Technology Tarena 达内科技

Unix系统高级编程

Memory DAY03

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	内存管理与进程映射
	10:30 ~ 11:20	
	11:30 ~ 12:20	虚拟内存
下午	14:00 ~ 14:50	
	15:00 ~ 15:50	虚拟内存的分配与释放
	16:00 ~ 16:50	
	17:00 ~ 17:30	总结和答疑

内存管理与进程映射

Technology Tarena 达内科技

内存管理

内存管理

- 从底层硬件到上层应用，各层都提供了各自的内存管理接口

Technology Tarena 达内科技

进程映射

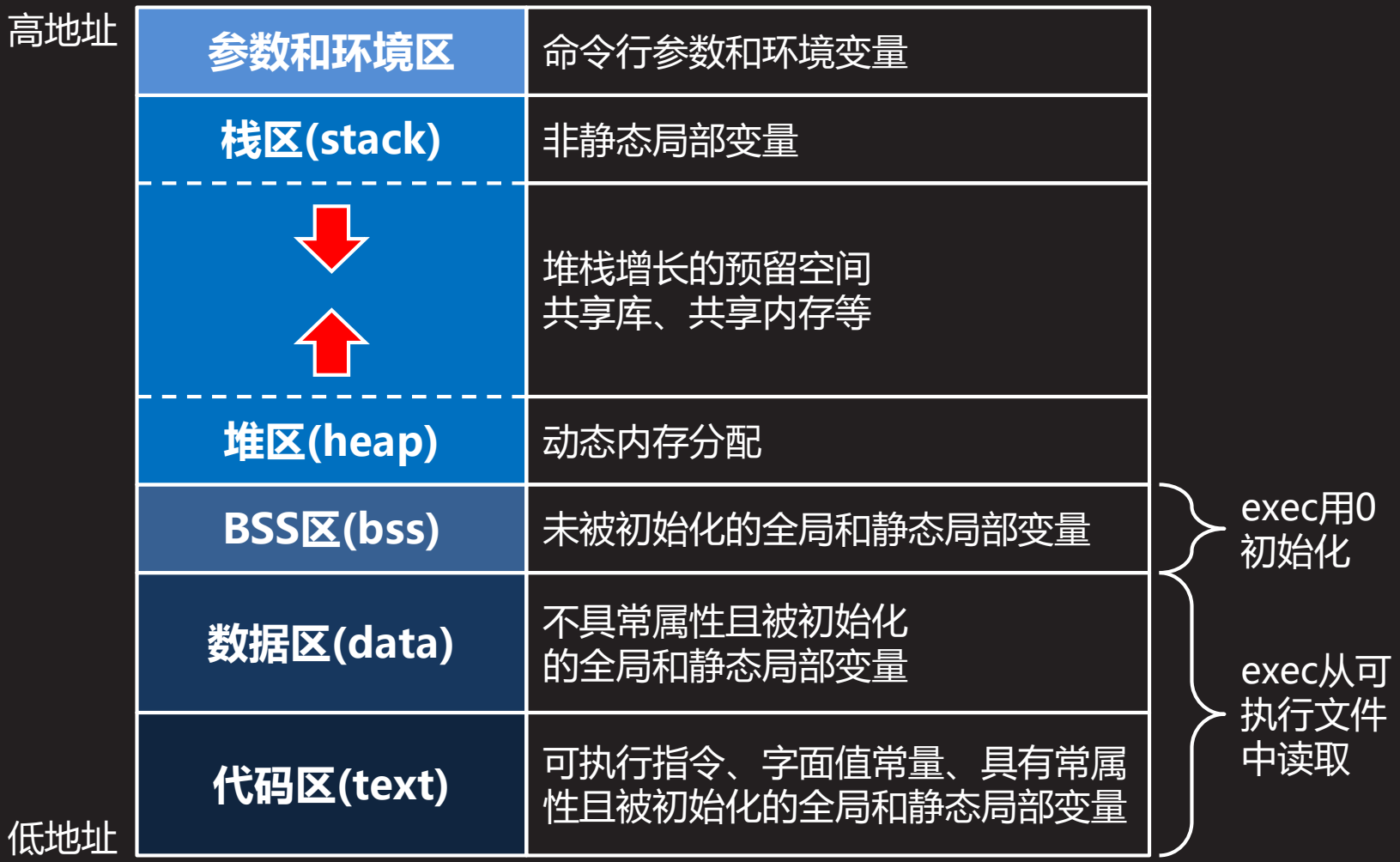
进程的内存布局

- 程序是保存在磁盘上的可执行文件
- 运行程序时，需要将可执行文件加载到内存，形成进程
- 一个程序(文件)可以同时存在多个进程(内存)
- 进程在内存空间中的布局形成进程映像，从低地址到高地址依次为
  - 代码区(text)
  - 数据区(data)
  - BSS区(bss)
  - 堆区(heap)
  - 栈区(stack)
  - 参数和环境区

知识讲解

进程的内存布局（续1）

- 基于Intel架构的Linux系统中的进程内存布局



知识讲解

进程的内存布局

【参见：maps.c】

课堂练习

- 进程的内存布局

进程映射文件

知识讲解

- 在/proc目录下可以看到很多以进程标识(PID)命名的子目录，这些子目录中存放的并不是真正的磁盘文件，而是操作系统内核为每个目前处于运行状态的进程，在内存中构建的审计数据结构，把它们挂载到文件系统中，只是为了方便系统管理人员实时监控系统的运行情况
  - 如/proc/2614目录下保存的就是2614进程的审计信息
- 在有关进程审计的诸多文件中，名为maps的文件反映了特定进程的内存映射情况，谓之进程映射文件

```
b738f000-b7533000 r-xp 00000000 08:01 983863 /lib/i386-
linux-gnu/libc-2.15.so
