First Come First Serve (FCFS)
- Pros: low overhead, no starvation, no priority inversion, high throughput (minimize context switching)
- Cons: sensitive to arrival order (poor predictability), Convoy Effect (bad for interactive tasks), bad at minimizing avg response time.
- Convoy effect: short process stuck behind long process. Non-preemptible policy

Shortest Job First (SJF)
- Pros: optimal avg completion/response time when arrive simultaneously for non-preemptive.
- Cons: starvation, convoy (sensitive to arrival order), need to know job duration.

Shortest Remaining Time First (SRTF)
- Pros: Preemption, so no convoy, optimal avg completion/response time
- Cons: starvation, requires knowledge about job duration.
- It's possible to build a scheduler that approximates SRTF using a moving avg. SRTF requires a way to predict future burst time.

Round Robin (RR)
- Increase time slice: avg completion down, avg response up
- Large q → FCFS, small q→ lots of interleaving/context switching
- Pros: No starvation / bounded wait time (no process waits more than (n-1)q time units). No convoy (only run a time slice at a time),
- Cons: completion time high, overhead of context switching (q must be large with respect to context switch) → low throughput, more cache misses bc of context switch
- Round-robin with Quantum from 10 ticks to 20 ticks: the system will preempt and context switch about half as many times. The total time to run all jobs will be less with a higher quantum, implying a higher throughput.
- RR is the fairest scheduler with regards to wait time for CPU (each task waits the same amount of time)

Strict Priority Scheduling (SPS)- Split by priority into n different queues, each run RR

Multilevel Feedback Queue (MLFQ)
-Remain on the highest priority queue in MLFQ: 1) every quanta-1 ticks do a IO operation; 2) manually yield the thread to CPU; 3) put the thread to sleep

Lottery Scheduling
- No priority inversion, no starvation, might choose high priority first
- Approximate SRTF by giving more tickets to shorter jobs
- Prevent starvation among low-priority tasks: each thread/process is given a certain number of lottery tickets based on its priority. We pseudo-randomly select a lottery ticket and schedule the thread that holds that lottery ticket. Since every thread gets at least one lottery ticket, we know every thread will be scheduled eventually.

Stride Scheduling
- Deterministic lottery scheduler. Stride for each job = big#w / Ni. stride is inversely proportional to number of tickets
- Pick the lowest pass, run it, add its stride to its pass. Low stride jobs (more tickets) run more often.

Linux Completely Fair Scheduler (CFS)
- Equal share of CPU. avg rate of execution = 1/N. share virtual runtime equally. Optimize globally, not locally. Choose thread with min CPU time
- Scheduling cost = O(logn), choose next thread O(1), red-black tree
- Target Latency (responsiveness): period of time over which every process gets service. Quanta = target_latency / n
- Minimum Granularity (throughput): min length of any time slice.
- Weighted shares allow different threads to have different rates of execution. Qi=(wi/sum(wp)) * target_latency
- Nice values to scale weights exponentially (rather than for priority). Weight = 1024/(1.25)^nice. Higher nice, lower priority.
- Higher weight: vr increases more slowly. Other way for lower weight.
- Fairness: schedules threads so that their virtual runtimes increase together and remain approximately equal.
- Timeliness: maintains target latency, adjusting the time quantum ac- cording to the number of threads so that all threads are serviced within the target latency.
- Throughput: ensures that the time quantum does not fall below the minimum granularity, to prevent context switch overhead from degrading throughput.

Deadlock
- Requirements: 1) mutual exclusion and bounded resources; 2) hold and wait; 3) no preemption; 4) circular wait *need all 4 to deadlock
- External intervention: kill thread; remove resource from thread
- possible to have a deadlock when there's only one single threaded process. Acquiring a non-reentrant lock twice in a row before releasing it will result in deadlock.

Banker's Algorithm - avoidance
```
[Avail] = [FreeResources]
Add all threads to UNFINISHED
do {  done = true
      Foreach thread in UNFINISHED:
            if ([Max threads] - [Alloc thread] <= [Avail]):
                  remove thread from UNFINISHED
                  [Avail] += [Alloc thread], done = false
} until (done)
```

Thrashing
- performance degrades when a process thrashes is due to frequent page faults (not heavy context switch), as the process' working set doesn't fit in the memory.
- Thrashing is characterized by slow performance and low CPU utilization (process do not get to progress)
- Thrashing happens when memory is too small for a working set and replacements need to frequently happen to lead to page fault. Increase cache size/associativity to improve, optimize memory access for better spatial/temporal locality, a better replacement algorithm

