XINU

VM

2020.12.9

胡浦云

2018202133

目录

实验目的......3
实验环境......3
设计思路......3
结构图......4
初始化......5
缺页中断......7
进程创建......9
进程退出......10
堆内存......10
Shell 参数......11
虚拟内存基本操作......11
测试......15
总结......19
参考文献......20

实验目的

Xinu 采用了最基础的段式内存管理机制，所有进程运行在同一个内存地址空间。然而一般的操作系统都支持通过页式管理机制来管理虚拟内存，并使 32 位架构下每个进程的逻辑地址空间达到 4GB。

本实验要求修改 Xinu，实现一个支持页式内存管理的 Xinu-vm 版本。

实验环境

实验所用系统为 Archlinux

QEMU 版本为 QEMU emulator version 5.1.0

系统为实体机 Macbook Pro 2015

设计思路

每个地址空间从 `0` 到 `_end`（对齐到 `4MiB`）为“内核地址空间”，为所有进程共享，一一映射，用于存储内核的代码、数据、内存分页功能专用的堆以及进程页目录和页表项。为了方便起见，这部分使用 `4MiB` 页进行映射，根据 `Intel` 手册，只需将 `CR4.PSE` 置一然后将 `PDE` 中的 `bit7 (PS)` 置一即可。

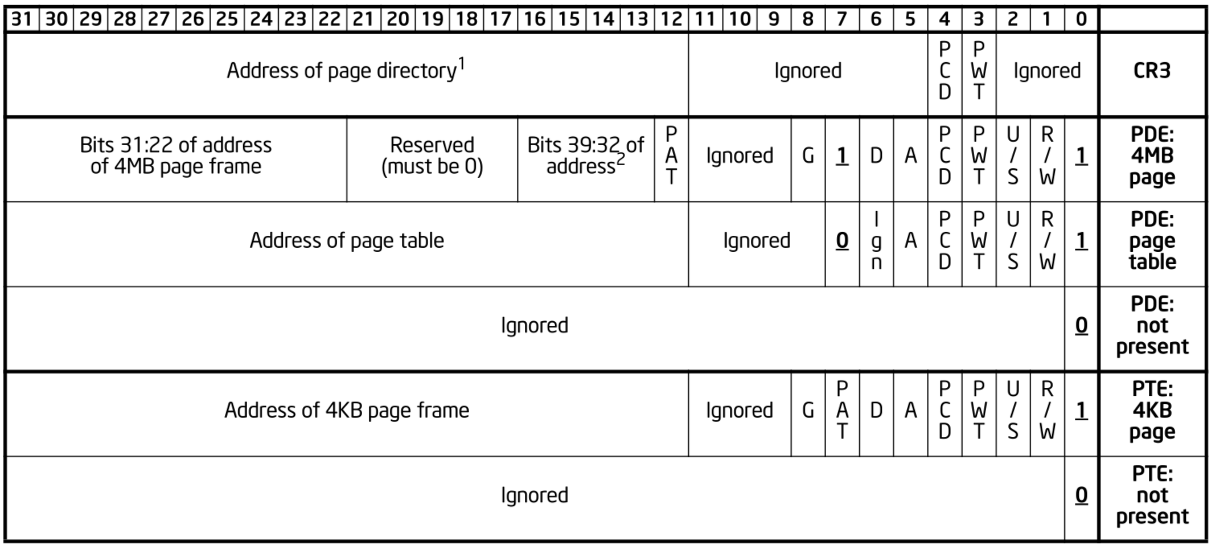


Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

NOTES:

