

操作系统实验报告

18364067 罗淦元

一、实验要求

在github上创建os-assignment2项目，提供

- (1) 虚存管理模拟程序源代码及结果（存成文本文件）；
- (2) 实验报告（word/pdf），包含所有实验的基本过程描述。

二、实验过程

1.虚存管理模拟程序

1.1 Chapter 10. Programming Projects: Designing a Virtual Memory Manager (OSC 10th ed.)，30分。

- (1) 保存为vm.c，使用如下测试脚本test.sh，进行地址转换测试，并和correct.txt比较

```
#!/bin/bash -e
echo "Compiling"
gcc vm.c -o vm
echo "Running vm"
./vm BACKING_STORE.bin addresses.txt > out.txt
echo "Comparing with correct.txt"
diff out.txt correct.txt
```

注：本小题不要求实现Page Replacement，TLB分别实现FIFO和LRU两种策略。

实验思路首先是通过一个掩码从读取的虚拟地址中提取页码和偏移，然后从TLB或页表中获取对应的帧码，并通过偏移读取到物理内存对应位置。当页表中找不到对应的页码则从后备存储中载入到内存里。从address.txt文件中获取的是一个16位的数字，其前8位是页码，后八位是偏移。因此可以通过位操作来提取出两者。如下。

```
/*页码和偏移的掩码，用于从虚拟地址地区页号和偏移*/
uint8_t pageMask(uint32_t integer){
    uint32_t mask = 255;           //和255进行与操作能够提取出后8位
    integer = integer >> 8;         //如果要提取前八位，可以先右移
    uint8_t page = integer & mask;
    return page;
}

uint8_t biasMask(uint32_t integer){
    uint32_t mask = 255;
    uint8_t bias = integer & mask;
    return bias;
}
```

首先是FIFO算法，通过定义一个TLB结构和page结构来形成TLB和数组。并通过一个16和256大小的数组，以队列的方式组织。每次在TLB中查找目标页后，如果找到则返回对应帧号。若找不到则在页表中重复查询操作。更新TLB时，则通过或运算直接覆盖TLB中最后一条条目的下一条，即将最久以前加入的条目直接覆盖为新条目。由于FIFO的实现非常简单，在这里不再过多叙述。相关完整源代码已经提交到github上vm.c文件中。

```
typedef struct {
    int page;
    int frame;
} TLBelem;

/*用FIFO时使用的page结构*/
typedef struct {
    int page;
    int frame;
} page;

int searchTLB(uint8_t page){
    for(int i = 0; i < TLB_SIZE; i++){
        if(TLB[i].page == page){
            //printf("TLB hit! on %d\n", i);
            return TLB[i].frame;
        }
    }
    //printf("TLB miss!\n");
    return -1;
}

void TLB_update_FIFO(int page, int frame){
    //printf("TLB_update_FIFO\n");
    TLB[TLB_HEAD].page = page;
    TLB[TLB_HEAD].frame = frame;
    TLB_HEAD = (TLB_HEAD+1)%TLB_SIZE;
}

int searchPageTable(uint8_t page){
    for(int i = 0; i < PAGETABLE_SIZE; i++){
        if(pageTable[i].page == page){
            //printf("page hit!\n");
            return pageTable[i].frame;
        }
    }
    //printf("page miss!\n");
    return -1;
}

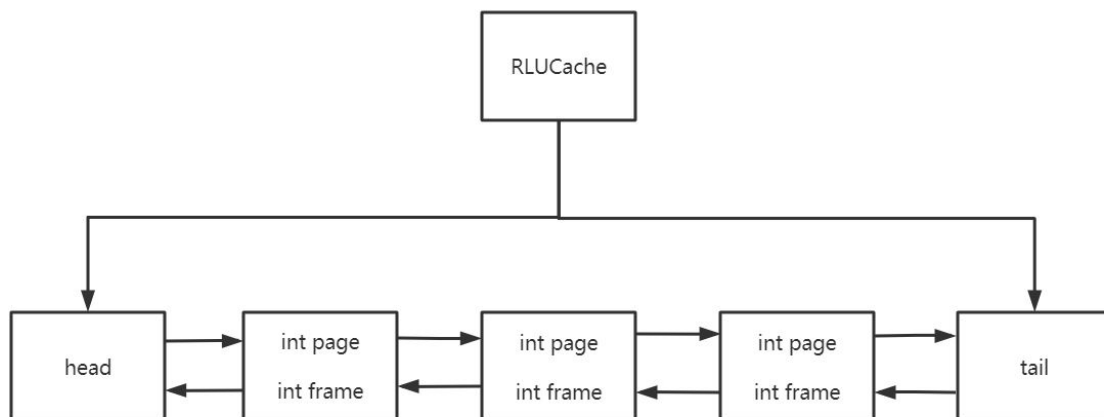
void pageTable_update_FIFO(int page, int frame){
    int swapOut;
    pageTable[PAGE_HEAD].page = page;
    pageTable[PAGE_HEAD].frame = frame;
    PAGE_HEAD = (PAGE_HEAD+1)%PAGETABLE_SIZE;
}
```

