


 chendoy / xv6-paging Public[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#)master **xv6-paging / vm.c**[Go to file](#)

...

 Liavbapp Co-authored-by: chendoy <chendoy@post.bgu.ac.il>Latest commit 2661877 on Jun 11, 2020 [History](#) 2 contributors  

1347 lines (1142 sloc) | 32.5 KB

[Raw](#)[Blame](#)

```
1  #include "param.h"
2  #include "types.h"
3  #include "defs.h"
4  #include "x86.h"
5  #include "memlayout.h"
6  #include "mmu.h"
7  #include "proc.h"
8  #include "elf.h"
9  #include "stat.h"
10
11 extern char data[]; // defined by kernel.ld
12 pde_t *kpgdir; // for use in scheduler()
13 static char buffer[PGSIZE];
14
15 void handle_cow_pagefault(struct proc*, pte_t*, char*);
16 void handle_pagedout(struct proc*, char*, pte_t*);
17
18 void printlist()
19 {
20     cprintf("printing list:\n");
21     struct fbblock *curr = myproc()->free_head;
22     for(int i = 0; i < MAX_PSYC_PAGES; i++)
23     {
24         cprintf("%d -> ", curr->off);
25         curr = curr->next;
26         if(curr == 0)
27             break;
28     }
29     cprintf("\n");
30 }
31
32 void printaq()
33 {
34     cprintf("\n\n\nprinting aq:\n");
35     cprintf("head: %d, tail: %d\n", myproc()->queue_head->page_index, myproc()->queue_tail->page_index);
36     if(myproc()->queue_head->prev == 0)
37         cprintf("null <-> ");
38     struct queue_node *curr = myproc()->queue_head;
39     while(curr != 0)
40     {
41         cprintf("%d <-> ", curr->page_index);
42         curr = curr->next;
43     }
44     if(myproc()->queue_tail->next == 0)
45         cprintf("null <-> ");
46     cprintf("\n");
47 }
48
49 // Set up CPU's kernel segment descriptors.
50 // Run once on entry on each CPU.
51 void
52 seginit(void)
53 {
54     struct cpu *c;
```

```

57 // Map "logical" addresses to virtual addresses using identity map.
58 // Cannot share a CODE descriptor for both kernel and user
59 // because it would have to have DPL_USR, but the CPU forbids
60 // an interrupt from CPL=0 to DPL=3.
61 c = &cpus[cpuid()];
62 c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
63 c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
64 c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
65 c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
66 lgdt(c->gdt, sizeof(c->gdt));
67 }
68
69 // Return the address of the PTE in page table pgdir
70 // that corresponds to virtual address va. If alloc!=0,
71 // create any required page table pages.
72 static pte_t *
73 walkpgdir(pte_t *pgdir, const void *va, int alloc)
74 {
75     pte_t *pde;
76     pte_t *pgtab;
77     pde = &pgdir[PDX(va)];
78     if(*pde & PTE_P){
79         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
80     } else {
81         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
82             return 0;
83
84         // Make sure all those PTE_P bits are zero.
85         memset(pgtab, 0, PGSIZE);
86         // The permissions here are overly generous, but they can
87         // be further restricted by the permissions in the page table
88         // entries, if necessary.
89         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
90     }
91     return &pgtab[PTX(va)];
92 }
93
94 // Create PTEs for virtual addresses starting at va that refer to
95 // physical addresses starting at pa. va and size might not
96 // be page-aligned.
97 static int
98 mappages(pte_t *pgdir, void *va, uint size, uint pa, int perm)
99 {
100     char *a, *last;
101     pte_t *pte;
102
103     a = (char*)PGROUNDDOWN((uint)va);
104     last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
105     for(;;){
106         if((pte = walkpgdir(pgdir, a, 1)) == 0)
107             return -1;
108         if(*pte & PTE_P)
109             panic("remap");
110         *pte = pa | perm | PTE_P;
111         if(a == last)
112             break;
113         a += PGSIZE;
114         pa += PGSIZE;
115     }
116     return 0;
117 }
118
119 // There is one page table per process, plus one that's used when
120 // a CPU is not running any process (kpgdir). The kernel uses the
121 // current process's page table during system calls and interrupts;
122 // page protection bits prevent user code from using the kernel's
123 // mappings.
124 //
125 // setupkvm() and exec() set up every page table like this:
126 //
127 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
128 //                   phys memory allocated by the kernel
129 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
130 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)

