虚存抽象

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

进程眼中的内存

*(char *)x

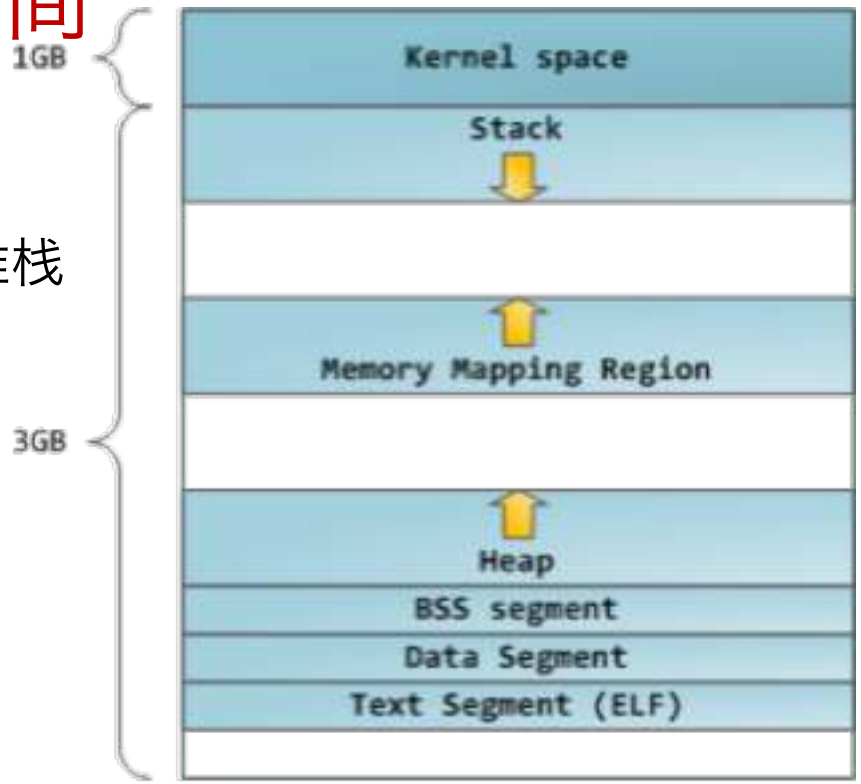
- 假设x是一个整数

- 这是……什么?
  - char* 是什么类型?
  - (char *)x 是强制类型转换
  - *p是一个L-value，代表p地址的值

*(char *)x: 例子

- 随机的数值
  - 大概率是Segmentation Fault
- 有意义的数值
  - objdump: 0x80482ac是0x53 (push %ebx)
  - x = 0x80482ac → 能读出0x53
- 程序的执行是(M, R)的状态转移
  - 进程能看到属于它自己的虚拟内存 (由VM把虚拟地址映射到物理地址)

x86 Linux进程的地址空间



- 程序执行必要的部分
  - 代码(code)、数据(data, bss)、堆栈
- 动态链接库
  - 代码、数据
  - 问题：如何解释ldd的结果？
- 操作系统的代码和数据
  - 与进程处于同一地址空间，但进程无权访问
  - 操作系统“上位”时只需要权限切换，就能访问进程的所有数据

虚拟内存的好处

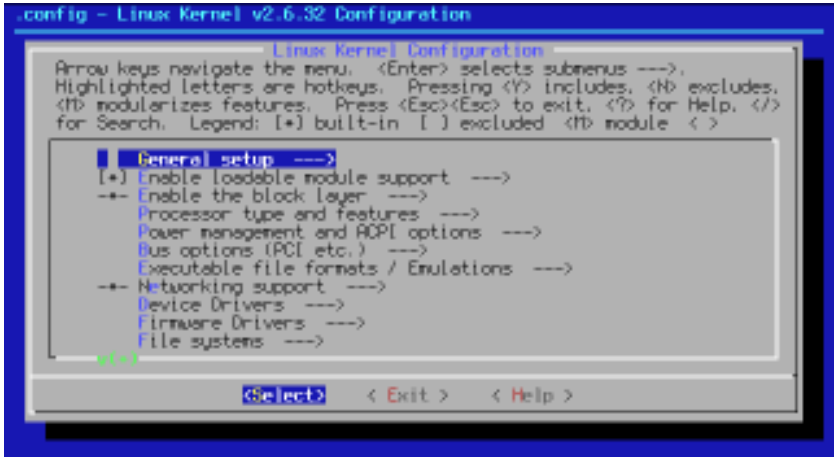
- 操作系统能灵活实现$VM(x)$
  - 限制进程的行为，防止非法访问
  - 能够在进程之间共享数据(如动态链接库的代码)
  - 能够在进程实际访问一页时再分配内存(copy-on-write)
  - ……

库函数眼中的内存

库函数：简介

- 世上本没有库，用的多了就成了库
  - 常用子程序(函数)的集合

- 例子
  - libc – C标准库 (ANSI C, POSIX)
  - glibc – C标准库和各种系统相关函数 (动态链接、线程……)
  - ncurses – 终端交互库
  - SDL – 多媒体库



Newlib: 嵌入式系统的libc

- 特点
  - 小而全(1995年发布的1.6.1版本只有107,254行代码)
  - 容易理解、容易定制(NEMU PA里的仙剑奇侠传链接了newlib)
  - 最新的版本有了很大的性能提升，有阅读的价值

- 帮助大家理解“应用程序眼中操作系统”的手段
  - busybox (应用) – newlib (C标准库) – 操作系统(POSIX系统调用)
  - 里面的代码非常好读
    - printf, vprintf, vsprintf, vsnprintf都是什么?
    - malloc/free是如何实现的?
    - setjmp/longjmp是什么?

