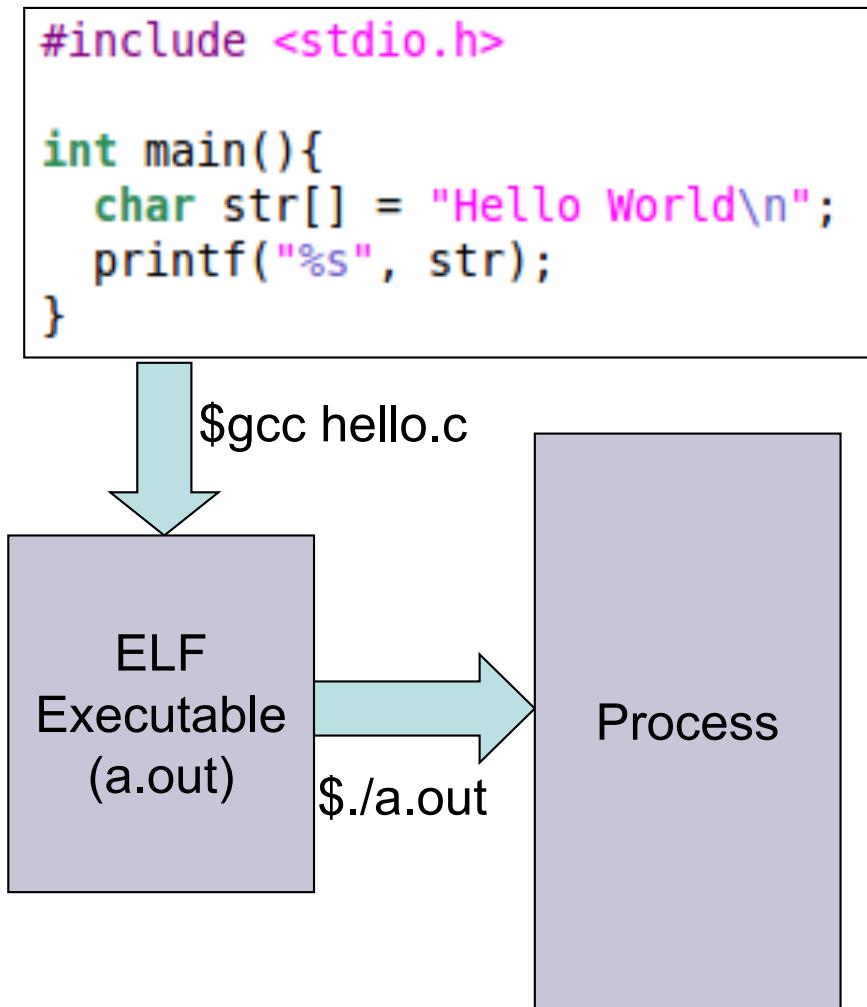


Processes

Chester Rebeiro

IIT Madras

Executing Apps (Process)



- **Process**
 - A program in execution
 - Most important abstraction in an OS
 - Comprises of
 - Code
 - Data
 - Stack
 - Heap
 - State in the OS
 - Kernel stack
 - State contains : registers, list of open files, related processes, etc.

In the user space of process

In the kernel space

note that each process has two stacks - the user stack to store variables and stuff a kernel stack to store local variables for syscall handlers.
the trapframe is used to store context during switches or system calls

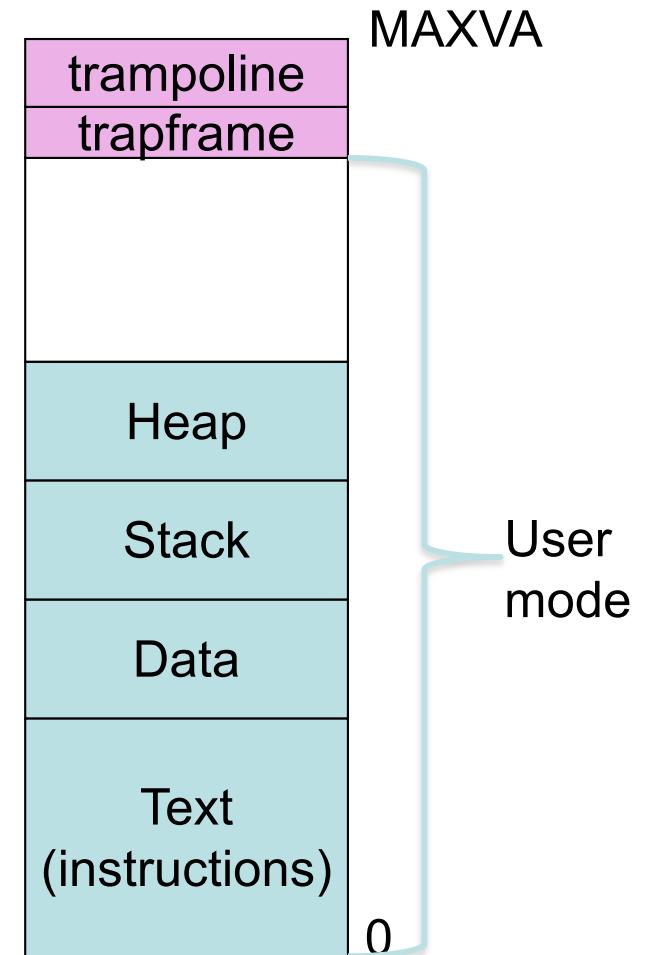
Program ≠ Process

Program	Process
code + static and global data	Dynamic instantiation of code + data + heap + stack + process state
One program can create several processes	A process is unique isolated entity

Process Address Space

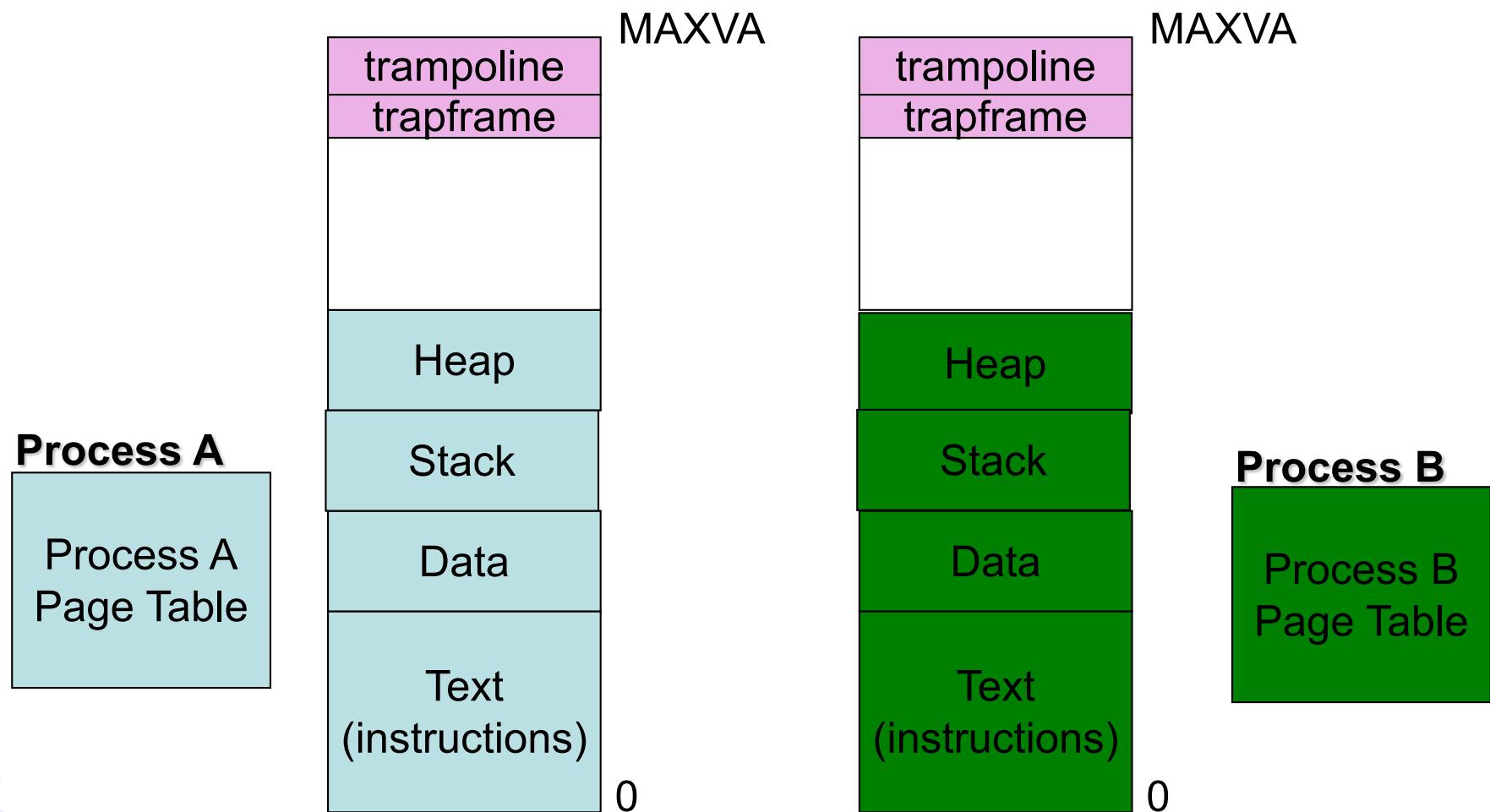
a user process can touch any of the regions except the pink(mapped to sv)

- **Virtual Address Map**
 - All memory a process can address
 - Large contiguous array of addresses from 0 to MAXVA

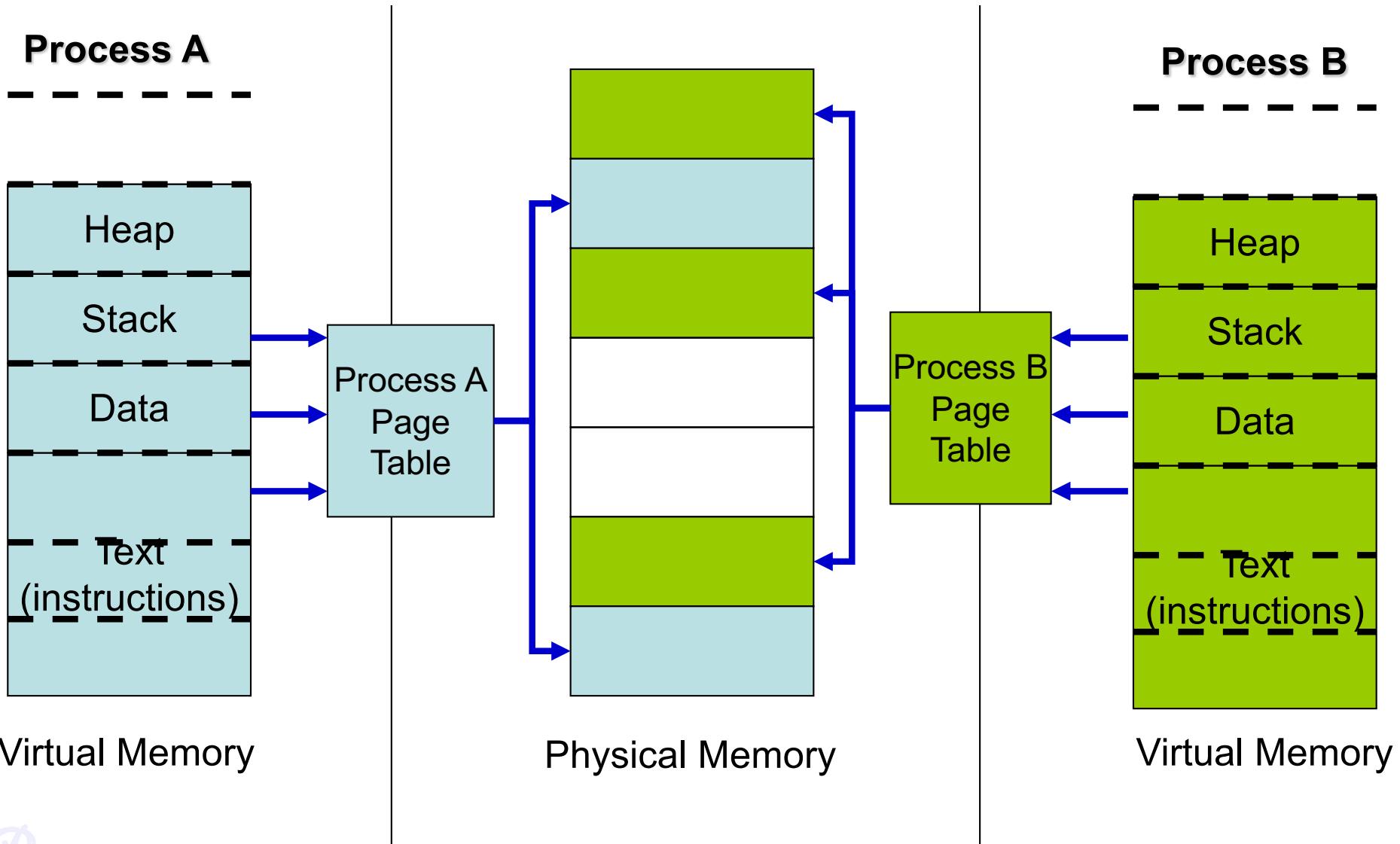


Process Address Space

- Each process has a different address space
- This is achieved by the use of virtual memory
- Ie. 0 to MAXVA are virtual memory addresses



