ab1214 L446 2019-07-14 19:56:52 地址映射与共享

“操作系统原理与实践”实验报告

地址映射与共享

第七次实验报告

1. 简介

深入理解操作系统的段、页式内存管理，深入理解段表、页表、逻辑地址、线性地址、物理地址。

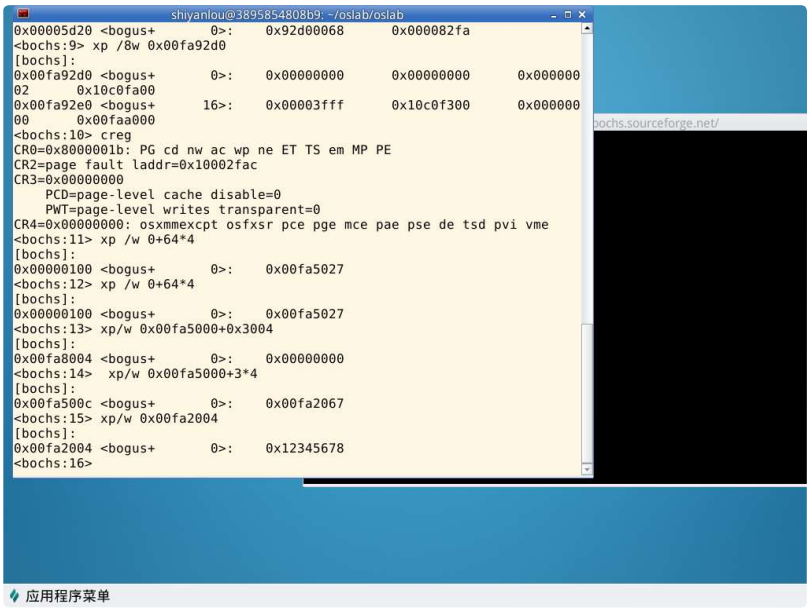
2. 实验内容

2.1. 跟踪地址翻译

通过地址ds:offset 查找目标物理地址的步骤列如下：

- 通过gdtr寄存器（存放物理地址）+ ldtr（存放选择子），来获得进程ldt的物理地址。
- 再根据段选择器（DS/CS）定位，段选择符在LDT中的位置，用以获得线性地址的基地址，其中LDT和GDT中的选择符结构一致。
- 得到的线性基地址+段内偏移量即为线性地址。
- 根据线性地址利用页表映射成物理地址，首先根据CR3寄存器获得页表项目录（page dir）基地址 .
- 根据线性地址的前10bit获取page table 在page dir中的位置。获取page table的物理地址，注意对于该物理的第三位需要ignor。
- 再根据线性地址的12-21bit来获取物理页项在页表（page table）中的位置。 获取物理页后，再根据线性地址中的低12位获取，目标数据。寻址完成。

2.2.1 实验结果



2.2.2 实验问题

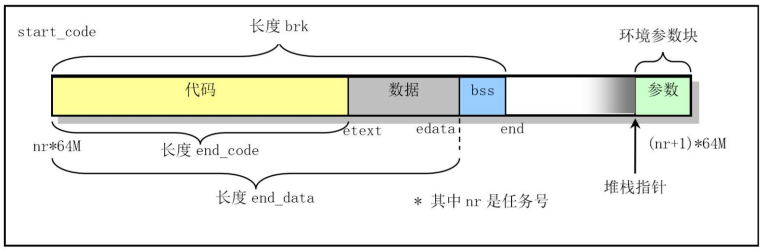
- 最重要的四步：
  i. 通过段寄存器确定查询的是LDT表或者是GDT表
  ii. 结合ldtr和ldtr寄存器来获得线性地址
  iii. 根据CR3中的目录基址获得线性地址对应的物理地址

- 在执行一次test会有的异同：
  i. 段基址可能会变化，因为段基址是根据进程次序由操作系统分配的64M空间。
  ii. 数据段内偏移量不会偏移量[逻辑地址]不会变，因为这是编译后就不变了。

2.2 增加共享内存功能，并将生产者—消费者程序移植到 Linux 0.11

2.2.1 要点

- 难点在于获得进程可用的线性地址，一个进程64M线性空间内，在低地址端存放了进程代码，数据，和bss段，高地址端存放了环境变量，用户栈;



- 根据上图，函数shmat将从bss段末处开始查找一页空闲的线性地址，即current->brk;
- 返回的地址应当是逻辑地址，所以不应是完整的线性地址，而是相对于段基址的段内偏移;
- 共享内存在put进进程的线性地址空间后，需要将mm_map中的引用递增1。

