# 内存基础

## 基本概念

### 内存在计算机中的位置

内存是计算机中重要的组成部分，它通过内存总线与其他设备项链，内存本身是整个存储系统的一部分，访问速度比磁盘快几个数量级，可以视为磁盘到CPU之间的缓存盘。



### 物理地址

通常计算机使用大容量的、单位容量价格较为低廉的DRAM作为内存。内存的最小单位为bit，不过访问内存时一般以8bit也就是字节为单位。硬件层会将计算机所有内存条抽象为一个连续的字节数组，内存条上的每一个字节大小的存储位置将对应于这个数组中的一个元素，该元素的数组下标称为这个字节的物理地址（Physical Address，简称PA）。

### 虚拟地址

现代处理器中往往不再使用直接物理寻址的方式访问内存，而通过所谓的虚拟地址VA来访问内存。在CPU芯片中增加了一个叫内存管理单元（MMU=Memory Manager Unit）的部分用于地址翻译。



上图是CPU将要访问的内存的虚拟地址发给MMU，MMU翻译为物理地址后请求内存，内存返回实际数据给CPU。

无论是编译型还是解释型的应用程序，其指令和静态数据都会被编码为一定的内存地址，这些地址在程序运行之前即已确定。当直接应用程序使用物理地址时，多个程序中的相同地址对应相同的内存块，在这些程序交替执行时，将不得不反复的替换物理内存中的内容。有了虚拟地址后，操作系统可以将不同程序的相同的虚拟地址映射到不同的物理地址上，这样大大提升了程序的执行效率。同时操作系统还可以对不同的应用程序进行有效的隔离，提高了安全性。

### 进程虚拟地址空间

每个进程都有自己的虚拟地址空间，如下图所示，两个不同的进程A、B的地址空间内，各自只使用了部分地址，它们的实际地址可以映射到物理内存中的不同的段。注意进程的虚拟地址空间可以比物理内存大，比如1G物理内存可以支持4G的虚拟地址空间，系统可以将暂时不用的内存交换到磁盘上。



### 指令集架构的特权级

现代指令集架构一般都包含多个特权等级，每个等级下可以执行的指令集和范围不同。以IntelX86为例，Ring3为最低特权级，只能执行受限制令，Ring0为最高特权级，可以执行所有指令。



说明：图中的Ring1和Ring2是Intel预期保留给设备驱动程序的，但在Intel系统实现时，由于只使用Ring0和Ring3，所以设备驱动程序实际在Ring0执行。

### 用户态和内核态

操作系统可以基于特权控制应用程序对各种资源包括内存的访问。对IntelX86，Linux只使用Ring0和Ring3两个特权级，Ring0对应内核态，Ring3对应用户态。用户态一般不能直接访问系统的资源，也不能直接访问操作系统内核的代码与数据，而需要通过如系统调用的方式，切换为内核态之后，由操作系统去访问具体的资源或实现某些功能。很多系统调用常由库函数包装后提供给应用程序使用。

### 虚拟地址的划分

虚拟地址空间被划分为内核空间和用户空间。

当进程处于内核态时（通常是发生系统调用时，此时执行的是内核和设备驱动代码）可以访问内核空间和用户空间的所有地址。当进程处于内核态时，只能访问用户空间的内存地址。

下图是32位Linux系统的典型地址划分，用户空间：内核空间=3:1。



对于64位系统，2^64非常巨大，实际用不到那么大的虚拟地址空间。Linux一般使用42位或47位地址空间，而此时内核空间与用户空间也不是连续的，从0x0开始的128T虚拟地址空间为用户空间，从而虚拟地址空间顶部开始向下的128T空间为内核空间。

Linux上用轻量级进程实现线程，因此实际上每个线程都可以各自处于用户态或内核态。

## 页与巨页

### 页

在使用虚拟地址空间时，虚拟地址不等于物理地址，需要两者建立映射关系，而这种映射关系也需要被保存在内存中以供MMU查询。显然，按字节来建立映射关系，整个内存都不够用。因此按照一定大小的内存块管理内存，并以块为单位来建立映射关系，就成为一种合理的选择。硬件系统本身也支持这种管理方式。