Virtual Address Mapping



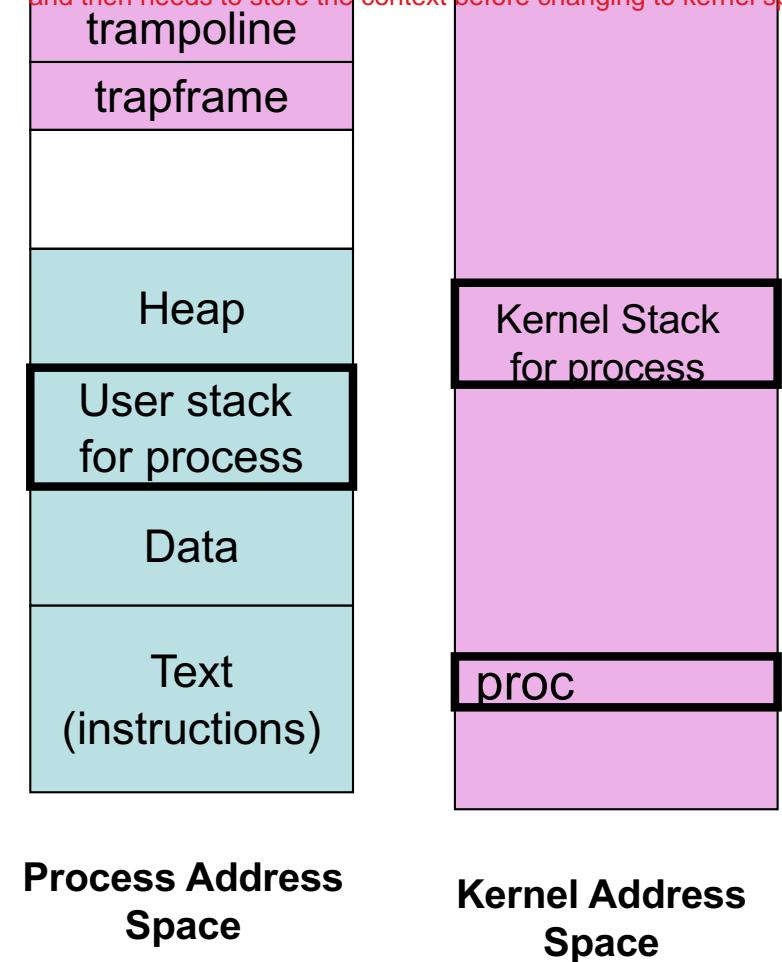
Advantages of Virtual Address Map

- **Isolation (private address space)**
 - One process cannot access another process' memory
- **Relocatable**
 - Data and code within the process is relocatable
- **Size**
 - Processes can be much larger than physical memory

Process Stacks

- Each process has 2 stacks
 - User space stack
 - Used when executing user code
 - Kernel space stack
 - Used when executing kernel code (for eg. during system calls)
 - Advantage : Kernel can execute even if user stack is corrupted
(Attacks that target the stack, such as buffer overflow attack, will not affect the kernel)

pink regions are accessible only to the kernel
the pink region in the user space is used when it has moved to sv mode
and then needs to store the context before changing to kernel space



the kernel does not have demand paging to prevent the possibility of the page fault handler getting swapped out and there being no way to handle page faults after that

the kernel is needed to store the context into satp
also to store the register context in memory locations of trapframes

Process Management in xv6

- Each process has a PCB (process control block) defined by `struct proc` in xv6
- Holds important process specific information
- Why?
 - Allows process to resume execution after a while
 - Keep track of resources used
 - Track the process state

Summary of entries in PCB

proc.c

```
struct proc proc[NPROC];
```

proc.h

```
86 struct proc {  
87     struct spinlock lock;  
88  
89     // p->lock must be held when using these:  
90     enum procstate state;          // Process state  
91     struct proc *parent;          // Parent process  
92     void *chan;                  // If non-zero, sleeping on chan  
93     int killed;                  // If non-zero, have been killed  
94     int xstate;                  // Exit status to be returned to parent's wait  
95     int pid;                     // Process ID  
96  
97     // these are private to the process, so p->lock need not be held.  
98     uint64 kstack;                // Bottom of kernel stack for this process  
99     uint64 sz;                   // Size of process memory (bytes)  
100    pagetable_t pagetable;        // Page table  
101    struct trapframe *tf;         // data page for trampoline.S  
102    struct context context;       // swtch() here to run process  
103    struct file *ofile[NOFILE];   // Open files  
104    struct inode *cwd;           // Current directory  
105    char name[16];               // Process name (debugging)  
106};
```

Entries in PCB

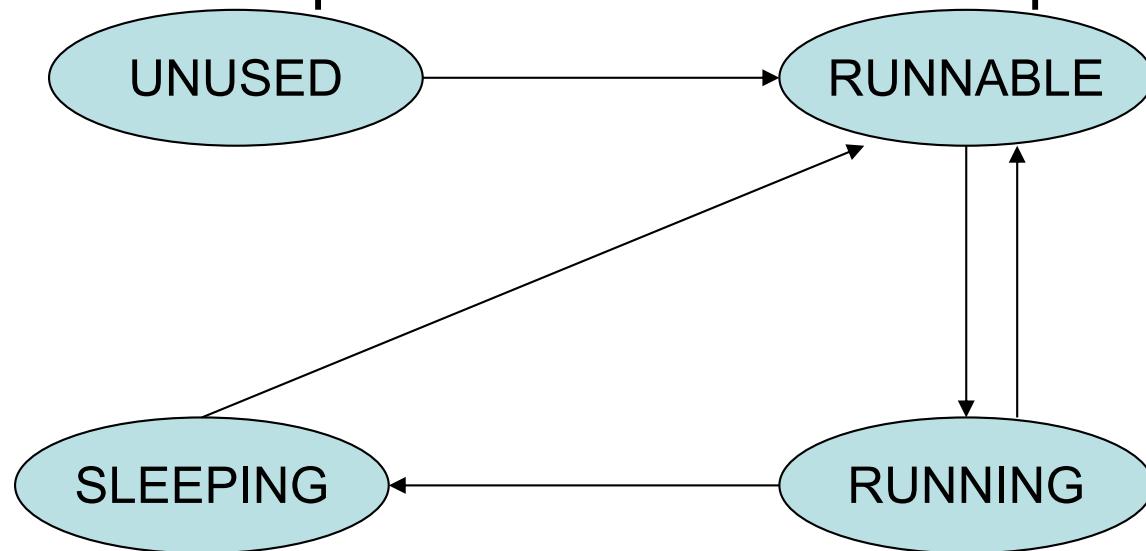
```
95 int pid; // Process ID
```

- PID
 - Process Identifier
 - Number incremented sequentially
 - When maximum is reached the size of the int type is limiting on the pid of the process
 - Reset and continue to increment.
 - This time skip already allocated PID numbers

Process States

```
90 enum procstate state; // Process state  
enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

Process State : specifies the state of the process



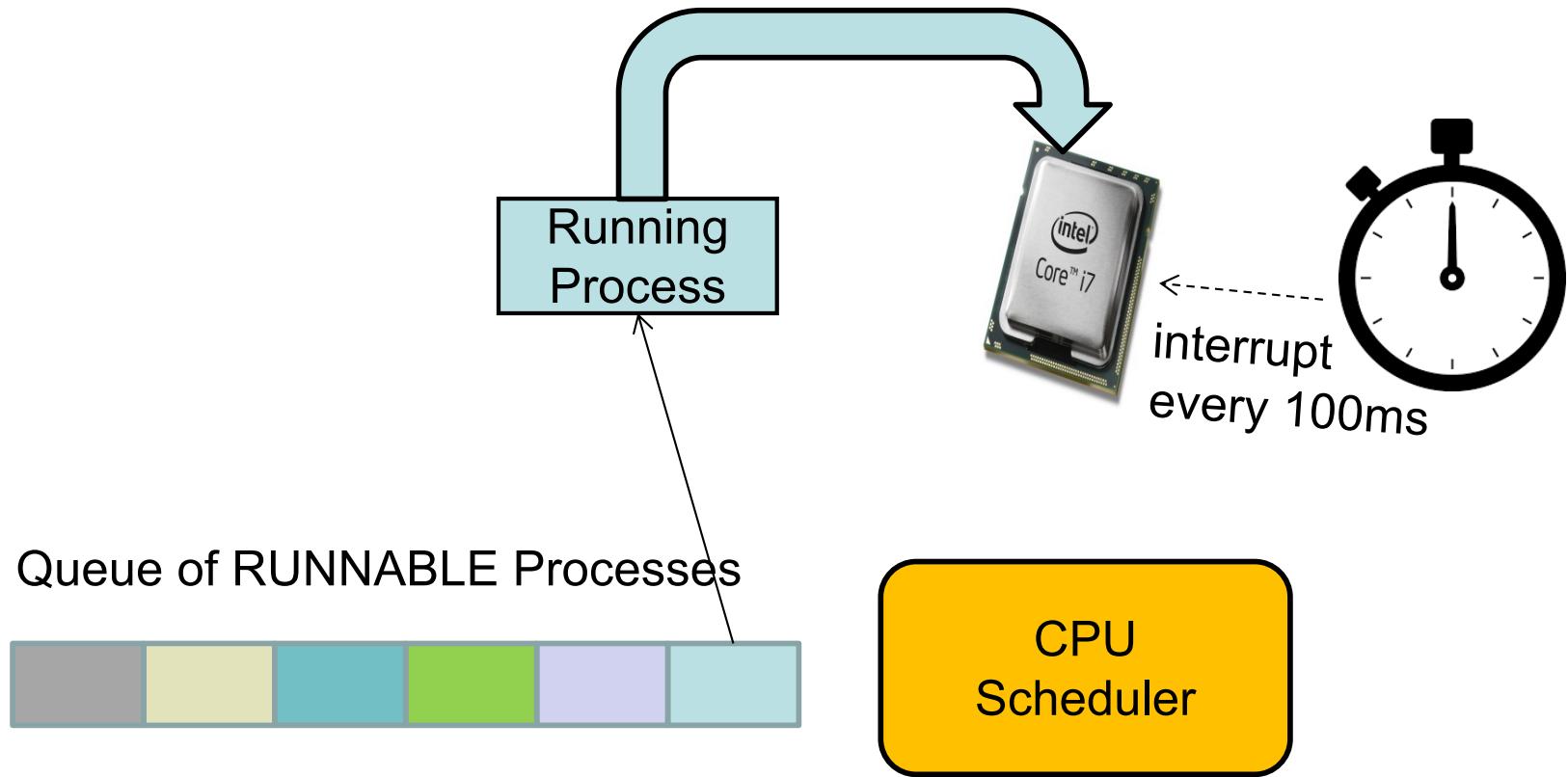
when a program is executed it is initially in the unused state. here the stack and other things required to run the process is setup and then finally assigns a pid to the process making it ready to be run on the cpu. this moves it to the runnable state where it is ready to run

when the cpu is allocated to a process in the runnable queue, it moves to the running state where the process is running on the cpu.

the number of running processes has to be equal to the number of cpus

- UNUSED → Unused PID
- RUNNABLE → Ready to run
- RUNNING → Currently executing
- SLEEPING → Blocked for an I/O
- Other states ZOMBIE (later)

Scheduling Runnable Processes

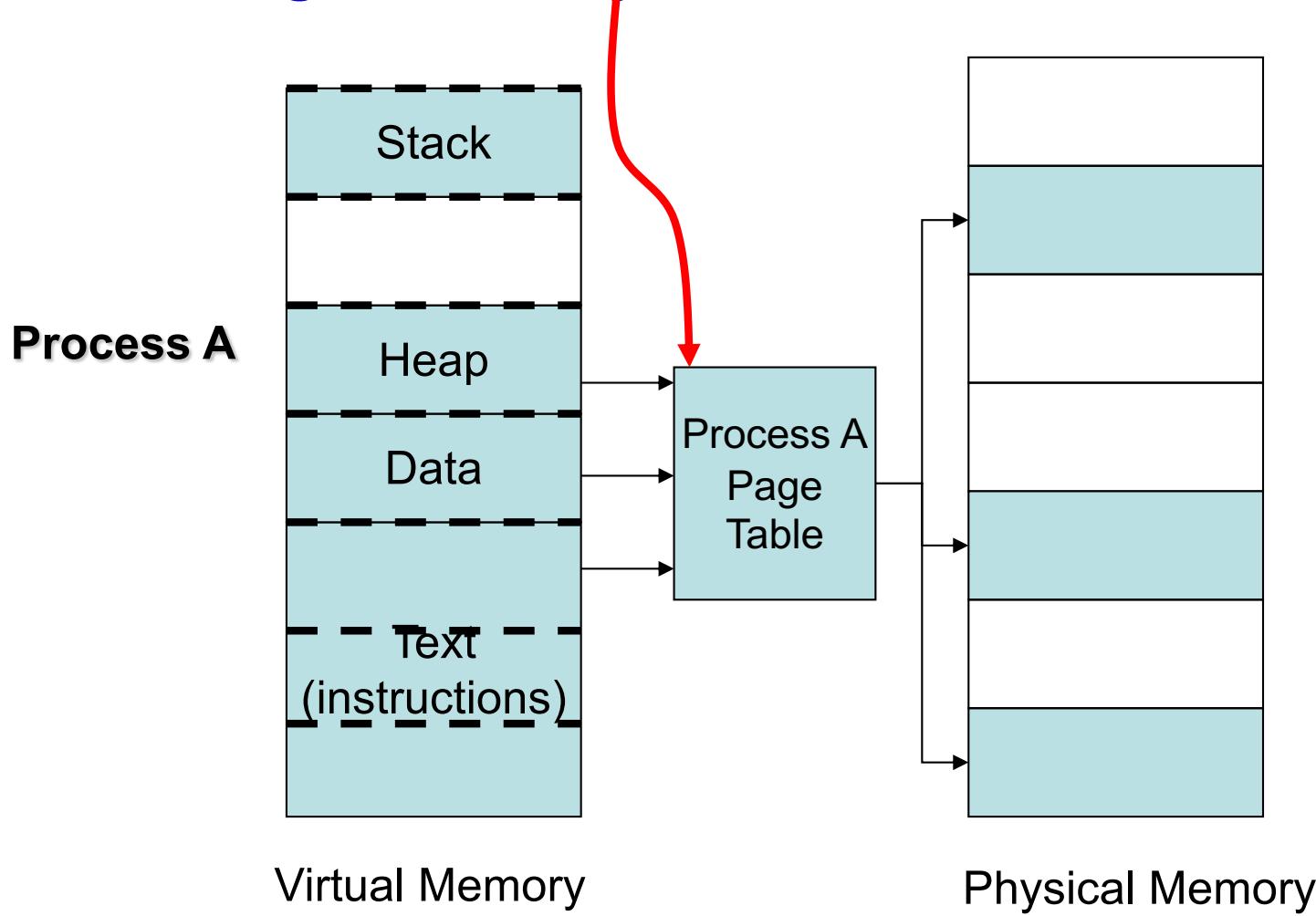


Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O
Scheduler picks another process from the ready queue
Performs a context switch

Page Directory Pointer

