内存管理

张雷

南京大学计算机系

2018-11-27

Memory

Memory

Memory

Two types of memory:

- stack memory: allocations and deallocations managed implicitly by the compiler for the programmer; sometimes called automatic memory.

```
void func() {
    int x; // declares an integer on the stack
    ...
}
```

- heap memory: long-lived memory, all allocations and deallocations handled explicitly by the programmer

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

EVERY ADDRESS YOU SEE IS VIRTUAL

Any address you can see as a programmer of a user-level program is a virtual address:

- value of pointers in C programs
- locations of the main() function

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("location of code : %p\n", (void *) main);
    printf("location of heap : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);
    return x;
}
```

Example output on 64-bit Mac:

```
location of code : 0x1095afe50
location of heap : 0x1096008c0
location of stack : 0x7fff691aea64
```

Userspace Memory API

Allocating Memory on the Heap

```c
/*
 * Not necessary, as all C programs link with  C library by default.
 * Adding the header just lets the compiler check whether you are
 * calling  malloc() correctly (e.g., passing the right number
 * of arguments to it, of the right type).
 */
#include <stdlib.h>

/*Returns pointer to allocated memory on success, or NULL on error.*/
void *malloc(size_t size);
void free(void *ptr);
```

`size_t size`: how many bytes you need, use the compile-time operator: sizeof().

```c
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));

int x[10];
printf("%d\n", sizeof(x));
```

Kernel memory API

kmalloc() function is a simple interface for obtaining kernel memory in byte-sized chunks.

```
#include <linux/slab.h>
void* kmalloc(size_t size, gfp_t flags);
void kfree(const void *ptr);
```

Example usage:

```
struct dog *buf;
buf = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!buf)
    /* error allocating memory ! */

kfree(buf);
```

Common errors

- Forgetting To Allocate Memory

```
char *src = "hello";
char *dst; // oops! unallocated
strcpy(dst, src); // segfault and die
```

- Not Allocating Enough Memory

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src));
strcpy(dst, src); // work properly
```

- Forgetting To Free Memory
- Freeing Memory Repeatedly
- ...

Memory

GETTING PAGES

`struct page`: the kernel representation of a physical page, defined in `<linux/mm.h>`.

```
struct page {
    unsigned long flags; /* dirty, locked, ...*/
    atomic_t _count; /*the usage count of the page*/
    atomic_t _mapcount;
    unsigned long private;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
    void *virtual;
};
```

Each architecture enforces its own page size :

- Most 32-bit architectures have 4KB pages
- Most 64-bit architectures have 8KB pages

GETTING PAGES

All interfaces allocate memory with **page-sized granularity** are declared in <linux/gfp.h>

```
struct page* alloc_pages(gfp_t gfp_mask, unsigned int n)
```

- Allocates 2^n contiguous physical pages and returns a pointer to the first page's page structure; on error it returns NULL.

```
struct page * alloc_page(unsigned int gfp_mask)
```

- Same as `alloc_pages` with order 0

```
/*Allocates 2^n pages and returns a pointer to the first page's logical addr*/
unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int n);
/*Allocates a single page and returns a pointer to its logical addr*/
unsigned long __get_free_page(unsigned int gfp_mask);
/*Allocates a single page, zero its contents and returns a pointer to
  its logical addr*/
unsigned long get_zeroed_page(unsigned int gfp_mask);
```

FREEING PAGES

```
void __free_pages(struct page *page, unsigned int order);
void free_pages(unsigned long addr, unsigned int order);
void free_page(unsigned long addr);
```

You must be careful to free only pages you allocate.

- Passing the wrong struct page or address, or the incorrest order, can result in corruption

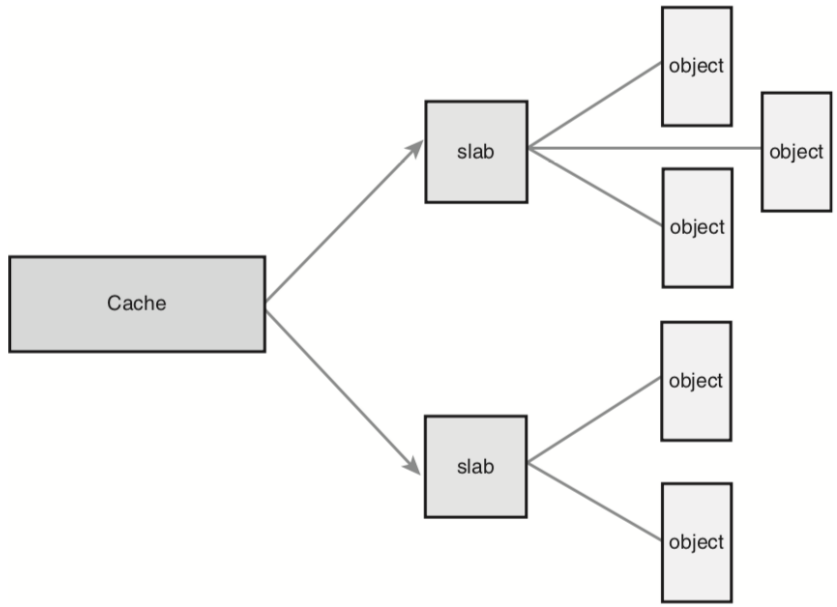
EXAMPLE

```
unsigned long page;
page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
    /* insufficient memory: you must handle this error! */
    return ENOMEM;
}
/* 'page' is now the address of the first of eight contiguous pa
free_pages(page, 3);
/*
 * our pages are now freed and we should no
 * longer access the address stored in 'page'
 */
```

Slab layer

The notion of object caching was therefore introduced in order to avoid the invocation of functions used to initialize object state.