页：一定大小的内存块。操作系统以页为单位来管理虚拟地址空间和物理内存，在Linux上一页一般为4K大小。

### 页表

虚拟页与物理页可以通过两个页的首地址的映射关系来关联，这种映射关系的记录保存在所谓的页表中，因此被称为页表项。页表使用树状的结构来保存所有的页表项，以方便页表项的快速查找。页表的详细组织结构以及页表项的查找可以参考操作系统相关书籍。

每个进程都有自己的页表来记录自己的虚拟页的映射关系，页表并不需要记录该进程整个虚拟地址空间的页映射情况，只需要记录已分配的部分即可，这样可以节省大量内存空间。

### TLB（Translation Lookaside Buffer）

TLB：旁路转换缓存，是继承在CPU芯片上的一个内存管理单元，用于缓存页表项。

在引入TLB之前，访问一个内存单元需要两步。首先要从内存中的页表查询页表项，然后根据页表项再访问与虚拟地址对应的物理地址。这样需要访问内存两次，效率比较低。在引入TLB之后，有了页表项缓存，可以提高内存访问效率。



问题：MMU本身是做地址翻译的，那么MMU是怎么知道内存中的页表在哪里呢？

回答：这个地方需要操作系统与硬件相配合。操作系统可以通过指令告知MMU页表的实际物理地址，MMU直接通过物理地址访问页表。

备注：如果MMU在TLB中查到映射关系，还是需要通过内存中的页表去查询的。

### 巨页

引入TLB后，内存访问效率有了提升。但现代的操作系统上运行的进程很多，所有进程的页表项数目非常可观，而TLB本身的空间有限，只能缓存所有页表项中较少的一部分，缓存命中率不高。为了提升TLB命中率，Linux操作系统采用了巨页技术，巨页大小为M量级甚至G量级，典型大小为2M或1G。一个2M的巨页等于512个4K页，原来需要512个页表项记录映射关系，现在仅需1个页表项，页表项总数大大缩减，TLB的缓存数/页表项总数的比值提高了，命中率也就提高了。

### 使用巨页操作系统

巨页可以由用户手动设置或由操作系统自行管理，手动设定可以用如下命令指定巨页文件系统：

mount –t hugetlbfs –o uid=<value>,gid=<value>,mode=<value>,size=<value>,

nr\_inodes=<value> none <mount\_path>

需要注意的是，该文件系统上不能使用常用的read/write访问文件，只能使用mmap即内存映射的模式或共享内存方式（shmget、shmat，最终仍然调用mmap）来访问。

命令参数中：uid和gid为指定用户的id和组id，mode为读写执行模式，size为要使用的巨页文件大小，nr\_inodes为该文件系统最大inode数，限制文件及文件夹的数目，mount\_path为挂载路径。

### 透明巨页

由操作系统自行管理的巨页称为透明巨页（THP，Transparent Huge Page）。开启了透明巨页后，操作系统会根据内存使用情况判断是都需要将若干较小的页（比如4K的页）合并为较大的页（比如2M的页）进行管理。

使用以下命令可以查询是否启用了THP：

cat /sys/kernel/mm/transparent\_hugepage/enabled

always：尽量使用；never：禁用；madvise：避免内存改变。

启用/禁用透明巨页：

1、开机启动设定

可以通过修改grub.cfg实现，在启动项中增加：

transparent\_hugepage=[always|never|madvise]

2、临时设置，关机或重启后失效

echo [always|never|madvise] > sys/kernel/mm/transparent\_hugepage/enabled

## swap分区

### swap空间

Linux可以借助磁盘空间暂时保存内存中的内容，来实现虚拟地址空间大于物理内存，这部分磁盘空间就称为swap空间。Swap空间可以类比于windows系统上的虚拟内存。

