链滴

进程，系统性能和计划任务 1

作者：Carey

原文链接：https://ld246.com/article/1609829765594

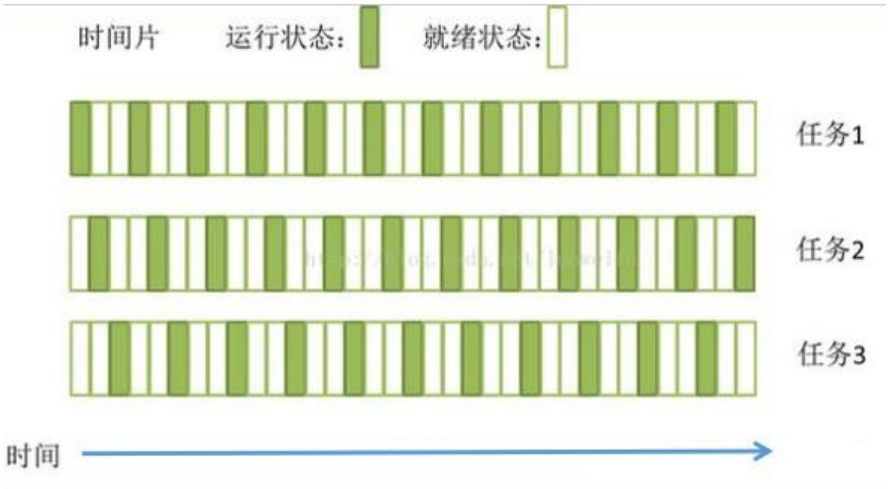
来源网站：链滴

内容概述

- 进程相关概念
- 进程工具
- 系统性能相关工具
- 计划任务

1 进程和内存管理

内核功用：进程管理、内存管理、文件系统、网络功能、驱动程序、安全功能等



1.1 什么是进程

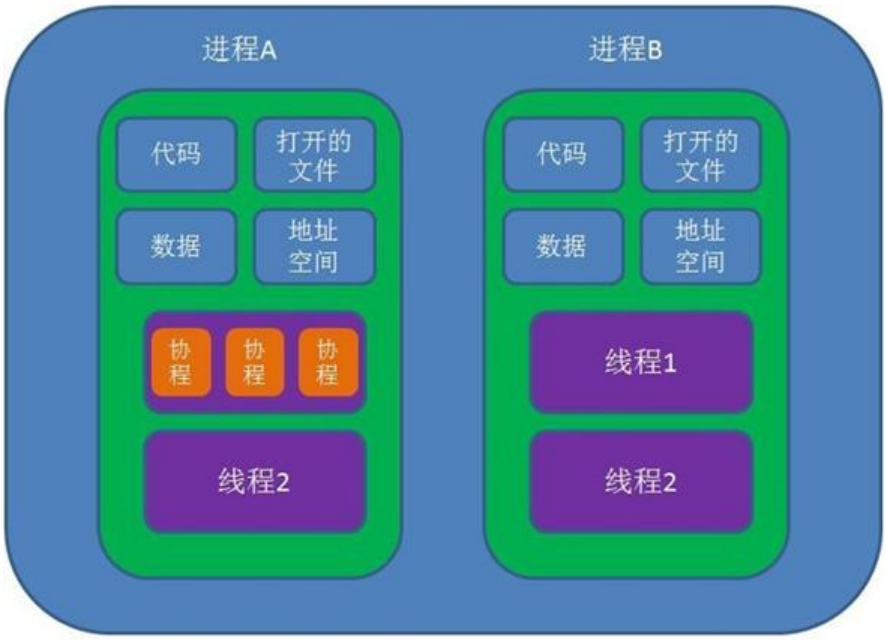
Process：运行中的程序的一个副本，是被载入内存的一个指令集合，是资源分配的单位

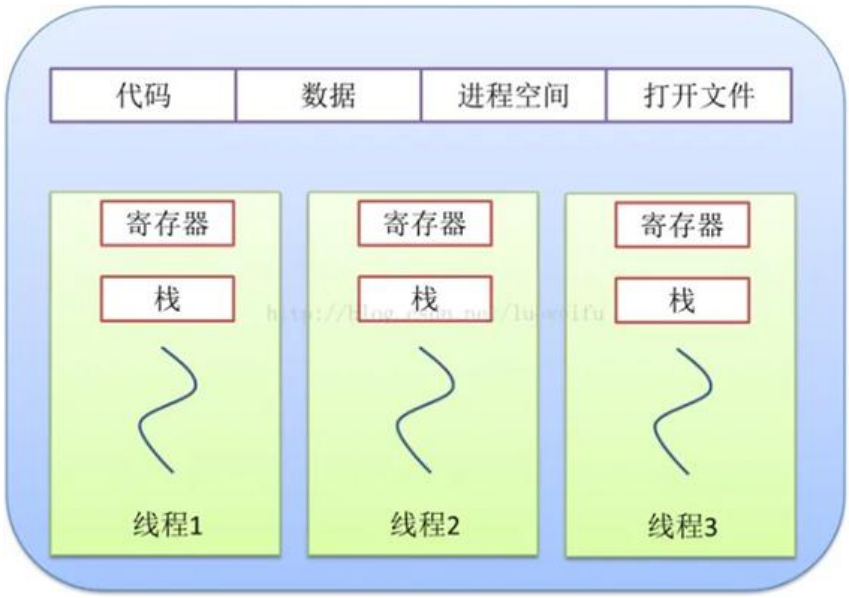
- 进程ID(Process ID，PID)号码被用来标记各个进程
- UID、GID和SELinux语境决定对文件系统的存取和访问权限
- 通常从执行进程的用户来继承
- 存在生命周期