```
100 pagetable_t pagetable; // Page table
```

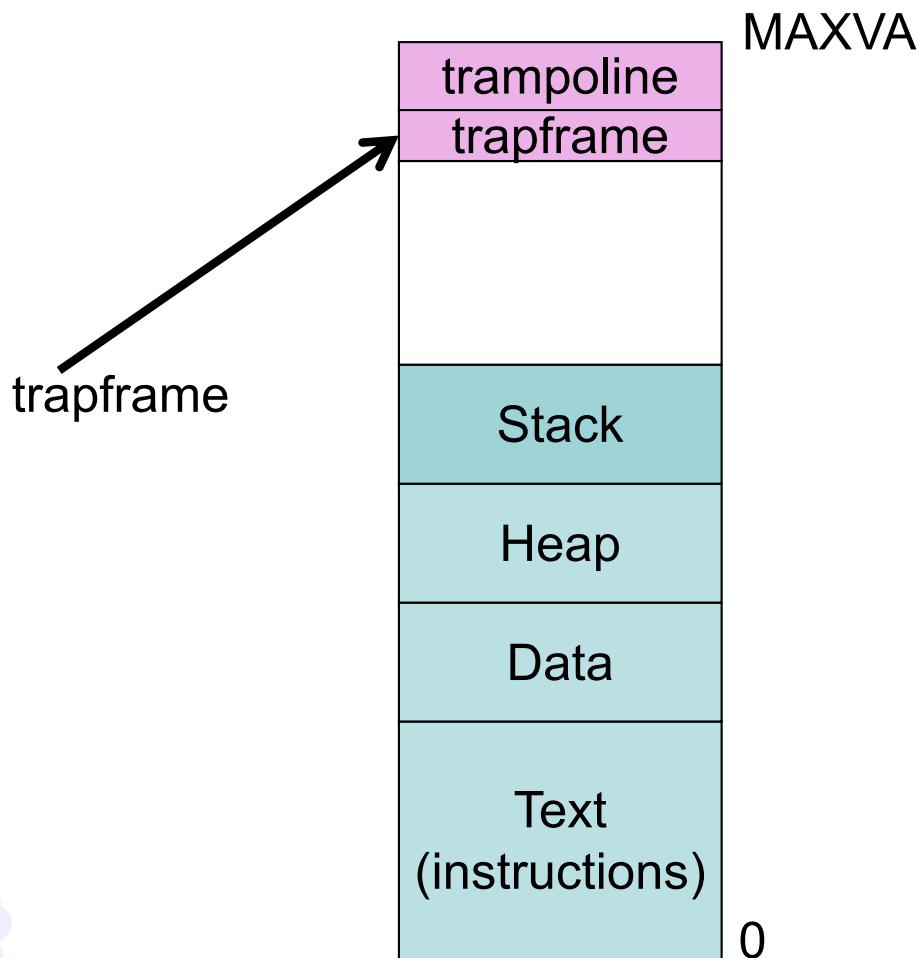
Page Directory Pointer



Entries in PCB

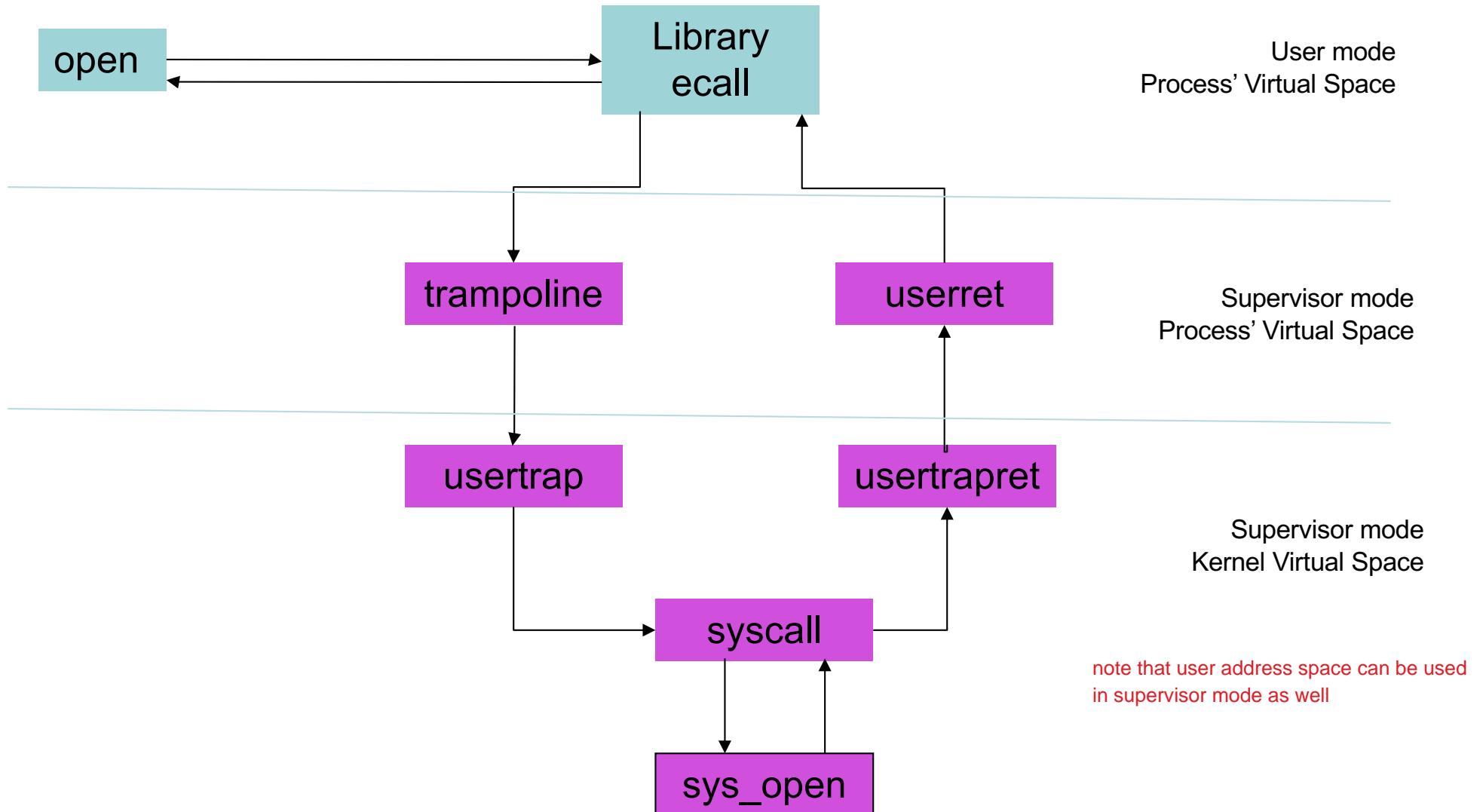
```
101 struct trapframe *tf; // data page for trampoline.S
```

- Pointer to trapframe



```
struct trapframe {  
    /* 0 */ uint64 kernel_satp;    // kernel page table  
    /* 8 */ uint64 kernel_sp;     // top of process's kernel stack  
    /* 16 */ uint64 kernel_trap;  // usertrap()  
    /* 24 */ uint64 epc;         // saved user program counter  
    /* 32 */ uint64 kernel_hartid; // saved kernel tp  
    /* 40 */ uint64 ra;  
    /* 48 */ uint64 sp;  
    /* 56 */ uint64 gp;  
    /* 64 */ uint64 tp;  
    /* 72 */ uint64 t0;  
    /* 80 */ uint64 t1;  
    /* 88 */ uint64 t2;  
    /* 96 */ uint64 s0;  
    /* 104 */ uint64 s1;  
    /* 112 */ uint64 a0;  
    /* 120 */ uint64 a1;  
    /* 128 */ uint64 a2;  
    /* 136 */ uint64 a3;
```

Progress of a system call



Context Pointer

- Context pointer
 - Contains registers used for context switches.
 - Callee saved registers

```
struct context {  
    uint64 ra;  
    uint64 sp;  
  
    // callee-saved  
    uint64 s0;  
    uint64 s1;  
    uint64 s2;  
    uint64 s3;  
    uint64 s4;  
    uint64 s5;  
    uint64 s6;  
    uint64 s7;  
    uint64 s8;  
    uint64 s9;  
    uint64 s10;  
    uint64 s11;  
};
```

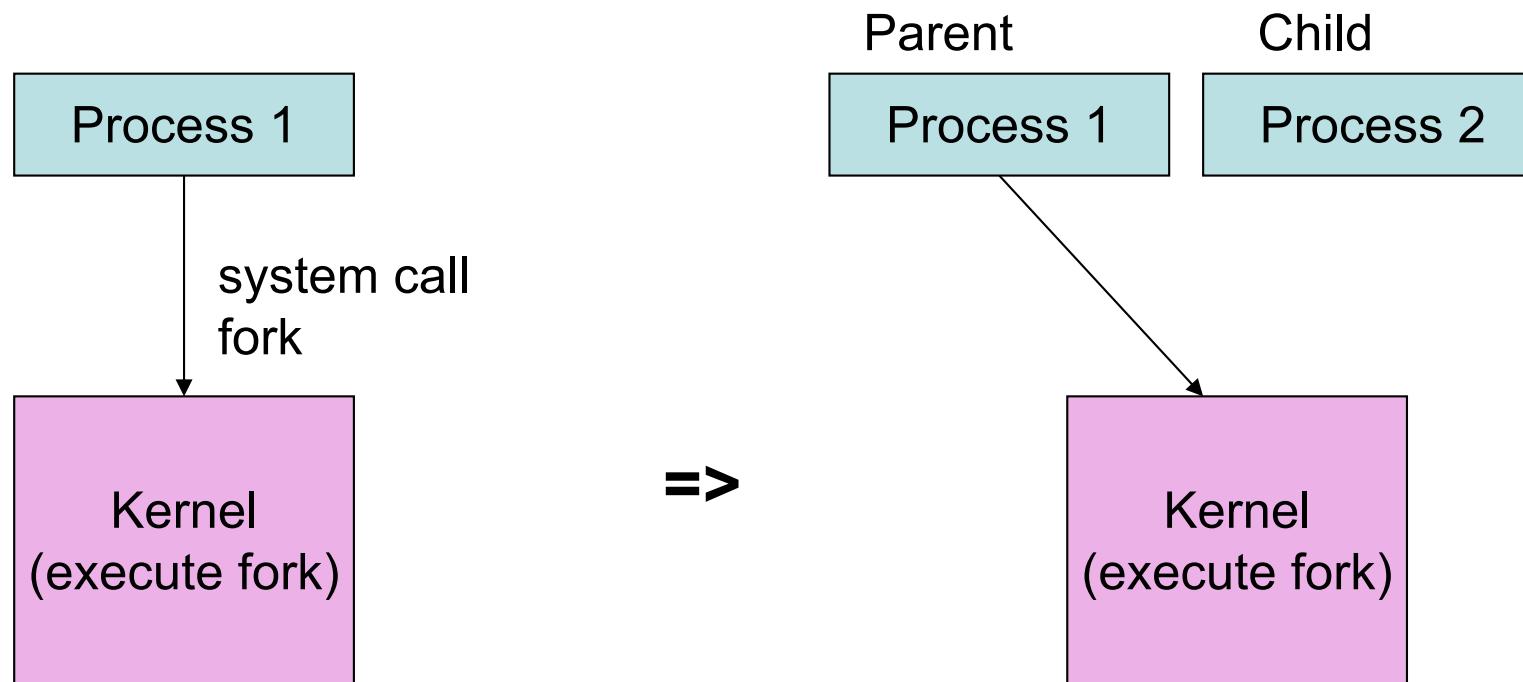
Storing procs in xv6

- In a globally defined array present in ptable
- NPROC is the maximum number of processes that can be present in the system (#define NPROC 64)
- Also present in ptable is a lock that serializes access to the array.

