

# Discussion Section

January 29, 2025

# Processes

Instance of a program being executed

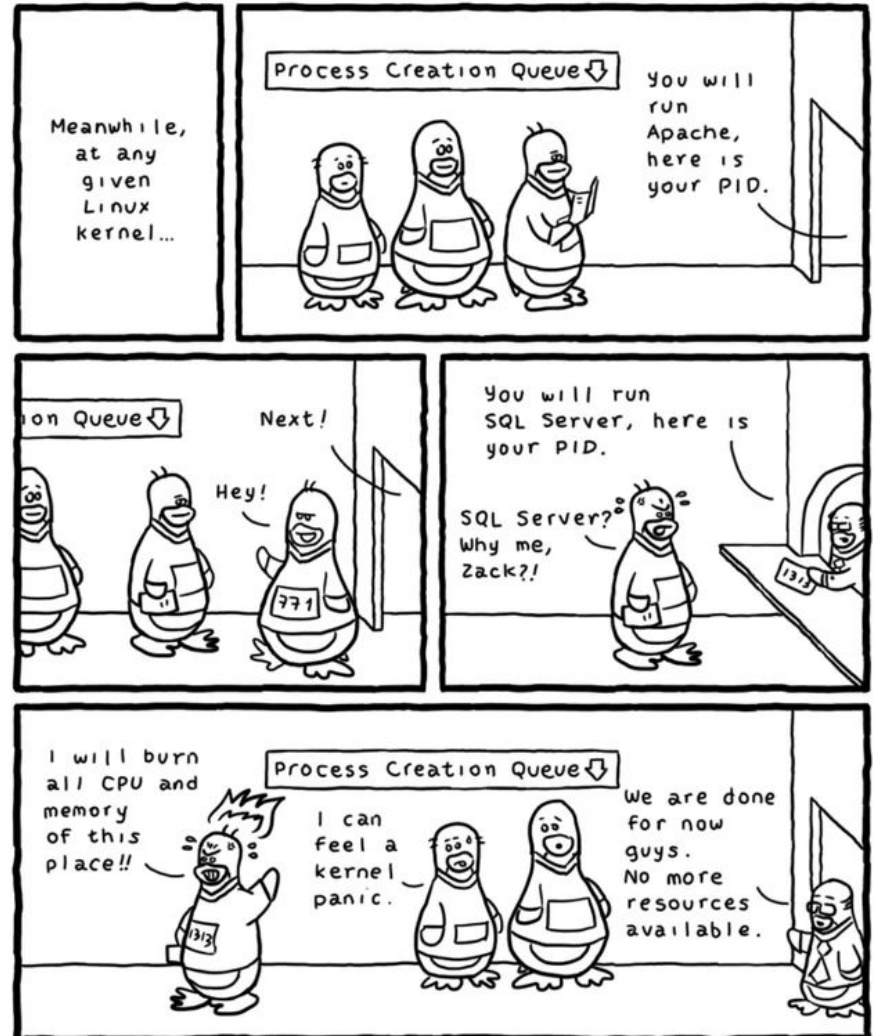
Each process presented with illusion of having resources entirely to itself -  
**virtualization**

How are resources virtualized?

Time and space sharing

Image credits:

<https://turnoff.us/geek/sql-server-on-linux/>



# Creating processes

`fork()`

`exec()`

# Creating processes

**fork()**

```
// Creating first child
int n1 = fork();
```

**exec()**

```
// Creating second child. First child
// also executes this line and creates
// grandchild.
int n2 = fork();
```

```
if (n1 > 0 && n2 > 0) {
    printf("parent\n");
    printf("%d %d \n", n1, n2);
    printf(" my id is %d \n", getpid());
}
else if (n1 == 0 && n2 > 0)
{
    printf("First child\n");
    printf("%d %d \n", n1, n2);
    printf("my id is %d \n", getpid());
}
```

```
else if (n1 > 0 && n2 == 0)
{
    printf("Second child\n");
    printf("%d %d \n", n1, n2);
    printf("my id is %d \n", getpid());
}
else {
    printf("third child\n");
    printf("%d %d \n", n1, n2);
    printf(" my id is %d \n", getpid());
}
return 0;
```

Multiple forks

Example from <https://www.geeksforgeeks.org/creating-multiple-process-using-fork/>

# Program output

```
$ gcc fork.c -o fork_ex
$ ./fork_ex
parent
28808 28809
  my id is 28807
First child
0 28810
my id is 28808
Second child
28808 0
my id is 28809
third child
0 0
my id is 28810
```

# Creating processes

**fork()**

```
int main( void ) {  
    char *argv[3] = {"Command-line", ".", NULL};
```

**exec()**

```
    int pid = fork();  
  
    if ( pid == 0 ) {  
        execvp( "find", argv );  
    }  
  
    /* Put the parent to sleep -- let the child finished executing */  
    wait( NULL );  
  
    printf( "Finished executing the parent process\n"  
           " - the child won't get here--you will only see this once\n" );  
  
    return 0;  
}
```

