OS part of NOSE2: interested in how programs interact with OS and how OS interacts with the hardware on behalf of these systems (OS as an intermediary between user of computer and computer hardware

1. Processes

* Management and scheduling
* Synchronization

1. Memory

* Virtual memory
* Paging

1. Storage

* Block storage
* File systems

1. Sustainability considerations

Computer architecture: (key idea/what makes it special/what defines it – have both instructions and data memory to allow for flexibility)

* ALU – perform basic math and logic(and/or) operations
* Control unit controls ALU by deciding what operands are used and where the result should be written to
* Together they are known as the CPU (typically contains some cache memory that is faster and smaller than main memory)
* Cache memory can also be outside the CPU
* We use ISA (write programs) that tell the processor what to do
* CPU doesn’t understand ISA so we have to convert it to machine code using a complier
* Registers is a single memory cell that is part of the processor and data that CPU works on needs to be in the register
* Registers are the fastest memory, but it is small and limited in number
* There are instructions to transfer data between registers and main memory
* These registers are call general purpose registers
* Special register called the program counter that keeps track of where we are at in a long and large program with many sets of instructions
* These days we have multiple ALUs (parallel architectures) as single thread processors used to become fast and fast but in the last decade, they have unable to become faster so instead they become more parallel
* Parallelism also extends to a higher level where computers are connected to each other using high speed networks typically local (known as a cluster) to increase performance
* This leads to the idea of servers that we use remotely through a network connection

OS:

* Program will not need to be recompiled/rewritten just because another network device is attached to your computer (this is something OS will do)
* OS acts as a control program that user applications can use to do anything with a computer
* Extracts details of the specific computer hardware by providing general API that programs can you regardless of the network interface attached
* Make it easier for user applications to make use of resources and isolating multiple programs against one another (user applications are written against an operating system where there are different versions available for windows/mac as they have different OS)
* Another main function is resource management (control and manage computer hardware and resources for the most efficient use)
* Core components of the OS is the kernel and bootstrap program (BIOS)
* Kernal is the part of the OS that is always running and manages other programs (both system and user programs)
* Basically, the OS is the kernel
* The BIOS starts the OS by starting its kernel
* It’s considered a firmware as it is somewhere between hardware and software
* This is because it is pre-installed in a read only memory part of the processor, so it is not easy to change
* Storage systems are normally organized in hierarchy where the characteristic is that as we move further away from the processor, things become slower but higher in capacity
* This also gives rise to implications for the durability of these memory
* Caching is storing a copy of data so that the next time you need this data you have faster access
* Copies data from a slower memory into a faster memory

Processes management: how do we create and kill processors, how processes have multiple states, how do we make multiple states work together and how the memory is shared (memory management)

System calls:

* The first main function of the OS (abstractions) is done using system calls
* System calls the principal mechanism for interacting with an OS from a user program
* System call instruction will tell OS that a program needs its help to access resources
* Instruction will give control to the kernel so OS can provide access
* Usually is not done directly by the instructions (maybe through higher level API or these OS have been implemented in high level system programming languages)
* User applications run on user mode and the kernel runs on kernel mode
* Difference between the two modes is the level of privilege
* User mode enters prepping where they stop execution and provide control over to the OS kernel
* E.g. instructions that require loading data from memory, OS needs to be in control for a while to check if the memory location belongs to the process trying to load the memory (e.g. one application should not be able to read the state of all other applications running on your computer)
* We can open a specific file through passing parameters
* We can do this by writing the parameter through a specific register
* Alternative is writing the memory location address (sometimes the parameter is too big to pass through the register state)
* An alternative to this uses the processes memory (stack memory)
* System calls can be different on different OS

Processes: (program control and execution)

* Process is a program that is currently in running
* It is equipped with some resources (it is not just static code on some media, it is a running program)
* Create a process by starting a program by loading it into memory
* This is when the OS starts providing the required resources for the process to execute
* All processes will need some memory from the OS (own memory address space)
* multiple processes can be executed simultaneously, or one program started multiple times or a single application consist of multiple processes (multi-process)
* Processes can be in multiple states in its lifetime
* When the process is started it is new and eventually when it has gotten all the resources it needs it becomes ready to run
* Once it has been scheduled and dispatched it will be running on the CPU
* This can be interrupted which the causes the process to be ready again
* These interrupts are normally handled quickly for the system not to crash so it has priority over all processes
* After interrupts are handle it can run again after it is scheduled and dispatched
* Running state can also wait for IO to be done (html will wait for data to be fully received) before it is ready again
* After all instruction are ran it will enter a terminated state
* State of the process is kept in a particular location in memory where all processes state is managed (operation system process table)
* In this table we have table entries (process table entry) that stores the process state, id and program counter (can have multiple program counters if there are multiple threads in a process)
* We have a state in memory for all the registers (these registers are not in the CPU)
* Whenever we interrupt a process, we need to store away register values so we can continue computation before it interrupted
* Memory limits is where in the entire memory is the memory for this process
* New processes can be created from existing processes using system calls
* We can execute multiple processes either independently or have parent processors wait for their children processes to finish
* There are different resource sharing options where resources can be fully shared parts or a subset can be shared or a the process can be fully independent (good to know)
* Timestamp 31.00 topic 10 for AE2

