111 Cheat Sheet

【API & ABI】

- **<API: Application Programming Interfaces>**
- they are written at the source programming level (e.g. C, C++ , Java, Python). eg. write()
- API specifications are a basis for software portability, but applications must be recompiled for each platform
- +++ an application written to a particular API should easily recompile and correctly execute on any platform that supports that API
- +++ any application written to supported APIs should easily port to their platform.
- --- the API must be defined in a platform-independent way (to have the above benefits)
- E.g. defined 64 bit int in 32 bit machine, accessed individual bytes of int in big-endian machine.
- E.g. particular feature implementation (e.g. fork(2)) not be implementable on some platforms
- **<ABI: Application Binary Interfaces>**
- **An Application Binary Interface is the binding of an Application Programming Interface to an Instruction Set Architecture.**
- Contains: binary representation of key data types, call to and return from a subroutine, Stack-frame structure,
- - and parameters, return values, registers, system calls, and signal deliveries.
- - and formats of load modules, shared objects, and dynamically loaded libraries
- portability benefits +++ allows us to distribute SW to different platforms.
- The resulting binary load module should correctly run, unmodified, on any platform that supports that ABI.
- Used by compiler, linkage editor, program loader, operating system, or programmer who writes assembly language subroutines.
- ABI only uses the system call interface for the system for which it was compiled for. System call interface is a subset of ABI.

【Linkage editing】

- **<Compiler (-> source.s)>**
- Reads source modules and included header files, parses the input language (e.g. C or Java), and infers the intended computations, for which it will generate lower level code (assembly code).
- **<Assembler (-> object.o)>**
- each line of code often translating directly to a single **machine language** instruction or data item.
- allows the declaration of variables, the use of macros, and references to externally defined code and data
- func & local symbols not have yet been assigned hard in-memory addresses. Only be expressed as offsets
- **<Linkage editor (static library.a -> executable)>**
- 把所有东西都放进 virtual address space. **filling in all of the linkages (connections) between the loaded modules (by linkage editor, ld(1))**
- reads a specified set of object modules, placing them consecutively into a virtual address space, and noting where (in that virtual address space) each was placed.
- searches a specified set of libraries to find object modules that can satisfy those references, and places them in the **evolving virtual address space** as well.
- The resulting bits represent a program that is ready to be loaded into memory and executed
- **<Program loader>**
- part of the operating system
- examines the information in a load module, creates an appropriate virtual address space, and reads the instructions and initialized data values from the load module into that virtual address space.
- map shared libraries into VAS.

【Modules】

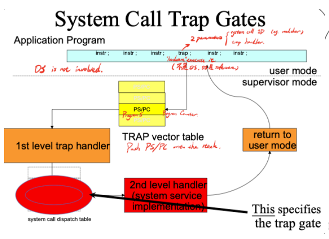
- **<executable (load) modules>**
- the code (machine language instructions) within an object module are Instruction Set Architecture (ISA) specific.
- ELF (Executable and Linkable Format) makes the formats common across ISAs.
    - header section
    - Code and Data section
    - symbol table (for external)
    - A collection of relocation entries
- **<relocatable object modules>**
- incomplete. They make references to code or data items that must be supplied from other modules.
- not yet been determined where (at which addresses). Only have relative address.
- **<segments>**
- **Code:** start with load module(all external references have been resolved), loaded into memory, read/execute only and sharable
- **Data**: must be initialized in address space(initial contents be copied from load module), read/write, private, grow/shrink by program(sbrk)
- **Stack**: Each procedure call allocates a new stack frame(for).

