



实验课程：操作系统

实验名称：物理内存与虚拟内存管理

专业名称：计算机科学与技术

学生姓名：吴臻

学生学号：21307371

实验地点：实验中心大楼D栋501

实验成绩：

报告时间：2023/5/14

- 1. 实验要求
- 2. 预备知识与实验环境
- 3. 实验任务
- 4. 实验步骤/关键代码/实验结果

- **Assignment 1** 物理页内存管理的实现
 - 子任务1
 - 子任务2
- **Assignment 2** 二级分页机制的实现
 - 子任务1
 - 子任务二
- **Assignment 3** 虚拟页内存管理的实现
 - 子任务1（和任务二的要求类似，不再重复）
 - 子任务2
 - 子任务3
- **Assignment 4** 页面置换算法的实现
- 5. 总结(对实验过程中遇到的问题进行总结，可以提出对实验设置的改进意见)
- 6. 参考资料清单

1. 实验要求

在本次实验中，我们首先学习如何使用位图和地址池来管理资源。然后，我们将实现在物理地址空间下的内存管理。接着，我们将会学习并开启二级分页机制。在开启分页机制后，我们将实现在虚拟地址空间下的内存管理。

2. 预备知识与实验环境

3. 实验任务

1. 物理页内存管理的实现
2. 二级分页机制的实现
3. 虚拟页内存管理的实现
4. 页面置换算法的实现

4. 实验步骤/关键代码/实验结果

Assignment 1 物理页内存管理的实现

具体要求如下：

1. 结合代码分析位图、地址池、物理页管理的初始化过程，以及物理页进行分配和释放的实现思路。
2. 构造测试用例来分析物理页内存管理的实现是否存在bug。如果存在，则尝试修复并再次测试。否则，结合测试用例简要分析物理页内存管理的实现的正确性。

子任务1

位图的初始化：

```
void BitMap::initialize(char *bitmap, const int length)
{
    this->bitmap = bitmap;
    this->length = length;

    int bytes = ceil(length, 8);
    memset(bitmap, 0, bytes);
}
```

bitmap是位图的首地址，**length**是被管理的资源个数，位图的初始化是将位图占据的内存空间全部置零。由于位图以比特（bit）为单位存储，而不是字节（byte），因此需要将长度 **length** 转换为字节数。具体实现中，使用了 **ceil** 函数将 **length** 除以8后向上取整，保证了能够存储所有位的字节数。

地址池的初始化：

```
// 设置地址池BitMap
void AddressPool::initialize(char *bitmap, const int length, const int
startAddress)
{
    resources.initialize(bitmap, length);
    this->startAddress = startAddress;
}
```

地址池是基于位图的，初始化时增加一个页首地址的初始化即可

物理页管理的初始化：

```
void MemoryManager::initialize()
{
```

```

this->totalMemory = 0;
this->totalMemory = getTotalMemory();

// 预留的内存
int usedMemory = 256 * PAGE_SIZE + 0x100000;
if(this->totalMemory < usedMemory) {
    printf("memory is too small, halt.\n");
    asm_halt();
}
// 剩余的空闲的内存
int freeMemory = this->totalMemory - usedMemory;

int freePages = freeMemory / PAGE_SIZE;
int kernelPages = freePages / 2;
int userPages = freePages - kernelPages;

int kernelPhysicalStartAddress = usedMemory;
int userPhysicalStartAddress = usedMemory + kernelPages * PAGE_SIZE;

int kernelPhysicalBitMapStart = BITMAP_START_ADDRESS;
int userPhysicalBitMapStart = kernelPhysicalBitMapStart + ceil(kernelPages, 8);

kernelPhysical.initialize((char *)kernelPhysicalBitMapStart, kernelPages,
kernelPhysicalStartAddress);
userPhysical.initialize((char *)userPhysicalBitMapStart, userPages,
userPhysicalStartAddress);
}