2.2.2 实验代码

- shm.c 增加2个系统调用函数，分别用于分配物理页空间，和绑定用户线性地址空间。

```
#include <errno.h>
#include <sys/types.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/kernel.h>
#define MAX_PAGES 5
#define PAGE_SIZE 4096
typedef struct {
  key_t key;
   int shmid;
  unsigned long page;
}shm_t;
shm_t shm[5] = { {0,0,0},{0,0,0},{0,0,0},{0,0,0},{0,0,0} };
int _shm_counter = 0;
int sys_shmget(key_t key, size_t size, int shmflg)
{
  int i,j;
  if (!key)
      return -1;
  for (i = 0,j= MAX_PAGES; i < MAX_PAGES; i++)
  {
      if (shm[i].key == key)
          break;
      if (shm[i].key == 0)// available
          j = i;
  }
  if (i < MAX_PAGES)
      return shm[i].shmid;
  else if (j == MAX_PAGES)
      return -1; //no idle key-page space found
  if (size > PAGE_SIZE)
      return -EINVAL;
  shm[j].page = get_free_page();
  if (!shm[j].page)
      return -ENOMEM;
  shm[j].key = key;
  shm[j].shmid = ++_shm_counter;
  return shm[j].shmid;
}
void* sys_shmat(int shmid, const void* shmaddr, int shmflg)
{
  //to more high level
  int i;
  unsigned long* page_table;
  unsigned long*  page_dir;
  unsigned long  offset,table_off;

  for (i = 0; i < MAX_PAGES; i++)
  {
      if (shm[i].shmid == shmid)
          break;
  }
  if(i == MAX_PAGES)
      return (void *)(-EINVAL);
  //find idle page table item
  /*
      page_dir = (unsigned long*)((current->start_code >> 20) & 0xffc);
  size = (0x4000000 + 0x3fffff) >> 22;
  dir_off = 0;
  table_off = 0;
  for (; size-- > 0; page_dir++)
  {
      if (!(*page_dir & 1))
          continue;
      page_table = (unsigned long*)(0xfffff000 & *page_dir);
      for (nr = 0; nr < 1024;nr++)
      {
          if (!(1 & *page_table))
          {
              put_page(shm[i].page, current->start_code+
dir_off+table_off);
              return (void*)((dir_off + table_off) & 0xfffff000);
          }
          page_table++;
          table_off += 0x1000;
      }
      dir_off += 0x400000;
  }
  */
  offset = current->brk;
  for (offset = current->brk; offset < 0x400000; offset = offset +
PAGE_SIZE)
  {
```

```
      page_dir = (unsigned long*)(((current->start_code + offset) >> 20) &
0xffc);
      if (!(*page_dir & 1))
          continue;
      page_table = (unsigned long*)(0xfffff000 & *page_dir);
      table_off = ((current->start_code + offset) >> 12) & 0x03FF;
      if (!(1 & *(page_table + table_off * 4)))
      {
          if (put_shm_page(shm[i].page, current->start_code + offset) < 0)
              return (void *)-1;
          else
              return offset;
      }
  }
  return (void*)-1;
}
```

- mm/memory 增加put_shm_page 用于将共享内存页增加到线性地址空间内

```
int  put_shm_page(unsigned long page, unsigned long address)
{
  unsigned long tmp, * page_table;
  /* NOTE !!! This uses the fact that _pg_dir=0 */
  if (page < LOW_MEM || page >= HIGH_MEMORY)
      printk("Trying to put page %p at %p\n", page, address);
  page_table = (unsigned long*)((address >> 20) & 0xffc);
  if ((*page_table) & 1)
      page_table = (unsigned long*)(0xfffff000 & *page_table);
  else {
      if (!(tmp = get_free_page()))
          return -1;
      *page_table = tmp | 7;
      page_table = (unsigned long*)tmp;
  }
  page_table[(address >> 12) & 0x3ff] = page | 7;
  /* no need for invalidate */
  page -= LOW_MEM;
  (mem_map[page >>12])++;
  return 0;
}
```

- sem.c 也做了部分改动，因为跟上次实验不同的是，消费者和生产者在不同的文件中执行，所以有俩次unlink 信号量，所以加了计数器，到0才释放 ```c /*
- linux/kernel/sem.c
- (C) 2019 Tsai Tsu Quen
- /

include <linux/sched.h>

include <linux/kernel.h>

include <linux/sem.h>

include <asm/system.h>

include <asm/segment.h>

sem_t sem_list[NR_SEM]; wait_node_t wait_node_list[NR_WAIT_NODE]; wait_node_t* _find_idle_wait_node() { int i = 0; for (i = 0; i < NR_WAIT_NODE; i++) {

```
  if (wait_node_list[i].idle == 1)
  {
      wait_node_list[i].idle = 0;
      wait_node_list[i].next_node  = NULL;
      return (wait_node_t*)(wait_node_list + i);
  }
```