Combining fork and exec

Example from [https://ece.uwaterloo.ca/~dwharder/icsrts/Tutorials/fork\\_exec/](https://ece.uwaterloo.ca/~dwharder/icsrts/Tutorials/fork_exec/)

# Program output

```
$ gcc exec.c -o exec_bin
```

```
$ ./exec_bin
```

```
.
```

```
./exec_bin
```

```
./exec.c
```

```
./fork.c
```

```
./fork
```

```
Finished executing the parent process
```

```
- the child won't get here--you will only see this once
```

To be PID 1 on Linux is like:

Let the party begin.



fork fork fork fork fork fork  
fork fork fork fork fork fork  
fork fork fork fork fork fork  
fork fork fork fork fork fork  
fork fork fork fork fork fork  
fork fork fork fork fork fork  
fork fork fork fork fork fork

Image credits:  
<https://turnoff.us/geek/pid1/>

Daniel Stori; {turnoff.us}



# Process state

Memory - address space

Registers

I/O information

# Process API

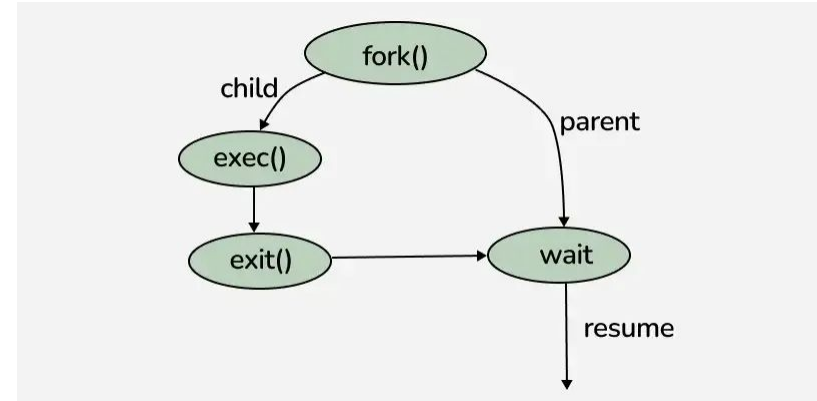
Creation

Destruction

Wait

Stop and Resume

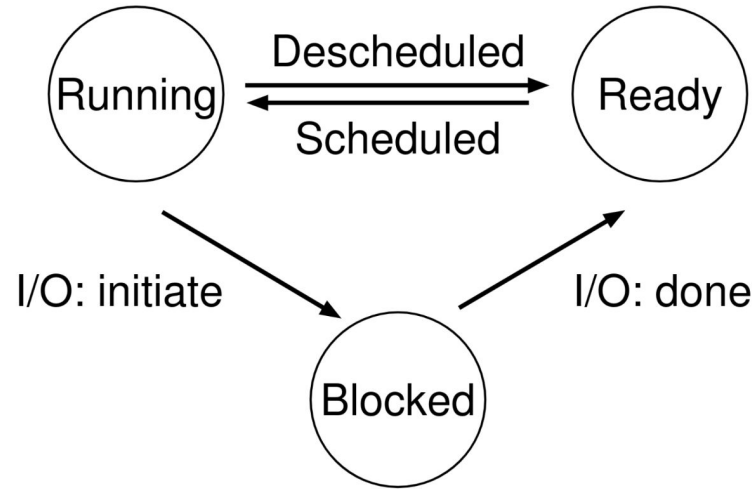
Status



Process creation example - how does the API factor in?

Image credits: <https://www.geeksforgeeks.org/process-creation-and-deletions-in-operating-systems/>

# Process state



**Figure 4.2: Process: State Transitions**

# Process state

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

Process states in xv6

# Limited Direct Execution

OS may need to take back control of CPU from process

- Interrupt issued to return control to OS

- Resources may be allocated to another process via **context switch**

Prevent malicious processes from accessing resources they aren't allowed to

- Manage access to resources using **system calls**

# Context switches

OS periodically interrupts processes to make decisions about future resource allocation

Decisions made by **scheduler**

Another process might be prioritized, current process stopped

Register state saved to stack

Register state for process chosen to be executed restored

Context switch can also occur when processes voluntarily wait for interrupts

# System calls

Processes may need to access resources like disk or memory

How do we prevent processes from accessing anything they aren't supposed to?