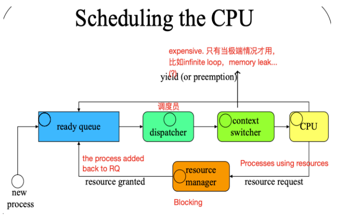
【Library】

- **<Shared Libraries (run-time loadable)>**
- Reserve an address for each shared library.
- SL -> read-only code segment.
- Assign a number (0-n) to each routine + put redirection table at the beginning of that shared segment
- stub library, that defines symbols for every entry point in the shared library
- Linkage edit the client program with the stub library.
- When run, OS notices that the program requires shared libraries, open the associated (sharable, read-only) code segments, and map into address space. (in ld.so(8))
- **+++**
- single copy implementation, use less space, faster program loads
- version controlled by a library path environment variable, easier and better library upgrades.
- program not affected by changes in the sizes of library routines or the addition of new modules.
- stub modules -> one shared library to make calls into another.
- **- - -**
- read-only -> no static data. Multiple processes accessing the shared global data at the same time is a problem.
- cannot make any static calls or reference global variables to/in the client program.
- Must relink if libraries changed
- **<Dynamically Loaded Libraries>**
- delay the loading of a module until it is actually needed.
- User mode: dlopen(3). OS mode: dynamically load device drivers or file system implementations.
- +++ pre-compiled applications can exploit plug-ins to support functionality that was not implemented.
- Calls from the client application into a shared or Dynamically Loaded library are generally handled through a table or vector of entry points
- SL impose numerous constraints and simple interfaces on the code. DLL can perform complex initialization and support complex (bi-directional) calls between the client application and library.
- **Process:**
- program choose and load that library into its address space
- calls the supplied initialization entry point, app and DLL bind to each other
- requests services from the DLL by making calls through the dynamically established vector of service entry points.
- When doesn't need, calls the shut-down method, and un-load.

【Traps and Interrupts】

- **<Procedure vs interrupts>**
- Procedure expects function performed and returned value.
- interrupts are initiated and defined by HW.
- computer state should be restored after interrupts.
- **<Interrupt mechanism>**
- Process is unable to execute privileged instructions, must ask kernel to do so。
- TRAP vector table: 连接对应的 possible external interrupt/execution exception, Program Counter, 和 Processor Status. Switch to supervisor mode.
- The **trap table** is used to specify what actions the operating system should take when a particular exception occurs. It will contain pointers to exception handling routines, indexed by a number specifying the particular exception that occurs. It is consulted whenever an exception actually occurs. It is loaded at boot time and typically will not change until the next reboot.
- CPU 在 table 里 load 新的 PC/PS，并 push 到 CPU stack，并继续新的 PC 指定位置的 execution
- First level handler saves **all other registers**, and chooses appropriate second level handler.
- Second level handler deals with the event, kill the process, and return to FLH, which restores registers, execute 'return', (privileged) reload PC/PS, and resumes execution at the interruption in user mode.
- 优点：translates HW-driven call to FLH (higher level language) -> SLH.
- 缺点：user mode -> processor mode, new address space(context switch)，会很慢，并 loss of CPU caches.


System Call Trap Gates
Application Program
2 parameters
system call ID (eg. read char)
trap handler.
instr ; instr ; instr ; trap ; instr ; instr ;
"hardware" execute it.
OS is not involved.
(不是 OS, 也不是 software)
user mode
supervisor mode
PS/PC
Program S
Program Counter
TRAP vector table
Push PS/PC onto the stack.
1st level trap handler
return to user mode
2nd level handler (system service implementation)
system call dispatch table
This specifies the trap gate



【Scheduling】

- 7.1 Workload assumptions: Run-time known, arrive at the same time, runs to completion, only use CPU
- 7.2 Scheduling metrics: $T_{turnaround} = T_{completion}-T_{arrival}$
- 7.3 FIFO(first in first out) or FCFS(First come first serve)：排队。
    - Convoy effect: 一个顾客占时太久，耽误大量小顾客。
- 7.4 Shortest Job First(SJF)
    - Optimal in theory. 但是如果小顾客稍微来晚点就也要等很久。
    - Non-preemptive. FIFO, SJF, RR,
- 7.5 Shortest Time-to-Completion First (STCF) / Preemptive Shortest Job First (PSJF)
    - it can **preempt** job A and decide to run another job, perhaps continuing A later.
- 7.6 $T_{response} = T_{firstrun}-T_{arrival}$
- 7.7 Round Robin(time slicing). instead of running jobs to completion, RR runs a job for a time slice (scheduling quantum) and then switches to the next job in the run queue.
    - amortize(分摊) the cost, but TOO MUCH context switches
- clock interrupt: ensure that runaway process doesn't keep control forever. -> preemptive scheduling