} printk("no wait node\n"); return NULL; } int _str_compare(char* buf_1, char* buf_2) { int i = 0; while ((buf_1[i] == buf_2[i]) && (buf_1[i] != '\0')) {

```
    i++;
```

} if (buf_1[i] == '\0' && buf_2[i] == '\0')

```
    return 0;
```

else

```
    return -1;
```

} int _str_copy(char* buf_s, char* buf_d) { int i = 0; do {

```
    buf_d[i] = buf_s[i];
    i++;
```

} while (buf_s[i] != '\0'); return 0; } sem_t* sys_sem_open(const char* name, int value) { int i = 0; int j = NR_SEM + 1; char name_buf[20]; while ((name_buf[i] = get_fs_byte(name + i)) != '\0')

```
    i++;
```

for (i = 0; i < NR_SEM; i++) {

```
    if (*(sem_list[i].name) == '\0')
        j = i;
    else if (_str_compare(sem_list[i].name, name_buf) == 0)
    {
        sem_list[i].counter++;
        return (sem_t*)(sem_list + i);
    }
```

} if (j < NR_SEM) {

```
    _str_copy(name_buf, sem_list[j].name);
    sem_list[j].counter = 1;
    sem_list[j].value = value;
    sem_list[j].p_wait_queue = NULL;
    return (sem_t*)(sem_list + j);
```

} else

```
    return NULL;
```

} int sys_sem_unlink(const char* name) { int i = 0; char name_buf[20]; while ((name_buf[i] = get_fs_byte(name + i)) != '\0')

```
    i++;
```

for (i = 0; i < NR_SEM; i++) {

```
    if (_str_compare(sem_list[i].name, name_buf) == 0)
    {
        sem_list[i].counter--;
        if (sem_list[i].counter == 0)
        {
            (sem_list[i].name)[0] = '\0';
            sem_list[i].p_wait_queue = NULL;
        }
        return 0;
    }
```

}

return -1; } void sem_init() { int i; for (i = 0; i < NR_SEM; i++) {

```
    (sem_list[i].name)[0] = '\0';
    sem_list[i].counter = 0;
```

} for (i = 0; i < NR_WAIT_NODE; i++) {

```
    (wait_node_list[i]).idle = 1;
```

} return; } int sys_sem_wait(sem_t * sem) { wait_node_t* p, * q; if (sem == NULL)

```
    return -1;
```

cli(); sem->value--; //printk("In wait ,sem %s value is %d\n",sem->name,sem->value); if (sem->value < 0) {

```
    current->state = TASK_UNINTERRUPTIBLE;
    p = _find_idle_wait_node();
    p->p_task_struct = current;
    if (sem->p_wait_queue)
    {
        q = sem->p_wait_queue;
        while (q->next_node)
            q = q->next_node;
        q->next_node = p;
    }
    else
        sem->p_wait_queue = p;
    schedule();
```

} sti(); return 0; } int sys_sem_post(sem_t * sem) { wait_node_t* p; if (sem == NULL)

```
    return -1;
```

cli(); sem->value++; //printk("In post sem %s value is %d\n",sem->name,sem->value); if (sem->value <= 0) {

```
    /*p = sem->sem_queue[i];*/
    p = sem->p_wait_queue;
    sem->p_wait_queue = p->next_node;
    (p->p_task_struct)->state = 0;
    p->idle = 1;
    p->next_node = NULL;
```

} sti(); return 0; } ```

- producer 共享内存页作为载体。

```c
#define __LIBRARY__
#include <linux/sem.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define BUFFER_SIZE 10
#define NUMS 500
_syscall2(sem_t*, sem_open, const char*, name, int, value);
_syscall1(int, sem_wait, sem_t*, sem);
_syscall1(int, sem_post, sem_t*, sem);
_syscall1(int, sem_unlink, const char*, name);
_syscall3(void *, shmat,int ,shmid,const void *,shmaddr,int,shmflg);
_syscall3(int, shmget, key_t, key, size_t, size, int, shmflg);
int main(void)
{
  int num,counter = 0;
  int state;
  int off = 0;
  int shmid;
  int* p_shm;
  key_t shmkey = 1225;
  sem_t* Empty, * Full, * Mutex;

  if (!fork())
  {
      if ((Empty = sem_open("Empty", BUFFER_SIZE)) == NULL)
          return -1;
      if ((Full = sem_open("Full", 0)) == NULL)
          return -1;
      if ((Mutex = sem_open("Mutex", 1)) == NULL)
          return -1;

      shmid = shmget(shmkey, 4096, 0);
      if (shmid < 0)
      {
          printf("error : %d", shmid);
          return 0;
      }
      p_shm = (int*)shmat(shmid, (const void*)0, 0);
      if ((int)p_shm < 0)
      {
          printf("error: %d,", ((int)p_shm));
          return 0;
      }
      while (counter < NUMS)
      {
          sem_wait(Empty);
          sem_wait(Mutex);
          num = counter;
          p_shm[counter % BUFFER_SIZE] = num;
          counter++;
          sem_post(Mutex);
          sem_post(Full);
      }
      if (sem_unlink("Empty") < 0)
          return -1;
      if (sem_unlink("Full") < 0)
          return -1;
      if (sem_unlink("Mutex") < 0)
          return -1;
      return 0;
  }
  while (wait(&state) > 0);
  return 0;
}
```

