Operating System Design and Implementation

Lecture 8: System call

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm

Classroom: ED-302

Acknowledgements and Disclaimer

- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

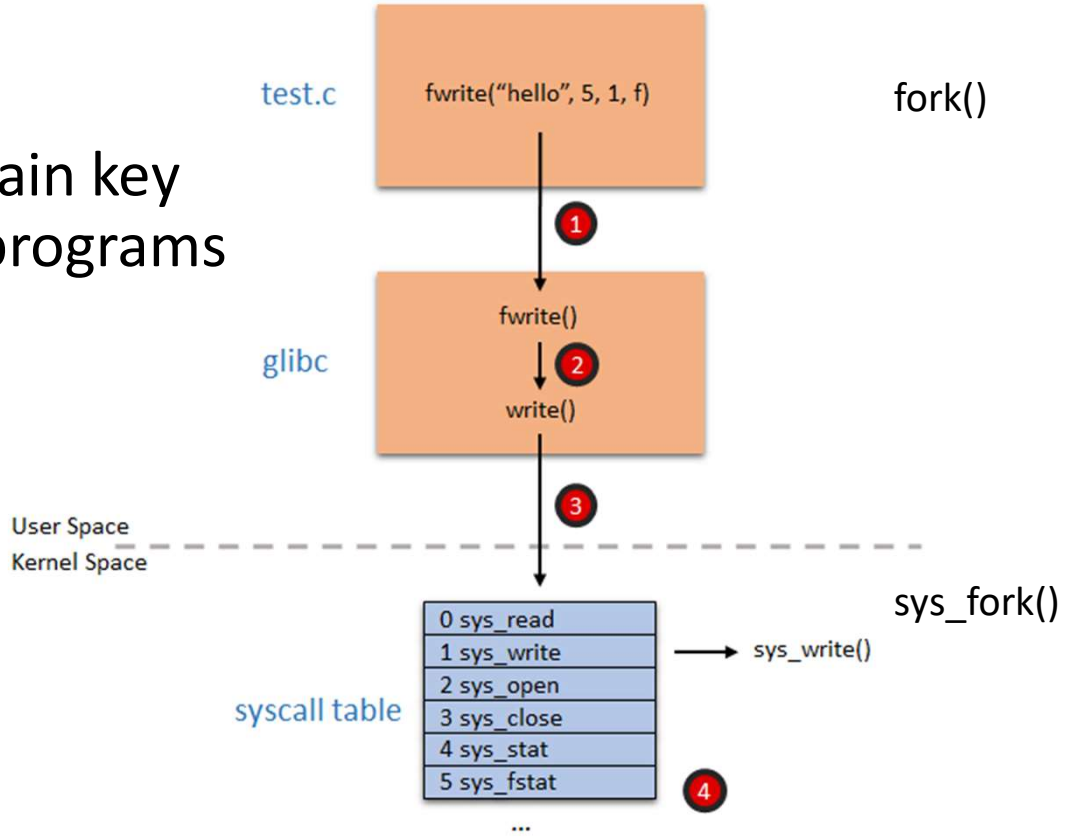
- System calls
  - User mode vs kernel mode
  - System call parameter passing
  - Trap instruction
  - System call example
  - Context switch

Execution mode, address space and context

- Mode is the level of privilege allowing some operations
  - **Kernel mode:**
    - CPU can perform any operations allowed by its architecture
  - **User mode:**
    - Instructions that could alter the global state of the machine are not permitted
    - Some memory areas cannot be access
  - Linux splits address in kernel space and user space
  - Context represents the current state of an execution flow
    - The process context can be seen as the content of the registers associated to this process: execution register, stack register …

System call

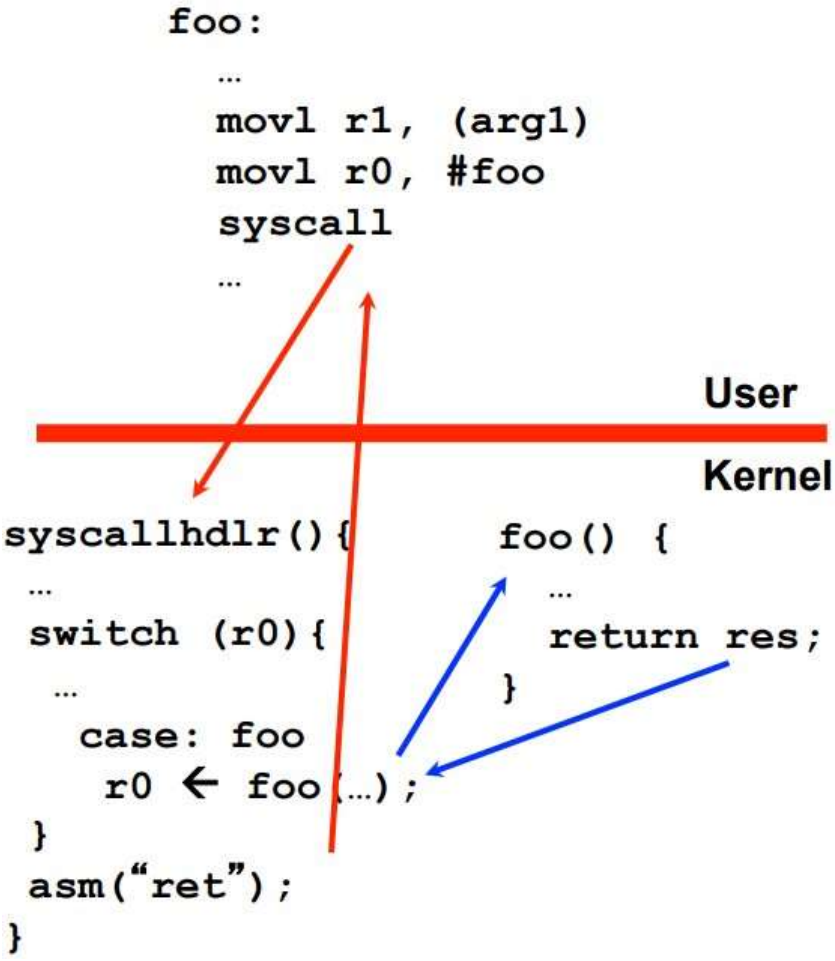
- **System call**
  - Allow the kernel to expose certain key pieces of functionality to user programs
    - Access the file system
    - Destroying processes
    - Communicating with other processes
    - Allocating more memory
    - To execute a system call, a program must execute a special **trap** instruction



Source: http://randibox.blogspot.tw/2016/02/the-fascinating-world-of-linux-system.html