【MLFQ：Multi-Level Feedback Queue】

- 1&2: Priority 更小，run first；同样，RR。
- 3: When job enters, placed at the highest priority.
- 4A: If used entire time slice, priority reduced
- 4B: If gives up CPU befure time slice, priority unchanged
- if an interactive job, for example, is doing a lot of I/O, it will relinquish the CPU before its time slice is complete. Don't penalize it.
- 5: periodically boost the priority of all the jobs: After some time period S, move all the jobs in the system to the topmost queue
- 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

【Segmentation】

- **segmentation**: With a base and bounds pair per segment, we can place each segment independently in physical memory. (code, heap, stack)
- **segmentation fault:** tried to refer to an illegal address, which is beyond the end of the heap. HW detects address out of bounds, traps into the OS.
- **Base & bound registers**. Check within the bound first, to ensure it is legal. -> exception handler(into kernel mode) -> save base and bounds registers in **process control block**(PCB))
- **virtual address = segment+offset**. Segment 00: code segment (refers to the **segment table**)-> refers to the base&bound pair, and plus the offset.
- hardware also has a bit to indicate "growing positive".
- **Protection bits**: defines read-write-execute for code sharing.
- **Internal fragmentation**: an allocator hands out chunks of memory bigger than that requested, any unasked for (unused) space in such a chunk.
- **Free list**: A linked list with each entry recording free space's address and length.
- **Splitting**: a free chunk of memory that can satisfy the request and split it into two. The first chunk will return to the caller; the second chunk will remain on the list.
- **Coalescing**: when returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly- freed space sits right next to one (or two, as in this example) existing free chunks, merge them into a single larger free chunk.
- Best fit, worst fit, first fit, next fit(a pointer pointing to the location last looked at)

- **Segment relocation**: external fragmentation still exists, because segments are still required to be contiguous.
- **Segregated List(buffer pool)**: having a chunk of memory dedicated for one particular size of requests.
- **Slab allocator**: allocates a number of object caches for kernel objects that are likely to be requested frequently 根据 frequency 变更 slabs 大小
- **binary buddy allocator**：free memory 慢慢变大找 2^N。-> internal fragment. Still has external frag.

【Paging】

- Segmentation -> variable-sized pieces -> fragment problems. holding the process' code or stack
- Paging -> fixed-sized pieces(page) -> no external fragmentation. to load, relocate, manage the space more flexibly. Internal frag: average only 1/2 page (of the last one).
- **Page table**: store address translations for each of the virtual pages of the address space
- **virtual address** = VPN(virtual page number) 4 bits(16 pages 看行数) + Offset 4bits(16 bytes)
- PFN: physical frame number, PTE: page table entry.
- **MMU**: memory management Unit. the part of the processor that helps with address translation.
- PTE 构成：PFN + valid bit, protection bit, present bit(physical memory or disk), dirty bit(whether the page has been modified since it was brought into memory), reference bit(whether a page has been accessed, popular or not), user/supervisor bit (U/S)

【TLB】

- TLB: translation-lookaside buffer. part of MMU.
- TLB hit: TLB holds the translation for this VPN.
- TLB miss: If VPN is not found in TLB -> the hardware locates the page table in memory (using the page table base register) and **looks up the page table entry** (PTE) for this page using the VPN as an index. If the page is valid and present in physical memory, the hardware extracts the PFN from the PTE, installs it in the TLB, and **retries** the instruction -> TLB hit
- Eg. access an array: the first time will be a TLB miss, and the second time will be a TLB hit. When comes to new page, it will have a new TLB miss.
- TLB rate is high due to spatial locality and temporal locality.
- A TLB entry = VPN | PFN | valid bit+protection bits+dirty bit
- When context switch, TLB's translations are only valid for the currently running process.
  - 方法 1 flush the TLB。清空
  - 方法 2 address space identifier (ASID), ≈ PID.