```

```

131 //          for the kernel's instructions and r/o data
132 //  data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
133 //          rw data + free physical memory
134 //  0xfe000000..0: mapped direct (devices such as ioapic)
135 //
136 // The kernel allocates physical memory for its heap and for user memory
137 // between V2P(end) and the end of physical memory (PHYSTOP)
138 // (directly addressable from end..P2V(PHYSTOP)).
139
140 // This table defines the kernel's mappings, which are present in
141 // every process's page table.
142 static struct kmap {
143     void *virt;
144     uint phys_start;
145     uint phys_end;
146     int perm;
147 } kmap[] = {
148     { (void*)KERNBASE, 0,          EXTMEM,  PTE_W}, // I/O space
149     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
150     { (void*)data,     V2P(data),    PHYSTOP,  PTE_W}, // kern data+memory
151     { (void*)DEVSPACE, DEVSPACE,     0,       PTE_W}, // more devices
152 };
153
154 // Set up kernel part of a page table.
155 pde_t*
156 setupkvm(void)
157 {
158     pde_t *pgdir;
159     struct kmap *k;
160
161     if((pgdir = (pde_t*)kalloc()) == 0)
162     {
163         return 0;
164     }
165     memset(pgdir, 0, PGSIZE);
166     if (P2V(PHYSTOP) > (void*)DEVSPACE)
167         panic("PHYSTOP too high");
168     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
169         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
170             (uint)k->phys_start, k->perm) < 0) {
171             cprintf("mappages failed on setupkvm");
172             freevm(pgdir);
173             return 0;
174         }
175     return pgdir;
176 }
177
178 // Allocate one page table for the machine for the kernel address
179 // space for scheduler processes.
180 void
181 kvmalloc(void)
182 {
183     kpgdir = setupkvm();
184     switchkvm();
185 }
186
187 // Switch h/w page table register to the kernel-only page table,
188 // for when no process is running.
189 void
190 switchkvm(void)
191 {
192     lcr3(V2P(kpgdir)); // switch to the kernel page table
193 }
194
195 // Switch TSS and h/w page table to correspond to process p.
196 void
197 switchvm(struct proc *p)
198 {
199     if(p == 0)
200         panic("switchvm: no process");
201     if(p->kstack == 0)
202         panic("switchvm: no kstack");
203     if(p->pgdir == 0)
204         panic("switchvm: no pgdir");

```

```

205
206     pushcli();
207     mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
208                                   sizeof(mycpu()->ts)-1, 0);
209     mycpu()->gdt[SEG_TSS].s = 0;
210     mycpu()->ts.ss0 = SEG_KDATA << 3;
211     mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
212     // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
213     // forbids I/O instructions (e.g., inb and outb) from user space
214     mycpu()->ts.iomb = (ushort) 0xFFFF;
215     ltr(SEG_TSS << 3);
216     lcr3(V2P(p->pgdir)); // switch to process's address space
217     popcli();
218 }
219
220 // Load the initcode into address 0 of pgdir.
221 // sz must be less than a page.
222 void
223 inituvm(pde_t *pgdir, char *init, uint sz)
224 {
225     char *mem;
226
227     if(sz >= PGSIZE)
228         panic("inituvm: more than a page");
229     mem = kalloc();
230     memset(mem, 0, PGSIZE);
231     mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
232     memmove(mem, init, sz);
233 }
234
235 // Load a program segment into pgdir. addr must be page-aligned
236 // and the pages from addr to addr+sz must already be mapped.
237 int
238 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
239 {
240     uint i, pa, n;
241     pte_t *pte;
242
243     if((uint) addr % PGSIZE != 0)
244         panic("loaduvm: addr must be page aligned");
245     for(i = 0; i < sz; i += PGSIZE){
246         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
247             panic("loaduvm: address should exist");
248         pa = PTE_ADDR(*pte);
249         if(sz - i < PGSIZE)
250             n = sz - i;
251         else
252             n = PGSIZE;
253         if(readi(ip, P2V(pa), offset+i, n) != n)
254             return -1;
255     }
256     return 0;
257 }
258
259 // Allocate page tables and physical memory to grow process from oldsz to
260 // newsz, which need not be page aligned. Returns new size or 0 on error.
261 int
262 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
263 {
264
265     char *mem;
266     uint a;
267
268     struct proc* curproc = myproc();
269
270
271     // cprintf("num swap file %d\n", myproc()->num_swap);
272     if(newsz >= KERNBASE)
273         return 0;
274     if(newsz < oldsz)
275         return oldsz;
276
277     a = PGROUNDUP(oldsz);
278

```

```

279     for(; a < newsz; a += PGSIZE){
280         mem = kalloc();
281         if(mem == 0){
282             cprintf("allocvm out of memory\n");
283             deallocvm(pgdir, newsz, oldsz);
284             return 0;
285         }
286         memset(mem, 0, PGSIZE);
287
288         if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
289             cprintf("allocvm out of memory (2)\n");
290             deallocvm(pgdir, newsz, oldsz);
291             kfree(mem);
292             return 0;
293         }
294
295
296         if(curproc->pid > 2)
297         {
298             allocvm_paging(curproc, pgdir, (char *)a);
299         }
300
301     }
302     return newsz;
303 }
304
305 void
306 allocvm_paging(struct proc * curproc, pde_t *pgdir, char* rounded_virtaddr)
307 {
308     #if SELECTION == NONE
309         allocvm_noswap(curproc, pgdir, rounded_virtaddr);
310     #else
311
312         if(curproc->num_ram < MAX_PSYC_PAGES) // there is space in RAM
313         {
314             allocvm_noswap(curproc, pgdir, rounded_virtaddr);
315         }
316
317         else // no space in RAM for this new page, will swap
318         {
319             allocvm_withswap(curproc, pgdir, rounded_virtaddr);
320         }
321     }
322     #endif
323 }
324
325
326
327 void allocvm_noswap(struct proc* curproc, pde_t *pgdir, char* rounded_virtaddr)
328 {
329     // cprintf("allocvm, not init or shell, there is space in RAM\n");
330
331     struct page *page = &curproc->ramPages[curproc->num_ram];
332
333     page->isused = 1;
334     page->pgdir = pgdir;
335     page->swap_offset = -1;
336     page->virt_addr = rounded_virtaddr;
337
338     update_selectionfiled_allocvm(curproc, page, curproc->num_ram);
339
340     // cprintf("filling ram slot: %d\n", curproc->num_ram);
341     // cprintf("allocating addr : %p\n\n", rounded_virtaddr);
342
343     curproc->num_ram++;
344 }
345
346
347
348
349 void
350 allocvm_withswap(struct proc* curproc, pde_t *pgdir, char* rounded_virtaddr)
351 {
352