Anatomy of a system call

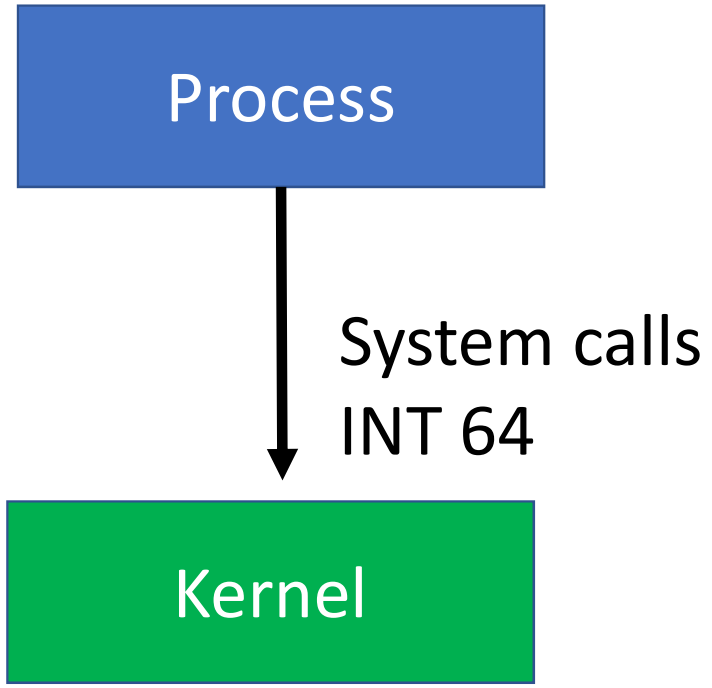
- **Anatomy of a system call**
  - Program puts syscall params in register
  - Program executes a **trap**
    - Processor state (PC, PSW) pushed on stack
    - CPU switches mode to KERNEL
    - CPU vectors to registered trap handler in the OS kernel
  - Trap handler uses param to jump to desired handler (e.g. fork, exec, open…)
  - When complete, reserve operation
    - Place return code in register
    - Return from exception



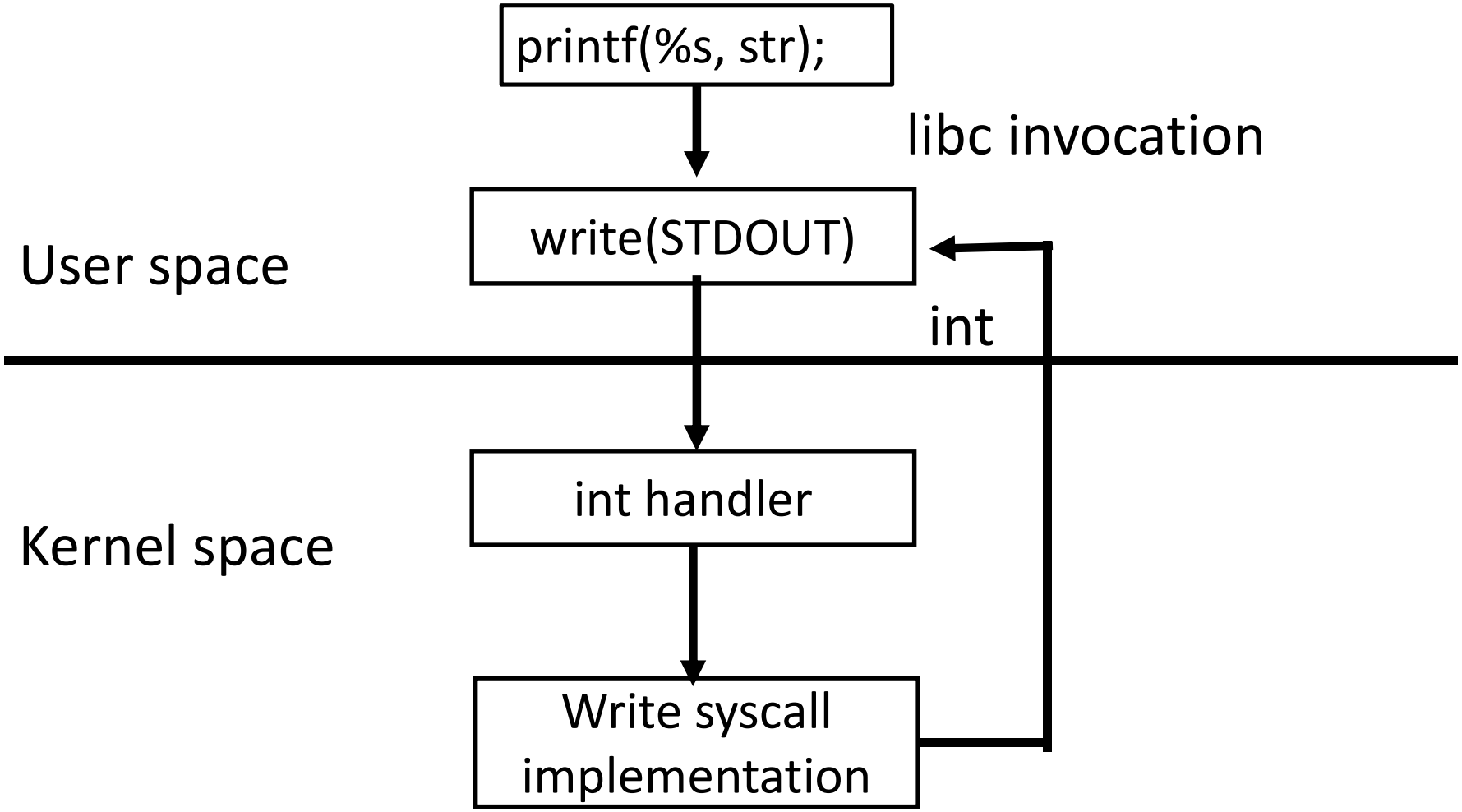
https://my.eng.utah.edu/~cs5460/slides/Lecture02.pdf

Software interrupt

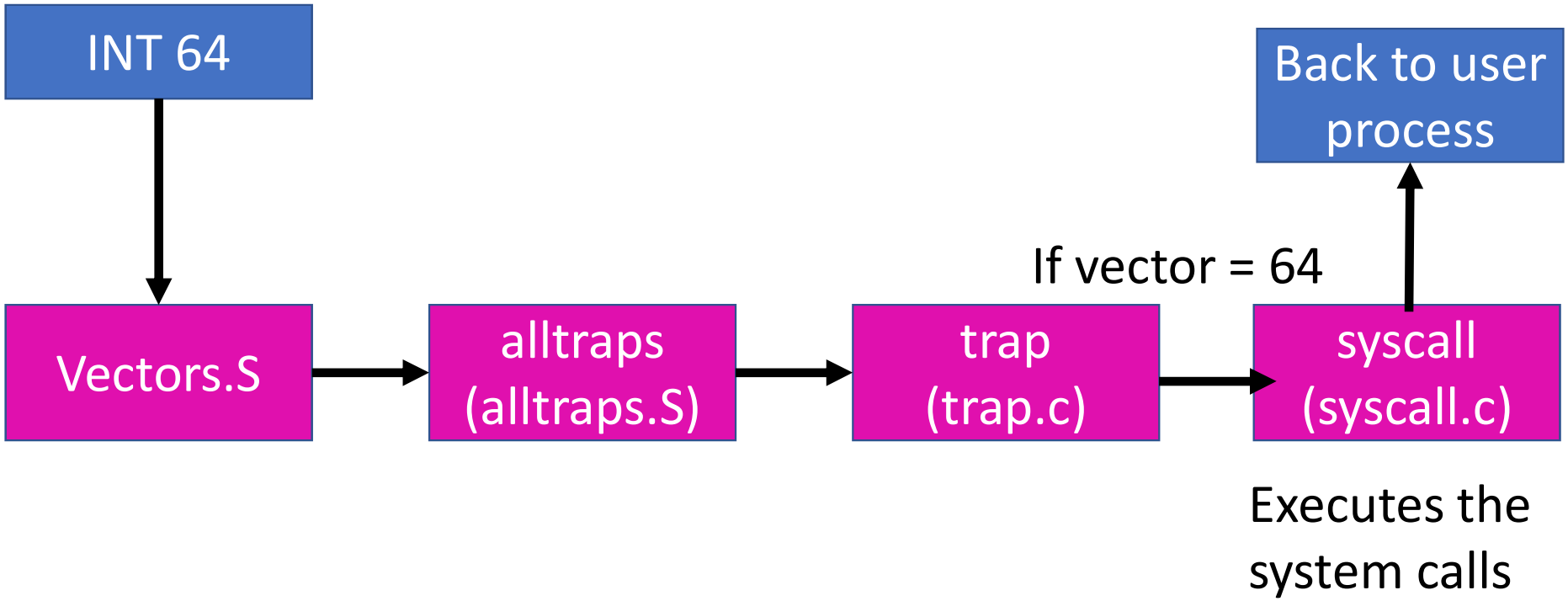
- Software interrupt used for implementing system calls
  - **INT** is an assembly language instruction for x86 processors that generates a software interrupt
  - In Linux INT 128 (0x80) (128 is interrupt number) used for system calls
  - In xv6, INT 64 is used for system calls



