Memory management system call implementation in xv6

Mythili Vutukuru

CSE, IIT Bombay

Memory allocation to user processes in xv6

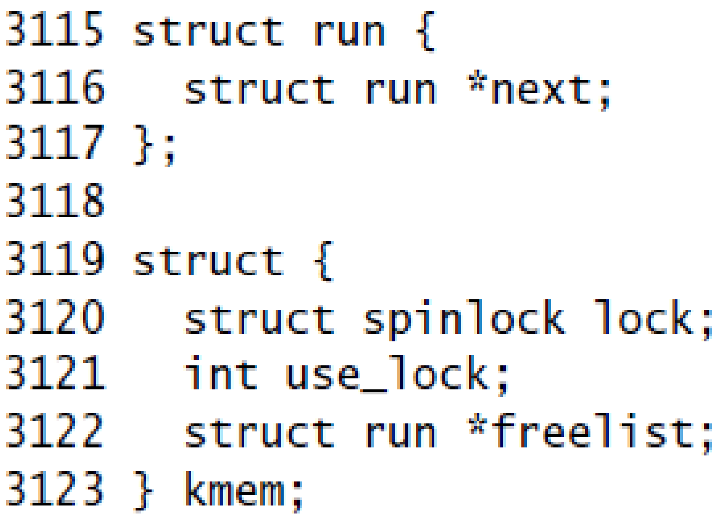
- How does OS implement various memory system calls?
- Understand how OS allocates memory to user processes using xv6 as example (real life OS have a much more complicated story)
- xv6 OS maintains list of free pages available in DRAM
  - All memory in [0, PHYSTOP] not used by OS is added to free list
- Whenever new process memory image (address space) needs to be created during system calls (e.g., fork, exec), OS allocates pages from free list to process, updates its page table
- Processes can also request more pages from OS using sbrk system call
  - sbrk invoked by malloc to expand heap
  - xv6 has no mmap system call

Memory allocation in the kernel

- OS needs memory for its data structures, must allocate it from its free pages only
- For large allocations, OS allocates a page for itself
- For smaller allocations, OS implements its own versions of slab allocator or buddy allocator
    - Cannot use libc and malloc in kernel!
    - Slab allocator for common data structures, e.g., slab of PCBs
    - Buddy allocator for variable sized allocations
    - OS does not use very general variable sized allocation for efficiency reasons
- xv6 uses only page sized allocations, other data structures are fixed size. e.g., ptable

Maintaining free memory in xv6

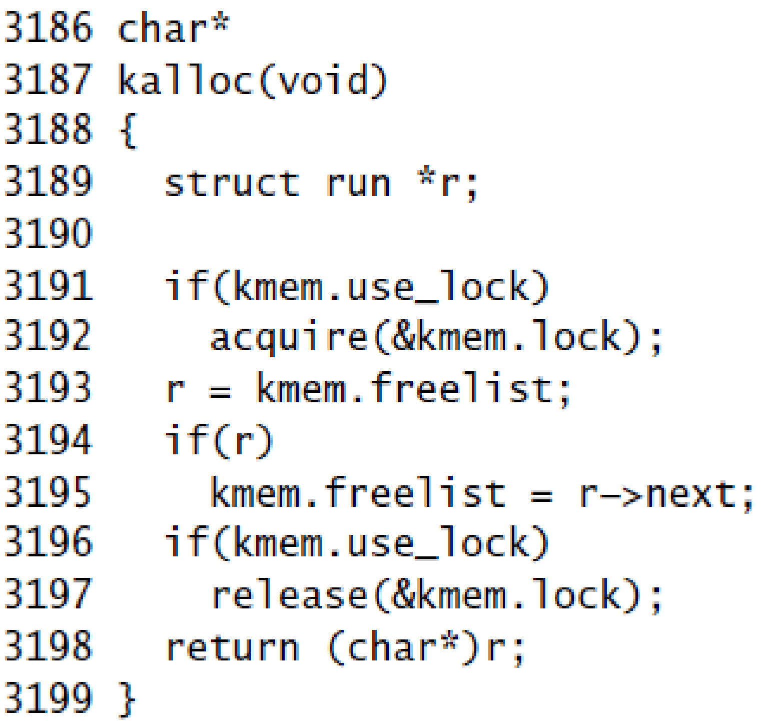
- After boot up, RAM contains OS code/data and free pages (physical memory frames)
- OS collects all free pages into a free list, so that they can be allocated to user processes
- Free list is a linked list, pointer to next free page embedded within previous free page
- Kernel maintains pointer to first page in the free list
- Pages from free list allocated for code/data/stack/heap as well as page table of process



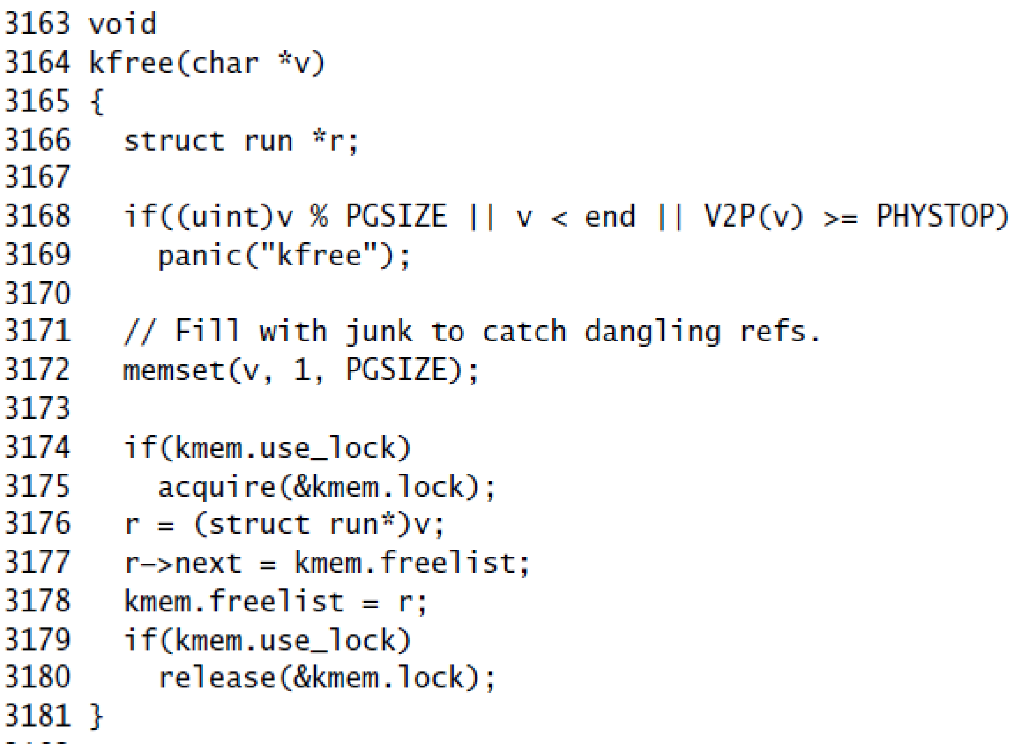
```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
```

Managing free pages in xv6: kalloc and kfree

- Anyone who needs a free page calls kalloc()
  - Sets free list pointer to next page and returns first free page on list
- When memory needs to be freed up, kfree() is called
  - Add free page to head of free list, update free list pointer