```

```

353     if(curproc->num_swap >= MAX_PSYC_PAGES)
354         panic("page limit exceeded");
355
356     // get info of the page to be evicted
357     uint evicted_ind = indexToEvict();
358     // cprintf("[allocvm] index to evict: %d\n",evicted_ind);
359     struct page *evicted_page = &curproc->ramPages[evicted_ind];
360     int swap_offset = curproc->free_head->off;
361
362     if(curproc->free_head->next == 0)
363     {
364         curproc->free_tail = 0;
365         // kfree((char*)curproc->free_head);
366         curproc->free_head = 0;
367     }
368     else
369     {
370         curproc->free_head = curproc->free_head->next;
371         // kfree((char*)curproc->free_head->prev);
372     }
373
374     cprintf("writing a page to swap\n");
375     if(writeToSwapFile(curproc, evicted_page->virt_addr, swap_offset, PGSIZE) < 0)
376         panic("allocvm: writeToSwapFile");
377
378
379     curproc->swappedPages[curproc->num_swap].isused = 1;
380     curproc->swappedPages[curproc->num_swap].virt_addr = curproc->ramPages[evicted_ind].virt_addr;
381     curproc->swappedPages[curproc->num_swap].pgdir = curproc->ramPages[evicted_ind].pgdir;
382     curproc->swappedPages[curproc->num_swap].swap_offset = swap_offset;
383     // cprintf("num swap: %d\n", curproc->num_swap);
384     lcr3(V2P(curproc->swappedPages[curproc->num_swap].pgdir)); // flush TLB
385     curproc->num_swap ++;
386
387
388     pte_t *evicted_pte = walkpgdir(curproc->ramPages[evicted_ind].pgdir, (void*)curproc->ramPages[evicted_ind].virt_addr, 0);
389
390
391
392     if(!(*evicted_pte & PTE_P))
393         panic("allocvm: swap: ram page not present");
394
395     char *evicted_pa = (char*)PTE_ADDR(*evicted_pte);
396
397     if(getRefs(P2V(evicted_pa)) == 1)
398     {
399         kfree(P2V(evicted_pa));
400     }
401     else
402     {
403         refDec(P2V(evicted_pa));
404     }
405
406
407
408
409     *evicted_pte &= 0xFFF;
410
411     *evicted_pte |= PTE_PG;
412     *evicted_pte &= ~PTE_P;
413
414
415     struct page *newpage = &curproc->ramPages[evicted_ind];
416     newpage->isused = 1;
417     newpage->pgdir = pgdir;
418     newpage->swap_offset = -1;
419     newpage->virt_addr = rounded_virtaddr;
420     update_selectionfiled_allocvm(curproc, newpage, evicted_ind);
421
422 }
423
424 void
425 update_selectionfiled_allocvm(struct proc* curproc, struct page* page, int page_ramindex)
426 {

```

```

427
428     #if SELECTION == NFUA
429         page->nfua_counter = 0xFFFFFFFF; // initial with '1's for debugging
430     #endif
431
432     #if SELECTION == LAPA
433         page->lapa_counter = 0xFFFFFFFF;
434     #endif
435
436     #if SELECTION == AQ
437         struct queue_node * node = (struct queue_node*)kalloc();
438         node->page_index = page_ramindex;
439         // cprintf("page ram index is: %d\n", page_ramindex);
440         if(curproc->queue_head == 0 && curproc->queue_tail == 0) //the first queue_node
441         {
442             curproc-> queue_head = node;
443             curproc-> queue_tail = node;
444             curproc-> queue_head->next = 0;
445             curproc-> queue_head->prev = 0;
446             curproc-> queue_head->next = 0;
447             curproc-> queue_head->prev = 0;
448         }
449         else
450         {
451             curproc->queue_head->prev = node;
452             node->next = curproc->queue_head;
453             curproc->queue_head = node;
454             curproc->queue_head->prev = 0;
455         }
456     #endif
457
458 }
459
460
461
462
463 // Deallocate user pages to bring the process size from oldsz to
464 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
465 // need to be less than oldsz.  oldsz can be larger than the actual
466 // process size.  Returns the new process size.
467 int
468 deallocvm(pde_t *pgdir, uint oldsz, uint newsz)
469 {
470     // struct proc *curproc = myproc();
471     pte_t *pte;
472     uint a, pa;
473
474     struct proc* curproc = myproc();
475
476
477     if(newsz >= oldsz)
478         return oldsz;
479
480     a = PGROUNDUP(newsz);
481
482     for(; a < oldsz; a += PGSIZE){
483         pte = walkpgdir(pgdir, (char*)a, 0);
484         if(!pte)
485         {
486             a += (NPENTRIES - 1) * PGSIZE;
487         }
488         else if((*pte & PTE_P) != 0)
489         {
490             pa = PTE_ADDR(*pte);
491             if(pa == 0)
492                 panic("kfree");
493             char *v = P2V(pa);
494
495             if(getRefs(v) == 1)
496             {
497                 kfree(v);
498             }
499             else
500             {

