Operation System
Project #4: Page Replacement

2014311577 Global Economics
Dongmin Kim

1. LRU List setups

LRU list has following components:

```
struct page pages[PHYSTOP/PGSIZE];
int num_free_pages;

struct page *page_lru_head;
int num_lru_pages;
```

To manage LRU list, 2 main functions are defined: insert / delete / logpage

Insertpage Insert free pages to the last of LRU list. This is done just by adding struct page to the prev node of page_lru_head.

```
// insert page to the last of LRU list.
void
insertpage(struct page * p)
{
  // case 1.
  // initial case.
  if(!page_lru_head){
    page_lru_head = p;
    p->next = p;
    p->prev = p;
    num_lru_pages++;
    return;
  }

  // insert to the last.
  struct page *last = page_lru_head -> prev;

  last->next = p;
  p->prev = last;
  p->next = page_lru_head;
  page_lru_head->prev = p;

  num_lru_pages++;
}
```

And deletepage removes pages with given pgdir, and vaddr. This is done by iterating the LRU list for a single round.

```
void
deletepage(pde_t *pgdir, void *va)
{
  if(!page_lru_head){
    // no entry to delete.
    // may imply an error.
    return;
  }

  struct page *cur = page_lru_head;
  while(1){
    // page is found.
    if(cur->pgdir == pgdir && cur->vaddr == (char*)va)
      break;

    cur = cur -> next;

    if(cur == page_lru_head){
      // page not found
      // may imply an error.
      return;
    }
  }

  // relocate page_lru_head
  if(cur == page_lru_head){
    if(cur->next == cur){
      page_lru_head = 0;
    }
    else{
      page_lru_head = cur->next;
    }
  }

  // rearrange
  struct page *nc, *pc;
  nc = cur->next; // next to cur
  pc = cur->prev; // prev to cur

  pc->next = nc;
  nc->prev = pc;
```

```
  // clear page
  cur->next = 0;
  cur->prev = 0;
  cur->pgdir = 0;
  cur->vaddr = 0;
  num_lru_pages--;
}
```

logpage logs pages with pgdir, va, pa to the pages array. For a given physical address, this logs according to the corresponding index of the page, which is given by pa/PGSIZE.

```
// log page to corresponding pages[] entry
struct page*
logpage(pde_t *pgdir, void *va, uint pa)
{
  int idx = pa / PGSIZE;
  pages[idx].pgdir = pgdir;
  pages[idx].vaddr = (char *) va;

  return &pages[idx];
}
```

2. Bitmap management

To log the swap status, bitmap of one page is defined and managed.

```
/*
  Bitmap management Unit
*/

// bitmap to log swap status
struct {
  struct spinlock lock;
  char * bitmap;
} bmap;

// initialize bitmap
void
bitmapinit(void)
{
  if(!bmap.bitmap)
  {
    initlock(&bmap.lock, "bmap");
    bmap.bitmap = kalloc();
    memset(bmap.bitmap, 0, PGSIZE);
  }
}
```

To prevent situations of mutual access, bitmap has spinlock, and allocated by kalloc function of kernel, and set to 0. And bitmapinit function is called right after kinit functions.

To manage bitmap operations, 2 functions are defined: getblkno, setbit.

getblkno iterates bitmap, and find for closest index 0 of bitmap.

```
// get block number from bitmap.
int
getblkno(void)
{
  uint idx = 0;
  int flag = 0;

  acquire(&bmap.lock);
  for(int i = 0; i < PGSIZE; i++)
  {
    // swap index is full.
    if(bmap.bitmap[i] == 0xFF)
      continue;
    else
    {
```

```
      // not full: search for 0.
      for(int j = 0; j < 8; j++)
      {
        int mask = 1<<j;
        if(!(bmap.bitmap[i] & mask))
        {
          idx = i * 8 + j;
          flag = 1;
          break;
        }
      }
      if(flag) break;
    }
  }
  release(&bmap.lock);
  return idx;
}
```

And setbit function sets given index of bitmap to zero/one.

```
// set bitmap bit to 0 or 1.
void
setbit(int idx, int boolean)
{
  acquire(&bmap.lock);
  int i = idx / 8;
  int j = 1 << (idx % 8);
  bmap.bitmap[i] = (bmap.bitmap[i] & ~j) | (boolean << (idx % 8));
  release(&bmap.lock);
}
```

3. Finding victim page

In kalloc function in kalloc.c, if there is no free list available, kernel searches for the page, by calling reclaim function.

```
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);

try_again:

  r = kmem.freelist;

  // bitmap exception
  if((char *) r == bmap.bitmap){
    // access to bitmap.
    // might imply an error.
    cprintf("bitmap access: skip.\n");
    kmem.freelist = r->next;
  }

  if(!r && reclaim())
    goto try_again;

  if(r)
    kmem.freelist = r->next;

  if(kmem.use_lock)
    release(&kmem.lock);

  if(!r) {
    cprintf("ERROR: OUT OF MEMORY!\n");
    return 0; // OOM error.
  }

  return (char*)r;
}
```