```
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

Creating a Process by Cloning

- Cloning
 - Child process is an exact replica of the parent
 - Fork system call



Creating a Process by Cloning (using fork system call)

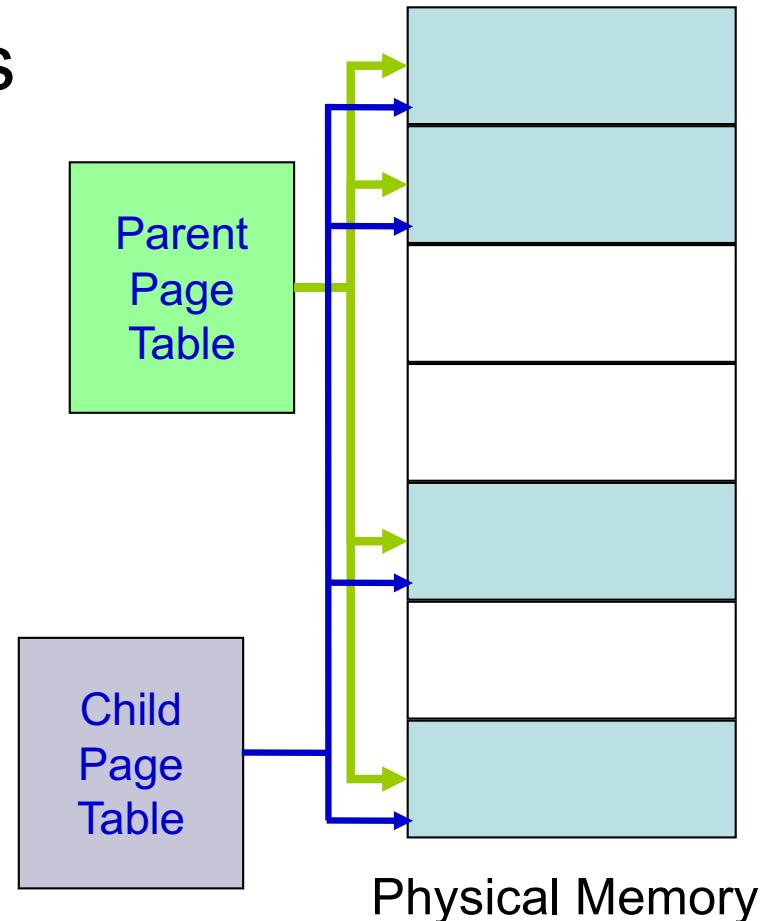
- In parent
 - fork returns child pid
- In child process
 - fork returns 0
- Other system calls
 - Wait, returns pid of an exiting child

```
int pid;

pid = fork();
if (pid > 0){
    printf("Parent : child PID = %d", pid);
    pid = wait();
    printf("Parent : child %d exited\n", pid);
} else{
    printf("In child process");
    exit(0);
}
```

Virtual Addressing Advantage (easy to make copies of a process)

- Making a copy of a process is called forking.
 - Parent (is the original)
 - child (is the new process)
- When fork is invoked,
 - child is an exact copy of parent
 - When fork is called all pages are shared between parent and child
 - Easily done by copying the parent's page tables



Modifying Data in Parent or Child

HOW ARE THE COW INSTANCES DETECTED?

Output

```
parent : 0
child  : 1
```

```
int i=0, pid;
pid = fork();
if (pid > 0){
    sleep(1);
    printf("parent : %d\n", i);
    wait();
} else{when the write is done, the page for i is duplicated and updated in the child
    i = i + 1;
    printf("child : %d\n", i);
}
```

in riscv the return values are placed in the a0 register - this is different for the parent and child
everything else is pretty much identical for the two processes

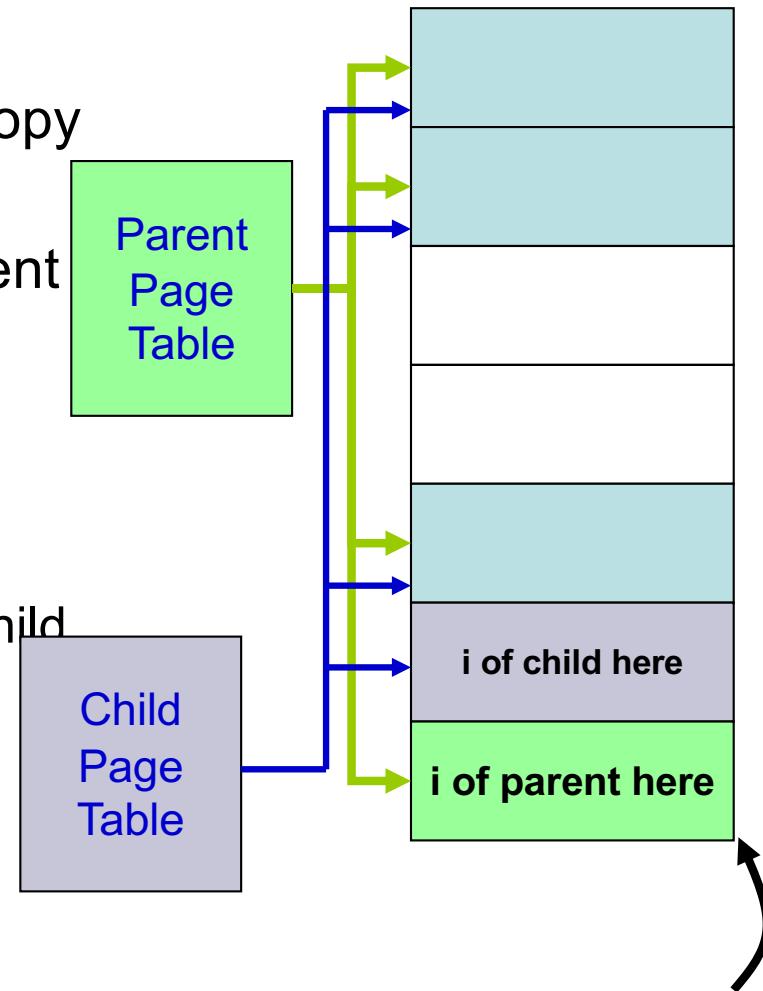
Executing a Program (exec system call)

- exec system call
 - Load into memory and then execute
- COW big advantage for exec
 - Time not wasted in copying pages.
 - Common code (for example shared libraries) would continue to be shared

```
int pid;  
  
pid = fork();  
if (pid > 0){  
    pid = wait();  
} else{  
    execlp("ls", "", NULL);  
    exit(0);
```

Copy on Write (COW)

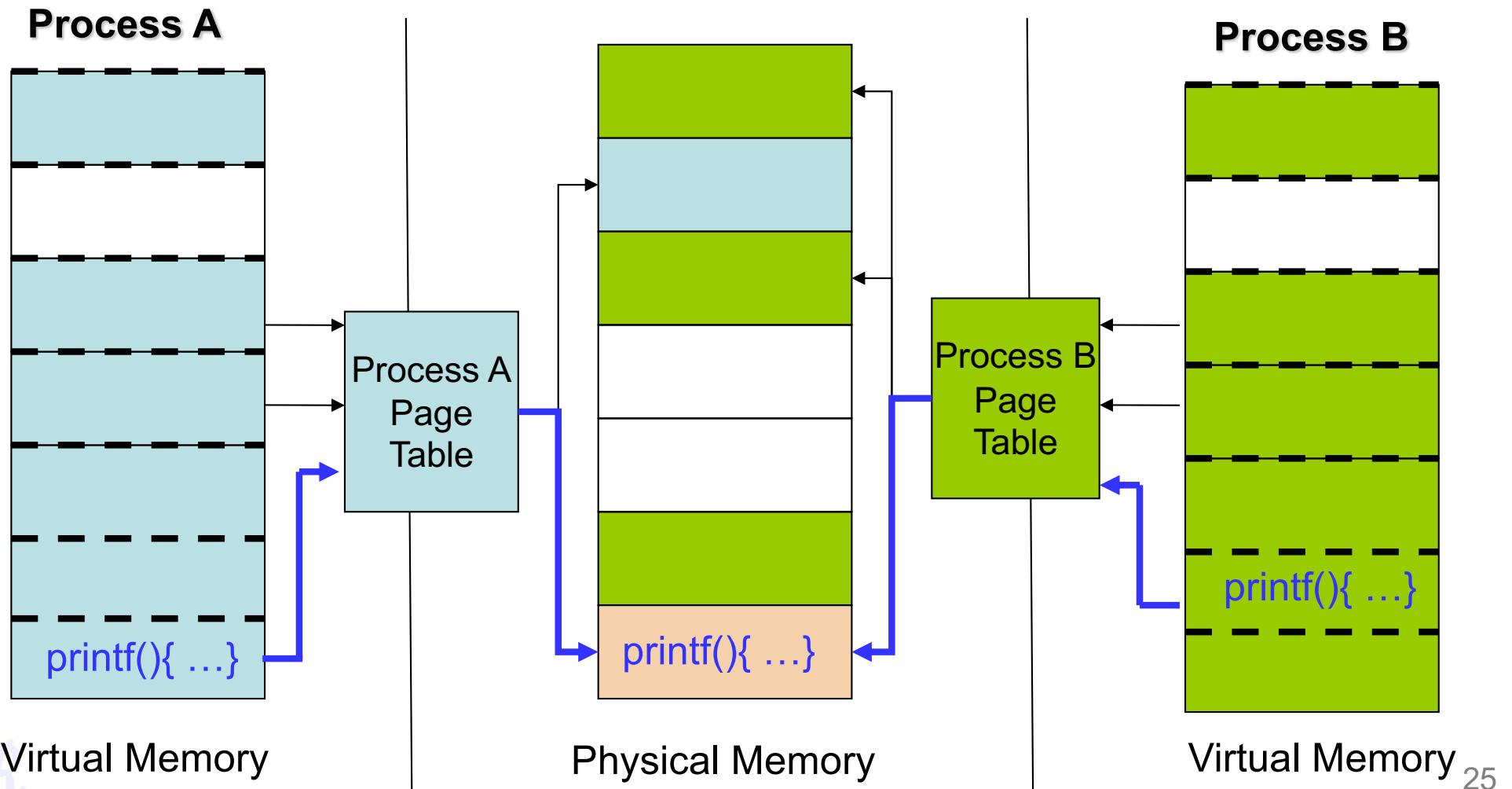
- When data in any of the shared pages change, OS intercepts and makes a copy of the page.
- Thus, parent and child will have different copies of **this** page
- **Why?**
 - A large portion of executables are not used.
 - Copying each page from parent and child would incur significant disk swapping.. huge performance penalties.
 - Postpone coping of pages as much as possible thus optimizing performance



This page now is no longer shared

Virtual Addressing Advantages (Shared libraries)

- Many common functions such as `printf` implemented in shared libraries
- Pages from shared libraries, shared between processes



How COW works

- When forking,
 - Kernel makes COW pages as read only
 - Any write to the pages would cause a page fault
 - The kernel detects that it is a COW page and duplicates the page

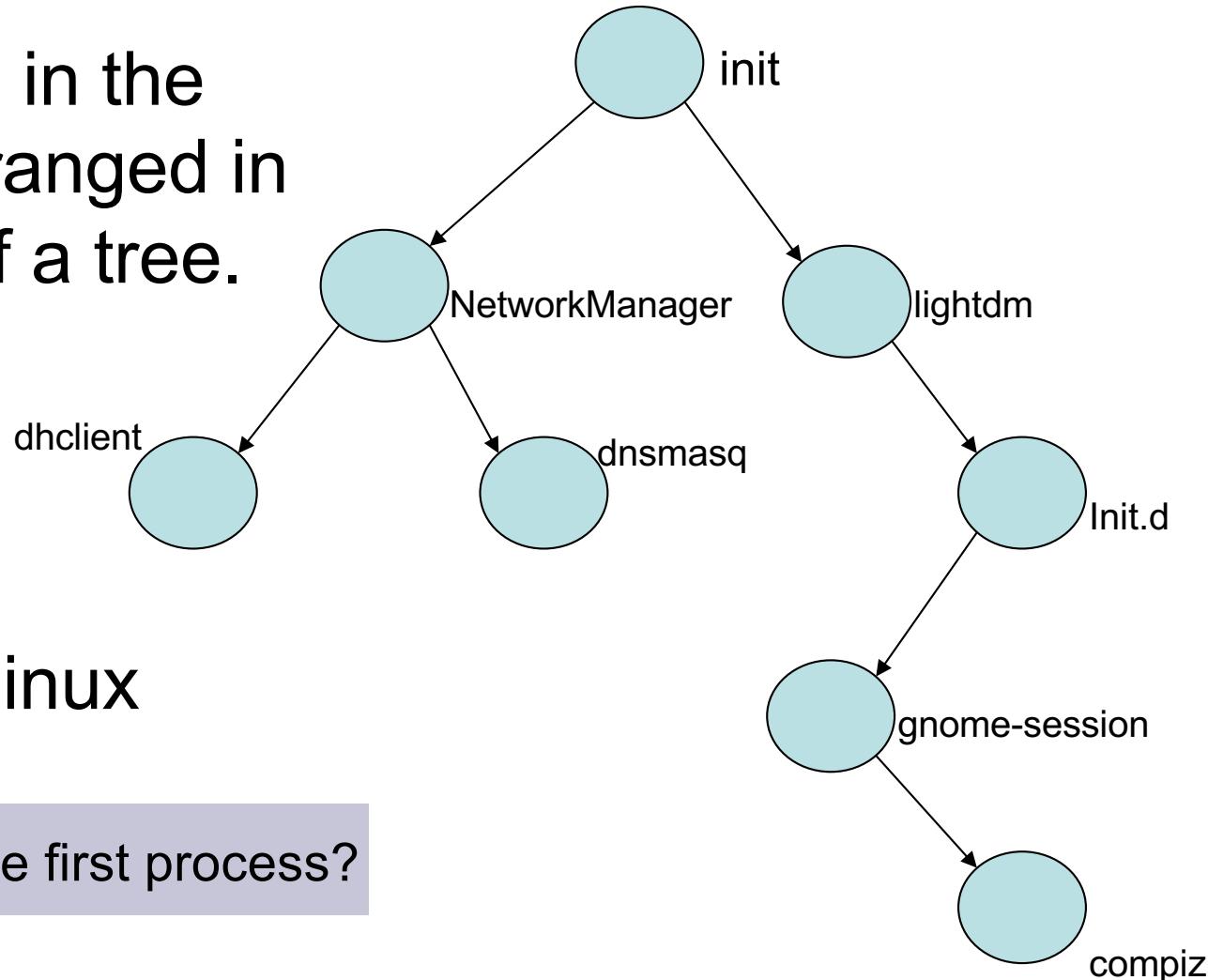
cow pages are detected as - when the fork is done, all the common pages with the W flag set is rewritten with the R flag
now whenever the daddy or the child tries to write to one of these pages, there is a page fault which then handles the copying functionality

The first process

- Unix : **/sbin/init**
 - Unlike the others, this is created by the kernel during boot
 - **Super parent.**
 - Responsible for forking all other processes
 - Typically starts several scripts present in **/etc/init.d** in Linux

Process tree

Processes in the system arranged in the form of a tree.



pstree in Linux

Who creates the first process?