```

```

501     refDec(v);
502 }
503
504
505 if(curproc->pid > 2)
506 {
507     // remove page a from current proc RAM pages and swap pages
508     int i;
509     for(i = 0; i < MAX_PSYC_PAGES; i++)
510     {
511         struct page p_ram = curproc->ramPages[i];
512         struct page p_swap = curproc->swappedPages[i];
513         if((uint)p_ram.virt_addr == a && p_ram.pgdir == pgdir)
514         {
515             memset((void*)&p_ram, 0, sizeof(struct page)); // zero that page struct
516             curproc->num_ram -- ;
517         }
518
519         if((uint)p_swap.virt_addr == a && p_swap.pgdir == pgdir)
520         {
521             memset((void*)&p_swap, 0, sizeof(struct page)); // zero that page struct
522             curproc->num_swap -- ;
523         }
524     }
525
526 }
527 *pte = 0;
528 }
529 }
530 return newsz;
531 }
532
533 // Free a page table and all the physical memory pages
534 // in the user part.
535 void
536 freevm(pde_t *pgdir)
537 {
538     uint i;
539
540     if(pgdir == 0)
541         panic("freevm: no pgdir");
542     deallocvm(pgdir, KERNBASE, 0); // panic: kfree
543     for(i = 0; i < NPENTRIES; i++){
544         if(pgdir[i] & PTE_P){
545             char * v = P2V(PTE_ADDR(pgdir[i]));
546             if(getRefs(v) == 1)
547             {
548                 kfree(v);
549             }
550             else
551             {
552                 refDec(v);
553             }
554         }
555     }
556     kfree((char*)pgdir);
557 }
558
559 // Clear PTE_U on a page. Used to create an inaccessible
560 // page beneath the user stack.
561 void
562 clearpteu(pde_t *pgdir, char *uva)
563 {
564     pte_t *pte;
565
566     pte = walkpgdir(pgdir, uva, 0);
567     if(pte == 0)
568         panic("clearpteu");
569     *pte &= ~PTE_U;
570 }
571
572 // Given a parent process's page table, create a copy
573 // of it for a child.
574

```



```

575
576 pde_t*
577 cowuvm(pde_t *pgdir, uint sz)
578 {
579     pde_t *d;
580     pte_t *pte;
581     uint pa, i, flags;
582
583     if((d = setupkvm()) == 0)
584         return 0;
585
586     for(i = 0; i < sz; i += PGSIZE)
587     {
588         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
589             panic("cowuvm: no pte");
590
591
592         if(!(*pte & PTE_P) && !(*pte & PTE_PG))
593             panic("cowuvm: page not present and not page faulted!");
594
595         if(*pte & PTE_PG) //there is pgfault, then not mark this entry as cow
596         {
597             cprintf("cowuvm, not marked as cow because pgfault \n");
598             pte = walkpgdir(d, (void*) i, 1);
599             *pte = PTE_U | PTE_W | PTE_PG;
600             continue;
601         }
602
603
604         *pte |= PTE_COW;
605         *pte &= ~PTE_W;
606
607         pa = PTE_ADDR(*pte);
608         flags = PTE_FLAGS(*pte);
609         if(mappages(d, (void *) i, PGSIZE, pa, flags) < 0)
610             goto bad;
611
612         char *virt_addr = P2V(pa);
613         refInc(virt_addr);
614
615         // lcr3(V2P(pgdir));
616         invlpg((void*)i); // flush TLB
617     }
618     lcr3(V2P(pgdir));
619     return d;
620
621 bad:
622     cprintf("bad: cowuvm\n");
623     freevm(d);
624     lcr3(V2P(pgdir)); // flush tlb
625     return 0;
626 }
627
628 int
629 getSwappedPageIndex(char* va)
630 {
631     struct proc* curproc = myproc();
632     int i;
633     for(i = 0; i < MAX_PSYC_PAGES; i++)
634     {
635         if(curproc->swappedPages[i].virt_addr == va)
636             return i;
637     }
638     return -1;
639 }
640
641 void
642 pagefault(void)
643 {
644     // cprintf("*** PAGEFAULT ***\n");
645     struct proc* curproc = myproc();
646     pte_t *pte;
647     uint va = rcr2(); // the address retrieved from the cr2 register, contains pagefault addr
648

```

```

649     curproc->totalPgfltCount++;
650     char *start_page = (char*)PGROUNDDOWN((uint)va); //round the va to closet 2 exponenet, to get the start of the page addr
651     pte = walkpgdir(curproc->pgdir, start_page, 0);
652
653     if((*pte & PTE_PG) && !(*pte & PTE_COW)) // paged out, not COW todo
654     {
655         handle_pagedout(curproc, start_page, pte);
656     }
657     else
658     {
659         // cprintf("pagefault - %s (pid %d) - maybe COW\n", curproc->name, curproc->pid);
660         // we should now do COW mechanism for kernel addresses
661         if(va >= KERNBASE || pte == 0)
662         {
663             cprintf("Page fault: pid %d (%s) accesses invalid address.\n", curproc->pid, curproc->name);
664             curproc->killed = 1;
665             return;
666         }
667
668         // if((pte = walkpgdir(curproc->pgdir, (void*)stra, 0)) == 0)
669         // {
670         //     panic("pagefult (cow): pte is 0");
671         // }
672
673         handle_cow_pagefault(curproc, pte, start_page);
674     }
675 }
676
677 void
678 handle_cow_pagefault(struct proc * curproc, pte_t* pte, char* va)
679 {
680     uint err = curproc->tf->err;
681     uint flags;
682     char* new_page;
683     uint pa, new_pa;
684
685     // checking that page fault caused by write
686     if(err & FEC_WR) // a cow pagefault is a write fault
687     {
688         // if the page of this address not includes the PTE_COW flag, kill the process
689         if(!(*pte & PTE_COW))
690         {
691             curproc->killed = 1;
692             return;
693         }
694         else // at this point: FEC_WR & PTE_COW are ON
695         {
696             int ref_count;
697             pa = PTE_ADDR(*pte);
698             char *virt_addr = P2V(pa);
699             flags = PTE_FLAGS(*pte);
700
701             // get how much processes share this page (i.e referece count)
702             ref_count = getRefs(virt_addr);
703
704             if (ref_count > 1) // more than one reference
705             {
706
707                 new_page = kalloc();
708                 //curproc->nummemorypages++;
709                 memmove(new_page, virt_addr, PGSIZE); // copy the faulty page to the newly allocated one
710                 new_pa = V2P(new_page);
711                 *pte = new_pa | flags | PTE_P | PTE_W; // make pte point to new page, turning the required bits ON
712                 invlpg((void*)va); // refresh TLB
713                 refDec(virt_addr); // decrement old page's ref count
714             }
715             else // ref_count = 1
716             {
717                 *pte |= PTE_W; // make it writeable
718                 *pte &= ~PTE_COW; // turn COW off
719                 invlpg((void *)va); // refresh TLB
720             }
721         }
722     }

