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

```c
#include "param.h"
#include "types.h"
#include "defs.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "elf.h"
#include "stat.h"

extern char data[];  // defined by kernel.ld
pde_t *kpgdir;  // for use in scheduler()
static char buffer[PGSIZE];

void  handle_cow_pagefault(struct proc*, pte_t*, char*);
void  handle_pagedout(struct proc*, char*, pte_t*);

void printlist()
{
  cprintf("printing list:\n");
  struct fblock *curr = myproc()->free_head;
  for(int i = 0; i < MAX_PSYC_PAGES; i++)
  {
    cprintf("%d -> ", curr->off);
    curr = curr->next;
    if(curr == 0)
      break;
  }
  cprintf("\n");
}

void printaq()
{
  cprintf("\n\n\n\nprinting aq:\n");
  cprintf("head: %d, tail: %d\n", myproc()->queue_head->page_index, myproc()->queue_tail->page_index);
  if(myproc()->queue_head->prev == 0)
    cprintf("null <-> ");
  struct queue_node *curr = myproc()->queue_head;
  while(curr != 0)
  {
    cprintf("%d <-> ", curr->page_index);
    curr = curr->next;
  }
  if(myproc()->queue_tail->next == 0)
    cprintf("null <-> ");
  cprintf("\n");
}


// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

```

```c
  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpuid()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
  lgdt(c->gdt, sizeof(c->gdt));
}

// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;
  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;

    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}

// There is one page table per process, plus one that's used when
// a CPU is not running any process (kpgdir). The kernel uses the
// current process's page table during system calls and interrupts;
// page protection bits prevent user code from using the kernel's
// mappings.
//
// setupkvm() and exec() set up every page table like this:
//
//   0..KERNBASE: user memory (text+data+stack+heap), mapped to
//                phys memory allocated by the kernel
//   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
//   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
```

```c
//                for the kernel's instructions and r/o data
//   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
//                                  rw data + free physical memory
//   0xfe000000..0: mapped direct (devices such as ioapic)
//
// The kernel allocates physical memory for its heap and for user memory
// between V2P(end) and the end of physical memory (PHYSTOP)
// (directly addressable from end..P2V(PHYSTOP)).

// This table defines the kernel's mappings, which are present in
// every process's page table.
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};

// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
  {
    return 0;
  }
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      cprintf("mappages failed on setupkvm");
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}

// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}

// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void
switchkvm(void)
{
  lcr3(V2P(kpgdir));   // switch to the kernel page table
}

// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
```

```c

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}

// Load the initcode into address 0 of pgdir.
// sz must be less than a page.
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}

// Load a program segment into pgdir.  addr must be page-aligned
// and the pages from addr to addr+sz must already be mapped.
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}

// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{

  char *mem;
  uint a;

  struct proc* curproc = myproc();


  // cprintf("num swap file %d\n", myproc()->num_swap);
  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);

```

```
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);

    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }


    if(curproc->pid > 2)
    {
        allocuvm_paging(curproc, pgdir, (char *)a);
    }


  }
  return newsz;
}

void
allocuvm_paging(struct proc * curproc, pde_t *pgdir, char* rounded_virtaddr)
{
    #if SELECTION == NONE
     allocuvm_noswap(curproc, pgdir, rounded_virtaddr);
    #else

