第二课　内存管理

一、错误处理

1. 通过函数的返回值表示错误

1) 返回合法值表示成功，返回非法值表示失败。

范例：bad.c

2) 返回有效指针表示成功，
　返回空指针(NULL/0xFFFFFFFF)表示失败。

范例：null.c

3) 返回0表示成功，返回-1表示失败，
　不输出数据或通过指针/引用型参数输出数据。

范例：fail.c

4) 永远成功，如：printf()。

练习：实现四个函数
slen()　- 求字符串的长度，若为空指针，则报错。
scpy()　- 字符串拷贝，考虑缓冲区溢出，
　　成功返回目标缓冲区地址，
　　目标缓冲区无效时报错。
intmin() - 求两个整数的最小值，若二者相等，则报错。
intave() - 求两个整数的平均值，考虑求和溢出，
　　该函数不会失败。

代码：error.c

2. 通过errno表示错误

```
#include <errno.h>
```

1) 根据errno得到错误编号。

2) 将errno转换为有意义的字符串：

```
#include <string.h>
char* strerror (int errnum);

#include <stdio.h>
void perror (const char* s);

printf ("%m");
```

范例：errno.c

3) errno在函数执行成功的情况下不会被修改，
   因此不能以errno非零，作为发生错误判断依据。

范例：iferr.c

4) errno是一个全局变量，其值随时可能发生变化。

二、环境变量

1. 环境表

1) 每个程序都会接收到一张环境表，
   是一个以NULL指针结尾的字符指针数组。

2) 全局变量environ保存环境表的起始地址。

```
           +---+
environ -> | * --> HOME=/root
           +---+
           | * --> SHELL=/bin/bash
           +---+
           | * --> PATH=/bin:/usr/bin:...:.
           +---+
           | . |
           | . |
           | . |
           +---+
           | 0 |
           +---+
```

图示：env_list.bmp

3) main函数的第三个参数是环境表的起始地址。

```
#include <stdio.h>
#include <unistd.h>

extern char** environ;

int main (int argc, char* argv[], char* envp[]) {
    printf ("%p, %p\n", environ, envp);
    return 0;
}
```

2. 环境变量函数

```
#include <stdlib.h>
```

环境变量：name=value

getenv   - 根据name获得value。

putenv   - 以name=value的形式设置环境变量，

name不存在就添加，存在就覆盖其value。

setenv - 根据name设置value，注意最后一个参数表示，
          若name已存在是否覆盖其value。

unsetenv - 删除环境变量。

clearenv - 清空环境变量，environ==NULL。

范例：env.c

三、内存管理

用 户 层	STL C++ 标C POSIX Linux	自动分配/释放内存资源 new/delete，构造/析构 malloc/calloc/realloc/free brk/sbrk mmap/munmap	调C++ 调标C 调POSIX 调Linux 调Kernel
系 统 层	Kernel Driver ...	kmalloc/vmalloc get_free_page ...	调Driver ... ...

四、进程映像
----

1. 程序是保存在磁盘上的可执行文件。

2. 运行程序时，需要将可执行文件加载到内存，形成进程。

3. 一个程序(文件)可以同时存在多个进程(内存)。

4. 进程在内存空间中的布局就是进程映像。
   从低地址到高地址依次为：

代码区(text)：可执行指令、字面值常量、
具有常属性且初始化的全局和静态局部变量。只读。

数据区(data)：不具常属性且初始化的全局和静态局部变量。

BSS区：未初始化的全局和静态局部变量。
进程一经加载此区即被清0。

数据区和BSS区有时被合称为全局区或静态区。

堆区(heap)：动态内存分配。从低地址向高地址扩展。

栈区(stack)：非静态局部变量，
包括函数的参数和返回值。从高地址向低地址扩展。

堆区和栈区之间存在一块间隙，
一方面为堆和栈的增长预留空间，
同时共享库、共享内存等亦位于此。

命令行参数与环境区：命令行参数和环境变量。

图示：maps.bmp

范例：maps.c

比对/proc/<pid>/maps

```
r = read
w = write
x = execute
s = shared
p = private (copy on write)

# size a.out

text    data     bss     dec     hex filename
2628     268      28    2924     b6c a.out
  |       |        |       |       |
  +-------+--------+  (10) +---+---+ (16)
          V                   ^
          +-------------------+
                    (+)
```

五、虚拟内存
---

1. 每个进程都有各自互独立的4G字节虚拟地址空间。

2. 用户程序中使用的都是虚拟地址空间中的地址，
   永远无法直接访问实际物理内存地址。

3. 虚拟内存到物理内存的映射由操作系统动态维护。

4. 虚拟内存一方面保护了操作系统的安全，
   另一方面允许应用程序，
   使用比实际物理内存更大的地址空间。

图示：vm.png

5. 4G进程地址空间分成两部分：
   0到3G-1为用户空间，
   如某栈变量的地址0xbfc7fba0=3,217,554,336，约3G；
   3G到4G-1为内核空间。

6. 用户空间中的代码，
   不能直接访问内核空间中的代码和数据，
   但可以通过系统调用进入内核态，
   间接地与系统内核交互。

图示：kernel.png

7. 对内存的越权访问，
   或试图访问没有映射到物理内存的虚拟内存，
   将导致段错误。

8. 用户空间对应进程，进程一切换，用户空间即随之变化。
   内核空间由操作系统内核管理，不会随进程切换而改变。
   内核空间由内核根据独立且唯一的页表init_mm.pgd
   进行内存映射，而用户空间的页表则每个进程一份。