Inter-process communication: communication between processes

* If you have multiple processes, they can either be independent or cooperating
* The advantages of cooperating processes the information sharing, speed up programs, modularity, convenience
* Programs can be sped up by breaking down large programs into multiple smaller ones and have them be worked out in parallel
* Modularity also breaks up large programs into independent processes for isolation to have information exchanged only through defined interfaces but also debundle the lifetimes a little (if some part of the program with one process fails the entire process will crash even with multiple threads so this prevents it)
* Convenience is when we are building a different application but need the same or similar function, we can just use the program that already exists
* There are two methods for inter-process communication, shared memory communication and message passing
* Shared memory communication is when multiple processes have access to the same part of memory so they can communicate by writing and reading the same memory locations
* Reading and writing from the same memory location requires synchronisation
* Message passing is when a message is sent from one process to another and delivered through the OS kernel memory/another machine
* Message passing requires communication links provided by the OS
* Two basic operations (sending and receiving)
* Subtle differences in the ways of communication such as direct or indirect, synchronous or asynchronous and automatic or explicit buffering
* Direct communication at least one of the processes is names (normally receiving end) where indirect has something in between the sender and receiver (follows the concept of a mailbox) so it only needs to know the mailbox not the sender and receiver (multiple mailboxes can be used)
* Synchronise message passing can be done on both ends using blocking (block send/receive) and asynchronies is when there is non-blocking (does not wait if message sent is successful)
* When something is used between sender and receiver to store the message(e.g. mailbox) we typically use buffering
* Buffering size matters
* If it is 0 there is no buffer, so it needs to be direct communication
* If it is bounded capacity, buffer with a certain size (normally preconfigured but can be dynamic) resources need to be managed carefully (sender must wait when link full)
* Unbounded capacity is a theory as real life is always bounded (sender never waits)
* Both methods are time sensitive as you first need to write memory before another process can read it

Process scheduling: when process state moves to running state we often need to select among multiple processes

* Process scheduling wants to achieve 2 goals which is multiprogramming and time-sharing
* Multiprogramming is when one process is running at all times
* Time-sharing is switching among processes to give all of them the chance to react to new input data again and again and again
* Any useful process will also itself switch between waiting and running state (switch between processing and IO)
* Running through instructions is called CPU burst and IO is IO burst
* Time stamp 6.40 for AE2
* Before going from IO burst to CPU burst, process scheduling must occur (algorithms are used for selection)
* Processes can either be CPU bound, which is the time it takes for the process to complete is predominantly determined by the CPU speed, or IO bound which is vice versa
* there are different types of schedulers such as long term scheduler which loads programs into memory (job scheduler) and it strives for a good mix of IO-bound and CPU bound processes
* Medium term schedulers which freezes and unloads (swaps out) a process to be reintroduced (swaped in) at a later stage
* Main focus is CPU scheduling or short-term schedulaer where it selects which process to execute next
* Main goal of CPU scheduling is the utilize the CPU well which means we want to schedule the process as quickly as possible of a process that is ready and run the instructions on our CPU
* CPU schedulers make use of multiple waiting queues for processes
* Job queue (all processes in the system)
* Ready queue (processes in RAM and ready to execute
* Device queues (processes waiting on some device)
* Processes migrate between these queues when they change their state
* 12.30 AE2
* The CPU scheduler make decisions when running process requires data from disk/network before it can continue so it is blocked before the CPU can go to another process
* Another time scheduler will make decision is when handling interrupts (has the highest priority)
* Another time is when a set of waiting processes is ready (depends on the priority of the processes)
* And last is when the process terminates so there is space for another process
* Two types of scheduling methods, preemptive and not preemptive scheduling
* Preemptive scheduling might pause/haut running processes to run another process
* In non preemptive, you cannot stop unless there is an interrupt as they always have priority (no interrupts in exercise so for the sake of the course we just say that non preemptive always run until they are finished)
* Dispatcher task is to implement the scheduling decisions
* Puts a process on the CPU by loading that processes registers, switching to the correct system mode and jumping back to where we previously left it
* What the dispatcher does here is called context switching that takes a moments (pure overhead) as during context switch, we are not using CPU just preparing it
* Criteria for comparing CPU schedulers depends on what goal we have (for all processes not just single one)

Process scheduling algorithms

* FCFS/FIFO uses a queue that is kept in order of arrival time of processes (executes first in the queue) so it is simple but the average waiting time is quite high (non-preemptive)

Memory management: what memory is currently used, which processors are using what memory

Storage management: how are files organized, how files are created editing and deleted, directories and backup