```

先读取之前在实模式下使用中断获取的内存大小，我们在内存中预留了部分内存。0x00000000~0x00100000（0MB-1MB）存放的是我们的内核，在预留内存中，1MB以上的剩余部分（1MB-2MB）存放内核页表。将剩余的空闲的内存等分为两部分，内核物理地址空间和用户物理地址空间，用户物理地址空间紧跟在内核物理地址空间后面。在1MB以下的空间处人为划分了存放位图的区域，用来存放内核空间和用户空间的位图（BitMap），之后对两部分空间的地址池进行初始化即可

物理页的分配和释放：

```

int MemoryManager::allocatePhysicalPages(enum AddressPoolType type, const int
count)
{
    int start = -1;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelPhysical.allocate(count);
    }
    else if (type == AddressPoolType::USER)
    {
        start = userPhysical.allocate(count);
    }
}

```

```

    return (start == -1) ? 0 : start;
}

void MemoryManager::releasePhysicalPages(enum AddressPoolType type, const int
paddr, const int count)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelPhysical.release(paddr, count);
    }
    else if (type == AddressPoolType::USER)
    {
        userPhysical.release(paddr, count);
    }
}

```

根据传入的AddressPoolType type来决定物理内存空间的分配，调用相应函数即可

子任务2

测试样例

```

int ker_addr = memoryManager.allocatePhysicalPages(KERNEL,3);
printf("Successfully allocated ker_addr :%d\n",ker_addr);
memoryManager.releasePhysicalPages(KERNEL,ker_addr,3);

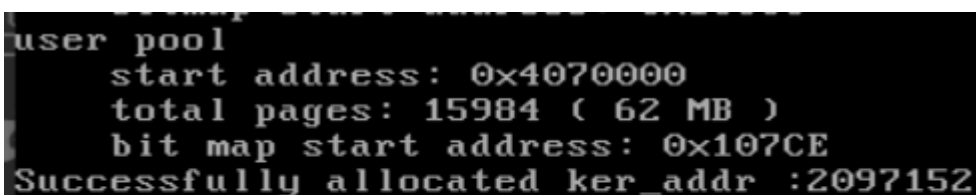
int user_addr = memoryManager.allocatePhysicalPages(USER,-3);
printf("Successfully allocated user_addr:%d\n",user_addr);
memoryManager.releasePhysicalPages(USER,user_addr,3);

printf("Done");

```

可能出现的bug:

1. 申请物理页时的count小于0导致程序运行缓慢（卡死）



```

user pool
start address: 0x4070000
total pages: 15984 ( 62 MB )
bit map start address: 0x107CE
Successfully allocated ker_addr :2097152

```

修改: `BitMap::allocate`(添加小于0的判断)

```
if (count <= 0)
    return -1;
```

结果

```
Successfully allocated ker_addr :2097152
Successfully allocated user_addr:0
Done_
```

2. 申请地址之后，没有考虑是否成功而调用该地址（0）的`release`函数，
`resources.release((address - startAddress) / PAGE_SIZE, amount);`会
使之后的索引为负数从而可能导致程序出错(实际运行时没有报错，不过存在风险)



```
../src/utils/bitmap.cpp
86     }
87
88     void BitMap::release(const int index, const int count)
89     {
90
91         for (int i = 0; i < count; ++i)
92         {
93             set(index + i, false);
94         }
95     }
96
97     char *BitMap::getBitmap()
98     {
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
remote Thread 1 In: BitMap::release L91
AddressPool::release (this=0x32984 <memoryManager+16>, address=0, a
    at ../src/utils/address_pool.cpp:25
(gdb) s
BitMap::release (this=0x32984 <memoryManager+16>, index=-16496, co
    at ../src/utils/bitmap.cpp:91
(gdb) print index
$1 = -16496
(gdb)
```

修改: `AddressPool::release`

```
void AddressPool::release(const int address, const int amount)
{
    if(address < startAddress){
        return ;
    }
    resources.release((address - startAddress) / PAGE_SIZE, amount);
}
```