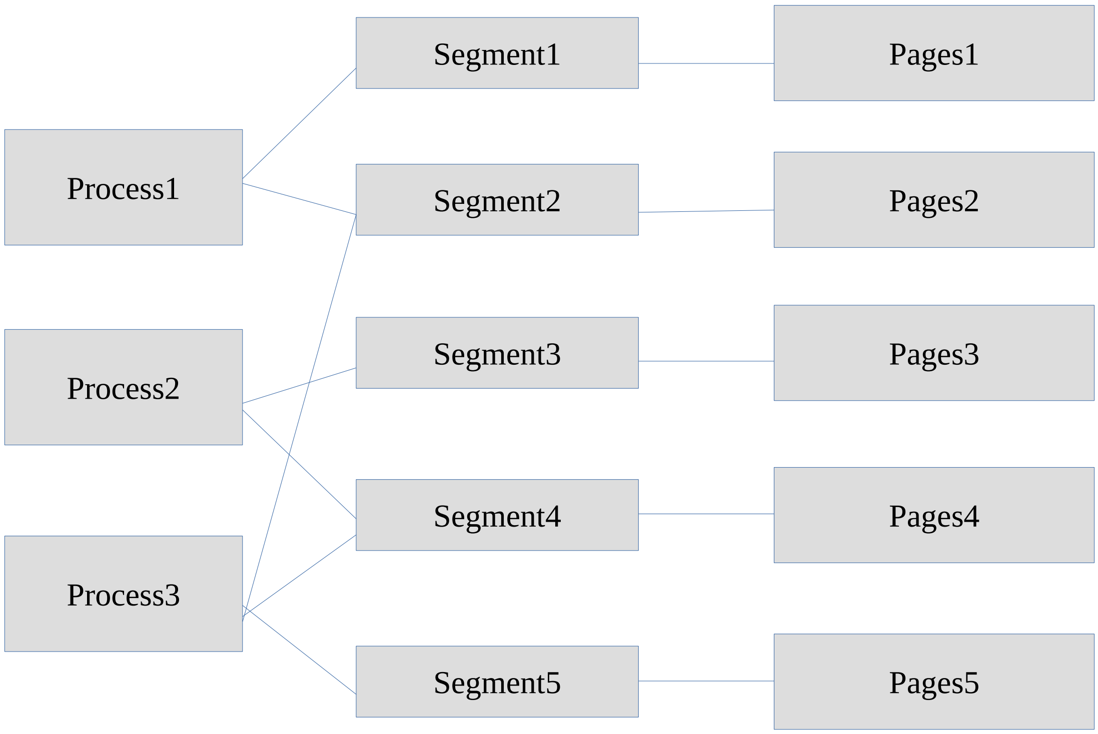
1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.

从 `ALIGN_CEIL(_end, 4MiB)` 开始是每个进程的用户地址空间。用户进程在创建 `Segment` 以后可以使用 `HeapInitialize` 将其初始化为一个堆，之后可以使用 HeapAlloc 等函数在其中分配内存，这些函数与某 `32 API` 中对应函数一致，在用户堆的基础上实现了动态内存分配。

在开启分页后，任何代码均无法直接访问非静态的物理内存。因此将页目录和页表项统一放在一个 `4MiB` 页中直接映射，从而可以运行时任意修改。

进程的栈在 `0xD0000000` 或 `0xF0000000` 处，视情况而定。
在每个进程的进程表项中添加一个字段，表示其内存映射，在进行上下文切换或缺页异常时使用。

结构图



尽管本次实验修改 xinu 源代码及编译工具链的地方非常多，以下仍尽可能展示所有内容。

初始化

由于 Xinu 在启动时，会将栈指针 esp 指向地址空间中，由 multiboot 汇报为可用内存的最高地址处，而该地址不在内核地址空间（一一映射）内，所以会在进入分页后产生错误（虚拟地址空间内该位置没有映射/被映射了其他物理页）。所以我们需要提前修改 esp，将它指向内核地址空间内。这里将其指向低 640KiB 内存。

```
	# Stack pointer set continue with boot
	movl	$0xa0000, %esp
	movl	%esp, %ebp

	/*
	 * Clear flags.
	 */
	pushl	$0
	popf
```

在 meminit 初始化完成后，初始化虚拟内存：

```
	/* Initialize free memory list */

	meminit();
	vminit();
	tssinit();
```

```
    void vminit()
    {
        // Initialize Kernel Heap

        hVMHeap = HeapInitialize(kernel_pg_pos - kernel_vm_heap_start,
kernel_vm_heap_start);

        PDirEntry_t *null_page_dir = KERNEL_PGDIR_AT(0);

        for (int i = 0; i < ENTRY_PER_PAGE; i++)
        {
            null_page_dir[i].table.present = 0;
            null_page_dir[i].table.page_size_zero = 0;
            null_page_dir[i].table.page_write_through = 0;
            null_page_dir[i].table.page_cache_disable = 0;
            null_page_dir[i].table.allow_write = 1;
            null_page_dir[i].table.allow_user_access = 0;
        }

        // Initialize Fat page
        for (uintptr now = 0; now < (uintptr)_end; now += FAT_PAGE_SIZE)
        {
            null_page_dir[PDX(now)].fat_page.address = PDX(now);
            null_page_dir[PDX(now)].fat_page.allow_user_access = 0;
            null_page_dir[PDX(now)].fat_page.allow_write = 1;
```

```
            null_page_dir[PDX(now)].fat_page.global = 1;
            null_page_dir[PDX(now)].fat_page.page_cache_disable = 0;
            null_page_dir[PDX(now)].fat_page.page_size_one = 1;
            null_page_dir[PDX(now)].fat_page.page_write_through = 0;
            null_page_dir[PDX(now)].fat_page.pat_zero = 0;
            null_page_dir[PDX(now)].fat_page.present = 1;
            null_page_dir[PDX(now)].fat_page.reserved_zero = 0;
        }

        // Initialize page manager

        PageManager_t *PManager = pageManagerFactory();

        // VM initialized
        vmLog = 1;

        // Enable Paging
        asm("mov %%eax, %%cr3\n\t"
            "mov %%cr4, %%eax\n\t"
            // Enable 4MB Page
            "or  $0x00000010, %%eax\n\t"
            "mov %%eax, %%cr4\n\t"
            // Enable Paging
            "mov %%cr0, %%eax\n\t"
            "or  $0x80000001, %%eax\n\t"
            "mov %%eax, %%cr0\n\t"
            :
            : "a"(null_page_dir)
            :);
    }
```