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

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```



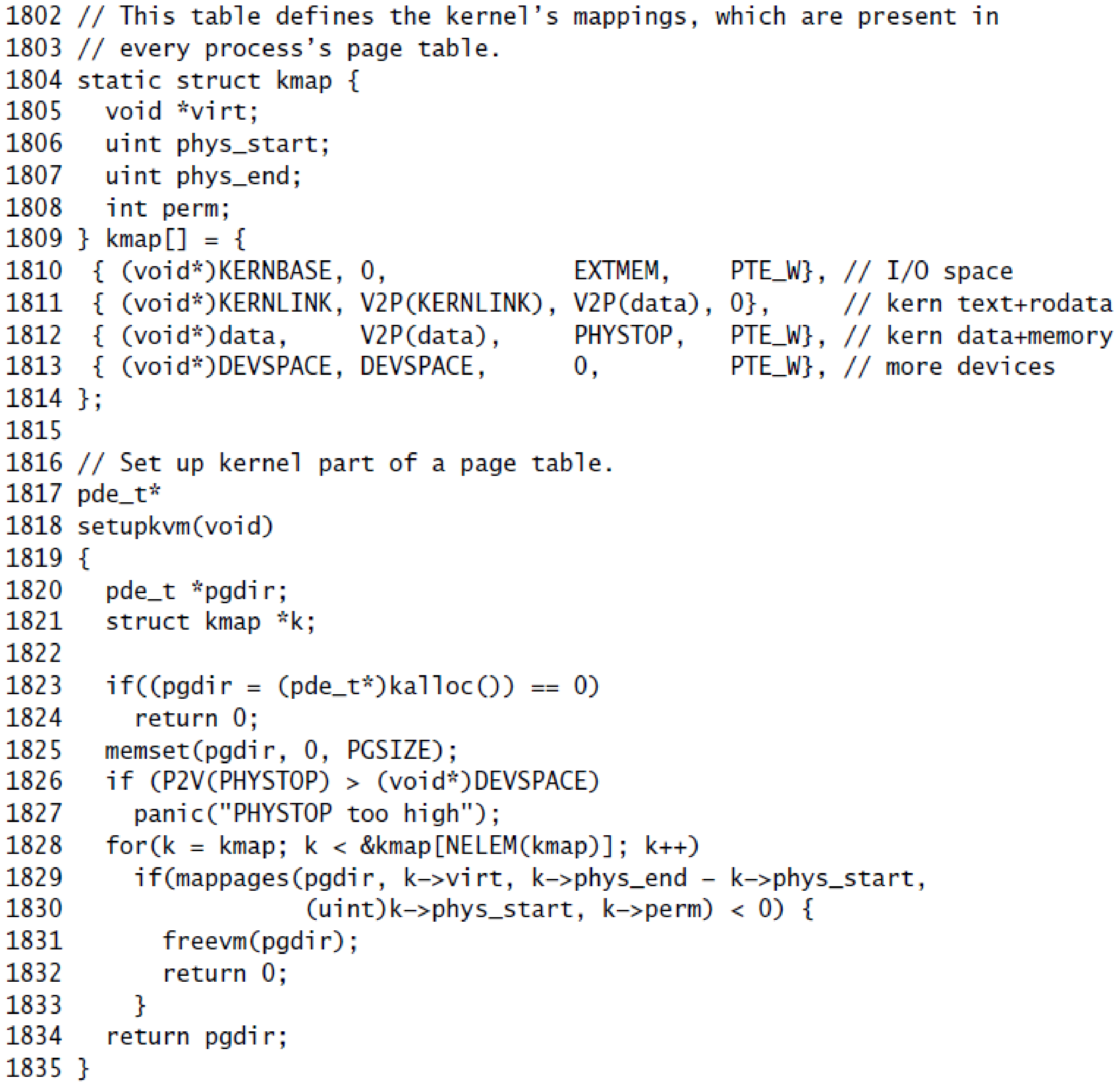
```
void
kfree(char *v)
{
  struct run *r;

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

Allocating memory to user processes

- System calls like fork, exec allocate memory from OS via kalloc()
- How is address space of process constructed in fork/exec?
  - Start with one page for the outer page directory of child
  - Allocate inner page tables as needed (if contains valid entries)
  - Add page table mappings for kernel code/data (starting at 2GB)
  - Allocate physical frames to store memory contents of process (code/data from executable, empty stack, ..) and map these into page table
- How is address space of process expanded in sbrk?
  - Allocate physical frames for new virtual addresses
  - Add mappings for newly allocated pages in page table

Functions to build page table (1)

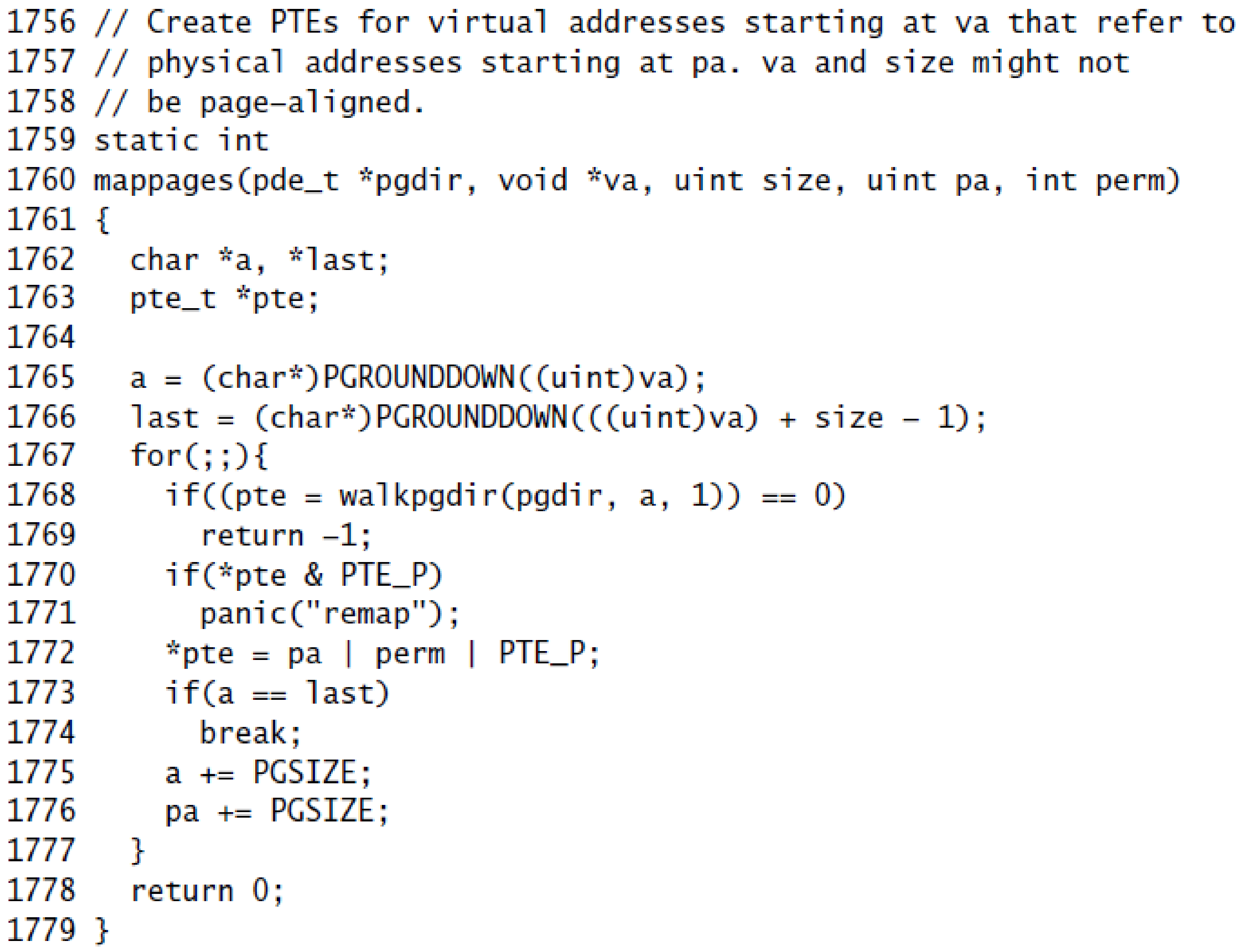
- Every page table begins with setting up kernel mappings in setupkvm()
- Outer pgdir allocated
- Kernel mappings defined in “kmap” added to page table by calling “mappages”
- After setupkvm(), user page table mappings added



```
// 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)

