

# 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`。

```
57
58     .data
59     #define INITIALIZE_STACK_SIZE 4194304 /* 4M initialization stack */
60     stack: .space INITIALIZE_STACK_SIZE

217     /*
218     * Call the nulluser with initialize stack to initialize the system
219     */
220     leal    stack, %esp
221     addl    $INITIALIZE_STACK_SIZE, %esp
222     call    nulluser
223     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：

```
130  /* Initialize virtual memory */
131
132  meminit();
133  PageDirectory initializationPageDirectoryPhysicalAddress = initializeVirtualMemory();

219  PageDirectory initializeVirtualMemory() {
220      initializeKernelPageTables();
221      PageDirectory initializationPageDirectoryPhysicalAddress = allocateVirtualMemorySpace();
222      switchPageDirectory(initializationPageDirectoryPhysicalAddress);
223
224      // Enable paging
225      asm volatile (
226          "mov %%cr0, %%eax\n\t"
227          "or $0x80000000, %%eax\n\t"
228          "mov %%eax, %%cr0\n\t"
229          :
230          :
231          : "eax"
232      );
233
234      return initializationPageDirectoryPhysicalAddress;
235  }
```

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。

```
61 static uint32 initialContentOfStackLastPage[VM_PAGE_SIZE / sizeof(uint32)];
62 uint32 *stackBufferAddress = initialContentOfStackLastPage + VM_PAGE_SIZE / sizeof(uint32);
63 uint32 *stackVirtualAddress = VM_STACK_VIRTUAL_ADDRESS_HIGH;
64 stackBufferAddress--;
65 stackVirtualAddress--;
66
67 *stackBufferAddress = STACKMAGIC;
68 savsp = (uint32)stackVirtualAddress;
69
70 /* Push arguments */
71 a = (uint32 *)(&nargs + 1); /* Start of args */
72 a += nargs - 1; /* Last argument */
73 for ( ; nargs > 0 ; nargs--) { /* Machine dependent; copy args */
74     --stackVirtualAddress;
75     *--stackBufferAddress = *a--; /* onto created process's stack */
76 }
77 --stackVirtualAddress;
78 *--stackBufferAddress = (long)INITRET; /* Push on return address */
79
80 /* The following entries on the stack must match what doContextSwitch */
81 /* expects a saved process state to contain: ret address, */
82 /* ebp, interrupt mask, flags, registers, and an old SP */
83
84 --stackVirtualAddress;
85 *--stackBufferAddress = (long)funcaddr; /* Make the stack look like it's */
86 /* half-way through a call to */
87 /* doContextSwitch that "returns" to the */
88 /* new process */
89 --stackVirtualAddress;
90 *--stackBufferAddress = savsp; /* This will be register ebp */
91 /* for process exit */
92 savsp = (uint32) stackVirtualAddress; /* Start of frame for doContextSwitch */
93 --stackVirtualAddress;
94 *--stackBufferAddress = 0x00000200; /* New process runs with */
95 /* interrupts enabled */
96
97 /* Basically, the following emulates an x86 "pushal" instruction */
98
99 --stackVirtualAddress;
100 *--stackBufferAddress = 0; /* %eax */
101 --stackVirtualAddress;
102 *--stackBufferAddress = 0; /* %ecx */
103 --stackVirtualAddress;
104 *--stackBufferAddress = 0; /* %edx */
105 --stackVirtualAddress;
106 *--stackBufferAddress = 0; /* %ebx */
107 --stackVirtualAddress;
108 *--stackBufferAddress = 0; /* %esp; value filled in below */
109 uint32 *pushSpBufferAddress = stackBufferAddress; /* Remember this location */
110 --stackVirtualAddress;
111 *--stackBufferAddress = savsp; /* %ebp (while finishing doContextSwitch) */
112 --stackVirtualAddress;
113 *--stackBufferAddress = 0; /* %esi */
114 --stackVirtualAddress;
115 *--stackBufferAddress = 0; /* %edi */
116 *stackBufferAddress = (unsigned long) (process->stackCurrent = (char *)stackVirtualAddress);
117
118 // Initialize the stack
119 allocateVirtualMemoryPages(process->pageDirectoryPhysicalAddress, VM_STACK_VIRTUAL_PAGE_ID_BEGIN, VM_STACK_PAGES_PER_PROCESS);
120 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。

```
6  /*-----