vminit 函数会初始化 vm 专用的内核堆，初始化页目录，将内核地址空间一一映射，然后启用分页。

```
void tssinit()
{
    // 4
    memset(&kernel_tss, sizeof(kernel_tss), 0);
    kernel_tss.link = 0;
    kernel_tss.iopb_offset = 104;

    // 5
    memset(&page_fault_tss, sizeof(page_fault_tss), 0);
    page_fault_tss.link = 0;
    page_fault_tss.eip = (uintptr)&PageFaultHandler;

    asm("movl %%cr3, %%eax\n\t"
        : "=a"(page_fault_tss.cr3)
        :
        :);

    // disable interrupt
    asm("pushfl\n\t"
        "pop %%eax\n\t"
        "andl $0xfffffdff, %%eax"
        : "=a"(page_fault_tss.eflags)
        :
        :);

    page_fault_tss.esp = page_fault_tss.esp0 = (uintptr)page_fault_stack + 4080;

    page_fault_tss.ds =
        page_fault_tss.es =
```

```
            page_fault_tss.fs =
                page_fault_tss.gs = 0x10;
    page_fault_tss.cs = 0x8;
    page_fault_tss.ss = page_fault_tss.ss0 = 0x18;

    page_fault_tss.ldtr = 0;
    page_fault_tss.iopb_offset = 104;

    asm("mov $0x20, %ax\n\t"
        "ltr %ax\n\t");

    int32 set_task_evec(uint32 xnum, uint32 handler);

    // Page fault
    set_task_evec(14, 0x28);
}
```

tssinit 会初始化内核和缺页处理的 tss，并注册缺页处理函数。
若不使用任务门直接使用陷阱门，当栈发生缺页时 CPU 无法写入返回地址，会直接导致 Triple Fault。

缺页中断

当处理器尝试访问不存在的页时会发生一个缺页中断 PF#，调用相关的中断程序。

发生缺页的地址储存在 CR2 寄存器中，Handler 会检查当前进程的段分配信息并尝试为其分配页。

```
    void PageFaultHandler()
    {
        asm("1:\n\t"
            "call PageFaultHandler_\n\t"
            "pop %eax\n\t" // unused error code
            "iret\n\t"
            "jmp 1b\n\t");
    }

    void PageFaultHandler_()
    {
        IRQMaskGuard disable_interrupt;
        uintptr addr, pteaddr;

        auto &proc = proctab[currpid];

        kernel_tss.cr3 = (uintptr)&KERNEL_PGDIR_AT(proc.procVMInfo->pgDirNo);

        asm("mov %%eax, %%cr3\n\t"
            :
            : "a"(&KERNEL_PGDIR_AT(proc.procVMInfo->pgDirNo))
            :);

        asm("mov %%cr2, %0\n\t"
            : "=r"(addr)
            :
            :);

        for (auto &segment : proc.procVMInfo->segments)
        {
            if (segment.segment &&
                addr >= (uintptr)segment.virtual_address &&
```

```
                addr < (uintptr)segment.virtual_address + segment.segment->segmentSize)
            {
                uintptr pdx = PDX(addr);
                uintptr pd_offset = pdx - PDX(segment.virtual_address);
                auto &procPG = KERNEL_PGDIR_AT(proc.procVMInfo->pgDirNo);

                if (!segment.segment->pageTables[pd_offset])
                {
                    segment.segment->pageTables[pd_offset] = PageTableAlloc();

                    if (segment.segment->pageTables[pd_offset] == SYSERR)
                    {
                        if (vmLog)
                            kprintf("Page fault: page limit exceeded\r\n");
                        panic("System halted");
                    }
                }
                procPG[pdx].table.address = (uintptr)&KERNEL_PGTABLE_AT(segment.segment->pageTables[pd_offset]) >> PTXSHIFT;
                procPG[pdx].table.present = 1;

                uintptr ptx = PTX(addr);

                auto &pgTable = KERNEL_PGTABLE_AT(segment.segment->pageTables[pd_offset]);

                if (!pgTable[ptx].present)
                {
                    pgTable[ptx].address = PageAlloc();
                    pgTable[ptx].present = 1;
                }

                if (vmLog)
                    kprintf("Page fault: %x -> %x\r\n", pgTable[ptx].address, addr);

                // Maybe not need
                invlpg(addr);

                return;
            }
        }

        if (vmLog)
            kprintf("Page fault: %x\r\n", addr);
        panic("System halted");
    }
}
```

