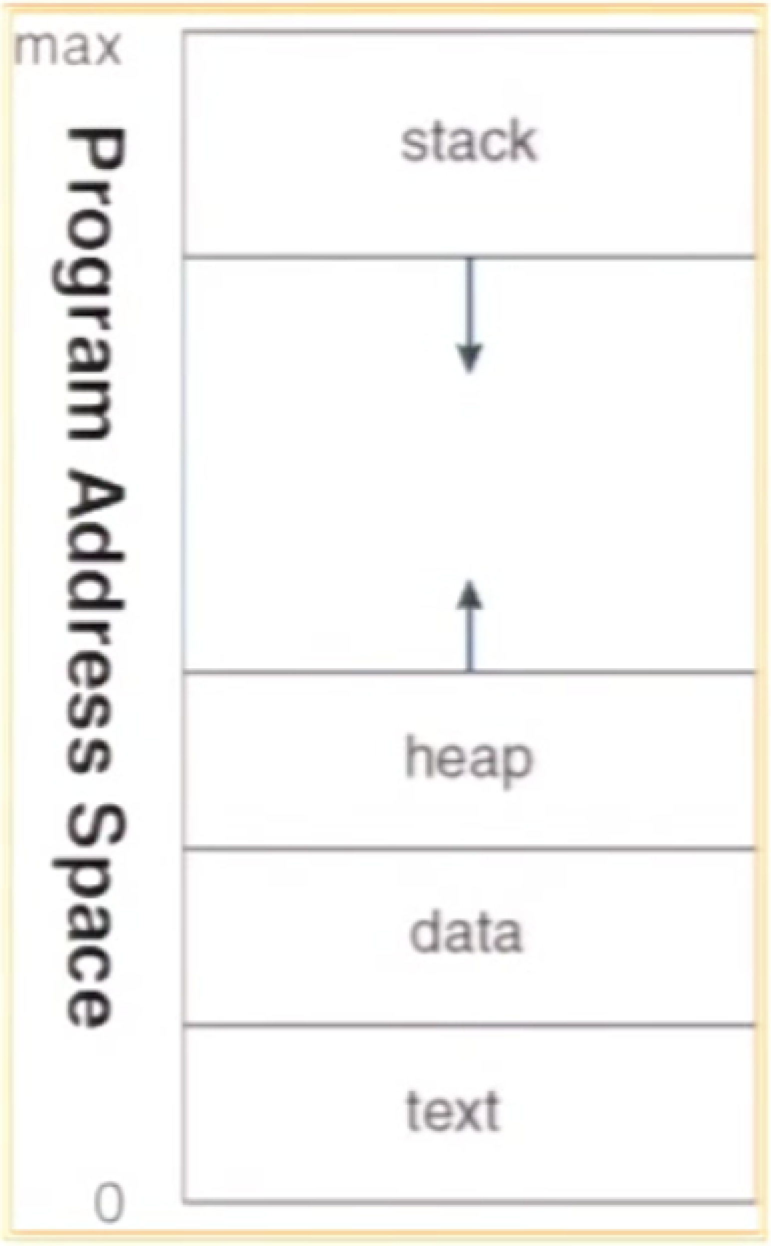
threads & processes

- thread
  - unit of execution. has its own fetch/decode/execute loop
  - unit of scheduling
  - has its own stack and registers
  - operating in some address space, possibly shared with other threads.
- uniprogramming vs multiprogramming:
  - uniprogramming means only one thread at a time! (and just 1 process)
  - multiprogramming means multiple threads & processes can run at the same time, in the same or different process. All OSes today are multiprogramming, but this wasn't always the case.
  - multiprogramming world → need to think about concurrency, preemption, respecting resources etc
- a process is the unit of resource allocation and execution
  - owns some memory address space
  - owns file descriptors
  - encapsulates threads sharing the process's resources
- why process vs thread:
  - processes provide more protection
  - threads can share memory and file descriptors
  - threads can access the data of every other thread. also threads can run the instructions of other threads (by starting a new thread!)
  - this concept of process vs thread navigates the fundamental tradeoff of protection vs efficiency
- application instance consists of one or more processes
- physical core vs virtual core. simultaneous multithreading or as intel calls it, hyperthreading
  - A single physical CPU core with hyper-threading appears as two logical CPUs to an operating system. The CPU is still a single CPU, so it's a little bit of a cheat. While the operating system sees two CPUs for each core, the actual CPU hardware only has a single set of execution resources for each core. The CPU pretends it has more cores than it does, and it uses its own logic to speed up program execution. In other words, *the operating system is tricked into seeing two CPUs for each actual CPU core*. it can schedule as though it is 2 CPUs
- multiplexing is the idea that you give time slices to different processes so that it seems like they are running together
- each thread's stack pointer, program counter and registers need to be saved and reloaded for multiplexing. such saving and reloading and scheduling logic is overhead of context switching. SMT/hyperthreading optimizes some of this overhead by asking the OS not to worry about it.
- the address space of the process is not the true address space → there is a layer of virtualization called virtual memory that makes it logical address vs physical address.
- states:
  - – new: The process is being created
  - – ready: The process is waiting to run
  - – running: Instructions are being executed
  - – waiting: Process waiting for some event to occur
  - – terminated: The process has finished execution