```

```

723
724     else // pagefault is not write fault
725     {
726         curproc->killed = 1;
727         return;
728     }
729 }
730
731 void
732 handle_pagedout(struct proc* curproc, char* start_page, pte_t* pte)
733 {
734     char* new_page;
735     void* ramPa;
736     cprintf("pagefault - %s (pid %d) - page was paged out\n", curproc->name, curproc->pid);
737
738     new_page = kalloc();
739     *pte |= PTE_P | PTE_W | PTE_U;
740     *pte &= ~PTE_PG;
741     *pte &= 0xFFF;
742     *pte |= V2P(new_page);
743
744     int index = getSwappedPageIndex(start_page); // get swap page index
745     struct page *swap_page = &curproc->swappedPages[index];
746
747
748
749     if(readFromSwapFile(curproc, buffer, swap_page->swap_offset, PGSIZE) < 0)
750         panic("allocuvmm: readFromSwapFile1");
751
752     struct fblock *new_block = (struct fblock*)kalloc();
753     new_block->off = swap_page->swap_offset;
754     new_block->next = 0;
755     new_block->prev = curproc->free_tail;
756
757     if(curproc->free_tail != 0)
758         curproc->free_tail->next = new_block;
759     else
760         curproc->free_head = new_block;
761
762     curproc->free_tail = new_block;
763
764     // cprintf("free blocks list after readFromSwapFile:\n");
765     // printlist();
766
767     memmove((void*)start_page, buffer, PGSIZE);
768
769     // zero swap page entry
770     memset((void*)swap_page, 0, sizeof(struct page));
771
772     if(curproc->num_ram < MAX_PSYC_PAGES) // there is sapce in proc RAM
773     {
774         // cprintf("there is space in RAM\n");
775         int new_idx = getNextFreeRamIndex();
776
777         cprintf("filling ram slot: %d\n", new_idx);
778
779         curproc->ramPages[new_idx].virt_addr = start_page;
780         curproc->ramPages[new_idx].isused = 1;
781         curproc->ramPages[new_idx].pgdir = curproc->pgdir;
782         curproc->ramPages[new_idx].swap_offset = -1; //change the swap offset by the new index
783
784         update_selectionfiled_pagefault(curproc, &curproc->ramPages[new_idx], new_idx);
785
786         curproc->num_ram++;
787         curproc->num_swap--;
788     }
789     else // no sapce in proc RAM, will swap
790     {
791         int index_to_evict = indexToEvict();
792         // cprintf("[pagefault] index to evict: %d\n", index_to_evict);
793         struct page *ram_page = &curproc->ramPages[index_to_evict];
794         int swap_offset = curproc->free_head->off;
795
796         if(curproc->free_head->next == 0)

```

```

797     {
798         curproc->free_tail = 0;
799         // kfree((char*)curproc->free_head);
800         curproc->free_head = 0;
801     }
802     else
803     {
804         curproc->free_head = curproc->free_head->next;
805         // kfree((char*)curproc->free_head->prev);
806     }
807
808     if(writeToSwapFile(curproc, (char*)ram_page->virt_addr, swap_offset, PGSIZE) < 0) // buffer now has bytes from swapped
809         panic("allocvm: writeToSwapFile");
810
811     swap_page->virt_addr = ram_page->virt_addr;
812     swap_page->pgdir = ram_page->pgdir;
813     swap_page->isused = 1;
814     swap_page->swap_offset = swap_offset;
815
816     // get pte of RAM page
817     pte = walkpgdir(curproc->pgdir, (void*)ram_page->virt_addr, 0);
818     if(!(*pte & PTE_P))
819         panic("pagefault: ram page is not present");
820     ramPa = (void*)PTE_ADDR(*pte);
821
822
823     if(getRefs(P2V(ramPa)) == 1)
824     {
825         kfree(P2V(ramPa));
826     }
827     else
828     {
829         refDec(P2V(ramPa));
830     }
831
832     *pte &= 0xFFF; // ???
833
834     // prepare to-be-swapped page in RAM to move to swap file
835     *pte |= PTE_PG; // turn "paged-out" flag on
836     *pte &= ~PTE_P; // turn "present" flag off
837
838     ram_page->virt_addr = start_page;
839     update_selectionfiled_pagefault(curproc, ram_page, index_to_evict);
840
841     lcr3(V2P(curproc->pgdir)); // refresh TLB
842 }
843 return;
844 }
845
846
847 void
848 update_selectionfiled_pagefault(struct proc* curproc, struct page* page, int page_ramindex)
849 {
850     #if SELECTION == NFUA
851         page->nfua_counter = 0xFFFFFFFF;
852     #endif
853
854     #if SELECTION == LAPA
855         page->lapa_counter = 0xFFFFFFFF;
856     #endif
857
858     #if SELECTION == AQ
859         struct queue_node * node = (struct queue_node*)kalloc();
860         node->page_index = page_ramindex;
861         if(curproc->queue_head == 0 && curproc->queue_tail == 0) //the first queue_node
862         {
863             curproc-> queue_head = node;
864             curproc-> queue_tail = node;
865             curproc-> queue_head->next = 0;
866             curproc-> queue_head->prev = 0;
867             curproc-> queue_head->next = 0;
868             curproc-> queue_head->prev = 0;
869         }
870     #else