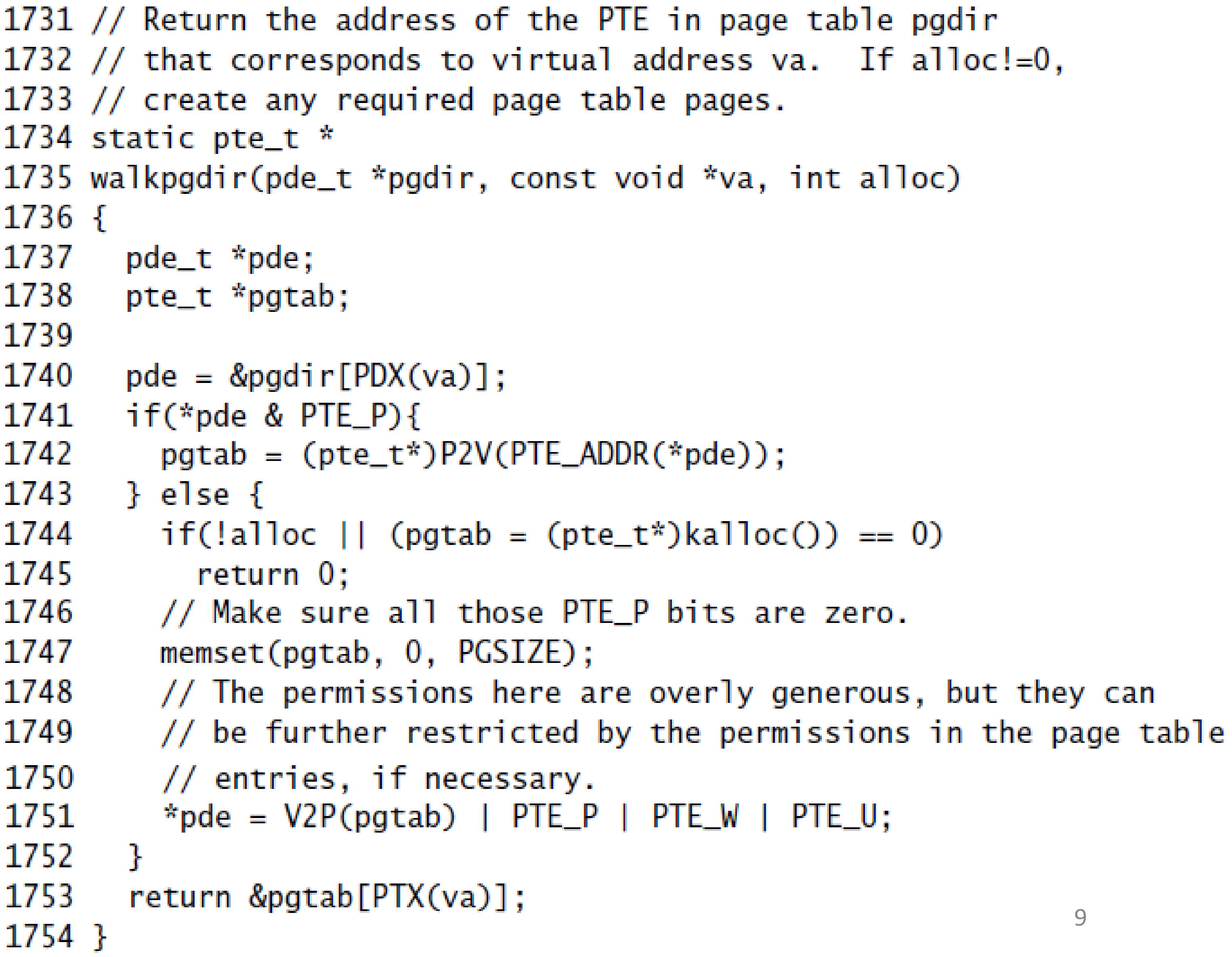
- Page table entries added by "mappages"
  - Arguments: page directory, range of virtual addresses, physical addresses to map to, permissions of the pages
  - For each page, walks page table, get pointer to PTE via function "walkpgdir", fills it with physical address and permissions



```
// 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;
}
```

Functions to build page table (3)

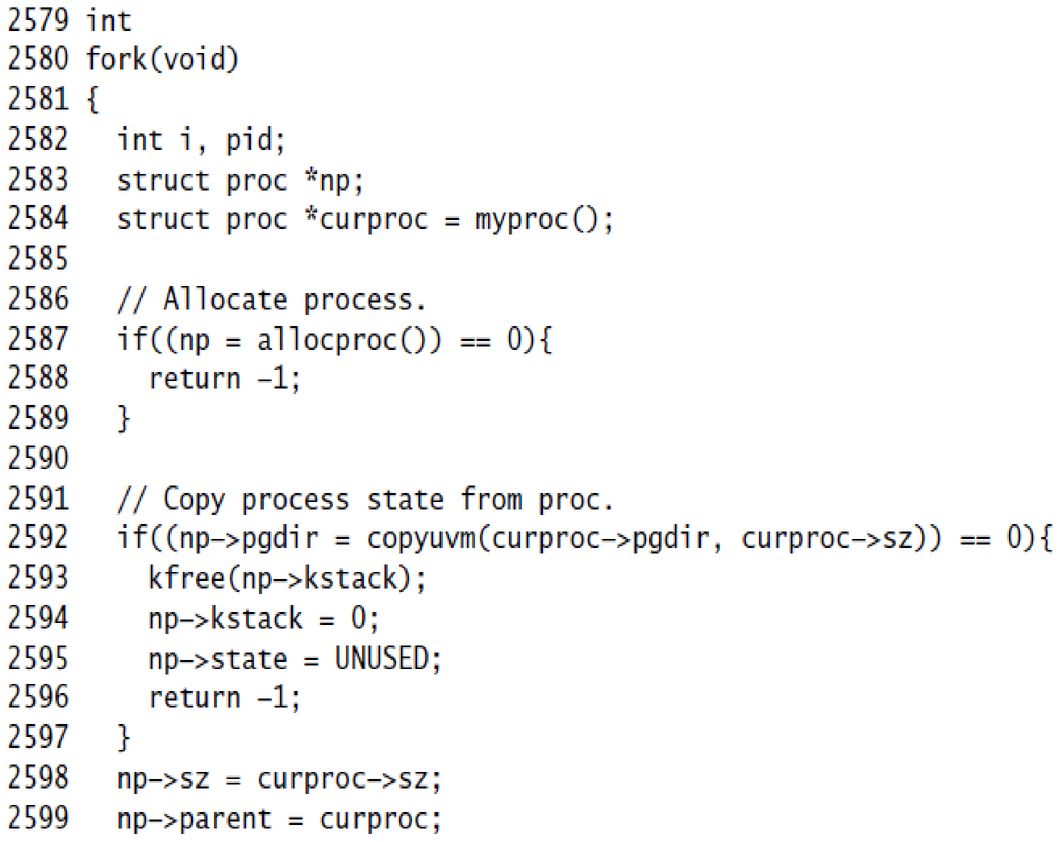
- Function “walkpgdir” walks page table, returns PTE of a virtual address
- Can allocate inner page table if it doesn’t exist (depending on value of last arg)



