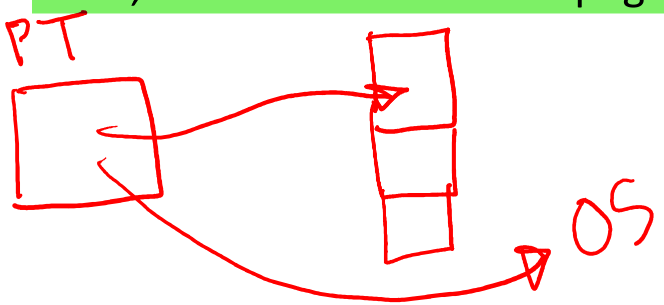
Lecture 28: Memory management of user processes in xv6

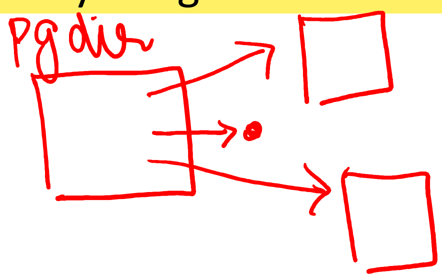
Mythili Vutukuru
IIT Bombay
https://www.cse.iitb.ac.in/~mythili/os/

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

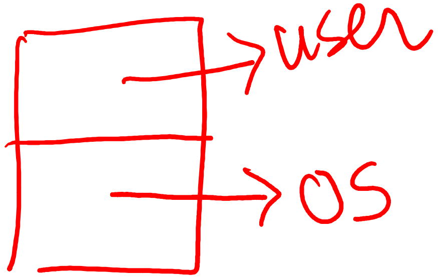




PHYSTOP: 0xE000000 which is equivalent to 224 MB which a kernel with all of its processes together can use/manage.

Functions to build page table (1)

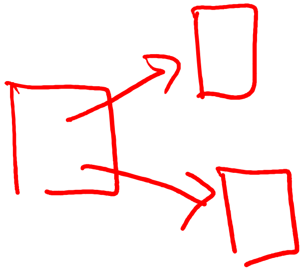
- Every page table begins with setting up kernel mappings in setupkvm()
- Outer pgdir allocated
- Kernel mappings defined in “kmap” added to page table by calling “mappages”
- After setupkvm(), user page table mappings added




```
// 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) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```



Functions to build page table (2)

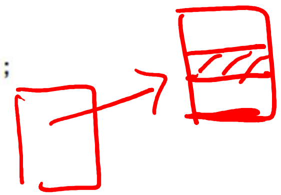
- Page table entries added by “mappages” actually assigns physical page frame to the pte.
  - Arguments: page directory, range of virtual addresses, physical addresses to map to, permissions of the pages
  - For each page, walks page table, get pointer to PTE via function “walkpgdir”, fills it with physical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

VA → PA

2 level page table, pde: page directory entry
pte: page table entry.

```
// 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;
}
```



```
// 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)];
}
```

allocate

Fork: copying memory image

```
// Copy process state from proc.
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
  kfree(np->kstack);
  np->kstack = 0;
  np->state = UNUSED;
  return -1;
}
```

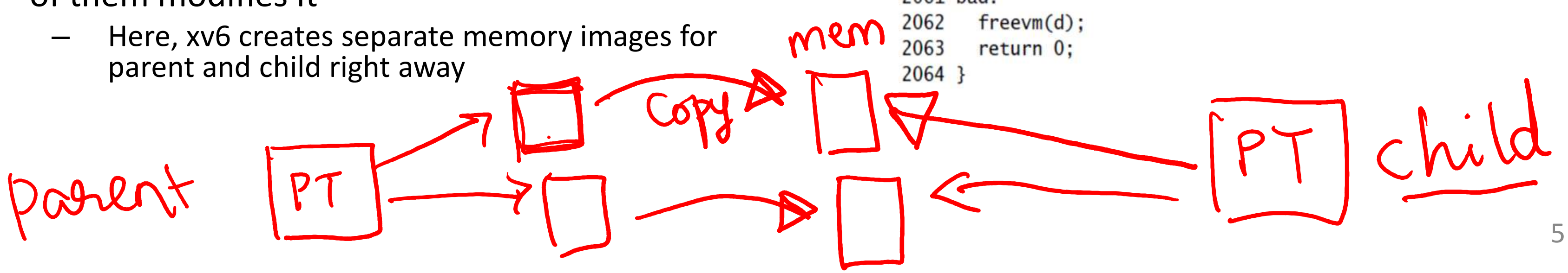
- Function “copyuvm” called by parent to copy parent memory image to child
  - Create new page table for child
  - Walk through parent memory image page by page and copy it to child, while adding child page table mappings
- For each page in parent
  - fetch PTE, get physical address, permissions
  - 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

```
// 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;
  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;
    }
  }
  return d;

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

copy the contents from parent's phy frame to child's phy frame.