7  * doContextSwitch - X86 context switch;
8  * the call is doContextSwitch(&old_sp, &new_sp, &new_page_directory, deallocateOldVirtualMemorySpace)
9  *-----
10 /*
11 doContextSwitch:
12     pushl    %ebp        /* Push ebp onto stack */
13     movl    %esp, %ebp   /* Record current SP in ebp */
14     pushfl   /* Push flags onto the stack */
15     pushal   /* Push general regs. on stack */
16
17     /* Save old segment registers here, if multiple allowed */
18
19     movl    8(%ebp), %eax /* Get mem location in which to */
20     /* save the old process's SP */
21     movl    %esp, (%eax) /* Save old process's SP */
22     movl    12(%ebp), %ebx /* Get location from which to */
23     /* restore new process's SP */
24     movl    16(%ebp), %eax
25     movl    %cr3, %ecx /* Save old page directory */
26     movl    20(%ebp), %edx /* Save whether to deallocate the old virtual memory space */
27     movl    %eax, %cr3 /* Switch to new page directory */
28
29     /* The next instruction switches from the old process's */
30     /* stack to the new process's stack. */
31
32     movl    (%ebx), %esp /* Pop up new process's SP */
33
34     /* Check whether the old virtual memory space needs to be deallocated */
35     testl   $1, %edx /* whether to deallocate the old virtual memory space is in edx */
36     jz      .skipDeallocate
37     push    %ecx /* Old virtual memory space is in %ecx */
38     call    deallocateVirtualMemorySpace
39     addl    $4, %esp
40
41 .skipDeallocate:
42     /* Restore new seg. registers here, if multiple allowed */
43
44     popal   /* Restore general registers */
45     movl    4(%esp), %ebp /* Pick up ebp before restoring */
46     /* interrupts */
47     popfl   /* Restore interrupt mask */
48     add $4, %esp /* Skip saved value of ebp */
49     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 > C heap.c > ...
1  #include <heap.h>
2
3  void changeHeapSize(int32 delta) {
4      struct ProcessEntry *currentProcess = &processTable[currentProcessID];
5      if (delta < 0 && (uint32)-delta > currentProcess->heapSize)
6          delta = -(int32)currentProcess->heapSize;
7
8      uint32 newHeapSize = currentProcess->heapSize + delta;
9      uint32 newEndPageId = VM_HEAP_START_PAGE_ID + getRoundedUpPageCountOfMemorySize(newHeapSize);
10     uint32 oldEndPageId = VM_HEAP_START_PAGE_ID + getRoundedUpPageCountOfMemorySize(currentProcess->heapSize);
11
12     if (newEndPageId > oldEndPageId)
13         allocateVirtualMemoryPages(currentProcess->pageDirectoryPhysicalAddress, oldEndPageId, newEndPageId - oldEndPageId);
14     else if (newEndPageId < oldEndPageId)
15         deallocateVirtualMemoryPages(currentProcess->pageDirectoryPhysicalAddress, newEndPageId, oldEndPageId - newEndPageId);
16
17     currentProcess->heapSize = newHeapSize;
18 }
```

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。

```
276     child = create(cmdtab[j].cfunc,
277                   SHELL_CMDPRIO, SHELL_CMD_TIME_SLICE,
278                   cmdtab[j].cname, 2, ntok, VM_SHELL_ARGUMENT_PAGE_ADDRESS);
279
280     /* Pass arguments to child process */
281
282     static char argumentsPageContent[VM_PAGE_SIZE];
283     void **argumentsBufferAddress = (void **)argumentsPageContent;
284     void **argumentsVirtualAddress = (void **)VM_SHELL_ARGUMENT_PAGE_ADDRESS;
285
286     char *argumentDataBufferAddress = (char *) (argumentsBufferAddress + ntok);
287     char *argumentDataVirtualAddress = (char *) (argumentsVirtualAddress + ntok);
288     memcpy(argumentDataBufferAddress, tokbuf, tlen);
289
290     for (uint32 i = 0; i < ntok; i++)
291         argumentsBufferAddress[i] = &argumentDataVirtualAddress[tok[i]];
292
293     allocateVirtualMemoryPages(processTable[child].pageDirectoryPhysicalAddress, VM_SHELL_ARGUMENT_PAGE_ID, 1);
294     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：