Scheduling Misc
- No good scheduling algo can achieve high throughput and low latency for any mix of jobs: high throughput is achieved by switching infrequently, whereas low latency is best achieved by interrupting long-running threads and switching to threads with the shortest burst.
- Priority inversion could occur even if only 2 possible priorities for threads: as long as high-priority threads block on a resource held by a low priority thread
- Effective priorities in PintOS are computed as the max of the thread's base priority and the max of each child's effective priority.
- In PintOS with priority donation, immediately after a thread releases a lock, the thread's effective priority may be different from the thread's base priority: There could be multiple locks and other threads with a priority higher than the base priority may have donated their priority.
- A thread in Pintos is holding a lock A. change thread's effective priority: thread releases lock; another thread tries to acquire lock; some other thread dies; the thread calls thread_set_priority and passes in a value less than its current base priority.
- A thread in a critical section will be context switched when its time quanta expires. It can be preempted by the OS.
- Most modern OSs use Banker's algo to avoid deadlocks: FALSE. In most modern OSs, the number of processes is not fixed and the total amount of resources used for each process is unknown, so not possible
- There are scheduler workloads where a non-preemptive scheduler has a better average wait time than a preemptive scheduler. N equal tasks run by RR or FCFS. FCFS has better avg wait time.
- If all jobs have the same size and quanta is larger than job size, both RR and SJF have the same avg response times.

VM
- 0=0000, 1=0001, 2=0010, 3=0011, 4=0100, 5=0101, 6=0110, 7=0111, 8=1000, 9=1001, A=1010, B=1011, C=1100, D=1101, E=1110, F=1111
- Kibi - Ki - 2^10; Mebi - Mi - 2^20; Gibi - Gi - 2^30; Tebi - Ti - 2^40 Pebi - Pi - 2^50; Exbi - Ei - 2^60; Zebi - Zi - 2^70; Yobi - Yi - 2^80
- Page Table: VPN -> PPN, reside in memory. PT is per process

#virtual_pages = 2^(#VPN bits) #physical_pages = 2^(#PPN bits)
Page size = 2^(#offset_bits)
Virtual memory size = #virtual_pages * page_size
Physical memory size = #physical_page * page_size
#PTE = #virtual pages
#PTE bits = #physical_page_num_bits + #metadata_bits
PTE size = #page_table_entry_bits / 8
Page_table_size = #page_table_entries * page_table_entry_size
#byte that can be mapped using a page table
    = #pages_pointed_to_by_page_table * page_size
    = (size of page table / size of PTE) * page_size
Multi-level page table:
    (Max_level * VPN #bits) = lg(#PTE per page)
    vaddr_bits <= levels * (Max_level * VPN #bits) + (offset #bits)
Working set = subset of the address space used during execution
Resident set = subset of address space being held in memory
TLB = VPN+PPN+metadata

- Single level PT require less memory lookups, so faster translation times. Sparse virtual address space results in wasted memory space due to internal fragmentation.
- Pages all being the same size solves external fragmentation.
- The number of entries decreases as the size of the pages increases.
- The size of each entry decreases as the size of the pages increases.
- With VM, 2 accesses to make each time to access a piece of data (read page table, use translated address to read data)
- TLB lookup doesn't count as physical memory operations bc it's a separate hardware.
- Not every virtual addr in a process's virtual address space always corresponds to a valid page table entry. There are more VA than PA so it requires pages to be swapped out for other pages on disk or evicted.
- Each process has its own page table, but TLB is shared across all processes.
- Base & bound registers stored in PCB, not TCB bc all threads in same process share b&b registers.
- Adding a TLB will NOT always make memory lookups and accesses faster. In case of TLB misses, add overhead than no TLB at all.
- Two mechanisms that ensure TLB functions correctly after context switch: 1) flush the TLB during context switch; 2) attach a process ID to each TLB entry to distinguish between processes.
- How does the OS make up for the lack of a use bit in the hardware-supported PTE? The use bit can be emulated in software by keeping one bit per page stored in kernel memory. Then, the OS would set the PTE for each mapped page to invalid so that the first acces of each page would cause a page fault, allowing the OS to both set the simulated use bit to 1 and changing the PTE back to valid.
- In a multiprocessor system with per-core TLB, create several forked processes, bad performance. Why? Any time a thread in a forked process tries to write to a page, it will be copied to a new physical frame and have its permissions updated. This results in TLB shutdown, invalidating all entries in all cores.
- Avoid invalidating TLB between context switches between different processes → add process ID to TLB entry.
- Two distinct page tables can have the same PT3 despite existing in separate processes. This happens when processes share pages.
- Under demand paging, if think of memory as cache for disk, conflict misses are not possible. Any page can be placed anywhere in memory -> cache fully associative. Write-back. Blocksize (assume 32bit addr, 4B words, 4KB pages) = 4KB, 1page.
- Address translation questions: 1) find vaddr, VPN, offset; 2) PTE@(PTPtr+VPN*(PTE size in bytes)); 3) read #PTE bits in table

Page fault

- occurs when a program attempts to access a virtual address not currently located in physical memory.