Process Termination

- Voluntary : **exit(status)**
 - OS passes exit status to parent via **wait(&status)**
 - OS frees process resources
- Involuntary : **kill(pid, signal)**
 - Signal can be sent by another process or by OS
 - pid is for the process to be killed
 - **signal** a signal that the process needs to be killed
 - Examples : SIGTERM, SIGQUIT (ctrl+\), SIGINT (ctrl+c), SIGHUP

when **ctl+c** is done,
there is a keyboard interrupt
which the cpu detects
and then infers that **sigin**
must be sent to the process

the processes themselves will have a signal handler, which will look for the entry for the signal provided and appropriately does something to exit if there is no signal handler, then an exit is done by default

Zombies

when the exit() call is made, all that happens is all the open files are closed and the process state is changed to the zombie
the PCB is not cleared up yet. this is done only after the parent can read the return or exit status and then the OS cleans up

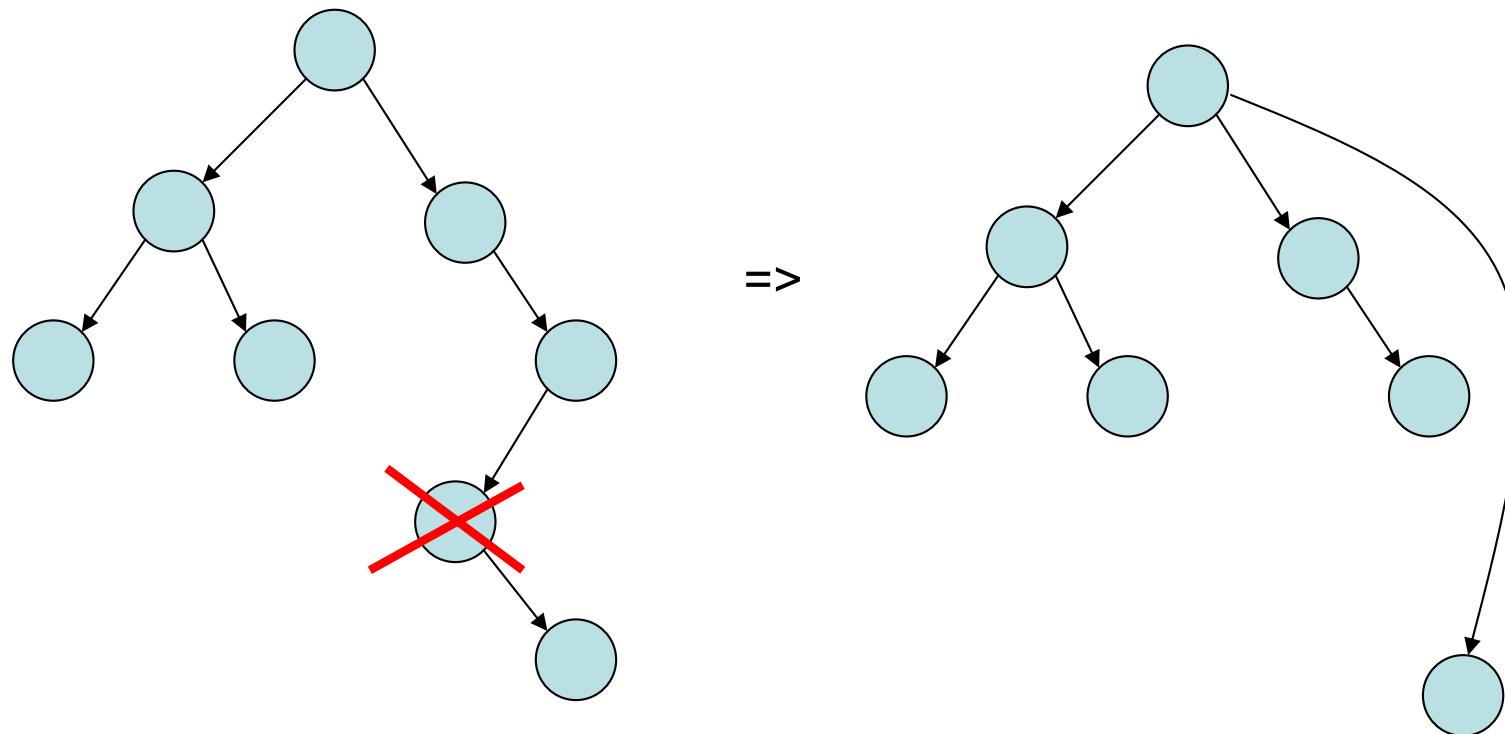
note that if this killed process has children, it makes the init process adopt these processes

- When a process terminates it becomes a **zombie** (or **defunct** process)
 - PCB in OS still exists even though program no longer executing
 - **Why?** So that the parent process can read the child's exit status (through **wait** system call)
- When parent reads status,
 - zombie entries removed from OS... **process reaped!**
- Suppose parent does' nt read status
 - Zombie will continue to exist infinitely ... **a resource leak**
 - These are typically found by a reaper process

CAN TWO PROCESSES DIE
AT THE SAME TIME?!

Orphans

- When a parent process terminates before its child
- Adopted by first process (/sbin/init)



Orphans contd.

- **Unintentional orphans**
 - When parent crashes
- **Intentional orphans**
 - Process becomes detached from user session and runs in the background
 - Called **daemons**, used to run background services
 - See **nohup**

The first process in xv6

The first process

- Initcode
- Creating the first process
 - main invokes userinit
 - userinit
 - allocate a process id, trapframe,
 - Create page directory
 - copy initcode.S to 0x0 this is the lowest address in the new address space that has been created
 - create a user stack at 0x0 (SP starts at 0x0+PGSIZE)
 - set process to runnable
 - the scheduler would then execute the process

userinit

proc.c

```
184 // od -t xc initcode
185 uchar initcode[] = {
186     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x05, 0x02,
187     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x05, 0x02,
188     0x9d, 0x48, 0x73, 0x00, 0x00, 0x00, 0x89, 0x48,
189     0x73, 0x00, 0x00, 0x00, 0xef, 0xf0, 0xbff, 0xff,
190     0x2f, 0x69, 0x6e, 0x69, 0x74, 0x00, 0x00, 0x01,
191     0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
192     0x00, 0x00, 0x00
193 };
194
195 // Set up first user process.
196 void
197 userinit(void)
198 {
199     struct proc *p;
200     p = allocproc(); effectively creates a new virtual address space
201     initproc = p;
202
203     // allocate one user page and copy init's instructions
204     // and data into it.
205     uvminit(p->pagetable, initcode, sizeof(initcode));
206     p->sz = PGSIZE;
207
208     // prepare for the very first "return" from kernel to user.
209     p->tf->epc = 0;           // user program counter
210     p->tf->sp = PGSIZE;    // user stack pointer
211
212     safestrcpy(p->name, "initcode", sizeof(p->name));
213     p->cwd = namei("/");
214     p->state = RUNNABLE;
215     release(&p->lock);
216 }
```

userinit

user/initcode.S

```
1 # Initial process execs /init.
2 # This code runs in user space.
3
4 #include "syscall.h"
5
6 # exec(init, argv)
7 .globl start
8 start:
9     la a0, init
10    la a1, argv
11    li a7, SYS_exec
12    ecall
13
14 # for(;;) exit();
15 exit:
16    li a7, SYS_exit
17    ecall
18    jal exit
19
20 # char init[] = "/init\0";
21 init:
22    .string "/init\0"
23
24 # char *argv[] = { init, 0 };
25 .p2align 2
26 argv:
27    .long init
28    .long 0
```

```
184 // od -t xc initcode
185 uchar initcode[] = {
186     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x05, 0x02,
187     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x05, 0x02,
188     0x9d, 0x48, 0x73, 0x00, 0x00, 0x00, 0x89, 0x48,
189     0x73, 0x00, 0x00, 0x00, 0xef, 0xf0, 0xbff, 0xff,
190     0x2f, 0x69, 0x6e, 0x69, 0x74, 0x00, 0x00, 0x01,
191     0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
192     0x00, 0x00, 0x00
193 };
194
195 // Set up first user process.
196 void
197 userinit(void)
198 {
199     struct proc *p;
200     p = allocproc();
201     initproc = p;
202
203     // allocate one user page and copy init's instru
204     // and data into it.
205     uvminit(p->pagetable, initcode, sizeof(initcode))
206     p->sz = PGSIZE;
207
208     // prepare for the very first "return" from kern
209     p->tf->epc = 0;           // user program counter
210     p->tf->sp = PGSIZE;     // user stack pointer
211
212     safestrcpy(p->name, "initcode", sizeof(p->name))
213     p->cwd = namei("/");
214     p->state = RUNNABLE;
215     release(&p->lock);
216 }
```

userinit

user/initcode.S

```
1 # Initial process execs /init.  
2 # This code runs in user space.  
3  
4 #include "syscall.h"  
5  
6 # exec(init, argv)  
7 .globl start  
8 start:  
9     la a0, init  
10    la a1, argv  
11    li a7, SYS_exec  
12    ecall  
13  
14 # for(;;) exit();  
15 exit:  
16    li a7, SYS_exit  
17    ecall  
18    jal exit  
19  
20 # char init[] = "/init\0";  
21 init:  
22     .string "/init\0"  
23  
24 # char *argv[] = { init, 0 };  
25 .p2align 2  
26 argv:  
27     .long init  
28     .long 0
```

```
184 // od -t xc initcode  
185 uchar initcode[] = {  
186     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x05, 0x02,  
187     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x05, 0x02,  
188     0x9d, 0x48, 0x73, 0x00, 0x00, 0x00, 0x89, 0x48,  
189     0x73, 0x00, 0x00, 0x00, 0xef, 0xf0, 0xbff, 0xff,  
190     0x2f, 0x69, 0x6e, 0x69, 0x74, 0x00, 0x00, 0x01,  
191     0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
192     0x00, 0x00, 0x00  
193 };  
194  
195 // Set up first user process.  
196 void  
197 userinit(void)  
198 {  
199     struct proc *p;  
200     p = allocproc();  
201     initproc = p;  
202  
203     // allocate one user page and copy init's instru  
204     // and data into it.  
205     uvminit(p->pagetable, initcode, sizeof(initcode))  
206     p->sz = PGSIZE;  
207  
208     // prepare for the very first "return" from kern  
209     p->tf->epc = 0;        // user program counter  
210     p->tf->sp = PGSIZE;   // user stack pointer  
211  
212     safestrcpy(p->name, "initcode", sizeof(p->name))  
213     p->cwd = namei("/");  
214     p->state = RUNNABLE;  
215     release(&p->lock);  
216 }
```

userinit

```
184 // od -t xc initcode
185 uchar initcode[] = {
186     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x05, 0x02,
187     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x05, 0x02,
188     0x9d, 0x48, 0x73, 0x00, 0x00, 0x00, 0x89, 0x48,
189     0x73, 0x00, 0x00, 0x00, 0xef, 0xf0, 0xbff, 0xff,
190     0x2f, 0x69, 0x6e, 0x69, 0x74, 0x00, 0x00, 0x01,
191     0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
192     0x00, 0x00, 0x00
193 };
194
195 // Set up first user process.
196 void
197 userinit(void)
198 {
199     struct proc *p;
200     p = allocproc();
201     initproc = p;
202
203     // allocate one u
204     // and data into
205     uvminit(p->pageta
206     p->sz = PGSIZE;
207
208     // prepare for th
209     p->tf->epc = 0;
210     p->tf->sp = PGSIZ
211
212     safestrcpy(p->nam
213     p->cwd = namei(".");
214     p->state = RUNNAB
215     release(&p->lock),
216 }
```

1. Find an unused proc entry
First entry will be used.
so, pid = 1
2. Create trapframe
3. Create empty user process' page table
 1. Map trampoline high
 2. Map trapframe just below it
4. Create context for process
 1. ra points to forkret
 2. Kernel stack pointer set

```
static struct proc*
allocproc(void)
{
    struct proc *p;

    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == UNUSED) {
            goto found;
        } else {
            release(&p->lock);
        }
    }
    return 0;
}

found:    checks the nextpid and gives it
p->pid = allocpid();

// Allocate a trapframe page.
if((p->tf = (struct trapframe *)kalloc()) == 0)
    release(&p->lock);
    return 0;
}

// An empty user page table.
p->pagetable = proc_pagetable(p);

// Set up new context to start executing at forkret,
// which returns to user space.
memset(&p->context, 0, sizeof p->context);
p->context.ra = (uint64)forkret;
p->context.sp = p->kstack + PGSIZE;

return p;
```

- 1
- 2
- 3
- 4

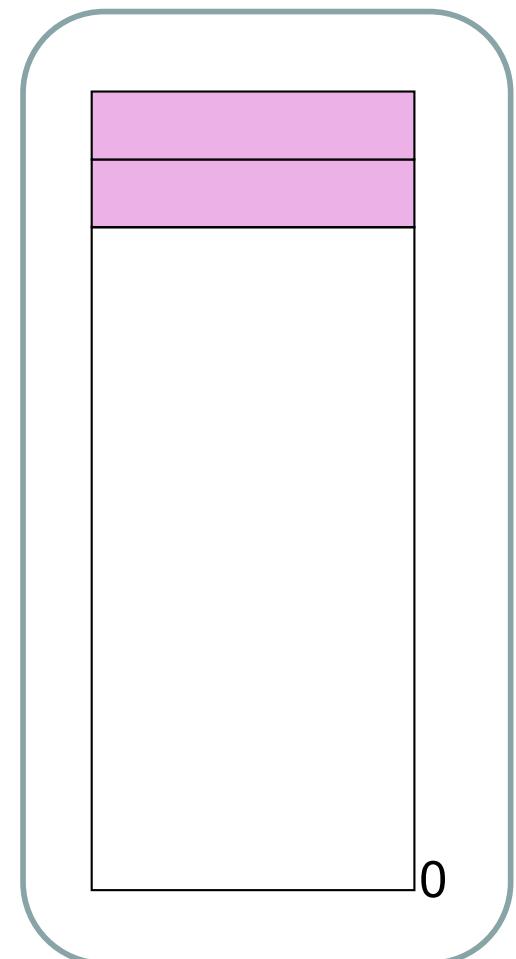
```

184 // od -t xc initcode
185 uchar initcode[] = {
186     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x05, 0x02,
187     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x05, 0x02,
188     0x9d, 0x48, 0x73, 0x00, 0x00, 0x00, 0x89, 0x48,
189     0x73, 0x00, 0x00, 0x00, 0xef, 0xf0, 0xbff, 0xff,
190     0x2f, 0x69, 0x6e, 0x69, 0x74, 0x00, 0x00, 0x01,
191     0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
192     0x00, 0x00, 0x00
193 };
194
195 // Set up first user process.
196 void
197 userinit(void)
198 {
199     struct proc *p;
200     p = allocproc();
201     initproc = p;
202
203     // allocate one u
204     // and data into
205     uvminit(p->pageta
206     p->sz = PGSIZE;
207
208     // prepare for th
209     p->tf->epc = 0;
210     p->tf->sp = PGSIZ
211
212     safestrcpy(p->n
213     p->cwd = namei(".");
214     p->state = RUNNAB
215     release(&p->lock),
216 }

