实验报告：虚拟内存

—— 黄浩睿 2018202143

在 Xinu 中使用页式内存管理实现虚拟内存。

规划

每个地址空间的前 32 MiB 是“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和数据。32 MiB 足够包含 Xinu 被装载到内存中后的所有函数（`.text` 段）与全局/静态变量（`.data`/`.bss` 段）。

使用一个全局变量数组来记录当前空闲的物理页地址。为了简化，这里支持的最大内存是有限的，代码中定义了最大支持 65536 个页，即 256 MiB 的物理内存。所有的页目录、页表、页均为动态分配。

从 32 MiB 开始，是每个进程的用户堆空间，堆空间的大小可以通过系统调用来改变（类似 POSIX 的 `sbrk` 函数），该函数会动态计算当前需要的页数量，分配增加的页，释放减少的页。尽管这是允许的，但用户态程序一般不需要直接操作堆空间，而是调用 `malloc`/`realloc`/`free` 函数，这些函数与标准 C 中的对应函数一致，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，也就无法直接访问任何页目录和页表。所以，这里使用了虚拟地址空间的**最后一页**，来作为**临时页**，用于访问任意物理内存页，在访问时按需映射。为了能够操作临时页，我们需要访问包含**临时页项的页表**，而该页表也仅位于物理内存中，为了能够在当前地址空间内访问，我们将该页表映射到地址空间的**倒数第二页**。

从倒数第三页开始，是进程的栈空间。为了简化，这里固定栈空间为 4 MiB（`create` 中栈空间的参数被删除）。
栈空间的前一页，作为 Shell 参数页，用于 Shell 向子进程传递 `argv` 参数列表。

在每个进程的进程表项中添加一个字段，表示其页目录的物理地址，在进行上下文切换时使用。

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，而该地址不在前 32 MiB 的内核地址空间（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 32 MiB 内核地址空间内。这里修改了 start.S，定义了一个全局数组作为初始化栈，并在调用 `nulluser` 前将其赋值给 `esp`。

```
        .data
#define INITIALIZE_STACK_SIZE 4194304 /* 4M initialization stack */
stack:  .space  INITIALIZE_STACK_SIZE
```

```
        /*
         * Call the nulluser with initialize stack to initialize the system
         */
        leal    stack, %esp
        addl    $INITIALIZE_STACK_SIZE, %esp
        call    nulluser
        call    halt
```

`meminit` 函数在读取 multiboot 提供的内存信息后，会将可用的内存区域添加到 `memlist` 中。由于要使用页式内存管理，我们不再需要 `memlist`，而是由 `meminit` 函数初始化可用的物理内存页列表。

在 `meminit` 初始化完成后，初始化虚拟内存：

```
    /* Initialize virtual memory */

    meminit();
    PageDirectory initializationPageDirectoryPhysicalAddress = initializeVirtualMemory();
```

```
PageDirectory initializeVirtualMemory() {
    initializeKernelPageTables();
    PageDirectory initializationPageDirectoryPhysicalAddress = allocateVirtualMemorySpace();
    switchPageDirectory(initializationPageDirectoryPhysicalAddress);

    // Enable paging
    asm volatile (
        "mov %%cr0, %%eax\n\t"
        "or $0x80000000, %%eax\n\t"
        "mov %%eax, %%cr0\n\t"
        :
        :
        :"eax"
    );

    return initializationPageDirectoryPhysicalAddress;
}
```

该函数首先初始化内核页表（一一映射的 32 MiB 共享内存空间），然后切换到新的页表，并修改 `cr0` 开启分页，最后将页目录返回。该页目录将被存放到空进程的进程表项中。

进程创建

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，由于新进程的栈空间不在同一个地址空间内，所以需要先临时构造出栈现场，然后调用虚拟内存模块提供的接口，将构造好的数据写入到子进程的栈空间中。