And reclaim function do the following things:

```
// reclaim pages by evicting
int
reclaim(void)
{
  struct page *victim;
  pte_t *pte;
  uint pa;
  uint blkno;

  // 1. get victim
  victim = getvictim(); // victim represented by LRU list.
  if(!victim){
    // out of memory error
    return 0;
  }
  pte = getpte(victim->pgdir, victim->vaddr);
  pa = PTE_ADDR(*pte);  // get physical address of the victim page.
  deletepage(victim->pgdir, victim->vaddr);

  // 2. log to bitmap
  blkno = getblkno();
  setbit(blkno, 1);

  // unlock
  if(kmem.use_lock)
    release(&kmem.lock);

  // 3. get swapspace by blkno,
  // and write it to swapspace.
  swapwrite(P2V(pa), blkno);

  // 4. free page.
  kfree(P2V(pa));

  // lock
  if(kmem.use_lock)
    acquire(&kmem.lock);

  // 5. set pte.
  *pte = 0;
  *pte = (blkno + SWAPBASE) << 12;
  *pte &= ~PTE_P;
```

```
  return 1;
}
```

First, it gets victim page by following getvictim function.

```
// get victim page
// if PTE_A is 1, set to 0 and push back
// else, that is victim
struct page*
getvictim(void)
{
  struct page *cur = page_lru_head;
  pte_t * pte;

  while(1)
  {
    pte = getpte(cur->pgdir, cur->vaddr);

    if(*pte & PTE_A){
      struct page *tmp = cur;
      pde_t * pgd = cur->pgdir;
      char * va = cur->vaddr;

      // set access bit to 0.
      *pte &= ~PTE_A;

      // delete and move back
      deletepage(pgd, va);
      tmp->pgdir = pgd;
      tmp->vaddr = va;
      insertpage(tmp);
      cur = page_lru_head;
    }
    else{
      break;
    }
  }

  return cur;
}
```

getvictim function searches the LRU list. If PTE_A bit is 0, that page is victim and returned. Else, if PTE_A is 1, PTE_A is set to 0 and go back to the last index of LRU list.

And the page is deleted, after the information regarding victim page is passed.

Second, logging is done, with bitmap lock. This is done by iterating bitmap data to search for the very first 0.

Third, Swapwrite, to store evicted page in swapspace

Fourth, victim page is freed, by calling kfree function.

Lastly, PTE is set to 0, and block offset is logged, right after flag bits.

4. User Page Management

Managing swappable pages is done by handling 4 functions:
inituvm, allocuvm, dealloc, copy

First, inituvm is set as follows:

```
// 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);

  // LRU LIST
  struct page *p;
  p = logpage(pgdir, (void*)0, V2P(mem));
  insertpage(p);
}
```

When a page is given to user process, that page is logged, and managed in LRU List.

Second, allocuvm is set as follows:

```
// 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;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
```

```
    mem = kalloc();

    // should not happen, if kalloc is done properly
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);

    // LRU LIST
    // if mem is guaranteed to be allocated,
    // log the page, and insert it to LRU list.
    struct page *p;
    p = logpage(pgdir, (void*) a, V2P(mem));
    insertpage(p);

    // should not happen, if kalloc is done properly
    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;
    }
  }
  return newsz;
}
```

If the kalloc is done properly, memory is allocated well enough, so the page is inserted, and managed in LRU List.

Third, deallocuvm is set as follows:

```
// 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);
  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
    else if((*pte & PTE_P) == 0){
      // present bit is zero
      if(*pte){
        // logged in swapspace
        // delete page in swapspace
        int offset = (*pte >> 12) - SWAPBASE;
        setbit(offset, 0);
        *pte = 0;
      }
    }
    else if((*pte & PTE_P) != 0){
      deletepage(pgdir, (char*) a);

      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = P2V(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}
```

When dealloc, two cases may happen:

1. PTE_P is 1: normal case, so just kfree and clean the bit
2. PTE_P is 0: swap-out case, so this is not in memory, but !(*pte) indicates that it saves offset. So in this case, bitmap is set to 0 to indicate no corresponding page in swapspace exists.