1) MMU traps the kernel and saves current state information.
2) Kernel finds out which virtual page was needed (e.g. CR2 in x86).
3) Check if the virtual address is valid. Use page replacement policy to remove a page from physical memory and write it to disk.
4) Bring in the page corresponding to the virtual address from disk (e.g. swap map in Linux) and update the page table entry.
5) Restore saved state from (1) and resume execution.

Fully Associative Cache

- only compulsory & capacity miss (no conflict miss)
- Byte offset bits = log2(line size), Tag bits = address bits - offset bits

Direct-mapped cache

Offset = log2(line size); index= log2(#lines); tag = address-I-O

Set Associative Cache

Offset = log2(line size); index = log2(cachesize/(blocksize * associativity)); index = log2(num of sets); tag = address - I - O

Capacity Miss → blocks replaced and later retrieved

Conflict Miss → set-associative or direct mapped, won't occur in fully associative

Belady's anomaly

- Increasing cache capacity can increase miss rate. Applies to algorithms that don't have stack property (FIFO, Nth chance, clock). MIN and LRU have.

Replacement Policy

- A direct mapped cache can have a higher hit rate than a fully associative cache with an LRU replacement policy: If a cache has N blocks, then repeatedly accessing N+1 sequential addresses will exhibit a higher hit rate for direct mapped caches.
- Approximation: {clock → LRU → MIN}, {FIFO}
- FIFO can have the same performance as MIN
- LRU is a poor approx. for MIN when lack of temporal locality

Cache Misc

- In a typical cache, the tag-bits are pulled from the top (most significant bits) of the address, while index and offset bits are pulled from less significant bits. Why wouldn't it make sense to take the cache index from the most significant bits? Cuz the resulting cache wouldn't work well for systems with spatial locality that extended beyond the cache-line. For instance, with the index in the normal place, a set of accesses to an array that was bigger than a cache-line size could still fit into the cache. In contrast, if the index were placed at the top of the address, the index bits would tend to stay constant when accessing such an array. Thus, every array access that changed cache lines would automatically conflict with every other cache line.

- Virtually Indexed Caches: virtual addresses go directly from processor to cache before translation. TLB is consulted only after a cache miss.
- Physically Indexed Caches: addresses must be translated to physical address before going into cache. TLB is consulted on every access before the cache can be accessed.
- It's possible to overlap the TBL lookup and cache access in hardware. The page offset stays the same. If the cache index fits entirely in the page offset, then the SRAM lookup of the cache can happen in parallel with the TLB lookup, with the Tag check happening after the TLB access finishes.
- If cache tag comparison always yields a mismatch, would not affect the correctness of end-to-end memory lookups since tag mismatches in the cache can be corrected by lookups to the memory. The bug would increase effective access time since every access would result in a cache miss.

Pintos Specific / Homework / Other

- The struct thread representing a user process is always stored on the same page as the thread's kernel stack, not user stack.
- A web server that spawns new process to handle each request is NOT well-conditioned.
- Reduce context switch overhead in a highly concurrent system: 1) Use fewer kernel threads (e.g., event-driven programming, or bounded thread pools). 2) Reduce the number of mode switches (e.g., user-level scheduling). 3) Use a scheduling algorithm that switches threads less (e.g., FIFO instead of RR).
- How can kernel synchronize access to data shared by a thread and an interrupt handler? Disable interrupts when accessing data from thread.
- How can a user program synchronize access to a data shared by a thread and a signal hander? Disable signals when accessing data from thread.
- Is there a situation where busy-waiting is more efficient? If the time the thread needs to sleep is less than the time it takes to put the thread to sleep and wake it up, busy waiting is more efficient. E.g. a thread news to enter a short critical section that is currently occupied by a thread currently executing on another core.
- Pipe must be used for when child process inherit FDs from parent process. Does not work for two different processes.
- Socket can be used between different processes.
- In start_process() in Pintos, why is the kernel able to directly modify esp returned by load() without performing an explicit translation? User virtual memory is a subset of kernel virtual memory.

AMAT

AMAT = hit time + miss rate * miss penalty
= L1 hit time + L1 miss rate * (L2 hit time + L2 miss rate * L2 miss penalty)
Memory access= (L1 cache lookup time) + (1 - L1 cache hit rate)(memory access time)
TLB miss time = (memory access) * N, n = level of page table
AMAT = (TLB lookup time) + (memory access) + (1 - TLB hit rate)(TLB miss time)

- TTLB+(1-PTLB*PF) * [TL1+PL1*(TL2+PL2*TM)] + PTLB*{2[TL1+PL1*(TL2+PL2*TM)] + PF*(TL1+TL2+TM+TD)}