CS 347M (Operating Systems Minor)

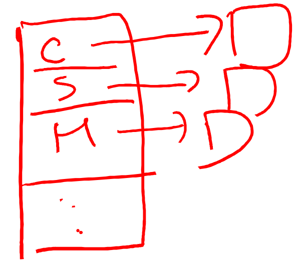
Spring 2022

Lecture 11: Memory management in xv6

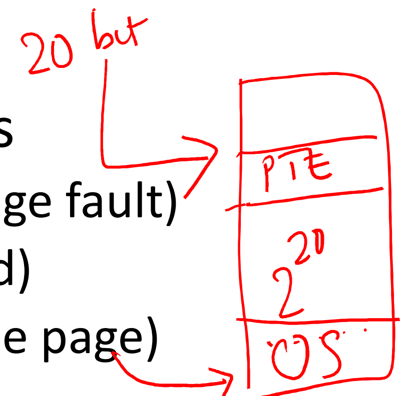
Mythili Vutukuru

CSE, IIT Bombay

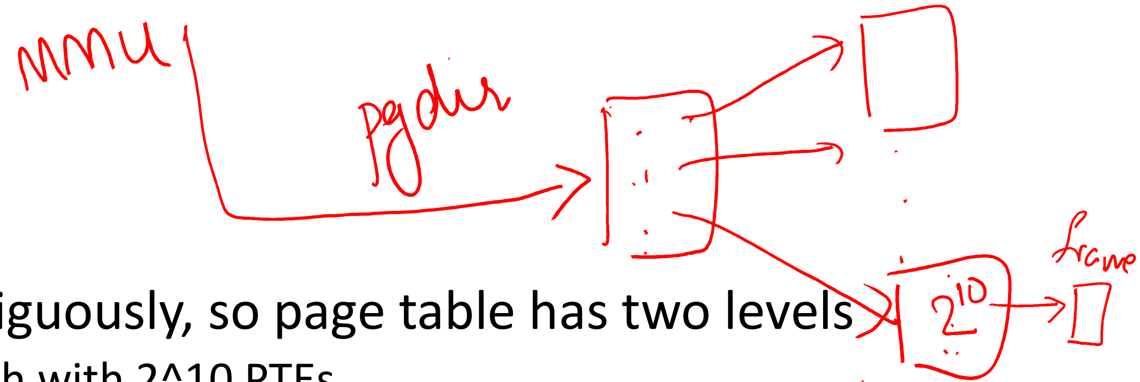
Virtual memory in xv6



- 32-bit OS, so 2^32=4GB virtual address space for every process
- 4KB pages, so 32 bit VA = 20 bit page number + 12 bit offset
- No demand paging, all pages assigned frames always
- Each PTE has 20 bit physical frame number, and some flags



valid present
  - PTE_P indicates if page is present (if not set, access will cause page fault)
  - PTE_W indicates if writeable (if not set, only reading is permitted)
  - PTE_U indicates if user page (if not set, only kernel can access the page)
- Address translation: use page number (top 20 bits of virtual address) to index into page table, find physical frame number, add 12-bit offset



Two level page table

- 2^20 PTEs cannot be stored contiguously, so page table has two levels
  - 2^10 "inner" page table pages, each with 2^10 PTEs
  - Outer page directory stores PTE-like references to the 2^10 inner page table pages
  - Physical address of outer page directory is stored in CPU's cr3 register, used by MMU during address translation

x86

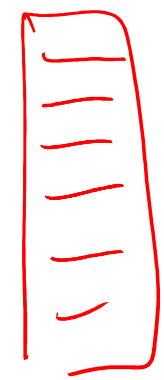
2^{10} inner

- 32 bit virtual address = 10 bits index into page directory, next 10 bits index into inner page table, last 12 bits are offset within page
  - PFN from PTE + offset = physical address

```
// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/
```