构造栈现场时，需要将栈地址本身写入到栈空间中，以模拟 `push` 指令。由于临时空间与真实的栈地址的基址不同，这里通过同时维护两个指针来计算真实栈空间下的偏移。

```c
	static uint32 initialContentOfStackLastPage[VM_PAGE_SIZE / sizeof(uint32)];
	uint32 *stackBufferAddress = initialContentOfStackLastPage + VM_PAGE_SIZE / sizeof(uint32);
	uint32 *stackVirtualAddress = VM_STACK_VIRTUAL_ADDRESS_HIGH;
	stackBufferAddress--;
	stackVirtualAddress--;

	*stackBufferAddress = STACKMAGIC;
	savsp = (uint32)stackVirtualAddress;

	/* Push arguments */
	a = (uint32 *)(&nargs + 1); /* Start of args		*/
	a += nargs -1;			/* Last argument		*/
	for ( ; nargs > 0 ; nargs--) {	/* Machine dependent; copy args */
		--stackVirtualAddress;
		*--stackBufferAddress = *a--;	/* onto created process's stack */
	}
	--stackVirtualAddress;
	*--stackBufferAddress = (long)INITRET;	/* Push on return address	*/

	/* The following entries on the stack must match what doContextSwitch	*/
	/*   expects a saved process state to contain: ret address, */
	/*   ebp, interrupt mask, flags, registers, and an old SP	*/

	--stackVirtualAddress;
	*--stackBufferAddress = (long)funcaddr; /* Make the stack look like it's*/
					/*   half-way through a call to */
					/*   doContextSwitch that "returns" to the*/
					/*   new process		*/
	--stackVirtualAddress;
	*--stackBufferAddress = savsp;		/* This will be register ebp	*/
					/*   for process exit		*/
	savsp = (uint32) stackVirtualAddress;		/* Start of frame for doContextSwitch	*/
	--stackVirtualAddress;
	*--stackBufferAddress = 0x00000200;	/* New process runs with	*/
					/*   interrupts enabled	*/

	/* Basically, the following emulates an x86 "pushal" instruction*/

	--stackVirtualAddress;
	*--stackBufferAddress = 0;		/* %eax */
	--stackVirtualAddress;
	*--stackBufferAddress = 0;		/* %ecx */
	--stackVirtualAddress;
	*--stackBufferAddress = 0;		/* %edx */
	--stackVirtualAddress;
	*--stackBufferAddress = 0;		/* %ebx */
	--stackVirtualAddress;
	*--stackBufferAddress = 0;		/* %esp; value filled in below	*/
	uint32 *pushSpBufferAddress = stackBufferAddress;		/* Remember this location	*/
	--stackVirtualAddress;
	*--stackBufferAddress = savsp;		/* %ebp (while finishing doContextSwitch)	*/
	--stackVirtualAddress;
	*--stackBufferAddress = 0;		/* %esi */
	--stackVirtualAddress;
	*--stackBufferAddress = 0;		/* %edi */
	*stackBufferAddress = (unsigned long) (process->stackCurrent = (char *)stackVirtualAddress);

	// Initialize the stack
	allocateVirtualMemoryPages(process->pageDirectoryPhysicalAddress, VM_STACK_VIRTUAL_PAGE_ID_BEGIN, VM_STACK_PAGES_PER_PROCESS);
	writeToAnotherVirtualMemorySpacePage(process->pageDirectoryPhysicalAddress, VM_STACK_VIRTUAL_PAGE_ID_END - 1, initialContentOfStackLastPage);
```

