Arm ISA

Available registers include:

- 64-bit general-purpose registers x0-x30 which are available as 32-bit by w0-w30.
- Stack Pointer in all exception levels SP_ELx (where x is 0,1,2,3).
- Three exception link registers ELR_EL1, ELR_EL2, ELR_EL3.
- Three saved program status registers SPSR_EL1, SPSR_EL2, SPSR_EL3.
- Program counter pc.
- Process state register including NZCV condition flags, DAIF interrupt disable flags, current exception level EL.

Instructions include:

- mov dst, src where dst must be a GPR.
- calc dst, src1, src2 where calc include add, sub, mul, div. There are smul/umul and sdiv/udiv determining whether calculation is signed or not. Adding suffix s to these instructions sets the condition codes.
- lsl/lsr dst, src1, src2 meaning logical shift left/right. asr ... means arithmetic right shift where the highest digit is extended according to the sign of the number.
- eor doing XOR, orr doing OR, and doing AND, mvn doing NOT (bit-wise).
- ldr/str reg, addr either load content into reg or store content in reg into addr.
- cmp/cmn/tst src1, src2 sets the condition code according to the result of src1-src2, src1+src2 and src1 & src2 respectively.
- b.xx <label> branches to <label> according to the condition codes. br reg jumps to the destination stored in a register.
- bl <label> and blr reg performs function calls (stores return address into x30 and jumps). ret jumps to the destination stored in x30.
- svc and eret does exception level switching.

In instructions, **modified registers** are used to reduce the number of instructions and number of registers occupied. *e.g.* add x2, x2, x2, lsl #1 means x2 = 3 * x2;. Bit-extension using NxtM where N is u or s indicating whether it is signed extension; M is B, H or W indicating number of digits (8, 16, 32). *e.g.* sxtx x1, w2 means signed extend 32-bit w2 into 64-bit x1. In calculations, use add x1, x1, w1, uxtw to do extension.

Addressing modes include: [reg], [reg, offset], [reg, offset]! (equal to reg += offset then find [reg]) and[reg], offset (equal to find [reg] then reg += offset). The offset can also be modified register, *e.g.* str [x0, x0, lsl #2].

Function calls:

```
stp x29, x30, [sp, #-32]!
mov x29, sp
...
ldp x29, x30, [sp], #32
ret
```

The first line stores x29 and x30 into top of the stack (with allocated size). x29 points to the top of the stack (**FP**) and x30 points to returning address. The last line restores the two registers.

(* Notice that the final level of function call doesn't need such operation.)

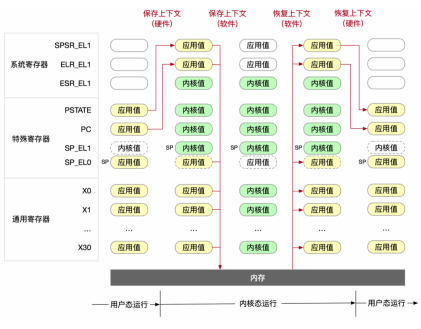
The caller passes at most 8 parameters using x0-x7 while the callee returns using x0. More than 8 parameters: the caller pushes to stack (in the order of right-to-left so that left-most parameter is at top); the callee visits through sp+offset.

It is assumed that x9-x15 are caller-save and x19-x28 are callee-save.

特权级切换

通过触发异常，作用是 (1) 将 CPU 控制权交给内核，即向 OS 请求服务，(2) 操作系统管理，防止程序错误/恶意程序。异常向量表在VBAR_EL1 中。分为**同步异常**（svc 或指令出错）、**异步异常**（中断、SError）。异常处理函数完成后，可能回到当前指令/下一条指令/其他进程。

OS 启动时设置异常向量表：msr vbar_el1, x0。为了判断异常类型，提供更多关于异常的信息，mrs x1, esr_el1。



在切换过程中，硬件将异常指令地址保存在ELR_EL1 中，原因保存在ESR_EL1 中，pstate 保存在SPSR_EL1 中，修改pstate 中特权级标志位，找到异常处理函数入口写入pc（pc 和栈必须由硬件完成）。软件 (OS) 将应用程序的 CPU 状态（processor context，包含 GPR、pc、sp、pstate 和一些系统寄存器如页表基地址寄存器）保存到内存。