在test.sh中运行结果如下。

```
osc@ubuntu:~/final-src-osc10e/ch10$ sh ./test.sh
Compiling
Running vm
Comparing with correct.txt
```

而LRU算法相对要复杂一些。如果使用一个整数来记录每个TLB或页表中节点的最长未访问时间，需要遍历整个TLB或页表来查找该节点。而同时和FIFO类似，如果采用遍历的方式来查找需要的页条目，两个操作（删除最久未访问节点和查找条目）都将会达到 $O(N)$ 的复杂度。其访问效率将非常低，且随着页表和TLB大小的增大访问时间变得不可接受。

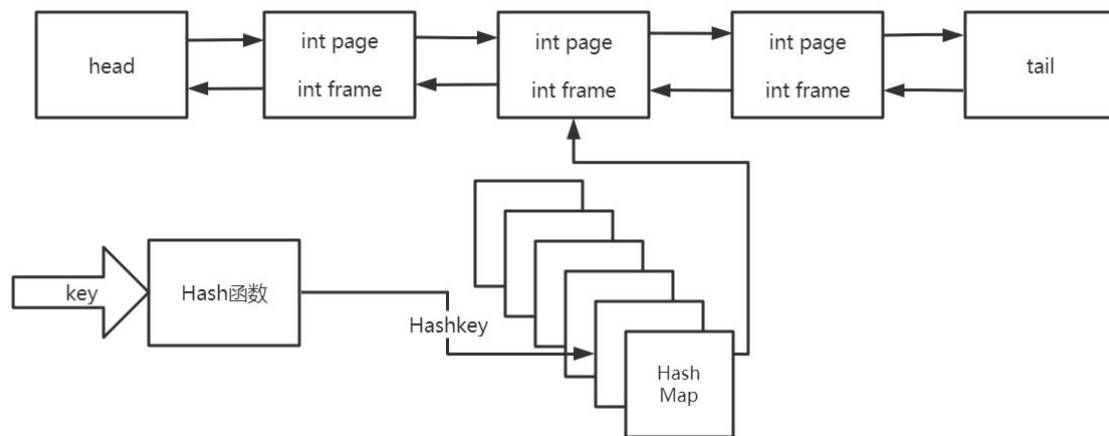
考虑到这一问题，本次实验决定尝试使用双向链表来维护LRU缓存结构。LRU缓存中带有指向一个双向链表首尾的指针，每次增加条目时，只要没有超出容积，就在表头添加一个节点，从而使得新添加的节点总在更靠近表头的位置。



```
/*双向链表结构*/
typedef struct DLinkedNode{
    int key;
    int frame;
    struct DLinkedNode* prev;
    struct DLinkedNode* next;
} DLinkedNode;
/*双向链表结构定义结束*/
```

而每次访问一个条目时，将会把该条目移动至表头，从而使得表尾总是最长时间未访问条目。通过这种方法，删除最久未访问节点的时间复杂度将被降为 $O(1)$ （通过删除`tail->prev`直接删除该节点），从而大大提高了效率。

除此之外，本实验使用哈希表来完成key值到节点的映射。这种做法的缺点很明显，例如要实现一个16条目的LRU缓存，则最少需要一块额外的最少16个哈希表条目大小的空间用于存放哈希表。但是相应的，在最理想情况下，我们能够实现对于条目的查找复杂度为 $O(1)$ ，而最糟糕的情况下也不会超过遍历的查找方式 $O(N)$ 。



```
//哈希表元素，包含一个键值、一个数据指针和一个状态
//其中，数据指针指向LRU缓存维护的双向链表中地元素，本身不存储值
//stat在Empty状态表示未使用，valid为已使用并分配内存，unvalid表示已删除并分配内存
typedef struct{
    int key;
    DLinkedNode* node;
    HashStat stat;
}HashElem;
```

为了对内存实施清晰有效的管理，我们设定所有的内存完全交给双向链表管理，哈希表相关函数仅仅负责接收指针和key(在本题中即page)来建立key到指针的映射。同时使用求余运算作为哈希函数，这是可以在未来进行优化的。

```
//哈希函数:key对哈希表大小求余
size_t HashFunction(int key){
    return key%HashMaxSize;
}
```

