Recall:

* OS is:
  + Referee – manage, isolation, security of shared resources
  + Glue – provide common set of services such as storage, window system, etc
  + Illusionist – Clean, easy to use software abstractions that make machine seem more powerful than it really is to the software

Recall OS Protection:

* Each process thinks it has hardware resources to itself
* Each process has access to software abstractions of hardware (threads, sockets, files, etc)
* Each process can’t modify memory, storage of another process
* Many processes on a single machine

World of hardware is very complex (High speed I/O, memory channels, disks, Integrated Ethernet, RAID drives, etc) 🡪 Need software abstractions to make writing programs easier

Complexity leaks into OS if not properly designed:

* Third-party device drivers are one of the most unreliable aspects of OS
  + Poorly written by non-stake holders
  + Ironically, the attempts to provide clean abstractions can lead to crashes
* Version skew on libraries can lead to problems with application execution
* Holes in security models or bugs in OS lead to instability and privacy breaches

Application

------------------------------------------------------ (Abstract Machine Interface)

OS

------------------------------------------------------ (Physical Machine Interface)

Hardware

OS abstracts underlying hardware to tame complexity

* Processor -> Thread
* Memory -> Address Space
* Disks, SSD.. -> Files
* Networks -> Sockets
* Machines -> Processes

OS as an illusionist:

* Remove hardware/software quirks (fight complexity)
* Optimize for convenience, utilization, reliability.. (help the programmer)

For any OS area:

* What hardware interface to handle? (physical reality)
* What software interface to provide? (nicer abstraction)

Four Fundamental OS Concepts:

* Thread: Execution context
* Address Space
* Process: an instance of a program
* Dual mode operation/protection

OS bottom line: Run programs

* Write them and compile them
* Load instructions and data segments of executable file into memory
* Create stack and heap
* “Transfer control to program”
* Provide services to program
* While protecting OS and programs

**Thread** – Single unique execution context:

* Program counter, registers, execution flags, stack, memory state
  + Thread is **executing** on a processor if its **resident** in the process registers
  + Resident means: Registers hold root state (context) of the thread:
    - Including program counter PC register and currently executing instruction
      * PC points at next instruction in memory
      * Instructions stored in memory
    - Including intermediate values for ongoing computations
      * Can include actual values (int, char, etc) or pointes to values in memory
    - Stack pointer holds address of top of stack (stack is in memory)
    - Rest is “in memory”
  + A thread is suspended (not executing) when its state is not loaded (resident) into the processor
    - Processor state is pointing at some other threads
    - Program counter register *is not* pointing at next instruction from this thread
    - Often: a copy of the last value for each register is stored in memory
  + Execution sequence:
    - Fetch instructions at PC
    - Decode
    - Execute
    - Write results to registers/mem
    - PC = Next Instruction (PC)
    - **REPEAT**
* Illusion of multiple processors:
  + How do we provide illusion of multiple processors?
    - Multiplex in time?
  + Threads are virtual cores
  + Contents of virtual core (thread)
    - Program counter, stack pointer
    - Registers
  + Where is it thread?
    - On real physical core, or
    - **A picture containing text, screenshot, font, diagram

      Description automatically generated**Save in chunk of memory 🡪 **Thread control block (TCB)**

What triggered this switch?

* Timer, voluntary yield, I/O, other things we will discuss

Thread Control Block (TCB)

* Holds contents of registers when thread not running
* TCB’s are stored in kernel (memory)
* PINTOS? (read thread.h and thread.c)

**Address Space** – set of accessible addresses + state associated with them

* For 32-bit processor: 2\*\*32 – 4 billion addresses
* For 64 bit processor: 2\*\*64 = 18 quadrillion addresses

What happens when you read of write to an address?

* Perhaps acts like regular memory
* Perhaps ignored writes
* Perhaps causes I/O operations
* Perhaps causes exception (fault)
* A picture containing text, screenshot, line, diagram

  Description automatically generatedCommunicates with another program
* Static data: contains static variables, global data, string constants
* Code segment: code
* Heap: dynamically allocated memory (malloc)
* Stack: local variables

All vCPU’s share non-CPU resources

* Memory. I/O devices
* OS job to make virtualization as true as possible
* If each thread can read/write memory
  + Perhaps data of others
  + Can overwrite OS?
  + Is this unusable?
  + Ideally, you would want if a program crashes, it doesn’t crash the entire system
* Simple multiplexing has NO PROTECTION!
* OS must protect itself from user programs
  + Reliability -> Compromising OS can cause it to crash
  + Security -> limit scope of what threads can do
  + Privacy-> limit each thread to the data it is permitted to access
  + Fairness-> each thread should be limited to its appropriate share of system resources
* OS must protect User Programs from one another
  + Prevent threads from one user from impacting threads owned by another user
* Simple protection: Base and Bound
  + Two registers-> base and bound (OS decides base, bound)
  + Specify what address space some thread can access (Program address must be less than bound and greater equal base)
  + Growing this space would copying code, static data, heap, etc -> Fragmentation issues

A picture containing text, diagram, screenshot, line

Description automatically generated

* Address Space translation:
  + Program operates in an address space that is distinct from the physical memory space on the machine
  + Processor -> (“virtual address”) -> Translator -> (“physical address”) -> Memory
* Paged virtual address space
  + What if break the entire virtual address space into equal size chunks with base for each
  + All pages the same size, so easy to place each page in memory
  + Hardware translates address using **page table**
    - Each page has separate base, “bound” is page size, special hardware registers store pointer to page table, treat memory as page size frames, put page into any frame
  + Instructions operate on virtual addresses
  + Translated to a physical address through a Page Table

**Process –** execution environment with Restricted Rights (+one or more threads)

* Protected address space w/ one or more threads
* Owns memory
* Owns file descriptors, file system context
* Encapsulate one or more threads
* Application program executes as a process
  + Complex applications can fork/exec child processes
* Why processes?
  + Protected from each other!
  + OS protected from them!
  + Processes provide memory protections
  + Fundamental tradeoff between protection and efficiency
    - Communication **easier** between threads**, slower** between processes
* **Threads encapsulate concurrency (**parallelism -> performance, concurrency -> handling I/O and other events)
* A picture containing text, screenshot, diagram, font

  Description automatically generated**Address spaces encapsulate protection**
* Why do we need processes?
  + Reliability: bugs can only overwrite memory of process they are in
  + Security, privacy: malicious compromised process can’t read or write other process’ data
  + Fairness: enforces shares of disk, CPU

Address translation: address space only contains its own data

Why can’t process change the page table pointer?

Hardware must support **privilege levels**

**Dual Mode Operation:**

* Kernel Mode
* User Mode
* Certain operations are prohibited when running in user mode
  + Changing the page table pointer, disabling interrupts, interacting directly w/ hardware, writing to kernel memory
  + Carefully controlled transitions between user mode and kernel mode
    - A picture containing text, screenshot, font, number

      Description automatically generatedSystem calls, interrupts, exceptions

3 types of switch User -> Kernel Mode Transfer

(ALL ARE UNOPROGRAMMED CONTROL TRANSFER)

* Syscall
  + Process request, like a function call but “outside” process, like a RPC
  + Marshall the syscall id and args in registers and exec syscall
* Interrupt
  + External asynchronous even triggers context switch (Timer, I/O device)
  + Independent of user process
* Trap or Execution
  + Internal asynchronous event in process triggers context switch
  + Seg fault (protection violation), divide by zero

We have basic mechanisms to:

* Switch between user processes and kernel
* Kernel can switch among user processes
* Protect OS from user processes and processes from each other