- Cache: cache represents a small amount of very fast memory. **A cache is a storage for a specific type of object**, such as semaphores, process descriptors, file objects, etc.
- Slab: slab represents a contiguous piece of memory, usually **made of several physically contiguous pages**. The slab is the actual container of data associated with objects of the specific kind of the containing cache.



Slab Allocator

- Each slab contains some number of objects, cached data structures.
- Each slab is in one of three states: full, partial, or empty.
- When the kernel requests a new object,
  - the request is satisfied from a partial slab, if one exists
  - otherwise, it is satisfied from an empty slab, if one exists
  - otherwise, one empty slab is created
- The above strategy reduces **fragmentation**.

Jeff Bonwick,The Slab Allocator: An Object-Caching Kernel Memory Allocator.In Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1 (USTC'94)

Slab Allocator

```
struct slab {
    struct list_head list; /* full, partial, or empty list */
    unsigned long colouroff; /* offset for the slab coloring */
    void *s_mem; /* first object in the slab */
    unsigned int inuse; /* allocated objects in the slab */
    kmem_bufctl_t free; /* first free object, if any */
};
```

Slab Allocator Interface

Create a new cache:

```
kmem_cache_t *
kmem_cache_create(const char *name, size_t size,
                  size_t align, unsigned long flags,
                  void (*ctor)(void*, kmem_cache_t *, unsigned long),
                  void (*dtor)(void*, kmem_cache_t *, unsigned long));
```

Destroy a cache:

```
int kmem_cache_destroy(kmem_cache_t *cachep);
```

Obtain an object from a cache:

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags);
```

Free an object and return it to its originating slab:

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp);
```

Example of Using the Slab Allocator: kernel/fork.c

1. The kernel has a global variable storing a pointer to the `task_struct` cache:
   ```
   struct kmem_cache *task_struct_cachep;
   ```
2. During kernel initialization, in `fork_init()`, the cache is created:
   ```
   task_struct_cachep = kmem_cache_create("task_struct",
                                          sizeof(struct task_struct),
                                          ARCH_MIN_TASKALIGN,
                                          SLAB_PANIC,
                                          NULL,
                                          NULL);
   ```
3. Each time we calls `fork()`, a new process descriptor must be created. This is done in `dup_task_struct()`, called from `do_fork()`:
   ```
   struct task_struct *tsk;
   tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
   if (!tsk)
       return NULL;
   ```

Example of Using the Slab Allocator: kernel/fork.c

4. After a task dies, if it has no waiting children, its process descriptor is freed and returned to the `task_struct_cachep` slab cache. This is done in `free_task_struct()`

   ```
   kmem_cache_free(task_struct_cachep, tsk);
   ```

5. The `task_struct_cachep` cache is never destroyed! You can destroy the cache via:

   ```
   int err;
   err = kmem_cache_destroy(task_struct_cachep);
   if (err)
       /* error destroying cache */
   ```

/proc/slabinfo: memory usage on the slab level

```
root@localhost:~# cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit>
#                    <batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
UDPv6                 832    832   1216   26    8 : tunables    0    0    0 : slabdata     32     32      0
TCPv6                 434    434   2240   14    8 : tunables    0    0    0 : slabdata     31     31      0
kcopyd_job              0      0   3312    9    8 : tunables    0    0    0 : slabdata      0      0      0
dm_uevent               0      0   2632   12    8 : tunables    0    0    0 : slabdata      0      0      0
cfq_queue            1088   1088    240   34    2 : tunables    0    0    0 : slabdata     32     32      0
fat_inode_cache         0      0    736   44    8 : tunables    0    0    0 : slabdata      0      0      0
fat_cache               0      0     40  102    1 : tunables    0    0    0 : slabdata      0      0      0
ext4_inode_cache   294208 330120   1080   30    8 : tunables    0    0    0 : slabdata  11004  11004      0
pid_namespace           0      0   2232   14    8 : tunables    0    0    0 : slabdata      0      0      0
vm_area_struct     104240 104400    200   40    2 : tunables    0    0    0 : slabdata   2610   2610      0
mm_struct            6817   6832   2048   16    8 : tunables    0    0    0 : slabdata    427    427      0
task_struct          3085   3100   5632    5    8 : tunables    0    0    0 : slabdata    620    620      0
pid                 13856  13856    128   32    1 : tunables    0    0    0 : slabdata    433    433      0
kmalloc-1024        15636  15840   1024   32    8 : tunables    0    0    0 : slabdata    495    495      0
kmalloc-512         22912  23040    512   32    4 : tunables    0    0    0 : slabdata    720    720      0
kmalloc-256         18695  19648    256   32    2 : tunables    0    0    0 : slabdata    614    614      0
kmalloc-192         17825  18060    192   42    2 : tunables    0    0    0 : slabdata    430    430      0
kmalloc-128          7540   7584    128   32    1 : tunables    0    0    0 : slabdata    237    237      0
kmalloc-64          52234  53568     64   64    1 : tunables    0    0    0 : slabdata    837    837      0
kmalloc-32          62751  64128     32  128    1 : tunables    0    0    0 : slabdata    501    501      0
kmalloc-16          31744  31744     16  256    1 : tunables    0    0    0 : slabdata    124    124      0
kmalloc-8           24064  24064      8  512    1 : tunables    0    0    0 : slabdata     47     47      0
```