最后，基于实现的HashMap和DLinkedNode结构，最终建立出了LRUCache结构用于进行LRU算法。

```
/*LRU缓存定义*/
/*LRU缓存由一张哈希表和一个双向链表组成*/
/*哈希表主要负责查找，链表维护了一个以节点最近*/
/*访问顺序排列的队列，并管理空间分配*/
typedef struct LRUCache{
    HashMap cache;
    int size;
    int capacity;
    DLinkedNode head;
    DLinkedNode tail;
}LRUCache;
```

在源代码vm.c中，本实验已经将哈希表、双向链表和LRU缓存结构的相关函数进行了模块化的封装，最后关键的函数在于put和get函数。

```
/*put函数负责向LRU中加入一个条目*/
/*如果key存在于LRU缓存中，则将其移到表头*/
/*否则，如果缓存未满，加入表头，如果已满，删除表尾并加入表头*/
```

```

void put(LRUCache* lc, int key, int value){
    int key_;
    if(HashMapFindKey(&(lc->cache), key, &key_)){           //如果哈希表有key元素
        LRURemoveNode(lc, (lc->cache.data[key_].node));       //删除key元素，并稍
        后在表头重加。由于其未从哈希表删除，我们可以通过下标访问被remove但未delete的节点
    }
    if(lc->size == lc->capacity) {                             //如果已满，删除尾部元素
        HashRemove(&(lc->cache), lc->tail->prev->key);
        LRURemoveTail(lc);
    }
    DLinkedNode* newNode = (DLinkedNode*)malloc(sizeof(DLinkedNode)); //建立新节
    点
    newNode->key = key; newNode->frame = value;                //节点的键
    值分别是虚拟内存页和帧页码
    LRUaddNode(lc, newNode);                                   //将节点添
    加到LRU缓存
    HashMapInsert(&(lc->cache), key, newNode);
}

/*get函数负责从哈希表中找到LRU链表节点，*/
int get(LRUCache* lc, int key){
    int key_;
    if(HashMapFindKey(&(lc->cache), key, &key_)==0) return -1; //在哈希表中找到
    key对应哈希表下标
    DLinkedNode* tempNode = lc->cache.data[key_].node;         //备份该节点
    LRURemoveNode(lc, tempNode);                               //删除该节点
    LRUaddNode(lc, tempNode);                                   //然后将其提到最前面
    return tempNode->frame;                                     //返回其值
}

```

其中put函数将接受一个新键值作为参数，首先检查确认链表中不存在这一键值，然后当LRU缓存已满，则删除链表中尾部元素，同时将为新节点分配内存并加入到链表头，接着更新哈希表。get函数则从哈希表中寻找键值对应的条目，如果找不到返回-1表示获取失败。否则将节点从链表中移除重加，即移动到头部，然后返回对应的值。基于这两个接口，我们可以很容易实现LRU算法并实现TLB和页表的更新。在test.sh中运行结果如下。

```

osc@ubuntu:~/final-src-osc10e/ch10$ sh ./test.sh
Compiling
Running vm
Comparing with correct.txt

```

由于vm.c在第二小题中需要进行主函数的更改，本实验将第一题封装为函数

```
int question1(int argc, char* argv[]);
```

只需要在主函数中将argc和argv直接传入到函数，并注意添加BACKING_STORE.bin和addresses.txt参数即可。“-r”参数是可选的，参数可以使用“FIFO”和“LRU”选择使用的算法。当此项为空时将默认使用LRU算法进行测试。

(2) 实现基于LRU的Page Replacement; 使用FIFO和LRU分别运行vm (TLB和页置换统一策略)，打印比较Page-fault rate和TLB hit rate，给出运行的截屏。提示：通过getopt函数，程序运行时通过命令行指定参数。

在第一题中，本实现已经考虑了页表小于物理内存的情况，因此可以直接在源代码中改变页表大小常量。

```
#define PAGETABLE_SIZE 128 //页表大小
```

程序即能够正常进行页表Page Replacement。运行结果如图。

```
osc@ubuntu:~/final-src-osc10e/ch10$ ./vm BACKING_STORE.bin addresses.txt -rFIFO
TLBHitRate: 5.400000%
pageFaultRate: 53.800000%
osc@ubuntu:~/final-src-osc10e/ch10$ ./vm BACKING_STORE.bin addresses.txt -rLRU
TLBHitRate: 5.500000%
pageFaultRate: 53.700000%
```

1.2 编写一个简单trace生成器程序

可以用任意语言，报告里面作为附件提供。运行生成自己的addresses-locality.txt，包含10000条访问记录，体现内存访问的局部性（参考Figure 10.21, OSC 10th ed.），绘制类似图表（数据点太密的话可以采样后绘图），表现内存页的局部性访问轨迹。然后以该文件为参数运行vm，比较FIFO和LRU策略下的性能指标，最好用图对比。给出结果及分析，10分。