```
29  PageDirectoryEntry *initializePageDirectoryEntry(PageDirectoryEntry *entry, bool8 present, uint32 physicalPageIdForPageTable) {
30      entry->present = present;
31      entry->readWrite = 1;
32      entry->userSupervisor = 1;
33      entry->writeThrough = 1;
34      entry->disableCache = 0;
35      entry->accessed = 0;
36      entry->dirty = 0;
37      entry->global = 0;
38      entry->zero1 = 0;
39      entry->zero2 = 0;
40      entry->physicalPageIdForPageTable = physicalPageIdForPageTable;
41      return entry;
42  }
43
44  PageTableEntry *initializePageTableEntry(PageTableEntry *entry, bool8 present, uint32 physicalPageIdForPage) {
45      entry->present = present;
46      entry->readWrite = 1;
47      entry->userSupervisor = 1;
48      entry->writeThrough = 1;
49      entry->disableCache = 0;
50      entry->accessed = 0;
51      entry->zero1 = 0;
52      entry->isLargePage = 0;
53      entry->zero2 = 0;
54      entry->zero3 = 0;
55      entry->physicalPageIdForPage = physicalPageIdForPage;
56      return entry;
57  }
```

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。

```
125  PageDirectoryEntry pageDirectoryEntriesForKernelPageTables[VM_KERNEL_PAGE_TABLE_COUNT];
126
127  void initializeKernelPageTables() {
128      uint32 currentPhysicalPageId = 0;
129      for (uint32 i = 0; i < VM_KERNEL_PAGE_TABLE_COUNT; i++) {
130          PageTable pageTable = allocatePhysicalPage();
131          initializePageDirectoryEntry(&pageDirectoryEntriesForKernelPageTables[i], TRUE, pageAddressToPageId(pageTable));
132          for (uint32 j = 0; j < VM_PAGES_PER_PAGE_TABLE; j++) {
133              initializePageTableEntry(&pageTable[j], TRUE, currentPhysicalPageId++);
134          }
135      }
136  }
```

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，返回原地或映射后的临时页地址：

```
99  uint32 saveCurrentVirtualMemorySpaceTemporaryPageMappedPhysicalPageId() {
100      PageTable pageTable = VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ADDRESS;
101      PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(VM_TEMPORARY_PAGE_ID)];
102      return pageTableEntry->physicalPageIdForPage;
103  }
104
105  void mapToCurrentVirtualMemorySpaceTemporaryPage(uint32 physicalPageId) {
106      PageTable pageTable = VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ADDRESS;
107      PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(VM_TEMPORARY_PAGE_ID)];
108      initializePageTableEntry(pageTableEntry, physicalPageId != 0, physicalPageId);
109      flushTlbForSinglePage(VM_TEMPORARY_PAGE_ADDRESS);
110  }
111
112  #define SAVE_TEMPORARY_PAGE \
113      uint32 temporaryPageBackup = !getPagingEnabled() ? 0 : saveCurrentVirtualMemorySpaceTemporaryPageMappedPhysicalPageId();
114  #define ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(Type, accessAddress, physicalAddress) \
115      Type accessAddress; \
116      if (getPagingEnabled()) { \
117          mapToCurrentVirtualMemorySpaceTemporaryPage(pageAddressToPageId(physicalAddress)); \
118          accessAddress = VM_TEMPORARY_PAGE_ADDRESS; \
119      } else \
120          accessAddress = physicalAddress;
121  #define RESTORE_TEMPORARY_PAGE \
122      if (temporaryPageBackup) \
123          mapToCurrentVirtualMemorySpaceTemporaryPage(temporaryPageBackup); \
124
125  PageDirectoryEntry pageDirectoryEntriesForKernelPageTables[VM_KERNEL_PAGE_TABLE_COUNT];
126
127  void initializeKernelPageTables() {
128      uint32 currentPhysicalPageId = 0;
129      for (uint32 i = 0; i < VM_KERNEL_PAGE_TABLE_COUNT; i++) {
130          PageTable pageTable = allocatePhysicalPage();
131          initializePageDirectoryEntry(&pageDirectoryEntriesForKernelPageTables[i], TRUE, pageAddressToPageId(pageTable));
132          for (uint32 j = 0; j < VM_PAGES_PER_PAGE_TABLE; j++) {
133              initializePageTableEntry(&pageTable[j], TRUE, currentPhysicalPageId++);
134          }
135      }
136  }
```

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。