    if(curproc->num_ram < MAX_PSYC_PAGES) // there is space in RAM
    {
       allocuvm_noswap(curproc, pgdir, rounded_virtaddr);
    }

    else // no space in RAM for this new page, will swap
    {
      allocuvm_withswap(curproc, pgdir, rounded_virtaddr);
    }

    #endif
}



void allocuvm_noswap(struct proc* curproc, pde_t *pgdir, char* rounded_virtaddr)
{
// cprintf("allocuvm, not init or shell, there is space in RAM\n");

  struct page *page = &curproc->ramPages[curproc->num_ram];

  page->isused = 1;
  page->pgdir = pgdir;
  page->swap_offset = -1;
  page->virt_addr = rounded_virtaddr;

  update_selectionfiled_allocuvm(curproc, page, curproc->num_ram);

  // cprintf("filling ram slot: %d\n", curproc->num_ram);
  // cprintf("allocating addr : %p\n\n", rounded_virtaddr);

  curproc->num_ram++;

}



void
allocuvm_withswap(struct proc* curproc, pde_t *pgdir, char* rounded_virtaddr)
{
```

```
   if(curproc-> num_swap >= MAX_PSYC_PAGES)
        panic("page limit exceeded");

      // get info of the page to be evicted
      uint evicted_ind = indexToEvict();
      // cprintf("[allocuvm] index to evict: %d\n",evicted_ind);
      struct page *evicted_page = &curproc->ramPages[evicted_ind];
      int swap_offset = curproc->free_head->off;

      if(curproc->free_head->next == 0)
      {
        curproc->free_tail = 0;
        // kfree((char*)curproc->free_head);
        curproc->free_head = 0;
      }
      else
      {
        curproc->free_head = curproc->free_head->next;
        // kfree((char*)curproc->free_head->prev);
      }

      cprintf("writing a page to swap\n");
      if(writeToSwapFile(curproc, evicted_page->virt_addr, swap_offset, PGSIZE) < 0)
        panic("allocuvm: writeToSwapFile");


      curproc->swappedPages[curproc->num_swap].isused = 1;
      curproc->swappedPages[curproc->num_swap].virt_addr = curproc->ramPages[evicted_ind].virt_addr;
      curproc->swappedPages[curproc->num_swap].pgdir = curproc->ramPages[evicted_ind].pgdir;
      curproc->swappedPages[curproc->num_swap].swap_offset = swap_offset;
      // cprintf("num swap: %d\n", curproc->num_swap);
      lcr3(V2P(curproc->swappedPages[curproc->num_swap].pgdir)); // flush TLB
      curproc->num_swap ++;


      pte_t *evicted_pte = walkpgdir(curproc->ramPages[evicted_ind].pgdir, (void*)curproc->ramPages[evicted_ind].virt_addr, 0);



      if(!(*evicted_pte & PTE_P))
        panic("allocuvm: swap: ram page not present");

      char *evicted_pa = (char*)PTE_ADDR(*evicted_pte);

      if(getRefs(P2V(evicted_pa)) == 1)
      {
        kfree(P2V(evicted_pa));
      }
      else
      {
        refDec(P2V(evicted_pa));
      }




      *evicted_pte &= 0xFFF;

      *evicted_pte |= PTE_PG;
      *evicted_pte &= ~PTE_P;


      struct page *newpage = &curproc->ramPages[evicted_ind];
      newpage->isused = 1;
      newpage->pgdir = pgdir;
      newpage->swap_offset = -1;
      newpage->virt_addr = rounded_virtaddr;
      update_selectionfiled_allocuvm(curproc, newpage, evicted_ind);

}

void
update_selectionfiled_allocuvm(struct proc* curproc, struct page* page, int page_ramindex)
{
```

```c

   #if SELECTION == NFUA
    page->nfua_counter = 0xFFFFFFFF; // initial with '1's for debugging
  #endif

  #if SELECTION == LAPA
    page->lapa_counter =  0xFFFFFFFF;
  #endif

  #if SELECTION == AQ
    struct queue_node * node = (struct queue_node*)kalloc();
    node->page_index = page_ramindex;
    // cprintf("page ram index is: %d\n", page_ramindex);
    if(curproc->queue_head == 0 && curproc->queue_tail ==0)  //the first queue_node
    {
      curproc-> queue_head = node;
      curproc-> queue_tail = node;
      curproc-> queue_head->next = 0;
      curproc-> queue_head->prev = 0;
      curproc-> queue_head->next = 0;
      curproc-> queue_head->prev = 0;
    }
    else
    {
      curproc->queue_head->prev = node;
      node->next = curproc->queue_head;
      curproc->queue_head = node;
      curproc->queue_head->prev = 0;
    }
  #endif


}



// Deallocate user pages to bring the process size from oldsz to
// newsz.  oldsz and newsz need not be page-aligned, nor does newsz
// need to be less than oldsz.  oldsz can be larger than the actual
// process size.  Returns the new process size.