查看swap空间信息的方法：

cat /proc/swaps

swapon –s

### 创建和使用swap

使用swap的相关命令：

mkswap <swap\_path>：在文件和设备上建立swap空间。

swapon/swapoff <swap\_path>：启动/关闭swap空间。

swapon/swapoff –a：启动/关闭所有swap空间。

启动时自动挂载swap空间，需要在/tec/fstab中增加：

<swap\_path> swap swap default 0 0

### 内存换出

Linux通过内核线程kswapd来周期性检测内存使用情况，并执行将内存移出到swap的操作。内存的移出以页为单位，所以也称为内存页换出。选择哪些页被换出基于LRU原则，即最近最少使用的页被优先换出。内核中设定了high、low、min三个内存水位线，低于high表示有一定压力，此时就有可能触发内存换出，低于low表示压力很大，低于min表示内存非常紧张。High和low基于min计算，min水位线可用以下命令查询：

cat /proc/sys/vm/min\_free\_kbytes

### 内存换入

如果进程要访问的内存页已经被移出到swap空间中，而不在物理内存中，那么进程对该页的访问会触发缺页异常。缺页异常会被内存捕获，内核检查发现内存页在swap中，则会发出调页请求，从swap中将页载入到物理内存中，然后进程可继续正常运行。



### Linux释放内存的机制

Linux并非只有一种释放内存的机制。由于Linux倾向于尽量使用内存作为缓存使用，因此实际上内存中还会有文件缓存。在内存不足时，Linux也可以通过清除文件缓存的方式来释放内存。

swappiness值（0~100）指明了系统在释放内存时的倾向，0表示尽量不使用swap方式而通过释放文件缓存来缓解内存压力，100则表示以同样优先级使用两种方式，该值可使用cat /proc/sys/vm/swapiness查看。如果仍无法释放足够的内存，那么系统会选择杀死一些进程来释放内存。

杀死进程指的是发生OMM（Out Of Memory）时，系统按照一定的准则对各个用户进程的内存使用情况计算一个数值，按照这个数值排序，按顺序杀死进程，直至有足够的内存可以使用。如果出现进程莫名其妙退出的情况，可能就是由于系统OOM被杀死了，查看是否发生了OMM，可检查/var/log/message文件或者dmesg命令的输出中是否有Out Of Memery及kill的字样。

## NUMA

### 术语

core：物理核，基本的逻辑执行单元

CPU：指集成了多个core的物理芯片

Socket：主板上的CPU芯片槽位

Thread：这里不是指软件线程而是硬件线程。基于超线程技术，一个core可以支持多个指令并发执行，可认为 core上有多个硬件线程，每个硬件线程在系统中体现为一个逻辑核

Processor：处理器，有时候指core，有时候指CPU，依上下文语境而定

Socket就是常说的“几路几核”服务器找那个的“路”

CPU情况可以查看系统的/proc/cpuinfo信息获取，其中显示的每一个processor一般指thread，即逻辑core。

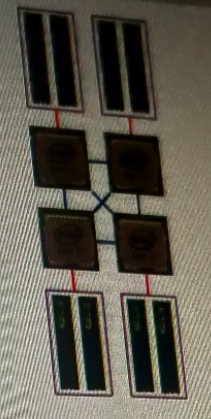
### UMA

UMA（uniform memory access），即一致性内存访问。在UMA架构下，所有CPU对所有内存的访问时间相同。UMA架构在CPu数量少的时候，性能非常好，但随着CPU数目增加，各CPU访问内存时对内存总线的争用回越来越剧烈，从而导致性能下降。因此，从长期发展来看，UMA架构越来越无法适应多核时代的要求。



### NUMA

NUMA（non-uniform memory access），即非一致性内存访问。在NUMA架构下，每个CPU都有自己的本地内存，如下图所示，紫框内为多个内存条组成的CPU本地内存，红线则为本地内存访问，其访问速度比较快。而如果一个CPU上的核要访问其他CPU的本地内存（远端内存），则需要通过图中的蓝线所示的CPU互联模块来进行，速度会明显慢于本地内存访问。

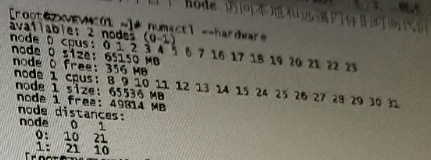


### 查看NUMA信息

#### numactl

在NUMA架构下，拥有本地内存的CPU称为node。使用numactl –hardware命令可以看到基本的NUMA信息，其中会显示node上的所有逻辑核编号以及本地内存大小、使用量等信息。