/usr/bin/slabtop: kernel slab cache info in real time

```
Active / Total Objects (% used)    : 13721309 / 13945916 (98.4%)
Active / Total Slabs (% used)      : 350954 / 350954 (100.0%)
Active / Total Caches (% used)     : 76 / 119 (63.9%)
Active / Total Size (% used)       : 2001105.93K / 2076581.61K (96.4%)
Minimum / Average / Maximum Object : 0.01K / 0.15K / 18.62K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
11303253 11233971   0%    0.10K 289827       39   1159308K buffer_head
466998 375403   0%    0.19K  11119       42     88952K dentry
362508 361982   0%    0.04K   3554      102     14216K ext4_extent_status
346416 328247   0%    0.57K  12372       28    197952K radix_tree_node
330150 294238   0%    1.05K  11005       30    352160K ext4_inode_cache
134016 134016 100%    0.06K   2094       64      8376K anon_vma_chain
104440 104280   0%    0.20K   2611       40     20888K vm_area_struct
 78285  78285 100%    0.08K   1535       51      6140K anon_vma
 64128  62842   0%    0.03K    501      128      2004K kmalloc-32
 53568  52422   0%    0.06K    837       64      3348K kmalloc-64
 51462  51062   0%    0.59K    953       54     30496K inode_cache
 34176  32158   0%    0.66K    712       48     22784K proc_inode_cache
 31744  31744 100%    0.02K    124      256       496K kmalloc-16
 24064  24064 100%    0.01K     47      512       188K kmalloc-8
 23040  22966   0%    0.50K    720       32     11520K kmalloc-512
 22264  22220   0%    0.69K    484       46     15488K sock_inode_cache
 19648  18528   0%    0.25K    614       32      4912K kmalloc-256
 18060  17812   0%    0.19K    430       42      3440K kmalloc-192
 16744  16744 100%    0.69K    364       46     11648K files_cache
 16192  15936   0%    0.06K    253       64      1012K ext4_io_end
 15840  15622   0%    1.00K    495       32     15840K kmalloc-1024
 13856  13856 100%    0.12K    433       32      1732K pid
 11968  11968 100%    1.00K    374       32     11968K signal_cache
 10718  10378   0%    0.69K    233       46      7456K shmem_inode_cache
  7584   7540   0%    0.12K    237       32       948K kmalloc-128
  6832   6817   0%    2.00K    427       16     13664K mm_struct
```

Memory

Memory management outside the kernel: process address space

- struct mm: memory descriptor (mm_types.h)
- struct `vm_area_struct` mmap: vma (mm_types.h)
- struct page: page descriptor (mm_types.h)
- pgd, pud, pmd, pte: pgtable entries (arch/x86/include/asm/page.h, page_32.h, pgtable.h, pgtable_32.h)
  - pgd: page global directory
  - pud page upper directory
  - pmd: page middle directory
  - pte: page table entry
- struct anon_vma: anon vma reverse map (rmap.h)
- struct prio_tree_root i_mmap: priority tree reverse map (fs.h)
- struct radix_tree_root page_tree: page cache radix tree (fs.h)

The Memory Descriptor: struct mm_struct

```
struct mm_struct {
    struct vm_area_struct *mmap; /* list of memory areas */
    struct rb_root mm_rb; /* red-black tree of memory areas */
    struct vm_area_struct *mmap_cache; /* last used memory area */
    unsigned long free_area_cache; /* 1st address space hole */
    pgd_t *pgd; /* page global directory */
    atomic_t mm_users; /* address space users */
    atomic_t mm_count; /* primary usage counter */
    int map_count; /* number of memory areas */
    struct rw_semaphore mmap_sem; /* memory area semaphore */
    spinlock_t page_table_lock; /* page table lock */
    struct list_head mmlist; /* list of all mm_structs,所有的mm_struct结构通过mmlist域链接在一个双向链表上。
    unsigned long start_code; /* start address of code */
    unsigned long end_code; /* final address of code */
    unsigned long start_data; /* start address of data */
    unsigned long end_data; /* final address of data */
    unsigned long start_brk; /* start address of heap */
    unsigned long brk; /* final address of heap */
    unsigned long start_stack; /* start address of stack */
    unsigned long arg_start; /* start of arguments */
    unsigned long arg_end; /* end of arguments */
    unsigned long env_start; /* start of environment */
    unsigned long env_end; /* end of environment */
    unsigned long rss; /* pages allocated */
    unsigned long total_vm; /* total number of pages */
    unsigned long locked_vm; /* number of locked pages */
    unsigned long saved_auxv[AT_VECTOR_SIZE]; /* saved auxv */
    cpumask_t cpu_vm_mask; /* lazy TLB switch mask */
    mm_context_t context; /* arch-specific data */
    unsigned long flags; /* status flags */
    int core_waiters; /* thread core dump waiters */
    struct core_state *core_state; /* core dump support */
    spinlock_t ioctx_lock; /* AIO I/O list lock */
    struct hlist_head ioctx_list; /* AIO I/O list */
};
```