异常处理完成后，eret 指令作用为：SPSR_EL1 中保存内容写入pstate，将ELR_EL1 中地址写入pc。

对于系统调用，提供类似于函数调用的接口。一般函数调用出错时返回-1，设置全局变量errno 为错误值。系统调用 API 则通过 wrapper code 将系统调用返回值转换为库函数形式的返回值。若寄存器放不下参数，则通过内存指针传参。

高效系统调用

系统调用频繁且成本很高，对此进行优化。

Virtual Dynamic Shared Object 可以将代码加载到与应用程序共享的内存页。

Flexible System Call 引入 system call page，应用程序 push 请求，内核 poll。这样不需要特权级切换，但是需要内核 polling 并且 system call 变为异步。

PrivBox：将用户代码以沙盒形式运行于 privileged CPU mode。

OS 抽象

通过进程实现的 CPU 抽象：分时复用。每个进程包含一个进程标识号 (Process ID)，运行状态，地址空间和打开的文件。操作系统根据 Timer 中断、用户执行的 read/sleep 等系统调用进行进程切换。

进程相关接口：

- Getpid() 返回进程 PID，Getppid() 返回父进程 PID
- exit(int status) 函数终止进程
- fork() 函数创建一个相同的子进程，在父进程中返回子进程 PID，在子进程中返回 0，本质是复制了 PCB 数据结构
- execve(char* filename, char* argv[], char* envp[]) 运行一个新的程序，不再返回（除非运行报错）
- waitpid(pid_t pid, int* status, int options) 挂起调用进程并等待某个子进程（若pid==-1 则等待所有子进程）返回，返回值为子进程 PID 或-1。若不存在子进程则返回-1，errno=ECHILD；若等待中断则返回-1，errno=EINTR。status 用于带回子线程的 exit 状态。

僵尸进程：子进程需要等待父进程回收，因此若父进程不回收子进程则产生僵尸进程。父进程终止时，内核的 init 进程会回收父进程创建的僵尸子进程。

通过虚拟内存实现的内存抽象。提供不同进程间内存的隔离和管理，使实际可用的内存不受物理内存容量的限制。

通过文件实现的 IO 抽象：文件定义为一串字节序列，所有 IO 都通过读写文件的接口完成。Directory 也是文件，由一组文件名到文件的映射构成。

- int open(char* filename, int flags, mode_t mode) 提供打开文件接口，返回文件标识符fd 或-1（错误）。
- int close(int fd) 返回成功（0）或失败（-1）。
- ssize_t read(int fd, void* buf, size_t count 返回读到的字节数，0 表示遇到 EOF，-1 表示失败。
- ssize_t write(int fd, const void* buf, size_t count 返回写入的字节数，-1 表示失败。

但是操作 IO 设备需要更多参数，如调节音量、调节分辨率，因此使用底层接口 ioctl 向设备发送/接受数据。

启动流程

- BIOS-ROM 上电自检、找到第一个可启动设备，加载第一个块（包含 bootloader）到内存中，跳转到 bootloader。
- Bootloader 进行硬件初始化，加载内核代码到内存中 (0x80000)。
- 内核初始化 MMU、外设和其他内核。
- 用户态进程加载 shell，开始等待键盘输入。

内核启动：设置异常级别为EL1，设置页表开启虚拟内存，设置异常向量表并打开中断。

虚拟内存

从虚拟内存到物理内存的映射通过**分页**机制实现，将虚拟/物理内存划分为等长、连续的页，映射关系由**页表**实现。虚拟内存 n bit，物理内存 m bit，页大小 $P = 2^p$，

则虚拟地址分为 $n-p$ bit 的虚拟页号 VPN 和 p bit 的页内偏移 VPO；物理地址分为 $m-p$ bit 的物理页号 PPN 和 p bit 的页内偏移 PPO。

地址翻译：MMU 接收虚拟地址 VA，将 VPN 通过页表映射为 PPN，PPO=VPO，访问对应的物理地址。Arm 中通过SCTLR_EL1 寄存器开启页表功能，页表基地址寄存器TTBR0_EL0/1 存储一级页表所在位置。页表映射中为了防止页表的空间消耗引入多级页表；为了加速多级页表的多次内存访问引入 TLB 以及大页。

