AhmedElshobaky / **XV6_PAGING** Public

Code Issues Pull requests Actions Projects Wiki Security Insights

main XV6_PAGING / REQ2_OS_PROJECT / **vm.c** Go to file

**AhmedElshobaky** Add files via upload Latest commit 0c71b12 on Jan 7, 2021 History

1 contributor

597 lines (533 sloc) | 16.2 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"

extern char data[];  // defined by kernel.ld
pde_t *kpgdir;  // for use in scheduler()
struct segdesc gdt[NSEGS];

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

  // Map cpu, and curproc
  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);

  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  // Initialize cpu-local storage.
  cpu = c;
  proc = 0;
}

// Return the address of the PTE in page table pgdir
// that corresponds to virtual address (in u.m) 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)]; //PDE index in page directory (0 to 1023 + FLAGS)
  if(*pde & PTE_P){       //Present bit is on in PDE
    pgtab = (pte_t*)p2v(PTE_ADDR(*pde)); //pgtab = virtual address to beginning of page table

  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0) //if alloc != 0, try to create new page table

      return 0; //page table (PDE) doesn't exist or kalloc failed
    // Make sure all those PTE_P bits are zero.
```

```c
    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; //link PDE to the new page table
  }
  return &pgtab[PTX(va)]; //return PTE in page table which corresponse to va address
}


// Create PTEs for virtual addresses starting at va (va in U.M) 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 was already initialized for some reason
    *pte = pa | perm | PTE_P; //adds page physical address, flags, present bit
    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)
//                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)
      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)
{
  pushcli();
  cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  lcr3(v2p(p->pgdir));  // switch to new 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;
}

int getPagePAddr(int userPageVAddr, pde_t * pgdir){
  pte_t *pte;
  pte = walkpgdir(pgdir, (int*)userPageVAddr, 0);
  if(!pte) //uninitialized page table
    return -1;
  return PTE_ADDR(*pte);
}

void fixPagedOutPTE(int userPageVAddr, pde_t * pgdir){
  pte_t *pte;
  pte = walkpgdir(pgdir, (int*)userPageVAddr, 0);
  if (!pte)
    panic("PTE of swapped page is missing");
  *pte |= PTE_PG;
  *pte &= ~PTE_P;
  *pte &= PTE_FLAGS(*pte); //clear junk physical address
  lcr3(v2p(proc->pgdir)); //refresh CR3 register
}

//This method cannot be replaced with mappages because mappages cannot turn off PTE_PG bit
void fixPagedInPTE(int userPageVAddr, int pagePAddr, pde_t * pgdir){
  pte_t *pte;
  pte = walkpgdir(pgdir, (int*)userPageVAddr, 0);
  if (!pte)
    panic("PTE of swapped page is missing");
  if (*pte & PTE_P)
        panic("PAGE IN REMAP!");
  *pte |= PTE_P | PTE_W | PTE_U;      //Turn on needed bits
  *pte &= ~PTE_PG;                                                  //Turn off inFile bit
  *pte |= pagePAddr;                                                //Map PTE to the new Page
  lcr3(v2p(proc->pgdir)); //refresh CR3 register
}

int pageIsInFile(int userPageVAddr, pde_t * pgdir) {
  pte_t *pte;
  pte = walkpgdir(pgdir, (char *)userPageVAddr, 0);
  return (*pte & PTE_PG); //PAGE IS IN FILE
}


int getFIFO(){
    pte_t * pte;
    int i = 0;
    int pageIndex;
    uint loadOrder;
  recheck:
    pageIndex = -1;
    loadOrder = 0xFFFFFFFF;
    for (i = 0; i < MAX_PYSC_PAGES; i++) {
      if (proc->ramCtrlr[i].state == USED && proc->ramCtrlr[i].loadOrder <= loadOrder){
        pageIndex = i;
        loadOrder = proc->ramCtrlr[i].loadOrder;
      }
    }
    return pageIndex;
  }


int getLRU(){
  int i;
  int pageIndex = -1;
  uint minAccess = 0xffffffff;


  for (i = 0; i < MAX_PYSC_PAGES; i++) {
    if (proc->ramCtrlr[i].state == USED && proc->ramCtrlr[i].accessCount <= minAccess) {
```

```
          minAccess = proc->ramCtrlr[i].accessCount;
          pageIndex = i;
    }
  }
  return pageIndex;
}

int getPageOutIndex(){

  #if FIFO
    return getFIFO();
  #endif
  #if LRU
    return getLRU();
  #endif
  panic("Unrecognized paging machanism");
}

void updateAccessCounters(struct proc * p){
  pte_t * pte;
  int i;
  for (i = 0; i < MAX_PYSC_PAGES; i++) {
    if (p->ramCtrlr[i].state == USED){
      pte = walkpgdir(p->ramCtrlr[i].pgdir, (char*)p->ramCtrlr[i].userPageVAddr,0);
      if (*pte & PTE_A) {
        *pte &= ~PTE_A; // turn off PTE_A flag
         p->ramCtrlr[i].accessCount++;
      }
    }
  }
}

int getFreeRamCtrlrIndex() {
  if (proc == 0)
    return -1;
  int i;
  for (i = 0; i < MAX_PYSC_PAGES; i++) {
    if (proc->ramCtrlr[i].state == NOTUSED)
      return i;
  }
  return -1; //NO ROOM IN RAMCTRLR
}

static char buff[PGSIZE]; //buffer used to store swapped page in getPageFromFile method