struct mm_struct

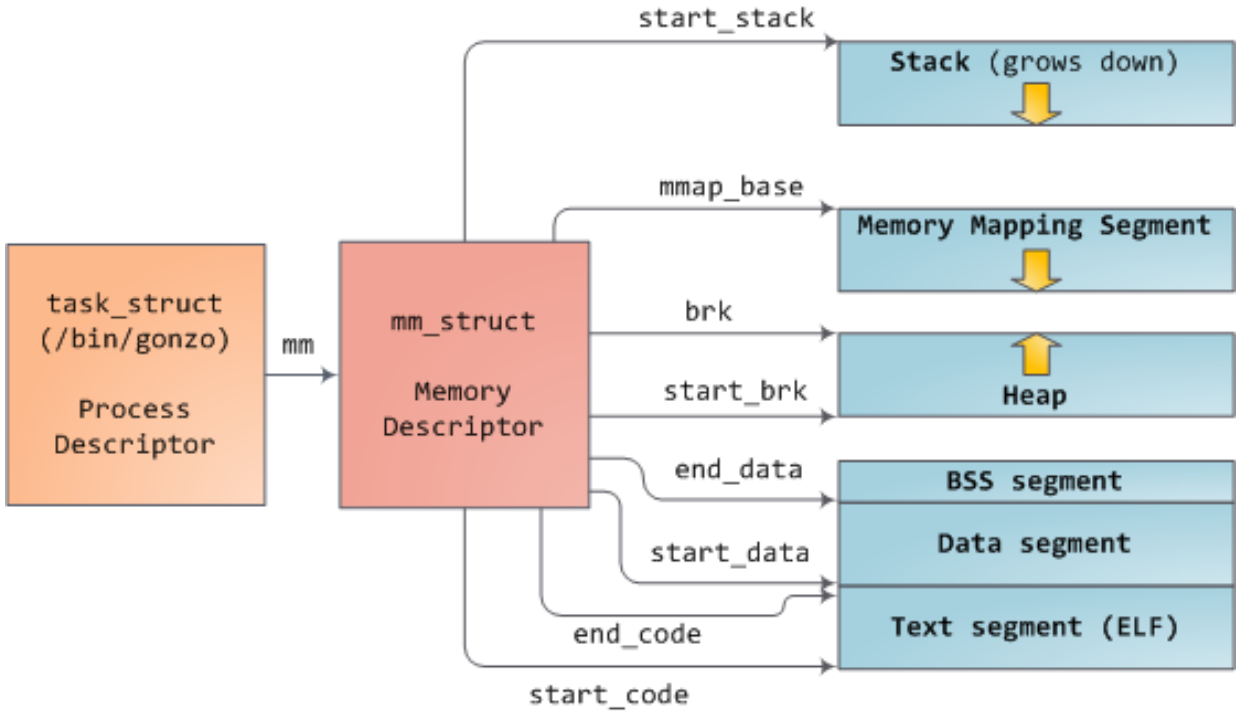


Figure 1: Memory Descriptor

内存描述符的分配

- fork()中通过调用copy_mm()从父进程拷贝内存描述符
- mm_struct结构的分配通过宏allocate_mm()从mm_cachep指向的slab缓存中分配得到
- 如果父进程在调用fork()创建子进程时，通过一些标志指明需要与父进程共享地址空间，只需要childtask->mm = parent->mm

```
static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
        struct mm_struct * mm, *oldmm;
        int retval;
        tsk->mm = NULL;
        tsk->active_mm = NULL;

        /* Are we cloning a kernel thread? We need to steal a active VM for that..*/
        oldmm = current->mm;
        if (!oldmm) return 0;

        if (clone_flags & CLONE_VM) {
                atomic_inc(&oldmm->mm_users);
                mm = oldmm;
                goto good_mm;
        }

        retval = -ENOMEM;
        mm = dup_mm(tsk);
        if (!mm) goto fail_nomem;
good_mm:
        tsk->mm = mm;
        tsk->active_mm = mm;
```

Virtual Memory Areas: `struct vm_area_struct`

```
struct vm_area_struct {
    struct mm_struct * vm_mm; /* The address space we belong to. */
    unsigned long vm_start; /* Our start address within vm_mm.*/
    unsigned long vm_end;
    struct vm_area_struct *vm_next; /* list of VMA's */
    pgprot_t vm_page_prot; /* Access permissions of this VMA. */
    unsigned long vm_flags; /* Flags, see mm.h. */
    struct rb_node vm_rb; /* VMA's node in the tree */
    struct raw_prio_tree_node prio_tree_node;
    struct list_head anon_vma_node; /* Serialized by anon_vma->lock */
    struct anon_vma *anon_vma; /* Serialized by page_table_lock */
    struct vm_operations_struct * vm_ops; /* associated ops */
    unsigned long vm_pgoff;
    struct file * vm_file; /* File we map to (can be NULL). */
    void * vm_private_data; /* was vm_pte (shared mem) */
};
```