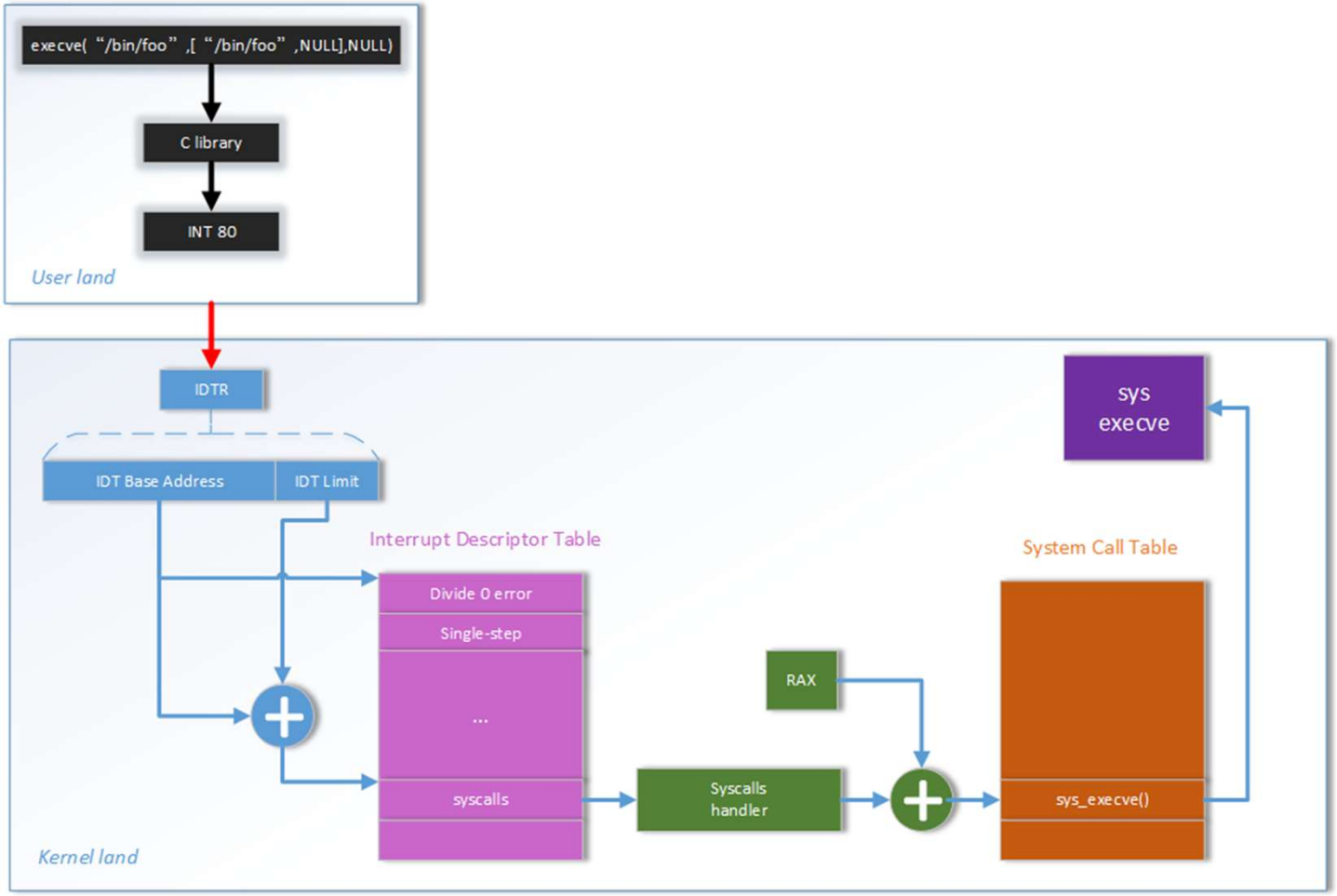
System call example



System call processing in kernel



Linux System Call Procedures



Source: http://monkee.esy.es/?p=1349

System call routines

System call	Description
fork()	Create a process
exit()	Terminate the current process
wait()	Wait for a child process to exit
open(filename, flag)	Open a file; the flags indicate read/write
read(fd, buf, n)	Read n bytes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
fstat(fd)	Return info about an open file
unlink(filename)	Remove a file

How does OS distinguish between the system calls ?

System call number

- **A system call number** is a unique integer
  - Assigned to each system call in a Unix-like operating system
  - The user code places the desired system call number **in a register** or **at a specified location on the stack**
  - The OS examines the system call number when handling the system call inside the trap handler

/usr/include/asm/unistd.h

```
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5
#define __NR_close         6
#define __NR_waitpid       7
#define __NR_creat         8
#define __NR_link          9
#define __NR_unlink        10
#define __NR_execve        11
#define __NR_chdir         12
#define __NR_time          13
#define __NR_mknod         14
#define __NR_chmod         15
#define __NR_lchown        16
```

System call number (cont.)

- System call number used to distinguish between system calls
  - Based on the system call number function syscall invokes the corresponding syscall handler

System call number

```
mov x, %eax
INT 64
```

System call numbers

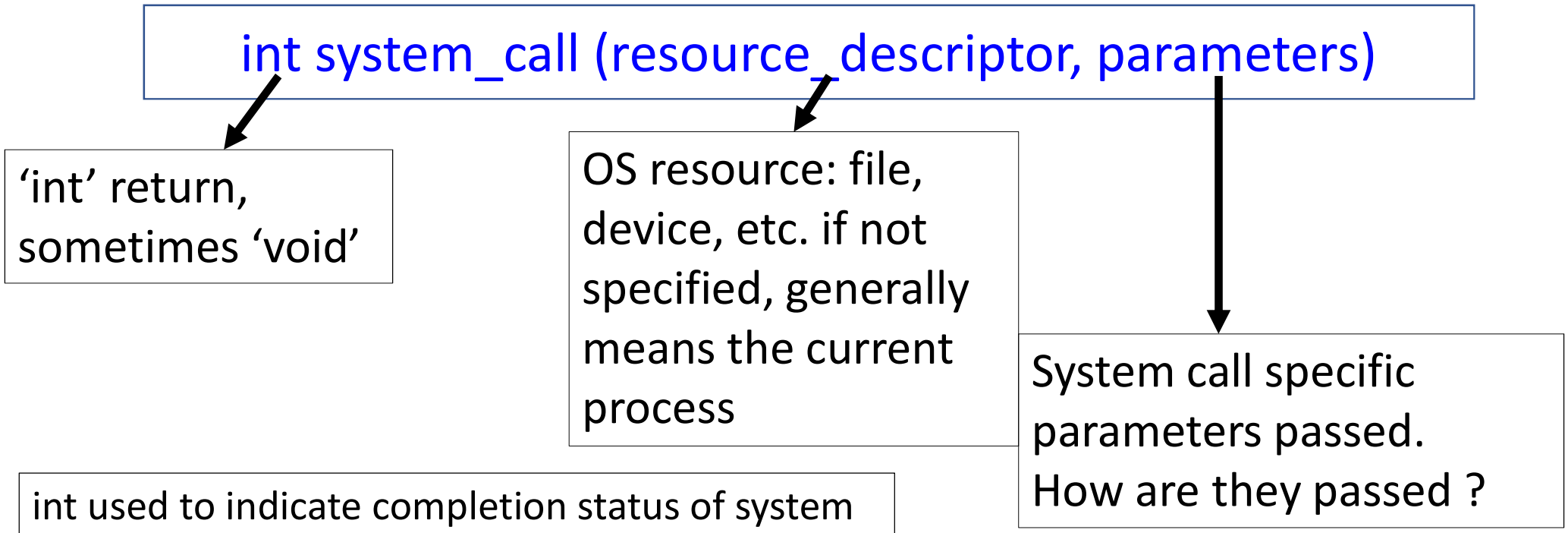
```
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

System call handlers