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。

这里我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，在上下文切换后立刻回收旧进程的资源。

这里为上下文切换的函数引入了第四个参数，表示是否需要释放旧的页目录。

```
/*------------------------------------------------------------------------
 * doContextSwitch -  X86 context switch;
 * the call is doContextSwitch(&old_sp, &new_sp, &new_page_directory, deallocateOldVirtualMemorySpace)
 *------------------------------------------------------------------------
 */
doContextSwitch:
        pushl   %ebp        /* Push ebp onto stack      */
        movl    %esp, %ebp  /* Record current SP in ebp */
        pushfl          /* Push flags onto the stack    */
        pushal          /* Push general regs. on stack  */

        /* Save old segment registers here, if multiple allowed */

        movl    8(%ebp), %eax   /* Get mem location in which to */
                    /*   save the old process's SP  */
        movl    %esp, (%eax)    /* Save old process's SP    */
        movl    12(%ebp), %ebx  /* Get location from which to   */
                    /*   restore new process's SP   */
        movl    16(%ebp), %eax
        movl    %cr3, %ecx  /* Save old page directory */
        movl    20(%ebp), %edx /* Save whether to deallocate the old virtual memory space */
        movl    %eax, %cr3  /* Switch to new page directory */

        /* The next instruction switches from the old process's */
        /*   stack to the new process's stack.          */

        movl    (%ebx), %esp    /* Pop up new process's SP  */

        /* Check whether the old virtual memory space needs to be deallocated */
        testl   $1, %edx        /* whether to deallocate the old virtual memory space is in edx */
        jz      .skipDeallocate
        push    %ecx            /* Old virtual memory space is in %ecx */
        call    deallocateVirtualMemorySpace
        addl    $4, %esp

.skipDeallocate:
        /* Restore new seg. registers here, if multiple allowed */

        popal           /* Restore general registers    */
        movl    4(%esp), %ebp   /* Pick up ebp before restoring */
                    /*   interrupts         */
        popfl           /* Restore interrupt mask   */
        add $4, %esp        /* Skip saved value of ebp  */
        ret         /* Return to new process    */
```

堆内存

这里采用了类似 POSIX 的 `sbrk` 函数的接口来管理堆内存，并在其上实现了 `malloc` 系列函数。

system > heap.c > ...

```c
#include <heap.h>

void changeHeapSize(int32 delta) {
    struct ProcessEntry *currentProcess = &processTable[currentProcessID];
    if (delta < 0 && (uint32)-delta > currentProcess->heapSize)
        delta = -(int32)currentProcess->heapSize;

    uint32 newHeapSize = currentProcess->heapSize + delta;
    uint32 newEndPageId = VM_HEAP_START_PAGE_ID + getRoundedUpPageCountOfMemorySize(newHeapSize);
    uint32 oldEndPageId = VM_HEAP_START_PAGE_ID + getRoundedUpPageCountOfMemorySize(currentProcess->heapSize);

    if (newEndPageId > oldEndPageId)
        allocateVirtualMemoryPages(currentProcess->pageDirectoryPhysicalAddress, oldEndPageId, newEndPageId - oldEndPageId);
    else if (newEndPageId < oldEndPageId)
        deallocateVirtualMemoryPages(currentProcess->pageDirectoryPhysicalAddress, newEndPageId, oldEndPageId - newEndPageId);

    currentProcess->heapSize = newHeapSize;
}
```

实现 `malloc` 时，由于全局空间均为所有进程共享，所以必须在堆内存上存储全部信息。

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。这个方法同样不适用于开启分页后的实现。

这里使用了单独的一页来存放参数（为了简便，假设 Shell 参数不会太多/太长），与栈初始化相同 —— 在当前地址空间内构造好 `argv` 数组，以及各个 `argv[i]` 的偏移地址，在创建子进程后映射参数页，并通过虚拟内存模块的接口将数据写入。

```c
        child = create(cmdtab[j].cfunc,
            SHELL_CMDPRIO, SHELL_CMD_TIME_SLICE,
            cmdtab[j].cname, 2, ntok, VM_SHELL_ARGUMENT_PAGE_ADDRESS);

        /* Pass arguments to child process */

        static char argumentsPageContent[VM_PAGE_SIZE];
        void **argumentsBufferAddress = (void **)argumentsPageContent;
        void **argumentsVirtualAddress = (void **)VM_SHELL_ARGUMENT_PAGE_ADDRESS;

        char *argumentDataBufferAddress = (char *)(argumentsBufferAddress + ntok);
        char *argumentDataVirtualAddress = (char *)(argumentsVirtualAddress + ntok);
        memcpy(argumentDataBufferAddress, tokbuf, tlen);

        for (uint32 i = 0; i < ntok; i++)
            argumentsBufferAddress[i] = &argumentDataVirtualAddress[tok[i]];

        allocateVirtualMemoryPages(processTable[child].pageDirectoryPhysicalAddress, VM_SHELL_ARGUMENT_PAGE_ID, 1);
        writeToAnotherVirtualMemorySpacePage(processTable[child].pageDirectoryPhysicalAddress, VM_SHELL_ARGUMENT_PAGE_ID, argumentsPageContent);
```