进程创建：

- init：第一个进程，从Centos7以后为systemd
- 进程：都由其父进程创建，fork()，父子关系，Cow：Copy On Write

进程，线程和协程





1.1.1 进程

进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分和调度的一个独立单位，是应用程序运行的载体。进程是一种抽象的概念，从来没有统一的标志定义

进程的组成：

进程一般由程序、数据集合和进程控制块三部分组成。

程序用于描述进程要完成的功能，是控制进程执行的指令集：

数据集合是程序在执行时所需要的数据和工作区：

程序控制块(Program Control Block，简称PCB)，包含进程的描述信息和控制信息，是进程存在的一标志。

进程具有的特征：

动态性：进程是程序的一次执行过程，是临时的，有生命期的，是动态产生，动态消亡的：

并发性：任何进程都可以同其他进程一起并发执行

独立性：进程是系统进行资源分配和调度的一个独立单位：

结构性：进程是由程序、数据和进程控制块三部分组成。

1.1.2 线程

在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的小单位。任务调度采用的是时间片轮转的抢占式调度方式，而进程是任务调度的最小单位，每个进程各自独立的一块内存，使得各个进程之间内存地址相互隔离。后来，随着计算机的发展，对CPU的要越来越高，进程之间的切换开销较大，已经无法满足越来越复杂的程序的要求了。于是就发明了线程。

线程是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本位。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间(也就是所在进程的内存空间。一个标准的线程由线程ID、当前指令指针(PC)、寄存器和堆栈组成。而进程是由内存空间(代码、数、进程空间、打开的文件)和一个或多个线程组成。

进程与线程的区别

线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位：

一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线：

进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段、数据集、堆等)一些进程级的资源(如打开文件和信号)，某个进程内的线程在其他进程不可见：

调度和切换：线程上下文切换比进程上下文切换要快的多

1.1.3 协程

协程，英文Coroutines，是一种基于线程之上，但又比线程更加轻量级的存在，这种由程序员自己写序来管理的轻量级线程叫做(用户空间线程)，具有对内核来说不可见的特性。

因为是自主开辟的异步任务，所以很多人也更喜欢叫它们纤程(Fiber)，或者绿色线程(GreenThread)正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。

协程的目的：

在传统的J2EE系统中都是基于每个请求占用一个线程去完成完整的业务逻辑(包扣事务)。所以系统的

吐能力取决于每个线程的操作耗时。如果遇到很耗时的I/O行为，则整个系统的吞吐立刻下降，因为个时候线程一直处于阻塞状态，如果线程很多的时候，会存在很多线程处于空闲状态(等待该线程执行才能执行)，造成了资源应用不彻底

最常见的例子就是JDBC(他是同步阻塞的)，这也是为什么很多人都说数据库是瓶颈的原因。这里的耗其实是让CPU一直在等待I/O返回，说白了线程根本没有利用CPU去做运算，而是处于空转状态。而外过多的线程，也会带来更多的ContextSwitch开销。

对于上述问题，现阶段行业里的比较流行的解决方案之一就是单线程加上异步回调。其代表派是node.s以及java里的新秀vert.x。

而协程的目的就是当出现长时间的I/O操作时，通过让出目前的协程调度，执行下一个任务的方式，消除ContextSwitch上的开销。

协程的特点

线程的切换由操作系统负责调度，协程由用户自己进行调度，因此减少了上下文切换，提高了效率。

线程的默认Stack大小是1M,而协程更轻量，接近1K。因此可以在相同的内存中开启更多的协程

由于在同一个线程上，因此可以避免竞争关系而使用锁

适用于被阻塞的，且需要大量并发的场景。但不适用于大量计算的多线程，遇到此种情况，更好实用程去解决

协程的原理：

当出现IO阻塞的时候，由协程的调度器进行调度，通过将数据流立刻yield掉(主动让出)，并且记录当栈上的数据，阻塞完后立刻在通过线程恢复栈，并把阻塞的结果放到这个线程上去跑，这样看上去好跟写同步代码没有任何差别，这整个流程可以称为coroutine，而跑在由coroutine负责调度的线程称Fiber。比如Golang里的go关键字其实就是负责开启一个Fiber，让func逻辑跑在上面

由于协程的暂停完全由程序控制，发生在用户态上：而线程的阻塞状态是由操作系统内核来进行切换发生在内核态上。因此，协程的开销远远小于线程的开销，也就没有了ContextSwitch上的开销。

