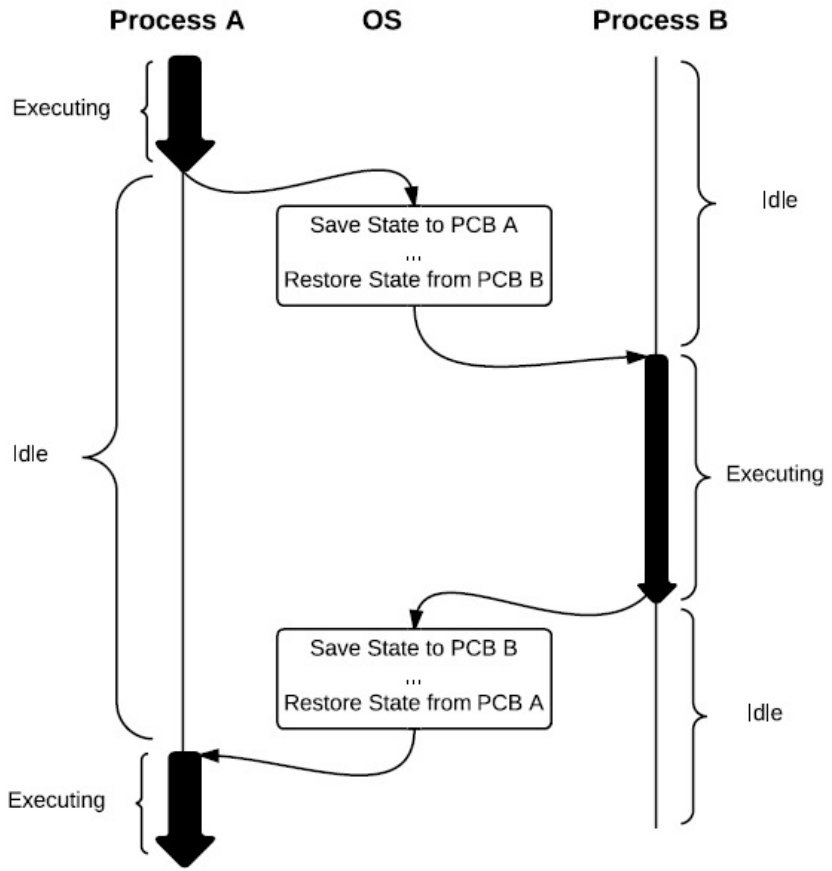
- Stopped by the OS/user (why?) or terminate itself

- Handle the output of the process
- Release the resources and reclaim the memory
- Unlink the PCB

- In embedded software, some processes may never terminate. Terminating implies a fault.

Context Switching

- The PCB makes context switching a bit easier
  - Scheduler will start or stop a process accordingly
  - Stores necessary information in the PCB to stop
    - Hardware registers
    - Program Counter
    - Memory states, stack and heap
    - State
  - Similarly, loads necessary information from the PCB
- Notice that context switching does consume time!
  - Could be up to several thousand CPU cycles
  - Overhead and bottleneck
  - Hardware support is also needed
- Multiprogramming, although only one active process at any given time