Assignment 2 二级分页机制的实现

具体要求如下：

1. 实现内存的申请和释放，保存实验截图并能够在虚拟机地址空间中进行内存管理，截图并给出过程解释（比如：说明哪些输出信息描述虚拟地址，哪些输出信息描述物理地址）。注意：建议使用的物理地址或虚拟地址信息与学号相关联（比如学号后四位作为页内偏移），作为报告独立完成的个人信息表征。
2. 相比于一级页表，二级页表的开销是增大的，但操作系统中往往使用的是二级页表而不是一级页表。结合你自己的实验过程，说说相比于一级页表，使用二级页表会带来哪些优势

子任务1

实验步骤

实现二级分页机制：

```
void MemoryManager::openPageMechanism()
{
    // 页目录表指针
    int *directory = (int *)PAGE_DIRECTORY;
    // 线性地址0~4MB对应的页表
    int *page = (int *) (PAGE_DIRECTORY + PAGE_SIZE);

    // 初始化页目录表
    memset(directory, 0, PAGE_SIZE);
    // 初始化线性地址0~4MB对应的页表
    memset(page, 0, PAGE_SIZE);

    int address = 0;
    // 将线性地址0~1MB恒等映射到物理地址0~1MB
    for (int i = 0; i < 256; ++i)
    {
        // U/S = 1, R/W = 1, P = 1
        page[i] = address | 0x7;
        address += PAGE_SIZE;
    }

    // 初始化页目录项

    // 0~1MB
    directory[0] = ((int)page) | 0x07;
    // 3GB的内核空间
    directory[768] = directory[0];
}
```

```

// 最后一个页目录项指向页目录表
directory[1023] = ((int)directory) | 0x7;

// 初始化cr3, cr0, 开启分页机制
asm_init_page_reg(directory);

printf("open page mechanism\n");

}

```

内存申请:

1. 从虚拟地址池中分配若干连续的虚拟页。

```

int allocateVirtualPages(enum AddressPoolType type, const int count)
{
    int start = -1;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelVvirtual.allocate(count);
    }

    return (start == -1) ? 0 : start;
}

```

2. 对每一个虚拟页，从物理地址池中分配1页。
3. 为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理页内。

```

bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const int
physicalPageAddress)
{
    // 计算虚拟地址对应的页目录项和页表项
    int *pde = (int *)toPDE(virtualAddress);
    int *pte = (int *)toPTE(virtualAddress);

    // 页目录项无对应的页表，先分配一个页表
    if (!(*pde & 0x00000001))
    {
        // 从内核物理地址空间中分配一个页表
        int page = allocatePhysicalPages(AddressPoolType::KERNEL, 1);
        if (!page)
            return false;

        // 使页目录项指向页表
        *pde = page | 0x7;
        // 初始化页表
        char *pagePtr = (char *)(((int)pte) & 0xfffff000);
    }
}

```



```

        memset(pagePtr, 0, PAGE_SIZE);
    }

    // 使页表项指向物理页
    *pte = physicalPageAddress | 0x7;

    return true;
}

```

页内存分配的函数

```

int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
{
    // 第一步：从虚拟地址池中分配若干虚拟页
    int virtualAddress = allocateVirtualPages(type, count);
    if (!virtualAddress)
    {
        return 0;
    }

    bool flag;
    int physicalPageAddress;
    int vaddress = virtualAddress;

    // 依次为每一个虚拟页指定物理页
    for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
    {
        flag = false;
        // 第二步：从物理地址池中分配一个物理页
        physicalPageAddress = allocatePhysicalPages(type, 1);
        if (physicalPageAddress)
        {
            //printf("allocate physical page 0x%x\n", physicalPageAddress);

            // 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理页内。
            flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
        }
        else
        {
            flag = false;
        }

        // 分配失败，释放前面已经分配的虚拟页和物理页表
        if (!flag)
        {
            // 前i个页表已经指定了物理页
            releasePages(type, virtualAddress, i);
            // 剩余的页表未指定物理页
            releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
            return 0;
        }
    }
}