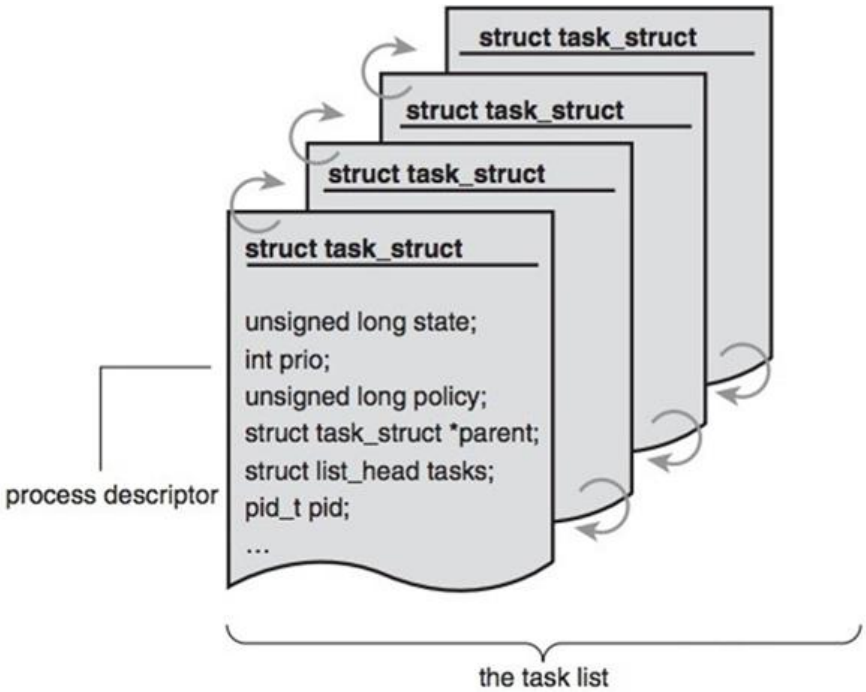
1.1.4 协程和线程的比较

比较项	线程	协程
占用资源	初始单位为1MB,固定不可变	初始一般为 2KB，可随需要而增大
调度所属	由 OS 的内核完成	由用户完成
切换开销	涉及模式切换(从用户态切换到内核态)、16个寄存器、PC、SP...等寄存器的刷新等	只有三个寄存器的值修改 - PC / SP / DX.
性能问题	资源占用太高，频繁创建销毁会带来严重的性能问题	资源占用小,不会带来严重的性能问题
数据同步	需要用锁等机制确保数据的一直性和可见性	不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

1.1.5 查看进程中的线程

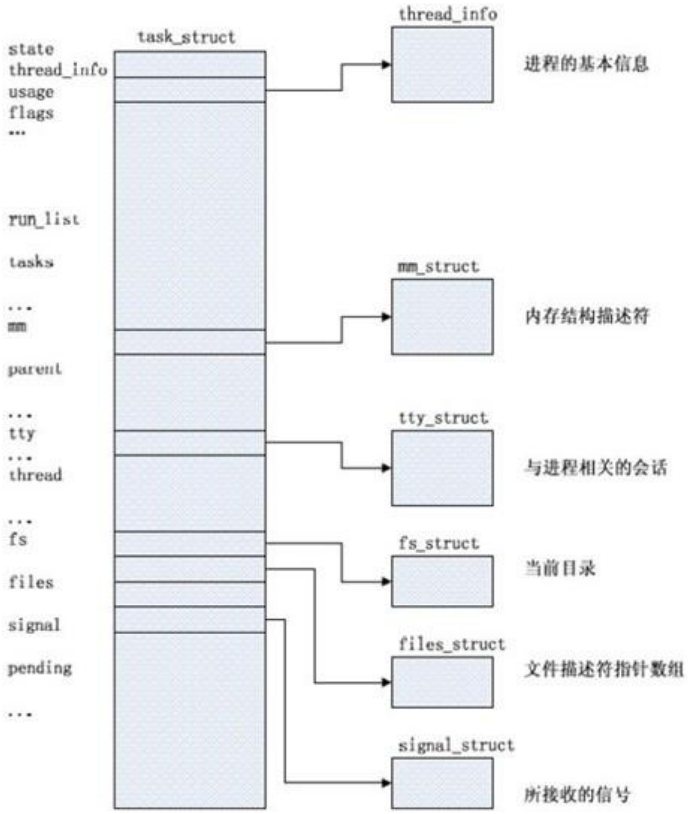
```
[17:56:04 root@centos8 ~]#grep -i threads /proc/945/status
Threads:    6
```

1.2 进程结构



内核把进程存放在叫做任务队列(task list)的双向循环链表中

链表中的每一项都是类型为task_struct，称为进程控制块(Processing Control Block),PCB中包含一具体进程的所有信息



进程控制块PCB包含信息：

- 进程id、用户id和组id
- 程序计数器
- 进程的状态(由就绪、运行、阻塞)
- 进程切换时需要保存和恢复的CPU寄存器的值
- 描述虚拟地址空间的信息
- 描述控制终端的信息
- 当前工作目录
- 文件描述符表，包含很多指向file结构体的指针
- 进程可以使用的资源上限(ulimit -a 命令可以查看)
- 输入输出状态：配置进程使用I/O设备

1.3 进程相关概念

Page Frame：页框，用存储页面数据，存储Page 4K

```
[18:46:50 root@centos8 ~]#getconf -a | grep -i size
PAGESIZE                           4096
PAGE_SIZE                          4096
SSIZE_MAX                          32767
_POSIX_SSIZE_MAX                   32767
_POSIX_THREAD_ATTR_STACKSIZE       200809
FILESIZEBITS                       64
POSIX_ALLOC_SIZE_MIN               4096
```