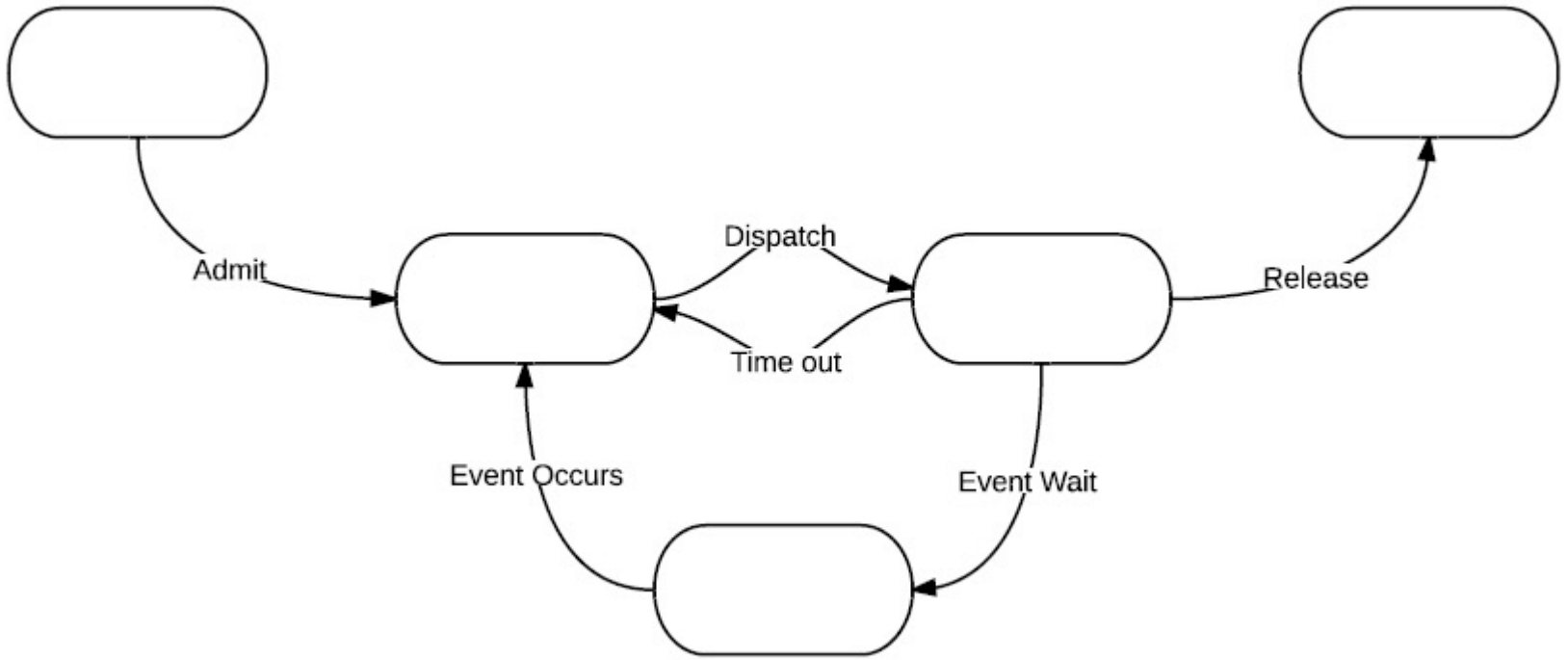


Process States

- Different process states during the lifetime cycle

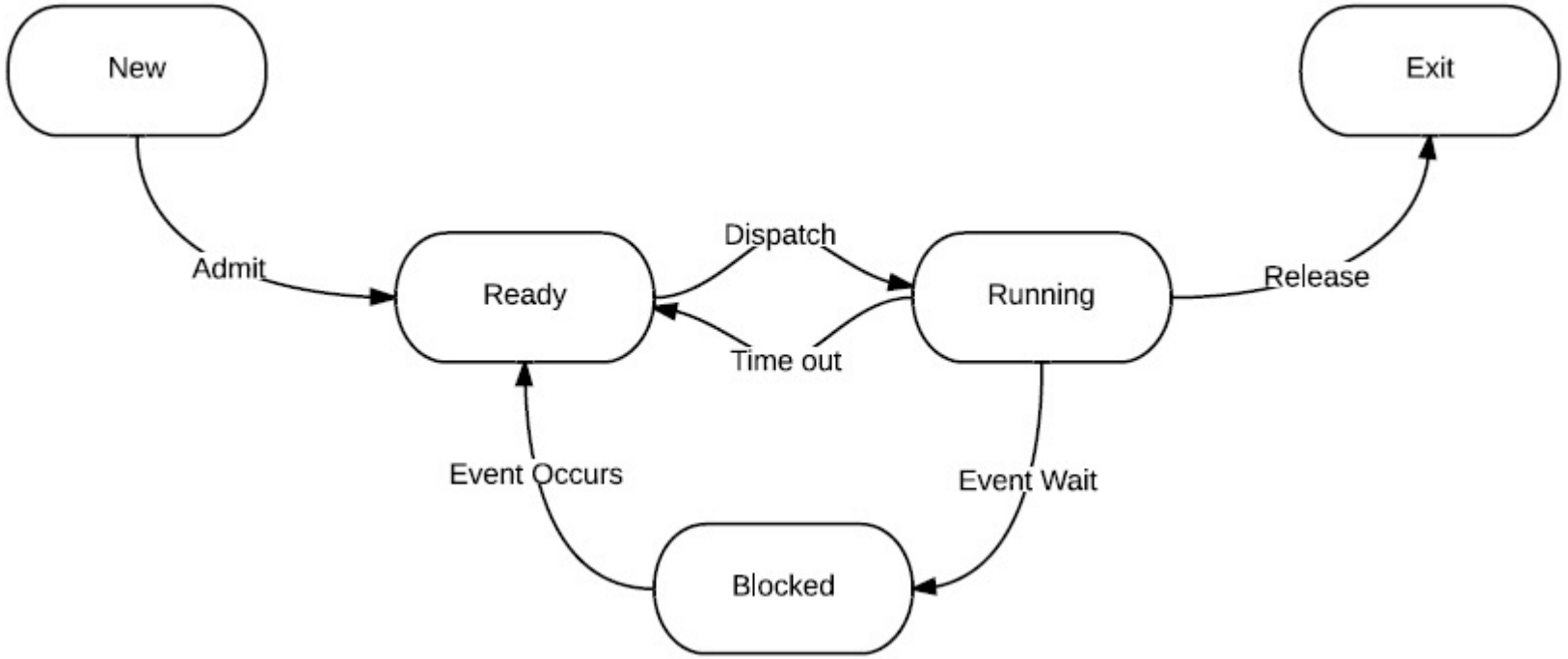
- Many variants but a standard model has five states:
  - New: just been created, not ready for the queue
  - Ready: can be loaded by the OS
  - Running: scheduler has picked this process from the queue and executed it, usually only one
  - Blocked: not in the queue, waiting
  - Exit: finished, needs to terminate

State Transition



- State transition as a result of OS scheduling, external interrupts or program requests
- Try to fill in the states in each block

State Transition



- Admit: process is fully loaded into memory and control is established
- Dispatch: scheduler assigns CPU to the process
- Time out: expired or preempted, pushed back to the queue
- Event Wait/Event Occurs: generally requests that cannot be met at the moment, has to wait until something occurs
  - OS not ready for a service
  - Unavailable resource
  - Wait for an input
- Release: release resources and end the process

Process State

- State information is also recorded by the PCB
- Context switch takes place whenever a process leaves/enters the running state
- Processes may make a transition voluntarily or involuntarily, e.g., end the program vs error
- OS typically maintains queue or queue-like (list) structures for processes in the same states (many pointers in the PCB)
  - RTX: rt_list.c
- More complicated models possible:
  - Some processes are stored in the secondary storage in their Ready or Blocked states
  - “Suspended” Ready and Blocked – the seven states model
  - To support swapping
  - Scheduler prefer those sit in main the memory