- program vs process
  - a process is an instance of a program
  - a program can invoke more than one process
- in virtual address space, stack and heap grow from opposite ends



- when we start a process, we launch a process with one thread. those threads may then launch more threads within that process.
- process control block is what encapsulates the process state.
- processes don't by default share any memory, so what if two processes need to communicate? shared memory (memory mapped to same physical memory) and message passing (`send()` and `receive()`) are the two ways.
  - shared memory is efficient but needs some synchronization to happen.
  - message passing can be either local or work across the network, using the networking stack and things like sockets
- a thread is sometimes, loosely, called a "lightweight process"

- if you care about efficiency and you control the whole system (like self driving cars or databases) you use threads. if you care about trust, protection and don't know what's running, you'll use processes. so next time when deciding between thread and processes for concurrency, efficiency vs trust/protection is a key tradeoff.
- difference between context switching vs two threads in the same process vs two different processes
  - context switching involves switching program counter, stack pointer, registers, memory meta-data and I/O state, when switching between processes.
  - when switching between threads, we don't switch memory metadata and I/O state since that is shared so its a bit faster → really not a lot faster though. 3 - 4 microseconds is a context switch. switching threads is on the order of 100 nanoseconds faster than switching processes. BUT the cost of cache misses from a different working set of another process can be high. see 3rd point below for more on this.
  - context switching between different cores is about 2x more expensive though! this is because a context switch between cores results in L1 cache misses.
  - context switching time also depends on the size of the working set (the part of the memory that has been in cache recently). if the other process blows this out of the cache, we get screwed when the first process is scheduled again → a bunch of cache misses! so a bigger working set is usually worse. again, threads can be written to use the same large working set, so they don't keep kicking out the other thread's stuff out of the cache.
  - by a similar argument, starting a process is easier than starting a thread
  - Moral: the cost of context switching mostly depends on cache issues, be it across cores or a change of working set on same core.
- Also, creating processes and creating threads are different overheads even though context switching between threads vs processes may not differ a lot. this is because we have to set aside some mapped memory etc for a new process. that's why for web servers, databases etc its better to multithread rather than multiprocess in addition to shared memory, file handlers etc.
- the above is talking about context switching by the OS. hyperthreaded context switches (those initiated by the processor) are much faster since a hyptherthreaded processor has two copies of registers etc and doesn't need to make copies each time it switches. the downside though is that if the other thread is waiting for I/O or something and is not ready to run, we still have an idle core until we switch out registers and schedule a totally different process to run on it.
- we should implement scheduling ready queues in the OS so that we can schedule as quickly as possible, instead of $O(n)$. maybe just pick the head of the list. if we have a large number of threads, $O(n)$ scheduling will really hurt us.
- how does a thread get switched out?
  - the act of requesting I/O implicitly gets it switched out waiting for I/O
  - waiting on a "signal" from another thread. `signal()` is a inter-thread communication functionality provided by the OS.
  - thread executes a `yield()` → this also affects its priority in scheduling
- how do interrupts work?
  - these are physical, electrical events
  - triggered by things like I/O being complete, data becoming available and so on
  - when a interrupt occurs, it calls an interrupt handler, that is registered at a particular address in the interrupt table → this hardware mechanism kind of is like a system call.
  - interrupts are priority aware. if a high priority interrupt is running, it can say, don't interrupt with me a lower priority interrupt.

- multiprogramming can be preemptive or nonpreemptive
    - preemptive means the OS preempts the thread. non-preemptive means the code is well behaved and keeps calling yield() now and then → those days are long gone, with today's app ecosystem and modern OSes we need preemptive approach.
- thread pools is a technique to limit the threads to prevent thrashing