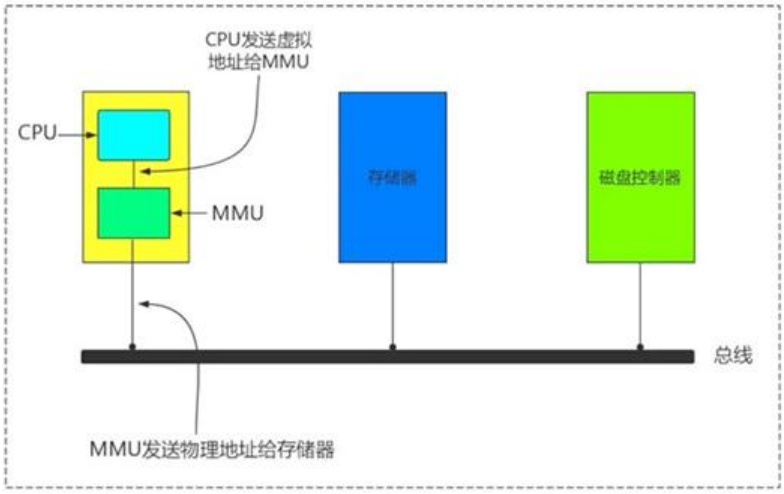
```
POSIX_REC_INCR_XFER_SIZE
POSIX_REC_MAX_XFER_SIZE
POSIX_REC_MIN_XFER_SIZE          4096
LEVEL1_ICACHE_SIZE              32768
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE              32768
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE              262144
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE             6291456
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE              0
LEVEL4_CACHE_LINESIZE              0
```

1.3.1 物理地址空间和虚拟地址空间

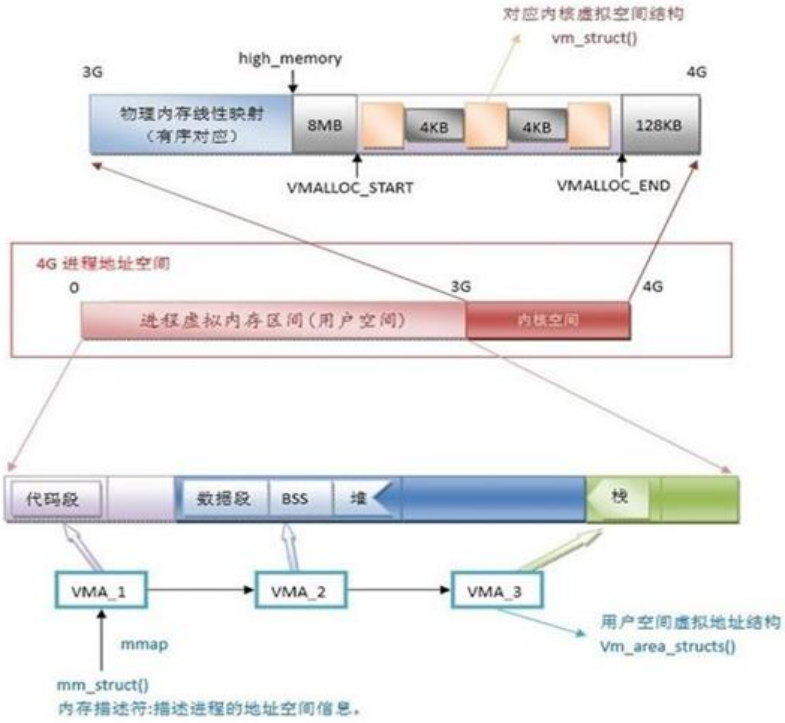
MMU：Menmory Management Unit 负责虚拟地址转换为物理地址

程序在访问一个内存地址指向的内存时，CPU不是直接把这个地址传送到内存总线上，而是被传送到MU，然后把这个内存地址映射到实际的物理内存地址上，然后通过总线再去访问内存，程序操作的址称为虚拟内存地址

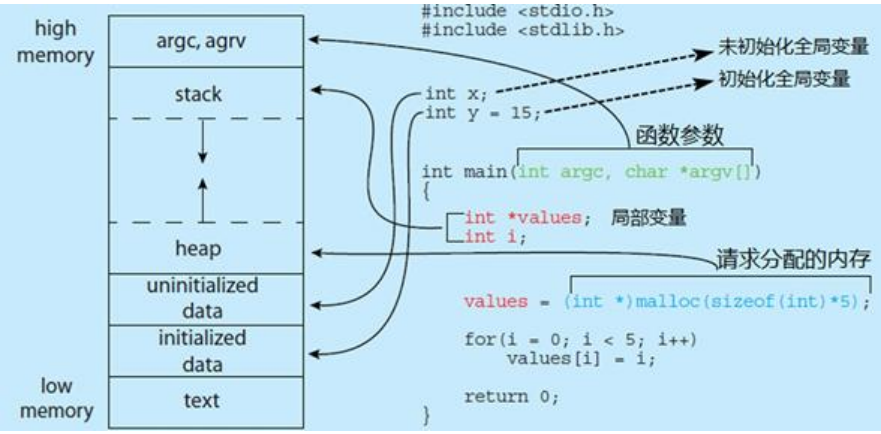
TLB：Translation Lookaside Buffer翻译后备缓冲区，用于保存虚拟地址和物理地址映射关系的缓存



1.3.2 用户和内核空间



1.3.3 C代码和内存布局之间的对应关系



每个进程都包扣5种不同的数据段

● 代码段：用来存放可执行文件的操作指令，也就是说是它是可执行程序在内存中的镜像。代码需要止在运行时被非法修改，所以只允许读取操作，而不允许写入(修改)操作——它是不可写的