b76ff000-b7700000 r-xp 00000000 00:00 0      [vdso]
b89fd000-b8a1e000 rw-p 00000000 00:00 0      [heap]
bf906000-bf927000 rw-p 00000000 00:00 0      [stack]
```

进程映射文件（续1）

- 进程映像文件中的每一行均包括六列，从左至右依次为
  - 形如“bf8ff000-bf920000”的地址范围
  - 形如“rwxp”的访问权限，其中
    - `r` ：可读
    - `w` ：可写
    - `x` ：可执行
    - `s/p`：共享/私有
  - 形如“000f6000”的库内偏移地址
  - 形如“08:01”的映射文件设备号(主设备号:次设备号)
  - 形如“983863”的映射文件i节点号
  - 形如“/lib/i386-linux-gnu/libc-2. 15.so”的库路径

知识讲解

进程映射文件（续2）

- 进程映像文件中的每一行均包括六列，从左至右依次为
  - 上述最后一列也可能是[heap]、[stack]或者[vdso]

    `[heap]` ：堆

    `[stack]`：栈

    `[vdso]` ：虚拟动态共享库

知识讲解

size命令

- 通过size命令可以观察特定可执行程序的代码区(text)、数据区(data)和BSS区(bss)的大小，以字节为单位

```
$ size a.out
```

```
text    data    bss    dec    hex    filename
2628    268     28     2924   b6c    a.out
```

- Linux系统下的可执行程序、静态库和共享库的文件格式均采用ELF(Executable and Linkable Format，可执行可链接文件格式)标准，该文件格式同时也是SVR4和Solaris等Unix操作系统的默认目标文件格式
- 符合ELF标准的可执行程序文件只包含代码区和数据区的内容，在进程加载阶段被exec函数读入进程的内存空间