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  // struct proc *curproc = myproc();
  pte_t *pte;
  uint a, pa;

  struct proc* curproc = myproc();


  if(newsz >= oldsz)
    return oldsz;

  a = PGROUNDUP(newsz);

  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
    {
      a += (NPTENTRIES - 1) * PGSIZE;
    }
    else if((*pte & PTE_P) != 0)
    {
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = P2V(pa);

      if(getRefs(v) == 1)
      {
        kfree(v);
      }
      else
      {
```

```
        refDec(v);
      }


      if(curproc->pid >2)
      {
          // remove page a from current proc RAM pages and swap pages
        int i;
        for(i = 0; i < MAX_PSYC_PAGES; i++)
        {
          struct page p_ram = curproc->ramPages[i];
          struct page p_swap = curproc->swappedPages[i];
          if((uint)p_ram.virt_addr == a && p_ram.pgdir == pgdir)
          {
            memset((void*)&p_ram, 0, sizeof(struct page)); // zero that page struct
            curproc->num_ram -- ;
          }

          if((uint)p_swap.virt_addr == a && p_swap.pgdir == pgdir)
          {
            memset((void*)&p_swap, 0, sizeof(struct page)); // zero that page struct
            curproc->num_swap --;
          }
        }

      }
      *pte = 0;
    }
  }
  return newsz;
}

// Free a page table and all the physical memory pages
// in the user part.
void
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0); // panic: kfree
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = P2V(PTE_ADDR(pgdir[i]));
      if(getRefs(v) == 1)
      {
        kfree(v);
      }
      else
      {
        refDec(v);
      }
    }
  }
  kfree((char*)pgdir);
}

// Clear PTE_U on a page. Used to create an inaccessible
// page beneath the user stack.
void
clearpteu(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if(pte == 0)
    panic("clearpteu");
  *pte &= ~PTE_U;
}

// Given a parent process's page table, create a copy
// of it for a child.

```

```c

pde_t*
cowuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;

  if((d = setupkvm()) == 0)
    return 0;

  for(i = 0; i < sz; i += PGSIZE)
  {
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("cowuvm: no pte");


    if(!(*pte & PTE_P) && !(*pte & PTE_PG))
      panic("cowuvm: page not present and not page faulted!");

    if(*pte & PTE_PG)  //there is pgfault, then not mark this entry as cow
    {
      cprintf("cowuvm,  not marked as cow because pgfault \n");
      pte = walkpgdir(d, (void*) i, 1);
      *pte = PTE_U | PTE_W | PTE_PG;
      continue;
    }


    *pte |= PTE_COW;
    *pte &= ~PTE_W;

    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if(mappages(d, (void *) i, PGSIZE, pa, flags) < 0)
      goto bad;

    char *virt_addr = P2V(pa);
    refInc(virt_addr);

    // lcr3(V2P(pgdir));
    invlpg((void*)i); // flush TLB
  }
  lcr3(V2P(pgdir));
  return d;

bad:
  cprintf("bad: cowuvm\n");
  freevm(d);
  lcr3(V2P(pgdir));  // flush tlb
  return 0;
}

int
getSwappedPageIndex(char* va)
{
  struct proc* curproc = myproc();
  int i;
  for(i = 0; i < MAX_PSYC_PAGES; i++)
  {
    if(curproc->swappedPages[i].virt_addr == va)
      return i;
  }
  return -1;
}

void
pagefault(void)
{
  // cprintf("*** PAGEFAULT ***\n");
  struct proc* curproc = myproc();
  pte_t *pte;
  uint va = rcr2(); //  the address retreived from the cr2 register, contains pagefault addr

```

```
  curproc->totalPgfltCount++;
  char *start_page = (char*)PGROUNDDOWN((uint)va); //round the va to closet 2 exponenet, to get the start of the page addr
  pte = walkpgdir(curproc->pgdir, start_page, 0);

  if((*pte & PTE_PG) && !(*pte & PTE_COW)) // paged out, not COW todo
  {
      handle_pagedout(curproc, start_page, pte);
  }
  else
  {
    // cprintf("pagefault - %s (pid %d) - maybe COW\n", curproc->name, curproc->pid);
    // we should now do COW mechanism for kernel addresses
    if(va >= KERNBASE || pte == 0)
    {
      cprintf("Page fault: pid %d (%s) accesses invalid address.\n", curproc->pid, curproc->name);
      curproc->killed = 1;
      return;
    }

    // if((pte = walkpgdir(curproc->pgdir, (void*)stra, 0)) == 0)
    // {
    //   panic("pagefult (cow): pte is 0");
    // }

    handle_cow_pagefault(curproc, pte, start_page);
  }
}

void
handle_cow_pagefault(struct proc * curproc, pte_t* pte, char* va)
{
  uint err = curproc->tf->err;
  uint flags;
  char* new_page;
