Operation System
Project #3: Virtual Memory System Calls

2014311577 Global Economics
Dongmin Kim

0. MMAP_AREA Declaration and Settings

In vm.c,

```
struct mmap_area {
  struct file * f;
  uint addr;
  int length;
  int offset;
  int prot;
  int flags;
  struct proc * p;
};
```

mmap area of array size 64, is defined here

```
struct mmap_area mmlist[64];
int mmlen = 0;
```

and mmlen will count the number of lists.

Some helper functions are defined.

1) copyEntry : copies mmap_area data of virtual memory, from parent process. entry is kept last part of the array. This function is for forked process.

```
void copyEntry(struct proc * curp, struct proc * copyp)
{
  int idx = 0, flag = 0;
  struct mmap_area *mmentry;
  for( ; idx < mmlen ; idx++)
  {
    mmentry = &mmlist[idx];
    if(curp->pid == mmentry->p->pid)
    {
      flag = 1;
      break;
    }
  }
```

```
  if(!flag) return;
  else{
    struct mmap_area newmm;
    newmm.f = mmentry->f;
    newmm.addr = mmentry->addr;
    newmm.length = mmentry->length;
    newmm.offset = mmentry->offset;
    newmm.prot = mmentry->prot;
    newmm.flags = mmentry->flags;
    newmm.p = copyp;
    mmlist[mmlen++] = newmm;
  }

}
```

In proc. c, fork() is calling this function as follows:

```
 // mmap area copy
  copyEntry(curproc, np);
```

2) deleteEntry(int idx): deletes given index's mmlist array.

```
void deleteEntry(int idx)
{
  if(idx < 0 || idx >= mmlen){
    cprintf("delete entry out of idx, exit\n");
    return;
  }

  for(int i = idx + 1; i < mmlen; i++)
    mmlist[i-1] = mmlist[i];
  mmlen--;
}
```

3) pagebelongs & getaddr : These functions check if given address belongs to some entries in mmap_areas(pagebelongs), and returns appropriate address to map.

```
int pagebelongs(uint pgbase, uint base, int length)
{
  if(base <= pgbase && pgbase < base+length)
    return 1;
  return 0;
}
```

```
uint getaddr(uint start, int length, struct proc * p)
{
  uint addr = start;
  struct mmap_area *mmentry;
  int numpgs = length % PGSIZE == 0? length/PGSIZE: length/PGSIZE + 1 ;

  // check mmap region by page unit
  for(int i = addr ; ; i += PGSIZE)
  {
    int pginentry = 0;
    for(int pick = i; pick < i + PGSIZE * numpgs; pick += PGSIZE)
    {
      // iterate through mmlist
      for(int k = 0; k < mmlen; k++)
      {
        mmentry = &mmlist[k];
        if(p->pid == mmentry->p->pid
          && pagebelongs(pick, mmentry->addr, mmentry->length)){
          pginentry = 1;
          break;
        }
      }
      if(pginentry) break;
    }

    if(!pginentry){
      addr = i;
      break;
    }
  }

  return addr;
}
```

In getaddr(), we iterate through the mmlists, and try to get available memory address. To get vaild, firstly, it should not conflict with other addresses allocated within same process id, and secondly, if not, it should provide address for different pids.

1. MMAP
    1) Resolving Address
    Address can be null, of fixed.
    if it is anonymous mapping, we call getaddr function to get appropriate address.
    Else, we analyse the given address, and if it is occupied, we just return 0.

```
if(addr && (flags & MAP_FIXED)){
    // if address is given, and flag is MAP_FIXED
    // should map into according address.
    va = MMAPBASE + addr;
    va_proper = getaddr(va, length, curproc);

    // if that address is already being occupied,
    // return 0
    if(va != va_proper)
      return 0;
  } else {
    va = getaddr(MMAPBASE, length, curproc);
  }
```

    1) register mmap_area

```
  nmmap.addr = va;
  nmmap.length = length;
  nmmap.offset = offset;
  nmmap.prot = prot;
  nmmap.flags = flags;
  nmmap.p = curproc;

  if(!(flags & MAP_ANONYMOUS)){
    f = curproc->ofile[fd];
    nmmap.f = f;

    // prot should match with file's open tag.
    if(!(f->writable) && (prot & PROT_WRITE))
      return 0;
  }

  mmlist[mmlen++] = nmmap;
```

    after error handle, we register mmap_area.