虚拟内存


虚拟内存
内存空间与地址空间
用户空间与内核空间
内存壁垒与段错误
标准内存分配函数
内存空间与地址空间
用户空间与内核空间
内存壁垒与段错误
标准内存分配函数

Technology Tarena 达内科技

内存空间与地址空间

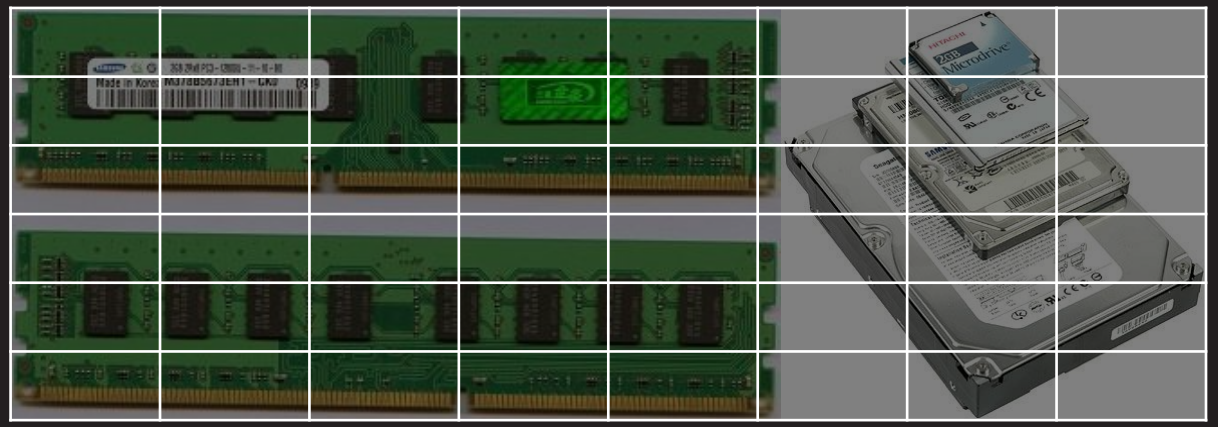
内存空间与地址空间

- 在系统中运行的每个进程，都拥有各自互独立的，4G字节大小的地址空间，谓之虚拟内存
- 用户程序中使用的都是地址空间中的虚拟内存，但真正存储代码和数据的却是永远无法被直接访问的物理内存
- 广义的物理内存不仅包括半导体内存，即通常所说的内存条，也包括磁盘上的交换分区(swap分区)或换页文件
- 当半导体内存不够用时，可以把一些长期闲置的代码和数据从半导体内存缓存到交换分区或换页文件中，这叫页面换出，一旦需要使用那些代码和数据，再把它们从交换分区或换页文件恢复到半导体内存中，这叫页面换入。因此，系统中的虚拟内存可以比半导体内存大很多

知识讲解

内存空间与地址空间（续1）

- 虚拟内存到物理内存，或者说地址空间到内存空间的映射关系，由操作系统内核通过内存映射表动态维护
- 虚拟内存技术一方面保护了系统的安全，把用户进程与内核进程，用户进程与用户进程，在对内存的访问上完全隔离开，另一方面又借助交换分区或换页文件，允许应用程序以一种完全透明的方式，使用比实际半导体内存更大的存储空间



