# 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

```
3115 struct run {
3116   struct run *next;
3117 };
3118
3119 struct {
3120   struct spinlock lock;
3121   int use_lock;
3122   struct run *freelist;
3123 } 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

```
3186 char*
3187 kalloc(void)
3188 {
3189   struct run *r;
3190
3191   if(kmem.use_lock)
3192     acquire(&kmem.lock);
3193   r = kmem.freelist;
3194   if(r)
3195     kmem.freelist = r->next;
3196   if(kmem.use_lock)
3197     release(&kmem.lock);
3198   return (char*)r;
3199 }
```

```
3163 void
3164 kfree(char *v)
3165 {
3166   struct run *r;
3167
3168   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3169     panic("kfree");
3170
3171   // Fill with junk to catch dangling refs.
3172   memset(v, 1, PGSIZE);
3173
3174   if(kmem.use_lock)
3175     acquire(&kmem.lock);
3176   r = (struct run*)v;
3177   r->next = kmem.freelist;
3178   kmem.freelist = r;
3179   if(kmem.use_lock)
3180     release(&kmem.lock);
3181 }
```

# 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

```
1802 // This table defines the kernel's mappings, which are present in
1803 // every process's page table.
1804 static struct kmap {
1805   void *virt;
1806   uint phys_start;
1807   uint phys_end;
1808   int perm;
1809 } kmap[] = {
1810  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1811  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1812  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1813  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1814 };
1815
1816 // Set up kernel part of a page table.
1817 pde_t*
1818 setupkvm(void)
1819 {
1820   pde_t *pgdir;
1821   struct kmap *k;
1822
1823   if((pgdir = (pde_t*)kalloc()) == 0)
1824     return 0;
1825   memset(pgdir, 0, PGSIZE);
1826   if (P2V(PHYSTOP) > (void*)DEVSPACE)
1827     panic("PHYSTOP too high");
1828   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1829     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1830                 (uint)k->phys_start, k->perm) < 0) {
1831       freevm(pgdir);
1832       return 0;
1833     }
1834   return pgdir;
1835 }
```

# 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

```
1756 // Create PTEs for virtual addresses starting at va that refer to
1757 // physical addresses starting at pa. va and size might not
1758 // be page-aligned.
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
1765   a = (char*)PGROUNDDOWN((uint)va);
1766   last = (char*)PGROUNDDOWN(((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 }
```

# 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)

```
1731 // Return the address of the PTE in page table pgdir
1732 // that corresponds to virtual address va.  If alloc!=0,
1733 // create any required page table pages.
1734 static pte_t *
1735 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737   pde_t *pde;
1738   pte_t *pgtab;
1739
1740   pde = &pgdir[PDX(va)];
1741   if(*pde & PTE_P){
1742     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1743   } else {
1744     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1745       return 0;
1746     // Make sure all those PTE_P bits are zero.
1747     memset(pgtab, 0, PGSIZE);
1748     // The permissions here are overly generous, but they can
1749     // be further restricted by the permissions in the page table
1750     // entries, if necessary.
1751     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1752   }
1753   return &pgtab[PTX(va)];
1754 }
```

# 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

```
2579 int
2580 fork(void)
2581 {
2582   int i, pid;
2583   struct proc *np;
2584   struct proc *curproc = myproc();
2585
2586   // Allocate process.
2587   if((np = allocproc()) == 0){
2588     return -1;
2589   }
2590
2591   // Copy process state from proc.
2592   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
2593     kfree(np->kstack);
2594     np->kstack = 0;
2595     np->state = UNUSED;
2596     return -1;
2597   }
2598   np->sz = curproc->sz;
2599   np->parent = curproc;
2600   *np->tf = *curproc->tf;
2601
2602   // Clear %eax so that fork returns 0 in the child.
2603   np->tf->eax = 0;
2604
2605   for(i = 0; i < NOFILE; i++)
2606     if(curproc->ofile[i])
2607       np->ofile[i] = filedup(curproc->ofile[i]);
2608   np->cwd = idup(curproc->cwd);
2609
2610   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
2611
2612   pid = np->pid;
2613
2614   acquire(&ptable.lock);
2615
2616   np->state = RUNNABLE;
2617
2618   release(&ptable.lock);
2619
2620   return pid;
2621 }
```

# 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