这里使用任务门响应中断避免当栈顶缺页时中断信息无法压入导致 DF# 和 TF# 的情况。

PageFaultHandler 为 Wrapper 函数，其会调用主函数 PageFaultHandler_ 之后返回。
调用一次以后 page_fault_tss 会指向 iret 的下一条指令，此处填一个 jmp 使其可以再次调用。

进程创建

以下是我对 Xinu 进程 procent 结构的改动。

2 include/process.h

```diff
@@ -54,6 +54,8 @@ struct procent {              /* Entry in the process table          */
         umsg32  prmsg;          /* Message sent to this process         */
         bool8   prhasmsg;       /* Nonzero iff msg is valid             */
         int16   prdesc[NDESC];  /* Device descriptors for process       */
+        struct ProcessVMInfo *procVMInfo; /* Paging Info Structure */
+        HANDLE procDefaultHeap;           /* Default heap handle */
 };
 
 /* Marker for the top of a process stack (used to help detect overflow)      */
```

其中， `procVMInfo` 为进程内存映射相关信息， `procDefaultHeap` 为进程的默认堆，由进程自己初始化。

创建新进程时，需要为在新进程的地址空间内分配栈。`Xinu` 原本是在同一个地址空间内分配的栈，所以可以直接进行初始化，以构造出上下文切换后的现场。启用虚拟内存后，则需要提前创建子进程栈段，`fork` 出子进程地址空间，向其中写入 `create` 所需信息后解挂子进程栈。

构造栈现场时，为了避免地址冲突，栈有两种可能的地址。子进程的栈会创建在和父进程不同的那个地址处。

```
      uintptr stack_base_addr = (((uintptr)&pid) & 0xF0000000) ==
PROCESS_STACK_VA_1
                                                   ? PROCESS_STACK_VA_2
                                                   : PROCESS_STACK_VA_1;
      saddr = (void *)(stack_base_addr + ssize);
      ...

      /* Initilize VM info */
      createNewSegment(ssize, (void *)stack_base_addr);
      prptr->procVMInfo = processVMInfoFactory(proctab[currpid].procVMInfo);
```

进程退出

在进程退出时，需要释放进程所占用的所有内存资源 —— 即释放整个地址空间。`Xinu` 进程退出的原理，是在 `exit` 系统调用中调用了 `kill`，来结束当前进程。`kill` 会将当前进程的进程表项置为空闲，并调用调度器，重新调度，以继续执行其他进程。但在这个过程中，始终要使用到当前进程的栈空间，所以如何释放地址空间，成了需要考虑的问题

这里某些同学想到了三种解决方案:

1. 在 `exit` 系统调用开始时，切换到一个临时的栈，在 `kill` 中即可安全地释放地址空间。

2. 在调度时检测旧进程是否已被释放，如果被释放，则在上下文切换中，进行完页目录切换与栈地址切换后，立刻释放旧的页目录（通过旧的 `cr3` 寄存器值释放）。

3. 引入 `Linux` 中“僵尸进程”的概念，当一个进程被 `kill` 后，它的状态转为僵尸，资源仍然在占用中。在父进程收到子进程退出的消息后，由父进程调用特定的系统调用来释放子进程的资源。

我选择的是第二种方案，在上下文切换后回收旧进程的资源。

当 resched 函数切入一个进程时，会让其所有段引用计数加一；切出后再将其所有段引用计数减一。如此可以保证当前运行时在使用的段不被释放。

```
	if (oldpid != -1)
	{
		vmLeaveProcess(oldpid);
		if (proctab[oldpid].prstate == PR_FREE) // Suicide
		{
			freeVMInfo(proctab[oldpid].procVMInfo);
		}
	}

	oldpid = currpid;
	currpid = dequeue(readylist);
	vmEnterProcess(currpid);
```