Solution: user and kernel mode

Processes typically execute in user mode

Trap into kernel and execute restricted operations in kernel mode via system calls

Jump to kernel using kernel defined trap table

Putting it all together with real examples from xv6



# What is xv6?

Teaching Operating System designed at MIT for undergraduate OS course

Source available at <https://github.com/mit-pdos/xv6-public>

Book: <https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf>

Chapters 0, 1, 3 and 5 relevant in context of this discussion and lectures on processes

# Process structures

```
// Per-process state
struct proc {
    uint sz;                // Size of process memory (bytes)
    pde_t* pgdir;           // Page table
    char *kstack;           // Bottom of kernel stack for this process
    enum procstate state;   // Process state
    int pid;                // Process ID
    struct proc *parent;    // Parent process
    struct trapframe *tf;   // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan;             // If non-zero, sleeping on chan
    int killed;             // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;      // Current directory
    char name[16];         // Process name (debugging)
};
```

# Context switch registers

```
// Saved registers for kernel context switches.  
// Contexts are stored at the bottom of the stack they  
// describe; the stack pointer is the address of the context.  
// The layout of the context matches the layout of the stack in switch.S  
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,  
// but it is on the stack and allocproc() manipulates it.  
struct context {  
    uint edi;  
    uint esi;  
    uint ebx;  
    uint ebp;  
    uint eip;  
};
```

<https://github.com/mit-pdos/xv6-public/blob/master/proc.h>

# Context switch code

```
# Context switch
# void swtch(struct context **old, struct context *new);
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.
```

```
.globl swtch
```

```
swtch:
```

```
    movl 4(%esp), %eax
```

```
    movl 8(%esp), %edx
```

```
    # Save old callee-saved registers
```

```
    pushl %ebp
```

```
    pushl %ebx
```

```
    pushl %esi
```

```
    pushl %edi
```

<https://github.com/mit-pdos/xv6-public/blob/master/swtch.S>

# Context switch code

```
# Switch stacks  
movl %esp, (%eax)  
movl %edx, %esp
```

```
# Load new callee-saved registers  
popl %edi  
popl %esi  
popl %ebx  
popl %ebp  
ret
```

<https://github.com/mit-pdos/xv6-public/blob/master/swtch.S>

# xv6 trap vector init

```
extern uint vectors[]; // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;

void
tvinit(void)
{
    int i;

    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

    initlock(&tickslock, "time");
}
...
```

<https://github.com/mit-pdos/xv6-public/blob/master/trap.c>

# xv6 user mode system call definition

```
#include "syscall.h"
```

```
#include "traps.h"
```

```
#define SYSCALL(name) \  
    .globl name; \  
    name: \  
        movl $SYS_ ## name, %eax; \  
        int $T_SYSCALL; \  
        ret
```

```
SYSCALL(fork)
```

```
SYSCALL(exit)
```

```
SYSCALL(wait)
```

```
SYSCALL(pipe)
```

```
...
```

<https://github.com/mit-pdos/xv6-public/blob/master/usys.S>

## xv6 trapframe (x86)

```
// Layout of the trap frame built on the stack by the  
// hardware and by trapasm.S, and passed to trap().
```

```
struct trapframe {  
    // registers as pushed by pusha  
    uint edi;  
    uint esi;  
    uint ebp;  
    uint oesp;    // useless & ignored  
    uint ebx;  
    uint edx;  
    uint ecx;  
    uint eax;
```

```
    // rest of trap frame  
    ushort gs;  
    ...
```

<https://github.com/mit-pdos/xv6-public/blob/master/x86.h>



# xv6 trap vector to trap

```
.globl alltraps
```

```
alltraps:
```

```
    # Build trap frame.
```

```
    pushl %ds
```

```
    pushl %es
```

```
    pushl %fs
```

```
    pushl %gs
```

```
    pushal
```

```
... (Segment stuff)
```

```
    # Call trap(tf), where tf=%esp
```

```
    pushl %esp
```

```
    call trap
```

```
...
```

<https://github.com/mit-pdos/xv6-public/blob/master/trapasm.S>

# xv6 trap to system call

```
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }

    switch(tf->trapno){
    case T_IRQ0 + IRQ_TIMER:
    ...
```

<https://github.com/mit-pdos/xv6-public/blob/master/trap.c>

# xv6 system call table

syscalls is an array of function pointers

sys\_fork, sys\_exec etc. are function pointers

<https://github.com/mit-pdos/xv6-public/blob/master/syscall.c>

```
static int (*syscalls[])(void) =
{
  [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_unlink]    sys_unlink,
  [SYS_link]      sys_link,
  [SYS_mkdir]     sys_mkdir,
  [SYS_close]     sys_close,
};
```

# xv6 system call table

SYS\_fork, SYS\_exit etc.

are syscall numbers, and

are used to index into the

array

<https://github.com/mit-pdos/xv6-public/blob/master/syscall.h>

```
// 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_unlink    18
#define SYS_link      19
#define SYS_mkdir     20
#define SYS_close     21
```

# xv6 generic system call handler

```
void
syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        curproc->tf->eax = syscalls[num]();
    } else {
        cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}
```

NELEM(x) is (sizeof(x)/sizeof((x)[0]))  
(defined in defs.h)

<https://github.com/mit-pdos/xv6-public/blob/master/syscall.c>

# Individual syscall function handlers

```
int
sys_fork(void)
{
    return fork();
}
```

<https://github.com/mit-pdos/xv6-public/blob/master/sysproc.c>

```
int
sys_exit(void)
{
    exit();
    return 0; // not reached
}
```

...

## Up next in class: OS Policy



Image credits:  
<https://turnoff.us/geek/highest-score/>