内存空间与地址空间（续2）

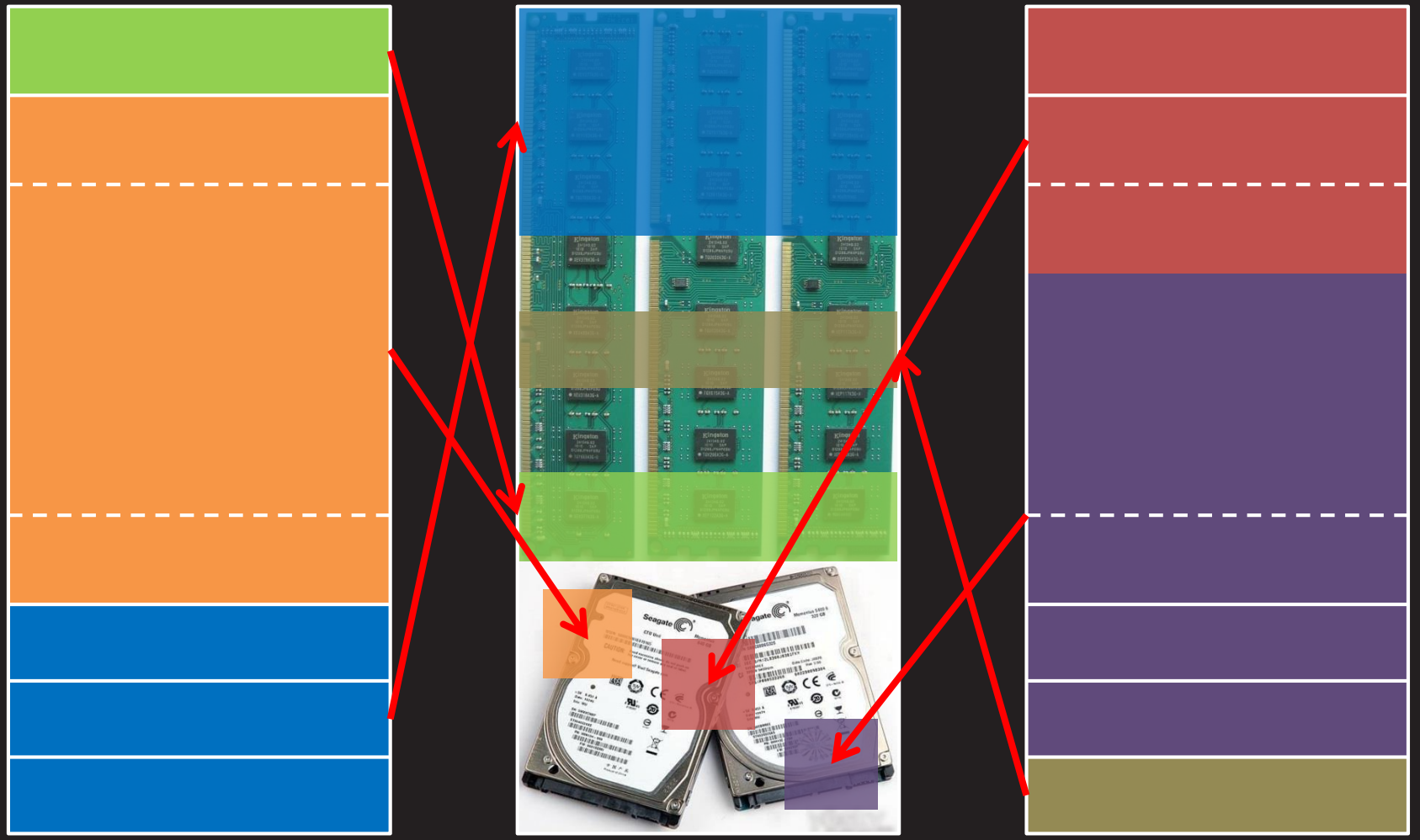
Technology Tarena 达内科技

知识讲解

进程A的地址空间
(虚拟内存)

系统的内存空间
(物理内存)

进程B的地址空间
(虚拟内存)



Technology Tarena 达内科技

用户空间与内核空间

用户空间与内核空间

Technology Tarena 达内科技

- 4G字节大小的进程地址空间分成两部分
  - 0~3G-1为用户空间，存放用户程序的代码和数据
    如某局部变量的地址0xbfc7fba0=3,217,554,336，约3G
  - 3G~4G-1为内核空间，存放系统内核的代码和数据
- 用户空间的代码不能直接访问内核空间的代码和数据，但可以通过系统调用进入内核态，间接地与系统内核交互