降低 TLB Flush 的开销：使用 Address Space ID。每个进程拥有一个 8/16 位的 ASID，并填写在TTBR0_EL1 的高位和 TLB 每一项中，在进行地址翻译时将二者对比，不匹配则 TLB Miss。这样在切换进程后不需要刷新 TLB，但是对于修改页表映射导致的 flush 仍然无法避免。

多核 TLB：一般而言修改页表不需要对其他核的 TLB 进行 flush，但是如果一个进程在多个核上运行，就需要通过进程调度信息刷新对应核的 TLB。刷新操作在 Arm 上通过指令TLBI ASIDE1IS 完成，在 x86 上通过中断完成。

操作系统管理页表映射：映射到物理内存的两种方式：立即映射/延迟映射。延迟映射解耦虚拟内存分配和物理内存分配，操作系统通过缺页异常进行物理内存的分配，因此缺页异常需要区分合法与非法。为了管理方便引入内存段（在 linux 中为vm_area_struct(VMA)，使用查找树存储，在进程创建时通过 ELF 段信息分配，或在运行中通过堆栈扩张/缩小、mmap（把一个文件映射到内存中）分配。同一段对应相同的权限，因此管理方便）。在硬件异常 page fault 触发后，操作系统通过比较FAR_EL1 和内存段区域判断是否合法，非法则触发软件错误 segmentation fault。

预先映射：延迟映射导致访问延迟，而应用程序具有时空 locality，因此采用预先映射。操作系统向应用提供系统调用，让应用程序决定策略。

- int madvise(void* addr, size_t length, int advice) 将语义信息传给内核用于优化。
- int mprotect(void* addr, size_t len, int prot) 改变内存权限，如动态生成的二进制代码可以借此转化为可执行。

共享内存 + 写时拷贝：不同进程的虚拟内存映射到同一段物理内存，可以用于节约物理内存或进程间通信。用于节约物理内存时使用写时拷贝，通过将物理内存设置为 read-only、在页表项进行 copy-on-write 标记实现。在fork 中可以提供性能提升。基于此可以实现**内存去重** (memory deduplication)，操作系统主动扫描内存，合并相同内容的物理页。

内存压缩：将最近不用的内存页进行压缩，win10 直接压缩不换出，因此速度更快，但是释放的内存空间更少；linux 压缩并写入磁盘缓存，减少磁盘实际的 I/O，增加磁盘寿命。

透明大页：在观察到应用程序需要连续的小页之后直接分配大页。

操作系统分配物理内存：应用程序触发延迟映射；内核自身需求；换入换出；申请设备的 DMA 内存。简单的物理内存分配通过对每个页设置 bitmap 实现，但当应用程序需要连续的页分配时，这会导致外部碎片。

伙伴系统 (buddy system)：将 2^k 个页统一管理，分配时一个 block 可以分裂为两个 buddy block，释放时两个 buddy block 可以合并为大 block。采用空闲链表记录每阶的空间 buddy block，分配时将满足需求的最小空闲块进行分裂，释放时则递归合并成为大的空闲块。寻找伙伴块很方便，因为互为 buddy 的两个块仅有 1bit 不同，该 bit 决定块的大小。摊还分析可得分配与合并的期望时间复杂度均为 $O(1)$。

SLAB/SLUB：对于较小的内存需求（即内核自身需求，如 VMA 等数据结构），使用kmalloc 和kfree 直接映射（不能使用 on-demand paging，因为内核处理缺页时不能再次触发缺页，并且内核有足够高的权限访问所有物理地址）。SLUB 从 buddy system 中获得大块的 slab，每个 slab 对应一个固定大小（如 32Byte）作为分配的单元 slot。资源池中则是对于每个 slot 大小，分配多个 slab。

Slab 内部采用空闲链表记录每个 slot 是否空闲，next_free 指针指向第一个空闲 slot，此后按照链表的工作方式进行内存分配与释放。资源池中有一个 current 指针指向当前 slab 以及 partial list 指向部分占用的其他 slab。整个 slab 分配完之后从 buddy system 中获得新的 slab，整个 slab 都被释放后向 buddy system 释放 slab，