9. 每个进程的内存空间完全独立。
   不同进程之间交换虚拟内存地址是毫无意义的。

范例：vm.c

10. 标准库内部通过一个双向链表，
    管理在堆中动态分配的内存。
    malloc函数分配内存时会附加若干(通常是12个)字节，
    存放控制信息。
    该信息一旦被意外损坏，可能在后续操作中引发异常。

范例：crash.c

11. 虚拟内存到物理内存的映射以页(4K=4096字节)为单位。
    通过malloc函数首次分配内存，至少映射33页。
    即使通过free函数释放掉全部内存，
    最初的33页仍然保留。

图示：address_space.png

```
#include <unistd.h>

int getpagesize (void);
```

返回内存页的字节数。

范例：page.c

```
char* pc = malloc (sizeof (char));
       |
       v<--------------- 33页 --------------->|
------+-------+----------+-------------------+------
       | 1字节 | 控制信息 |                   |
------+-------+----------+-------------------+------
  ^       ^         ^                 ^           ^
```

段错误   OK   后续错误   不稳定   段错误

六、内存管理APIs
---

1. 增量方式分配虚拟内存

```
#include <unistd.h>

void* sbrk (
    intptr_t increment // 内存增量(以字节为单位)
);
```

返回上次调用brk/sbrk后的末尾地址，失败返回-1。

increment取值：

 0 - 获取末尾地址。

>0 - 增加内存空间。

<0 - 释放内存空间。

内部维护一个指针，
指向当前堆内存最后一个字节的下一个位置。
sbrk函数根据增量参数调整该指针的位置，
同时返回该指针原来的位置。
若发现页耗尽或空闲，则自动追加或取消页映射。

```
void* p=sbrk(4);       p=sbrk(0);
      |                 |
 返回 *-- increment ->* 返回
      |                 |
      v                 v
--+---+---+---+---+---+---+--
  | B | B | B | B | B | B |
--+---+---+---+---+---+---+--
      |<--------- 页 --------
```

2. 修改虚拟内存块末尾地址

```
#include <unistd.h>

int brk (
    void* end_data_segment // 内存块末尾地址
);
```

成功返回0，失败返回-1。

内部维护一个指针，
指向当前堆内存最后一个字节的下一个位置。
brk函数根据指针参数设置该指针的位置。
若发现页耗尽或空闲，则自动追加或取消页映射。

```
void* p=sbrk(0); brk(p+4);
      |                 |
      |                 v
 返回 *                 * 设置
      |                 |
      v                 v
--+---+---+---+---+---+---+--
  | B | B | B | B | B | B |
--+---+---+---+---+---+---+--
      |<--------- 页 --------
```

sbrk/brk底层维护一个指针位置，
以页(4K)为单位分配和释放虚拟内存。
简便起见，可用sbrk分配内存，用brk释放内存。

范例：brk.c

3. 创建虚拟内存到物理内存或文件的映射

```
#include <sys/mman.h>

void* mmap (
    void*  start,  // 映射区内存起始地址，
                   // NULL系统自动选定，成功返回之
    size_t length, // 字节长度，自动按页(4K)对齐
    int    prot,   // 映射权限
    int    flags,  // 映射标志
    int    fd,     // 文件描述符
    off_t  offset  // 文件偏移量，自动按页(4K)对齐
);
```

成功返回映射区内存起始地址，失败返回MAP_FAILED(-1)。

prot取值：

PROT_EXEC - 映射区域可执行。

PROT_READ - 映射区域可读取。

PROT_WRITE - 映射区域可写入。

PROT_NONE - 映射区域不可访问。

flags取值：

MAP_FIXED   - 若在start上无法创建映射，
                则失败(无此标志系统会自动调整)。

MAP_SHARED   - 对映射区域的写入操作直接反映到文件中。

MAP_PRIVATE   - 对映射区域的写入操作只反映到缓冲区中，
                不会真正写入文件。

MAP_ANONYMOUS - 匿名映射，
                将虚拟地址映射到物理内存而非文件，
                忽略fd。

MAP_DENYWRITE - 拒绝其它对文件的写入操作。

MAP_LOCKED   - 锁定映射区域，保证其不被置换。

4. 销毁虚拟内存到物理内存或文件的映射

```
int munmap (
    void*  start,  // 映射区内存起始地址
    size_t length, // 字节长度，自动按页(4K)对齐
);
```

成功返回0，失败返回-1。

范例：mmap.c

mmap/munmap底层不维护任何东西，只是返回一个首地址，
所分配内存位于堆中。

brk/sbrk底层维护一个指针，记录所分配的内存结尾，
所分配内存位于堆中，底层调用mmap/munmap。

malloc底层维护一个双向链表和必要的控制信息，
不可越界访问，所分配内存位于堆中，底层调用brk/sbrk。

范例：malloc.c

每个进程都有4G的虚拟内存空间，
虚拟内存地址只是一个数字，
并没有和实际的物理内存将关联。
所谓内存分配与释放，
其本质就是建立或取消虚拟内存和物理内存间的映射关系。

作业：实现一个基于顺序表的堆栈类模板，
其数据缓冲区内存可根据数据元素的多少自动增减，
但不得使用标准C的内存分配与释放函数。

代码：stack.cpp、stack.c

思考：该堆栈模板是否适用于类类型的数据元素。