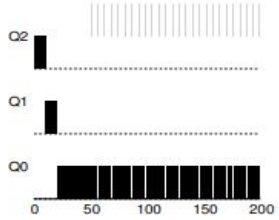
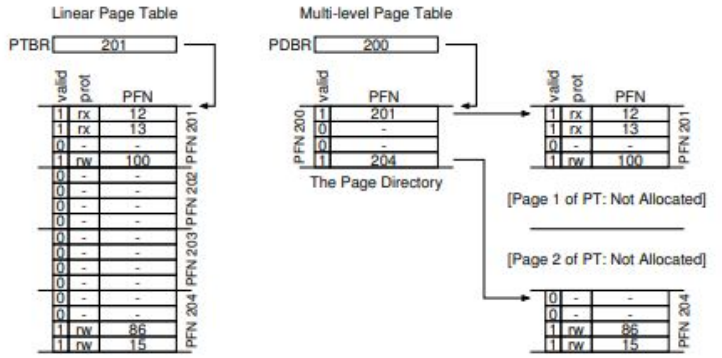
Scheduling Policies

- Batch: FIFO, SJF, STCF
- Proportional sharing: RR, Lottery, MLFQ
- MLFQ (demotion policy, promotion policy)
  - • Rule 1: If Priority(A) > Priority(B), A runs (B doesn't). • Rule 2: If Priority(A) = Priority(B), A & B run in RR.• Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue). • Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue). • Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level., also Rule 5: Priority boost and gaming tolerance.
  - 

- I/O blocking == sleeping

Virtualizing Memory / MMU mechanisms/address translation schemes

- Base & Bounds (2 registers)
  - translate via addition
  - - B&B are two registers that are used to translate registers
  - - they provide an offset to the address when you're converting from virtual address to physical address - BASE
  - - Bounds - applied before or after (depends) that says what is the max address that you are allowed to use.
  - Priviledge mode is needed so programs cant do bad things, also exceptions
- Segmentation (Nx2 registers, typically N=2,3,4)
  - translate via choice+addition
  - Segmentation is like BnB but multiple, one for code, one for data ...
- Paging
  - fixed-sized pages (4K typical)
  - translate via lookup+offset
  - (virtual) page numbers vs (physical) frame numbers
  - uses a translation table, not mathematical relation
  - page table entries (PTE) have control bits:
    - valid
    - readable / writable / executable
    - (dirty, accessed, present, ...)
- Multi-level paging
  - page directory vs page tables
  - - virtual boxes and physical boxes of same size
  - - each virtual mem unit has an offset from start of box and box number
  - - Page table say has 5 entries in it, which is a pointer to one of the physical memory spaces.
  - - T[1] = 6, meaning VM box 1 = found in PM box 6
  - - there also would be flags n such.
  - - each entry is a word, the "6" is the really important stuff, and the rest of the bits can be used to store other information such as validity, user status, read/write, and other features.
  - - if we think about multiple level tables,

- ○ - so instead we have an array of PTE (page table entries, but some PTEs refer to other PTE which then actually point to where they want to in physical mem.
- ○ 

- Translation Lookaside Buffers (TLBs)
  - ○ address space IDs (ASIDs), they are stored on this so it is quicker and don't need to view page table every time. This is to "Make the Common Case Fast"
  - ○ VPN | PFN | bits

Virtual Memory Problems

- Base & bounds requires contiguous physical memory allocation
- Segmentation requires helps trivial cases, but ignores: loadable code, threads, shared memory, ...
- B&B, Segmentation require "malloc-like" memory management
- Paging has excessive overhead due to page table size
- Paging doubles all memory accesses (thus significantly harms performance)
- Multi-level paging has even more memory access overhead (thus ...ugh!)
- TLBs significantly impact context switches and violate isolation

Multi-level Paging w/ TLB Features

- relatively low space and time overhead (but violates isolation, information can leak)
- shared code between processes (DLLs, SOs)
- demand loading of code
- shared memory between processes (low overhead inter-process communication)
- automatic stack growth
- null/stray pointer detection
- memory-mapped I/O (e.g., a file mapped into process address space)
- hardware emulation (e.g., old display hardware)
- virtual memory (see below)

Virtual memory (swapping to disk)

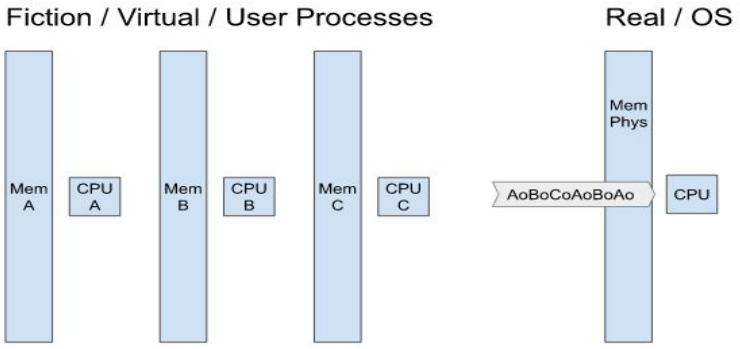
- swap space (relative access cost)
- page fault handling, present page control bit, blocking
- background page writer
- policies: optimal (prescient) FIFO, random, LRU, LFU, clock
- historical (stateful) accounting costs, page control bits: accessed, dirty
- terms: thrashing, working set

Concurrency Snchronization (30 31 32)

- Problems
  - ○ Atomicity - interruption of operations assumed to be atomic
  - ○ Ordering - order is different than expected
  - ○ Deadlock - waiting without moving

    - Livelock - waiting while looping
- User-Space Sync Mechanisms
    - None - Simple, atomic
    - Block Interrupts - Prevents scheduling during atomic instruction sequences, bad OS authority
    - Instructions - Used to implement others
    - Spin-wait -waste of CPU
- OS Sync Services
    - Thread join - Very high-level order management
    - Mutex - Informs the scheduler of critical sections
        - lock(a) - begin crit section : unlock(a)- end crit section
    - Cond Var - identifies threads waiting for a resource
        - signal(a) : wait(a)
    - Semaphore - Counts and queues up threads waiting for a resource (Producer/Consumer Model)
        - post(a) : wait(a)
        - Contributed by Dijkstra, allows data to be transferred between threads
- Common Sync Problems
    - Producer/consumer - making and using data, such as games
    - Read/Writer - simultaneously access data for reading, others exclusively write, databases
    - Dining Philosophers - threads require multile resources simultaneously, order of obtaining matters
- Thread safe Data structures
    - Concurrent access across threads, not by OS, Usually via mutex/locks, ideally atomic sync
    - Push and pop could implement producer/consumer without visible sync code

Threads - Multiple CPUs per Process

- 

- Thread Models
    - User-space threads(old unix) - OS schedules between processes, proces scheudles between threads, blocked thread blocks the whole process
    - Kernal Threads - Threads = Processes(Linux) same address space tho, Threads as unique OS Resource(windows) but kernal schedules threads and processes

xv6 per-process state

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed ?????
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

-