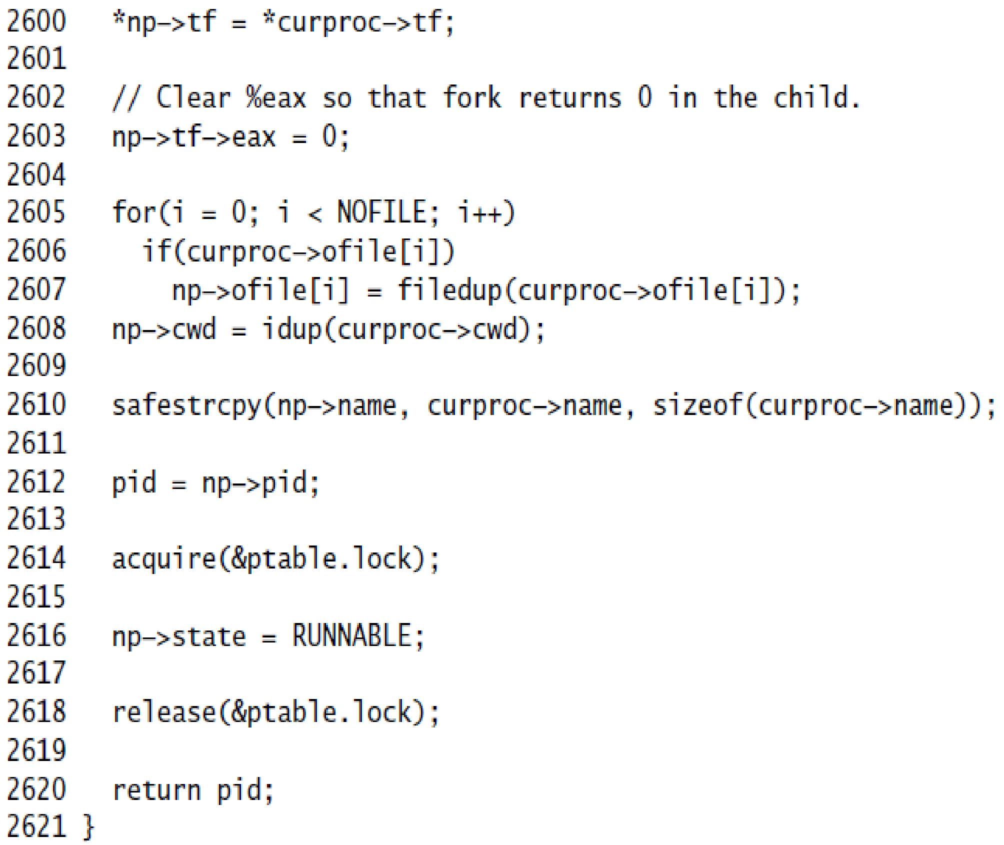
```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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)];
}
```

Recap: fork system call implementation

- Parent allocates new process in ptable, copies parent state to child
- Child process set to runnable, scheduler runs it at a later time
- Return value in parent is PID of child, return value in child is set to 0



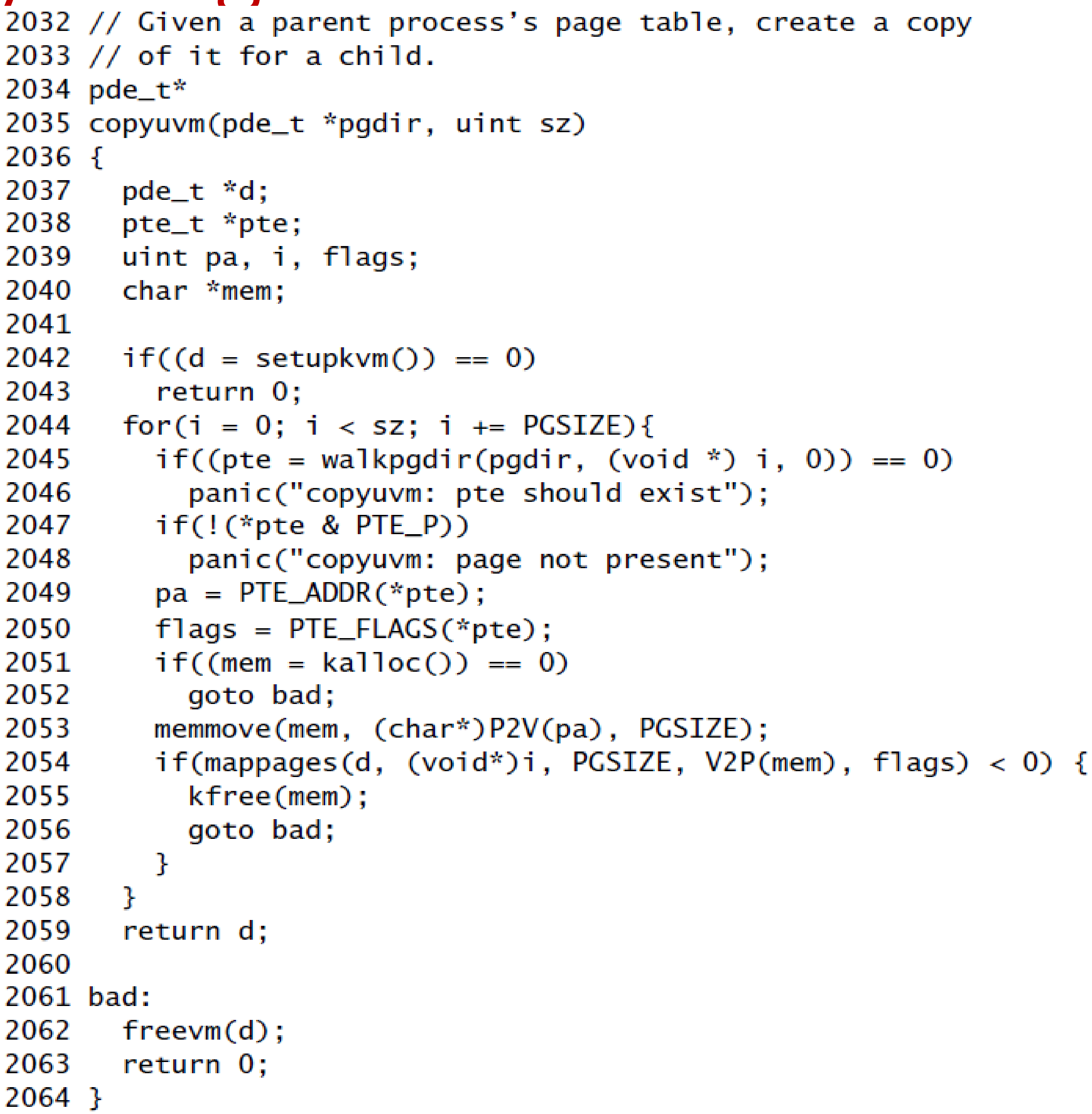
```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // 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;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
```



```
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

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
- For each page in parent
  - Fetch PTE, get physical address, permissions
  - Allocate new frame for child, 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



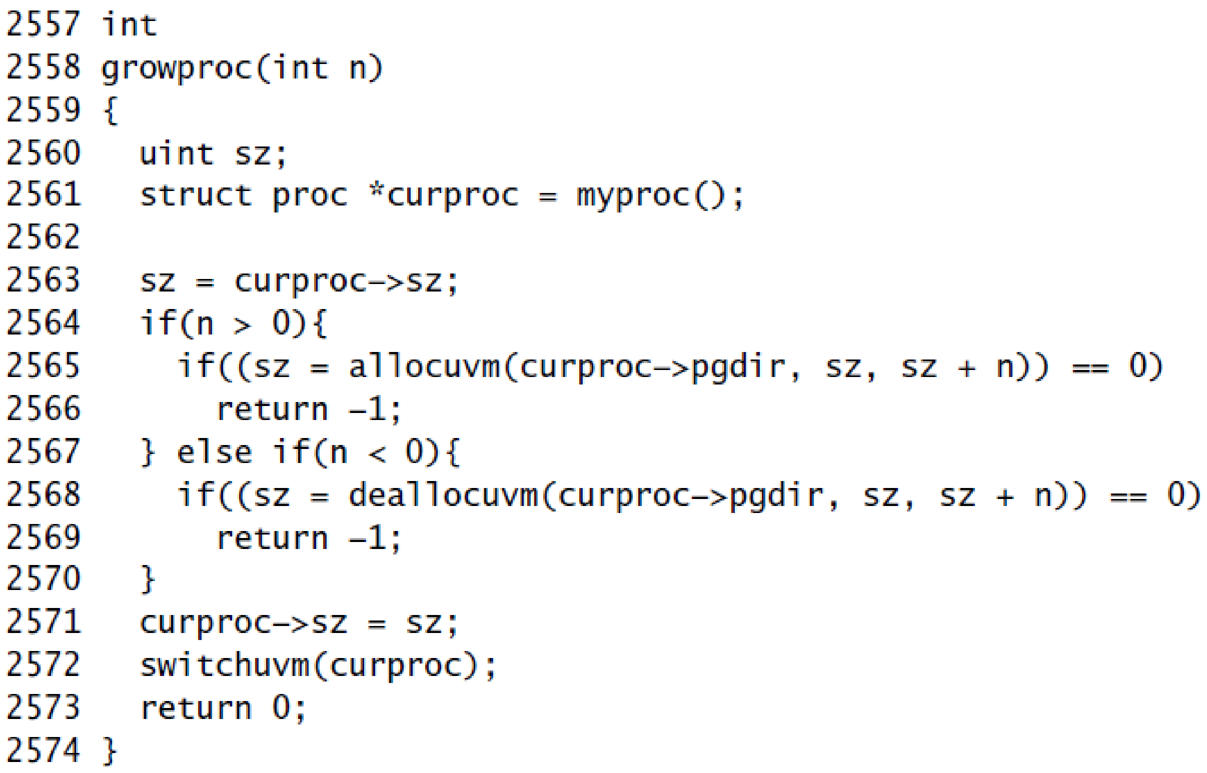
```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  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-on-write fork