MMU

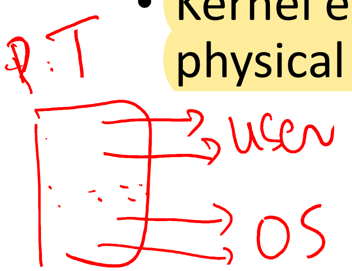
2^{20}

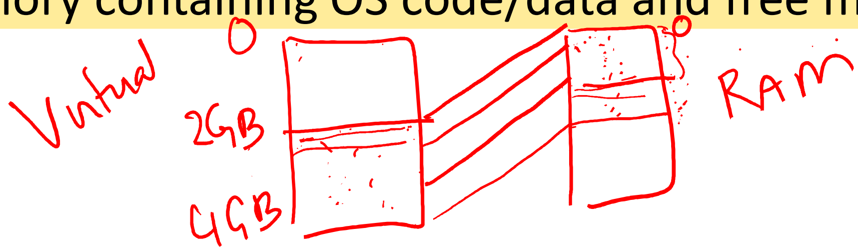


Process virtual address space in xv6

- Memory image of a process starting at address 0
  - Contains code/data from executable, one page stack (with extra, guard page), expandable heap
- Kernel code/data is mapped beginning at address KERNBASE (2GB)
  - Contains kernel code/data, and free pages maintained by kernel
- Page table of a process contains two sets of PTEs
  - User entries map low virtual addresses to physical memory used by the process for its code/data/stack/heap
  - Kernel entries map high virtual addresses (KERNBASE to KERNBASE+PHYSTOP) to physical memory containing OS code/data and free memory (0 to PHYSTOP)

0
exec
Stack
heap
2GB

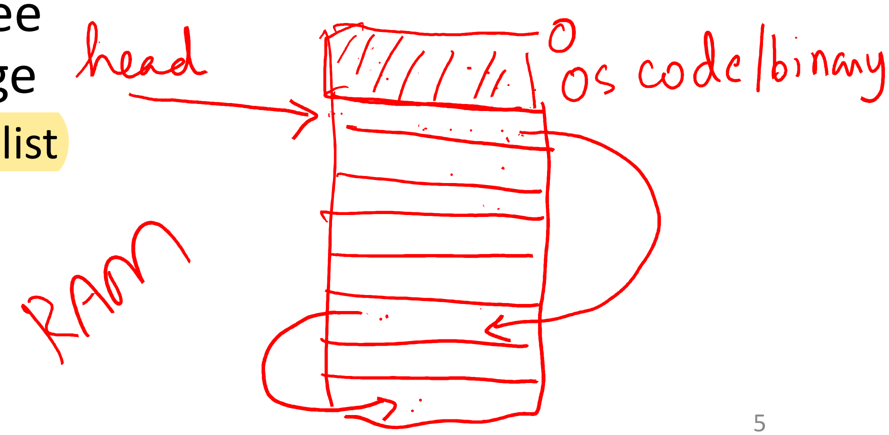




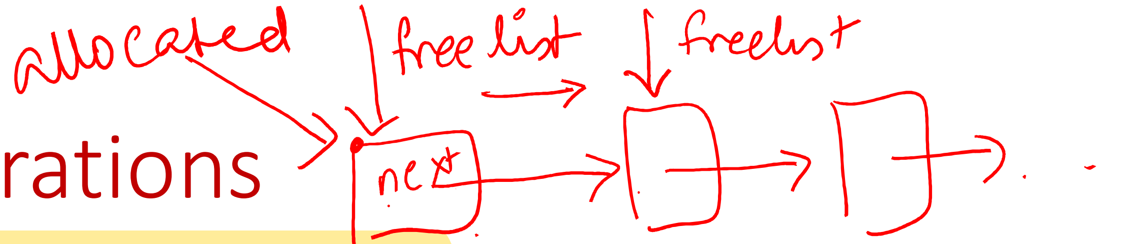
Maintaining free memory

- After boot up, RAM contains OS code/data and free pages
- OS collects all free pages into a free list, so that it can be assigned to user processes
  - Used for user memory (code/data/stack/heap) and page tables of 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 list

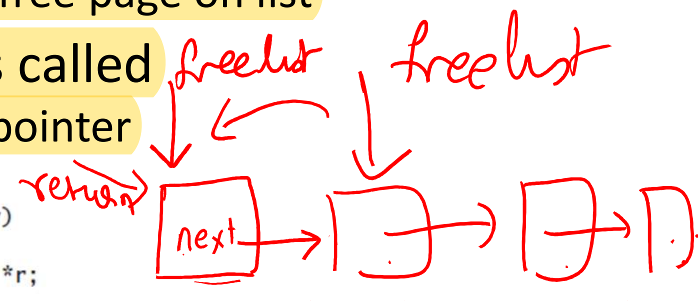
```
struct run {
  struct run *next;
};

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



alloc and free operations



- 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);
}
```