【Physical Memory】

- **swap space**: reserve some space on the disk for moving pages back and forth
- **age fault**: accessing a page that is not in physical memory
- When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to **disk** to fetch the page into memory
- **page-replacement policy**: process of picking a page to kick out or replace when memory is full.
- **control flow for fetch data in memory**: 1. Access TLB; 2. look up page table for address in physical memory; 3. page fault, replace page in physical memory with page in disk.
- **Low watermark** (LW): when too many pages in memory. **high watermark** (HW): evict pages until here.
- **cache hits**: the number of times a page that is accessed is found in memory.
- **average memory access time** (AMAT) = (P_Hit · T_Mem)+(PMiss · T_Disk)
- **cold-start miss** (or compulsory miss): the cache begins in an empty state
- FIFO ≈ Random

- page stealing algorithm. Every process is continuously losing pages and stealing pages.
- working set 会根据 reference page 的多少和频率来变大变小。

【知识点】

- **<Copy-on-write>**
- Fork() creates an identical process from a parent process
- Copy-on-write means it only makes a copy when the memory address is written to(by parent or child), and otherwise two process refer to the same physical memory
- Copy-on-write makes fork() extremely fast because initially the two process' virtual address space heap to the same physical address, and no copies are made, which reduces the overhead
- **<IPC vs forking>**
- - Major difference is fork results in copy-onwrite, while shared memory IPC doesn't.
- -   new process has its own page table, but its contents are the same.
- - Stack isn't shared in shared memory IPC.
- **<exec>**
- code, stack, heap replacement
- it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized.
- **<abstraction>**
- Encapsulate implementation details, and provide more convenient or powerful behavior.
- Interpreter: instruction reference + repertoire + environment reference + interrupts.
- **Federation Frameworks**: A structure that allows many similar, but somewhat different, things to be treated uniformly.-> interface + plugging in implementations + optional features
- **Layering**: allows good modularity(build multiple services, use multiple underlying services to support high layer..), but add performance penalties(change data structures when change layer) and limit what upper layer do.
- **Object state**: priority, pointer into a file, completion condition of I/O, list of memory pages-> managed by OS itself(not by user code, must ask OS to access or alter)
- **Process state**: registers(generals, program counter, processor status, stack pointer, frame pointer), process' own OS resources(open files, current working directory, locks), OS-related state info
- **Process Descriptor**: basic OS data structure for dealing with processes, store all the info of process(pid. state to restore, references to allocated resources, info to support process operations), managed by OS. PD example: linux process control block.
- A lot of process state is stored in the other memory areas.
- **Process Table:** data structure of OS to organize all currently active processes.
- **Process creation:** Windows**:** blank process; Linux: same stuff as the old one.
- **Process running: LDE(limited direct execution):** let the program run directly on the hardware; however, at certain key points in time (such as when a process issues a system call, or a timer interrupt occurs), arrange so that the OS gets involved and makes sure the "right" thing happens. **Key to good system performance!**