- 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
- Copy-on-write fork (not present in xv6, but easy to do):
    - During fork, new page table allocated to child
    - Child page table entries have physical frame numbers of parent memory image pages only, no copy created for child
    - Parent’s memory image is marked as read only
    - When parent or child tries to modify, MMU traps to OS
    - As part of trap handling, separate copy of memory image created
    - Finally, two separate copies of memory image for parent and child

Growing memory image: sbrk

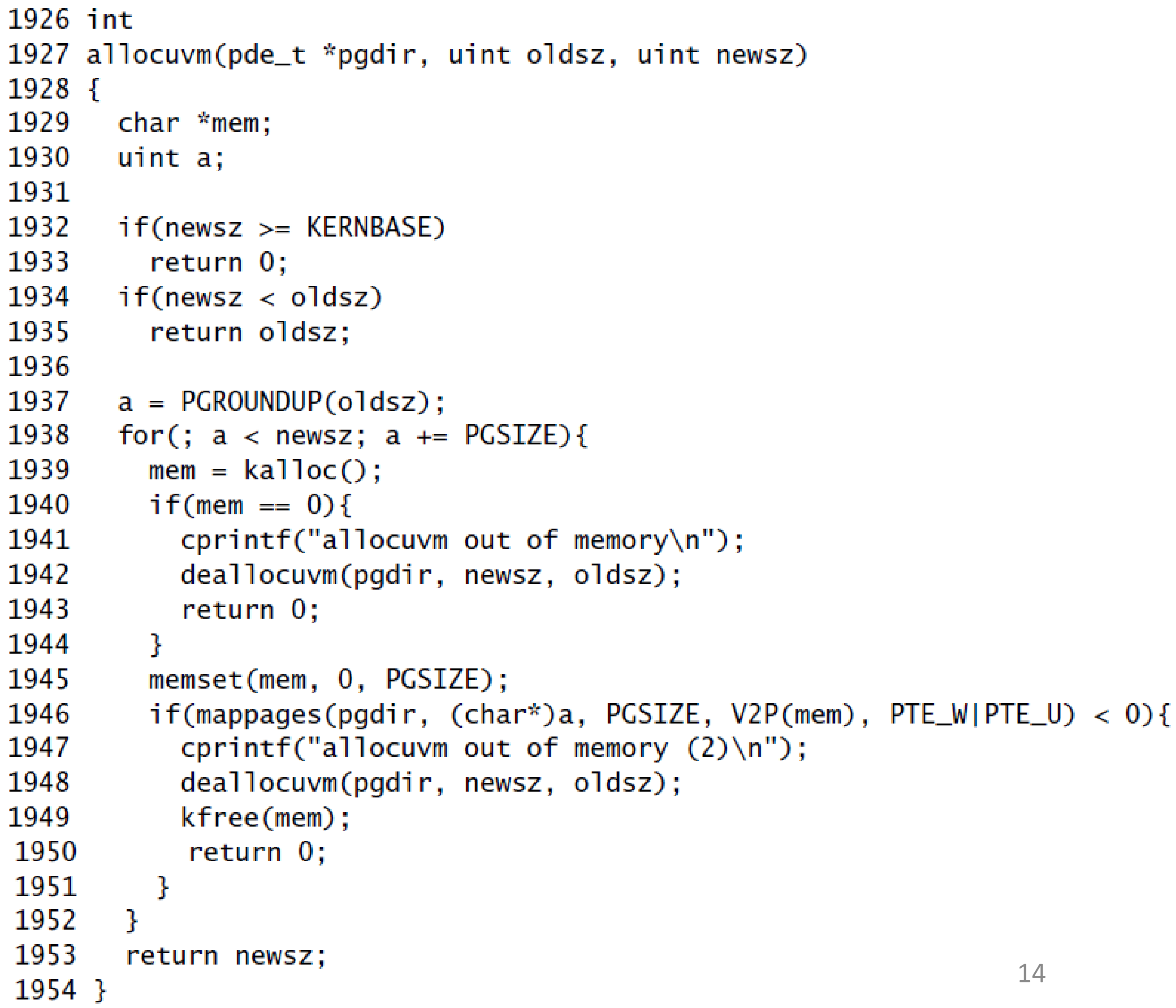
- Initially heap is empty, program “break” is at end of stack
    - sbrk() system call invoked by malloc to expand heap
    - Calls “growproc” to grow memory
- To grow memory, allocuvm allocates new pages, adds mappings into page table for new pages
- Whenever page table updated, must update cr3 register and TLB (done even during context switching)



```
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

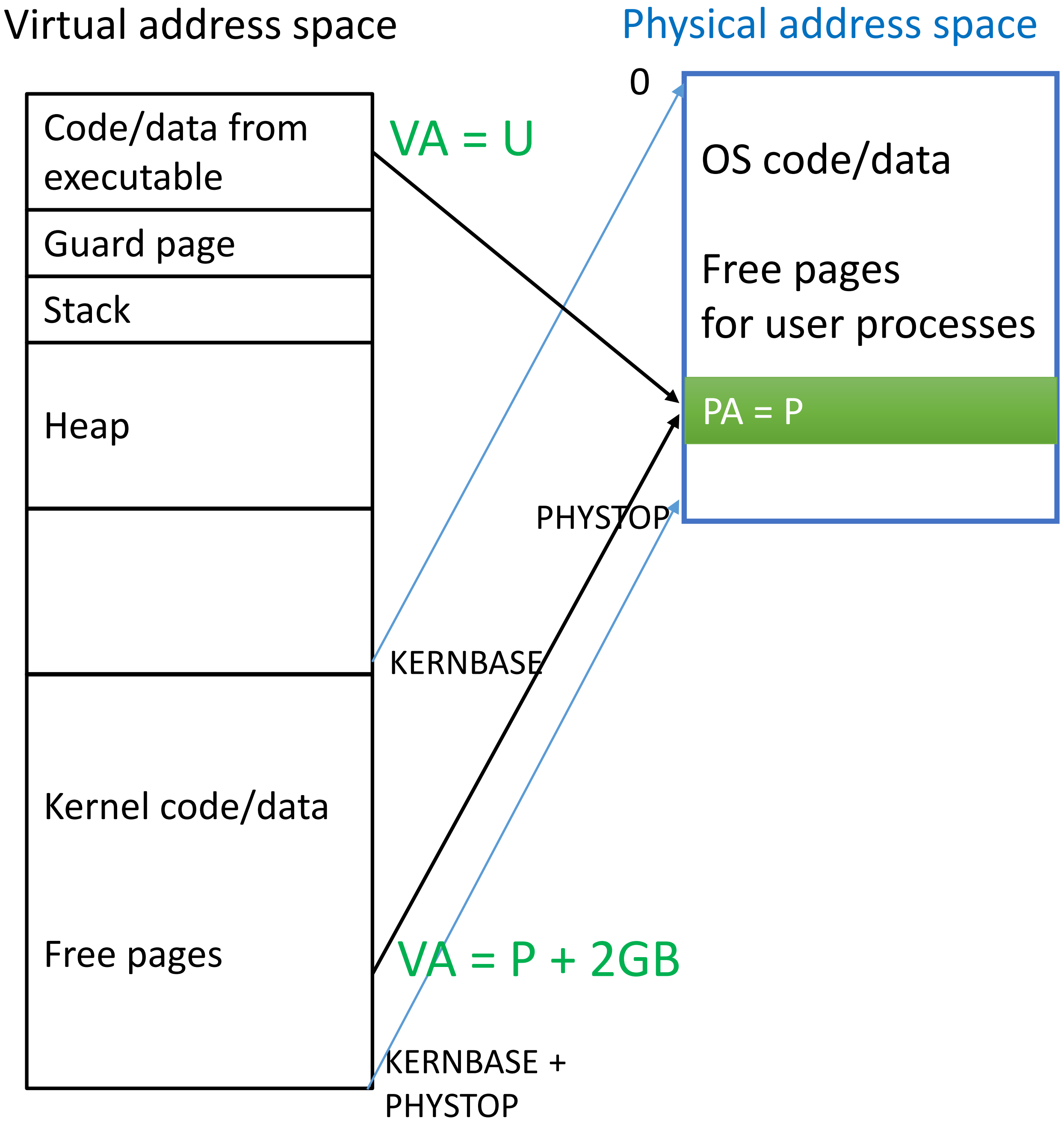
- Walk through new virtual addresses, page by page
- Allocate new frame, add mapping to page table with suitable user permissions
- Similarly deallocuvm shrinks memory image, frees up pages