Fourth, copyuvm is set as follows:

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  cprintf("copyuvm\n");
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
```

```
  char *mem;

  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))
      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;
    }

    // LRU LIST
    struct page *p;
    p = logpage(d, (void *)i, V2P(mem));
    insertpage(p);

  }
  return d;

bad:
  // this shouldn't be happening.
  cprintf("copyuvm-bad\n");
  freevm(d);
  return 0;
}
```

copyuvm may have 2 cases:
if PTE_P is 1 : normal case, just log the pate to the LRU List.
if PTE_P is 0: this may have offset data in *pte. So when trying memmove, pagefault handler will be operating.

5. Page Fault Handler

In trap.c, pagefault is raised by checking trapno. That is:

```
  if(tf->trapno == T_PGFLT){
    // call pagefault
    int pgf = handle_pgfault(myproc(), rcr2());
    if(pgf == -1)
      myproc()->killed = 1;
    return;
  }
```

And handle_pgfault is called in kalloc.c.

```
int handle_pgfault(struct proc *p, uint faultaddr)
{
  char *mem;
  uint pg_loc = PGROUNDDOWN(faultaddr);
  int offset;
  pte_t *pte;

  mem = kalloc();
  if(!mem){
    cprintf("mem alloc failed\n");
    return -1;
  }
  memset(mem, 0, PGSIZE);

  if((pte = walkpgdir(p->pgdir, (char*)pg_loc, 0)) == 0)
    return -1;
  else {
    // this case, PTE_P is guaranteed to be 0.
    // PTE_P were to be 1, no page fault would have occured.

    if(*pte & PTE_P) return -1;
    if(!(*pte))
      panic("Pagefault Handler: PTE_P is 0, and No data for
swapspace.");

    offset = (*pte >> 12) - SWAPBASE;

    *pte = V2P(mem) | PTE_P;
    swapread(mem, offset);
    setbit(offset, 0); // turn off the swap bitmap bit.

    return 1;
  }
}
```

pagefault handler gets process id and pgdir by struct proc passed, and virtual address of fault address is passed by faultaddr.

Pagefault is occured in either case: one is page directory missing cases, and the other is page itself is missing.
However, In this project, pagefault is caused only by swapouts, and page directory page shouldn’t be swapped out.
Therefore, PTE_P is checked only for page itself, and it should have offset there. So Swapread is done here.

5. Testing

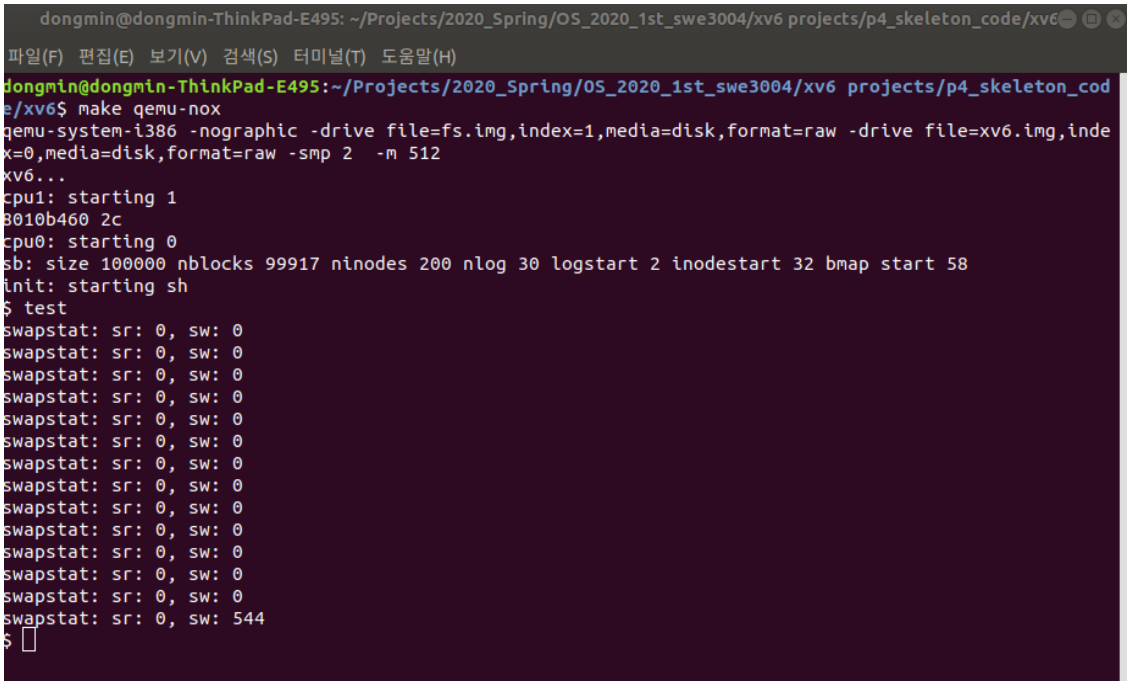
Testing is done using following codes:

Test 1: sbrk calls only

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void){

  int sr, sw;
  for(int i = 0 ; i < 13 ; i++){
    sbrk(0x1000000);
    swapstat(&sr, &sw);
    printf(1, "swapstat: sr: %d, sw: %d\n", sr, sw);
  }

  sbrk(0xD00000);
  swapstat(&sr, &sw);
  printf(1, "swapstat: sr: %d, sw: %d\n", sr, sw);

  exit();
}
```

This test code firstly calls sbrk for 13 times, and each with the argument of 0x1000000. This corresponds to the storage of around 4096 pages, in total, 53248 pages.