换页 (Swapping)：为了获得大于实际物理内存的内存容量，通过磁盘的 swap 分区换入换出。换页机制类似于延迟映射，以时间换空间，因此同样存在预取 (prefetching) 机制。对于任何一个虚拟页，可能 **(1)** 未分配，**(2)** 已分配但没有物理页映射，**(3)** 已经映射到物理页，**(4)** 物理页被换出（没有物理页映射的虚拟页在页表中为全 0，而被换出的页仅是 valid bit 被置 0，可以借此区分）。因此我们只需要决策何时触发换页、如何换页。

- **何时换页**：可以在用完所有物理页后按需换出，但是在内存资源不足时分配时延很高；也可以设立阈值，在空闲物理页数量较低时内核在空闲时进行换出，确保空闲页在阈值以上（linux watermark）。
- **换页策略**：理想策略为选择最长时间内不会被访问的页。

  FIFO：维护一个队列记录换入顺序，但是会导致 **Belady's Anomaly** 问题。改进为 **Second Chance**：每个页有一个访问标志位，访问在队列中的页则置 1。换出时无标志的直接换出，没有标志的放入队尾。问题在无法获取访问频率信息。

  LRU：反应访问频率信息，使用链表（需要频繁修改前后链接关系），每次访问将当前页放在链表尾端，换出链表头部页面。但是循环访问中 LRU 表现很差，并且排序的内存页很多，会有额外负载。改进为 **Clock Algorithm**：每个物理页一个访问位，被访问或加入内存时置 1，时针指向下一个物理页。当需要换出时，时针旋转检查，将 1 变为 0，0 则驱逐。实现时需要反向页表映射，即记录每段物理内存对应的虚拟页号。

Thrashing：在活跃进程过多时，需要不断地进行换页而几乎完全无法执行程序，且调度器认为 CPU 利用率下降，载入更多进程，进一步加剧。使用**工作集模型**避免 thrashing：$W(t,x)$ 为进程在 $(t-x,t)$ 内使用的内存页集合，被推定为接下来 x 的时间内也会使用，因此要么将全部工作集保存在内存中，要么全部换出。跟踪工作集需要时钟中断，扫描每个物理页，若该页被访问，则记录 Age $=0$，否则更新 Age。于是工作集为 Age 小于一定值的内存集合（最后由内核对访问位清零）。

物理内存管理的**需求**：有能力分配连续的物理页，能够随时回收。**评价指标**：资源利用率（内部碎片和外部碎片）、分配速度（分配与释放操作的时延）、公平性（避免应用独占内存）。

进程、线程

进程：程序运行时的抽象，包含代码、数据、寄存器、内存空间、pstate 等。状态包含新生、就绪、运行、僵尸和终止。

进程控制块 (Process Control Block)：存储在内核态，保存了进程 PID、退出状态、执行状态、子进程列表（父进程只允许对子进程进行waitpid 系统调用，防止信息泄露）、所有处理器上下文、页表映射以及应用的内核栈（该栈给内核使用，在切换进程/系统调用/异常处理时临时保存上下文）。应用自身的栈通常不同于操作系统的内核栈（可行性取决于是否支持内核态抢占）。进程调度中从调度队列（PCB 数组）中去除一个 PCB，根据 PCB 恢复上下文，eret 恢复执行。

进程创建：调用process_create 后，初始化 PCB，加载可执行文件（创建页表映射），准备运行环境（在int main(int argc, char* argv[], char* envp[]) 中的参数），初始化处理器上下文。

进程退出：调用process_exit(0) 后（一般直接return 由libc 执行），一般思路是直接销毁上下文、内核栈、虚拟内存空间、PCB，最后调用schedule()，但问题在于不能在系统调用中销毁内核栈，否则无法返回进程执行后续销毁任务，因此在进程中记录int exit_status 和bool is_exit，只设置退出状态，在父进程的waitpid 系统调用中执行回收、销毁操作。由此导致不调用waitpid 的父进程会创建出**僵尸子进程**，即退出后没有回收的子进程，被init 进程回收。

比较不同复制接口：fork 函数直接复制 PCB，完全拷贝，因此性能和可扩展性差，在需要 fork+exec 时采用vfork 函数，父子进程共享同一地址空间。clone 函数提供更可控的接口进行选择性拷贝，但复杂性高容易出错。posix_spawn 函数提供 fork+exec 的替代，性能好但是不够灵活。