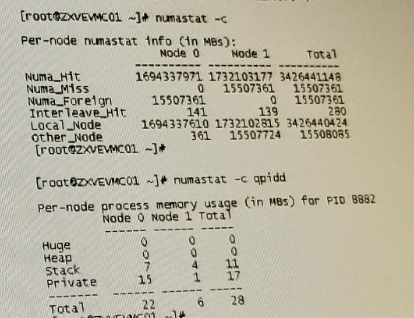
最下方的node distances显示了各个node访问本地和远端内存的时间代价的相对值。



#### numastat

numastat –c命令可以查看系统的NUMA使用情况，其中NUMA hit表示访问本地内存的总量，NUMA Miss表示访问远端内存的总量。

numastat –c <pro\_name>可以查看进程内存在node上的分布：



### 启用NUMA自平衡

由于NUMA架构下访问本地内存和远端内存的差异，将进程的内存集中在执行该进程命令的逻辑核所在的node上，显然可以提高进程的性能。系统提供了numad命令来对进程的内存做调解，它会逐步将进程使用的逻辑核与内存集中在相同的node上以提高性能。

numad：系统自动执行NUMA自平衡

numad –p <pid?：对指定的进程执行NUMA自平衡

nomad –r <pid>：对指定的进程取消NUMA自平衡

### NUMA的影响

一般情况下，使用NUMA自平衡可以提高性能，但在特殊情况下，反而会导致性能大幅下降，典型的如数据库场景，数据库进程往往需要大量内存，当启动NUMA自平衡后，这些内存集中在同一个node。当数据库需要更多的内存时，由于Linux的策略原因，会优先将内存置换到swap空间上，而不是在其他node上分配空间，这样就会导致大量的内存换页，造成数据库性能大幅降低甚至卡死。因此在使用NUMA自平衡策略前，需要对进程的内存行为以及系统的策略设置有充分了解。

关于NUMA对数据库的影响，参见：

http://cenalulu.github.io/linux/numa

## 内存统计信息

### 内存指标(/proc/meminfo)

/proc/meminfo包含了系统内存的统计信息，可以使用cat 命令查看其中的内容。

MemTotal：物理内存总量

MemFree：空闲物理内存

MemAvailable：可用物理内存，注意可用物理内存不等于空间物理内存，而是大致上等于MemFree+Cached

Cached：用来缓存文件数据的物理内存大小，内存不足时可以释放这部分来缓解内存压力

Buffer：用来缓存块设备数据占用的物理内存大小，这部分占用一般比较小

SwapCached：内存中和swap空间中都存在的内存大小，在内存紧张时这部分内存无需换出到swap空间，可直接释放

Active：物理内存中较活跃的部分，即最近被使用过的部分

Inactive：物理内存中不活跃的部分，即最近未被访问过的部分

Active+Inactive：一般略小于MemTotal-MemFree

Active(file)：活跃内存中属于文件缓存的部分

Inactive(file)：不活跃内存中术语文件缓存的部分，有内存压力时优先释放的部分

Active(anon)：活跃内存中术语各进程使用的部分

Inactive(anon)：不活跃内存中术语各进程使用的部分

Active=Active(file)+Active(anon)

Inactive=Inactive(file)+Inactive(anon)

Cache大致等于Active(file)+Active(anon)

SwapTotal：swap空间总量

SwapFree：swap空间的空闲大小，前面的SwapCached也算SwapFree

Dirty：脏页大小。Linux处理I/O时一般不直接写入磁盘，而先缓存到内存中，后续再集中写入。这部分缓存可能包括新写入的数据，也有可能包括从磁盘中读入，但已做修改的数据，Linux中称这种数据为“脏”的，而数据又是以页为单位管理的，因而这些数据被称为脏页。如果机器突然断电，那么脏页中的数据将因为未及时写入磁盘而丢失。

/proc/meminfo中各项指标可以用来对系统的内存情况进行判断，比如MemFree很小但MemAvailabel很大，那说明再启动新的进程可能需要释放文件缓存或交换内存到swap空间来空出一部分内存，但如果Inactive(file)也比较小，那么说明文件缓存正在被使用，释放文件缓存可能无法空出足够的内存空间，而需要使用swap空间，则性能明显会受到影响。另外，Active(file)和Dirty两项的值比较大，可以从侧面表明系统的I/O比较活跃。