虚拟内存基本操作

初始化页目录项与页表项:

```c
PageDirectoryEntry *initializePageDirectoryEntry(PageDirectoryEntry *entry, bool8 present, uint32 physicalPageIdForPageTable) {
    entry->present                 = present;
    entry->readWrite               = 1;
    entry->userSupervisor          = 1;
    entry->writeThrough            = 1;
    entry->disableCache            = 0;
    entry->accessed                = 0;
    entry->dirty                   = 0;
    entry->global                  = 0;
    entry->zero1                   = 0;
    entry->zero2                   = 0;
    entry->physicalPageIdForPageTable = physicalPageIdForPageTable;
    return entry;
}

PageTableEntry *initializePageTableEntry(PageTableEntry *entry, bool8 present, uint32 physicalPageIdForPage) {
    entry->present               = present;
    entry->readWrite             = 1;
    entry->userSupervisor        = 1;
    entry->writeThrough          = 1;
    entry->disableCache          = 0;
    entry->accessed              = 0;
    entry->zero1                 = 0;
    entry->isLargePage           = 0;
    entry->zero2                 = 0;
    entry->zero3                 = 0;
    entry->physicalPageIdForPage = physicalPageIdForPage;
    return entry;
}
```

初始化内核页表（32 MiB 的一一映射地址）。由于当前未开启分页，所以可以直接访问页表项。

```c
PageDirectoryEntry pageDirectoryEntriesForKernelPageTables[VM_KERNEL_PAGE_TABLE_COUNT];

void initializeKernelPageTables() {
    uint32 currentPhysicalPageId = 0;
    for (uint32 i = 0; i < VM_KERNEL_PAGE_TABLE_COUNT; i++) {
        PageTable pageTable = allocatePhysicalPage();
        initializePageDirectoryEntry(&pageDirectoryEntriesForKernelPageTables[i], TRUE, pageAddressToPageId(pageTable));
        for (uint32 j = 0; j < VM_PAGES_PER_PAGE_TABLE; j++) {
            initializePageTableEntry(&pageTable[j], TRUE, currentPhysicalPageId++);
        }
    }
}
```

为了在开启分页后能够访问任意物理内存页，这里定义了一些辅助函数和宏，判断分页是否已开启，返回原地址或映射后的临时页地址：

```
uint32 saveCurrentVirtualMemorySpaceTemporaryPageMappedPhysicalPageId() {
    PageTable pageTable = VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ADDRESS;
    PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(VM_TEMPORARY_PAGE_ID)];
    return pageTableEntry->physicalPageIdForPage;
}

void mapToCurrentVirtualMemorySpaceTemporaryPage(uint32 physicalPageId) {
    PageTable pageTable = VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ADDRESS;
    PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(VM_TEMPORARY_PAGE_ID)];
    initializePageTableEntry(pageTableEntry, physicalPageId != 0, physicalPageId);
    flushTlbForSinglePage(VM_TEMPORARY_PAGE_ADDRESS);
}

#define SAVE_TEMPORARY_PAGE \
    uint32 temporaryPageBackup = !getPagingEnabled() ? 0 : saveCurrentVirtualMemorySpaceTemporaryPageMappedPhysicalPageId();
#define ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(Type, accessAddress, physicalAddress) \
    Type accessAddress; \
    if (getPagingEnabled()) { \
        mapToCurrentVirtualMemorySpaceTemporaryPage(pageAddressToPageId(physicalAddress)); \
        accessAddress = VM_TEMPORARY_PAGE_ADDRESS; \
    } else \
        accessAddress = physicalAddress;
#define RESTORE_TEMPORARY_PAGE \
    if (temporaryPageBackup) \
        mapToCurrentVirtualMemorySpaceTemporaryPage(temporaryPageBackup); \

PageDirectoryEntry pageDirectoryEntriesForKernelPageTables[VM_KERNEL_PAGE_TABLE_COUNT];

void initializeKernelPageTables() {
    uint32 currentPhysicalPageId = 0;
    for (uint32 i = 0; i < VM_KERNEL_PAGE_TABLE_COUNT; i++) {
        PageTable pageTable = allocatePhysicalPage();
        initializePageDirectoryEntry(&pageDirectoryEntriesForKernelPageTables[i], TRUE, pageAddressToPageId(pageTable));
        for (uint32 j = 0; j < VM_PAGES_PER_PAGE_TABLE; j++) {
            initializePageTableEntry(&pageTable[j], TRUE, currentPhysicalPageId++);
        }
    }
}
```