```
154  PageDirectory allocateVirtualMemorySpace() {
155      PageDirectory pageDirectoryPhysicalAddress = allocatePhysicalPage();
156
157      SAVE_TEMPORARY_PAGE
158      ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
159
160      memset(pageDirectory, 0, VM_PAGE_SIZE);
161
162      uint32 i = 0;
163      for (; i < VM_KERNEL_PAGE_TABLE_COUNT; i++)
164          pageDirectory[i] = pageDirectoryEntriesForKernelPageTables[i];
165
166      for (; i < VM_PAGE_TABLES_PER_DIRECTORY; i++)
167          initializePageDirectoryEntry(&pageDirectory[i], FALSE, 0);
168
169      // Initialize the page table for the temporary page
170      PageTable newPageTableForTemporaryPagePhysicalAddress = allocateEmptyPageTable(&pageDirectory[pagingEnabled() ? VM_TEMPORARY_PAGE_ID : 0]);
171      // Map the page table for temporary page to the virtual memory space
172      ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, newPageTableForTemporaryPage, newPageTableForTemporaryPagePhysicalAddress)
173      initializePageTableEntry(&newPageTableForTemporaryPage[pagingEnabled() ? VM_TEMPORARY_PAGE_ID : 0], TRUE, pageAddressToPageId(newPageTableForTemporaryPagePhysicalAddress));
174
175      RESTORE_TEMPORARY_PAGE
176      return pageDirectoryPhysicalAddress;
177  }
```

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），也不要再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。

```
179 void deallocateVirtualMemorySpace(PageDirectory pageDirectoryPhysicalAddress) {
180     kprintf("deallocateVirtualMemorySpace: deallocating vm directory %x from process [%s]\n", pageDirectoryPhysicalAddress, processTable[currentProcessID].processName);
181
182     SAVE_TEMPORARY_PAGE
183
184     for (uint32 i = VM_KERNEL_PAGE_TABLE_COUNT; i < VM_PAGE_TABLES_PER_DIRECTORY; i++) {
185         ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
186         PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[i];
187         if (!pageDirectoryEntry->present) continue;
188
189         PageTable pageTablePhysicalAddress = pageIdToPageAddress(pageDirectoryEntry->physicalPageIdForPageTable);
190         ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, pageTable, pageTablePhysicalAddress);
191         for (uint32 j = 0; j < VM_PAGES_PER_PAGE_TABLE; j++) {
192             PageTableEntry *pageTableEntry = &pageTable[j];
193             if (!pageTableEntry->present) continue;
194
195             // Don't touch the temporary page since it's owned by another virtual memory space
196             if (
197                 i == pageIdToPageDirectoryEntryIndex(VM_TEMPORARY_PAGE_ID) &&
198                 j == pageIdToPageTableEntryIndex(VM_TEMPORARY_PAGE_ID)
199             )
200                 continue;
201             // Don't touch the page table for temporary page since it will be deallocated when deallocating page tables
202             if (
203                 i == pageIdToPageDirectoryEntryIndex(VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ID) &&
204                 j == pageIdToPageTableEntryIndex(VM_PAGE_TABLE_FOR_TEMPORARY_PAGE_PAGE_ID)
205             )
206                 continue;
207
208             deallocatePhysicalPage(pageIdToPageAddress(pageTableEntry->physicalPageIdForPage));
209         }
210         deallocatePhysicalPage(pageTablePhysicalAddress);
211     }
212     deallocatePhysicalPage(pageDirectoryPhysicalAddress);
213
214     RESTORE_TEMPORARY_PAGE
215 }
216
217 }
```

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。