After that, it calls sbrk with 0xD00000. This corresponds to around 3328 pages, so page requested by the code above calls 56576 pages, and it calls swap-out page mechanism(page replacement by reclaim function) while allocating space for this process. Result Screenshot is as follows:



Test 2: sbrk in malloc, with forked processes

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#define N 10

int main(void){

  int pids[N];
  int i;
  int n = N;


  // Start children.
  for (i = 0; i < n; i++) {

    if((pids[i] = fork()) < 0) {
      printf(1, "forkerror\n");
      exit();
    }
    else if (pids[i] == 0) {
      // child
      printf(1, "fork: child %d. \n", i+1);

      int *page = malloc(0x1d00000);
      int data;
```

```
      page[0] = 0;
      page[1024] = 1024;

      printf(1, "data before loop: [%d %d]\n", page[0], page[1024]);

      for(int i = 0; i < 2048; i++){
        data = (int)(i * 3.14 - 10);
        page[i] = data;
      }

      printf(1, "data after loop: [%d %d]\n", page[0], page[1024]);
      printf(1, "exit: child %d.\n", i+1);

      exit();
    }
  }


  // Wait for children to exit.
  int pid;
  while (n > 0) {
    pid = wait();

    int sr, sw;
    swapstat(&sr, &sw);
    printf(1, "swapstat: sr: %d, sw: %d\n", sr, sw);
    printf(1, "Child with PID %d exits \n", pid);
    n--;
  }

  exit();
}
```

Testing Code forks 10 childs, and each process calls malloc function with 0x1d00000, which corresponds to 7424 pages. Malloc function does the same thing as sbrk, as it allocates memory by calling sbrk, which grows the process size to a certain amount.
Following is the testing result. This took quite a long time, but all 10 processes exited, with swapread(sr) and swapwrite(sw) operations of 24, 9048 for each.

```
dongmin@dongmin-ThinkPad-E495:~/Projects/2020_Spring/OS_2020_1st_swe3004/xv6 projects/p4_skeleton_cod
e/xv6$ make qemu-nox
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-fra
me-pointer -fno-stack-protector -fno-pie -no-pie   -c -o test2.o test2.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _test2 test2.o ulib.o usys.o printf.o umalloc.o
objdump -S _test2 > test2.asm
objdump -t _test2 | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > test2.sym
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertes
ts _wc _zombie _swaptest _test _test2
nmeta 83 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 25) blocks 99917 total 100000
balloc: first 683 blocks have been allocated
balloc: write bitmap block at sector 58
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,inde
x=0,media=disk,format=raw -smp 2  -m 512
xv6...
cpu1: starting 1
8010b460 2c
cpu0: starting 0
sb: size 100000 nblocks 99917 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test2
ffork: child 3.
fork: child 8.
fork: child 2.
fork: child 4.
fork: child 5.
ffork: child 7.
fork: child 9.
ork: child 1.
ork: child 6.
fork: child 10.
data before loop: [0 1024]
data after loop: [-10 3205]
exit: child 7.
swapstat: sr: 0, sw: 9022
Child with PID 10 exits
data before loop: [0 1024]
data after loop: [-10 3205]
exit: child 1.
swapstat: sr: 0, sw: 9029
Child with PID 4 exits
data before loop: [0 1024]
data after loop: [-10 3205]
exit: child 4.
swapstat: sr: 0, sw: 9038
Child with PID 7 exits
data before loop: [0 1024]
data after loop: [-10 3205]
exit: child 9.
swapstat: sr: 0, sw: 9043
Child with PID 12 exits
data before loop: [0 1024]
```

```
data after loop: [-10 3205]
exit: child 6.
swapstat: sr: 0, sw: 9048
Child with PID 9 exits
data befodata before loop: [0 1024]
data after loop: [-10 3205]
exit: child 5.
data before loop: [0 1024]
data after loop: [-10 3205]
exit: child s2.
data before loop: [0 1024]
data after loop: [-10 3205]
exit: child 8.
wapstat: resr: 3, sw: 9048
Child with PID 8 exits
 loop: [0 1024]
data after loop: [-10 3205]
exit: child 10.
swapstat: sr: 10, sw: 9048
Child with PID 5 exits
swapstat: sr: 10, sw: 9048
Child with PID 11 exits
swapstat: sr: 11, sw: 9048
Child with PID 13 exits
data before loop: [0 1024]
data after loop: [-10 3205]
exit: child 3.
swapstat: sr: 24, sw: 9048
Child with PID 6 exits
$
```