```
2032 // Given a parent process's page table, create a copy
2033 // of it for a child.
2034 pde_t*
2035 copyuvm(pde_t *pgdir, uint sz)
2036 {
2037   pde_t *d;
2038   pte_t *pte;
2039   uint pa, i, flags;
2040   char *mem;
2041
2042   if((d = setupkvm()) == 0)
2043     return 0;
2044   for(i = 0; i < sz; i += PGSIZE){
2045     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2046       panic("copyuvm: pte should exist");
2047     if(!(*pte & PTE_P))
2048       panic("copyuvm: page not present");
2049     pa = PTE_ADDR(*pte);
2050     flags = PTE_FLAGS(*pte);
2051     if((mem = kalloc()) == 0)
2052       goto bad;
2053     memmove(mem, (char*)P2V(pa), PGSIZE);
2054     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
2055       kfree(mem);
2056       goto bad;
2057     }
2058   }
2059   return d;
2060
2061 bad:
2062   freevm(d);
2063   return 0;
2064 }
```

# 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)

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

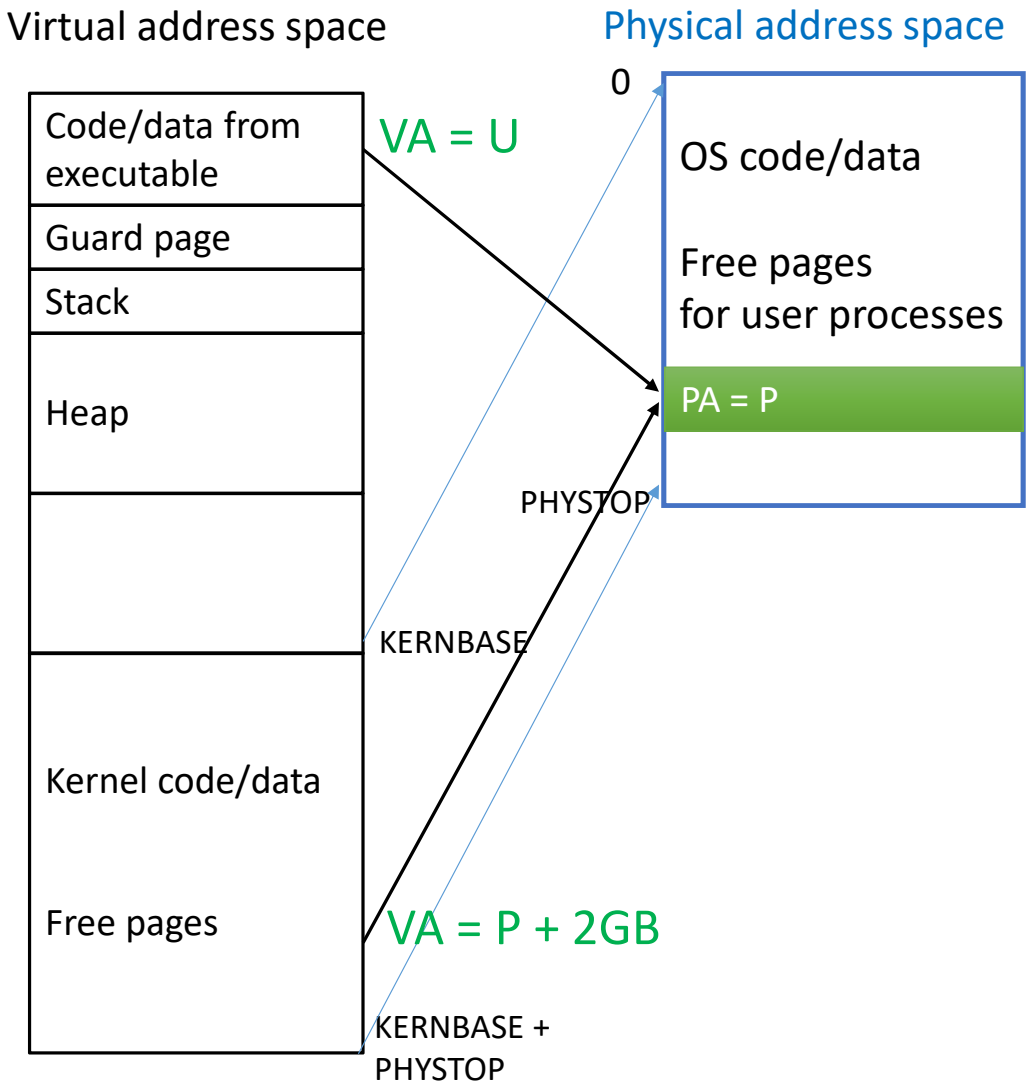
# allocuvm: grow address space

- 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