```

```
    return virtualAddress;
}
```

内存释放:

1. 对每一个虚拟页，释放为其分配的物理页。首先，由于物理地址池存放的是物理地址，为了释放物理页，我们要找到虚拟页对应的物理页的物理地址，如下所示。

```
int MemoryManager::vaddr2paddr(int vaddr)
{
    int *pte = (int *)toPTE(vaddr);
    int page = (*pte) & 0xfffff000;
    int offset = vaddr & 0xfff;
    return (page + offset);
}
```

2. 释放虚拟页。

```
void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int vaddr,
const int count)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelVirtual.release(vaddr, count);
    }
}
```

页内存释放的函数

```
void MemoryManager::releasePages(enum AddressPoolType type, const int
virtualAddress, const int count)
{
    int vaddr = virtualAddress;
    int *pte, *pde;
    bool flag;
    const int ENTRY_NUM = PAGE_SIZE / sizeof(int);

    for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
    {
        releasePhysicalPages(type, vaddr2paddr(vaddr), 1);

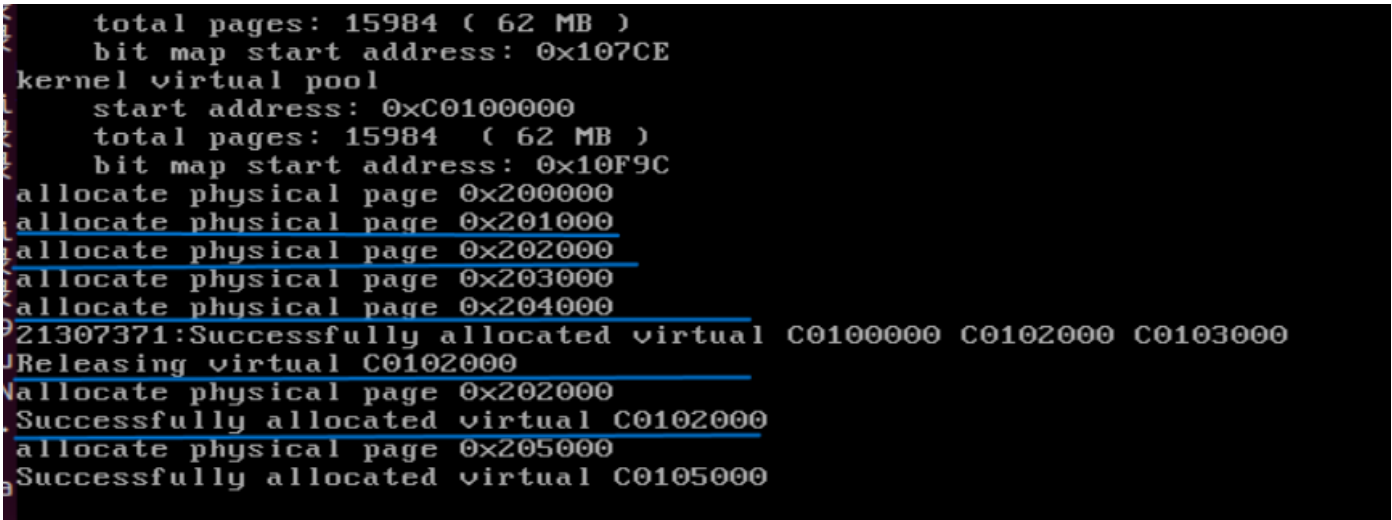
        // 设置页表项为不存在，防止释放后被再次使用
        pte = (int *)toPTE(vaddr);
        *pte = 0;
    }
}
```

```
    releaseVirtualPages(type, virtualAddress, count);  
}
```