```
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
```

syscall.h, syscall() in syscall.c

Prototype of a typical system call



Passing parameters in system calls

- Passing parameters to system calls **not similar** to passing parameters in function calls
- Typical methods
  - Pass by **registers** (e.g. Linux)
  - Pass via **user mode stack** (e.g. xv6)
    - complex
  - Pass via a **designated memory region**
    - Address passed through registers

Pass by registers (Linux in x86)

- System calls with fewer than 6 parameters passed in registers
  - %eax (sys call number), %ebx, %ecx, %esi, %edi, %ebp
- If 6 or more arguments
  - Pass pointer to block structure containing argument list
  - Max size of argument is the register size (e.g. 32 bit)

System call example

Source

```
void foo (void) {
  write(1, "hello\n", 6);
}
```

Assembly code

```
<main>:
  pushq   %rax
  mov     $0x6,%edx
  mov     $0x694010,%esi
  mov     $0x1,%edi
  callq   libc_write
  xorl    %eax,%eax
  popq    %rdx
  ret
<libc_write>:
  mov     $0x1,%eax
  syscall
  cmp     $0xfffffffffffff001,%rax
  jae     <__syscall_error>
  retq
```

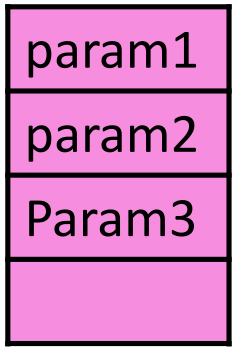
https://my.eng.utah.edu/~cs5460/slides/Lecture02.pdf

Pass via user mode stack (xv6)

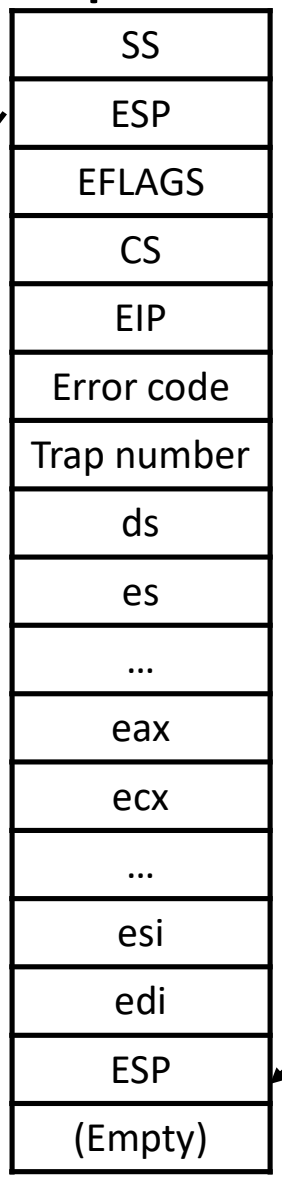
User process