```

```

871     {
872         curproc->queue_head->prev = node;
873         node->next = curproc->queue_head;
874         curproc->queue_head = node;
875         curproc->queue_head->prev = 0;
876     }
877 #endif
878
879
880
881 }
882
883 pde_t*
884 copyuvm(pde_t *pgdir, uint sz)
885 {
886     pde_t *d;
887     pte_t *pte;
888     uint pa, i, flags;
889     char *mem;
890
891     #if SELECTION != NONE
892     if((d = setupkvm()) == 0)
893         return 0;
894     for(i = 0; i < sz; i += PGSIZE){
895         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
896             panic("copyuvm: pte should exist");
897         if(!(*pte & PTE_P) && !(*pte & PTE_PG))
898             panic("copyuvm: page not present and also not paged out to disk");
899
900         if (*pte & PTE_PG) {
901             pte = walkpgdir(d, (void*) i, 1);
902             *pte = PTE_U | PTE_W | PTE_PG;
903             continue;
904         }
905
906         pa = PTE_ADDR(*pte);
907         flags = PTE_FLAGS(*pte);
908         // if(*pte & PTE_PG)
909         // {
910         //     if(mappages(d, (void*)i, PGSIZE, 0, flags) < 0)
911         //         panic("copyuvm: mappages failed");
912         //     continue;
913         // }
914         if((mem = kalloc()) == 0)
915             goto bad;
916         memmove(mem, (char*)P2V(pa), PGSIZE);
917         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
918             cprintf("copyuvm: mappages failed\n");
919             goto bad;
920         }
921     }
922 #else
923     if((d = setupkvm()) == 0)
924         return 0;
925     for(i = 0; i < sz; i += PGSIZE){
926         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
927             panic("copyuvm: pte should exist");
928         if(!(*pte & PTE_P))
929             panic("copyuvm: page not present");
930         pa = PTE_ADDR(*pte);
931         flags = PTE_FLAGS(*pte);
932         if((mem = kalloc()) == 0)
933             goto bad;
934         memmove(mem, (char*)P2V(pa), PGSIZE);
935         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
936             kfree(mem);
937             goto bad;
938         }
939     }
940 #endif
941     return d;
942
943 bad:
944     cprintf("bad: copyuvm\n");

```

```

945     freevm(d);
946     return 0;
947 }
948
949 //PAGEBREAK!
950 // Map user virtual address to kernel address.
951 char*
952 uva2ka(pde_t *pgdir, char *uva)
953 {
954     pte_t *pte;
955
956     pte = walkpgdir(pgdir, uva, 0);
957     if((*pte & PTE_P) == 0)
958         return 0;
959     if((*pte & PTE_U) == 0)
960         return 0;
961     return (char*)P2V(PTE_ADDR(*pte));
962 }
963
964 // Copy len bytes from p to user address va in page table pgdir.
965 // Most useful when pgdir is not the current page table.
966 // uva2ka ensures this only works for PTE_U pages.
967 int
968 copyout(pde_t *pgdir, uint va, void *p, uint len)
969 {
970     char *buf, *pa0;
971     uint n, va0;
972
973     buf = (char*)p;
974     while(len > 0){
975         va0 = (uint)PGROUNDDOWN(va);
976         pa0 = uva2ka(pgdir, (char*)va0);
977         if(pa0 == 0)
978             return -1;
979         n = PGSIZE - (va - va0);
980         if(n > len)
981             n = len;
982         memmove(pa0 + (va - va0), buf, n);
983         len -= n;
984         buf += n;
985         va = va0 + PGSIZE;
986     }
987     return 0;
988 }
989
990 int
991 getNextFreeRamIndex()
992 {
993     int i;
994     struct proc * currproc = myproc();
995     for(i = 0; i < MAX_PSYC_PAGES ; i++)
996     {
997         if(((struct page)currproc->ramPages[i]).isused == 0)
998             return i;
999     }
1000     return -1;
1001 }
1002
1003 //PAGEBREAK!
1004 // Blank page.
1005 //PAGEBREAK!
1006 // Blank page.
1007 //PAGEBREAK!
1008 // Blank page.
1009
1010 void updateLapa(struct proc* p)
1011 {
1012     struct page *ramPages = p->ramPages;
1013     int i;
1014     pte_t *pte;
1015     for(i = 0; i < MAX_PSYC_PAGES; i++)
1016     {
1017         struct page *cur_page = &ramPages[i];