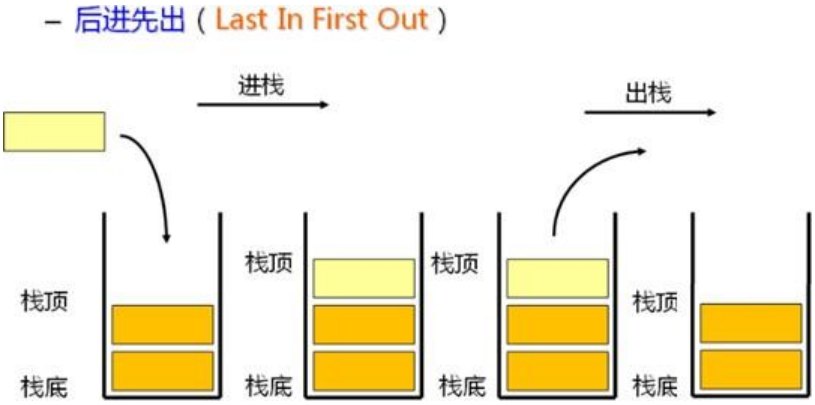
● 数据段：用来存放可执行文件中已初始化全局变量，换句话说就是存放程序静态分配的变量和全局量

● BSS段：Block Started by Symbol的缩写，意为“以符号开始的块，BSS段包含了程序中未初始化全局变量，在内存中bss段全部置零

● 堆(heap)：存放数组和对象，堆是用于存放进程运行中被动态分配的内存段，他的大小并不固定，动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上(堆被扩张；当利用free等函数释放内存时，被释放的内存从堆中被剔除(堆被缩减)

● 栈(stack):栈是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}“中定义的变量(但不括static声明的变量，static意味着在数据段中存放变量)。除此以外，在函数被调用时，其参数也会压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放会栈中。由于栈的后进先特点，所以栈特别方便用来保存/恢复调用现场。可以把栈堆看成一个寄存器、交换临时数据的内存区

喝多了吐就是栈，吃多了拉就是队列：栈先进后出，队列先进先出



1.3.4 进程使用内存问题

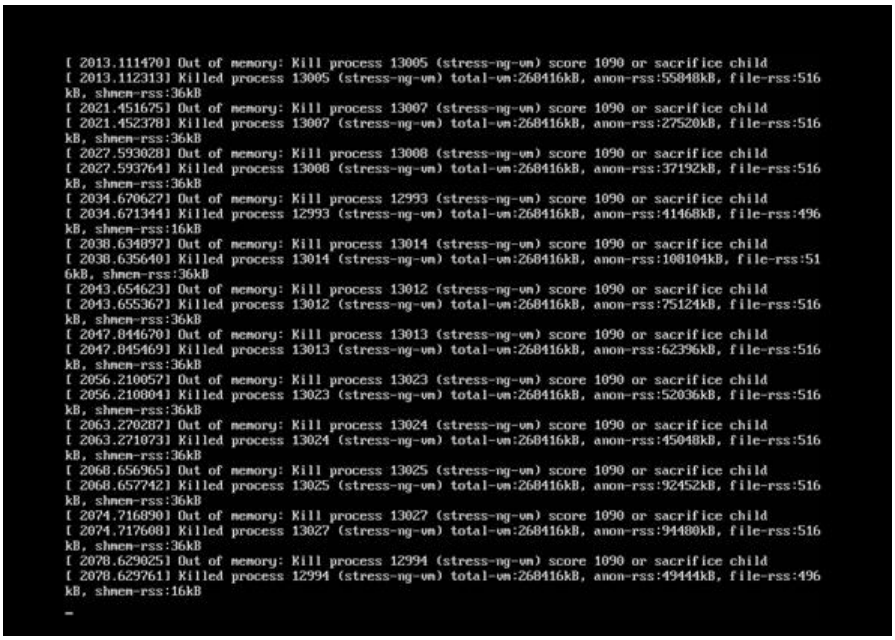
1.3.4.1 内存泄露：Memory Leak

指程序中用malloc或new申请了一块内存，但是没有用free或delete将内存释放，导致这块内存一直于占用状态

1.3.4.2 内存溢出：Memory Overflow

指程序申请了10M的空间，但是在这个空间写入10M以上字节的数据，就是溢出。

1.3.4.3 内存不足：OOM



OOM 即 Out Of Memory， “内存用完了”,在情况在java程序中比较常见。系统会选一个进程将之死， 在日志messages中看到类似下面的提示

Jul 10 10:20:30 kernel: Out of memory: Kill process 9527 (java) score 88 or sacrifice child

当JVM因为没有足够的内存来为对象分配空间并且垃圾回收器也已经没有空间可回收时，就会抛出这

error，因为这个问题已经严重到不足以被应用处理）。

原因：

- 给应用分配内存太少：比如虚拟机本身可使用的内存（一般通过启动时的VM参数指定）太少。
- 应用用的太多，并且用完没释放，浪费了。此时就会造成内存泄露或者内存溢出。

使用的解决办法：

1，限制java进程的max heap，并且降低java程序的worker数量，从而降低内存使用2，给系统增加swap空间

设置内核参数（不推荐），不允许内存申请过量：

```
echo 2 > /proc/sys/vm/overcommit_memory
echo 80 > /proc/sys/vm/overcommit_ratio
echo 2 > /proc/sys/vm/panic_on_oom
```

说明：

Linux默认是允许memory overcommit的，只要你来申请内存我就给你，寄希望于进程实际上用不那么多内存，但万一用到那么多了呢？Linux设计了一个OOM killer机制挑选一个进程出来杀死，以出部分内存，如果还不够就继续。也可通过设置内核参数 vm.panic_on_oom 使得发生OOM时自动启系统。这都是有风险的机制，重启有可能造成业务中断，杀死进程也有可能导致业务中断。所以