- Least-Frequently-Used (LFU). & Least-Recently- Used (LRU).
- **Clock algorithm**: A clock hand points to some particular page to begin with (it doesn't really matter which). When a replacement must occur, the OS checks if the currently-pointed to page P has a use bit of 1 or 0. If 1, this implies that page P was recently used and thus is not a good candidate for replacement. Thus, the use bit for P set to 0 (cleared), and the clock hand is incremented to the next page (P +1). The algorithm continues until it finds a use bit that is set to 0, implying this page has not been recently used.
- **dirty bit**: whether a page has been modified or not while in memory. VM systems prefer to evict clean pages over dirty pages. 因为 evict 不用重新 write to disk. we can do background write-out of dirty pages (by turning dirty pages into clean pages) to further reduce I/O involved in their eviction.
- **Page frame:** we divide it into fixed-sized units, each of which we call a page. Correspondingly, we view physical memory as an array of fixed-sized slots called page frames; each of these frames can contain a single virtual-memory page.

【Garbage Collection】

- seeking out no-longer-in-use resources and recycling
- Commands: close(), free(), delete, return, exit.
- 共享资源 Resource manager: automatically free the object when the reference count reaches zero.
- 取消手动档：reference can be copied or deleted without notify the OS；有些语言减少难度；太久没用自动删；记录 allocation/release 太多太费时。
- 方法：建一个 list of existed resources，从头到尾只要是 reachable resource，就 remove；最后 free 所有还在 list 上的
- 缺点：电脑可能一直运行流畅直到突然中断来做 GC

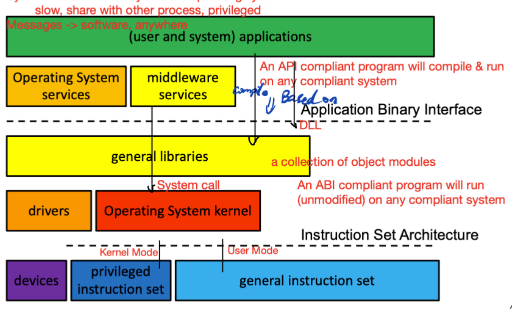
【Defragmentation】

- reassigning and relocating previously allocated resources to eliminate external fragmentation
- **Coalescing** is only effective if adjacent memory chunks happen to be free at the same time.
- In SSD, erasure can only be done in large(64MB) units. 所以先找到一块，把在用的 4K blocks 移到新的地方，再删这块并加入 free list
- Clean up the free space（像汉诺塔）：移出所有有用的 block，清理，再把 blocks 放回去，达成连续，并不断重复。

【Working Set】

- LRU 成功原因：most programs exhibit some degree of temporal and spatial locality (Only for single program/process).
- 和 Round-Robin 互相排斥，因为 RR runs periodic.
- RR 时采用 per-process LRU。
- Thrashing：The dramatic degradation in system performance associated with not having enough memory to efficiently run all of the ready processes。内存变少，Page fault 变多。
- working set size：增加 number of page frames 对表现没啥影响，而减少则可见的变差。
- WS 大小：不同 process 以及不同时候都不一样，可以根据 page faults 数量来观察：minimize page faults(overhead), maximize throughput(efficiency).
- **<Implementation>**
- 类似 Clock Algorithm：循环 pages，如果最近被 referenced，记下现在 accumulated time，并标记为非 referenced；如果没有，则算它的 age，如果大于 target age 则 return（否则小于就换算是 thrashing）。
- Age 取决于 accumulated CPU time，只有当 ower 运行并没有 referencing 它的时候才增加。
- 如果没有任何小于 target age 的，就 too many process 了。replace oldest 或者减少 process。
- **<Dynamic Equilibrium to the rescue>**

Software Layering



Subroutines(fastest, high level) -> functions in libraries
System calls -> an entry into the operating system
slow, share with other process, privileged
Messages -> software, anywhere
(user and system) applications
Operating System services
middleware services
An API compliant program will compile & run on any compliant system
compile
Based on
Application Binary Interface
DLL
general libraries
a collection of object modules
System call
An ABI compliant program will run (unmodified) on any compliant system
drivers
Operating System kernel
Instruction Set Architecture
Kernel Mode
User Mode
devices
privileged instruction set
general instruction set