  uint pa, new_pa;

   // checking that page fault caused by write
  if(err & FEC_WR) // a cow pagefault is a write fault
  {
    // if the page of this address not includes the PTE_COW flag, kill the process
    if(!(*pte & PTE_COW))
    {
      curproc->killed = 1;
      return;
    }
    else // at this point: FEC_WR & PTE_COW are ON
    {
      int ref_count;
      pa = PTE_ADDR(*pte);
      char *virt_addr = P2V(pa);
      flags = PTE_FLAGS(*pte);

      // get how much processes share this page (i.e referece count)
      ref_count = getRefs(virt_addr);

      if (ref_count > 1) // more than one reference
      {

        new_page = kalloc();
        //curproc->nummemorypages++;
        memmove(new_page, virt_addr, PGSIZE); // copy the faulty page to the newly allocated one
        new_pa = V2P(new_page);
        *pte = new_pa | flags | PTE_P | PTE_W; // make pte point to new page, turning the required bits ON
        invlpg((void*)va); // refresh TLB
        refDec(virt_addr); // decrement old page's ref count
      }
      else // ref_count = 1
      {
        *pte |= PTE_W;    // make it writeable
        *pte &= ~PTE_COW; // turn COW off
        invlpg((void *)va); // refresh TLB
      }
    }
  }
```

```c

  else // pagefault is not write fault
  {
    curproc->killed = 1;
    return;
  }
}

void
handle_pagedout(struct proc* curproc, char* start_page, pte_t* pte)
{
    char* new_page;
    void* ramPa;
    cprintf("pagefault - %s (pid %d) - page was paged out\n", curproc->name, curproc->pid);

    new_page = kalloc();
    *pte |= PTE_P | PTE_W | PTE_U;
    *pte &= ~PTE_PG;
    *pte &= 0xFFF;
    *pte |= V2P(new_page);

    int index = getSwappedPageIndex(start_page); // get swap page index
    struct page *swap_page = &curproc->swappedPages[index];



    if(readFromSwapFile(curproc, buffer, swap_page->swap_offset, PGSIZE) < 0)
      panic("allocuvm: readFromSwapFile1");

    struct fblock *new_block = (struct fblock*)kalloc();
    new_block->off = swap_page->swap_offset;
    new_block->next = 0;
    new_block->prev = curproc->free_tail;

    if(curproc->free_tail != 0)
      curproc->free_tail->next = new_block;
    else
      curproc->free_head = new_block;

    curproc->free_tail = new_block;

    // cprintf("free blocks list after readFromSwapFile:\n");
    // printlist();

    memmove((void*)start_page, buffer, PGSIZE);

    // zero swap page entry
    memset((void*)swap_page, 0, sizeof(struct page));

    if(curproc->num_ram < MAX_PSYC_PAGES) // there is sapce in proc RAM
    {
      // cprintf("there is space in RAM\n");
      int new_indx = getNextFreeRamIndex();

      cprintf("filling ram slot: %d\n", new_indx);

      curproc->ramPages[new_indx].virt_addr = start_page;
      curproc->ramPages[new_indx].isused = 1;
      curproc->ramPages[new_indx].pgdir = curproc->pgdir;
      curproc->ramPages[new_indx].swap_offset = -1;//change the swap offset by the new index

      update_selectionfiled_pagefault(curproc, &curproc->ramPages[new_indx], new_indx);

      curproc->num_ram++;
      curproc->num_swap--;
    }
    else // no sapce in proc RAM, will swap
    {
      int index_to_evicet = indexToEvict();
      // cprintf("[pagefault] index to evict: %d\n", index_to_evicet);
      struct page *ram_page = &curproc->ramPages[index_to_evicet];
      int swap_offset = curproc->free_head->off;

      if(curproc->free_head->next == 0)
```

```
      {
        curproc->free_tail = 0;
        // kfree((char*)curproc->free_head);
        curproc->free_head = 0;
      }
      else
      {
        curproc->free_head = curproc->free_head->next;
        // kfree((char*)curproc->free_head->prev);
      }

      if(writeToSwapFile(curproc, (char*)ram_page->virt_addr, swap_offset, PGSIZE) < 0)   // buffer now has bytes from swapped
        panic("allocuvm: writeToSwapFile");

      swap_page->virt_addr = ram_page->virt_addr;
      swap_page->pgdir = ram_page->pgdir;
      swap_page->isused = 1;
      swap_page->swap_offset = swap_offset;

      // get pte of RAM page
      pte = walkpgdir(curproc->pgdir, (void*)ram_page->virt_addr, 0);
      if(!(*pte & PTE_P))
        panic("pagefault: ram page is not present");
      rampa = (void*)PTE_ADDR(*pte);


       if(getRefs(P2V(ramPa)) == 1)
      {
           kfree(P2V(ramPa));
      }
      else
      {
           refDec(P2V(ramPa));
      }

      *pte &= 0xFFF;   // ???