```

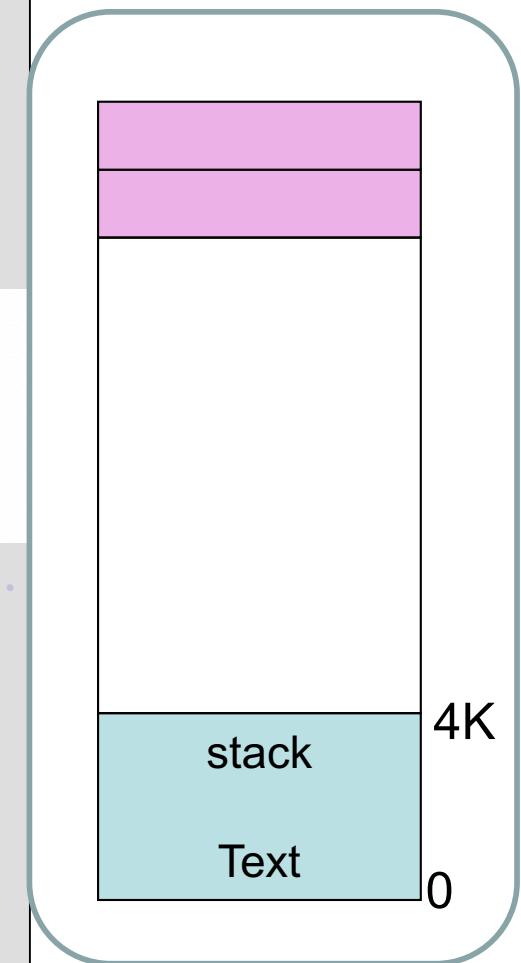
1. Find an unused proc entry
First entry will be used.
so, pid = 1
2. Create trapframe
3. Create empty user process' page table
 1. Map trampoline high
 2. Map trapframe just below it
4. Create context for process
 1. ra points to forkret
 2. Kernel stack pointer set

userinit



userinit

```
184 // od -t xc initcode
185 uchar initcode[] = {
186     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x05, 0x02,
187     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x05, 0x02,
188     0x9d, 0x48, 0x73, 0x00, 0x00, 0x00, 0x89, 0x48,
189     0x73, 0x00, 0x00, 0x00, 0xef, 0xf0, 0xbf, 0xff,
190     0x2f, 0x69, 0x6e, 0x69, 0x74, 0x00, 0x00, 0x01,
191     0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
192 };
193 }
194 void
195 uvminit(pagetable_t pagetable, uchar *src, uint sz)
196 {
197     char *mem;
198     if(sz >= PGSIZE)
199         panic("inituvvm: more than a page");
200     mem = kalloc();
201     memset(mem, 0, PGSIZE);
202     mappages(pagetable, 0, PGSIZE, (uint64)mem, PTE_W|PTE_R|PTE_X|PTE_U);
203     memmove(mem, src, sz);
204 }
205
206 // allocate one user page and copy init's instructions
207 // and data into it.
208 uvminit(p->pagetable, initcode, sizeof(initcode));
209 p->sz = PGSIZE;
210
211 // prepare for the very first "return" from kernel to user.
212 p->tf->epc = 0;           // user program counter
213 p->tf->sp = PGSIZE;    // user stack pointer
214 safestrcpy(p->name, "initcode", sizeof(p->name));
215 p->cwd = namei("/");
216 p->state = RUNNABLE;
217 release(&p->lock);
218 }
```



userinit

```
184 // od -t xc initcode
185 uchar initcode[] = {
186     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x05, 0x02,
187     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x05, 0x02,
188     0x9d, 0x48, 0x73, 0x00, 0x00, 0x00, 0x89, 0x48,
189     0x73, 0x00, 0x00, 0x00, 0xef, 0xf0, 0xbf, 0xff,
190     0x2f, 0x69, 0x6e, 0x69, 0x74, 0x00, 0x00, 0x01,
191     0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
192     0x00, 0x00, 0x00
193 };
194
195 // Set up first user process.
196 void
197 userinit(void)
198 {
199     struct proc *p;
200     p = allocproc();
201     initproc = p;
202
203     // allocate one user page and copy init's instructions
204     // and data into it.
205     uvminit(p->pagetable, initcode, sizeof(initcode));
206     p->sz = PGSIZE;
207
208     // prepare for the very first "return" from kernel to user.
209     p->tf->epc = 0;          // user program counter
210     p->tf->sp = PGSIZE;    // user stack pointer
211
212     safestrcpy(p->name, "initcode", sizeof(p->name));
213     p->cwd = namei("/");
214     p->state = RUNNABLE;
215     release(&p->lock);
216 }
```

Executing User Code

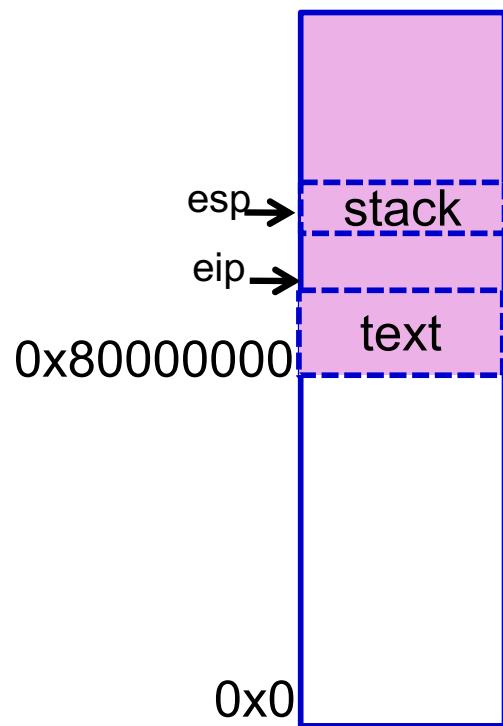
- The kernel stack of the process has a trap frame and context
- The process is set as RUNNABLE
- The scheduler is then invoked from main
 - main →scheduler

The initcode process is selected
(as it is the only process runnable)

- ...and is then executed

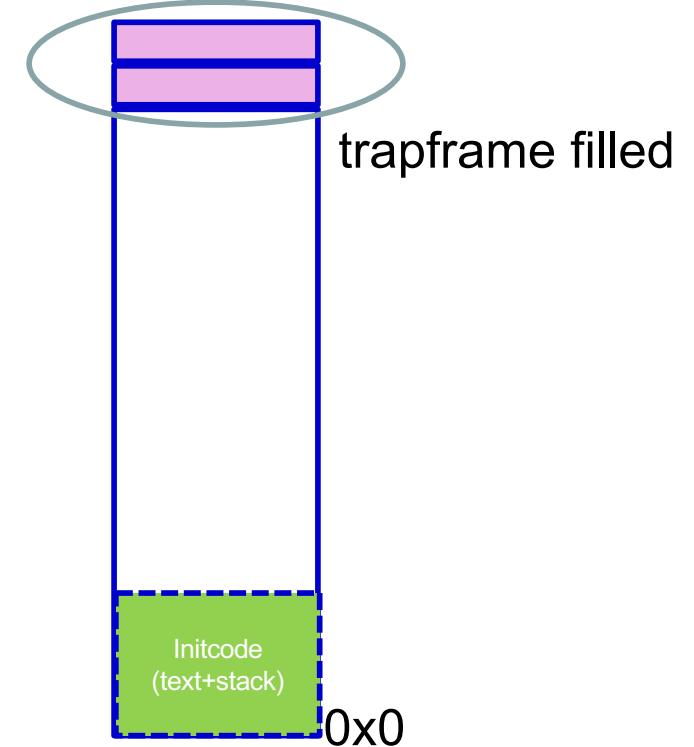
Scheduling the first process

What we need!



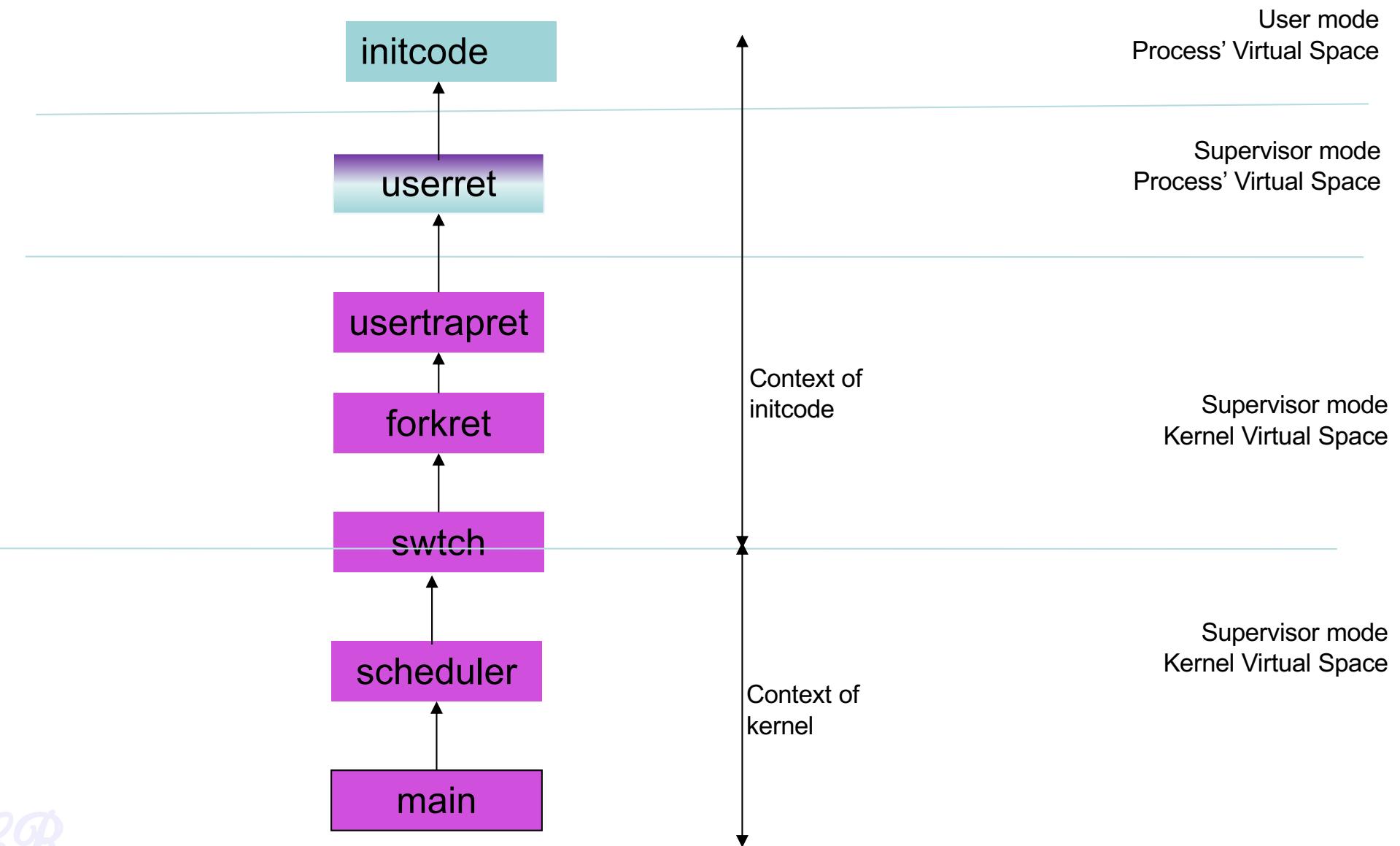
before userinit

Paging Unit uses the kernel page tables



Paging Unit uses the initcode's newly formed page tables

Progress of a system call



Scheduler ()

sched.c

```
438 void
439 scheduler(void)
440 {
441     struct proc *p;
442     struct cpu *c = mycpu();
443
444     c->proc = 0;
445     for(;;){
446         // Avoid deadlock by ensuring that devices can interrupt.
447         intr_on();
448
449         for(p = proc; p < &proc[NPROC]; p++) {
450             acquire(&p->lock);
451             if(p->state == RUNNABLE) {
452                 // Switch to chosen process. It is the process's job
453                 // to release its lock and then reacquire it
454                 // before jumping back to us.
455                 p->state = RUNNING;
456                 c->proc = p;
457                 swtch(&c->scheduler, &p->context); ←
458
459                 // Process is done running for now.
460                 // It should have changed its p->state before coming back. asm("%gs:4"); // this is a per cpu
461                 c->proc = 0;
462             }
463             release(&p->lock);
464         }
465     }
466 }
```

1. Find the process which is RUNNABLE.
 - currently only one process is RUNNABLE
 - set it to RUNNING
 - set CPU to run the initcode
2. context switch to initcode (swtch)