2.xv6-lab-2020页表实验（Lab:page tables），20分

完成Print a page table任务。要求按图1格式打印页表内容；其中括号内表示页表项权限，R表示可读，W表示可写，X表示可执行，U表示用户可访问。物理页后的数字（pa 32618）表示第几个物理页帧。要求在报告中提供实现所需的源代码和运行截屏，代码要求有充分注释。然后，回答接下来的6个问题（分别对应代码注释行中的标签）。

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 () pa 32618(th pages) //问题1
.. ..0: pte 0x0000000021fda401 () pa 32617(th pages)
.. .. ..0: pte 0x0000000021fdac1f (RWXU) pa 32619(th pages) //问题2
.. .. ..1: pte 0x0000000021fda00f (RWX) pa 32616(th pages) //问题3
.. .. ..2: pte 0x0000000021fd9c1f (RWXU) pa 32615(th pages) //问题4
..255: pte 0x0000000021fdb401 () pa 32621(th pages)
.. ..511: pte 0x0000000021fdb001 () pa 32620(th pages)
.. .. ..510: pte 0x0000000021fdd807 (RW) pa 32630(th pages) //问题5
.. .. ..511: pte 0x0000000020001c0b (RX) pa 7(th pages) //问题6
```

实现代码如下图，放置在vm.c文件中。为了格式化输出，我们可以使用一个格外的参数depth记录页表访问深度或者在vmprint中使用一个独立的子函数完成。这里选择前者实现，当调用外界vmprint时，depth初值应为0。

```
void vmprint(pagetable_t pagetable, int depth){
    if(depth == 0) printf("page table %p\n", pagetable); //最高层页表，输出页表地址
    for(int i = 0; i < 512; i++){ //遍历页表项
        pte_t pte = pagetable[i]; //获取一个条目
        if((pte & PTE_V)){ //如果条目有效
            printf("..");
            uint64 pa = PTE2PA(pte); //从页表项中获取物理地址
            for(int j = 0; j < depth; j++) printf(" .."); //根据当前深度格式化输出
            printf("%d: pte %p ", i, pte);
            printf("(");
            if(pte & (PTE_R)) printf("R"); //查看是否有读权限
            if(pte & (PTE_W)) printf("W");
            if(pte & (PTE_X)) printf("X");
            if(pte & (PTE_U)) printf("U");
            printf(") ");
        }
    }
}
```



```

//pa中第12位-55位是PPN，再减去一个起始地址得到页号
printf("pa %d(th pages)\n", ((pa>>12)-0x80000));
//如果pte不可读不可写不可执行，说明这页页表的子项应该是下一层页表的地址，则深度+1并
输出下一层页表
    if((pte & (PTE_R|PTE_W|PTE_X)) == 0) vmprint((pagetable_t)pa,
depth+1);
    }
}
}

```

其中PTE2PA () 函数在riscv.h中被定义为，即去掉RSW、D、A等位后左移12个offset位得到。

```
#define PTE2PA(pte) (((pte) >> 10) << 12)
```

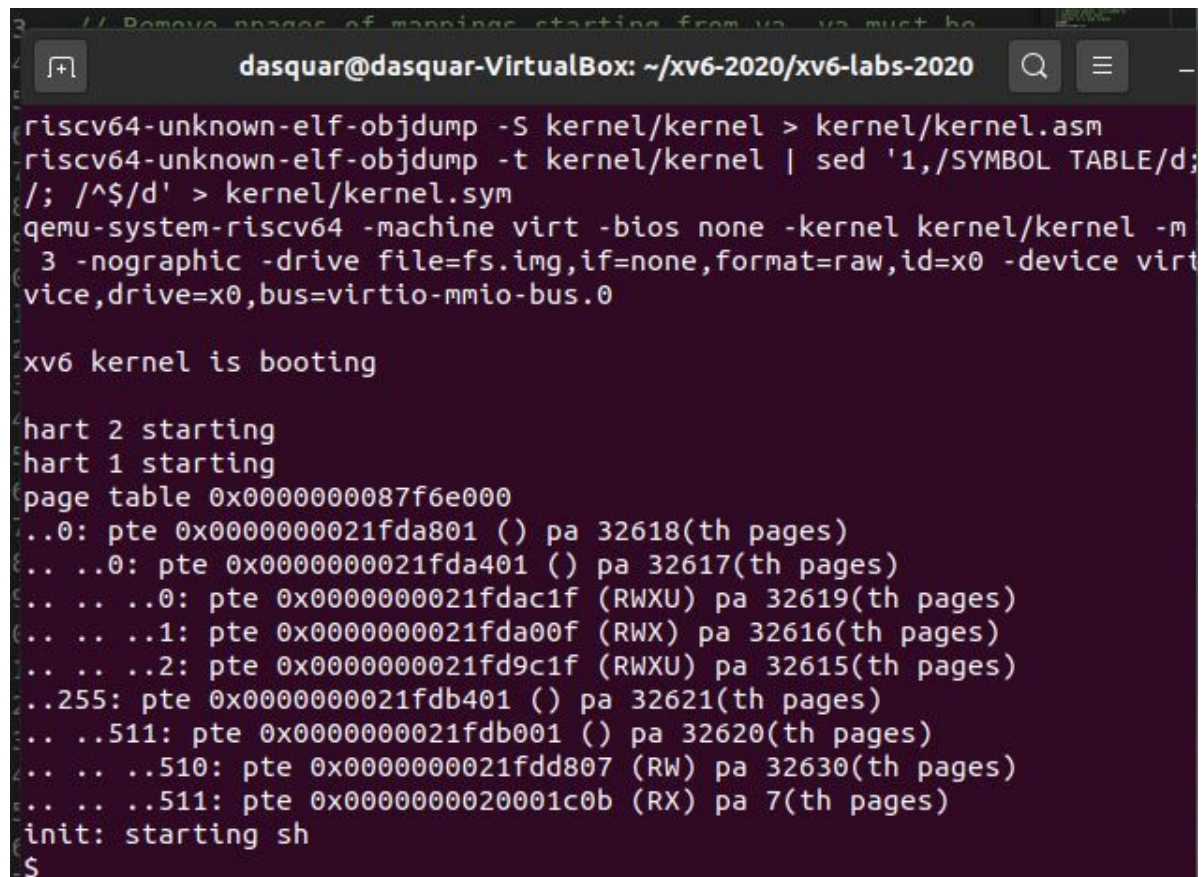
而PTE_V、PTE_R等权限位掩码在riscv.h中被定义为