struct vm_area_struct

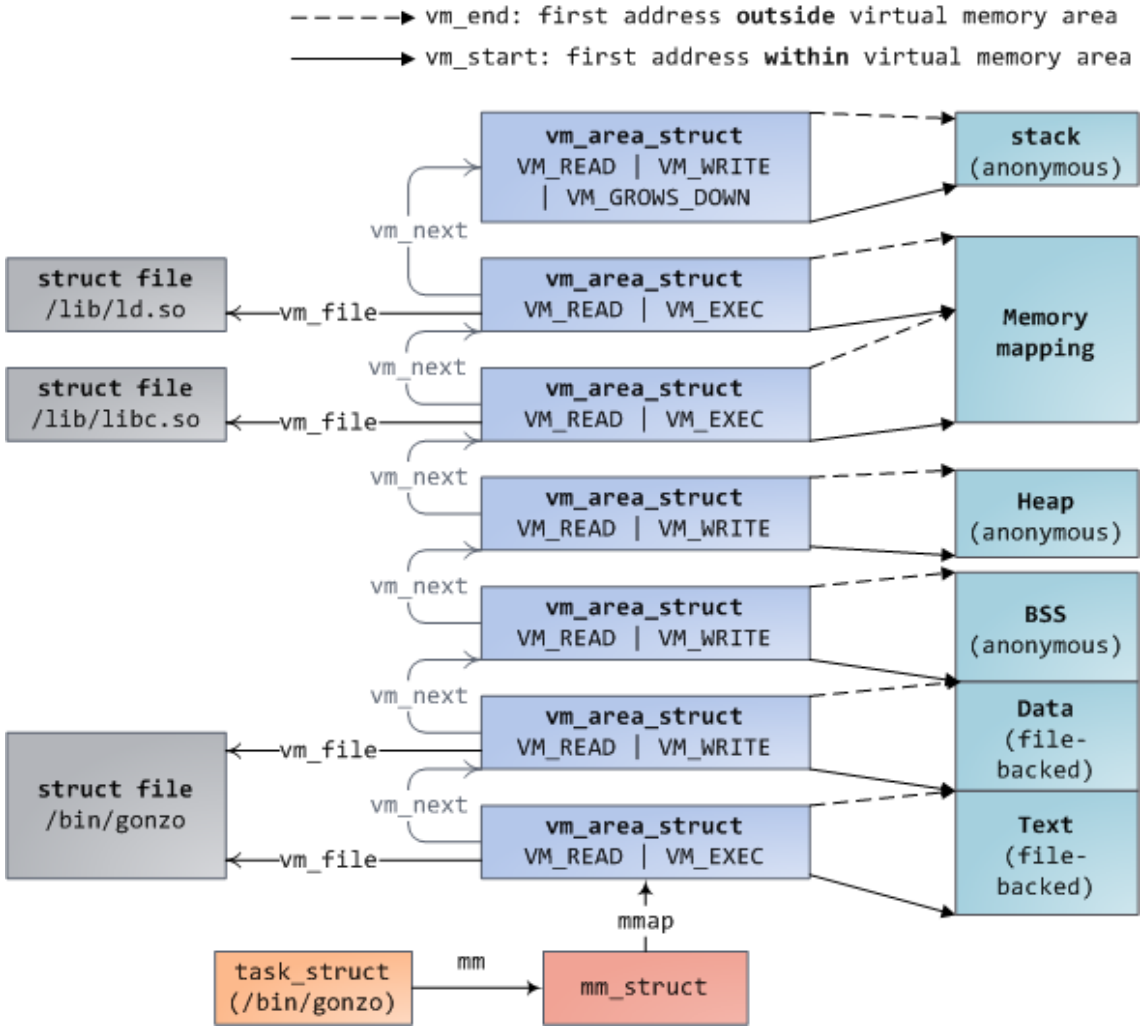


Figure 2: Memory Areas
https://manybutfinite.com/post/how-the-kernel-manages-your-memory

struct vm_area_struct in real life

```
int main(int argc, char *argv[])  { return 0; }
```

```
$ cat /proc/1426/maps
00e80000-00faf000 r-xp 00000000 03:01 208530 /lib/tls/libc-2.5.1.so
00faf000-00fb2000 rw-p 0012f000 03:01 208530 /lib/tls/libc-2.5.1.so
00fb2000-00fb4000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 03:03 439029 /home/rlove/src/example
08049000-0804a000 rw-p 00000000 03:03 439029 /home/rlove/src/example
40000000-40015000 r-xp 00000000 03:01 80276  /lib/ld-2.5.1.so
40015000-40016000 rw-p 00015000 03:01 80276  /lib/ld-2.5.1.so
4001e000-4001f000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

```
$ pmap 1426
00e80000    (1212 KB)   r-xp    (03:01 208530)  /lib/tls/libc-2.5.1.so
00faf000    (12 KB)     rw-p    (03:01 208530)  /lib/tls/libc-2.5.1.so
00fb2000    (8 KB)      rw-p    (00:00 0)
08048000    (4 KB)      r-xp    (03:03 439029)  /home/rlove/src/example
08049000    (4 KB)      rw-p    (03:03 439029)  /home/rlove/src/example
40000000    (84 KB)     r-xp    (03:01 80276)   /lib/ld-2.5.1.so
40015000    (4 KB)      rw-p    (03:01 80276)   /lib/ld-2.5.1.so
4001e000    (4 KB)      rw-p    (00:00 0)
bfffe000    (8 KB)      rwxp        (00:00 0)   [ stack ]
mapped: 1340 KB writable/private: 40 KB shared: 0 KB
```

struct vm_operations_struct

```
struct vm_operations_struct {
    /*invoked when the given memory area is added to an address space.*/
    void (*open)(struct vm_area_struct * area);
    /*invoked when the given memory area is removed from an address space.*/
    void (*close)(struct vm_area_struct * area);
    /* invoked by the page fault handler when a page that is not present
       in physical memory is accessed.*/
    int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
    /* notification that a previously read-only page is about to become
    * writable, if an error is returned it will cause a SIGBUS */
    int (*page_mkwrite)(struct vm_area_struct *vma, struct page *page);
    /* called by access_process_vm when get_user_pages() fails, typically
    * for use by special VMAs that can switch between memory and hardware
    */
    int (*access)(struct vm_area_struct *vma, unsigned long addr,
     void *buf, int len, int write);
};
```

Manipulating Memory Areas

The kernel often has to perform operations on a memory area, e.g. checking whether a given address exists in a given VMA. These operations are frequent and form the basis of the mmap() routine. Helper functions are defined to assist these jobs.

```
#include <linux/mm.h>

struct vm_area_struct * find_vma(struct mm_struct *mm, unsigned long addr);

struct vm_area_struct * find_vma_prev(struct mm_struct *mm, unsigned long addr,
                                      struct vm_area_struct **pprev);

static inline struct vm_area_struct *
find_vma_intersection(struct mm_struct *mm,
                      unsigned long start_addr,
                      unsigned long end_addr)
{
    struct vm_area_struct *vma;

    vma = find_vma(mm, start_addr);
    if (vma && end_addr <= vma->vm_start)
    vma = NULL;
    return vma;
}
```

find_vma: searching for the VMA a given address resides