```c
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;
}
```

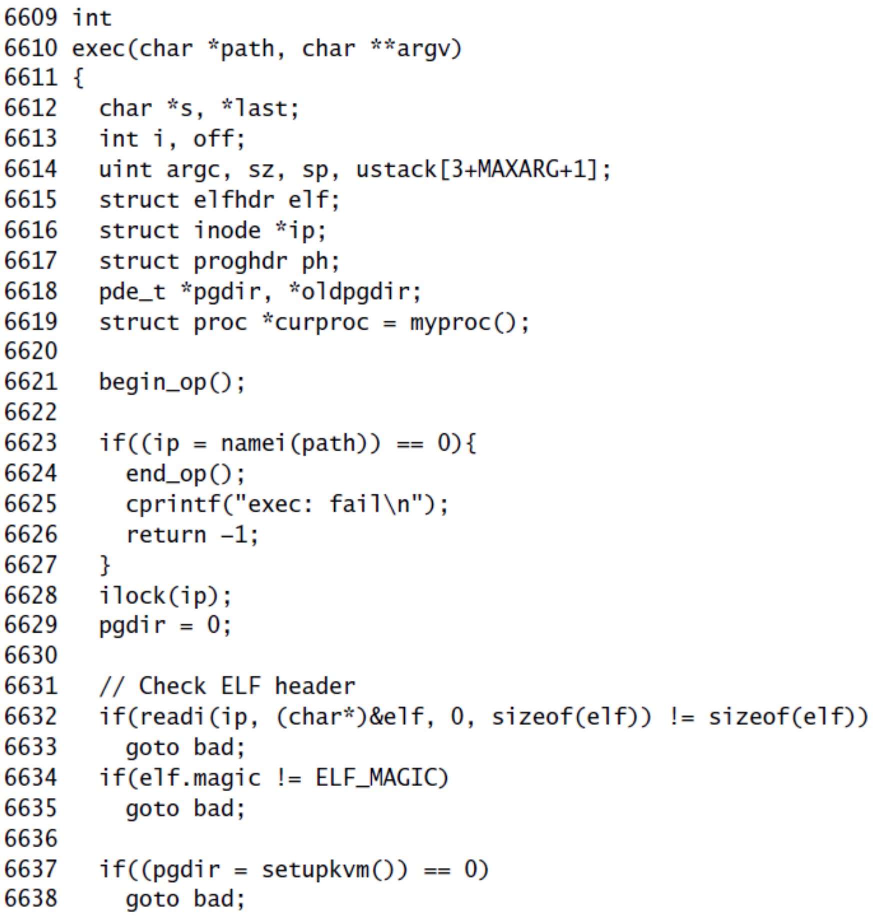
Maximum addressable memory in xv6

- PA=P is initially mapped into kernel address space at VA=P+2GB
- When assigned to user, P is assigned another VA=U (<2GB)
- Kernel and user access same memory using different virtual addresses
- Every physical address may be mapped to 2 virtual addresses in xv6
- Max virtual address is 4GB, so xv6 can only handle max physical address 2GB
- Real kernels deal with this better, e.g., remove kernel VA-PA mapping once assigned to user process

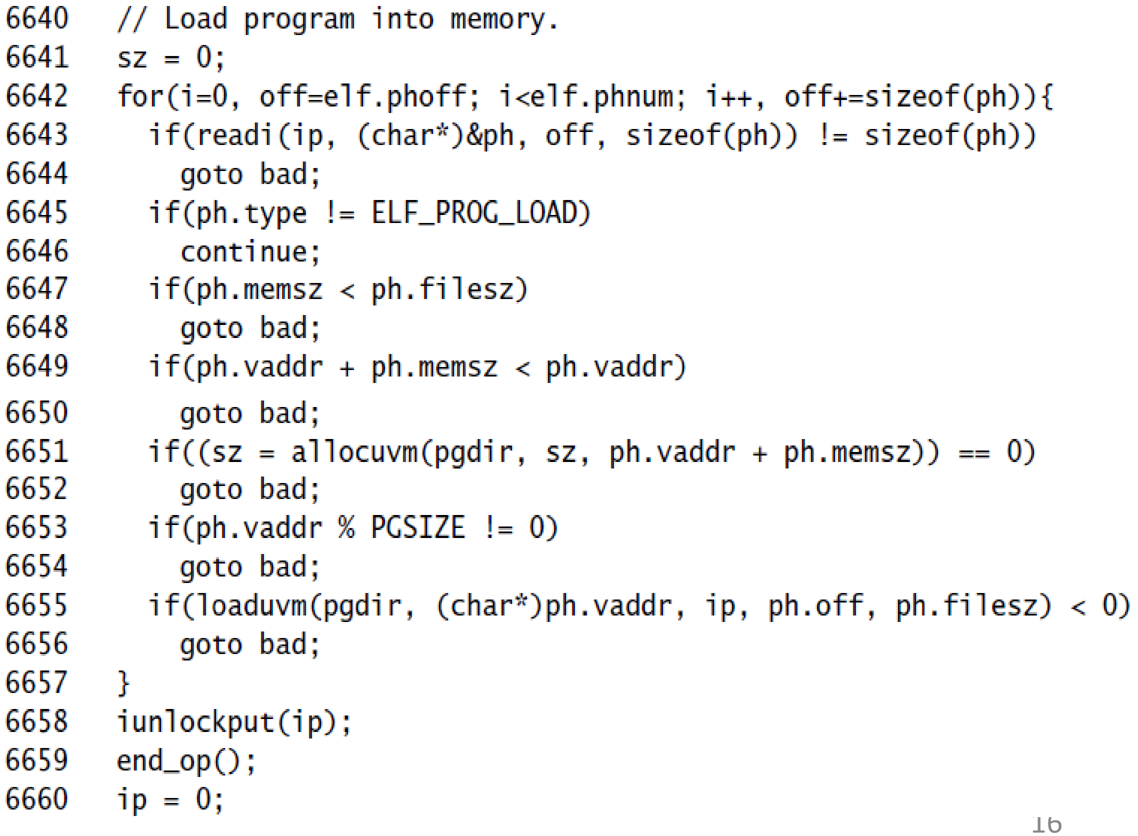


Exec system call (1)

- Read ELF binary file from disk into memory
- Start with new page table (not overwriting old page table)
- Use function “loaduvm” to load executable from disk to memory



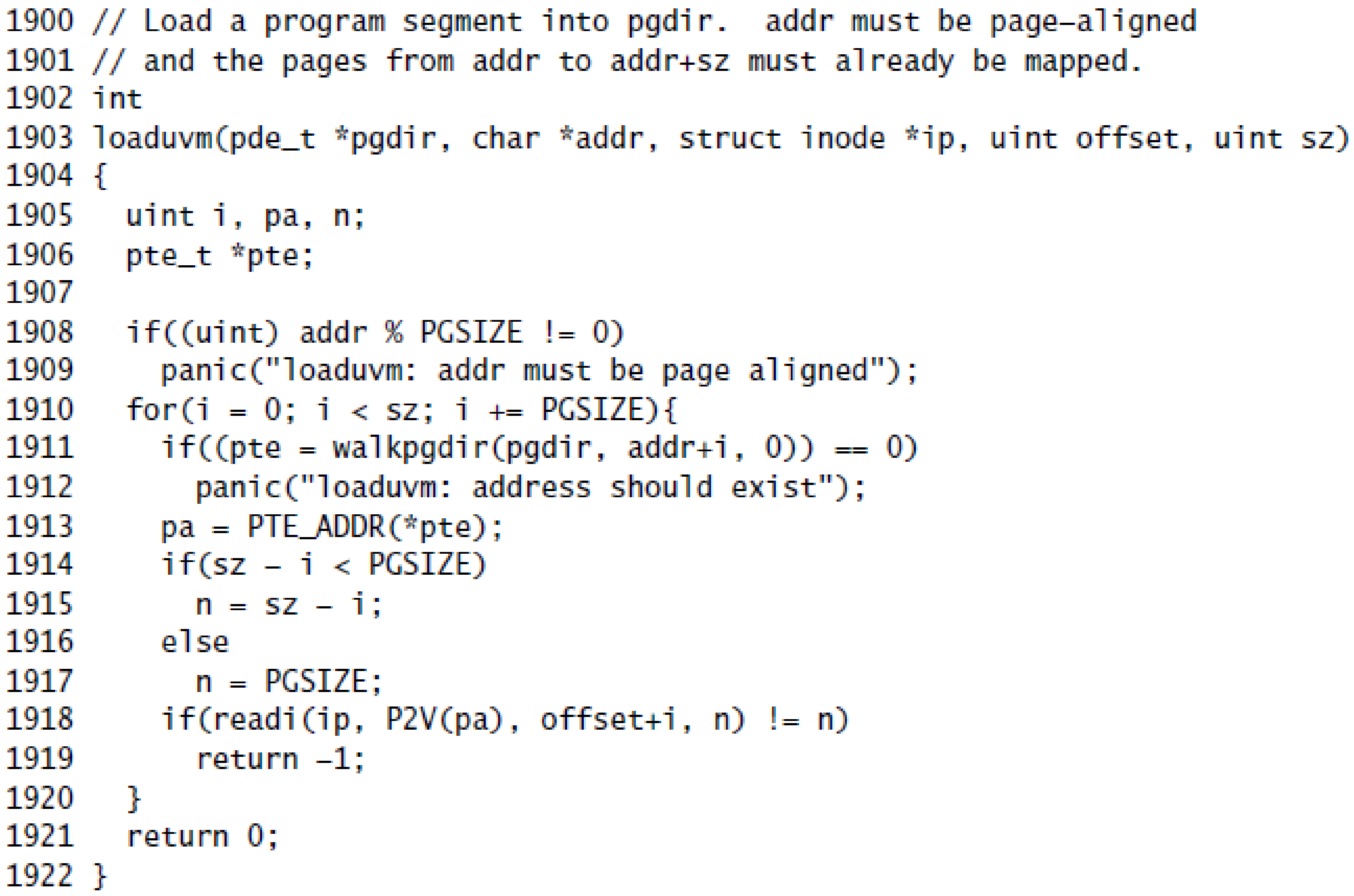
```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  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;
```