```

#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access

```

其运行结果如图所示。



```

// Remove pages of mappings starting from va - va must be
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
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
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
```

问题1：为什么第一对括号为空？32618在物理内存的什么位置，为什么不从低地址开始？结合源代码内容进行解释。

第一对括号是根页表，包含了下一层页表的物理地址，只能在内核态进行读写，对用户进程不可见。对于用户态，前两层页表都是指令不可访问的，同样指令也就不可读写和执行，因此括号内权限为空。本xv6版本的KERNBASE和PHYSTOP在memlayout.h中定义为

```
// the kernel expects there to be RAM
// for use by the kernel and user pages
// from physical address 0x80000000 to PHYSTOP.
#define KERNBASE 0x80000000L
#define PHYSTOP (KERNBASE + 128*1024*1024)
```

由于页大小为4KB，这一空间中一共有32768页。而32618页处于这一空间中较高地址部分。在kinit函数中，xv6通过freerange(end, (void*)PHYSTOP);函数初始化了内核后到PHYSTOP为止的物理空间，并将空闲物理空间通过run结构组织起来。

```
void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);           //从p向上按页取整
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
        kfree(p);
}

kfree(void *pa)
{
    struct run *r;

    //如果pa不是一页倍数、物理地址比栈
    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");

    memset(pa, 1, PGSIZE);                            //内存初始化

    r = (struct run*)pa;

    acquire(&kmem.lock);
    r->next = kmem.freelist;                            //把物理地址pa指向的空间组织为链表
    kmem.freelist = r;                                  //头结点指向空余内存中地址最大的
    release(&kmem.lock);
}
```

使得地址大的空余内存存在freelist中更靠近表头。而调用kalloc分配物理内存时，就可以通过表头直接取得一页大小的内存空间。而获得地址时，kalloc函数直接从freelist表头获取地址，导致先分配的内存地址更高。

问题2：这是什么页？装载的什么内容？结合源代码内容进行解释。

这是一段 program segment，负责存储程序代码。它在exec.c中被分配

```
if((sz1 = uvmmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
```

其中，ph是一个Program section header结构，exec从initcode中被调用，通过readi函数从ip指向的文件中读取init的二进制代码到ph结构。

```
if(readi(ip, 0, (uint64*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
```

然后通过uvmmalloc为init进程增长一次空间。因为xv6使用的是三层页表结果，在uvmmalloc及其中的mappage的过程中，创建出了打印结果中前三行的三层页表。然后loadseg将一段 program segment 载入到虚拟地址为ph.vaddr所指的位置中

```
if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
```

问题3：这是什么页，有何功能？为什么没有U标志位？

这一页在exec.c中的

```
if((sz1 = uvmmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
```

被分配，然后在其后

```
uvmclear(pagetable, sz-2*PGSIZE);
```

被马上取消了用户模式访问权限。exec在栈页下面会放置这样一个不可访问页guard page，这样程序试图使用超过一页的空间时就会因为访问到这一区域而报错，同时这一页能够帮助exec处理太大的参数arguments，使得copyout函数试图将变量复制到用户栈时，会发现目标页不可访问而返回-1。为了实现这一目的，其没有U标志位。

问题4：这是什么页？装载的什么内容？指出源代码初始化该页的位置。

这是用户栈，和上一页一起在