```
push param1
push param2
push param3
mov sysnum, %eax
int 64
```

User stack



trapframe



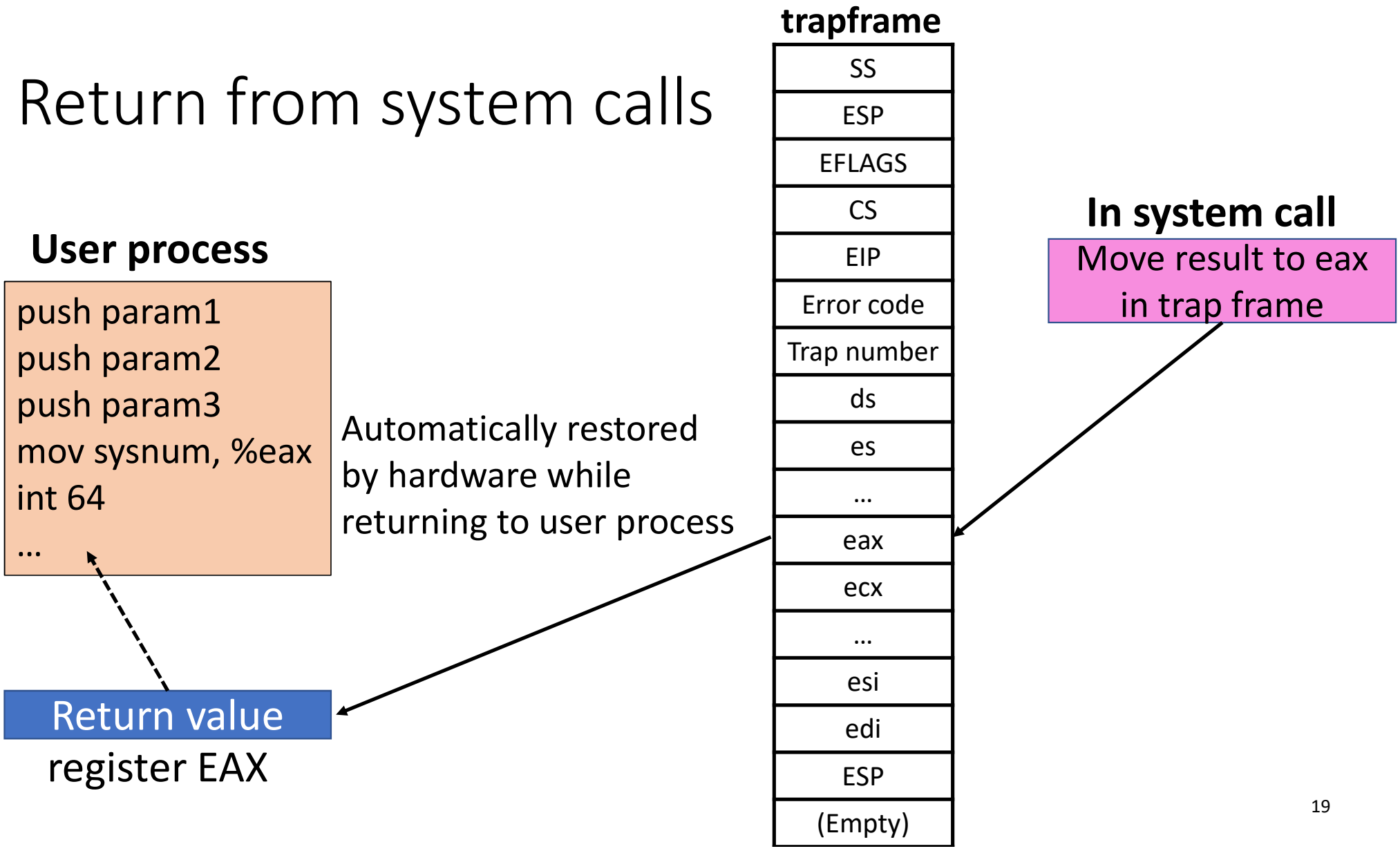
ESP pushed by hardware contains user mode stack pointer

Proc entry for process

Points to trapframe

sys_open(sysfile.c)

Return from system calls

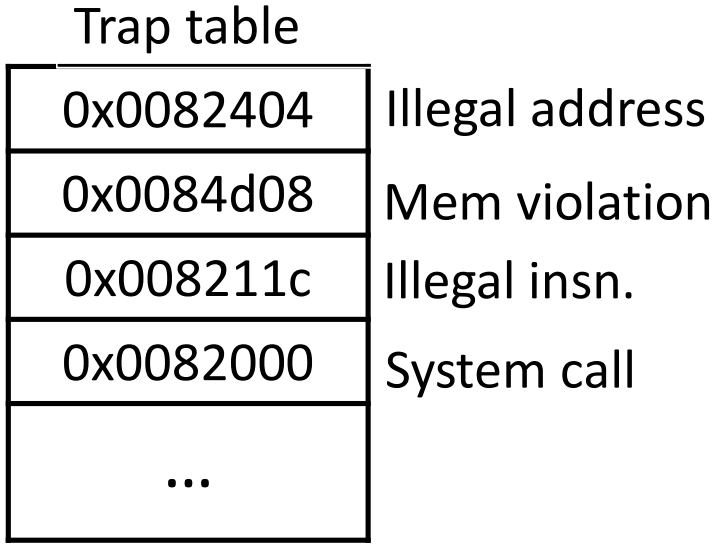


Trap process

- **The trap instruction**
  - The trap jumps into the kernel and raises the privilege level to kernel mode
  - When finished, the OS calls a special **return-from-trap** instruction
  - Return into the calling user program while simultaneously reducing the privilege back to user mode
- How does the trap know which code to run side the OS ?
  - Using the trap table
  - The kernel does so by setting up a **trap table** at boot time

Traps

- Architecture detects special events
  - Trap instruction, invalid memory access
  - Floating point exception, privileged instruction by user mode code
- When processor detects above conditions:
  - Save minimal CPU state (PC, sp, ..) - done by hardware
  - Switches to KERNEL mode
  - Transfers control to trap handler
    - Indexes trap table w/ trap number
    - Jumps to address in trap table
    - Handler saves more state and may disable interrupts
  - Return-from-trap (RTE) instruction reserves operation



0x82404 is address of handle_illegal addr()

Trap table at boot time

- When the machine boots up
  - First, the OS is to tell the hardware **what code to run when certain exceptional events occur**
    - What code should run when a hardware interrupt take place ?
    - When a program makes a system call ?
  - OS informs the hardware of the locations of these trap handlers through the special instruction
    - The location of these handles is remembered until the machine next reboots

OS @ boot (kernel mode)	**Hardware**
Initialize trap table	Remember address of syscall handler

Limited direction execution protocol

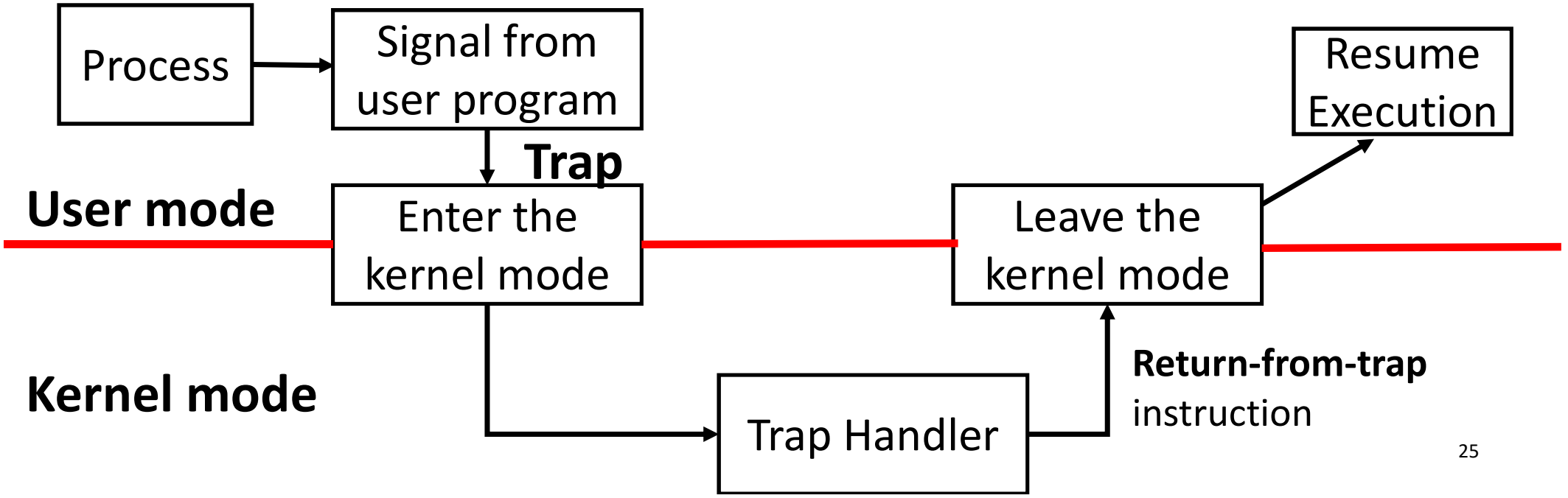
OS @ boot (kernel mode)	Hardware	Program (user mode)
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Setup user stack with argv 5. Fill kernel stack with reg/PC 6. Return-from-trap		
	1. Restore regs (from kernel stack) 2. Move to user mode 3. Jump to main	
		1. Run main() … 2. Call system call trap into OS

Limited direction execution protocol (cont.)

OS @ boot (kernel mode)	Hardware	Program (user mode)
	1. Save regs to kernel stack 2. Move to kernel mode 3. Jump to trap handler	
1. Handle trap 2. Do work of syscall **3. Return-from-trap**		
	1. Restore regs (from kernel stack) 2. Move to user mode 3. Jump to PC after trap	
		1. Return from main **2. Trap** (via exit())

Trap process

- **Return-from-trap instruction**
  - Switches the CPU to user mode and begins running the process
- When the process wishes to issue a system call, it traps back into the OS



The fork() system call

- The **fork()** system call is used to create a new process
  - The newly-created process called **child**
  - The child process doesn't start running at main()
  - The child isn't an exact copy of the parent process
  - The child receives a return code of zero

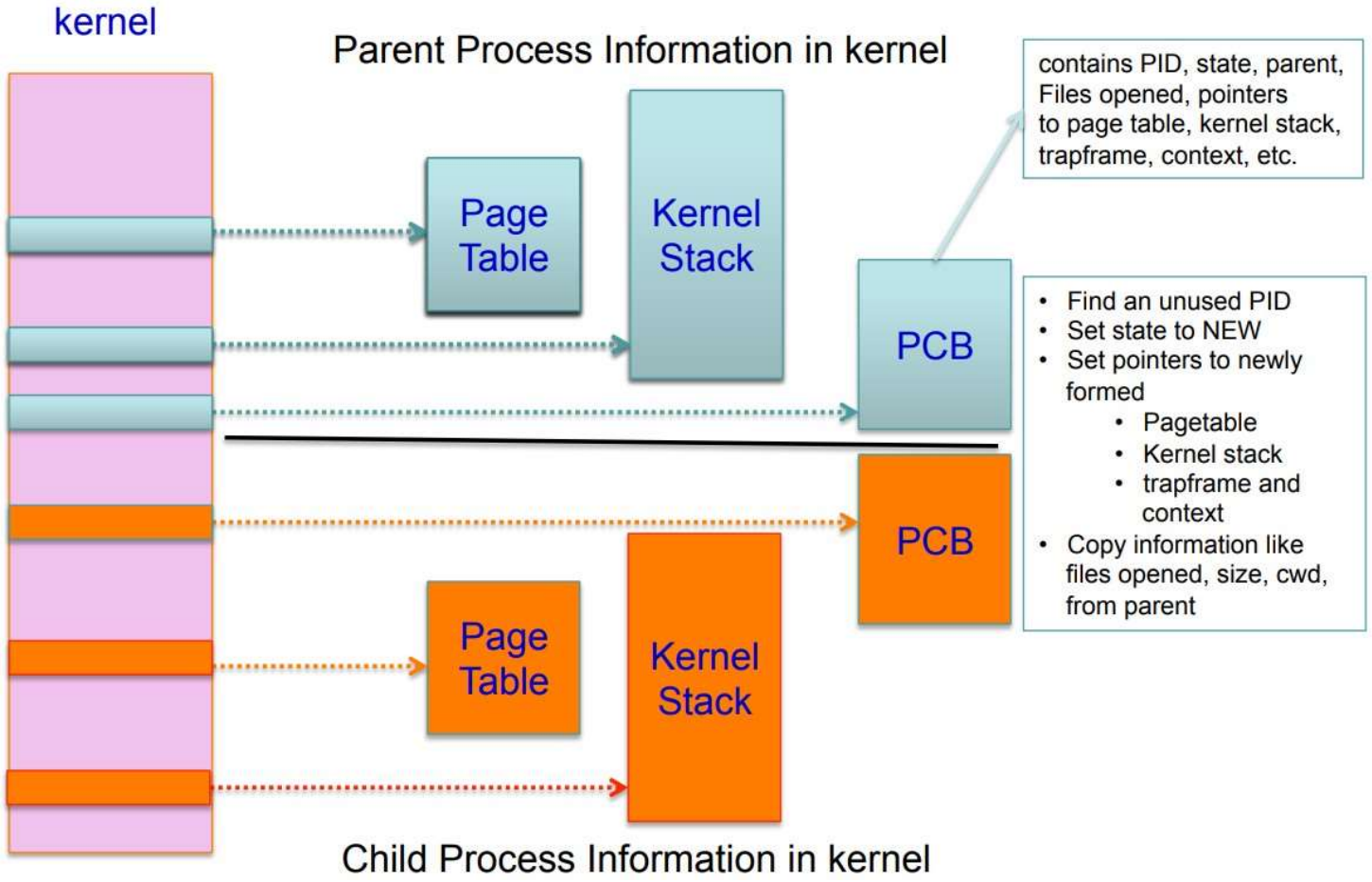
```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6    printf("hello world (pid:%d)\n", (int) getpid());
7    int rc = fork();
8    if (rc < 0) {
9      // fork failed
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) {
13     // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15   } else {
16     // parent goes down this path (main)
17     printf("hello, I am parent of %d (pid:%d)\n",
18            rc, (int) getpid());
19   }
20   return 0;
21 }
22
```

https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

fork: form an OS perspective

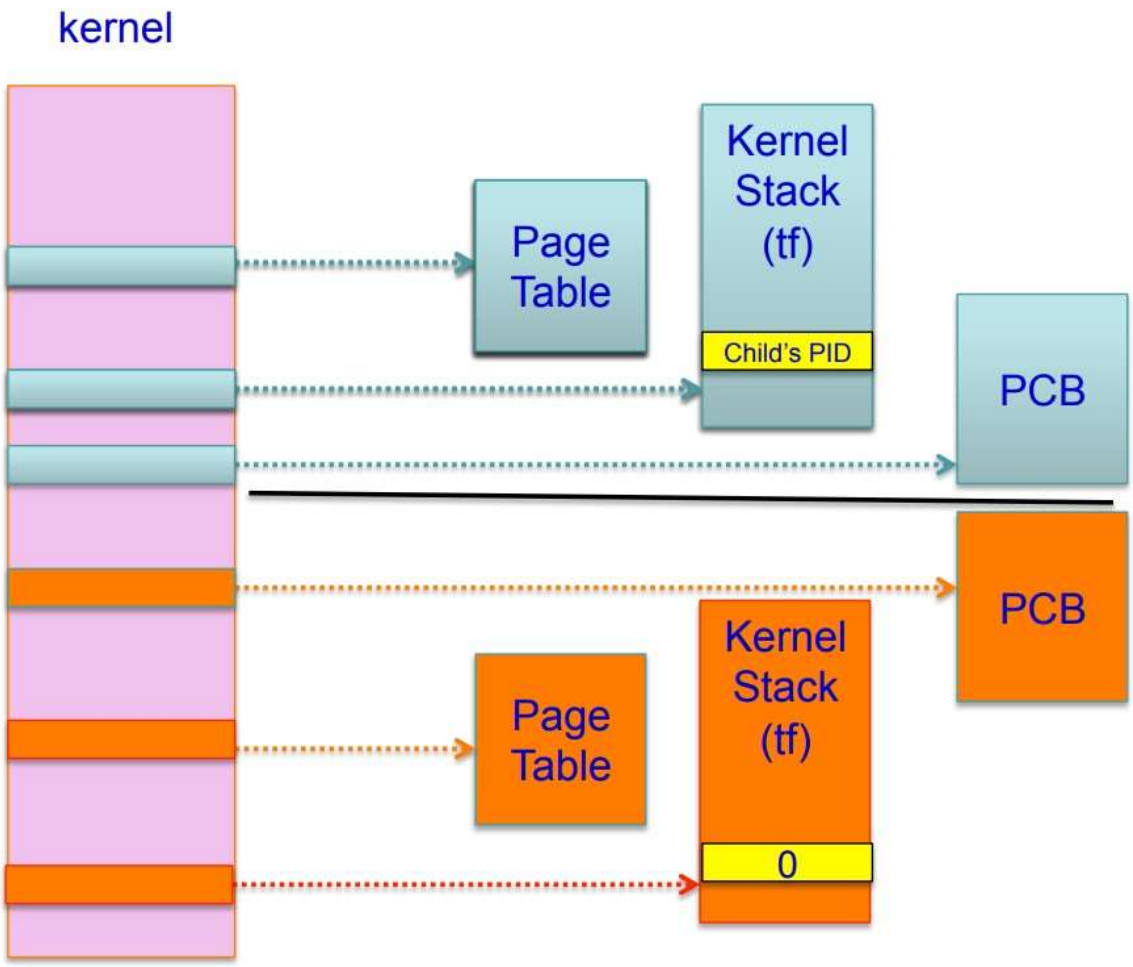
- **Set State to NEW** means the pid has been taken, the process is being created, but not ready to run



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Return from fork

- Return from fork is placed in the kernel stack
- Return value in parent has new child's PID
- Return value in child has 0
  - Registers modified in child process
  - %eax = 0 so that pid = 0 in child
- The eax entry in the trapframe has each process's return value



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

The xv6 fork

