Linux 进程、线程与调度

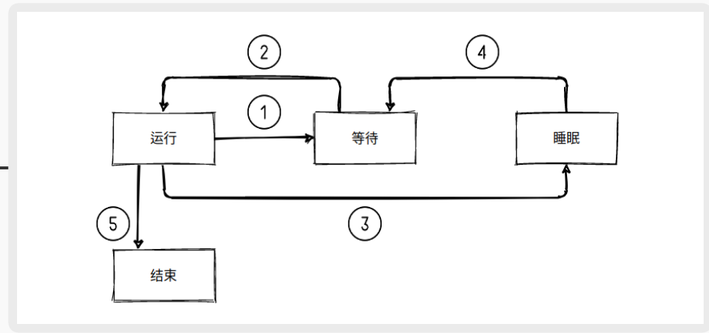
关于调度

进程与线程

- 在 Linux 操作系统中，不区分进程与线程，或者说，进程的创建和线程的创建本质上是一样的，均调用 do_fork() 系统函数创建
- 进程的创建
  - 使用 fork() 或者是 vfork() 均可以创建出新的进程，创建出来的进程一般称之为子进程
  - 父进程和子进程拥有相同的程序文本段，但是却各自拥有不同的栈段、数据段以及堆段。尽管系统采用了 copy-on-write，即写时复制的方式来延迟内存内容的复制，但仍然有复制开销
- 系统调用: clone()
  - int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg, ...)
  - 在该场景下，我们仅关心 flags 这个参数，将直接揭露出线程和进程的根本区别
  - NPTL 线程使用标志位 — CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD | CLONE SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE SYSVSEM
  - fork 创建进程使用标志位 — SIGCHLD
  - 我们简单地解释一些一些重要的标志位
    - CLONE_VM — 父、子进程共享虚拟内存
    - CLONE_FILES — 父、子进程共享打开文件描述符表
    - CLONE_SIGHAND — 父、子进程共享对信号的处置设置
- 因此，进程和线程其实都是内核调度实体（KSE），只是对一些系统资源，例如虚拟内存、打开文件描述符、对信号的处置、进程 ID 等资源的共享程度不同而已

Base

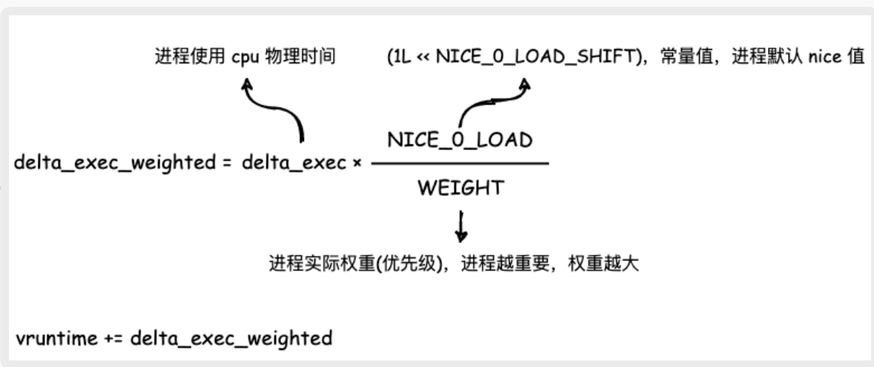
- 这里有必要对 Linux 进程/线程调度做出一个解释，这对理解互斥量、条件变量、sleep_for 等非常有帮助
- 进程状态及其状态变化



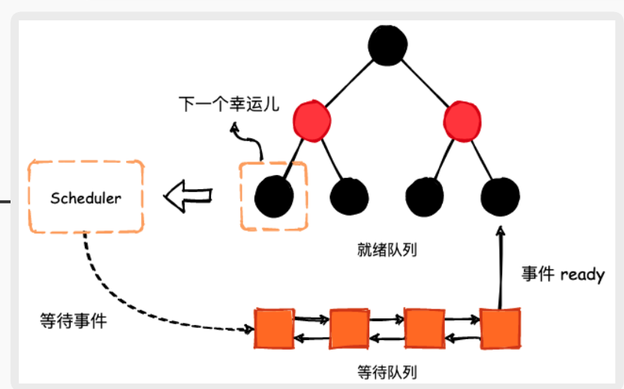
  - 等待状态 — 此时进程一切准备就绪，只差调度器调度该进程，并给与 CPU 资源执行，这些进程被保存在就绪队列中
  - 睡眠状态 — 处于睡眠状态的进程在等待外部事件的发生，例如 I/O 操作的数据抵达，创建的定时器到期等等，处于睡眠状态的进程永远不会被调度器进行选择并执行。当期望的事件到达后，进程由睡眠状态更改为等待状态，等待调度器的下一次选择。
- 因此，当一个进程/线程被阻塞时，将不会占用 CPU 资源。当期望的事件发生时，内核将保存在等待队列中的进程移动至就绪队列中，等待调度器调度
- std::this_thread::sleep_for(2s)
  - 1️⃣ 由用户态切换至内核态，调用系统 sleep函数，并创建定时器
  - 2️⃣ 该系统调用将调用 wait 方法，使得当前进程/线程进入等待队列
  - 3️⃣ 定时器到期并发出通知，内核将该进程/线程由等待队列中移至就绪队列中，准备执行