知识讲解

Technology Tarena 达内科技

内存壁垒与段错误

内存壁垒与段错误

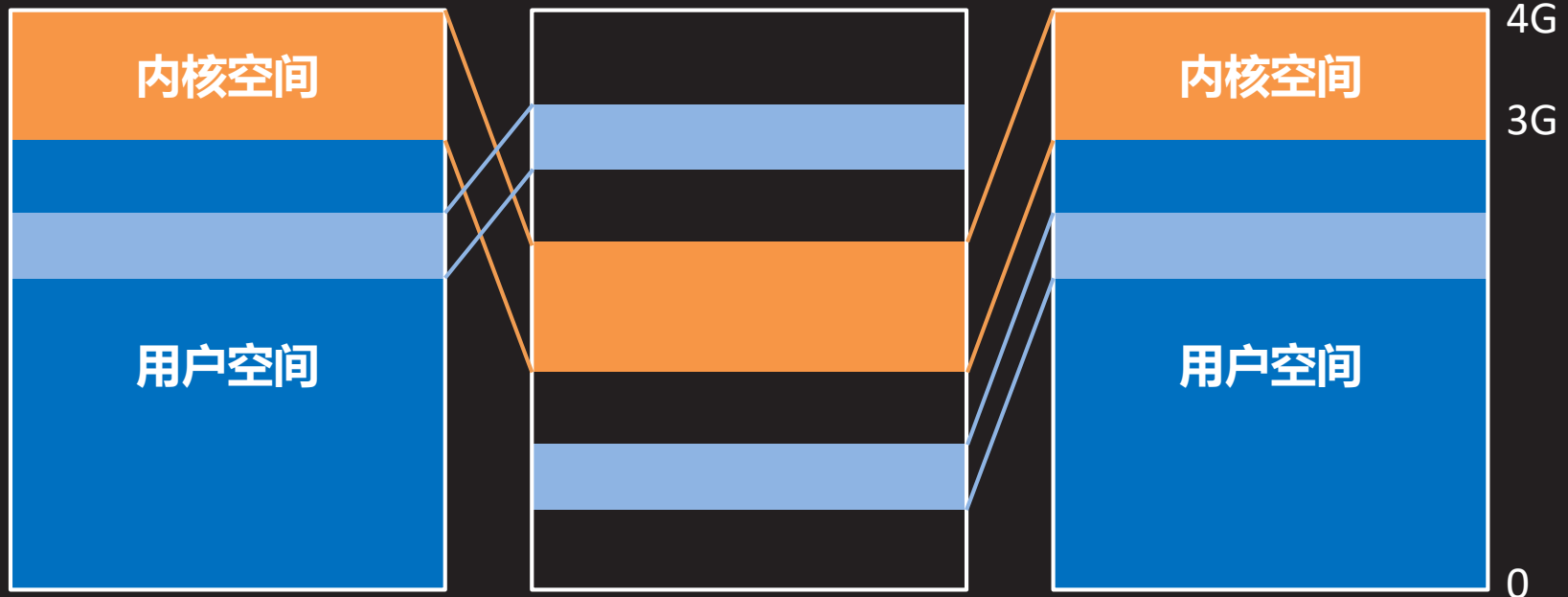
- 每个进程的用户空间都是0~3G-1，但它们所对应的物理内存却是各自独立的，系统为每个进程的用户空间维护一张专属于该进程的内存映射表，记录虚拟内存到物理内存的对应关系，因此在不同进程之间交换虚拟内存地址是毫无意义的
- 所有进程的内核空间都是3G~4G-1，它们所对应的物理内存只有一份，系统为所有进程的内核空间维护一张内存映射表init_mm.pgd，记录虚拟内存到物理内存的对应关系，因此不同进程通过系统调用所访问的内核代码和数据是同一份

知识讲解

内存壁垒与段错误（续1）

- 用户空间的内存映射表会随着进程的切换而切换，内核空间的内存映射表则无需随着进程的切换而切换
- 一切对虚拟内存的越权访问，都将导致段错误
  - 试图访问没有映射到物理内存的虚拟内存
  - 试图以非法方式访问虚拟内存，如对只读内存做写操作等