```
2553 int
2554 fork(void)
2555 {
2556   int i, pid;
2557   struct proc *np;
2558
2559   // Allocate process.
2560   if((np = allocproc()) == 0)
2561     return -1;
2562
2563   // Copy process state from p.
2564   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2565     kfree(np->kstack);
2566     np->kstack = 0;
2567     np->state = UNUSED;
2568     return -1;
2569   }
2570   np->sz = proc->sz;
2571   np->parent = proc;
2572   *np->tf = *proc->tf;
2573
2574   // Clear %eax so that fork returns 0 in the child.
2575   np->tf->eax = 0;
2576
2577   for(i = 0; i < NOFILE; i++)
2578     if(proc->ofile[i])
2579       np->ofile[i] = filedup(proc->ofile[i]);
2580   np->cwd = idup(proc->cwd);
2581
2582   safestrcpy(np->name, proc->name, sizeof(proc->name));
2583
2584   pid = np->pid;
2585
2586   // lock to force the compiler to emit the np->state write last.
2587   acquire(&ptable.lock);
2588   np->state = RUNNABLE;
2589   release(&ptable.lock);
2590
2591   return pid;
2592 }
```

Pick an UNUSED proc. Set pid. Allocate kstack. fill kstack with (1) the trapframe pointer, (2) trapret and (3) context
np is the proc pointer for the new process

Copy page directory from the parent process (proc→pgdir) to the child process (np→pgdir)

Set size of np same as that of parent
Set parent of np
Copy trapframe from parent to child

In child process, set eax register in trapframe to 0. This is what fork returns in the child process

Other things… copy file pointer from parent, cwd, executable name

Child process is finally made runnable

Parent process returns the pid of the child

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

The fork() is not deterministic

- **The output of fork() is not deterministic**
  - When the child process is created, there are two active processes in the system
  - The parent did and thus printed out its message first
  - The opposite might happen
  - The CPU scheduler determines which process runs at a given moment in time

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

The wait() system call

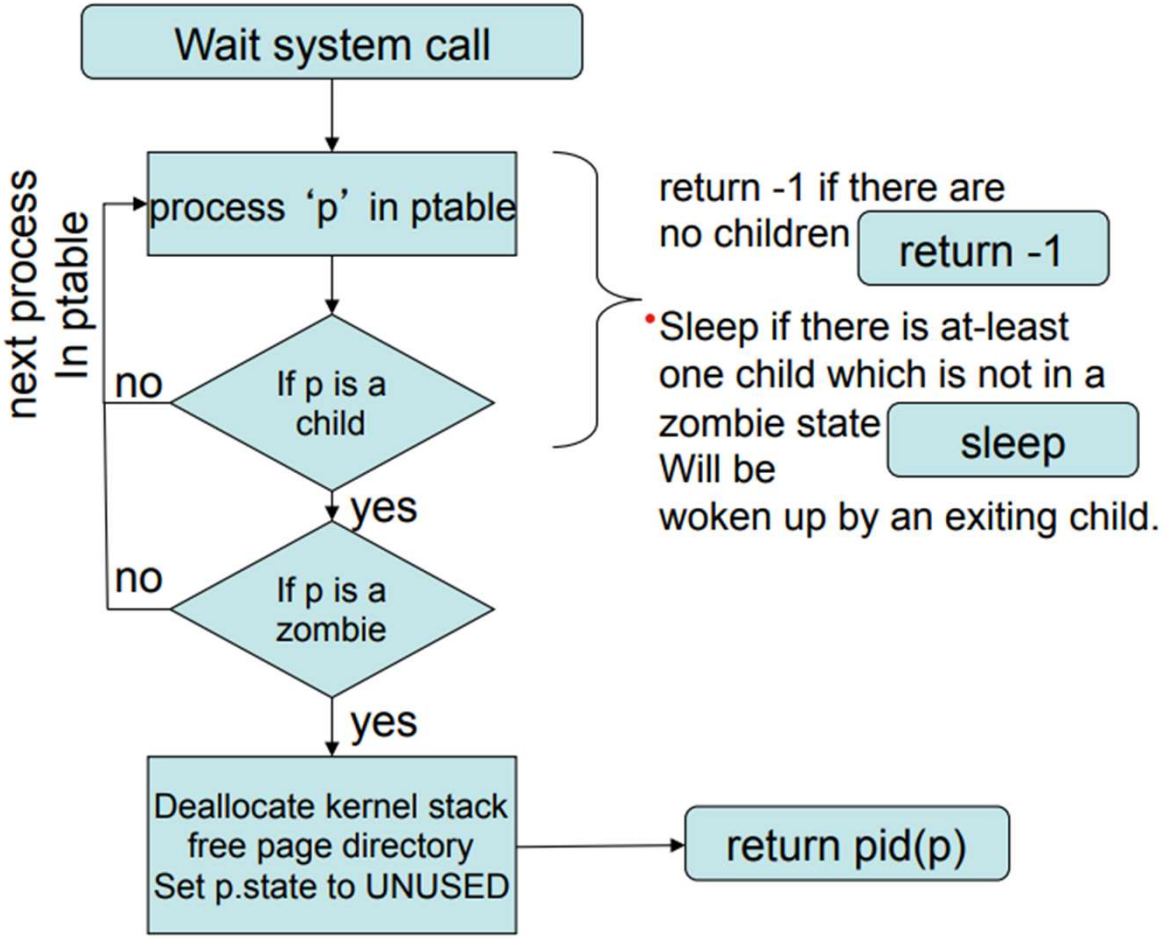
- **The wait() system call**
  - The parent process calls wait() to delay its execution until the child finishes executing
  - The wait() call makes the output deterministic
  - The child will always print first

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char *argv[]) {
7    printf("hello world (pid:%d)\n", (int) getpid());
8    int rc = fork();
9    if (rc < 0) {          // fork failed; exit
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) { // child (new process)
13     printf("hello, I am child (pid:%d)\n", (int) getpid());
14   } else {               // parent goes down this path (main)
15     int rc_wait = wait(NULL);
16     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
17            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
```