Memory management of user processes (1)

- 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)
- Note: only one copy of OS in memory, copied during boot time
- 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
  - Invoked by malloc to expand heap

init $\xrightarrow{\text{fork}}$ Shell

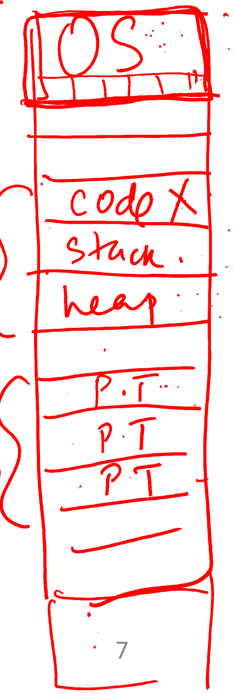
fork

command

RAM

contents of process

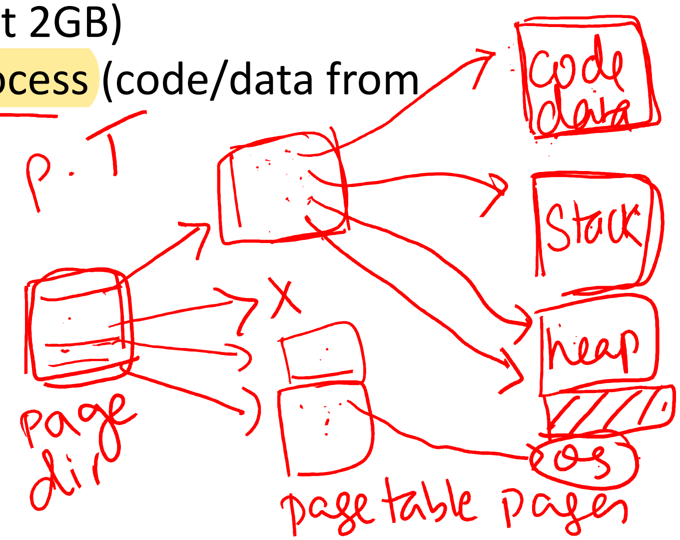
P.T of process



Memory management of user processes (2)

- Across all system calls, memory pages obtained from OS via kalloc()
- How is address space of process constructed in fork/exec?
  - 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)
  - 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, ..)
  - Add mappings for user pages in 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

mem img



ret = fork()

Recap: fork system call implementation

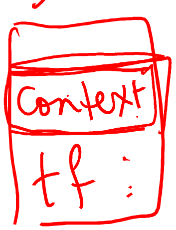
- Parent allocates new process in ptable, copies parent state to child
- Child process set to runnable, scheduler runs it at a later time
- Return value in parent is PID of child, return value in child is set to 0

P -> C

P pid of child

Kernel mode

Kernel stack

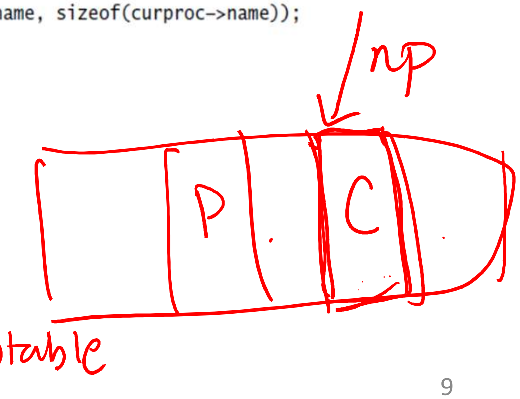



```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // 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;
}
```

