Lecture 28: Memory management of user processes in xv6

Mythili Vutukuru
IIT Bombay
https://www.cse.iitb.ac.in/~mythili/os/
Memory management of user processes

- User process needs memory pages to build its address space
  - User part of memory image (user code/data/stack/heap)
  - Page table (mappings to user memory image, as well as to kernel code/data)
- Free list of kernel used to allocate memory for user processes via kalloc()
- New virtual address space for a process is created during:
  - init process creation
  - fork system call
  - exec system call
- Existing virtual address space modified in sbrk system call (expand heap)
- How is page table of a process constructed?
  - Start with one page for the outer page directory
  - Allocate inner page tables on demand (if no entries present in inner page table, no need to allocate a page for it) as memory image created or updated
Functions to build page table (1)

- Every page table begins with setting up kernel mappings in `setupkvm()`
- Outer pgdir allocated
- Kernel mappings defined in “kmap” added to page table by calling “mappages”
- After `setupkvm()`, user page table mappings added
Functions to build page table (2)

- Page table entries added by “mappages”
  - Arguments: page directory, range of virtual addresses, physical addresses to map to, permissions of the pages
  - For each page, walks page table, get pointer to PTE via function “walkpgdir”, fills it with physical address and permissions
- Function “walkpgdir” walks page table, returns PTE of a virtual address
  - Can allocate inner page table if it doesn’t exist

```c
1756 // Create PTEs for virtual addresses starting at va that refer to
1757 // physical addresses starting at pa. va and size might not
1759 static int
1760 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1761 {
1762    char *a, *last;
1763    pte_t *pte;
1764    487 //物理地址和虚拟地址的映射关系
1765    a = (char*)PGROUNDOWN((uint)va);
1766    last = (char*)PGROUNDOWN((uint)va + size - 1);
1767    for (;;) {
1768        if ((pte = walkpgdir(pgdir, a, 1)) == 0)
1769            return -1;
1770        if (pte & PTE_P)
1771            panic("remap");
1772        *pte = pa | perm | PTE_P;
1773        if (a == last)
1774            break;
1775        a += PGSIZE;
1776        pa += PGSIZE;
1777    }
1778    return 0;
1779 }
1780
1781 // Return the address of the PTE in page table pgdir
1782 // that corresponds to virtual address va. If alloc=0,
1783 // create any required page table pages.
1784 static pte_t *
1785  walkpgdir(pde_t *pgdir, const void *va, int alloc)
1786 {
1787    pde_t *pde;
1788    pte_t *pteb;
1789    pde = 4pgdir[PA(va)];
1790    if ((pde & PTE_P) &&
1791       (pteb = (pte_t*)P2V(PTE_ADDR(*pde)))
1792    ) else {
1793        if (alloc || (pteb = (pte_t*)kalloc()) == 0)
1794            return 0;
1795        // Make sure all those PTE_P bits are zero.
1796        memset(pteb, 0, PGSIZE);
1797        // The permissions here are overly generous, but they can
1798        // be further restricted by the permissions in the page table
1799        // entries, if necessary.
1800        if (pde = VPt(pgtab) | PTE_P | PTE_W | PTE_U;)
1801    } else {
1802        return &pteb[PTX(va)];
1803    }
1804 }
```
Fork: copying memory image

- Function “copyuvm” called by parent to copy parent memory image to child
  - Create new page table for child
  - Walk through parent memory image page by page and copy it to child, while adding child page table mappings
- For each page in parent
  - Fetch PTE, get physical address, permissions
  - Allocate new page for child, and copy contents of parent’s page to new page of child
  - Add a PTE from virtual address to physical address of new page in child page table
- Real operating systems do copy-on-write: child page table also points to parent pages until either of them modifies it
  - Here, xv6 creates separate memory images for parent and child right away
Growing memory image: sbrk

• Initially heap is empty, program “break” (end of user memory) is at end of stack
  — Sbrk() system call invoked by malloc to expand heap

• To grow memory, allocuvm allocates new pages, adds mappings into page table for new pages

• Whenever page table updated, must update cr3 register and TLB (done even during context switching)

```c
2557 int
2558 growproc(int n)
2559 {
2560     uint sz;
2561     struct proc *curproc = myproc();
2562     sz = curproc->sz;
2563     if(n > 0){
2564         if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
2565             return -1;
2566     } else if(n < 0){
2567         if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
2568             return -1;
2569     }
2570     curproc->sz = sz;
2571     newsz = sz;
2572     switchuvm(curproc);
2573     return 0;
2574 }
```
allocuvm: grow address space

- Walk through new virtual addresses to be added in page size chunks
- Allocate new page, add it to page table with suitable user permissions
- Similarly deallocuvm shrinks memory image, frees up pages
Exec system call (1)

- Read ELF binary file from disk into memory
- Start with new page table, add mappings to new executable pages and grow virtual address space
  - Do not overwrite old page table yet

```c
6609 int
6610 exec(char *path, char **argv)
6611 {
6612 char *s, *last;
6613 int i, off;
6614 uint argc, sz, sp, ustack[3+MAXARG+1];
6615 struct elfhdr elf;
6616 struct inode *ip;
6617 struct proghdr ph;
6618 pde_t *pgdir, *oldpgdir;
6619 struct proc *curproc = myproc();
6620 begin_op();
6621 if((ip = namei(path)) == 0){
6622   end_op();
6623   fprintf("exec: fail\n");
6624   return -1;
6625 }
6626 if(!lock(ip);
6627 }
6628 pgdir = 0;
6629 // Check ELF header
6630 if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6631 goto bad;
6632 if(elf.magic != ELF_MAGIC)
6633 goto bad;
6634 if((pgdir = setupvm()) == 0)
6635 goto bad;
6636 // Load program into memory.
6637 int sz = 0;
6638 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6639   if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6640     goto bad;
6641   if(ph.type != ELF_PROC_LOAD)
6642     goto bad;
6643   if(ph.memsz < ph.filesz)
6644     goto bad;
6645   if(ph.vaddr + ph.memsz < ph.vaddr)
6646     goto bad;
6647   if((sz = allocvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6648     goto bad;
6649   goto bad;
6650   if((sz = allocvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6651     goto bad;
6652   goto bad;
6653   if(ph.vaddr % PGSIZE != 0)
6654     goto bad;
6655   if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6656     goto bad;
6657 }
6658 iunlock(ip);
6659 end_op();
6660 ip = 0;
```
Exec system call (2)

- After executable is copied to memory image, allocate 2 pages for stack (one is guard page, permissions cleared, access will trap)
- Push exec arguments onto user stack for main function of new program
  - Stack has return address, argc, argv array (pointers to variable sized arguments), and the arguments themselves
Exec system call (3)

- If no errors so far, switch to new page table that is pointing to new memory image
  - If any error, go back to old memory image (exec returns with error)
- Set eip in trapframe to start at entry point of new program
  - Returning from trap, process will run new executable
Summary

• Memory management for user processes
  – Build page table: start with kernel mappings, add user entries to build virtual address space
  – Memory management code in fork, exec, sbrk