```
  // 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)

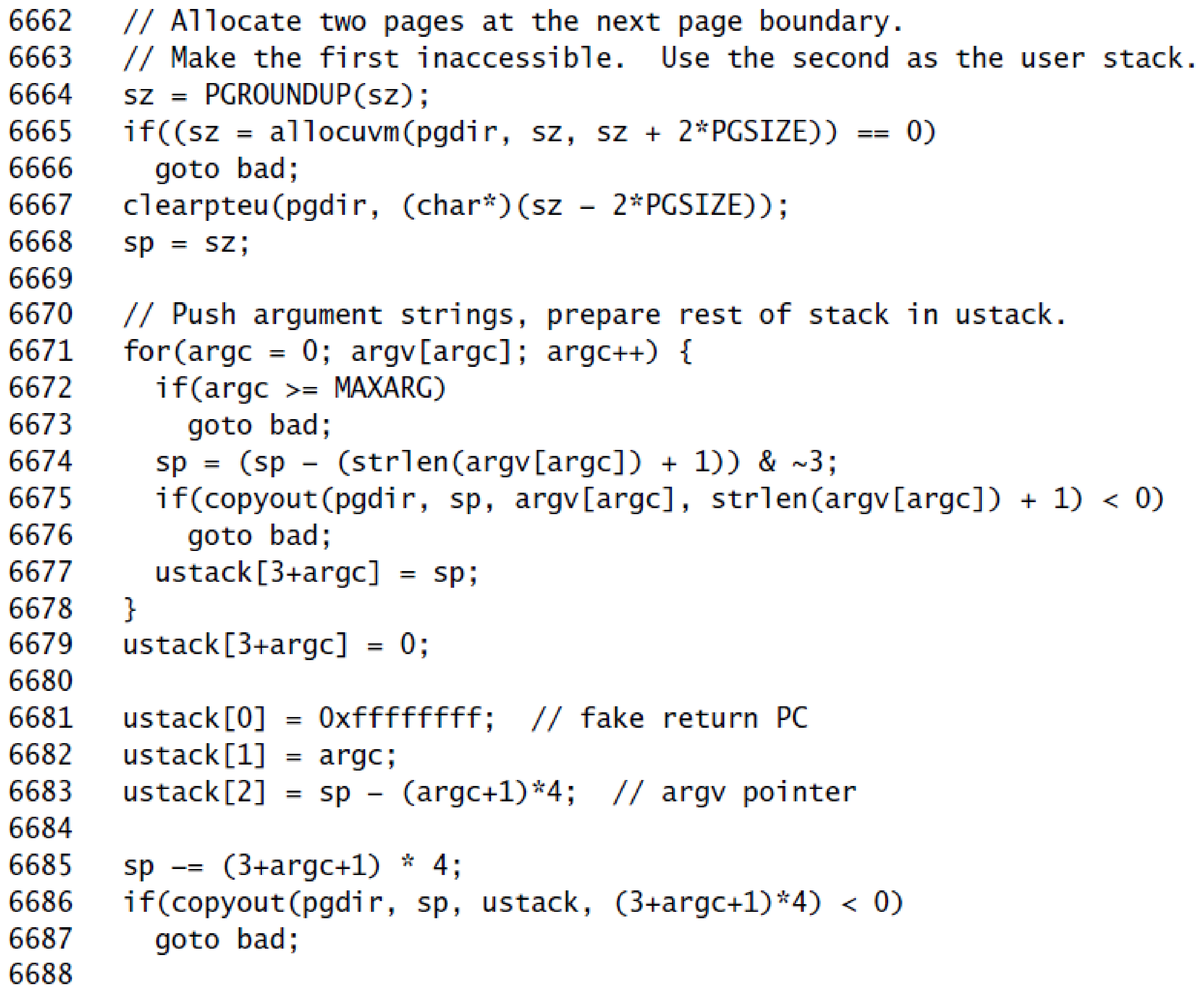
- Function allocuvm allocates new memory frame, updates page table entries
- Function loaduvm reads the corresponding part of executable from disk into the allocated memory frame
- Calls to allocuvm and loaduvm repeated for each segment of executable



```
// 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;
}
```

Exec system call (3)