### 内存相关命令

#### top/free

top命令：显示物理内存、swap空间、Cached、Buffer使用情况，另外还有进程内存使用情况

free命令

free –m/g：以M/G为单位显示内存使用信息

free –h：以可读性最佳的方式显示内存使用信息

free –s <N>：以N秒为间隔不断获取内存使用信息

free –c <N>：显示N次信息后退出

#### vmstat

vmstate –s：显示类似于/proc/meminfo中的主要内存指标，另外还显示一些CPU统计信息

vmstate <N>：以N秒为间隔显示MemFree、SwapFree、Buffers、Cached、si、so信息

vmstate –a <N>：以N秒为间隔显示MemFree、SwapFree、Active、Inactive、si、so信息

其中，si、so分别是每秒从swap换入内存的数据量和每秒从内存换出到swap的数据量

#### sar

sar命令可以显示CPU、内存、网络、磁盘等方面统计信息，是Linux上监控系统使用状况的最强大的命令之一。

sar <N>：以N秒为间隔不断显示统计信息，如果不携带N这个参数，那么会显示从当日0点开始至当前时间以10分钟为粒度的统计信息

sar –r：显示与/proc/meminfo类似的主要内存指标

sar –R：显示物理内存、Buffers、Cached单位时间内变化情况，单位为页/s

sar –B：以页为单位统计了内存与磁盘的交互、缺页异常发生情况、kswapd扫描情况等多种信息

sar –H：显示巨页的统计信息

sar –S：显示swap的使用情况

sar –W：显示单位时间内swap换页情况

另外，系统的/var/log/sa目录下以sar开头的文件记录了每天的系统性能统计信息，每一类统计信息均以10分钟为粒度，可以在排查问题时追溯当时的系统运行状况。

sar命令包含在sysstet工具中。

#### nmon

nmon是一款IBM开发的AIX/Linux系统性能监控工具，它的功能与前面的sar功能上比较接近。

Nmon提供两种监控方式。一种与top类似，动态刷新性能数据，并提供不同的按键以选择具体的监控项，内存项监控为M；一种是通过设置命令行参数，将需要统计的信息输出到文件中。对包含输出信息的文件，可以使用nmon analyser工具将统计数据导入到excel中，以生成形象化的报表。

# 进程内存布局

## 应用程序内存布局

### 内存分段

应用程序的内存分为多段，一般包括：

text段：保存二进制形式的应用程序可执行代码

data段：保存代码中在定义时就被初始化的全局/静态变量

bss段：保存代码中在定义时未被初始化的全局/静态变量

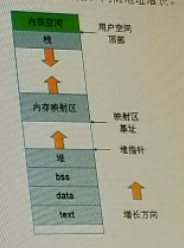
stack段：进程栈区域，代码执行时使用

heap段：进程的堆，动态分配内存时使用

内存映射区：可用于动态库、文件映射以及动态内存分配

### 32位布局

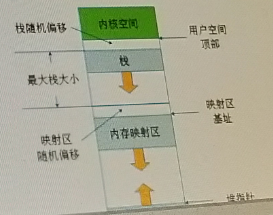
Linux的经典内存布局（早期使用）如下所示，进程的内存地址空间一般为4G，上部为内核空间，一般为1G，下部为用户空间，一般为3G。从0地址开始有一段保留地址，然后依次是text、data、bss段，堆中bss顶部开始，向高地址增长。内存映射区从映射区基地址开始，向高地址增长。栈从用户空间顶部开始，向低地址增长。

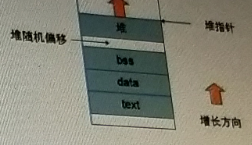


现代的32位Linux布局与经典布局不同，主要有如下两点：

内存映射区改为从栈顶部开始向低地址增长，即和堆相向增长，这样可以最大限度的利用中间的空闲地址空间。

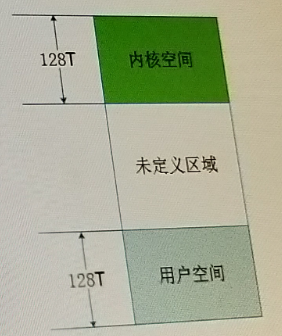
在栈、内存映射区和堆的开始处均增加了一段随机大小的偏移，以防御和堆溢出这样的攻击行为。