memory copy

copy trapframe

files



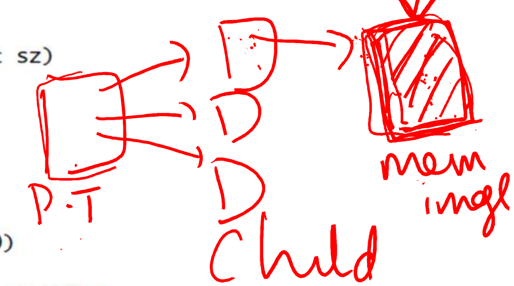
Parent

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

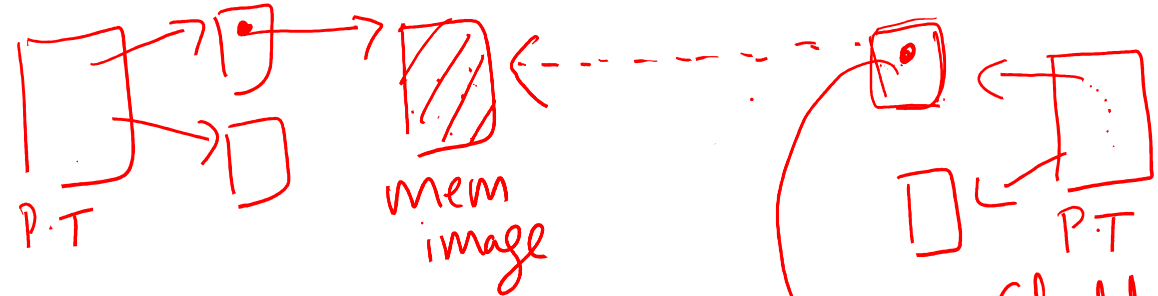
```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  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;
}
```

OS entries



Parent

Copy-on-write fork



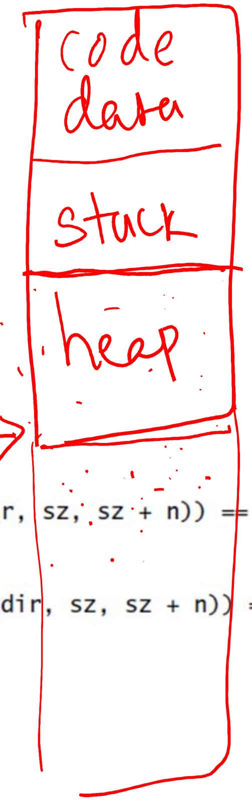
- Real operating systems do copy-on-write: child page table also points to parent pages until either of them modifies it
  - Here, xv6 creates separate memory images for parent and child right away
- Copy-on-write fork (not present in xv6, but easy to do):
  - During fork, new page table allocated to 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" (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)

```
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;
}
```

break



update pagetable

MMU

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

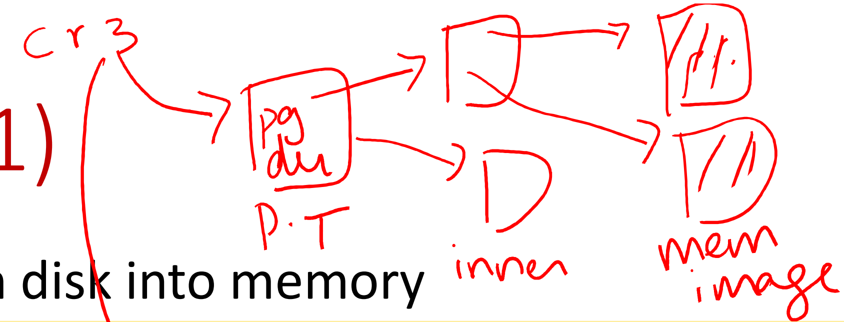
```
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  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;
}
```

zero

old

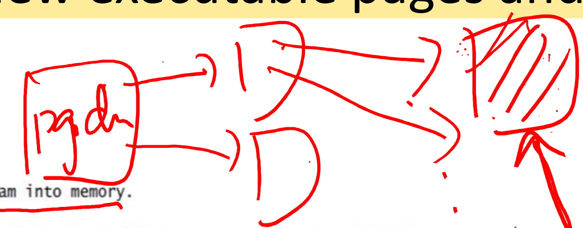
newsize

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

Switch




```
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;
```

read from disk into memory & add PTE

Start new pgdir

copy new exec from disk

exec(comm-d, "__", "__")

a.out

code

stack

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

main(argc ~~argv~~)

local

```
// 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;
```

2 pages stack

command line argumen

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;
}
```

switch to new pagetable

free up old memory

eip → exec( )

eip → main{

}

old code

new