在分配新的虚拟地址空间时，首先写入共享页表项，将其余页表项置空，然后为该地址空间的临时页**的页表**分配页，并将该页表映射到该地址空间内。

```
PageDirectory allocateVirtualMemorySpace() {
    PageDirectory pageDirectoryPhysicalAddress = allocatePhysicalPage();

    SAVE_TEMPORARY_PAGE
    ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)

    memset(pageDirectory, 0, VM_PAGE_SIZE);

    uint32 i = 0;
    for (; i < VM_KERNEL_PAGE_TABLE_COUNT; i++)
        pageDirectory[i] = pageDirectoryEntriesForKernelPageTables[i];

    for (; i < VM_PAGE_TABLES_PER_DIRECTORY; i++)
        initializePageDirectoryEntry(&pageDirectory[i], FALSE, 0);

    // Initialize the page table for the temporary page
    PageTable newPageTableForTemporaryPagePhysicalAddress = allocateEmptyPageTable(&pageDirectory[pageIdToPageDirectoryEntryIndex(VM_TEMPORARY_PAGE_ID)]);
    // Map the page table for temporary page to the virtual memory space
    ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, newPageTableForTemporaryPage, newPageTableForTemporaryPagePhysicalAddress)
    initializePageTableEntry(&newPageTableForTemporaryPage[pageIdToPageTableEntryIndex(VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ID)], TRUE, pageAddressToPageId(newPageTableForTemporaryPagePhysicalAddress));

    RESTORE_TEMPORARY_PAGE
    return pageDirectoryPhysicalAddress;
}
```

释放虚拟地址空间时，需要注意，不要将临时页释放（因为这个页不一定属于这个进程，也有可能是这个进程或其他进程的页表/页目录），也不要在释放页时将临时页的页目录释放（否则会 double free）。

由于我没有实现动态的共享内存，所以每个动态页只被一个进程所使用，也就不需要维护引用计数。

```
void deallocateVirtualMemorySpace(PageDirectory pageDirectoryPhysicalAddress) {
    kprintf("deallocateVirtualMemorySpace: deallocating vm directory %x from process [%s]\n", pageDirectoryPhysicalAddress, processTable[currentProcessID].processName);

    SAVE_TEMPORARY_PAGE

    for (uint32 i = VM_KERNEL_PAGE_TABLE_COUNT; i < VM_PAGE_TABLES_PER_DIRECTORY; i++) {
        ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
        PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[i];
        if (!pageDirectoryEntry->present) continue;

        PageTable pageTablePhysicalAddress = pageIdToPageAddress(pageDirectoryEntry->physicalPageIdForPageTable);
        ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, pageTable, pageTablePhysicalAddress);
        for (uint32 j = 0; j < VM_PAGES_PER_PAGE_TABLE; j++) {
            PageTableEntry *pageTableEntry = &pageTable[j];
            if (!pageTableEntry->present) continue;

            // Don't touch the temporary page since it's owned by another virtual memory space
            if (
                i == pageIdToPageDirectoryEntryIndex(VM_TEMPORARY_PAGE_ID) &&
                j == pageIdToPageTableEntryIndex(VM_TEMPORARY_PAGE_ID)
            )
                continue;
            // Don't touch the page table for temporary page since it will be deallocated when deallocating page tables
            if (
                i == pageIdToPageDirectoryEntryIndex(VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ID) &&
                j == pageIdToPageTableEntryIndex(VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ID)
            )
                continue;

            deallocatePhysicalPage(pageIdToPageAddress(pageTableEntry->physicalPageIdForPage));
        }

        deallocatePhysicalPage(pageTablePhysicalAddress);
    }

    deallocatePhysicalPage(pageDirectoryPhysicalAddress);

    RESTORE_TEMPORARY_PAGE
}
```