2) Classify
case 1: non- populate mapping- we have nothing to do. pagefault handler will do the remainings.
case 2: populate mapping: we need to assign corresponding pages, by analysing length of the file to allocate.

```
// 3. check if populate is given
  if(flags & MAP_POPULATE) {
    // check if it is anonymous mapping.
    if(flags & MAP_ANONYMOUS) {
      for(int i = 0; i < numpgs; i++) {
        // allocate physical memories for each pages.
        // set to zero(anonymous mapping)
        mem = kalloc();
        memset(mem, 0, PGSIZE);
        mappages(curproc->pgdir, (char *) (va + i*PGSIZE), PGSIZE,
V2P(mem), mm_flag);
      }
    } else {
      for(int i = 0; i < numpgs; i++) {
        // allocate physical memories for each pages.
        // set to 0, and read file. (file mapping)
        mem = kalloc();
        memset(mem, 0, PGSIZE);

        // readi operations
        ilock(f->ip);
        begin_op();
        readi(f->ip, mem, i*PGSIZE+offset, PGSIZE);
        end_op();
        iunlock(f->ip);

        // mappages
        mappages(curproc->pgdir, (char *) (va + i*PGSIZE), PGSIZE,
V2P(mem), mm_flag);
      }
    }
  } // else, for non-populate mappings,
    // nothing else left to do.
    // tasks will be handled at handle_pgfault()
```

2. Page Fault Handling

In trap.c, page fault handling is called.
For each trap, if trapno is T_PGFLT(14), pagefault function is called.

```
  if(tf->trapno == T_PGFLT){// page fault handling
    int pgfh = handle_pgfault(myproc(), rcr2(), tf->err);
    if(pgfh == -1) // kill current process if it is not handled properly
      myproc()->killed = 1;
    return;
  }
```

In vm.c,

1) check if mmlist stores information. if not, returns an error.

```
// find according mapping region in mmlist
  for(int i = 0; i < mmlen; i++){
    if((mmlist[i].p)->pid == p->pid && pagebelongs(pg_loc, mmlist[i].addr,
mmlist[i].length)){
      in_mmlist = 1;
      mmentry = &mmlist[i];
      break;
    }
  }

  // check if there exists corresponding mmap_area
  if(!in_mmlist) return -1;
  // check if access is write, with mmap area is write prohibited
  if((errorcode == 1) && !(mmentry->prot & PROT_WRITE)) return -1;
```

2) After, we need to allocate 1 page for handling pagefault. firstly, set it to 0s.

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

3) if it is file mapping, we need to fill up the files, by calling readi() function.

```
 if(!(mmentry->flags & MAP_ANONYMOUS)){ // if it is not anonymous mapping
    // set file, and offset
    f = mmentry->f;
```

```
    offset = mmentry->offset + pg_loc - mmentry->addr;

    // readi operations
    ilock(f->ip);
    begin_op();
    readi(f->ip, mem, offset, PGSIZE);
    end_op();
    iunlock(f->ip);

  } // else, do nothing,
    //leave memory with 0's
```

if it is anonymous mapping, nothing left to do but mappings.

```
mappages(p->pgdir, (char *) pg_loc, PGSIZE, V2P(mem), mm_flag);
```

3. MUNMAP

munmap frees allocated address which were assigned by mmap. only address is parameter for the function.

1) Firstly, we need to find according mmap_area entry.
   if not, return -1 and exit.

```
// find according mapping region in mmlist
  for(i = 0; i < mmlen; i++){
    if((mmlist[i].p)->pid == curproc->pid && pagebelongs(addr,
mmlist[i].addr, mmlist[i].length)){
      in_mmlist = 1;
      mmentry = &mmlist[i];
      break;
    }
  }
```

2) Secondly, we walk through the page table & entries, to find according physical address.

```
for(int pg = 0; pg < numpgs; pg++){
    int va = addr + PGSIZE * pg;
    pte = walkpgdir(p->pgdir, (char *) va, 0);
    if(*pte & PTE_P){
      int pa = PTE_ADDR(*pte);
      char * v = P2V(pa);

      // before return, write back
      // just for the file && write case
      if(mmentry->prot & PROT_WRITE && !(mmentry->flags & MAP_ANONYMOUS)) {
        ilock(mmentry->f->ip);
        begin_op();
        writei(mmentry->f->ip, (char *)va, mmentry->offset + pg * PGSIZE,
PGSIZE);
        end_op();
        iunlock(mmentry->f->ip);
      }

      // free memory here
      kfree(v);
      *pte = 0;
    }
  }
```

3) according information is stored by writei function, data passed inodes of the files.
4) delete entry, by calling pre-defined helper function

```
deleteEntry(i);
```

4. Freemem

To log the number of free memories, we define variable “numfreemem”

```
uint numfreemem = 0;
```

while iterating through by freerange() function, kfree() is called.

so, we have two function to focus on:

1) kfree() : this function frees the physical memory, so numfreemem is increased, whenever it is called.

```
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;

  numfreemem++; // page freed: inc free memory cnt

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

2) kalloc():opposite to kfree, numfreemem is decreased as it allocates memory to given process.

Finally, we can define functions for system call as:

```
int
freemem(void)
{
  return numfreemem;
}
```

This will return static variable numfreemem.