```

```

1019     if(!cur_page->isused)
1020         continue;
1021     if((pte = walkpgdir(cur_page->pgdir, cur_page->virt_addr, 0)) == 0)
1022         panic("updateLapa: no pte");
1023     if(*pte & PTE_A) // if accessed
1024     {
1025         cur_page->lapa_counter = cur_page->lapa_counter >> 1; // shift right one bit
1026         cur_page->lapa_counter |= 1 << 31; // turn on MSB
1027         *pte &= ~PTE_A;
1028     }
1029     else
1030     {
1031         cur_page->lapa_counter = cur_page->lapa_counter >> 1; // just shit right one bit
1032     }
1033 }
1034 }
1035
1036 void updateNfua(struct proc* p)
1037 {
1038     struct page *ramPages = p->ramPages;
1039     int i;
1040     pte_t *pte;
1041     for(i = 0; i < MAX_PSYC_PAGES; i++)
1042     {
1043         struct page *cur_page = &ramPages[i];
1044         if(!cur_page->isused)
1045             continue;
1046         if((pte = walkpgdir(cur_page->pgdir, cur_page->virt_addr, 0)) == 0)
1047             panic("updateNfua: no pte");
1048         if(*pte & PTE_A) // if accessed
1049         {
1050             cur_page->nfua_counter = cur_page->nfua_counter >> 1; // shift right one bit
1051             cur_page->nfua_counter |= 0x80000000; // turn on MSB
1052             *pte &= ~PTE_A;
1053         }
1054         else
1055         {
1056             cur_page->nfua_counter = cur_page->nfua_counter >> 1; // just shit right one bit
1057         }
1058     }
1059 }
1060
1061 uint indexToEvict()
1062 {
1063     #if SELECTION==DUMMY
1064         return 3;
1065     #endif
1066     #if SELECTION==SCFIFO
1067         return scfifo();
1068     #endif
1069     #if SELECTION==NFUA
1070         return nfua();
1071     #endif
1072     #if SELECTION==LAPA
1073         return lapa();
1074     #endif
1075     #if SELECTION==AQ
1076         return aq();
1077     #else
1078         return 11; // default
1079     #endif
1080 }
1081
1082 uint aq()
1083 {
1084     struct proc* curproc = myproc();
1085     int res = curproc->queue_tail->page_index;
1086     struct queue_node* new_tail;
1087     if(curproc->queue_tail == 0 || curproc->queue_head == 0)
1088     {
1089         panic("AQ INDEX SELECTION: empty queue cann't make index selection!");
1090     }
1091
1092     if(curproc->queue_tail == curproc->queue_head)

```

```

1093 {
1094     curproc->queue_head=0;
1095     new_tail = 0;
1096 }
1097 else
1098 {
1099     curproc->queue_tail->prev->next = 0;
1100     new_tail = curproc->queue_tail->prev;
1101 }
1102
1103 // kfree((char*)curproc->queue_tail);
1104 curproc->queue_tail = new_tail;
1105
1106 return res;
1107
1108
1109 }
1110 uint lapa()
1111 {
1112     struct proc *curproc = myproc();
1113     struct page *ramPages = curproc->ramPages;
1114     /* find the page with the smallest number of '1's */
1115     int i;
1116     uint minNumOfOnes = countSetBits(ramPages[0].lapa_counter);
1117     uint minloc = 0;
1118     uint instances = 0;
1119
1120     for(i = 1; i < MAX_PSYC_PAGES; i++)
1121     {
1122         // cprintf("i = %d, lapa_counter : %d\n", i, ramPages[i].lapa_counter);
1123         uint numOfOnes = countSetBits(ramPages[i].lapa_counter);
1124         if(numOfOnes < minNumOfOnes)
1125         {
1126             minNumOfOnes = numOfOnes;
1127             minloc = i;
1128             instances = 1;
1129         }
1130         else if(numOfOnes == minNumOfOnes)
1131             instances++;
1132     }
1133     if(instances > 1) // more than one counter with minimal number of 1's
1134     {
1135         uint minvalue = ramPages[minloc].lapa_counter;
1136         for(i = 1; i < MAX_PSYC_PAGES; i++)
1137         {
1138             uint numOfOnes = countSetBits(ramPages[i].lapa_counter);
1139             if(numOfOnes == minNumOfOnes && ramPages[i].lapa_counter < minvalue)
1140             {
1141                 minloc = i;
1142                 minvalue = ramPages[i].lapa_counter;
1143             }
1144         }
1145         return minloc;
1146     }
1147     else
1148         return minloc;
1149 }
1150 uint nfua()
1151 {
1152     struct proc *curproc = myproc();
1153     struct page *ramPages = curproc->ramPages;
1154     /* find the page with the minimal nfua */
1155     int i;
1156     uint minval = ramPages[0].nfua_counter;
1157     uint minloc = 0;
1158
1159     for(i = 1; i < MAX_PSYC_PAGES; i++)
1160     {
1161         // cprintf("i = %d, nufa_counter : %d\n", i, ramPages[i].nfua_counter);
1162         if(ramPages[i].nfua_counter < minval)
1163         {
1164             minval = ramPages[i].nfua_counter;
1165             minloc = i;
1166         }