Switches context from scheduler's context to p
Function does not return

swtch

swtch.S

```
.globl swtch
swtch:
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)
```

a0 (1st parameter passed to swtch) contains pointer to scheduler context.
Here we store the CPU registers into the scheduler's context

```
ld ra, 0(a1)
ld sp, 8(a1)
ld s0, 16(a1)
ld s1, 24(a1)
ld s2, 32(a1)
ld s3, 40(a1)
ld s4, 48(a1)
ld s5, 56(a1)
ld s6, 64(a1)
ld s7, 72(a1)
ld s8, 80(a1)
ld s9, 88(a1)
ld s10, 96(a1)
ld s11, 104(a1)
```

a1 (2nd parameter passed to swtch) contains pointer to p->context. Here we load the context into CPU registers

```
ret
```

```
// Saved registers
struct context {
```

```
    uint64 ra;
    uint64 sp;
```

```
// callee-saved
```

```
    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
    uint64 s4;
    uint64 s5;
    uint64 s6;
    uint64 s7;
    uint64 s8;
    uint64 s9;
    uint64 s10;
    uint64 s11;
```

```
};
```

For initcode,
* ra set to forkret
* sp set to kstack+PGSIZE
(refer allocproc)

This function returns to forkret and SP= kstack of initcode



forkret

proc.c

```
// A fork child's very first scheduling by scheduler()
// will switch to forkret.
void
forkret(void)
{
    static int first = 1;

    // Still holding p->lock from scheduler.
    release(&myproc()->lock);

    if (first) {
        // File system initialization must be run in the context of a
        // regular process (e.g., because it calls sleep), and thus cannot
        // be run from main().
        first = 0;
        fsinit(ROOTDEV);
    }
    usertrapret();
}
```

usertrapret

trap.c

```
void
usertrapret(void)
{
    struct proc *p = myproc();

    // turn off interrupts, since we're switching
    // now from kerneltrap() to usertrap().
    intr_off();

    // send syscalls, interrupts, and exceptions to trampoline.S
    w_stvec(TRAMPOLINE + (uservec - trampoline));

    // set up trapframe values that uservec will need when
    // the process next re-enters the kernel.
    p->tf->kernel_satp = r_satp();           // kernel page table
    p->tf->kernel_sp = p->kstack + PGSIZE;   // process's kernel stack
    p->tf->kernel_trap = (uint64)usertrap;    // hartid for cpuid()
    p->tf->kernel_hartid = r_tp();           // hartid for cpuid()

    // set up the registers that trampoline.S's sret will use
    // to get to user space.

    // set S Previous Privilege mode to User.
    unsigned long x = r_sstatus();
    x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
    x |= SSTATUS_SPIE; // enable interrupts in user mode
    w_sstatus(x);

    // set S Exception Program Counter to the saved user pc.
    w_sepc(p->tf->epc);

    // tell trampoline.S the user page table to switch to.
    uint64 satp = MAKE_SATP(p->pagetable);

    // jump to trampoline.S at the top of memory, which
    // switches to the user page table, restores user registers,
    // and switches to user mode with sret.
    uint64 fn = TRAMPOLINE + (userret - trampoline);
    ((void (*)(uint64,uint64))fn)(TRAPFRAME, satp);
}
```

Set up interrupt handlers so in the trampoline

This will act as interrupt handler when
syscalls / interrupts occur.

These trapframe entries will permit handling
syscalls, interrupts and traps

Set SPP so that when interrupt
returns, switches to user mode.

p->tf->epc set to 0 in userinit

satp is going as first argument (a1 register)
to userret

userret

```
88 userret:  
89     # userret(TRAPFRAME, pagetable)  
90     # switch from kernel to user.  
91     # usertrapret() calls here.  
92     # a0: TRAPFRAME, in user page table.  
93     # a1: user page table, for satp.  
94  
95     # switch to the user page table.  
96     csrw satp, a1  
97     sfence.vma zero, zero  
98  
99     # put the saved user a0 in sscratch, so we  
100    # can swap it with our a0 (TRAPFRAME) in the last step.  
101    ld t0, 112(a0)  
102    csrw sscratch, t0  
103  
104    # restore all but a0 from TRAPFRAME  
105    ld ra, 40(a0)  
106    ld sp, 48(a0)  
107    ld gp, 56(a0)  
108    ld tp, 64(a0)  
109    ld t0, 72(a0)  
110    ld t1, 80(a0)  
111    ld t2, 88(a0)  
112    ld s0, 96(a0)  
113    ld s1, 104(a0)  
114    ld a1, 120(a0)  
115    ld a2, 128(a0)  
116    ld a3, 136(a0)  
117    ld a4, 144(a0)  
118    ld a5, 152(a0)  
119    ld a6, 160(a0)  
120    ld a7, 168(a0)  
121    ld s2, 176(a0)  
122    ld s3, 184(a0)  
123    ld s4, 192(a0)  
124    ld s5, 200(a0)  
125    ld s6, 208(a0)  
126    ld s7, 216(a0)  
127    ld s8, 224(a0)  
128    ld s9, 232(a0)  
129    ld s10, 240(a0)  
130    ld s11, 248(a0)  
131    ld t3, 256(a0)  
132    ld t4, 264(a0)  
133    ld t5, 272(a0)  
134    ld t6, 280(a0)
```

finally ... initcode.S ☺

- Invokes system call exec to invoke /init

exec('/init')

```
# Initial process execs /init.

#include "syscall.h"
#include "traps.h"

# exec(init, argv)
.globl start
start:
    pushl $argv
    pushl $init
    pushl $0 // where caller pc would be
    movl $SYS_exec, %eax
    int $T_SYSCALL

# for(;;) exit();
exit:
    movl $SYS_exit, %eax
    int $T_SYSCALL
    jmp exit

# char init[] = "/init\0";
init:
    .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
    .long init
    .long 0
```

init.c

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
    int pid, wpid;

    if(open("console", O_RDWR) < 0){
        mknod("console", 1, 1);
        open("console", O_RDWR);
    }
    dup(0); // stdout
    dup(0); // stderr

    for(;){
        printf(1, "init: starting sh\n");
        pid = fork();
        if(pid < 0){
            printf(1, "init: fork failed\n");
            exit();
        }
        if(pid == 0){
            exec("sh", argv);
            printf(1, "init: exec sh failed\n");
            exit();
        }
        while((wpid=wait()) >= 0 && wpid != pid)
            printf(1, "zombie!\n");
    }
}
```

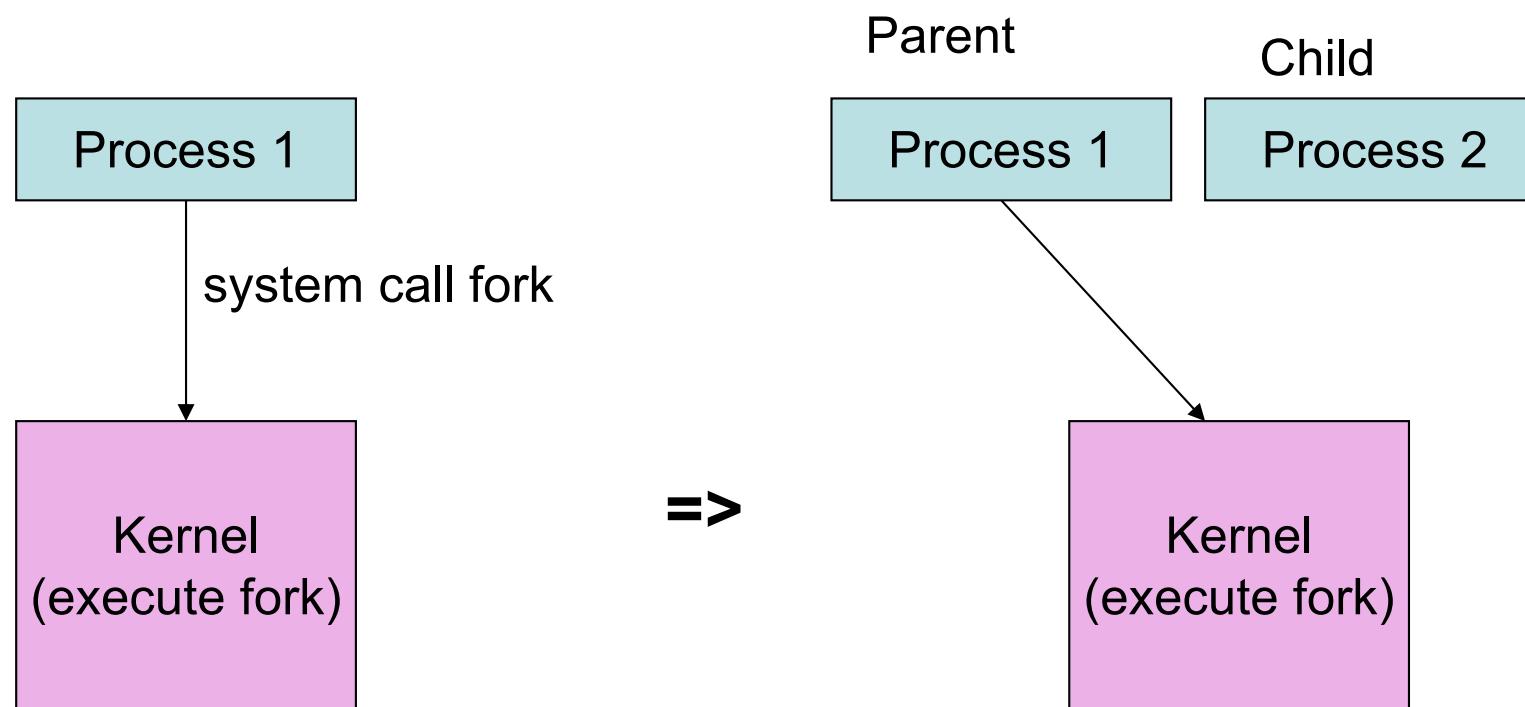
- forks and creates a shell (sh)

Handle orphans

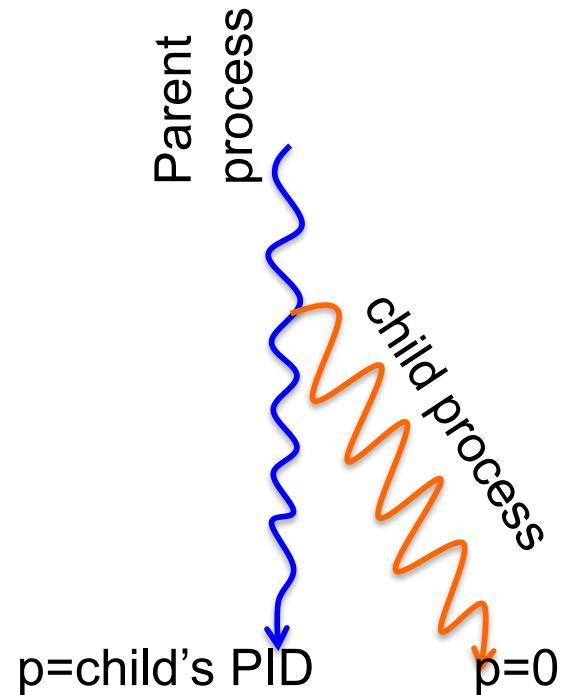
System Calls for Process Management

Creating a Process by Cloning

- Cloning
 - Child process is an exact replica of the parent
 - Fork system call

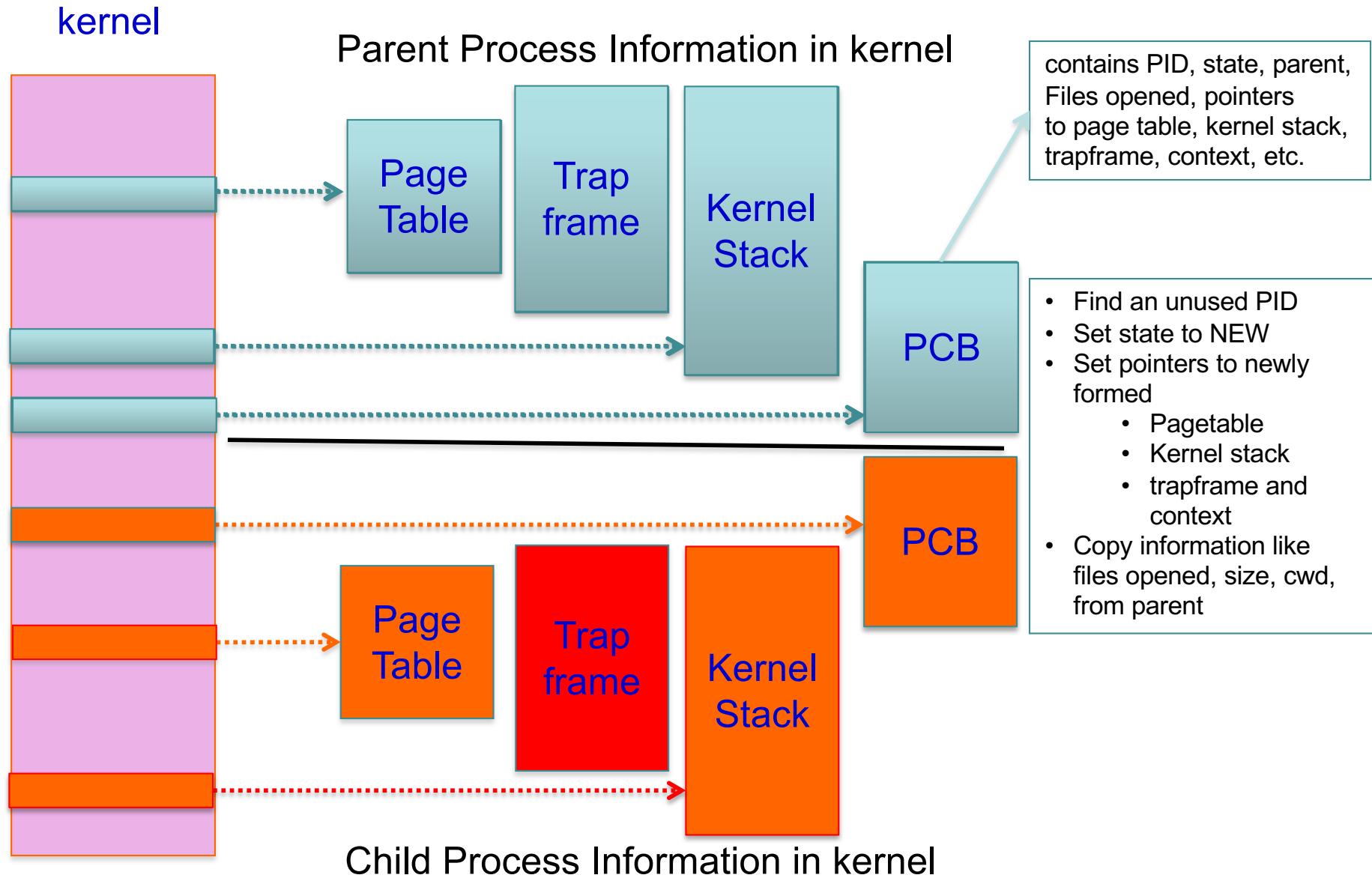


Creating a Process by Cloning (using fork system call)

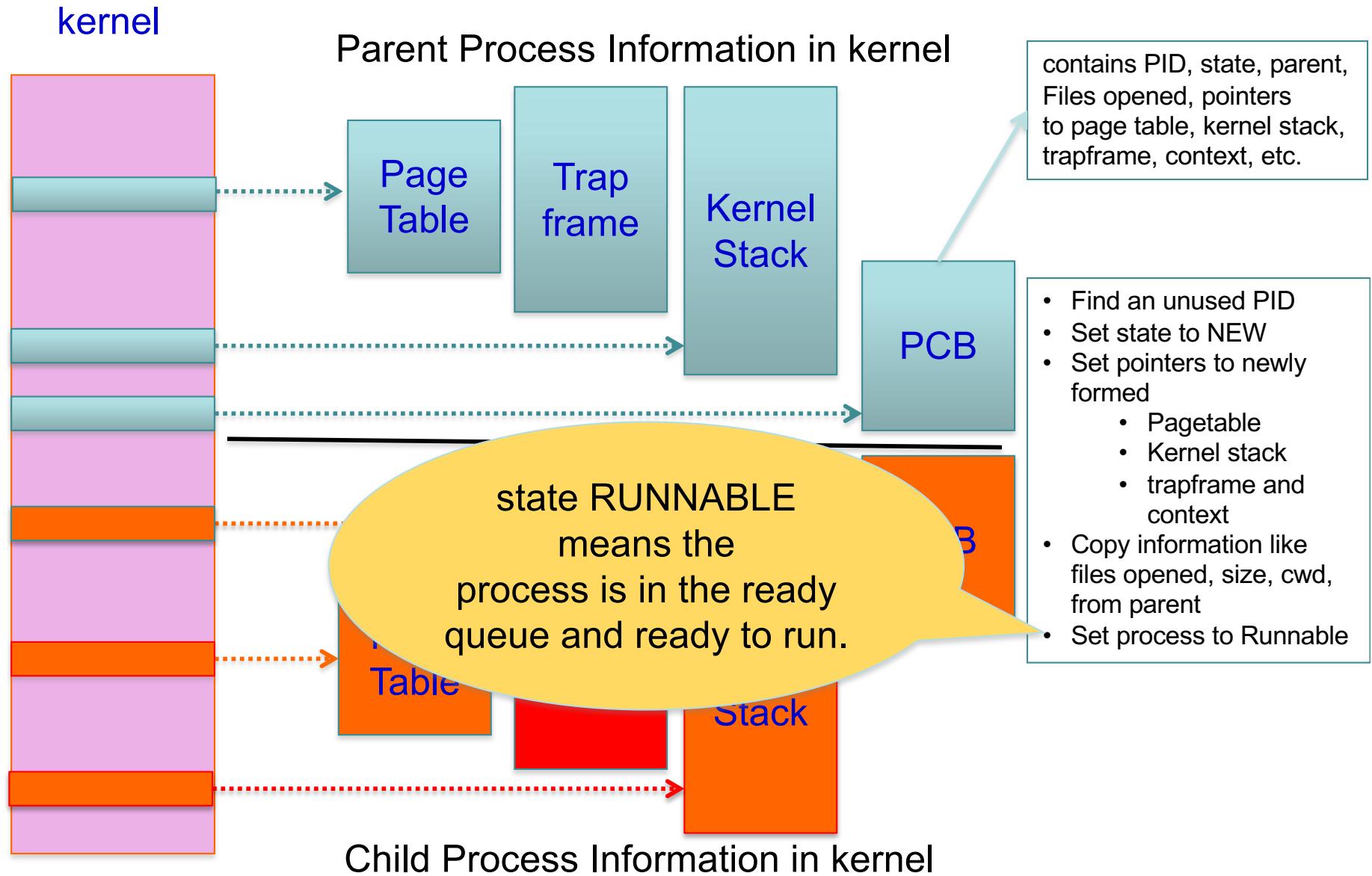