```
if((sz1 = uvmmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
```

中被分配，并由uvmmalloc中的

```
memset(mem, 0, PGSIZE);
```

初始化为0后建立页表映射，并在exec.c的下一段中将argument和其他推入到用户栈中

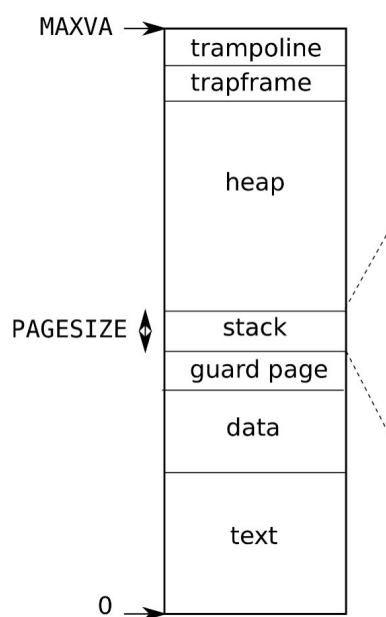
```
// 装载argument
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16;
    if(sp < stackbase)
        goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0) //推入一个
argument
        goto bad;
    ustack[argc] = sp;
}
ustack[argc] = 0;
```

然后压入了指向这些argument的指针和一个空指针。

```
// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
    goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
    goto bad;
```

问题5：这是什么页，为何没有X标志位？

这是trapframe。根据xv6 book 其位于虚拟地址的高位，保存了一些关键信息。



这些信息包括帮助trap函数判断是什么引起了中断的标志、系统调用号、内核栈顶地址等等。显然其不是可执行的数据，同时也不允许用户态的更改。

问题6：这是什么页，为何没有W标志位？装载的内容是什么？为何这里的物理页号处于低地址区域（第7页）？结合源代码对应的操作进行解释。

这是trampoline。这段代码不允许进行更改，并装载了trampoline.s中的代码。其虚拟页号总是位于整个虚拟地址的最高位（511），保证在不同的进程中都可以正确进入trampoline中。

由于其由系统内核创建。在kvm_init中的kvm_make中，按照先后顺序，内核将trampoline分配在了物理地址第7页。

```
pagetable_t
kvm_make(void)
{
    pagetable_t kpgtbl;

    kpgtbl = (pagetable_t) kalloc();          //分配一页用于放置根页表
    memset(kpgtbl, 0, PGSIZE);

    kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

    kvmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

    kvmmap(kpgtbl, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

    kvmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

    kvmmap(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R |
PTE_W);

    //将trampoline存储到物理内存，并添加页表映射
    kvmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);    //物
理地址第7页

    // 映射内核栈
    proc_mapstacks(kpgtbl);

    return kpgtbl;
}
```

它完全由trampoline.s中载入，并只能读取和执行，同时只能属于内核，只能由内核对代码进行访问。每个进程虚拟地址中的trampoline都指向物理地址中这一位置

3.xv6-lab-2020内存分配实验 (Lab: xv6 lazy page allocation), 40分

3.1 完成Lazy allocation子任务，要求echo hi正常运行，报告中可以描述自己的尝试过程，以及一些中间变量。

首先删除sys_sbrk中的grow_proc操作，仅仅更改进程的size。

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;           //更改进程大小,但是暂未分配空间
    myproc()->sz += n;
    return addr;
}
```

考虑到n有可能为负数，其中如果n绝对值大于sz且为负数的情况，需要异常处理。

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;

    struct proc* p = myproc();
    addr = p->sz;           //addr是p原大小,同时也是分配内存的起始位置
    if(n < 0){              //新大小比进程原大小小
        if(p->sz+n < 0) return -1;    //不能使程序大小为负
        uvmdealloc(p->pagetable, addr, addr+n);    //释放部分物理内存
    }
    p->sz += n;             //更新进程大小
    return addr;
}
```

而单纯这样处理，会导致usertrap的panic。原因是usertrap无法判断这一次trap为什么发生，以及uvmunmap函数发现有空间未分配。

```
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
          sepc=0x00000000000012ac stval=0x0000000000004008
panic: uvmunmap: not mapped
```

为了解决这一问题，我们自然可以把目光放到这两个函数上。对于uvmunmap函数，有两个判断语句会对lazy allocation产生影响。

```
if((pte = walk(pagetable, a, 0)) == 0)
    panic("uvmunmap: walk");
if((*pte & PTE_V) == 0)
    panic("uvmunmap: not mapped");
```

前者通过将walk函数的alloc参数为0，将walk函数以一种巧妙的方式重新运用，使walk函数从上到下查询页表，并且当任意一级页表PTE_V位无效时产生一次panic。如果正常返回，将得到最低一层的物理地址。第二条将判断这一地址是否有效。我们可以简单地跳过这两条判断，将使得uvmunmap对于这种未分配的情况视为一种正常情况忽略，当做该页成功释放。