进程调度

- 抢占与非抢占
  - 抢占式任务调度是指，内核来决定什么时候停止一个进程的运行，以便其它进程能够得到运行机会。从进程的角度来看，自己的 CPU 被内核抢走并被其它进程所占有了，因而称为抢占
  - 非抢占则是指除非进程主动让出 CPU，否则它就能一直执行下去。显然，在多任务系统中，非抢占式任务调度将带来极大的不稳定性
- CFS
  - CFS 即 Completely Fair Scheduler，完全公平调度器，为 Linux 默认使用的调度器。不再跟踪进程的睡眠时间，也不去试图区分是否是交互式进程，将所有的进程统一对待，这就是完全公平的含义
  - CFS 并没有直接对进程进行优先级分配，而是使用 vruntime 来记录进程的虚拟执行时间。之所以是虚拟执行时间，是因为该值是通过进程的优先级（如 nice 值）与实际进程执行时间加权所得到的一个值，以 ns 为单位



    - 可以粗略认为进程的 vruntime 是如左图计算的
    - 也就是说，delta_exec_weighted 其实是一个逆向权重值，当两个进程的物理执行时间相同时，优先级更高的进程将会得到更低的delta_exec_weighted 值，因此累加的 vruntime 会更小
    - 此外，进程的默认 nice 值为 0，进程的权重将换算成 120，NICE_0_LOAD 相等。也就是说，此时进程的虚拟时间和物理时间相等
  - 关于 nice 值 — 😀 nice 值是指一个进程的友好度，一个进程的 nice 值越高，表示该进程越友好，就更乐意把 cpu 让给别人。换而言之，nice 值越大，进程优先级越低，其取值为 [-20, 19]
  - 当我们知道了 vruntime 的计算方式之后，就可以知道一个默认优先级的进程实际运行 200ms，其 vruntime 也是 200 ms。一个更高优先级的进程运行 200 ms，其 vruntime 的值将小于 200 ms。同理，一个更低优先级的进程运行 200 ms 后，其 vruntime 将大于 200 ms。
  - 当决定下一个调度进程时，调度器将选择最小虚拟运行时间的任务
    - 以具有相同优先级的 I/O 密集型任务和 CPU 密集型任务为例
    - I/O 密集型任务通常在运行很短的时间以后就开始等待 I/O 事件并让出 CPU。而 CPU 密集型任务只要能拿到 CPU，就可能一直运行
    - 因此，一段时间以后，I/O 密集型任务的 vruntime 将小于 CPU 密集型任务的 vruntime，从而将拥有更高的调度优先级。这是合理的，因为该 I/O 任务很可能是交互程序，需要有更低的响应延迟
  - 为了能够更快的找到具有最小 vruntime 的进程，内核使用 RBTree 来保存已就绪的进程，也就是我们上面提到的就绪队列



- Linux 也实现了实时调度。采用 SCHED_FIFO 或 SCHED_RR 实时策略来调度的任务，与普通（非实时的）任务相比，具有更高的优先级

进程与线程的区别

- 前面我们已经提到了，进程和线程最大的区别就在于资源共享的层级不同。父进程与子进程之间仅共享程序文本段，而同一个进程下的所有线程却共享数据段、堆段、文件描述符等
- 基于上述的不同，我们可以推断出
  - 1️⃣ 线程间的通信更容易，因为拥有着同一块数据段，因而可以使用全局变量进行通信
  - 2️⃣ 线程的创建速度要高于进程，因为"数据重合度"更高，子线程只需要拥有自己的堆栈即可
  - 3️⃣ 线程之间的上下文切换也要比进程间的上下文切换更快
- I/O 密集型与 CPU 密集型程序
  - I/O 密集型
    - I/O 密集型的程序绝大部分时间都是在等待网络数据的到达，这个过程与 CPU 无关，因而我们可以让出 CPU 资源，给那些真正需要 CPU 资源的程序
    - 所以，I/O 密集型的程序通常使用多线程实现，从而获得远超于 CPU 核心数的线程数量。若使用多进程，一来进程上下文切换开销比较多，二来创建进程对资源的消耗要多于线程
  - CPU 密集型
    - CPU 密集型的程序是渴望得到更多的 CPU 资源的，因此，我们的子进程/子线程数量应与 CPU 核心数相同，再多就没有意义了。
    - 具体到底使用多进程还是多线程实现，和设计角度有关。多进程在信号处理上有着天然的优势，并且不会争抢虚拟内存空间
- 啰嗦版
  - 进程是资源分配的最小单位，而线程则是系统调度的最小单位
    - 例如 Linux Cgroups (Linux Control Group) 只能限制一个进程组所使用的 CPU、内存、网络等资源的上限
    - 正如我们并没有听说 Docker 能够限制某一个线程所使用的资源一样
  - 进程下多个线程间共享虚拟内存、文件描述符、信号处理方式等资源，但进程间拥有独立的虚拟内存、文件描述符与信号处理等资源。在创建线程时，由于虚拟内存、文件描述符等资源共享，故不需要进行额外的内存复制。当创建进程时，OS 使用 Copy-On-Write 的方式延迟对内存内容的复制，但仍需要一些额外的开销。因此，尽管子进程可能在创建后立即调用 ` exec()` 执行新的程序，其创建速度依然慢于线程的创建速度
  - 通信方式不同。线程间可通过全局变量、互斥锁或者是条件变量来进行通信，但进程间只能使用管道、OS 提供的共享内存等进行通信，需要投入更多的资源
  - 对信号的支持不同。由于线程是"依附"在进程之上的，因此，同一个进程下的多个线程在使用信号时会有问题，无法准确的将信号传递至某一个具体的线程，例如 Python 中的信号不支持在多线程环境中使用。而对于进程而言，我们可通过 ` kill()` 或者是 ` raise()` 的方式将信号传递给具体的进程，不管它是父进程还是子进程
  - 上下文切换速度不同。因为线程间共享了虚拟内存、文件描述符等诸多信息，因此 OS 只需要在上下文保存线程的堆栈、寄存器等少量信息，所以其切换速度要高于进程间的上下文切换
- 最后，Linux 中的进程和线程均使用 do_fork() 实现，它们之间最大的区别就在于共享资源的不同