```



```

1167     }
1168     return minloc;
1169 }
1170
1171 uint scfifo()
1172 {
1173     struct proc* curproc = myproc();
1174     int i;
1175     while(1)
1176     {
1177         for(i = curproc->clockHand ; i < MAX_PSYC_PAGES ; i++)
1178         {
1179             pte_t *pte = walkpgdir(curproc->ramPages[i].pgdir, curproc->ramPages[i].virt_addr, 0);
1180             if(!(*pte & PTE_A)) //ref bit is off
1181             {
1182                 if(curproc->clockHand == MAX_PSYC_PAGES - 1)
1183                 {
1184                     curproc->clockHand = 0;
1185                 }
1186                 else
1187                 {
1188                     curproc->clockHand = i + 1;
1189                 }
1190                 return i;
1191             }
1192             else
1193             {
1194                 // turn off access bit
1195                 *pte &= ~PTE_A;
1196             }
1197         }
1198     }
1199     int j;
1200     for(j=0; j< curproc->clockHand ;j++)
1201     {
1202         pte_t *pte = walkpgdir(curproc->ramPages[j].pgdir, curproc->ramPages[j].virt_addr, 0);
1203         if(!(*pte & PTE_A)) //ref bit is off
1204         {
1205             curproc->clockHand = j + 1;
1206             return j;
1207         }
1208         else
1209         {
1210             // turn off access bit
1211             *pte &= ~PTE_A;
1212         }
1213     }
1214 }
1215
1216 panic("scfifo: not found any index!");
1217 return -1;
1218 }
1219
1220 uint countSetBits(uint n)
1221 {
1222     uint count = 0;
1223     while (n) {
1224         count += n & 1;
1225         n >>= 1;
1226     }
1227     return count;
1228 }
1229
1230
1231 void updateAQ(struct proc* p)
1232 {
1233     struct queue_node *curr_node = p->queue_tail;
1234     struct page *ramPages = p->ramPages;
1235     struct page *curr_page = &ramPages[curr_node->page_index];
1236     struct page *prev_page;
1237     pte_t *pte_curr;
1238     pte_t *pte_prev;
1239
1240     if(p->queue_tail == 0 || p->queue_head == 0)

```

```

1241     return;
1242
1243     if(curr_node->prev == 0)
1244         return;
1245
1246     prev_page = &ramPages[curr_node->prev->page_index];
1247
1248     // cprintf("found page index: %d\n", p->queue_tail->page_index);
1249
1250
1251     while(curr_node != 0)
1252     {
1253         // printaq();
1254         if((pte_curr = walkpgdir(curr_page->pgdir, curr_page->virt_addr, 0)) == 0)
1255             panic("updateAQ: no pte");
1256         if(*pte_curr & PTE_A) // an accessed page
1257         {
1258             if(curr_node->prev != 0) // there is a page behind it
1259             {
1260                 if((pte_prev = walkpgdir(prev_page->pgdir, prev_page->virt_addr, 0)) == 0)
1261                     panic("updateAQ: no pte");
1262
1263                 if(!(*pte_prev & PTE_A)) // was not accessed, will swap
1264                     swapAQ(curr_node);
1265             }
1266             *pte_curr &= ~PTE_A;
1267         }
1268
1269         if(curr_node != 0)
1270         {
1271             curr_page = &ramPages[curr_node->page_index];
1272
1273             if(curr_node->prev != 0)
1274                 prev_page = &ramPages[curr_node->prev->page_index];
1275
1276             curr_node = curr_node->prev;
1277         }
1278     }
1279 }
1280
1281
1282 // will swap curr_node with the node preceding it in the queue.
1283 // assumes there exist a page preceding curr_node.
1284 // queue structure at entry point:
1285 // [maybeLeft?] <-> [prev_node] <-> [curr_node] <-> [maybeRight?]
1286
1287 void swapAQ(struct queue_node *curr_node)
1288 {
1289     struct queue_node *prev_node = curr_node->prev;
1290     struct queue_node *maybeLeft, *maybeRight;
1291
1292     if(curr_node == myproc()->queue_tail)
1293     {
1294         myproc()->queue_tail = prev_node;
1295         myproc()->queue_tail->next = 0;
1296     }
1297
1298     if(prev_node == myproc()->queue_head)
1299     {
1300         myproc()->queue_head = curr_node;
1301         myproc()->queue_head->prev = 0;
1302     }
1303
1304     // saving maybeLeft and maybeRight pointers for later
1305     maybeLeft = prev_node->prev;
1306     maybeRight = curr_node->next;
1307
1308     // re-connecting prev_node and curr_node (simple)
1309     curr_node->next = prev_node;
1310     prev_node->prev = curr_node;
1311
1312     // updating maybeLeft and maybeRight
1313     if(maybeLeft != 0)
1314     {

```

```
1315     curr_node->prev = maybeLeft;
1316     maybeLeft->next = curr_node;
1317 }
1318
1319 if(maybeRight != 0)
1320 {
1321     prev_node->next = maybeRight;
1322     maybeRight->prev = prev_node;
1323 }
1324 }
1325
1326 int
1327 getNumRefswapper(int idx)
1328 {
1329     struct proc * curproc = myproc();
1330     pte_t *evicted_pte = walkpgdir(curproc->ramPages[idx].pgdir, (void*)curproc->ramPages[idx].virt_addr, 0);
1331     char *evicted_pa = (char*)PTE_ADDR(*evicted_pte);
1332     return getRefs(P2V(evicted_pa));
1333 }
1334
1335
1336 int
1337 getRamPageIndexByVirtAddr(char* virtaddr)
1338 {
1339     struct proc* curproc = myproc();
1340     int i;
1341     for(i = 0; i < MAX_PSYC_PAGES; i++)
1342     {
1343         if(curproc->swappedPages[i].virt_addr == virtaddr)
1344             return i;
1345     }
1346     return -1;
1347 }
```