需要向另一地址空间写入数据时，首先映射临时页来读取页表和页目录，然后将目标页映射到临时页上，进行写操作。

```
void writeToAnotherVirtualMemorySpacePage(PageDirectory pageDirectoryPhysicalAddress, uint32 virtualPageId, void *src) {
    SAVE_TEMPORARY_PAGE

    ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)

    PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[pageIdToPageDirectoryEntryIndex(virtualPageId)];
    if (!pageDirectoryEntry->present) {
        kprintf("writeToAnotherVirtualMemorySpacePage: page directory entry not present\n");
        return;
    }
    PageTable pageTablePhysicalAddress = pageIdToPageAddress(pageDirectoryEntry->physicalPageIdForPageTable);

    ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, pageTable, pageTablePhysicalAddress)

    PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(virtualPageId)];
    if (!pageTableEntry->present) {
        kprintf("writeToAnotherVirtualMemorySpacePage: page table entry not present\n");
        return;
    }

    writeToPhysicalMemoryPage(pageTableEntry->physicalPageIdForPage, src);

    RESTORE_TEMPORARY_PAGE
}
```

申请新的虚拟内存页时，首先检查其页表是否存在，如果不存在则创建新的空页表。另外全程需要通过临时页表来访问该地址空间的页目录/页表。

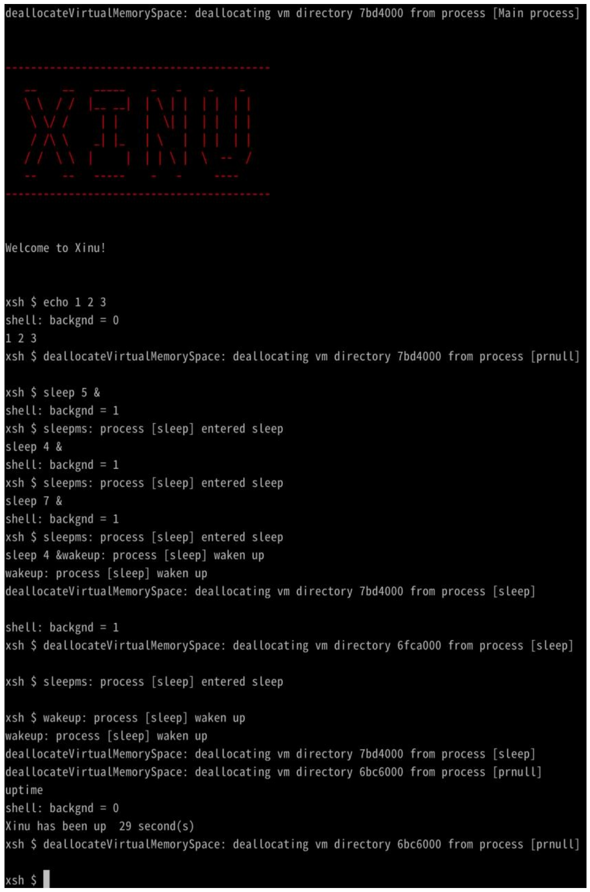
```c
void allocateVirtualMemoryPage(PageDirectory pageDirectoryPhysicalAddress, uint32 virtualPageId, void *initialPageContent) {
    SAVE_TEMPORARY_PAGE

    ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)

    PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[pageIdToPageDirectoryEntryIndex(virtualPageId)];
    PageTable pageTablePhysicalAddress;
    if (!pageDirectoryEntry->present)
        pageTablePhysicalAddress = allocateEmptyPageTable(pageDirectoryEntry);
    else
        pageTablePhysicalAddress = pageIdToPageAddress(pageDirectoryEntry->physicalPageIdForPageTable);

    ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, pageTable, pageTablePhysicalAddress)

    PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(virtualPageId)];
    if (pageTableEntry->present) {
        kprintf("allocateVirtualMemoryPage: page %u already mapped\n", virtualPageId);
        RESTORE_TEMPORARY_PAGE
        return;
    }

    void *page = allocatePhysicalPage();
    uint32 pageId = pageAddressToPageId(page);
    initializePageTableEntry(pageTableEntry, TRUE, pageId);
    writeToPhysicalMemoryPage(pageId, initialPageContent);

    if (pageDirectoryPhysicalAddress == getCurrentPageDirectory())
        flushTlbForSinglePage(pageIdToPageAddress(virtualPageId));

    RESTORE_TEMPORARY_PAGE
}
```