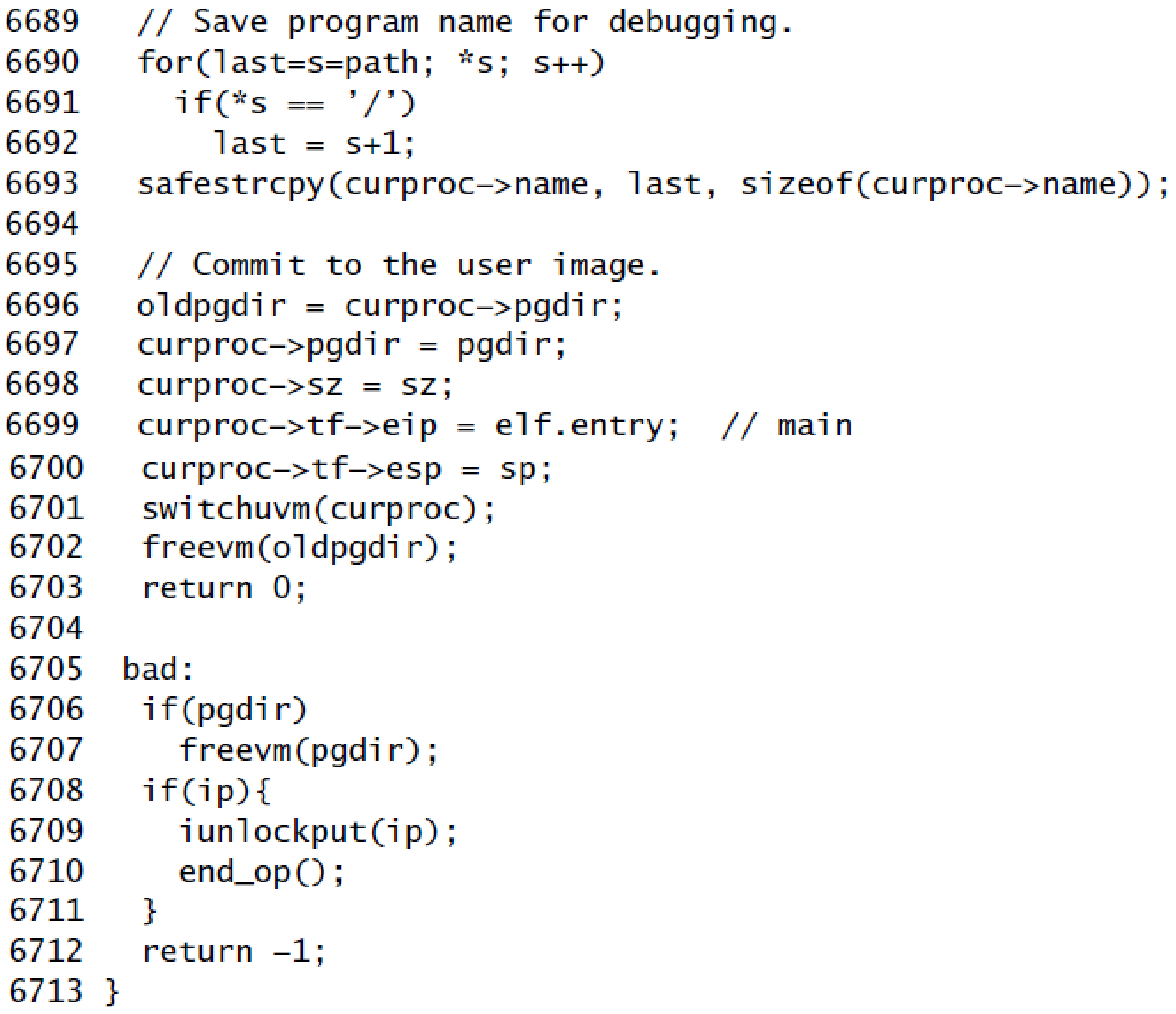
- After executable is copied to memory image, allocate 2 pages for stack (one is guard page, permissions cleared, access will trap)
- Push exec arguments onto user stack for main function of new program



```
// 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;
```

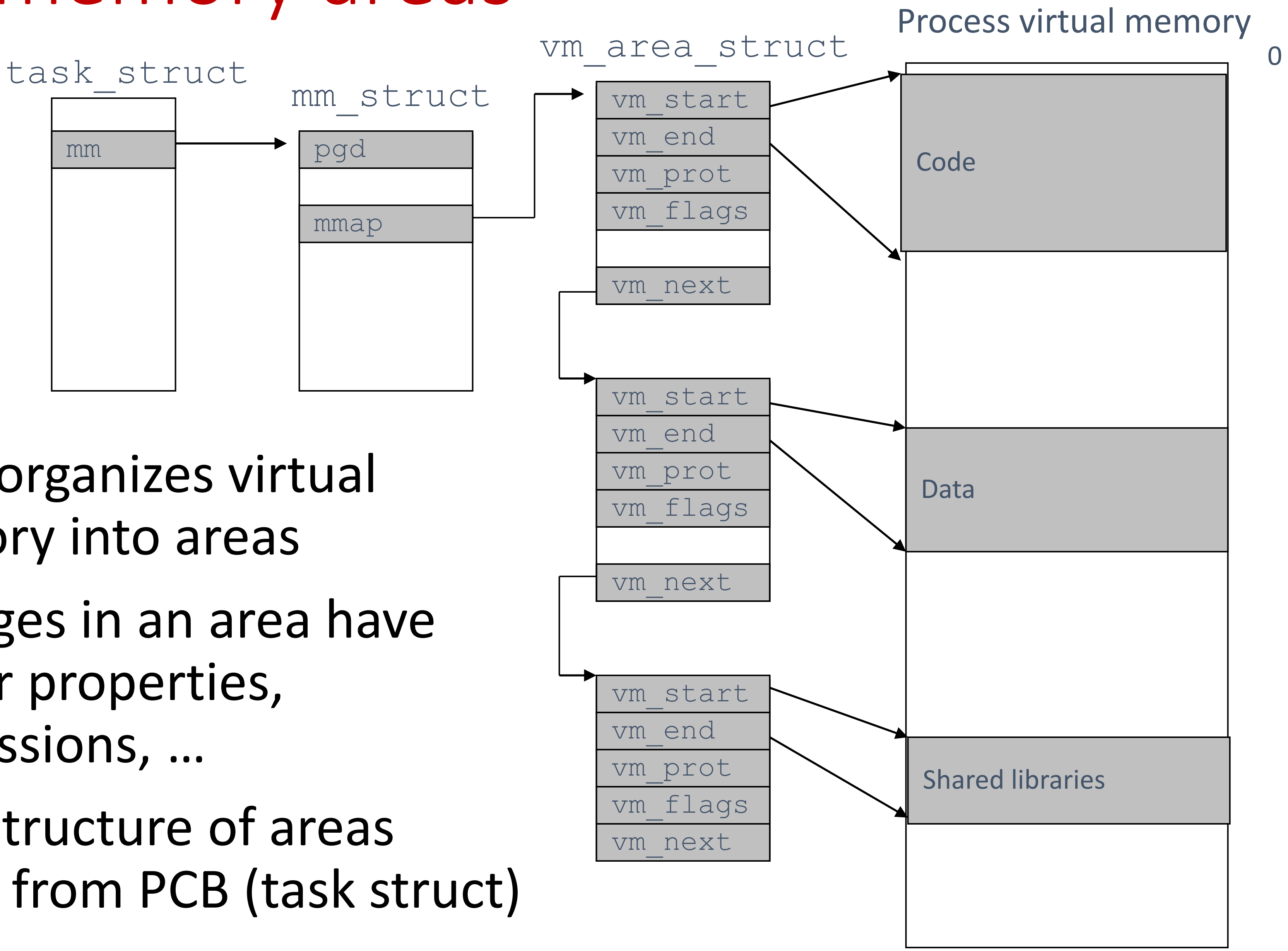
Exec system call (4)

- 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;
}
```

Linux memory areas



- Linux organizes virtual memory into areas
- All pages in an area have similar properties, permissions, …
- Data structure of areas linked from PCB (task struct)

Image credit: CSAPP