这里释放的是在上一次 resched 切出的函数，因为 ctxsw 不一定返回（ctxsw 至新进程会返回值入口点，导致 vmEnterProcess 和 vmLeaveProcess 不配对。

堆内存

这里采用了类似某 32 API 的接口来管理堆内存，并在其上实现了一系列函数。

Shell 参数

Xinu 原本处理 Shell 参数的方法是，首先创建子进程，argv 参数的值填写为 magic number，然后将解析好的参数数据与 argv 数组写入到子进程的栈空间中，并在栈空间中寻找刚刚写入的 magic number，替换为新构造的 argv 数组地址。这个方法同样适用于开启分页后的实现，只需要创建子进程后调用 attachRemoteSegment 挂载子进程栈段即可。

```
child = create(cmdtab[j].cfunc,
	SHELL_CMDSTK, SHELL_CMDPRIO,
	cmdtab[j].cname, 2, ntok, &tmparg);

	/* If creation or argument copy fails, report error */

if ((child == SYSERR) ||
	(attachRemoteSegment(child,
				proctab[child].prstkbase - proctab[child].prstklen,
				proctab[child].prstkbase - proctab[child].prstklen)
== NULL) ||
	(addargs(child, ntok, tok, tlen, tokbuf, &tmparg) == SYSERR))
{
	fprintf(dev, SHELL_CREATMSG);
	continue;
}

detachSegment(proctab[child].prstkbase - proctab[child].prstklen);
```

虚拟内存基本操作

初始化进程页目录和页表项。

```
intptr PageTableAlloc()
{
    if (PManager->free_pgtables <= 0)
        return SYSERR;
    else
    {
        intptr ret = PManager->free_pgtable_list[--(PManager->free_pgtables)];

        auto &pageTable = KERNEL_PGTABLE_AT(ret);

        for (auto &entry : pageTable)
        {
            entry.present = 0;
            entry.allow_user_access = 0;
            entry.allow_write = 1;
            entry.global = 0;
            entry.page_cache_disable = 0;
            entry.page_write_through = 0;
        }

        if (vmLog)
            kprintf("Page table allocated: %d\r\n", ret);

        return ret;
    }
}

ProcessVMInfo *processVMInfoFactory(ProcessVMInfo *parent)
{
    ProcessVMInfo *info = new ((uintptr)hVMHeap) ProcessVMInfo;

    if (info == NULL)
        panic("Error allocting vminfo");

    info->pgDirNo = PageTableAlloc();

    memcpy(&KERNEL_PGDIR_AT(info->pgDirNo),
           &KERNEL_PGDIR_AT(parent ? parent->pgDirNo : 0),
           PAGE_SIZE);

    for (int i = 0; i < PROC_SEGMENT_COUNT; i++)
        if (parent)
        {
            info->segments[i] = parent->segments[i];
            if (info->segments[i].segment)
                segmentIncRef(info->segments[i].segment);
        }
        else
            info->segments[i].segment = NULL;

    return info;
}
```

在分配新的虚拟地址空间时，首先创建新段，将其页表项置空，然后将该段附加到当前进程上。
由于我实现了动态的共享内存，所以每个段会被多个进程所使用，也就需要维护引用计数。

```
    void *createNewSegmentTo(pid32 pid, uintptr size, void *va)
    {
        // Disable Interrupt
        IRQMaskGuard disable_interrupt;

        size = ALIGN_CEIL(size, FAT_PAGE_SIZE);
        va = (void *)ALIGN_FLOOR(va, FAT_PAGE_SIZE);
```

```
        uintptr pageNum = size / PAGE_SIZE;
        uintptr pageDirNum = ALIGN_CEIL(pageNum, ENTRY_PER_PAGE) /
ENTRY_PER_PAGE;

        auto &nowProcent = proctab[pid];
        ProcessVMMappingRecord_t *record = NULL;

        for (int i = 0; i < PROC_SEGMENT_COUNT; i++)
            if (nowProcent.procVMInfo->segments[i].segment == 0)
            {
                record = &nowProcent.procVMInfo->segments[i];
                break;
            }

        if (record == NULL)
            return NULL;

        record->virtual_address = va;
        record->segment = new (sizeof(VirtualSegment_t) + sizeof(intptr) *
pageDirNum,
                               (uintptr)hVMHeap) VirtualSegment_t;

        // Ref by this process | Ref by running process
        record->segment->refs = pid == currpid ? 2 : 1;
        record->segment->segmentSize = size;

        for (int i = 0; i < pageDirNum; i++)
            record->segment->pageTables[i] = NULL;

        return va;
    }
```