测试代码

```
char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 2);  
char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);  
char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 2);  
  
printf("21307371:Successfully allocated virtual %x %x %x\n", p1, p2, p3);  
  
memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 1);  
printf("Releasing virtual %x\n", p2);  
p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);  
  
printf("Successfully allocated virtual %x\n", p2);  
  
p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);  
  
printf("Successfully allocated virtual %x\n", p2);
```

实验截图



```
total pages: 15984 ( 62 MB )  
bit map start address: 0x107CE  
kernel virtual pool  
start address: 0xC0100000  
total pages: 15984 ( 62 MB )  
bit map start address: 0x10F9C  
allocate physical page 0x200000  
allocate physical page 0x201000  
allocate physical page 0x202000  
allocate physical page 0x203000  
allocate physical page 0x204000  
21307371:Successfully allocated virtual C0100000 C0102000 C0103000  
Releasing virtual C0102000  
allocate physical page 0x202000  
Successfully allocated virtual C0102000  
allocate physical page 0x205000  
Successfully allocated virtual C0105000
```

分析：蓝线划分了每次操作，一开始先申请两个页，物理地址为0x200000和0x201000，虚拟地址为0xc0100000，其他申请操作同上，可以看到释放后的内存可以再次被分配

子任务二

二级页表的优势

当前进程只需要载入自己需要的页表即可，对于自己用不到的页表，则无需载入其对应的页目录中来，这样可以大大节约内存，并且访存次数也只需要两次即可。

Assignment 3 虚拟页内存管理的实现

具体要求如下：

1. 结合代码，描述虚拟页内存分配的三个基本步骤，以及虚拟页内存释放的过程。
2. 构造测试用例来分析虚拟页内存管理的实现是否存在bug。如果存在，则尝试修复并再次测试。否则，结合测试用例简要分析虚拟页内存管理的实现的正确性。
3. 在PDE（页目录项）和 PTE（页表项）的虚拟地址构造中，我们使用了第1023个页目录项。第1023个页目录项指向了页目录表本身，从而使得我们可以构造出PDE和PTE的虚拟地址。现在，我们将这个指向页目录表本身的页目录项放入第1000个页目录项，而不再是放入了第1023个页目录项。请同学们借助第1000个页目录项，构造出第141个页目录项的虚拟地址，和第891个页目录项指向的页表中第109个页表项的虚拟地址。

子任务1（和任务二的要求类似，不再重复）

子任务2

测试用例：(创建另一个进程来申请内存)

```
void second_thread(void *arg){
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    printf("Successfully allocated %x \n", p1);
}
void first_thread(void *arg)
{
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    printf("Successfully allocated %x \n", p1);
    asm_halt();
}
```

修改BitMap::allocate函数（延时）

```
// 存在连续的count个资源
if (empty == count)
{
    for(int i = 0; i < 1e8; i++){ //延时
        for (int i = 0; i < count; ++i)
        {
            set(start + i, true);
        }
    }
}
```

存在问题：

如果有多个线程同时申请内存，它们会共同访问Bitmap，产生竞争现象，即多个线程同时占有同一片内存空间，引起内存分配的错误

```
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
Successfully allocated C0100000
Successfully allocated C0100000
```

解决方法：

当多个线程想要访问Bitmap时，加一个信号量，确保每次只有一个线程能访问Bitmap。

```
int BitMap::allocate(const int count)
{
    if (count == 0)
        return -1;

    int index, empty, start;

    s.P();
    index = 0;
    while (index < length)
    {
        // 越过已经分配的资源
        while (index < length && get(index))
            ++index;

        // 不存在连续的count个资源
        if (index == length)
            return -1;

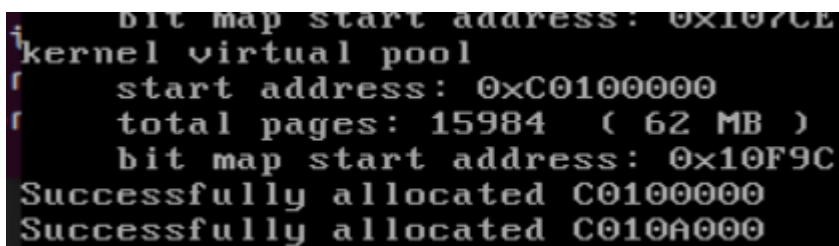
        // 找到1个未分配的资源
        // 检查是否存在从index开始的连续count个资源
        empty = 0;
```

```

start = index;
while ((index < length) && (!get(index)) && (empty < count))
{
    ++empty;
    ++index;
}

// 存在连续的count个资源
if (empty == count)
{
    for(int i = 0; i < 1e8; i++){
        for (int i = 0; i < count; ++i)
        {
            set(start + i, true);
        }
        s.V();
        return start;
    }
}
s.V();
return -1;
}