进程切换：进入内核态，保存处理器上下文（硬件），切换进程上下文（软件，即切换指向 PCB 的全局指针，将vmspace 和stack 分别存入sp_el1 和ttbr0_el1），恢复新进程的处理器上下文，返回用户态。

线程：为了单一进程利用多核资源，减小进程间的隔离和管理开销。线程只包括执行所需的最小状态（寄存器和栈），代码和数据由进程提供。调用pthread_create 即可创建。进程内所有线程独立执行（因此需要join 函数阻塞线程确保执行顺序），但是系统调用会影响所有线程。线程需要回收，因此在主线程不一定手动回收资源的情况下可以在子线程中调用detach(pthread_self()) 函数防止资源泄露（但是使用了detach 后无法 join，因为 join 之前就会销毁线程；并且 detach 后的线程仍然会受到主线程终止隐式调用exit 带来的影响而无法继续执行，若希望子线程单独执行下去可以使用pthread_exit(0) 只退出当前线程，但是无法结束）。

多线程进程：地址空间中每个线程有自己的用户栈和内核栈。**内核态线程**：内核创建，信息存放于内核。**用户态线程**：用户态创建，信息存放于应用数据。**多对一模型**：单个内核态线程映射多个用户态线程，轻量化（切换方便）但是易阻塞。**一对一模型**：一一映射，内核态线程方便操作系统执行调度，可扩展性好，但是线程切换需要经过内核。**多对多模型**：N 个用户态映射到 M 个内核态（$N>M$）。

在一对一模型中，线程数据结构为 **TCB(Thread Control Block)**。内核态线程的 TCB 与 PCB 类似，在 linux 中使用同一种数据结构，在线程切换中使用。应用态线程由线程库定义，pthread 结构体，内核 TCB 的扩展。

线程本地存储 (Thread-Local Storage)：在不同线程中同名变量取不同值，相互不可见，如errno。

支持多线程需要修改 PCB，上下文、内核栈和退出、执行状态移入 TCB 中，PCB 保存一个 TCB 数组。则创建线程只需要初始化 TCB、记录线程和进程所属关系、准备运行环境即可（退出、合并同样任务更少）。Linux 中通过clone 创建线程，共

享地址空间等属性通过 flag 标记。应该避免用 fork 拷贝多线程程序。

处理器调度：最小单元为任务，或进程或线程。调度器决策下一个被执行的任务和执行任务使用的 CPU 以及执行时长。指标包括**周转时间**（任务进入系统到执行结束时间）、**响应时间**（进入系统到第一次给出输出时间）、**实时性**（任务 ddl 前完成）、可扩展（任务数量增加后仍能工作）、高资源利用率、公平、低开销。由于缺少信息，所以在多方面进行 trade-off（开销 vs 效果，优先级 vs 公平，能耗 vs 性能）。**First-Come-First-Served**：平均周转和响应时间太长。**Shortest Job First**：不公平，任务饿死。**Preemptive Scheduling**：通过时钟中断定期切换任务。**Round-Robin**：固定时间片轮询，公平、响应时间短，但是在各个任务时间差不多时周转时间非常长，时间片过长则变为 FCFS，过短则调度开销很大。**Priority Scheduling**：IO 密集型任务应该有更高优先级以获得更高资源利用率，FCFS 和 SJF 为优先级调度，而 Round-Robin 不是（可以通过优先队列实现）。需要动态优先级，否则低优先级任务饿死。其中一种方案：使用响应比作为优先级 Priority $= T_{\text{response}}/T_{\text{run}}$（视为 FCFS 和 SJF 的结合）。

Multi-Level Feedback Queue：假定工作场景变化频率不高，**(1)** 高优先级抢占低优先级，**(2)** 相同优先级使用 round-robin，**(3)** 被创建时假设任务短，分配最高优先级，**(4a)** 任务耗尽一个时间片后优先级-1，**(4b)** 耗尽前放弃时间片则优先级不变。对于 CPU 密集型任务（耗时长），优先级会逐渐降低，而 IO 密集型任务会在时间片耗尽前放弃 CPU，因此优先级保持不变。