知识讲解



内存壁垒

【参见：vm.c】

课堂练习

- 内存壁垒

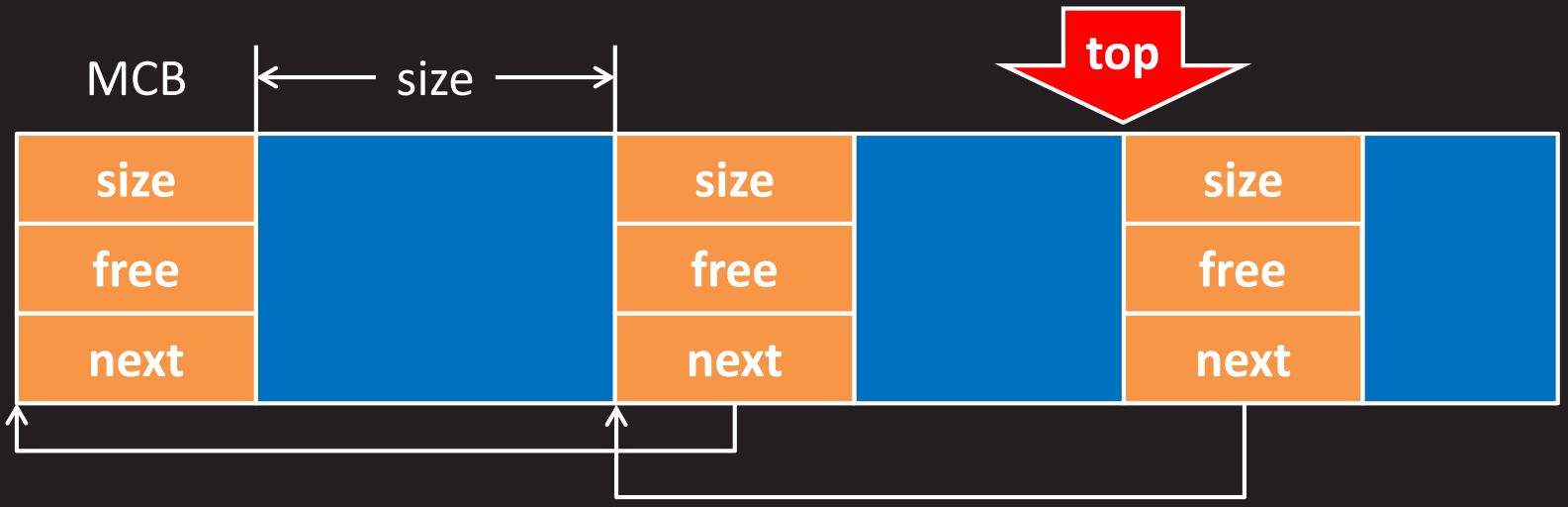
Technology Tarena 达内科技

标准内存分配函数

标准内存分配函数

- 标准库提供的内存分配函数(malloc/calloc/realloc)在标准库内部维护一个线性链表，管理堆中动态分配的内存
- 标准内存分配函数在分配内存时会附加若干(通常是12个)字节，存放控制信息(MCB，内存控制块)，该信息一旦被意外损坏，可能在后续操作中引发异常

知识讲解



标准内存分配函数（续1）

- 虚拟内存到物理内存的映射以页(4K=4096字节)为单位

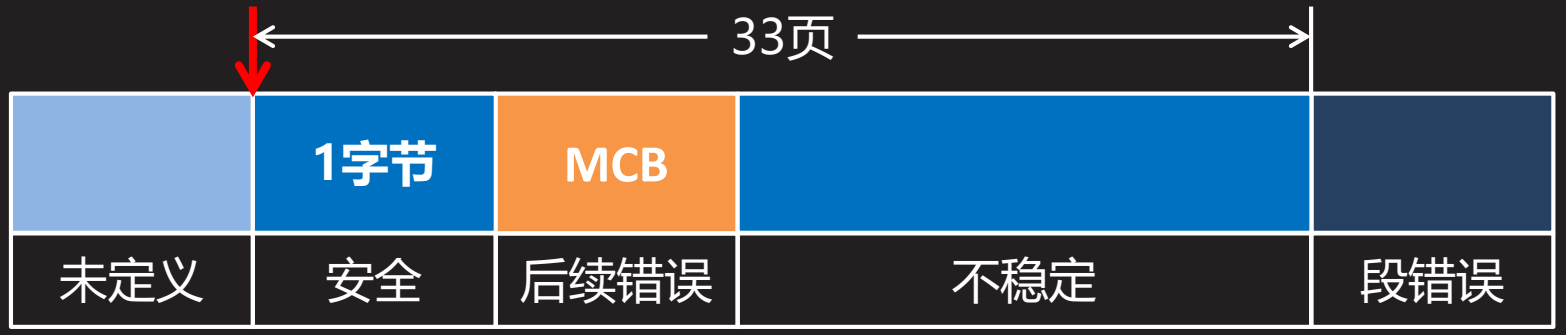
```
#include <unistd.h>

int getpagesize (void);
```

返回内存页的字节数

- 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

```
char* p = (char*)malloc (sizeof (char));
```



知识讲解

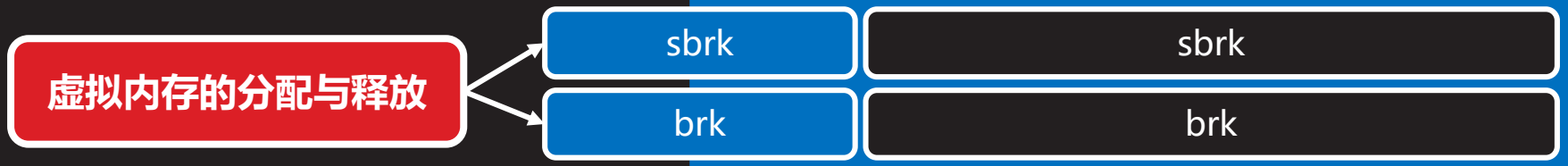
标准内存分配函数

【参见：page.c】

课堂练习

- 标准内存分配函数

虚拟内存的分配与释放



sbrk

sbrk

- 以相对方式分配和释放虚拟内存

```
#include <unistd.h>

void* sbrk (intptr_t increment);

        成功返回上次调用sbrk/brk后的堆尾指针，失败返回-1
```