```



```

bit map start address: 0x107CE
kernel virtual pool
start address: 0xC0100000
total pages: 15984 ( 62 MB )
bit map start address: 0x10F9C
Successfully allocated C0100000
Successfully allocated C010A000

```

子任务3

1. 第141个页目录项的虚拟地址:

$pde[31:22] = 11\ 1110\ 1000$

$pde[21:12] = 11\ 1110\ 1000$

$pde[11:0] = 4 \times \text{virtual}[31:22] = 0010\ 0011\ 0100$

$pde = 0xfa3e8234$

2. 第891个页目录项指向的页表中第109个页表项的虚拟地址:

$pte[31:22] = 11\ 1110\ 1000$

$pte[21:12] = \text{virtual}[31:22] = 11\ 0111\ 1011$

pte[11:0] = 4 × virtual[21:12] = 0001 1011 0100

pte = 0xfa37b1b4

Assignment 4 页面置换算法的实现

(LRU) 设计思路:

1. 构建一个LRU数组（初始设置五个空闲帧）来记录虚拟页的访问时间（因为现在还没学习到页的访问，所以只能通过页的创建和释放来观察LRU数组的变化）
2. 在每次时钟中断时按时检查每个页面是否被访问，这样即可按一定周期更新 LRU 列表值。该列表维护在地址池数据结构中。
3. 当页面被访问（创建）时，记录此时的clock，方便后续页面的置换
4. 当没有空闲帧时，会在LRU数组中找到值最小（表示最早访问的）的，并将其置换掉（释放），从而创造一个新的空闲帧

代码实现:

class AddressPool 添加变量

```
class AddressPool
{
public:
    BitMap resources;
    int startAddress;
    int LRU[5];
    int clock;
    int virtual_num;
public:
    AddressPool();
    // 初始化地址池
    void initialize(char *bitmap, const int length, const int startAddress);
    // 从地址池中分配count个连续页，成功则返回第一个页的地址，失败则返回-1
    int allocate(const int count);
    // 释放若干页的空间
    void release(const int address, const int amount);
    void updateLRU();
    int swapout();
};
```

页面访问时间更新函数（利用页表项的A位来判断是否被再次访问）

```

void AddressPool::updateLRU()
{
    clock++;
    for (int i = 0; i < resources.length; i++)
    {
        if(!resources.get(i))
            // 如果该虚拟地址还未被占有，则跳过
            continue;
        //得到虚拟地址
        int virtual_addr = startAddress + i * PAGE_SIZE;
        //求出虚拟地址的页目录项
        unsigned int* pte = (unsigned int*)(0xffc00000 + ((virtual_addr &
0xffc00000) >> 10) + (((virtual_addr & 0x003ff000) >> 12) * 4));
        //观察页目录项A位是否被访问过
        if((*pte) & (1<<5))
        {
            printf("Accessing page %d\n", i);
            LRU[i] = clock;
            // 重新置零
            (*pte) = (*pte) & (~ 3 << 5);
        }
    }
}