释放虚拟内存页时，可以判断当前页表的页是否全部被释放，以回收页表。当然为了效率起见，也可不进行这个判断，将所有页表本身的释放留到进程结束后进行。

```c
void deallocateVirtualMemoryPage(PageDirectory pageDirectoryPhysicalAddress, uint32 virtualPageId) {
    SAVE_TEMPORARY_PAGE

    ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
    PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[pageIdToPageDirectoryEntryIndex(virtualPageId)];
    if (!pageDirectoryEntry->present) {
        kprintf("deallocateVirtualMemoryPage: page %u not mapped (page directory entry not present)\n", virtualPageId);
        RESTORE_TEMPORARY_PAGE
        return;
    }
    PageTable pageTablePhysicalAddress = pageIdToPageAddress(pageDirectoryEntry->physicalPageIdForPageTable);

    ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, pageTable, pageTablePhysicalAddress)
    PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(virtualPageId)];
    if (!pageTableEntry->present) {
        kprintf("deallocateVirtualMemoryPage: page %u not mapped (page table entry not present)\n", virtualPageId);
        RESTORE_TEMPORARY_PAGE
        return;
    }

    uint32 pageId = pageTableEntry->physicalPageIdForPage;
    deallocatePhysicalPage(pageIdToPageAddress(pageId));
    initializePageTableEntry(pageTableEntry, FALSE, 0);

    uint32 presentPageTableEntryCount = 0;
    for (uint32 i = 0; i < VM_PAGES_PER_PAGE_TABLE; i++)
        if (pageTableEntry[i].present)
            presentPageTableEntryCount++;

    if (presentPageTableEntryCount == 0) {
        // Deallocate the page table
        ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
        PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[pageIdToPageDirectoryEntryIndex(virtualPageId)];
        uint32 pageId = pageDirectoryEntry->physicalPageIdForPageTable;
        deallocatePhysicalPage(pageIdToPageAddress(pageId));
        initializePageDirectoryEntry(pageDirectoryEntry, FALSE, 0);
    }

    if (pageDirectoryPhysicalAddress == getCurrentPageDirectory())
        flushTlbForSinglePage(pageIdToPageAddress(virtualPageId));

    RESTORE_TEMPORARY_PAGE
}
```