- **问题**：长任务饿死，任务特征会变，如 CPU 密集会转变为 IO 密集，因此引入 **(5)** 经过一定时间所有任务优先级升为最高；无法应对恶意通过 IO 抢占 CPU 的任务，因此将 **(4a)** 和 **(4b)** 改为 **(4)** 累积任务使用的时间而非时间片，在满一个时间片后优先级降低。
- **参数**：优先级数量、每个队列时间片长短、优先级提升的时间间隔。高优先级用短时间片（提升响应时间），低优先级用长时间片（降低调度开销）。

Fair-Share Scheduling：ticket 表示份额，T 表示 ticket 总量。**彩票调度 (Lottery Scheduling)**：根据不同任务的 T 生成随机数 $R \in [0, T)$，根据 R 决定当前时间片执行哪个任务。**步幅调度 (Stride Scheduling)**：Stride $=$ MaxStride/ticket，步幅代表当前任务执行一个时间片所增加的 Pass，而每次总是选取 Pass 最小的任务执行。**多核调度**：不能使用全局运行队列，否则同一个线程在不同 CPU 上切换带来很大开销。每个 CPU 核心维护本地运行队列，将任务从高负载 CPU 迁移到低负载 CPU 上。需要一个良好定义的“负载”。**Affinity**：程序员通过接口指定任务使用的 CPU 核心。

进程间通信：linux 中使用 pipe 单向传输无格式数据，需要预设缓存大小。消息队列：用链表组织，支持异步通信。Signal：不需要查询，只需要等待信号（如 kill 命令即为 signal 实现）。共享内存：没有空间时发送者阻塞，没有通知机制浪费 CPU 轮询资源或时延较高，需要 OS 的支持。**Time-of-Check to Time-of-Use 问题**：发送者可以在完成检查但尚未使用的时间篡改数据。

简单 IPC：需要实现发送、接收、远程方法调用（例如应用程序调用文件系统打开文件）、远程方法返回。在进行发送、接收前需要先建立通信连接，即进程间信道。数据包括 Header 和 Payload。数据传递可以使用**共享内存**（定制能力强，无需额外拷贝，但是需要避免 TOCTOU 问题。解决方案为拷贝共享内存或修改页表映射，即发送后去除发送者的映射，建立接收者的映射，同时可以防止内存拷贝开销，但是需要刷新 TLB，因为内核不知道该进程在哪些 CPU 核心上执行过）或**操作系统内核接口**（抽象简单，安全性保证）。通知机制可以基于**轮询**（浪费 CPU 资源）或**控制流转移**（向操作系统发起请求，挂起调用者进程，执行被调用进程，再返回；由此降低 CPU 浪费，但是在多核情况下轮询效率更高）。控制流：**同步**（阻塞调用者）或**异步**。IPC 需要**超时**机制。通信连接：**直接通信**（显式建立点对点连接，如 pipe），**间接通信**（message queue，通过信箱通信）。大多数接口都为一个进程，通过 IPC 调用，因此需要命名服务作为一个单独的进程，建立看板帮助应用查询服务进程。

同步：在多个发送者情况下，通过检测 empty slot 进行的消息发送函数会导致数据竞争问题。**临界区 (Critical Section)**：任意时刻有且仅有一个线程可以进入该区域执行。实现方法为**同步原语 (Synchronization Primitives)**：操作系统提供的接口。

- **互斥锁 (Mutual Exclusive Lock)**：临界区可以通过lock; critical section; unlock; 实现，在没有锁的时候lock 循环。
- **条件变量**：增加资源利用率，主动进入睡眠，等待其他线程更新条件变量，操作系统唤醒等待该变量的线程，接口为cond_wait(cond* cond_var, lock* mutex)，在睡眠时释放锁，唤醒时上锁（此处释放锁、上锁需要交给操作系统完成，因为生产者释放锁之后消费者会先发信号，因此生产者无法等待到信号）。
- **信号量 (Semaphore)**：原先使用锁 + 条件变量 + 资源计数器实现同步修改一段内存。可以直接由信号量取代，使用sem_wait 接口即可。有两种操作P（消耗资源）和V（释放资源）。例如在生产者-消费者中，需要两个信号量 empty slots 和 filled slots。注意到信号量允许多少线程使用资源取决于初始资源量init_cnt，值为 1 时为二元信号量（由不同进程消耗、释放，但是互斥锁由同一个进程占用、释放），大于 1 时为计数信号量。
- **读写锁**：读者之间可以并行，读者与写者互斥，可以用于实现公告栏。读写锁**偏向性**：考虑有读者在临界区，然后新写者等待，接下来到来的另一个读者能否进入临界区。能则偏向读者，可能饿死写者；不能则偏向写者，公平性更好。