```
1926 int
1927 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1928 {
1929   char *mem;
1930   uint a;
1931
1932   if(newsz >= KERNBASE)
1933     return 0;
1934   if(newsz < oldsz)
1935     return oldsz;
1936
1937   a = PGROUNDUP(oldsz);
1938   for(; a < newsz; a += PGSIZE){
1939     mem = kalloc();
1940     if(mem == 0){
1941       cprintf("allocuvm out of memory\n");
1942       deallocuvm(pgdir, newsz, oldsz);
1943       return 0;
1944     }
1945     memset(mem, 0, PGSIZE);
1946     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1947       cprintf("allocuvm out of memory (2)\n");
1948       deallocuvm(pgdir, newsz, oldsz);
1949       kfree(mem);
1950       return 0;
1951     }
1952   }
1953   return newsz;
1954 }
```

# 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

```
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
6621   begin_op();
6622
6623   if((ip = namei(path)) == 0){
6624     end_op();
6625     cprintf("exec: fail\n");
6626     return -1;
6627   }
6628   ilock(ip);
6629   pgdir = 0;
6630
6631   // Check ELF header
6632   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6633     goto bad;
6634   if(elf.magic != ELF_MAGIC)
6635     goto bad;
6636
6637   if((pgdir = setupkvm()) == 0)
6638     goto bad;
```

```
6640   // Load program into memory.
6641   sz = 0;
6642   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6643     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6644       goto bad;
6645     if(ph.type != ELF_PROG_LOAD)
6646       continue;
6647     if(ph.memsz < ph.filesz)
6648       goto bad;
6649     if(ph.vaddr + ph.memsz < ph.vaddr)
6650       goto bad;
6651     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
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   iunlockput(ip);
6659   end_op();
6660   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

```
1900 // Load a program segment into pgdir.  addr must be page-aligned
1901 // and the pages from addr to addr+sz must already be mapped.
1902 int
1903 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1904 {
1905   uint i, pa, n;
1906   pte_t *pte;
1907
1908   if((uint) addr % PGSIZE != 0)
1909     panic("loaduvm: addr must be page aligned");
1910   for(i = 0; i < sz; i += PGSIZE){
1911     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1912       panic("loaduvm: address should exist");
1913     pa = PTE_ADDR(*pte);
1914     if(sz - i < PGSIZE)
1915       n = sz - i;
1916     else
1917       n = PGSIZE;
1918     if(readi(ip, P2V(pa), offset+i, n) != n)
1919       return -1;
1920   }
1921   return 0;
1922 }
```