Linux 2.6之后允许通过内核参数 vm.overcommit_memory 禁止memory overcommit。

vm.panic_on_oom 决定系统出现oom的时候，要做的操作。接受的三种取值如下：

0 - 默认值，当出现oom的时候，触发oom killer
1 - 程序在有cpuset、memory policy、memcg的约束情况下的OOM，可以考虑不panic，而是启OOM killer。其它情况触发 kernel panic，即系统直接重启
2 - 当出现oom，直接触发kernel panic，即系统直接重启

vm.overcommit_memory 接受三种取值:

0 – Heuristic overcommit handling. 这是缺省值，它允许overcommit，但过于明目张胆的overcommit会被拒绝，比如malloc一次性申请的内存大小就超过了系统总内存。Heuristic的意思是“试 探的”，内核利用某种算法猜测你的内存申请是否合理，它认为不合理就会拒绝overcommit。
1 – Always overcommit. 允许overcommit，对内存申请来者不拒。内核执行无内存过量使用处理使用这个设置会增大内存超载的可能性，但也可以增强大量使用内存任务的性能。
2 – Don’t overcommit. 禁止overcommit。 内存拒绝等于或者大于总可用 swap 大小以及overcommit_ratio 指定的物理 RAM 比例的内存请求。如果希望减小内存过度使用的风险，这个设置就是最的。

Heuristic overcommit算法：

单次申请的内存大小不能超过以下值，否则本次申请就会失败。

free memory + free swap + pagecache的大小 + SLAB

vm.overcommit_memory=2 禁止overcommit,那么怎样才算是overcommit呢?

kernel设有一个阈值，申请的内存总数超过这个阈值就算overcommit，在/proc/meminfo中可以看到这个阈值的大小：

```
[18:46:58 root@centos8 ~]#grep -i commit /proc/meminfo
CommitLimit:     2584584 kB
Committed_AS:     297536 kB
```

CommitLimit 就是overcommit的阈值，申请的内存总数超过CommitLimit的话就算是overcommit 此值通过内核参数vm.overcommit_ratio或vm.overcommit_kbytes间接设置的，公式如下：

```
CommitLimit = (Physical RAM * vm.overcommit_ratio / 100) + Swap
```

vm.overcommit_ratio 是内核参数，缺省值是50，表示物理内存的50%。如果你不想使用比率，也以直接指定内存的字节数大小，通过另一个内核参数 vm.overcommit_kbytes 即可；

如果使用了huge pages，那么需要从物理内存中减去，公式变成：

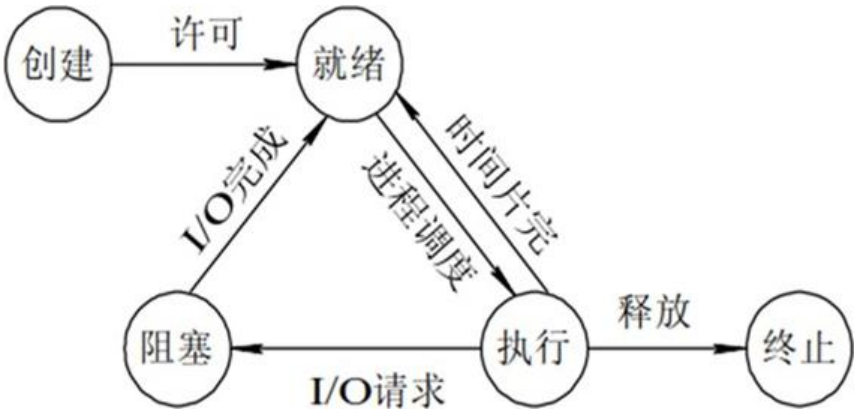
```
CommitLimit = ([total RAM] – [total huge TLB RAM]) * vm.overcommit_ratio / 100 + swap
```

/proc/meminfo中的 Committed_AS 表示所有进程已经申请的内存总大小，（注意是已经申请的不是已经分配的），如果 Committed_AS 超过 CommitLimit 就表示发生了 overcommit，超出越表示overcommit 越严重。Committed_AS 的含义换一种说法就是，如果要绝对保证不发生OOM (ou of memory) 需要多少物理内存。

范例：

```
[root@centos8 ~]#cat /proc/sys/vm/panic_on_oom 0
[root@centos8 ~]#cat /proc/sys/vm/overcommit_memory 0
[root@centos8 ~]#cat /proc/sys/vm/overcommit_ratio 50
[root@centos8 ~]#grep -i commit /proc/meminfo
CommitLimit:    3021876 kB
Committed_AS:     340468 kB
```

1.4 进程状态



进程的基本状态

● 创建状态：进程在创建时需要申请一个空白PCB(process control block进程控制块)，向其中填写制和管理进程的信息，完成资源分配。如果创建工作无法完成，比如资源无法满足，就无法被调 度行，把此时进程所处状态称为创建状态

● 就绪状态：进程已准备好，已分配到所需资源，只要分配到CPU就能够立即运行执行状态：进程处就绪状态被调度后，进程进入执行状态

● 阻塞状态：正在执行的进程由于某些事件（I/O请求，申请缓存区失败）而暂时无法运行，进程受阻塞。在满足请求时进入就绪状态等待系统调用

● 终止状态：进程结束，或出现错误，或被系统终止，进入终止状态。无法再执行

状态之间转换六种情况

● 运行——>就绪：1，主要是进程占用CPU的时间过长，而系统分配给该进程占用CPU的时间是有的； 2，在采用抢先式优先级调度算法的系统中,当有更高优先级的进程要运行时，该进程就被迫让出PU， 该进程便由执行状态转变为就绪状态