```

if((pte = walk(pagetable, a, 0)) == 0)
    continue;
if((*pte & PTE_V) == 0)
    continue;

```

而在实验中，一开始发现在注释掉panic语句后会发生kfree panic错误，经过仔细检查发现，如果不在if语句中增加continue语句，会导致这些未被分配的内存存在其后

```

if(do_free){
    uint64 pa = PTE2PA(*pte);
    kfree((void*)pa);
}

```

产生尝试释放未分配内存的错误，continue是不可以去除的。

对于usertrap,我们根据提示可以知道当r_scause()寄存器为13或15时，说明当次陷入由页读取或也写入造成，即产生了一次缺页错误。那么对于这种情况，我们需要做的就是为缺失的页分配一块内存，然后返回到读写页的部分。

```

else if (r_scause() == 13 || r_scause() == 15){
    printf("Page Fault Occured\n");
    uint64 va = r_stval();
    //如果发生缺页错误
    //r_stval()得到产生缺页错误
    //的位置
    if(va < p->sz && va > PGROUNDDOWN(p->trapframe->sp)){
        //如果位置有效
        uint64 ka = (uint64) kalloc();
        //尝试分配一块地址
        if (ka == 0) p->killed = 1;
        //分配失败，杀死进程
        else{
            memset((void*)ka, 0, PGSIZE);
            //分配成功，初始化
            va = PGROUNDDOWN(va);
            //地址取整
            //尝试映射空间到页表的va位置
            if (mappages(p->pagetable, va, PGSIZE, ka, PTE_U | PTE_W | PTE_R) != 0){
                //如果失败，释放空间并杀死进程
                kfree((void*)ka);
                p->killed = 1;
            }
        }
    }
    //曾经遇到问题的位置
    else{
        //如果缺页错误位置大于进程size或者虚拟地址超出了用户栈
        p->killed = 1;
        //杀死进程
    }
} else {
    //printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    //printf("sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
}

```

到此位置，echo hi指令产生缺页错误时，就能够被正确分配空间。

```

$ echo hi
Page Fault Occured
Page Fault Occured
hi

```

在实验中碰到的一个问题出现在上述标注的位置。因为这一部分和紧接着的else语句即为相似，很容易被忽略掉。这会导致程序因为if语句进入到缺页错误处理区块，而当分配的内存非法时，无法进入p->killed = 1;语句导致进程不被杀死进入死循环。

3.2 完成Lazytests and Usertests子任务。对于Lazytests，要求屏幕输出如下图所示；对于usertests任务，要求通过所有除sbrkarg之外的测试。给出运行截屏。在阅读报告中提供代码修改片段，说明针对哪些文件，哪些函数进行了修改，新代码加上充分注释；可以写一些体会。

lazytests和usertests主要需要解决的问题有

- 处理sbrk参数为负数的情况（已完成，前一题sysproc.c的sys_sbrk()函数中）
- 页错误发生在一个超过程序大小的虚拟地址的情况（已完成，前一题trap.c的usertrap()函数中）
- fork () 函数中，父进程到子进程的内存拷贝问题
- 系统调用时，发生了地址的内存未分配的情况
- kalloc未能成功分配内存的情况（已完成，前一题trap.c的usertrap()函数中）
- 在用户栈低位的无效页情况（已完成，前一题trap.c的usertrap()函数中）

那么需要解决的主要问题主要集中在fork函数和系统调用上。

对于fork函数，我们首先追踪到造成错误的位置

```
if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){ //复制父进程的用户内存到
子进程
    freeproc(np); //创建子进程失败，释放子
进程内存
    release(&np->lock);
    return -1;
}
```

这一位置调用的uvmcopy函数使用了和uvmunmap()类似的做法，通过walk和PTE_V来对页表页的有效性进行判断。

```
if((pte = walk(old, i, 0)) == 0)
    panic("uvmcopy: pte should exist");
if((*pte & PTE_V) == 0)
    panic("uvmcopy: page not present");
```

同样的解决方案也是可行的。

```
if((pte = walk(old, i, 0)) == 0)
    continue;
if((*pte & PTE_V) == 0)
    continue;
```

而对于系统调用问题，主要就是要找到系统调用中通过什么函数把虚拟地址映射到物理地址中并添加相应页表。通过查询可以看到，copyin/copyout/copyinstr/loadseg等函数均使用walkaddr函数来转换虚拟和物理内存地址。那么我们可以在walkaddr中处理缺页问题，保证获得一个有效虚拟地址时能够返回一个有效的物理地址。

具体的实现和前面的usertrap相似。