需要映射其他进程段时，首先找到空闲的段结构，然后将其指向目标段，增加引用计数（进程引用一次，当前正在运行引用一次）。

```
    void *attachRemoteSegment(pid32 rpid, void *la, void *ra)
    {
        // Disable Interrupt
        IRQMaskGuard disable_interrupt;

        la = (void *)ALIGN_CEIL(la, FAT_PAGE_SIZE);
        ra = (void *)ALIGN_CEIL(ra, FAT_PAGE_SIZE);

        auto &targetProcent = proctab[currpid];
        auto &sourceProcent = proctab[rpid];

        for (auto &seg : sourceProcent.procVMInfo->segments)
            if (seg.segment && seg.virtual_address == ra)
            {
                for (auto &newseg : targetProcent.procVMInfo->segments)
                    if (newseg.segment == NULL)
                    {
                        newseg.segment = seg.segment;

                        // Twice
                        segmentIncRef(seg.segment);
                        segmentIncRef(seg.segment);
                        newseg.virtual_address = la;

                        return la;
                    }
                return NULL;
```

```
            }

        return NULL;
    }
```

释放段时，清除当前进程对应位置所有页表项，减少段的引用计数，再刷新 TLB 使其失效。

```
    int detachSegmentFrom(pid32 pid, void *va)
    {
        // Disable Interrupt
        IRQMaskGuard disable_interrupt;

        auto &nowProcent = proctab[pid];
        va = (void *)ALIGN_FLOOR(va, FAT_PAGE_SIZE);

        for (int i = 0; i < PROC_SEGMENT_COUNT; i++)
            if (nowProcent.procVMInfo->segments[i].segment &&
                nowProcent.procVMInfo->segments[i].virtual_address == va)
            {
                auto &segment = nowProcent.procVMInfo->segments[i].segment;
                uintptr pageNum = segment->segmentSize / PAGE_SIZE;
                uintptr pageDirNum = ALIGN_CEIL(pageNum, ENTRY_PER_PAGE) /
ENTRY_PER_PAGE;

                for (int i = PDX(va); i < PDX(va) + pageDirNum; i++)
                {
                    if (auto &table = (KERNEL_PGDIR_AT(nowProcent.procVMInfo-
>pgDirNo)[i]).table; table.present)
                    {
                        if (pid == currpid)
                            for (int j = 0; j < ENTRY_PER_PAGE; j++)
                            {
                                invlpg((uintptr)va + i * FAT_PAGE_SIZE + j *
PAGE_SIZE);
                            }
                        table.present = 0;
                    }
                }

                segmentDecRef(segment);

                // current ref
                if (pid == currpid)
                    segmentDecRef(segment);

                nowProcent.procVMInfo->segments[i].segment = NULL;

                return OK;
            }

        return SYSERR;
    }
```