```
247 void writeToAnotherVirtualMemorySpacePage(PageDirectory pageDirectoryPhysicalAddress, uint32 virtualPageId, void *src) {
248     SAVE_TEMPORARY_PAGE
249
250     ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
251
252     PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[pageIdToPageDirectoryEntryIndex(virtualPageId)];
253     if (!pageDirectoryEntry->present) {
254         kprintf("writeToAnotherVirtualMemorySpacePage: page directory entry not present\n");
255         return;
256     }
257     PageTable pageTablePhysicalAddress = pageIdToPageAddress(pageDirectoryEntry->physicalPageIdForPageTable);
258
259     ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, pageTable, pageTablePhysicalAddress)
260
261     PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(virtualPageId)];
262     if (!pageTableEntry->present) {
263         kprintf("writeToAnotherVirtualMemorySpacePage: page table entry not present\n");
264         return;
265     }
266
267     writeToPhysicalMemoryPage(pageTableEntry->physicalPageIdForPage, src);
268
269     RESTORE_TEMPORARY_PAGE
270 }
```



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。

```
272 void allocateVirtualMemoryPage(PageDirectory pageDirectoryPhysicalAddress, uint32 virtualPageId, void *initialPageContent) {
273     SAVE_TEMPORARY_PAGE
274
275     ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
276
277     PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[pageIdToPageDirectoryEntryIndex(virtualPageId)];
278     PageTable pageTablePhysicalAddress;
279     if (!pageDirectoryEntry->present)
280         pageTablePhysicalAddress = allocateEmptyPageTable(pageDirectoryEntry);
281     else
282         pageTablePhysicalAddress = pageIdToPageAddress(pageDirectoryEntry->physicalPageIdForPageTable);
283
284     ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, pageTable, pageTablePhysicalAddress)
285
286     PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(virtualPageId)];
287     if (pageTableEntry->present) {
288         kprintf("allocateVirtualMemoryPage: page %u already mapped\n", virtualPageId);
289         RESTORE_TEMPORARY_PAGE
290         return;
291     }
292
293     void *page = allocatePhysicalPage();
294     uint32 pageId = pageAddressToPageId(page);
295     initializePageTableEntry(pageTableEntry, TRUE, pageId);
296     writeToPhysicalMemoryPage(pageId, initialPageContent);
297
298     if (pageDirectoryPhysicalAddress == getCurrentPageDirectory())
299         flushTlbForSinglePage(pageIdToPageAddress(virtualPageId));
300
301     RESTORE_TEMPORARY_PAGE
302 }
```

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。