● 就绪——>运行：运行的进程的时间片用完，调度就转到就绪队列中选择合适的进程分配CPU

● 运行——>阻塞：正在执行的进程因发生某等待事件而无法执行，则进程由执行状态变为阻塞状态如 发生了I/O请求

● 阻塞——>就绪:进程所等待的事件已经发生，就进入就绪队列

以下两种状态是不可能发生的：

● 阻塞——>运行：即使给阻塞进程分配CPU，也无法执行，操作系统在进行调度时不会从阻塞队列行挑选，而是从就绪队列中选取

● 就绪——>阻塞：就绪态根本就没有执行，谈不上进入阻塞态

进程更多的状态：

● 运行态：running

● 就绪态：ready

● 睡眠态：分为两种，可中断：interruptable，不可中断：uninterruptable

● 停止态：stopped，暂停于内存，但不会被调度，除非手动启动

● 僵死态：zombie，僵尸态，结束进程，父进程结束前，子进程不关闭，杀死父进程可以关闭僵死态的子进程

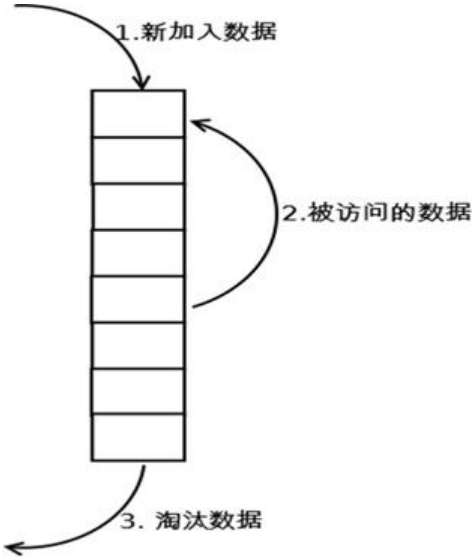
```
root        1725  0.0  0.0      0     0 ?        I    11:03   0:00 [kworker/1:0-events_freezable_power_]
root        1733  0.0  0.0      0     0 ?        I    11:04   0:00 [kworker/0:2-events]
root        1809  0.0  0.0      0     0 pts/1    Z+   11:41   0:00 [bash] <defunct>
root        1828  0.0  0.0      0     0 ?        I    11:41   0:00 [kworker/1:2-ata_sff]
root        1839  0.0  0.0      0     0 ?        I    11:45   0:00 [kworker/u256:2-events_unbound]
root        1844  0.0  0.0      0     0 ?        I    11:46   0:00 [kworker/1:1-ata_sff]
root        1848  0.0  0.4  57184  3888 pts/0    R+   11:48   0:00 ps aux
[root@centos8 ~]#
```

范例：僵尸态

```
[root@centos8 ~]#bash [root@centos8 ~]#echo $BASHPID 1809
[root@centos8 ~]#echo $PPID 1436

#将父进程设为停止态
[root@centos8 ~]#kill -19 1436

#杀死子进程，使其进入僵尸态
[root@centos8 ~]#kill -9 1809
[root@centos8 ~]#ps aux  #可以看到上面图示的结果，STAT为Z，表示为僵尸态

#方法1:恢复父进程
[root@centos8 ~]#kill -18 1436 #方法2:杀死父进程
[root@centos8 ~]#kill -9 1436

#再次观察，可以僵尸态的进程不存在了
[root@centos8 ~]#ps aux
```

1.5 LRU算法

LRU：Least Recently Used 近期最少使用算法（喜新厌旧），释放内存



范例:

假设序列为 4 3 4 2 3 1 4 2， 物理块有3个，则

第1轮 4调入内存 4

第2轮 3调入内存 3 4

第3轮 4调入内存 4 3

第4轮 2调入内存 2 4 3

第5轮 3调入内存 3 2 4

第6轮 1调入内存 1 3 2

第7轮 4调入内存 4 1 3

第8轮 2调入内存 2 4 1

1.6 IPC进程间通信

IPC: Inter Process Communication

- 同一主机：

pipe 管道，单向传输
socket 套接字文件
Memory-maped file 文件映射,将文件中的一段数据映射到物理内存，多个进程共享这片内存
shm shared memory 共享内存
signal 信号
Lock 对资源上锁，如果资源已被某进程锁住，则其它进程想修改甚至读取这些资源，都将被阻塞，直到锁被打开
semaphore 信号量，一种计数器

- 不同主机：socket=IP和端口号

RPC remote procedure call
MQ 消息队列，生产者和消费者，如：Kafka，RabbitMQ，ActiveMQ

范例：利用管道文件实现进IPC

```
[19:28:52 root@centos8 ~]#mkfifo /root/test.fifo
[19:33:27 root@centos8 ~]#ll /root/test.fifo
prw-r--r-- 1 root root 0 Jan  2 19:33 /root/test.fifo
[19:33:35 root@centos8 ~]#cat > /root/test.fifo
zhangzhuo

#在另一个终端可以从文件中读取数据
[19:34:12 root@centos8 ~]#cat /root/test.fifo
zhangzhuo
```

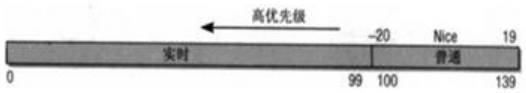
范例：查找socket文件

```
[19:36:17 root@centos8 ~]#find / -type s -ls
```

1.7 进程优先级

linux2.6内核将任务优先级进行了一个划分, 实时优先级范围是0到MAX_RT_PRIO-1（即99），而普通进程的静态优先级范围是从MAX_RT_PRIO到MAX_PRIO-1（即100到139）.