      // prepare to-be-swapped page in RAM to move to swap file
      *pte |= PTE_PG;     // turn "paged-out" flag on
      *pte &= ~PTE_P;     // turn "present" flag off

      ram_page->virt_addr = start_page;
      update_selectionfiled_pagefault(curproc, ram_page, index_to_evicet);

      lcr3(V2P(curproc->pgdir));             // refresh TLB
    }
    return;
}


void
update_selectionfiled_pagefault(struct proc* curproc, struct page* page, int page_ramindex)
{
  #if SELECTION == NFUA
    page->nfua_counter = 0xFFFFFFFF;
  #endif

  #if SELECTION == LAPA
    page->lapa_counter =  0xFFFFFFFF;
  #endif

  #if SELECTION == AQ
    struct queue_node * node = (struct queue_node*)kalloc();
    node->page_index = page_ramindex;
    if(curproc->queue_head == 0 && curproc->queue_tail ==0)  //the first queue_node
    {
      curproc-> queue_head = node;
      curproc-> queue_tail = node;
      curproc-> queue_head->next = 0;
      curproc-> queue_head->prev = 0;
      curproc-> queue_head->next = 0;
      curproc-> queue_head->prev = 0;
    }
    else
```

```
    {
      curproc->queue_head->prev = node;
      node->next = curproc->queue_head;
      curproc->queue_head = node;
      curproc->queue_head->prev = 0;
    }
  #endif



}

pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  #if SELECTION != NONE
  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P) && !(*pte & PTE_PG))
      panic("copyuvm: page not present and also not paged out to disk");

    if (*pte & PTE_PG) {
      pte = walkpgdir(d, (void*) i, 1);
      *pte = PTE_U | PTE_W | PTE_PG;
      continue;
    }

    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    // if(*pte & PTE_PG)
    // {
    //   if(mappages(d, (void*)i, PGSIZE, 0, flags) < 0)
    //     panic("copyuvm: mappages failed");
    //   continue;
    // }
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      cprintf("copyuvm: mappages failed\n");
      goto bad;
    }
  }
  #else
  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  #endif
  return d;

bad:
  cprintf("bad: copyuvm\n");
```

```
  freevm(d);
  return 0;
}

//PAGEBREAK!
// Map user virtual address to kernel address.
char*
uva2ka(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if((*pte & PTE_P) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  return (char*)P2V(PTE_ADDR(*pte));
}

// Copy len bytes from p to user address va in page table pgdir.
// Most useful when pgdir is not the current page table.
// uva2ka ensures this only works for PTE_U pages.
int
copyout(pde_t *pgdir, uint va, void *p, uint len)
{
  char *buf, *pa0;
  uint n, va0;

  buf = (char*)p;
  while(len > 0){
    va0 = (uint)PGROUNDDOWN(va);
    pa0 = uva2ka(pgdir, (char*)va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (va - va0);
    if(n > len)
      n = len;
    memmove(pa0 + (va - va0), buf, n);
    len -= n;
    buf += n;
    va = va0 + PGSIZE;
  }
  return 0;
}

int
getNextFreeRamIndex()
{
  int i;
  struct proc * currproc = myproc();
  for(i = 0; i < MAX_PSYC_PAGES ; i++)
  {
    if(((struct page)currproc->ramPages[i]).isused == 0)
      return i;
  }
  return -1;
}


//PAGEBREAK!
// Blank page.
//PAGEBREAK!
// Blank page.
//PAGEBREAK!
// Blank page.