```
304 void deallocateVirtualMemoryPage(PageDirectory pageDirectoryPhysicalAddress, uint32 virtualPageId) {
305     SAVE_TEMPORARY_PAGE
306
307     ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
308     PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[pageIdToPageDirectoryEntryIndex(virtualPageId)];
309     if (!pageDirectoryEntry->present) {
310         kprintf("deallocateVirtualMemoryPage: page %u not mapped (page directory entry not present)\n", virtualPageId);
311         RESTORE_TEMPORARY_PAGE
312         return;
313     }
314     PageTable pageTablePhysicalAddress = pageIdToPageAddress(pageDirectoryEntry->physicalPageIdForPageTable);
315
316     ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageTable, pageTable, pageTablePhysicalAddress)
317     PageTableEntry *pageTableEntry = &pageTable[pageIdToPageTableEntryIndex(virtualPageId)];
318     if (!pageTableEntry->present) {
319         kprintf("deallocateVirtualMemoryPage: page %u not mapped (page table entry not present)\n", virtualPageId);
320         RESTORE_TEMPORARY_PAGE
321         return;
322     }
323
324     uint32 pageId = pageTableEntry->physicalPageIdForPage;
325     deallocatePhysicalPage(pageIdToPageAddress(pageId));
326     initializePageTableEntry(pageTableEntry, FALSE, 0);
327
328     uint32 presentPageTableEntryCount = 0;
329     for (uint32 i = 0; i < VM_PAGES_PER_PAGE_TABLE; i++)
330         if (pageTableEntry[i].present)
331             presentPageTableEntryCount++;
332
333     if (presentPageTableEntryCount == 0) {
334         // Deallocate the page table
335         ACCESS_PHYSICAL_PAGE_WITH_TEMPORARY_PAGE(PageDirectory, pageDirectory, pageDirectoryPhysicalAddress)
336         PageDirectoryEntry *pageDirectoryEntry = &pageDirectory[pageIdToPageDirectoryEntryIndex(virtualPageId)];
337         uint32 pageId = pageDirectoryEntry->physicalPageIdForPageTable;
338         deallocatePhysicalPage(pageIdToPageAddress(pageId));
339         initializePageDirectoryEntry(pageDirectoryEntry, FALSE, 0);
340     }
341
342     if (pageDirectoryPhysicalAddress == getCurrentPageDirectory())
343         flushTlbForSinglePage(pageIdToPageAddress(virtualPageId));
344
345     RESTORE_TEMPORARY_PAGE
346 }
```

## 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]

-----
XINU
-----

Welcome to Xinu!

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

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]

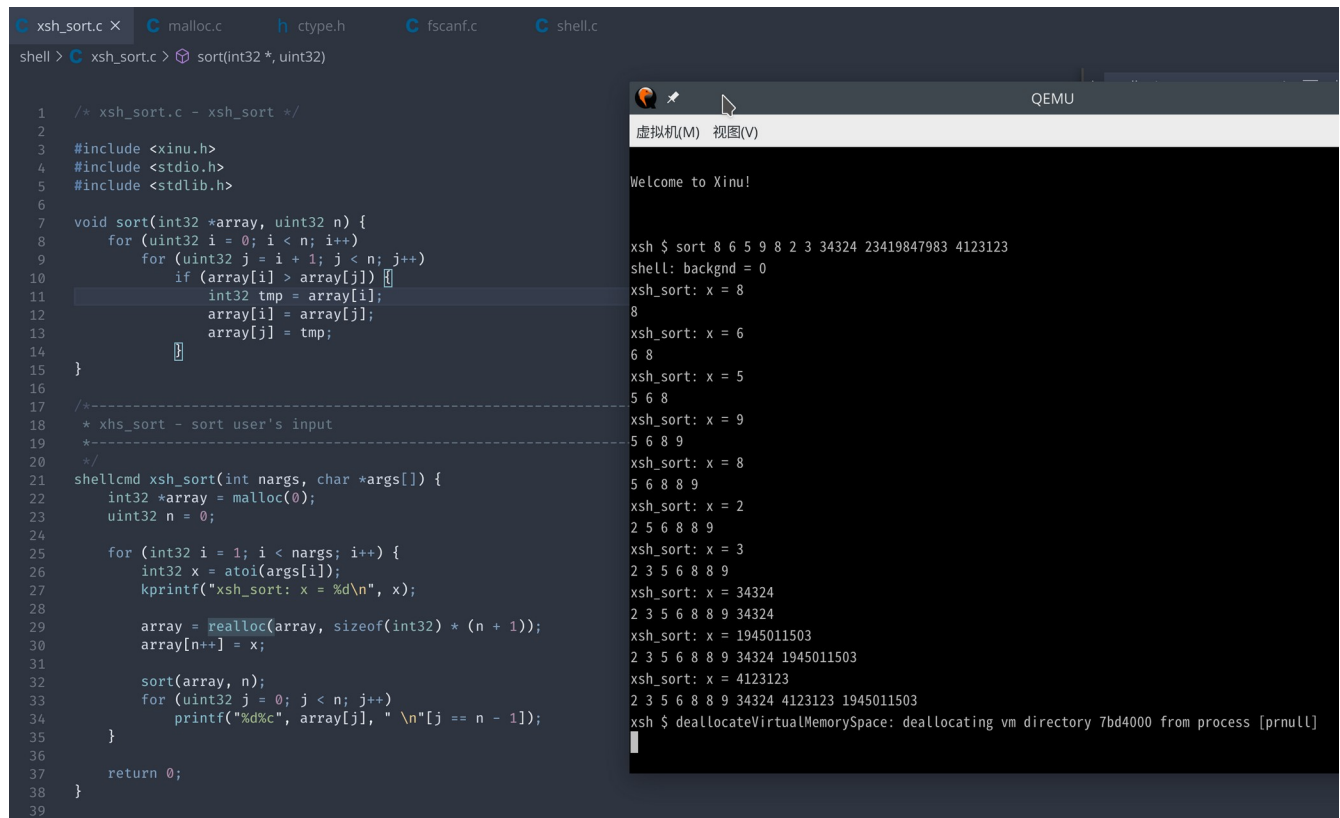
xsh $ sleepms: process [sleep] entered sleep

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 [prnu]
uptime
shell: backgnd = 0
Xinu has been up 29 second(s)
xsh $ deallocateVirtualMemorySpace: deallocating vm directory 6bc6000 from process [prnu]

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：



The screenshot displays a QEMU virtual machine environment. On the left, a terminal window shows the source code of a C program named `xsh_sort.c`. The program includes `<xinu.h>`, `<stdio.h>`, and `<stdlib.h>`. It defines a `sort` function that implements a bubble sort algorithm on an array of `uint32` integers. The `main` function, wrapped in a `shellcmd` macro, parses command-line arguments into integers, prints them, sorts them using the `sort` function, and prints the sorted array. The program uses `malloc` and `realloc` for dynamic memory allocation.

On the right, the QEMU window shows the execution output. It starts with a "Welcome to Xinu!" message. The user enters the command `sort 8 6 5 9 8 2 3 34324 23419847983 4123123`. The program prints the input numbers, sorts them, and displays the sorted output: `2 3 5 6 8 8 9 34324 4123123 1945011503`. The final line shows the program deallocating virtual memory space.