优先级范围	描述
0——99	实时进程
100——139	非实时进程



CentOS 优先级

进程优先级：

系统优先级：0-139, 数字越小，优先级越高,各有140个运行队列和过期队列
实时优先级：　99-0　值最大优先级最高
nice值：-20到19，对应系统优先级100-139或

Big O：时间（空间）复杂度，用时（空间）和规模的关系
O(1), O(logn), O(n)线性, O(n^2)抛物线, O(2^n)

1.8 进程分类

操作系统分类：

- 协作式多任务：早期 windows 系统使用，即一个任务得到了 CPU 时间，除非它自己放弃使用CPU，否则将完全霸占 CPU ，所以任务之间需要协作——使用一段时间的 CPU ，主动放弃使用
- 抢占式多任务：Linux内核，CPU的总控制权在操作系统手中，操作系统会轮流询问每一个任务是需要使用 CPU ，需要使用的话就让它用，不过在一定时间后，操作系统会剥夺当前任务的 CPU 使用，把它排在询问队列的最后，再去询问下一个任务

进程类型：

- 守护进程: daemon,在系统引导过程中启动的进程，和终端无关进程
- 前台进程：跟终端相关，通过终端启动的进程

注意：两者可相互转化

按进程资源使用的分类：

- CPU-Bound：CPU 密集型，非交互
- IO-Bound：IO 密集型，交互

1.9 IO调度算法

- **NOOP**

  - NOOP算法的全写为No Operation。该算法实现了最简单的FIFO队列，所有IO请求大致按照来 后到的顺序进行操作。之所以说“大致”，原因是NOOP在FIFO的基础上还做了相邻IO请求的合， 并不是完完全全按照先进先出的规则满足IO请求。NOOP假定I/O请求由驱动程序或者设备做了优或者重排了顺序(就像一个智能控制器完成的工作那样)。在有些SAN环境下，这个选择可能是最好选。Noop 对于 IO 不那么操心，对所有的 IO请求都用 FIFO 队列形式处理，默认认为 IO 不会存在性问题。这也使得 CPU 也不用那么操心。当然，对于复杂一点的应用类型，使用这个调度器，用户自就会非常操心。

- **CFQ**

  - CFQ算法的全写为Completely Fair Queuing。该算法的特点是按照IO请求的地址进行排序，不是按照先来后到的顺序来进行响应。 在传统的SAS盘上，磁盘寻道花去了绝大多数的IO响应时间。FQ的出发点是对IO地址进行排序，以尽量少的磁盘旋转次数来满足尽可能多的IO请求。在CFQ算法，SAS盘的吞吐量大大提高了。但是相比于NOOP的缺点是，先来的IO请求并不一定能 被满足，能会出现饿死的情况。

  - Completely Fair Queuing （cfq, 完全公平队列) 在 2.6.18 取代了 Anticipatory scheduler 成 Linux Kernel 默认的 IO scheduler 。cfq 对每个进程维护一个 IO 队列，各个进程发来的 IO 请求会被 cfq 以轮循方式处理。也就是对每一个 IO 请求都是公平的。这使得 cfq 很适合离散读的应用(eg: OLT DB)

- **Deadline scheduler**

  - DEADLINE在CFQ的基础上，解决了IO请求饿死的极端情况。deadline 算法保证对于既定的 IO 请求以最小的延迟时间，除了CFQ本身具有的IO排序队列之外，DEADLINE额外分别为读IO和写IO提 供了FIFO队列。读FIFO队列的最大等待时间为500ms，写FIFO队列的最大等待时间为5s。FIFO队内的IO请求优先级要比CFQ队列中的高，，而读FIFO队列的优先级又比写FIFO队列的优先级高。优级可以表示如下：

  - FIFO(Read) > FIFO(Write) > CFQ

- **Anticipatory scheduler**

    - CFQ和DEADLINE考虑的焦点在于满足零散IO请求上。对于连续的IO请求，比如顺序读，并没做　优化。为了满足随机IO和顺序IO混合的场景，Linux还支持ANTICIPATORY调度算法。

    - ANTICIPATORY的在DEADLINE的基础上，为每个读IO都设置了6ms 的等待时间窗口。如果在6ms内OS收到了相邻位置的读IO请求，就可以立即满足　Anticipatory scheduler（as) 曾经一度是 Linux 2.6 Kernel 的 IO scheduler 。Anticipatory 的中文含义是”预料的, 预想的”, 这个词的确揭示这个算法的特点，简单的说，有个 IO 发生的时候，如果又有进程请求 IO 操作，则将产生一个默认的 毫秒猜测时间，猜测下一个 进程请求 IO 是要干什么的。这对于随即读取会造成比较大的延时，对数库应用很糟糕，而对于 Web Server 等则会表现的不错。这个算法也可以简单理解为面向低速磁盘的因为那个”猜测”实际上的目的是为了减少磁头移动时间。

范例：查看IO调度算法

```
[19:36:37 root@centos8 ~]#cat /sys/block/sda/queue/scheduler
[mq-deadline] kyber bfq none
#每个Linux发行版可能都不一样
```