```

页面置换函数

```

int AddressPool::swapout()
{
    //找出最早被访问的帧，来进行替换
    int min_time = clock + 1;
    int index = 0;
    for (int i = 0; i < resources.length; i++)
    {
        if(!resources.get(i)){
            // 如果该虚拟地址还未被占有，则跳过
            continue;
        }

        if(LRU[i] < min_time)
        {
            min_time = LRU[i];
            index = i;
        }
    }
    //返回被替换帧的虚拟地址
    return startAddress + index * PAGE_SIZE;
}

```

虚拟页申请函数


```

int MemoryManager::allocateVirtualPages(enum AddressPoolType type, const int count)
{
    int start = -1;
    int out_address;

    //申请页面超过限制
    if(count > kernelVirtual.virtual_num){
        return 0;
    }
    //如果没有足够的空闲帧，则在循环中不断产生空闲帧
    while (start == -1)
    {
        //尝试申请虚拟内存
        if (type == AddressPoolType::KERNEL)
        {
            start = kernelVirtual.allocate(count);
        }
        //申请成功
        if(start != -1) break;
        //申请失败（换出页面）
        if(type == AddressPoolType::KERNEL){
            out_address = kernelVirtual.swapout();
            printf("Release virtual 0x%x , %d pages\n",out_address,1);
            releasePages(AddressPoolType::KERNEL,out_address,1);
        }
    }

    return (start == -1) ? 0 : start;
}

```

测试函数（通过修改A位来模拟页面的访问）

```

void first_thread(void *arg)
{
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 6);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 5);

    //模拟访问页面0
    int virtual_addr = memoryManager.kernelVirtual .startAddress + 0* PAGE_SIZE;
    //求出虚拟地址的页目录项
    unsigned int* pte = (unsigned int*)(0xffc00000 + ((virtual_addr & 0xffc00000)
>> 10) + (((virtual_addr & 0x003ff000) >> 12) * 4));
    (*pte) = (*pte) | (1<<5);

    //延时
    for(int cnt = 0; cnt < 1e7; cnt++){
        char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 3);
        asm_halt();
    }
}

```

结果截图

```
bit map start address: 0x107CE
kernel virtual pool
start address: 0xC0100000
total pages: 5 ( 20 KB )
bit map start address: 0x10F9C
Unsuccessfully allocated virtual 6 pages
Successfully allocated virtual 0xC0100000, 5 pages
allocate physical page 0x200000
allocate physical page 0x201000
allocate physical page 0x202000
allocate physical page 0x203000
allocate physical page 0x204000
Accessing page 0
Release virtual 0xC0101000 , 1 pages
Release virtual 0xC0102000 , 1 pages
Release virtual 0xC0103000 , 1 pages
Successfully allocated virtual 0xC0101000, 3 pages
allocate physical page 0x201000
allocate physical page 0x202000
allocate physical page 0x203000
```

分析：初始化时设置共有五个空闲帧，所以当想要申请六个空闲帧时会失败；接着成功申请五个空闲帧，为了显示LRU算法，我在这时模拟访问了第一个页面（改变了该页面的访问时间），接着尝试再申请三个空闲帧，因为此时已经没有空闲帧了，所以需要进行页面置换，页面0刚被访问，所以页面1，2，3被置换出来（释放），然后分配三个空闲帧。

5. 总结(对实验过程中遇到的问题进行总结，可以提出对实验设置的改进意见)

此次实验内容难度较大，虽然理论课上已经讲了大致的知识点，但到了实际操作，还是有许多难点，比如虚拟地址池、物理地址池位置的分配，pte、pde虚拟地址的构造（因为cpu默认访问是虚拟地址，所以需要在页目录表中专门设置页目录表的地址，有点像是在页目录表做循环，每循环一次，就相当于从虚拟地址到物理地址转换过程少走一步），还有就是页面置换算法的实现，虽然道理都懂，但是一到代码层面就容易一头雾水，在和同学的讨论以及搜索引擎的帮助下，最终实现该算法，对页面置换算法有了更深的理解。

6. 参考资料清单