void updateLapa(struct proc* p)
{
  struct page *ramPages = p->ramPages;
  int i;
  pte_t *pte;
  for(i = 0; i < MAX_PSYC_PAGES; i++)
  {
    struct page *cur_page = &ramPages[i];
```

```
    if(!cur_page->isused)
      continue;
    if((pte = walkpgdir(cur_page->pgdir, cur_page->virt_addr, 0)) == 0)
      panic("updateLapa: no pte");
    if(*pte & PTE_A) // if accessed
    {
      cur_page->lapa_counter = cur_page->lapa_counter >> 1; // shift right one bit
      cur_page->lapa_counter |= 1 << 31; // turn on MSB
      *pte &= ~PTE_A;
    }
    else
    {
      cur_page->lapa_counter = cur_page->lapa_counter >> 1; // just shit right one bit
    }
  }
}

void updateNfua(struct proc* p)
{
  struct page *ramPages = p->ramPages;
  int i;
  pte_t *pte;
  for(i = 0; i < MAX_PSYC_PAGES; i++)
  {
    struct page *cur_page = &ramPages[i];
    if(!cur_page->isused)
      continue;
    if((pte = walkpgdir(cur_page->pgdir, cur_page->virt_addr, 0)) == 0)
      panic("updateNfua: no pte");
    if(*pte & PTE_A) // if accessed
    {
      cur_page->nfua_counter = cur_page->nfua_counter >> 1; // shift right one bit
      cur_page->nfua_counter |= 0x80000000; // turn on MSB
      *pte &= ~PTE_A;

    }
    else
    {
      cur_page->nfua_counter = cur_page->nfua_counter >> 1; // just shit right one bit
    }
  }
}
uint indexToEvict()
{
  #if SELECTION==DUMMY
    return 3;
  #endif
  #if SELECTION==SCFIFO
    return scfifo();
  #endif
  #if SELECTION==NFUA
    return nfua();
  #endif
  #if SELECTION==LAPA
    return lapa();
  #endif
  #if SELECTION==AQ
    return aq();
  #else
  return 11; // default
  #endif
}

uint aq()
{
  struct proc* curproc = myproc();
  int res = curproc->queue_tail->page_index;
  struct queue_node* new_tail;
  if(curproc->queue_tail == 0 || curproc->queue_head == 0)
  {
    panic("AQ INDEX SELECTION: empty queue cann't make index selection!");
  }

  if(curproc->queue_tail == curproc->queue_head)
```

```
  {
    curproc->queue_head=0;
    new_tail = 0;
  }
  else
  {
    curproc->queue_tail->prev->next = 0;
    new_tail =  curproc->queue_tail->prev;
  }

  // kfree((char*)curproc->queue_tail);
  curproc->queue_tail = new_tail;

  return  res;


}
uint lapa()
{
  struct proc *curproc = myproc();
  struct page *ramPages = curproc->ramPages;
  /* find the page with the smallest number of '1's */
  int i;
  uint minNumOfOnes = countSetBits(ramPages[0].lapa_counter);
  uint minloc = 0;
  uint instances = 0;

  for(i = 1; i < MAX_PSYC_PAGES; i++)
  {
    // cprintf("i = %d, lapa_counter : %d\n", i, ramPages[i].lapa_counter);
    uint numOfOnes = countSetBits(ramPages[i].lapa_counter);
    if(numOfOnes < minNumOfOnes)
    {
      minNumOfOnes = numOfOnes;
      minloc = i;
      instances = 1;
    }
    else if(numOfOnes == minNumOfOnes)
      instances++;
  }
  if(instances > 1) // more than one counter with minimal number of 1's
  {
      uint minvalue = ramPages[minloc].lapa_counter;
      for(i = 1; i < MAX_PSYC_PAGES; i++)
      {
        uint numOfOnes = countSetBits(ramPages[i].lapa_counter);
        if(numOfOnes == minNumOfOnes && ramPages[i].lapa_counter < minvalue)
        {
          minloc = i;
          minvalue = ramPages[i].lapa_counter;
        }
      }
      return minloc;
  }
  else
    return minloc;
}
uint nfua()
{
  struct proc *curproc = myproc();
  struct page *ramPages = curproc->ramPages;
  /* find the page with the minimal nfua */
  int i;
  uint minval = ramPages[0].nfua_counter;
  uint minloc = 0;

  for(i = 1; i < MAX_PSYC_PAGES; i++)
  {
    // cprintf("i = %d, nufa_counter : %d\n", i, ramPages[i].nfua_counter);
    if(ramPages[i].nfua_counter < minval)
    {
      minval = ramPages[i].nfua_counter;
      minloc = i;
    }
```

```
  }
  return minloc;
}

uint scfifo()
{
  struct proc* curproc = myproc();
   int i;
   while(1)
   {