测试

```
Xinu for QEMU -- version #74  (ceerrep)  Wed Dec  9 12:41:20 PM CST 2020

Page table allocated: 1023
     30688 free pages of
     32736 total.
     67212 bytes of Xinu code.
           [0x00100000 to 0x0011068B]
    135160 bytes of data.
           [0x00113920 to 0x00134917]

Page table allocated: 1022
Page table allocated: 1021
Page allocated: 32735
Page fault: 7fdf -> f0010000
Page allocated: 32734
Page fault: 7fde -> f000fffc
Page fault: 7fde -> f000ffd0
Page table allocated: 1020
Page table allocated: 1019
Page allocated: 32733
Page fault: 7fdd -> d0010000
Page allocated: 32732
Page fault: 7fdc -> d000fffc
Page fault: 7fdc -> d000ffd0
Page table allocated: 1018
Page table allocated: 1017
Page allocated: 32731
Page fault: 7fdb -> f0002000
Page allocated: 32730
Page fault: 7fda -> f0001ffc
Page fault: 7fda -> f0001fcc


------------------------------------------
   __    __   _____    _   _    _    _
   \ \  / /  |__ __|  | \ | |  | |  | |
    \ \/ /     | |    |  \| |  | |  | |
    / /\ \    _| |_   | \   |  | |  | |
   / /  \ \  |     |  | | \ |  \ \__/ /
   --    --   -----    -   -    ----
------------------------------------------


Welcome to Xinu!


xsh $ Page deallocated: 32734
Page deallocated: 32735
Page table deallocated: 1021
Page table deallocated: 1022

xsh $ echo 1
Page table allocated: 1022
```

```
Page table allocated: 1021
Page allocated: 32735
Page fault: 7fdf -> d0002000
Page allocated: 32734
Page fault: 7fde -> d0001ffc
Page allocated: 32729
Page fault: 7fd9 -> d0000004
Page fault: 7fde -> d0001fc8
1
xsh $ echo 2Page deallocated: 32729
Page deallocated: 32734
Page deallocated: 32735
Page table deallocated: 1021
Page table deallocated: 1022

Page table allocated: 1022
Page table allocated: 1021
Page allocated: 32735
Page fault: 7fdf -> d0002000
Page allocated: 32734
Page fault: 7fde -> d0001ffc
Page allocated: 32729
Page fault: 7fd9 -> d0000004
Page fault: 7fde -> d0001fc8
2
xsh $ echo 3Page deallocated: 32729
Page deallocated: 32734
Page deallocated: 32735
Page table deallocated: 1021
Page table deallocated: 1022

Page table allocated: 1022
Page table allocated: 1021
Page allocated: 32735
Page fault: 7fdf -> d0002000
Page allocated: 32734
Page fault: 7fde -> d0001ffc
Page allocated: 32729
Page fault: 7fd9 -> d0000004
Page fault: 7fde -> d0001fc8
3
xsh $ exitPage deallocated: 32729
Page deallocated: 32734
Page deallocated: 32735
Page table deallocated: 1021
Page table deallocated: 1022

Shell closed
Page deallocated: 32730
Page deallocated: 32731
Page table deallocated: 1017
Page table deallocated: 1018


Main process recreating shell

Page table allocated: 1018
```

```
Page table allocated: 1017
Page allocated: 32731
Page fault: 7fdb -> f0001000
Page allocated: 32730
Page fault: 7fda -> f0000ffc
Page fault: 7fda -> f0000fcc


------------------------------------------
   __    __   _____    _   _    _    _
   \ \  / /  |__ __|  | \ | |  | |  | |
    \ \/ /     | |    |  \| |  | |  | |
    / /\ \    _| |_   | \   |  | |  | |
   / /  \ \  |     |  | |\  |   \  --  /
   --    --   -----    -   -     ----
------------------------------------------


Welcome to Xinu!


xsh $ echo 2
Page table allocated: 1022
Page table allocated: 1021
Page allocated: 32735
Page fault: 7fdf -> d0002000
Page allocated: 32734
Page fault: 7fde -> d0001ffc
Page allocated: 32729
Page fault: 7fd9 -> d0000004
Page fault: 7fde -> d0001fc8
2
xsh $ Page deallocated: 32729
Page deallocated: 32734
Page deallocated: 32735
Page table deallocated: 1021
Page table deallocated: 1022

xsh $ sleep 5 &
Page table allocated: 1022
Page table allocated: 1021
Page allocated: 32735
Page fault: 7fdf -> d0002000
Page allocated: 32734
Page fault: 7fde -> d0001ffc
Page allocated: 32729
Page fault: 7fd9 -> d0000004
Page fault: 7fde -> d0001fc8
xsh $ echo 1
Page table allocated: 1016
Page table allocated: 1015
Page allocated: 32728
Page fault: 7fd8 -> d0002000
Page allocated: 32727
Page fault: 7fd7 -> d0001ffc
Page allocated: 32726
Page fault: 7fd6 -> d0000004
```