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

Wait() system call internal



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

wait

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

If 'p' is infact a child of proc and is in the ZOMBIE state then free remaining entries in p and return pid of p

note : page directory, kernel stack, deallocated here
… allows parent to peek into exited child's process

If 'p' is infact a child of proc and is not a ZOMBIE then block the current process

ref : proc.c

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Exit() system call

- **init**, the first process, never exit
- For all other processes on exit,

1. Decrement the usage count of all open files
   a. If usage count is 0, close file
2. Drop reference to in-memory inode
3. Wakeup parent
   a. If parent state is sleeping, make it runnable
   b. Needed, because parent may be sleeping due to a wait
4. Make init adopt child of exited process
5. Set process state to ZOMBIE
6. Force context switch to scheduler

exit

```
exit(void)
{
  struct proc *p;
  int fd;

  if(proc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(proc->cwd);
  end_op();
  proc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

initproc can never exit

Close all open files

Decrement in-memory inode usage

Wakeup parent of child

For every child of exiting process,
Set its parent to initproc

Set exiting process state to zombie
and invoke the scheduler, which performs
a context switch

ref : proc.c

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

The exec() system call

- The **exec()** system call
  - Load a program into memory and then execute it
  - Loads code and static data from the executable and **overwrites its current code segment**
  - The heap, stack and memory space of the program are re-initialized

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char *argv[]) {
8    printf("hello world (pid:%d)\n", (int) getpid());
9    int rc = fork();
10   if (rc < 0) {          // fork failed; exit
11     fprintf(stderr, "fork failed\n");
12     exit(1);
13   } else if (rc == 0) { // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15     char *myargs[3];
16     myargs[0] = strdup("wc");   // program: "wc" (word count)
17     myargs[1] = strdup("p3.c"); // argument: file to count
18     myargs[2] = NULL;           // marks end of array
19     execvp(myargs[0], myargs);  // runs word count
20     printf("this shouldn't print out");
21   } else {               // parent goes down this path (main)
22     int rc_wait = wait(NULL);
23     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
24            rc, rc_wait, (int) getpid());
25   }
26   return 0;
27 }
```