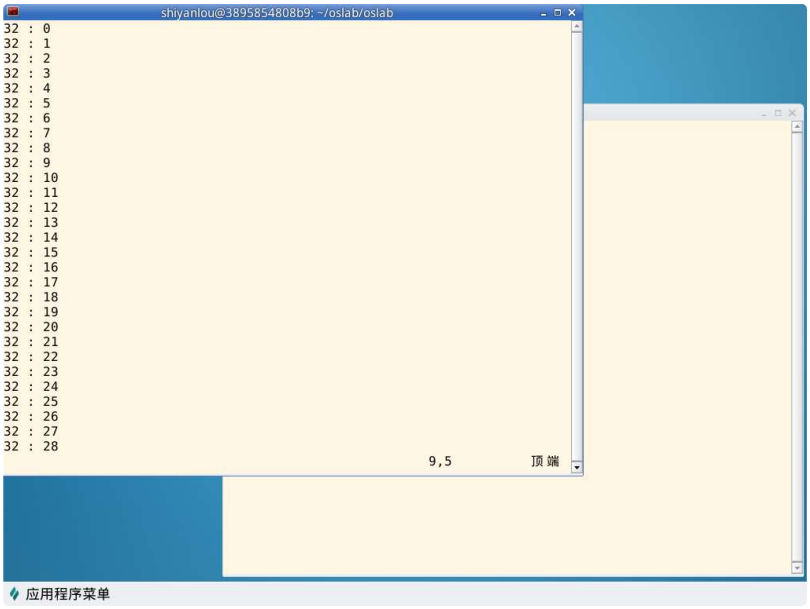
- consumer 使用共享内存

```
#define __LIBRARY__
#include <linux/sem.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define BUFFER_SIZE 10
#define NUMS 500
_syscall2(sem_t*, sem_open, const char*, name, int, value);
_syscall1(int, sem_wait, sem_t*, sem);
_syscall1(int, sem_post, sem_t*, sem);
_syscall1(int, sem_unlink, const char*, name);
_syscall3(void*, shmat, int, shmid, const void*, shmaddr, int, shmflg);
_syscall3(int, shmget, key_t, key, size_t, size, int, shmflg);
int main(void)
{
  int i;
  int state;
  int shmid;
  int* p_shm;
  key_t shmkey = 1225;
  FILE * fp;
  sem_t* Empty, * Full, * Mutex;

  if (!fork())
  {
      fp = freopen("output.txt", "w", stdout);
      if ((Empty = sem_open("Empty", BUFFER_SIZE)) == NULL)
          return -1;
      if ((Full = sem_open("Full", 0)) == NULL)
          return -1;
      if ((Mutex = sem_open("Mutex", 1)) == NULL)
          return -1;

      shmid = shmget(shmkey, 4096, 0);
      if (shmid < 0)
      {
          printf("error : %d", shmid);
          return 0;
      }
      p_shm = (int*)shmat(shmid, (const void*)0, 0);
      if ((int)p_shm < 0)
      {
          printf("error: %d,", ((int)p_shm));
          return 0;
      }
      for (i = 0; i < NUMS; i++)
      {
          sem_wait(Full);
          sem_wait(Mutex);
          printf("%d : %d\n", getpid(), p_shm[i % BUFFER_SIZE]);
          sem_post(Mutex);
          sem_post(Empty);
      }
      fclose(fp);
      if (sem_unlink("Empty") < 0)
          return -1;
      if (sem_unlink("Full") < 0)
          return -1;
      if (sem_unlink("Mutex") < 0)
          return -1;
      return 0;
  }
  while (wait(&state) > 0);
  return 0;
}
```

2.2.4 实验结果



2.2.5 遗留问题

共享页的物理内存并不会在进程结束后释放，事实上，在进程的exit中中执行free_pages(LDT(data))，但只会将mem_map对应物理地址的计数器减一，。可能造成memory leaks.

0

Markdown 语法

请输入想说的话

0 / 2000

发表评论

最新评论