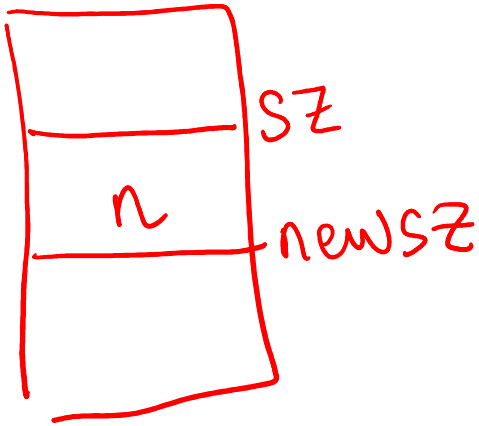


Growing memory image: sbrk

- 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) This step is important.


```
int
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```



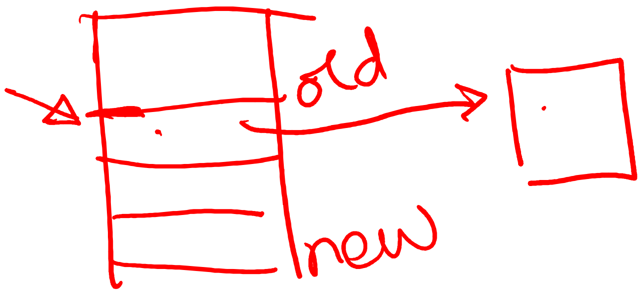
allocuvm: grow address space

- 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

```
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();
    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;
    }
  }
  return newsz;
}
```



Need to go through its implementation again.

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
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pgdir = setupkvm()) == 0)
    goto bad;
```

```c
  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  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

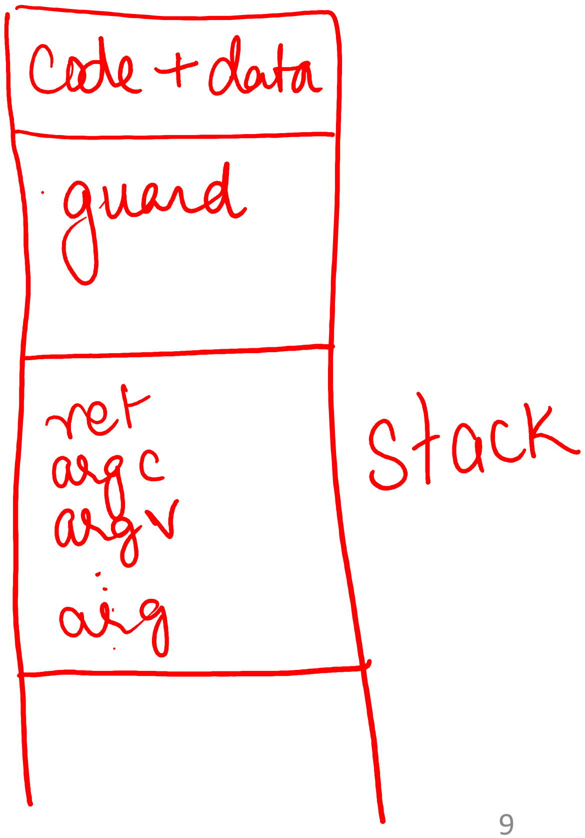
```
// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff;  // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
  goto bad;
```

argc
argv



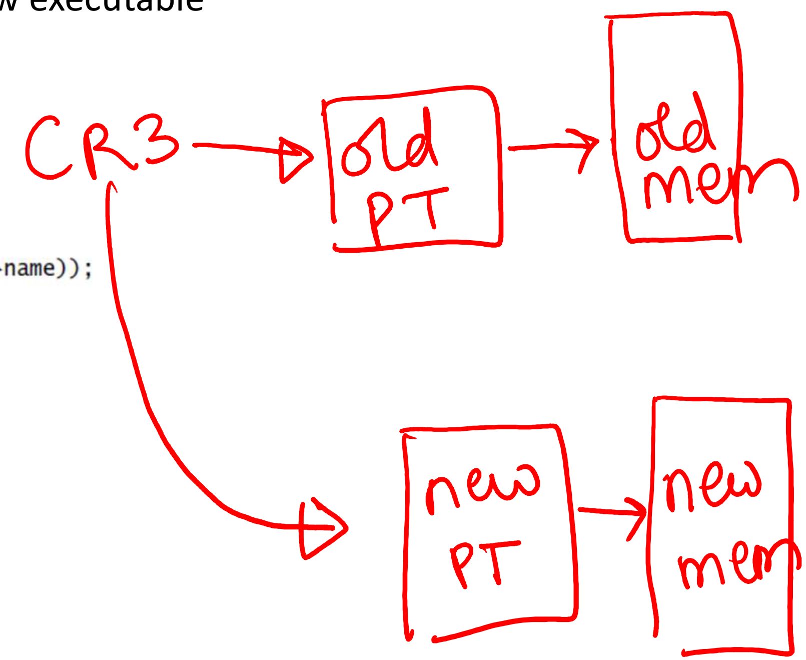
Exec system call (3)

- 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

```
  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(curproc->name, last, sizeof(curproc->name));

  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```



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