实现锁：lock 函数和unlock 函数都不是原子操作，因此仍然可能存在竞争。因此硬件提供**原子指令**。**Test-and-Set**：在 lock 中定义一个 flag，TestAndSet(int* ptr, int new) 原子指令将ptr 指针的值设置为new 并返回原本的值。**Compare-and-Swap**(x86)：CompareAndSwap(int* ptr, int expected, int new) 指令判断ptr 的值是否为expected，是则将指针置为new，返回指针的实际值。**Load-Linked 与 Store-Conditional**：Load-Linked(int* ptr) 指令读取ptr 存储的值，StoreConditional(int* ptr, int value) 指令在执行上一次LoadLinked 到当前指令中ptr 值不发生变化则进行 store 操作，返回 1 代表成功，否则返回 0。**Fetch-and-Add**：返回ptr 旧值，对ptr 加一，借此通过ticket 和turn 实现公平的排号锁（lock 函数申请一个ticket 然后等待自己的turn，unlock 只需要对turn 增加 1 即可）。

使用场景：使用互斥锁实现共享资源互斥访问，使用条件变量实现条件等待与唤醒，使用信号量实现多资源管理。负载均衡中空闲 CPU 允许窃取其他 CPU 的任务，本质上是对共享队列的互斥访问，只需要使用互斥锁即可。Mapper-Reducer 工作框架中，类似于线程等待与唤醒，在 Mapper 结束时唤醒 Reducer 进行统计；也可以使用信号量，等待 Mapper 的返回数量达到一定值进行 Reduce。

死锁：条件为互斥访问、持有一部分资源并等待另一部分，且资源非抢占，循环等待。例如 A 进程用 A 锁和 B 锁，B 进程用 B 锁和 A 锁，并发时卡死。解决方法：出问题再处理、锁的设计中预防、运行中防止。

- 检测死锁：已分配的资源指向进程，进程指向等待的资源，找出资源和进程的环即为死锁。
- 预防死锁：使用代理线程，或不允许持有锁的同时等待其他锁（trylock 接口，不阻塞，立即返回成功或失败），或对锁编号，要求必须先拿小锁再拿大锁，或允许资源抢占（让原先持有锁的线程回滚到持有锁之前的状态）。
- 运行时避免：**银行家算法**，线程获取资源需要管理员同意，管理员进行沙盘模拟。对于一组线程 $\{P_1, ..., P_n\}$，安全状态为至少存在一个执行序列使得所有资源需求满足，否则为非安全状态。假设 m 个资源，数据结构有 Available[m]，MaxDemand[n][m]，Allocated[n][m]，Need[n][m]（Max=Allocated+Need）。安全状态下总是执行 Need[i]<Available 的进程 i 即可，非安全状态下阻塞进程即可。

文件系统

文件：有名字且持久化的数据。文件系统：提供操作文件的 API。虚拟文件系统：帮助操作系统管理不同的真实文件系统（如 Ext4、NTFS）。与内存相同，假设每个磁盘块为 4KB。

Index Node 文件系统：为了支持不同大小的文件，引入 inode，包括元数据的 size 和各个磁盘块号。对于读写操作（给定 inode 和 offset）只需要根据 offset 计算出对应磁盘块号即可。在整个磁盘上，存有**超级块**（整个系统的元数据），**存储块分配信息**，**inode 分配信息**（inode 表中每一项是否被分配），**inode 表**，因此支持的总文件数有上限。**多级 inode**：与单级页表类似，单级 inode 消耗空间太大（例如可以用 12 个直接指针、3 个间接指针、1 个二级间接指针表示一个 $48K + 6M + 1G$ 的文件），因此引入二级索引块指向索引块指向数据块（索引快都存储在数据区）。**文件名**：inode 没有文件名，并且名字依赖于 inode 表的位置，在移动数据时需要改变名字，因此为了用字符串为文件名，通过**目录**建立字符串到 inode 号的映射（因此文件名不是文件的一部分也不是元数据的一部分，是目录的一部分）。**目录**中存储的目录项即为字符串向 inode 的映射，大小只取决于每个目录项（文件名）的长度。**根目录**固定为 1 号 inode。