测试

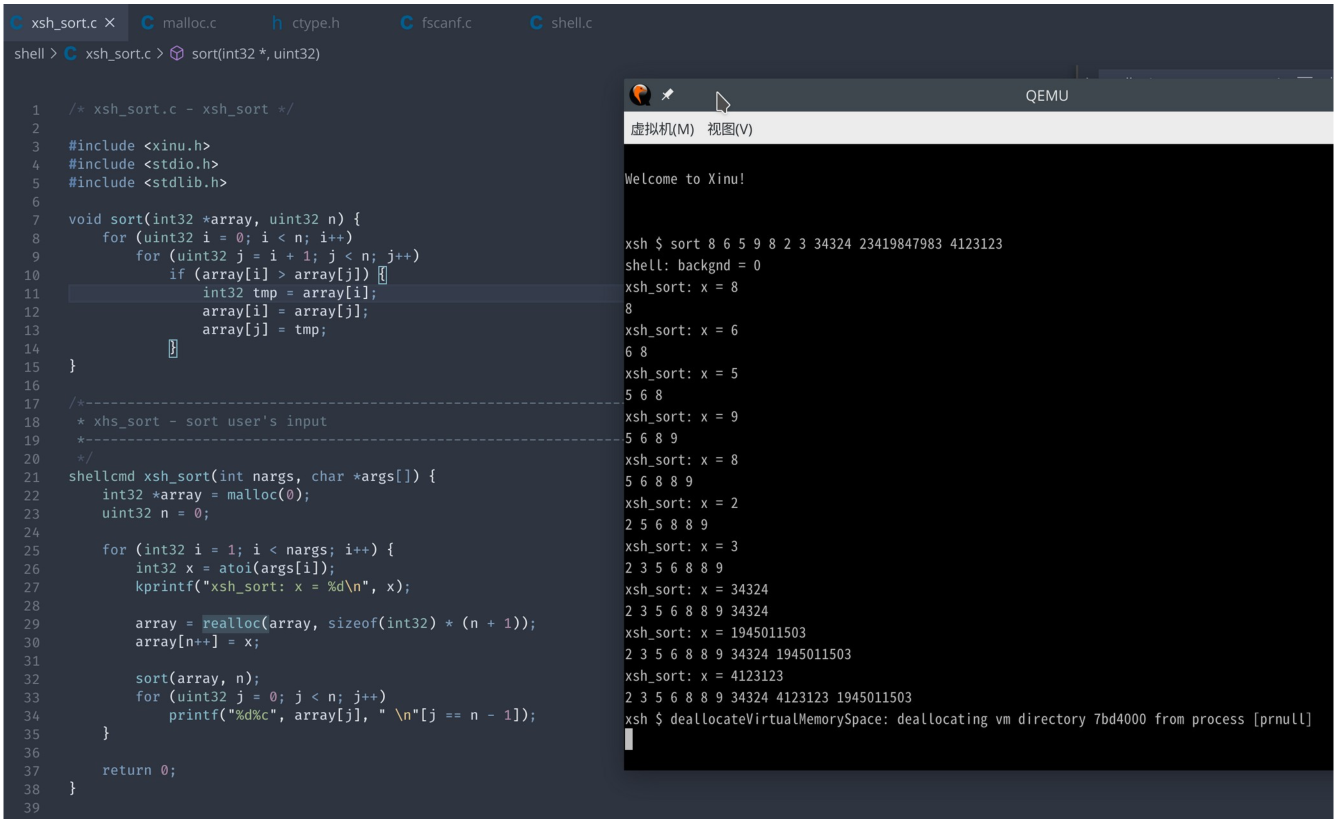
首先测试基本的进程管理及栈是否可用，通过多次执行 `echo` 和 `sleep` 命令：



```
deallocateVirtualMemorySpace: deallocating vm directory 7bd4000 from process [Main process]

Welcome to Xinu!

xsh $ echo 1 2 3
shell: backgnd = 0
1 2 3
xsh $ deallocateVirtualMemorySpace: deallocating vm directory 7bd4000 from process [prnull]

xsh $ sleep 5 &
shell: backgnd = 1
xsh $ sleepms: process [sleep] entered sleep
sleep 4 &
shell: backgnd = 1
xsh $ sleepms: process [sleep] entered sleep
sleep 7 &
shell: backgnd = 1
xsh $ sleepms: process [sleep] entered sleep
sleep 4 &wakeup: process [sleep] waken up
wakeup: process [sleep] waken up
deallocateVirtualMemorySpace: deallocating vm directory 7bd4000 from process [sleep]

shell: backgnd = 1
xsh $ deallocateVirtualMemorySpace: deallocating vm directory 6fca000 from process [sleep]

xsh $ sleepms: process [sleep] entered sleep

xsh $ wakeup: process [sleep] waken up
wakeup: process [sleep] waken up
deallocateVirtualMemorySpace: deallocating vm directory 7bd4000 from process [sleep]
deallocateVirtualMemorySpace: deallocating vm directory 6bc6000 from process [prnull]
uptime
shell: backgnd = 0
Xinu has been up  29 second(s)
xsh $ deallocateVirtualMemorySpace: deallocating vm directory 6bc6000 from process [prnull]

xsh $
```

多个后台执行的 `sleep` 进程都成功进行了睡眠，`echo` 也可以成功输出信息。通过释放虚拟地址空间时输出的调试信息可知，并行运行的进程的页目录地址不同。而依次执行的多个命令的页目录地址相同，表明内存回收工作正常。

然后通过一个小程序来测试堆空间分配：



该程序会将用户输入的参数解析为整数，存放到通过 `malloc` 和 `realloc` 动态分配的内存空间中，并对其进行排序。测试结果表明堆内存工作正常。