### 64位局部

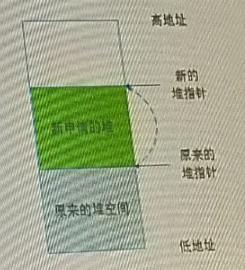
64位Linux的布局与32位有些不同。如下图所示，在64位场景下，内核空间与用户空间不再是连续的了，而是由一大块未定义区域隔开，两者各有128T的地址空间。在这种情况下，地址空间一般都很充裕，而且不同段距离很远，无需像现代的32位布局那样保留随机间隔，因而64位用户空间的内存布局实际与32位经典布局类似。



### 申请内存方式

#### brk

进程有一个称为program break的指针指向堆的顶部的地址，通过系统函数brk可以改变这个指针的位置，从而改变堆的大小（堆可以扩大也可以缩小）。这种申请内存的方法存在一个缺陷，即如果堆顶部分的内存一直在使用，就算其他部分未被使用，堆页无法收缩，这可能会造成隐形的内存泄露。



#### mmap

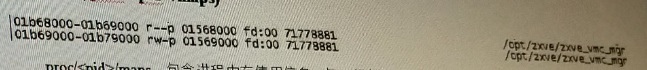
mmap函数通常被应用程序用来将文件映射到内存中，其实它也可以用来向系统申请新的内存映射区域作为动态内存来使用。

与堆不同的是，mmap可以在内存映射区中寻找空闲的合适大小的区域来分配，应用程序可以多次调用从而申请多个内存映射区，每一块映射的内存只要不再使用，都可以单独释放掉，不像堆那样受到堆顶内存的影响。

mmap申请的内存大小以页为单位。

## 进程内存段分析

### 内存信息



/proc/<pids>/maps包含进程内存使用信息，每一行包含：

01b68000-01b69000：虚拟地址段的地址范围

rwxp：rwx读、写、执行权限；p/s：私有/共享

01568000：可执行文件/共享库文件不同段的偏移量

fd:00：可执行文件/共享文件主次设备号，其余为00:00

71778881：可执行文件/共享库文件的inode号

/opt/.zxve/\*\*\*：可执行文件名/共享库名/段名

### text段

text段如下所示，textdaunt有可执行权限，即x属性：



可以通过objdump命令来查看可执行文件的反编译信息：

objdump –D <exec\_path>

在输出信息中，可以看到<\_FRAME\_END\_>，这个就是text段的结束位置，与maps中的结束位置可以匹配上：

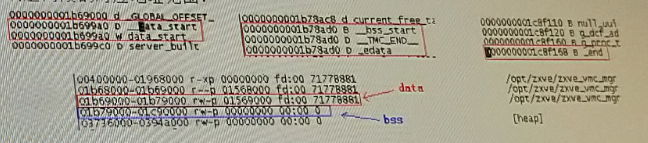


### data段和bss段

类似于text段，可以通过nm命令来获取可执行程序的符号表信息，来确定data段和bss段的一些信息：

nm <exec\_path> | sort

从nm的输出可以看到data段和bss段的起始地址和整个数据段的截止地址，从而确定两段的对应地址范围：



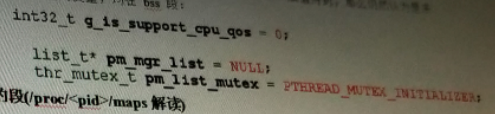
nm命令后面加上sort是为了按照虚拟地址排序，分析起来更加方便。

关于data段和bss段：

bss段的起始位置实际在data段的地址范围内，这个是由于maps中是以页（一般为4K）为单位来显示的。

有的变量已经初始化了，但却在bss段中。

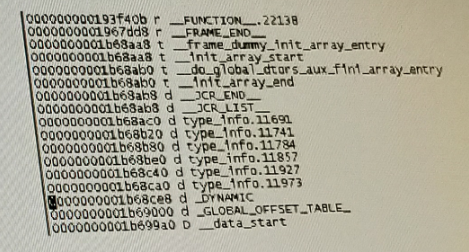
全局/静态变量初始化时如果只由0或NULL或’\0’赋值得到，那么仍然认为是未初始化的，以下全局变量均在bss段：