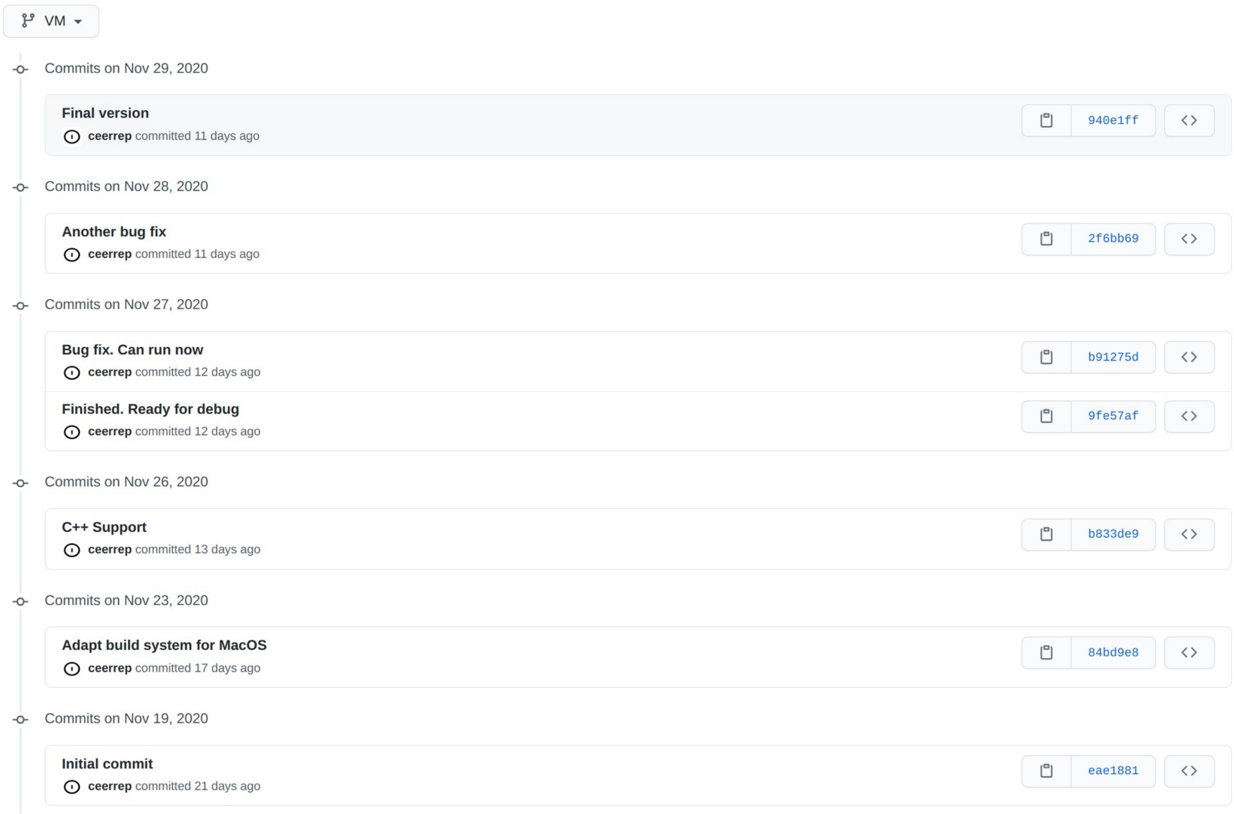
```
Page fault: 7fd7 -> d0001fc8
1
xsh $ Page deallocated: 32726
Page deallocated: 32727
Page deallocated: 32728
Page table deallocated: 1015
Page table deallocated: 1016
ps
Page table allocated: 1016
Page table allocated: 1015
Page allocated: 32728
Page fault: 7fd8 -> d0002000
Page allocated: 32727
Page fault: 7fd7 -> d0001ffc
Page allocated: 32726
Page fault: 7fd6 -> d0000004
Page fault: 7fd7 -> d0001fc8
Pid Name             State Prio Ppid Stack Base Stack Ptr  Stack Size
--- ---------------- ----- ---- ---- ---------- ---------- ----------
  0 prnull           ready    0    0 0x000A0000 0x0009FF24     8192
  2 Main process     recv    20    1 0xD0010000 0xD000FF64    65536
  7 shell            recv    20    2 0xF0001000 0xF0000C80     4096
  9 sleep            sleep   20    7 0xD0002000 0xD0001F2C     8192
 11 ps               curr    20    7 0xD0002000 0xD0001DE8     8192
xsh $ Page deallocated: 32726
Page deallocated: 32727
Page deallocated: 32728
Page table deallocated: 1015
Page table deallocated: 1016
Page deallocated: 32729
Page deallocated: 32734
Page deallocated: 32735
Page table deallocated: 1021
Page table deallocated: 1022

xsh $ sort abc dev sada cxcz wweqep ccx
Page table allocated: 1022
Page table allocated: 1021
Page allocated: 32735
Page fault: 7fdf -> d0002000
Page allocated: 32734
Page fault: 7fde -> d0001ffc
Page allocated: 32729
Page fault: 7fd9 -> d0000004
Page fault: 7fde -> d0001fc8
Page table allocated: 1016
Page allocated: 32728
Page fault: 7fd8 -> 10000000
abc
ccx
cxcz
dev
sada
wweqep
```
其中， sort 为归并排序，对堆进行测试。

总结

本次实验，总的来说，不算太难，完整实现下来大大加深了我对于 xinu 和虚拟内存的理解。但美中不足的是这样实现下来还是与实际相离较远，极重要的 Copy-On-Write 机制也没实现。个人认为这次实验难度适中，工作量也不太大，希望下一届再上这门课的时候能更贴近真实实现，比如 Copy-On-Write 的具体实现。

本次实验为了便于查看我修改的地方和原本 Xinu 的区别，我使用了 github 作为辅助工具。下图为我实现本次实验的全部过程。



参考文献

[1] Intel. 2020. *Intel® 64 And IA-32 Architectures Software Developer Manuals*. [online] Available at: <https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html> [Accessed 9 December 2020].