```
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
{
    struct vm_area_struct *vma = NULL;

    if (mm) {
        /* Check the cache first. */
        /* (Cache hit rate is typically around 35%.) */
        vma = mm->mmap_cache;
        if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
            struct rb_node * rb_node;

            rb_node = mm->mm_rb.rb_node;
            vma = NULL;

            while (rb_node) {
                struct vm_area_struct * vma_tmp;

                vma_tmp = rb_entry(rb_node,
                        struct vm_area_struct, vm_rb);

                if (vma_tmp->vm_end > addr) {
                    vma = vma_tmp;
                    if (vma_tmp->vm_start <= addr)
                        break;
                    rb_node = rb_node->rb_left;
                } else
                    rb_node = rb_node->rb_right;
            }
            if (vma)
                mm->mmap_cache = vma;
        }
    }
    return vma;
}
```

Creating an Address Interval

Types of VMA Mappings:

- File/device backed mappings (mmap):
  - Code pages (binaries), libraries
  - Data files
  - Shared memory
  - Devices
- Anonymous mappings:
  - Stack
  - Heap
  - CoW pages

Creating an Address Interval: mmap() and do_mmap()

```
void* mmap (void *addr, size t len, int prot,
            int flags, int fd, off_t offset);
```

- Map file specified by `fd` at virtual address `addr`
- If `addr` is NULL, let kernel choose the address
- prot: protection of region
  - OR of `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`
- flags:
  - `MAP_ANON` anonymous memory (fd should be -1);
  - `MAP_PRIVATE` modifications are private;
  - `MAP_SHARED` modifications seen by everyone

```
unsigned long do_mmap(struct file *file, unsigned long addr,
                      unsigned long len, unsigned long prot,
                      unsigned long flag, unsigned long offset);
```

File mappings

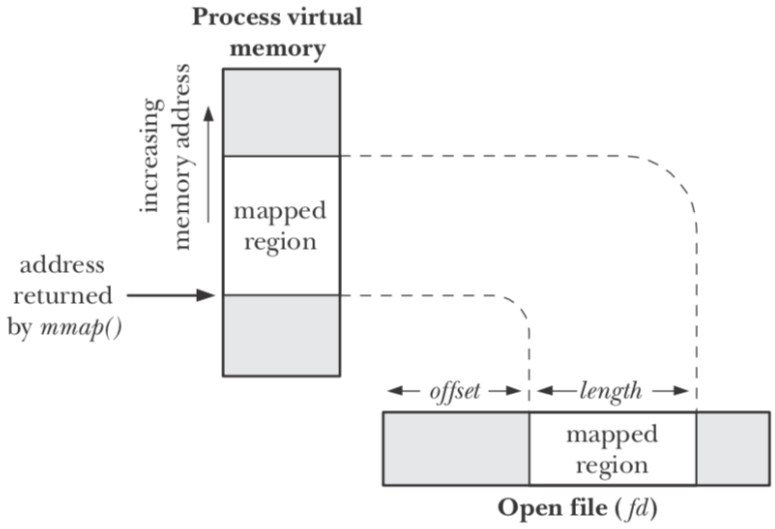


Figure 3: Overview of memory-mapped file

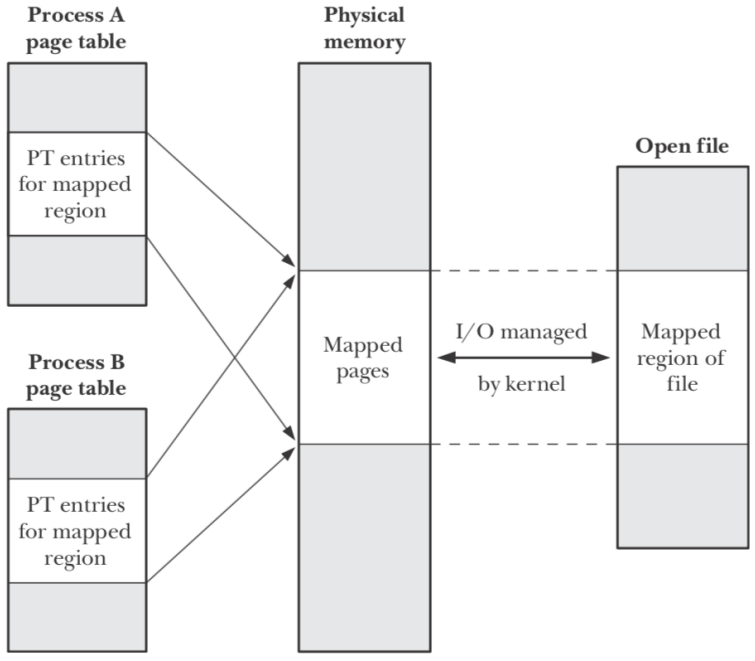


Figure 4: Two processes with a shared mapping of the same region of a file. IPC.

munmap() and do_munmap(): Removing an Address Interval

```
/*removing a mapping from the calling process's virtual address
int munmap(void *start, size_t length);
int do_munmap(struct mm_struct *mm, unsigned long start,
              size_t len);

asmlinkage long sys_munmap(unsigned long addr, size_t len)
{
    int ret;
    struct mm_struct *mm;
    mm = current->mm;
    down_write(&mm->mmap_sem);
    ret = do_munmap(mm, addr, len);
    up_write(&mm->mmap_sem);
    return ret;
}
```

mmap() example

```c
#include <sys/mman.h>
/* also stdio.h, stdlib.h, string.h, fcntl.h, unistd.h */

int something = 162;

int main (int argc, char *argv[]) {
    int myfd;
    char *mfile;

    printf("Data  at: %16lx\n", (long unsigned int) &something);
    printf("Heap at : %16lx\n", (long unsigned int) malloc(1));
    printf("Stack at: %16lx\n", (long unsigned int) &mfile);

    /* Open the file */
    myfd = open(argv[1], O_RDWR | O_CREAT);
    if (myfd < 0) { perror("open failed!");exit(1); }

    /* map the file */
    mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED, myfd, 0);
    if (mfile == MAP_FAILED) {perror("mmap failed"); exit(1);}

    printf("mmap at : %16lx\n", (long unsigned int) mfile);

    puts(mfile);
    strcpy(mfile+20,"Let's write over it");
    close(myfd);
    return 0;
}
```

mmap() example

Example output:

```
$ ./mmap test
Data  at:        105d63058
Heap at :     7f8a33c04b70
Stack at:     7fff59e9db10
mmap at :        105d97000
This is line one
This is line two
This is line three
This is line four

$ cat test
This is line one
ThiLet's write over its line three
This is line four
```

In Linux, you can open up a raw block device just by opening a file like `/dev/hda1` and use `mmap()` straight from there, so this gives **database implementors** a way to control the whole disk with the same interface. This is great if you're a typical database developer who doesn't like the OS and doesn't trust the file system.

- MongoDB,MonetDB,LMDB...

内核态和用户态分配内存的差别

- 内核函数可直接获得动态内存
  - 内核是操作系统中优先级最高的成分，内核信任自己
  - 采用页面级内存分配和小内存分配
- 用户态进程分配内存时
  - 请求被认为是不紧迫的，用户进程不可信任
  - 不能立即获得实际物理页框，仅获得对一个新线性地址区间的使用权
  - 该线性地址区间会成为进程地址空间的一部分，称作内存区域

Page Tables

- 虚存与实存的映射:通过查询页表完成
  - 将虚拟地址分段，每段作为一个索引指向页表
  - 页表则指向下一级别的页表或指向最终物理页面
- Linux使用三级页表完成地址转换

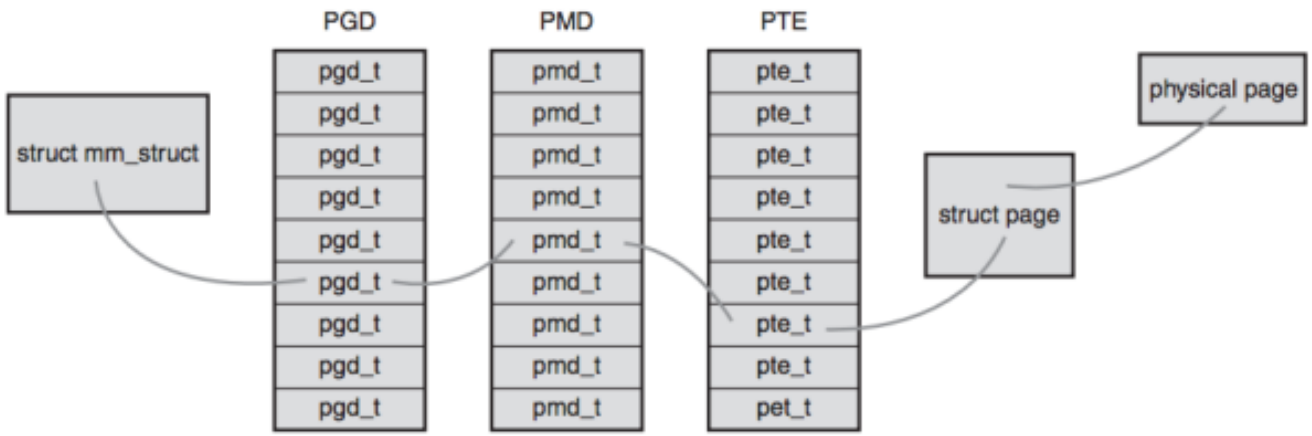


Figure 5: virtual to physical address lookup using page tables

请页机制

进程运行时，CPU访问的是用户空间的虚地址，Linux仅把当前要使用的少量页面装入内存，需要时再通过请页机制将特定的页面调入内存,当要访问的虚页不在内存时，产生一个页故障并报告故障原因

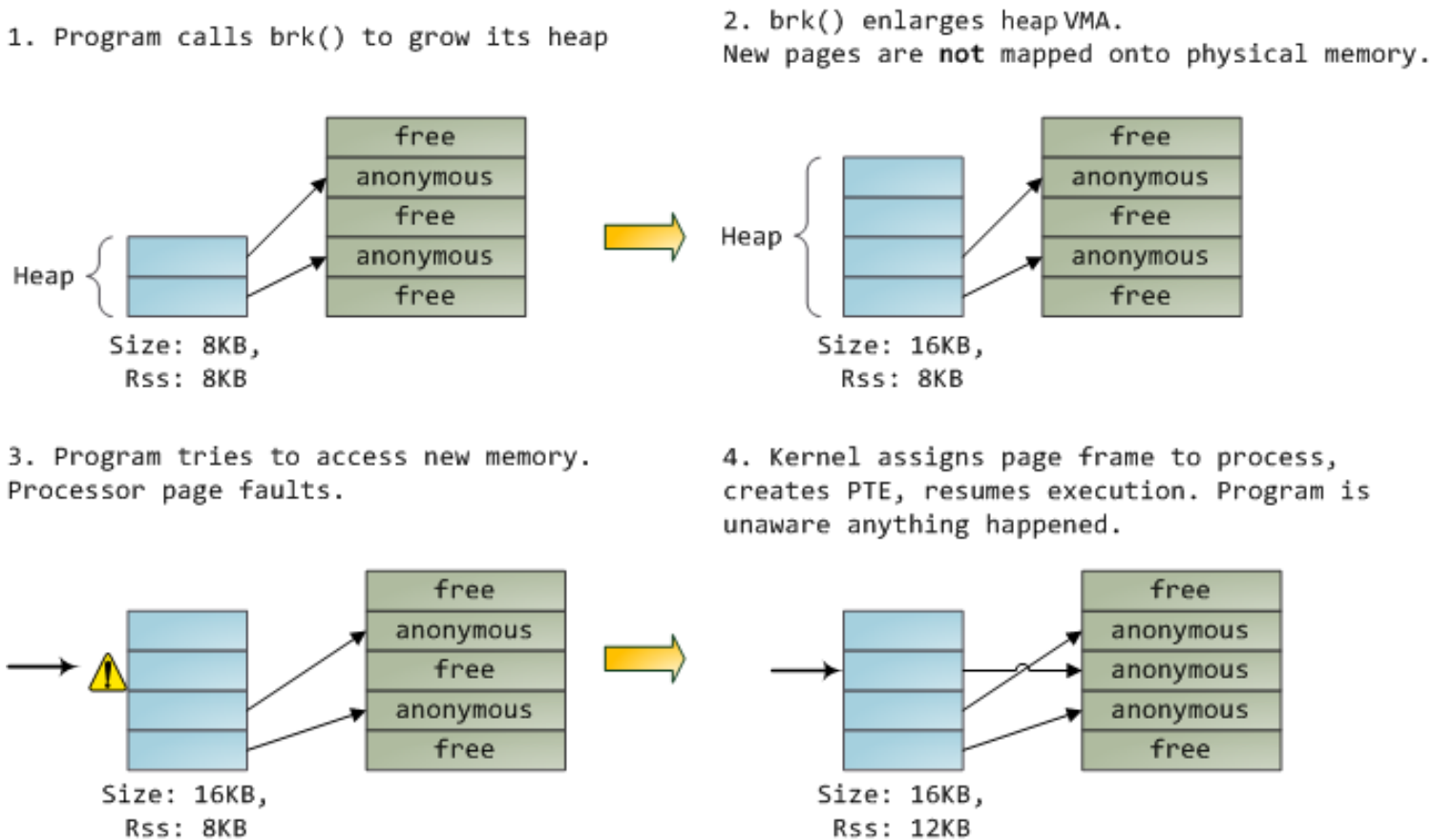


Figure 6: Heap Memory Allocation: malloc

页故障原因

- 程序出现错误: 如虚地址在PAGE_OFFSET（3GB）之外，则该地址无效，Linux将向进程发送一个信号并终止进程的运行
- 缺页异常
  - 虚地址有效，但其所对应的页当前不在物理内存中
  - 操作系统必须从磁盘或交换文件（此页被换出）中将其装入物理内存
- 保护错误
  - 要访问的虚地址被写保护
  - 操作系统必须判断如果是某个用户进程正在写当前进程的地址空间，则发送一个信号并终止进程的运行
  - 如果错误发生在旧共享页上时，则处理方法有所不同需要要对该共享页进行复制，即“写时复制”(CoW)技术

Copy on Write

(Ab)use page protection, mark pages as read-only in both parent and child address space, on write, page fault occurs.

- PTE entry is marked as un-writeable
- But VMA is marked as writeable
- Page fault handler notices difference
  - Must mean CoW
  - Make a duplicate of physical page
  - Update PTEs, flush TLB entry
  - `do_wp_page`

Memory

Summary

Linux memory management

- kernel memory:
  - `kmalloc`: byte-sized
  - `alloc_pages`: page-sized
  - `Slab layer`: generic data structure-caching layer
- user memory:
  - `struct mm_struct`, `struct vm_area_struct`: virtual address space, memory areas
  - `mmap`: create memory areas/mappings
- ...

Memory

Lab4: A Multicore Round-Robin Scheduler

Add a new scheduling policy to the Linux kernel to support weight round-robin scheduling. Call this policy WRR. The algorithm should run in constant time and work as follows:

- The new scheduling policy should serve as the default scheduling policy for init and all of its descendants.
- Multicore systems must be fully supported. Every task has a time slice (quantum), which is a multiple of a base 10ms time slice.