库函数：不过是普通的代码

- 例子：
  - libm中的数学库
  - string, stdio, stdlib

- 代码量：glibc >> newlib-3.0 >> newlib-1.6
  - 虽然实现相同功能，但性能差距很大
  - glibc有大量平台相关代码

C标准库：malloc & free

- 内存管理的两个API (newlib中的malloc有3700行)
  - void *malloc(size_t size);
  - void free(void *ptr);
  - 所有内存管理都可以用这两个API实现
    - strdup; fopen; std::vector; std::set; ...

- 有没有想过这些内存是从哪里来的?
  - 似乎是问操作系统要的?
  - malloc / free是不是系统调用呢?

strace告诉我们答案

- 在malloc/free的程序中，执行的系统调用只有brk

- 操作系统设置一个“break”
  - [&end, break] – 进程可用的堆区
  - 堆区可以“生长”也可以“缩小”

- 讨论：如何用brk实现malloc/free?
  - 这个问题会在这门课中多次遇到

一些问题

- 栈区能任意增长吗?
  - brk与栈似乎没什么关系
  - 但栈区的确是可以无限的(ulimit –s unlimited)
- brk只管理了一个堆区
  - 但栈区和堆区好像没有本质区别(一段内存；一端生长/收缩)
  - 我能拥有几个堆区吗?
- 讨论：如何解决日益增长的内存管理需求?
  - 能够动态增长的多个内存区域
  - 支持动态链接

进程地址空间的管理

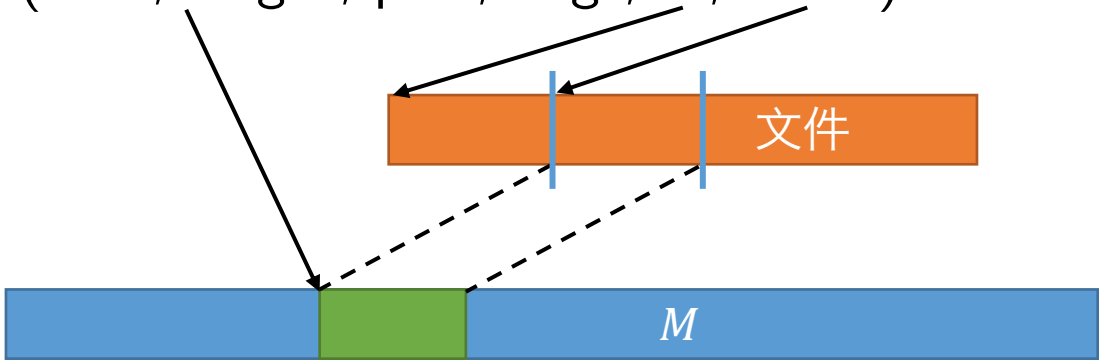
如果只给你一个API管理内存，它是什么？

- 通常，一段连续的内存存放相关的数据
  - 物理世界中常见的“局部性”

- 自然也可以把地址空间分成若干连续的部分
  - 代码、数据、堆栈、堆区、动态链接库……

mmap & mprotect

- mmap(addr, length, prot, flags, fd, offset)



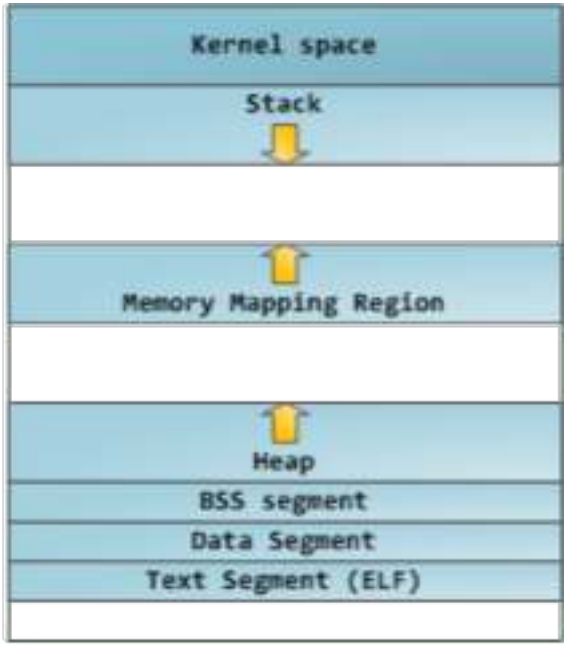
  - 映射一段内存，访问权限为prot，选项为flags (例如MAP_ANONYMOUS)
- mprotect(addr, length, prot) – 同理

- 讨论：mmap的行为应该是怎样的？是否解决了问题？

一劳永逸的解决方案

- mmap/munmap管理了操作系统对进程地址空间的认识
  - 定义了$VM(x)$
  - 操作系统利用硬件提供的指令实现这一映射

- Linux do_brk()的实现位于mmap.c
  - 通过vma_merge()扩展已有的内存映射



如何实现mmap?

- 操作系统维护每个进程的映射表
  - $[x, y)$映射到匿名内存/文件/...
  - 但实际不映射任何页面

- 进程执行指令访问内存x将产生缺页
  - 操作系统代码接管执行，在映射中查找
  - 如果x属于该进程，则实际分配页面

- 能实现诸多有趣功能
  - 多进程以不同权限共享映射文件的同一部分
  - 多进程之间的共享内存