### libc段

在text段和data段之间还有一小段内存地址是gcc编译时插入的。main函数并不是程序的真正入口，在main函数执行前和执行后libc都做了大量工作，包括程序的初始化与终结，这一小段就是供libc使用的。

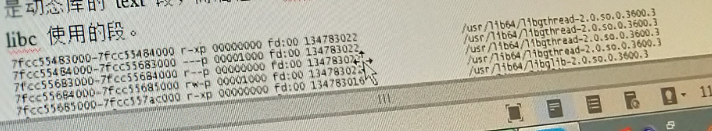
nm的输出中，\_FRAME\_END\_和\_GLOBAL\_OFFSET\_TABLE\_之间的就是libc相关的段：



### 内存映射区

在/proc/<pid>/maps里面可以看到类似于如下的段，这些就是进程内存映射区中动态库占用的内存部分。

与应用程序类似，动态库页包含text段、data段、bss段，在下图中，属性为rxp的是动态库的text段，而属性为rwp的则是动态库的data段和bss段，属性为rp的为libc使用的段。



### 动态库

#### 映射方式

#### 动态库数据修改

#### 注意点

从前面所述动态库的特点，可以得到如下注意点：

由于修改时新建数据段拷贝，因此不可以通过动态库中的全局变量，在进程间传递数据。因为修改后两个进程中的动态库的全局变量已指向不同的物理地址。

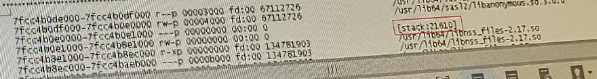
由于代码段被共用，因此如果出现某动态库S正在被进程A使用，然后动态库S的文件被替换为新版本，再启动一个使用S的进程B，那么当A下次调用S中的函数时，就会崩溃，因此此时S已被系统重新加载，原来函数地址的内容已变。

第二种情况是实际发生过的，在某软件升级过程设计中，没有注意另一升级相关的进程引用了被替换的动态库，导致替换动态库后该进程崩溃，软件升级被中断。

### 栈

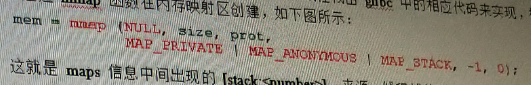
#### /proc/<pid>/maps

在/proc/<pid>/maps的末尾可以看到标注为[stack]的内存区域，这个与标准的布局一致。但在maps的中间段，还有标记为[stack:<number>]的区域，这是怎么回事？



#### C程序栈

我们在maps底部看到的[stack]实际上只是进程的主线程的栈。进程的其他线程都是POSIX线程，使用pthread\_create创建。POSIX线程栈由glibc中的相应代码来实现，栈空间则通过mmap函数在内存映射区创建，如下图所示：



这就是maps信息中间出现的[stack:<number>]来源。线程栈的默认大小一般为8M或者10M，依Linux发行版不同而有所区别。

较早的Linux不会在maps中显示线程栈信息。

#### 查看C程序栈

pstack命令可以查看进程的栈（包括线程），用法为：

pstack <pid>

另一个可以查看进程堆栈的命令为gstack，用法与pstack一样，实际上Linux的pstack一般为指向gstack的一个软链接。

pstack和gstack对Java程序也有效，只是输出内容包含的是native层的栈信息，不包含Java层的信息。

pstack和gstack在排查进程死锁等问题时非常有效。

有时候，线程本身已经挂死，pstack和gstack查看进程栈时也会挂住，这时可以通过proc文件系统来查看进程栈信息：

cat /proc/<pid>/stack

cat /proc/<pid>/task/\*/stack

task下包含了进程中的所有线程，上面的命令可以查到所有线程的栈信息。不过需要注意的是，这个方法只能显示内核栈，不包含用户栈。此方法在排查进程因系统调用（比如I/O）挂死时很有效。

#### Java程序栈

#### 查看Java程序栈

### 堆

### 其他内存段

# 进程内存使用情况