     for(i = curproc->clockHand ; i < MAX_PSYC_PAGES ; i++)
    {
      pte_t *pte = walkpgdir(curproc->ramPages[i].pgdir, curproc->ramPages[i].virt_addr, 0);
       if(!(*pte & PTE_A)) //ref bit is off
       {
          if(curproc->clockHand == MAX_PSYC_PAGES - 1)
          {
            curproc->clockHand = 0;
          }
          else
          {
            curproc->clockHand = i + 1;
          }
          return i;
       }
       else
       {
          // turn off acess bit
          *pte &= ~PTE_A;
       }
    }

    int j;
    for(j=0; j< curproc->clockHand ;j++)
    {
      pte_t *pte = walkpgdir(curproc->ramPages[j].pgdir, curproc->ramPages[j].virt_addr, 0);
       if(!(*pte & PTE_A)) //ref bit is off
       {
          curproc->clockHand = j + 1;
          return j;
       }
       else
       {
          // turn off acess bit
          *pte &= ~PTE_A;
       }
    }
   }

    panic("scfifo: not found any index!");
    return -1;
}

uint countSetBits(uint n)
{
    uint count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}


void updateAQ(struct proc* p)
{
  struct queue_node *curr_node = p->queue_tail;
  struct page *ramPages = p->ramPages;
  struct page *curr_page = &ramPages[curr_node->page_index];
  struct page *prev_page;
  pte_t *pte_curr;
  pte_t *pte_prev;

  if(p->queue_tail == 0 || p->queue_head == 0)
```

```
    return;

  if(curr_node->prev == 0)
    return;

  prev_page = &ramPages[curr_node->prev->page_index];

  // cprintf("found page index: %d\n", p->queue_tail->page_index);


  while(curr_node != 0)
  {
    // printaq();
    if((pte_curr = walkpgdir(curr_page->pgdir, curr_page->virt_addr, 0)) == 0)
      panic("updateAQ: no pte");
    if(*pte_curr & PTE_A) // an accessed page
    {
      if(curr_node->prev != 0) // there is a page behind it
      {
        if((pte_prev = walkpgdir(prev_page->pgdir, prev_page->virt_addr, 0)) == 0)
          panic("updateAQ: no pte");

        if(!(*pte_prev & PTE_A)) // was not accessed, will swap
          swapAQ(curr_node);
      }
      *pte_curr &= ~PTE_A;
    }

    if(curr_node != 0)
    {
      curr_page = &ramPages[curr_node->page_index];

      if(curr_node->prev != 0)
        prev_page = &ramPages[curr_node->prev->page_index];

      curr_node = curr_node->prev;
    }
  }
}


// will swap curr_node with the node preceding it in the queue.
// assumes there exist a page preceding curr_node.
// queue structure at entry point:
// [maybeLeft?] <-> [prev_node] <-> [curr_node] <-> [maybeRight?]

void swapAQ(struct queue_node *curr_node)
{
  struct queue_node *prev_node = curr_node->prev;
  struct queue_node *maybeLeft, *maybeRight;

  if(curr_node == myproc()->queue_tail)
  {
    myproc()->queue_tail = prev_node;
    myproc()->queue_tail->next = 0;
  }

  if(prev_node == myproc()->queue_head)
  {
    myproc()->queue_head = curr_node;
    myproc()->queue_head->prev = 0;
  }

  // saving maybeLeft and maybeRight pointers for later
    maybeLeft = prev_node->prev;
    maybeRight = curr_node->next;

  // re-connecting prev_node and curr_node (simple)
  curr_node->next = prev_node;
  prev_node->prev = curr_node;

  // updating maybeLeft and maybeRight
  if(maybeLeft != 0)
  {
```

```
    curr_node->prev = maybeLeft;
    maybeLeft->next = curr_node;
  }

  if(maybeRight != 0)
  {
    prev_node->next = maybeRight;
    maybeRight->prev = prev_node;
  }
}

 int
  getNumRefsWarpper(int idx)
  {
    struct proc * curproc = myproc();
    pte_t *evicted_pte = walkpgdir(curproc->ramPages[idx].pgdir, (void*)curproc->ramPages[idx].virt_addr, 0);
    char *evicted_pa = (char*)PTE_ADDR(*evicted_pte);
    return getRefs(P2V(evicted_pa));

  }

int
getRamPageIndexByVirtAddr(char* virtaddr)
{
  struct proc* curproc = myproc();
  int i;
  for(i = 0; i < MAX_PSYC_PAGES; i++)
  {
    if(curproc->swappedPages[i].virt_addr == virtaddr)
      return i;
  }
  return -1;
}
```