int getPageFromFile(int cr2){
  proc->faultCounter++;
  int userPageVAddr = PGROUNDDOWN(cr2);
  char * newPg = kalloc();
  memset(newPg, 0, PGSIZE);
  int outIndex = getFreeRamCtrlrIndex();
  lcr3(v2p(proc->pgdir)); //refresh CR3 register
  if (outIndex >= 0) { //Free location in RamCtrlr is available, no need for swapping
    fixPagedInPTE(userPageVAddr, v2p(newPg), proc->pgdir);
    readPageFromFile(proc, outIndex, userPageVAddr, (char*)userPageVAddr);
    return 1; //Operation was successful
  }
  proc->countOfPagedOut++;
  //If reached here - Swapping is needed.
  outIndex = getPageOutIndex(); //select a page to swap to file
  struct pagecontroller outPage = proc->ramCtrlr[outIndex];
  fixPagedInPTE(userPageVAddr, v2p(newPg), proc->pgdir);
  readPageFromFile(proc, outIndex, userPageVAddr, buff); //automatically adds to ramctrlr
  int outPagePAddr = getPagePAddr(outPage.userPageVAddr, outPage.pgdir);
  memmove(newPg, buff, PGSIZE);
  writePageToFile(proc, outPage.userPageVAddr, outPage.pgdir);
  fixPagedOutPTE(outPage.userPageVAddr, outPage.pgdir);
  char *v = p2v(outPagePAddr);
  kfree(v); //free swapped page
  return 1;
}


void addToRamCtrlr(pde_t *pgdir, uint userPageVAddr) {
  int freeLocation = getFreeRamCtrlrIndex();
```

```c
  proc->ramCtrlr[freeLocation].state = USED;
  proc->ramCtrlr[freeLocation].pgdir = pgdir;
  proc->ramCtrlr[freeLocation].userPageVAddr = userPageVAddr;
  proc->ramCtrlr[freeLocation].loadOrder = proc->loadOrderCounter++;
  proc->ramCtrlr[freeLocation].accessCount = 0;
}


void swap(pde_t *pgdir, uint userPageVAddr){
  proc->countOfPagedOut++;
  int outIndex = getPageOutIndex();
  int outPagePAddr = getPagePAddr(proc->ramCtrlr[outIndex].userPageVAddr, proc->ramCtrlr[outIndex].pgdir);
  writePageToFile(proc, proc->ramCtrlr[outIndex].userPageVAddr, proc->ramCtrlr[outIndex].pgdir);
  char *v = p2v(outPagePAddr);
  kfree(v); //free swapped page
  proc->ramCtrlr[outIndex].state = NOTUSED;
  fixPagedOutPTE(proc->ramCtrlr[outIndex].userPageVAddr, proc->ramCtrlr[outIndex].pgdir);
  addToRamCtrlr(pgdir, userPageVAddr);
}


int isNONEpolicy(){
        #if NONE
                return 1;
        #endif
        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;
  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  if (!isNONEpolicy()){
     if (PGROUNDUP(newsz)/PGSIZE > MAX_TOTAL_PAGES && proc->pid > 2) {
                    cprintf("proc is too big\n", PGROUNDUP(newsz)/PGSIZE);
                    return 0;
                  }
        }

  a = PGROUNDUP(oldsz);
  int i = 0; //debugging
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    i++;
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
    if (!isNONEpolicy() && proc->pid > 2){
      if (PGROUNDUP(oldsz)/PGSIZE + i > MAX_PYSC_PAGES)
        swap(pgdir, a);
      else //there's room
        addToRamCtrlr(pgdir, a);
          }
  }
  return newsz;
}


//This must use userVaddress+pgdir addresses!
//(The proc has identical vAddresses on different page directories until exec finish executing)
void removeFromRamCtrlr(uint userPageVAddr, pde_t *pgdir){
  if (proc == 0)
    return;

  int i;
  for (i = 0; i < MAX_PYSC_PAGES; i++) {
```

```
    if (proc->ramCtrlr[i].state == USED
        && proc->ramCtrlr[i].userPageVAddr == userPageVAddr
        && proc->ramCtrlr[i].pgdir == pgdir){
      proc->ramCtrlr[i].state = NOTUSED;
      return;
    }
  }
}

void removeFromFileCtrlr(uint userPageVAddr, pde_t *pgdir){
  if (proc == 0)
    return;
  int i;
  for (i = 0; i < MAX_TOTAL_PAGES-MAX_PYSC_PAGES; i++) {
    if (proc->fileCtrlr[i].state == USED
        && proc->fileCtrlr[i].userPageVAddr == userPageVAddr
        && proc->fileCtrlr[i].pgdir == pgdir){
      proc->fileCtrlr[i].state = NOTUSED;
      return;
    }
  }
}
// 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){
  pte_t *pte;
  uint a, pa;

  if(newsz >= oldsz)
    return oldsz;

  a = PGROUNDUP(newsz);
  int i = 0; //debugging
  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte) //uninitialized page table
      a += (NPTENTRIES - 1) * PGSIZE; //jump to next page table
    else if((*pte & PTE_P) != 0){     //page table exists and page is present
      pa = PTE_ADDR(*pte);            //pa = beginning of page physical address
      if(pa == 0)
        panic("kfree");
      char *v = p2v(pa);
      kfree(v); //free page
      if (!isNONEpolicy())
        removeFromRamCtrlr(a, pgdir);

      i++;
      *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);
  int j = 0;
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){ //PDE exists
      char * v = p2v(PTE_ADDR(pgdir[i]));
      kfree(v); //free page table
      j++;
    }
  }
  kfree((char*)pgdir); //free page directory
}


// Clear PTE_U on a page. Used to create an inaccessible
```

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

  if((d = setupkvm()) == 0)
    return 0;
  int j = 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if (*pte & PTE_PG){
        fixPagedOutPTE(i, d);
        continue;
    }

    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);
    j++;
    if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
      goto bad;
  }
  return d;

bad:
  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;
}

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

```