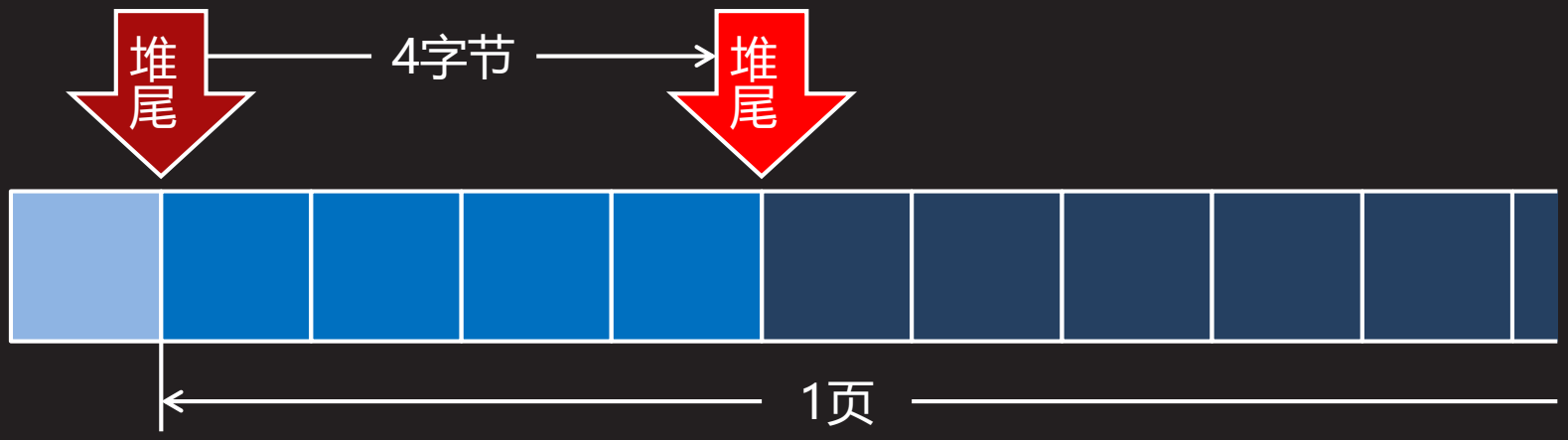
  - *increment*：虚拟内存增量(以字节为单位)
    - >0 - 分配虚拟内存
    - <0 - 释放虚拟内存
    - =0 - 当前堆尾指针

sbrk（续1）

- 系统内部维护一个指针，指向当前堆尾，即堆区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，同时返回该指针在调整前的位置，其间若发现内存页耗尽或空闲，则自动追加或取消内存页的映射

知识讲解

```
void* p = sbrk (4);          p = sbrk (0);
```



sbrk（续2）

- 例如

```
— p1 = sbrk ( 4); // BBBB ^--- ---- ---- -
  p2 = sbrk ( 4); // BBBB BBBB ^--- ---- -
  p3 = sbrk ( 4); // BBBB BBBB BBBB ^--- -
  p4 = sbrk ( 4); // BBBB BBBB BBBB BBBB ^
  p5 = sbrk ( 0); // BBBB BBBB BBBB BBBB ^
  p6 = sbrk (-8); // BBBB BBBB ^--- ---- _
  p7 = sbrk (-8); // ^--- ---- ---- ---- -
  // B: 已分配的字节
  // -: 未分配的字节
  // _: 函数返回指针
  // ^: 当前堆尾指针
```