```

// Look up a virtual address, return the physical address,
// or 0 if not mapped.
// Can only be used to look up user pages.
uint64
walkaddr(pagetable_t pagetable, uint64 va)
{
    pte_t *pte;
    uint64 pa;

    if(va >= MAXVA)
        return 0;

    pte = walk(pagetable, va, 0);                //获取物理地址

    struct proc* p = myproc();                  //当前进程
    if(pte == 0 || (*pte & PTE_V) == 0)          //如果物理地址无效，尝试分配
    {
        if (va >= p->sz || va < PGROUNDUP(p->trapframe->sp)) //超出进程大小或超出用户栈
            return 0;                                       //转换失败
        uint64 ka = (uint64)kalloc();                 //尝试分配内存
        if (ka == 0) return 0;                         //如果kalloc分配失败，返回

        //如果映射物理地址到虚拟地址失败，释放空间并返回
        if (mappages(p->pagetable, PGROUNDUP(va), PGSIZE, ka,
PTE_W|PTE_X|PTE_R|PTE_U) != 0){
            kfree((void*)ka);
            return 0;
        }
        //分配成功，返回物理地址
        return ka;
    }

    /*
    if(pte == 0)
        return 0;
    if((*pte & PTE_V) == 0)
        return 0;
    */

    if((*pte & PTE_U) == 0)
        return 0;
    pa = PTE2PA(*pte);
    return pa;
}

```

到此就完成了简单的lazy allocation机制。其运行结果如下。由于lazytests中的fault address和mappages failed过长，故和测试本身分开输出截图。

usertests:

```
init: starting sh
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test exectest: OK
```

```
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: OK
test sbrkfail: OK
test sbrkarg: OK
test validate: OK
test stacktest: OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

lazytasks:


```
usertrap(): fault address 0x0000000023584000 beyond heap range
usertrap(): fault address 0x00000000235c4000 beyond heap range
usertrap(): fault address 0x0000000023604000 beyond heap range
usertrap(): fault address 0x0000000023644000 beyond heap range
usertrap(): fault address 0x0000000023684000 beyond heap range
usertrap(): fault address 0x00000000236c4000 beyond heap range
usertrap(): fault address 0x0000000023704000 beyond heap range
usertrap(): fault address 0x0000000023744000 beyond heap range
usertrap(): fault address 0x0000000023784000 beyond heap range
usertrap(): fault address 0x00000000237c4000 beyond heap range
usertrap(): fault address 0x0000000023804000 beyond heap range
usertrap(): fault address 0x0000000023844000 beyond heap range
usertrap(): fault address 0x0000000023884000 beyond heap range
usertrap(): fault address 0x00000000238c4000 beyond heap range
usertrap(): fault address 0x0000000023904000 beyond heap range
usertrap(): fault address 0x0000000023944000 beyond heap range
usertrap(): fault address 0x0000000023984000 beyond heap range
usertrap(): fault address 0x00000000239c4000 beyond heap range
usertrap(): fault address 0x0000000023a04000 beyond heap range
usertrap(): fault address 0x0000000023a44000 beyond heap range
usertrap(): fault address 0x0000000023a84000 beyond heap range
usertrap(): fault address 0x0000000023ac4000 beyond heap range
usertrap(): fault address 0x0000000023b04000 beyond heap range
usertrap(): fault address 0x0000000023b44000 beyond heap range
usertrap(): fault address 0x0000000023b84000 beyond heap range
```

running test lazy unmap

```
mappages failed, va=0x0000000000004000
mappages failed, va=0x000000000001004000
mappages failed, va=0x000000000002004000
mappages failed, va=0x000000000003004000
mappages failed, va=0x000000000004004000
mappages failed, va=0x000000000005004000
mappages failed, va=0x000000000006004000
mappages failed, va=0x000000000007004000
mappages failed, va=0x000000000008004000
mappages failed, va=0x000000000009004000
mappages failed, va=0x00000000000a004000
mappages failed, va=0x00000000000b004000
mappages failed, va=0x00000000000c004000
mappages failed, va=0x00000000000d004000
mappages failed, va=0x00000000000e004000
mappages failed, va=0x00000000000f004000
mappages failed, va=0x000000000010004000
mappages failed, va=0x000000000011004000
mappages failed, va=0x000000000012004000
mappages failed, va=0x000000000013004000
mappages failed, va=0x000000000014004000
mappages failed, va=0x000000000015004000
mappages failed, va=0x000000000016004000
mappages failed, va=0x000000000017004000
mappages failed, va=0x000000000018004000
mappages failed, va=0x000000000019004000
```

```
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
test lazy unmap: OK
running test out of memory
test out of memory: OK
ALL TESTS PASSED
```