# 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

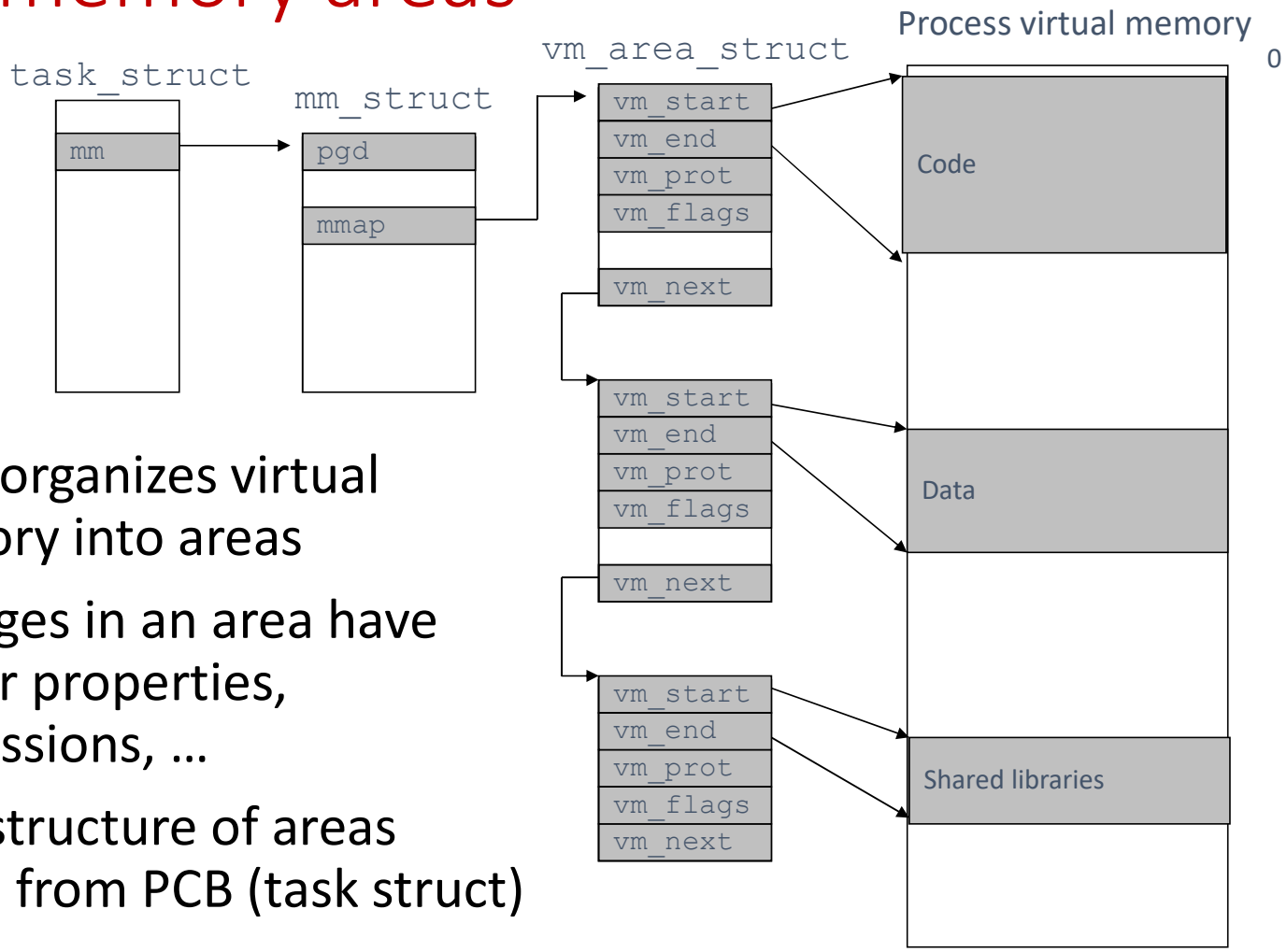
```
6662  // Allocate two pages at the next page boundary.
6663  // Make the first inaccessible.  Use the second as the user stack.
6664  sz = PGROUNDUP(sz);
6665  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6666    goto bad;
6667  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6668  sp = sz;
6669
6670  // Push argument strings, prepare rest of stack in ustack.
6671  for(argc = 0; argv[argc]; argc++) {
6672    if(argc >= MAXARG)
6673      goto bad;
6674    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6675    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6676      goto bad;
6677    ustack[3+argc] = sp;
6678  }
6679  ustack[3+argc] = 0;
6680
6681  ustack[0] = 0xffffffff;  // fake return PC
6682  ustack[1] = argc;
6683  ustack[2] = sp - (argc+1)*4;  // argv pointer
6684
6685  sp -= (3+argc+1) * 4;
6686  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6687    goto bad;
6688
```

# 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

```
6689   // Save program name for debugging.
6690   for(last=s=path; *s; s++)
6691     if(*s == '/')
6692       last = s+1;
6693   safestrcpy(curproc->name, last, sizeof(curproc->name));
6694
6695   // Commit to the user image.
6696   oldpgdir = curproc->pgdir;
6697   curproc->pgdir = pgdir;
6698   curproc->sz = sz;
6699   curproc->tf->eip = elf.entry;  // main
6700   curproc->tf->esp = sp;
6701   switchuvm(curproc);
6702   freevm(oldpgdir);
6703   return 0;
6704
6705 bad:
6706   if(pgdir)
6707     freevm(pgdir);
6708   if(ip){
6709     iunlockput(ip);
6710     end_op();
6711   }
6712   return -1;
6713 }
```

# 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