硬链接：ln target link-name 命令，此后写入link-name 能够在target 中得到反馈，即创建文件名link-name 到target 的 inode 号的映射。unlink 操作可以解除文件名到 inode 号的绑定关系，最后一个绑定关系解除时 inode 和对应数据块被放入 free-list，因此需要 reference counter。硬链接不允许指向目录，否则会出现环（除了. 和..），导致 refcnt 失效。

软链接：ln -s target link-name 命令，得到的link-name 的 inode 号和target 不同，文件中只记录了target 的路径。对于不同文件系统、不同命名空间的 link，无法使用 inode 映射，因此使用软链接作为第三种 inode 类型（例如 windows 快捷方式）。对自身进行软链接会因为递归导致报错。

文件系统 API：磁盘中元数据还包括时间戳（最后一次访问、最后一次修改等），在内存中系统维护了一个file_table（存储 inode 号，文件游标和 refcnt），每个进程维护了一个fd_table 对应每个fd 在file_table 中的索引。**文件游标 Cursor**：记录下一次

操作的位置（用seek 修改），在父进程将fd 传给子进程时共享游标，而不同进程的fd 不同，游标也不共享。对一个被其他进程打开的文件进行写操作，如删除可能导致其他进程错误，因此在前一个进程 close 之后再执行删除（windows 上不能删除被打开的文件）。

性能：open 和read 中的磁盘访问需要从 inode 开始不断遍历，并且需要更新时间戳执行写操作，因此性能很糟糕。**页缓存：**将磁盘中的 inode 表等数据结构复制到内存中作为缓存，提升性能表现（实际上硬盘本身还有缓存），通过 **Vnode** 实现（可以用 mmap，速度比 read 要快，read 是系统调用，需要特权级切换）。因此从缓存写回磁盘需要应用程序手动调用fsync 接口。

mmap 实现：内核地址空间 VMA 中有 Vnode 到文件的映射。用户调用 mmap 之后操作系统将用户地址空间和页缓存映射到同一段物理内存。用户态触发 page fault 之后操作系统根据 Vnode 的元数据访问磁盘，将数据搬到页缓存中。

崩溃一致性：例如在一次 append 中，需要修改块分配信息、修改 inode 表、最后修改数据块。需要确保在崩溃后保持一致性。故障发生：六种可能，三次写成功一次、两次各三种。用户期望不会因为崩溃导致文件不变量修改、一段时间前的操作被保存到磁盘中、并且执行顺序没有错误。

- 同步元数据写 +fsck：每次元数据的写入立即进行sync()，并且在非正常重启时检查磁盘。**(1)** 检查 superblock，损坏则直接用备份。**(2)** 扫描存储块和 inode 的 bitmap。**(3)** 检查 inode 状态，清除错误的 inode。**(4)** 检查 inode 链接数量，若 inode 存在且不在任何目录则放到 lost-and-found 文件夹中。**(5)** 对于指向同一个磁盘块的两个 inode，除非一个明显有问题否则进行复制。**(6)** 检查超出磁盘空间的磁盘块 ID。**(7)** 检查目录，确保. 和.. 为头部，链接数只有一个，且没有相同文件名。问题在于太慢。
- **日志 (Journaling)**：在磁盘预留空间，先把修改记录（如怎么修改 bitmap）到日志中再进行修改。仅在所有修改都记录完毕后提交日志成功才修改数据和元数据。修改完成后删除日志。

  Ext4 中，三种模式：**Data mode**（所有数据写入日志，日志量大但是一致性保证好）；**Ordered mode**（日志区仅包含元数据内容，而数据直接写入，写入完成后更新元数据日志，最后更新元数据，因此数据正确性不受保证）；**Writeback mode**（不确保日志和数据之间的顺序）。

  Ordered mode 中，先写入数据和元数据的日志，进行 Flush；然后进行元数据日志的提交，确保元数据日志完整，再进行 Flush。最后在空闲时间写入元数据即可。