知识讲解

Technology Tarena 达内科技

sbrk

【参见：brk.c】

课堂练习

- sbrk

Technology Tarena 达内科技

brk

brk

- 以绝对方式分配和释放虚拟内存

```
#include <unistd.h>

int brk (void* end_data_segment);
```

成功返回0，失败返回-1

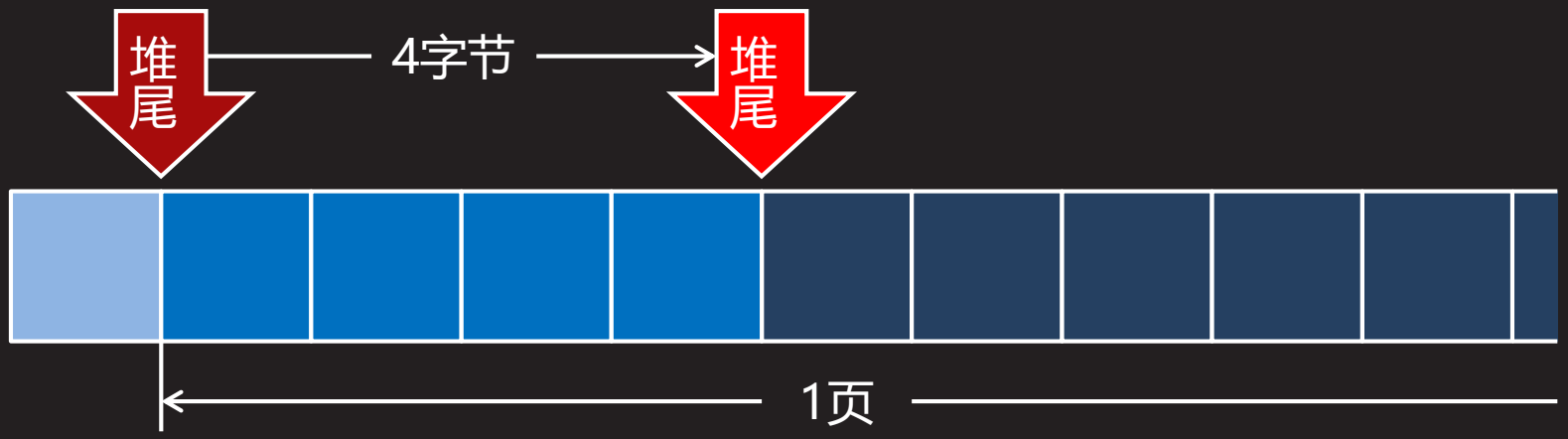
- *end_data_segment*：堆尾指针的新位置
  - >堆尾指针的原位置：分配虚拟内存
  - <堆尾指针的原位置：释放虚拟内存
  - =堆尾指针的原位置：什么也没有做

知识讲解

brk（续1）

- 系统内部维护一个指针，指向当前堆尾，即堆区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，其间若发现内存页耗尽或空闲，则自动追加或取消内存页的映射

```
void* p = sbrk (0);   brk (p+4);
```



知识讲解

brk（续2）

- 例如

```
— p1 = sbrk (0);      // ^--- ---- ---- ---- -
  brk (p2 = p1 + 4); // BBBB ^--- ---- ---- -
  brk (p3 = p2 + 4); // BBBB BBBB ^--- ---- -
  brk (p4 = p3 + 4); // BBBB BBBB BBBB ^--- -
  brk (p5 = p4 + 4); // BBBB BBBB BBBB BBBB ^
  brk (p3);          // BBBB BBBB ^--- ---- -
  brk (p1);          // ^--- ---- ---- ---- -
  // B：已分配的字节
  // -：未分配的字节
  // _：函数返回指针
  // ^：当前堆尾指针
```

brk（续3）

- sbrk和brk
  - 事实上，sbrk和brk不过是移动堆尾指针的两种不同方法，移动过程中还要兼顾虚拟内存和物理内存之间映射关系的建立和解除(以页为单位)
  - 用sbrk分配内存比较方便，用多少内存就传多少增量参数，同时返回指向新分配内存区域的指针，但用sbrk做一次性内存释放比较麻烦，因为必须将所有的既往增量进行累加
  - 用brk释放内存比较方便，只需将堆尾指针设回到一开始的位置即可一次性释放掉之前分多次分配的内存，但用brk分配内存比较麻烦，因为必须根据所需要的内存大小计算出堆尾指针的绝对位置
  - 用sbrk分多次分配适量内存，最后用brk一次性整体释放

brk

【参见：brk.c】

课堂练习

- brk

Technology Tarena 达内科技

总结和答疑