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

exec()

```
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;

  begin_op();
  if((ip = namei(path)) == 0){
    end_op();
    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;
    •
    •
    •
    •
```

Parameters are the path of executable and command line arguments

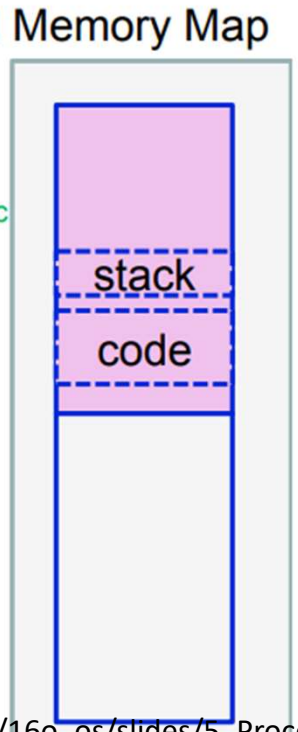
Get pointer to the inode for the executable

Executable files begin with a signature.

Sanity check for magic number. All executables begin with a ELF Magic number string : “\x7fELF”

Set up kernel side of the page tables again!!!

Do we really need to do this?



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

exec() : Load segments into memory

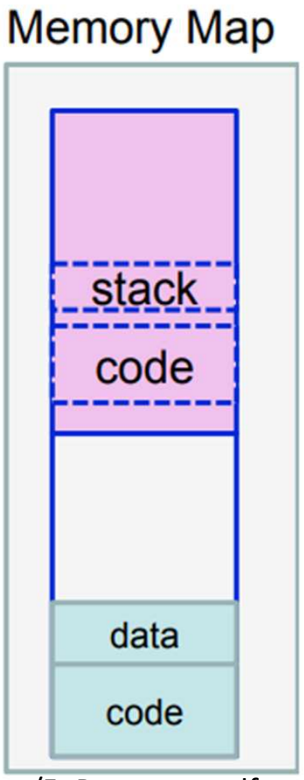
```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr +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

Parse through all the elf program headers.

Only load into memory segments of type LOAD

Add more page table entries to grow page tables from old size to new size (ph.vaddr + ph.memsz)

Copy program segment from disk to memory at location ph.vaddr. (3rd param is inode pointer, 4th param is offset of segment in file, 5th param is the segment size in file)

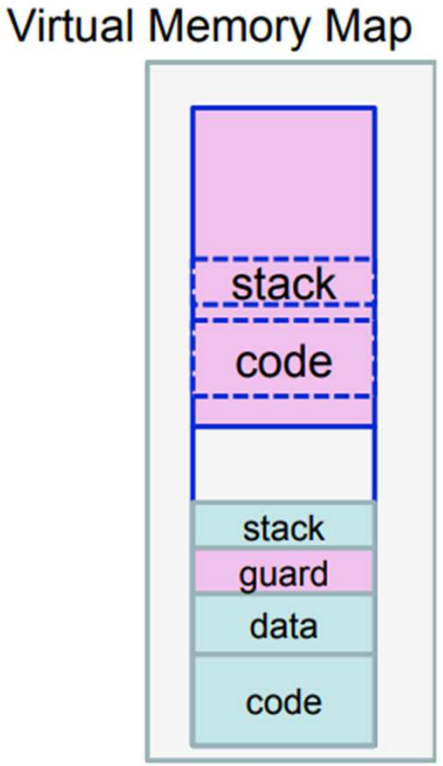


http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

exec(): user stacks

```
// 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;
```

The first acts as a guard page protecting stack overflows



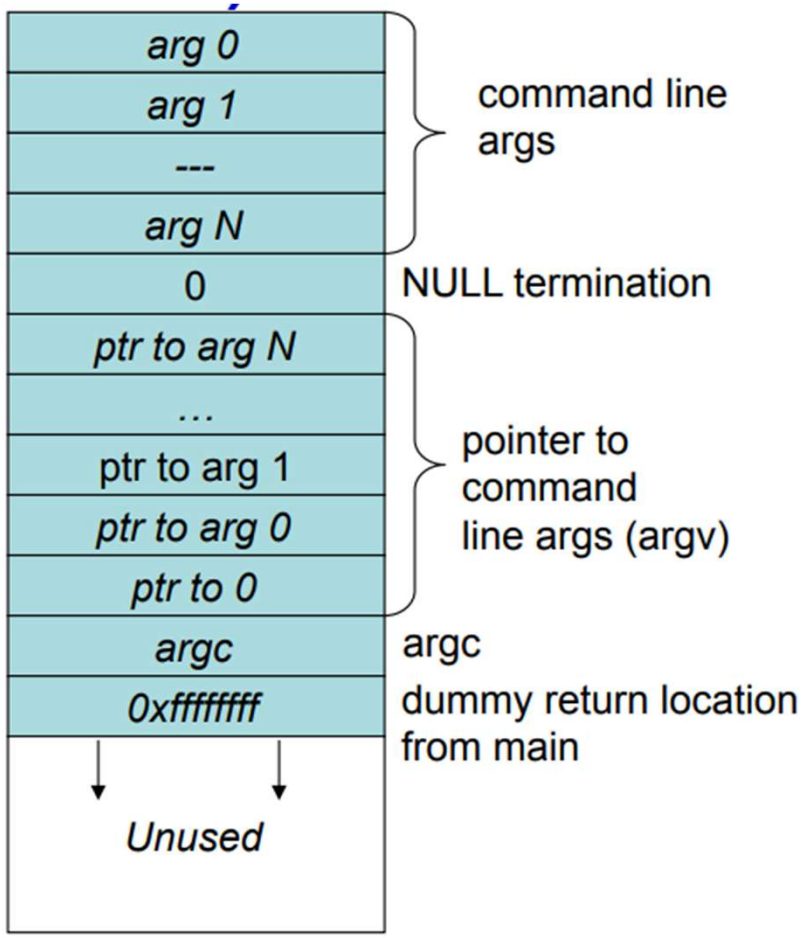
http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

exec(): fill user stack

```
// 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;
```



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

exec(): proc, trapframe

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

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

Set the executable file name in proc

these specify where execution should start for the new program.
Also specifies the stack pointer

Alter TSS segment's sp and esp.
Switch cr3 to the new page tables.

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Summary

- **System calls**
  - Arguments places in well-known registers
  - Perform trap instruction -> vector to system call handler
    - Low level code carefully saves CPU state
    - Processor switches to protected/kernel mode
    - Syscall handler checks param and jumps to desired handler
  - Return from system call
    - Perform RTE instruction: switches to user mode and returns to location where trap was called
- **OS manages trap/interrupt tables**
  - Traps are synchronous; interrupts are asynchronous