- The multiple should be the weight of the process which can be set using Linux scheduling system calls.
- When deciding which CPU a task should be assigned to, it should be assigned to the CPU with the least total weight. This means you have to keep track of the total weight of all the tasks running on each CPU. If there are multiple CPUs with the same weight, you should assign the task to the CPU with higher affinity, where affinity is measured by the number of other tasks assigned to the CPU that are part of the same thread group.

Lab4: A Multicore Round-Robin Scheduler

- Your scheduler should operate alongside the existing Linux scheduler. Therefore, you should add a new scheduling policy, `SCHED_WRR`. The value of `SCHED_WRR` should be 7. `SCHED_WRR` should be made the default scheduler class of init.
- Only tasks whose policy is set to `SCHED_WRR` should be considered for selection by your new scheduler.
- Tasks using the `SCHED_WRR` policy should take priority over tasks using the `SCHED_NORMAL` policy, but not over tasks using the `SCHED_RR` or `SCHED_FIFO` policies.
- Your scheduler must be capable of working on both uniprocessor systems and multicore/multiprocessor systems; you can change the number of cores on your VM for testing. All cores should be utilized on multiprocessor systems.
- Proper synchronization and locking is crucial for a multicore scheduler, but not easy. Pay close attention to the kind of locking used in existing kernel schedulers. This is extremely critical for the next part of the assignment. It is very easy to overlook a key synchronization aspect and cause a deadlock!

Lab4: A Multicore Round-Robin Scheduler

- For a more responsive system, you may want to set the scheduler of kernel threads to be `SCHED_WRR` as well (otherwise, `SCHED_WRR` tasks can starve the `SCHED_NORMAL` tasks to a degree). To do this, you can modify kernel/kthread.c and replace `SCHED_NORMAL` with `SCHED_WRR`. It is strongly suggested that you do this to ensure that your VM is responsive enough for the test cases.

Investigate and Demo

- Demonstrate that your scheduler effectively balances weights across all CPUs. The total weight of the processes running on each CPU should be roughly the same. You should start with a CPU-bound test program such as a while loop then try running more complex programs that do I/O to show that the scheduler continues to operate correctly. A test program that calls the system call should be included in the test directory. Experiment with different weights and see what impact they have.

Thanks!

基础实验楼乙126