```
1  /* xsh_sort.c - xsh_sort */
2
3  #include <xinu.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  void sort(uint32 *array, uint32 n) {
8      for (uint32 i = 0; i < n; i++)
9          for (uint32 j = i + 1; j < n; j++)
10             if (array[i] > array[j]) {
11                 uint32 tmp = array[i];
12                 array[i] = array[j];
13                 array[j] = tmp;
14             }
15 }
16
17 /*-----
18 * xsh_sort - sort user's input
19 *-----
20 */
21 shellcmd xsh_sort(int nargs, char *args[]) {
22     int32 *array = malloc(0);
23     uint32 n = 0;
24
25     for (int32 i = 1; i < nargs; i++) {
26         int32 x = atoi(args[i]);
27         kprintf("xsh_sort: x = %d\n", x);
28
29         array = realloc(array, sizeof(int32) * (n + 1));
30         array[n++] = x;
31
32         sort(array, n);
33         for (uint32 j = 0; j < n; j++)
34             printf("%d%c", array[j], " \n"[j == n - 1]);
35     }
36
37     return 0;
38 }
39
```

```
QEMU
虚拟机(M) 视图(V)

Welcome to Xinu!

xsh $ sort 8 6 5 9 8 2 3 34324 23419847983 4123123
shell: backgnd = 0
xsh_sort: x = 8
8
xsh_sort: x = 6
6 8
xsh_sort: x = 5
5 6 8
xsh_sort: x = 9
5 6 8 9
xsh_sort: x = 8
5 6 8 8 9
xsh_sort: x = 2
2 5 6 8 8 9
xsh_sort: x = 3
2 3 5 6 8 8 9
xsh_sort: x = 34324
2 3 5 6 8 8 9 34324
xsh_sort: x = 1945011503
2 3 5 6 8 8 9 34324 1945011503
xsh_sort: x = 4123123
2 3 5 6 8 8 9 34324 4123123 1945011503
xsh $ deallocateVirtualMemorySpace: deallocating vm directory 7bd4000 from process [prnull]
```

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。