```
int p;  
  
p = fork();  
if (p > 0){  
    printf("Parent : child PID = %d", p);  
    p = wait();  
    printf("Parent : child %d exited\n", p);  
} else{  
    printf("In child process");  
    exit(0);  
}
```

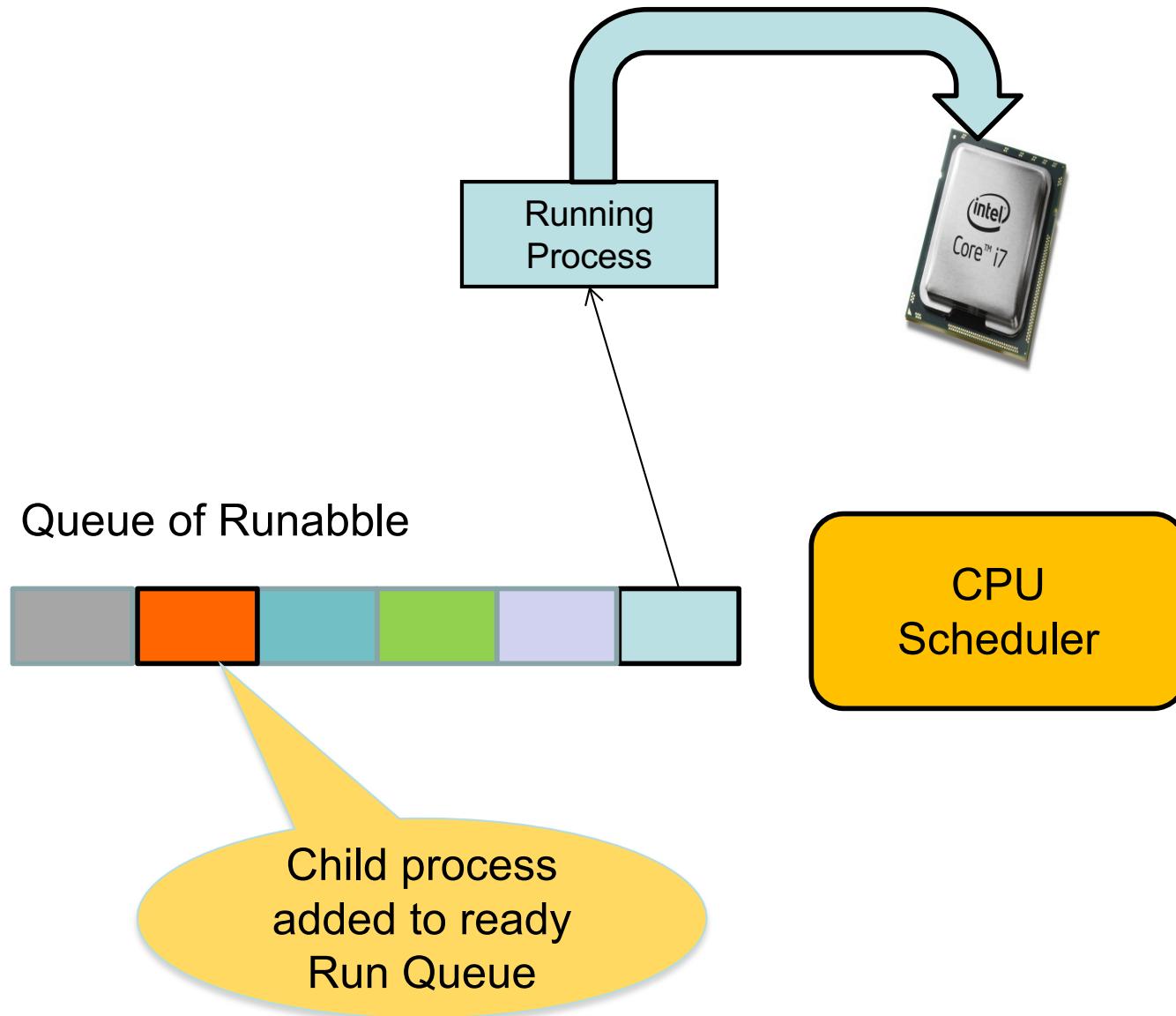
fork : from an OS perspective



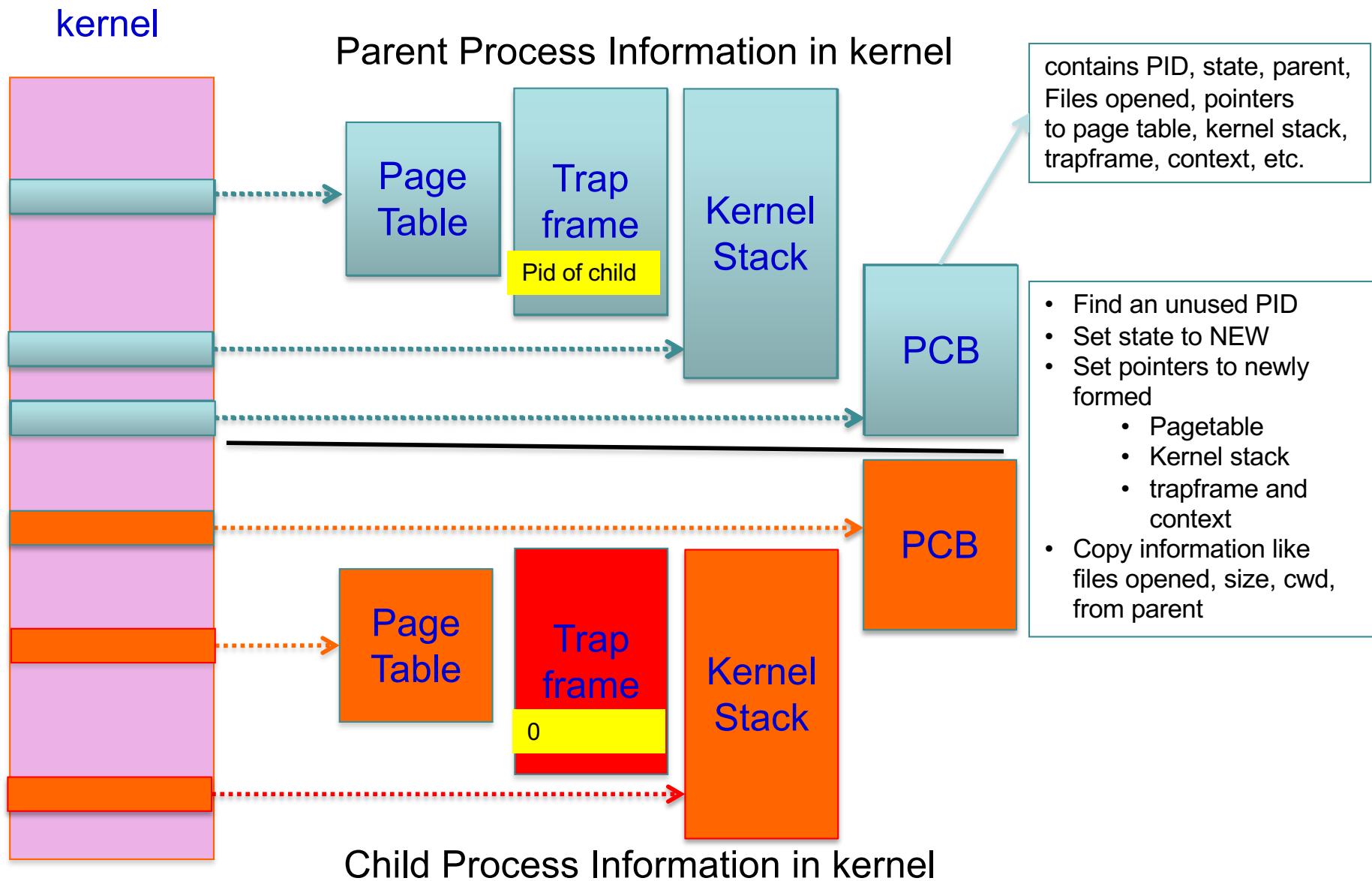
fork : from an OS perspective



Child Process in Ready Queue



Return from fork



fork, fills in the proc structure for a new process

```
86 struct proc {  
87     struct spinlock lock;  
88  
89     // p->lock must be held when using these:  
90     enum procstate state;          // Process state  
91     struct proc *parent;         // Parent process  
92     void *chan;                 // If non-zero, sleeping on chan  
93     int killed;                 // If non-zero, have been killed  
94     int xstate;                 // Exit status to be returned to parent's wait  
95     int pid;                    // Process ID  
96  
97     // these are private to the process, so p->lock need not be held.  
98     uint64 kstack;               // Bottom of kernel stack for this process  
99     uint64 sz;                  // Size of process memory (bytes)  
100    pagetable_t pagetable;       // Page table  
101    struct trapframe *tf;        // data page for trampoline.S  
102    struct context context;      // swtch() here to run process  
103    struct file *ofile[NOFILE];  // Open files  
104    struct inode *cwd;          // Current directory  
105    char name[16];              // Process name (debugging)  
106};
```

```

240 int
241 fork(void)
242 {
243     int i, pid;
244     struct proc *np;
245     struct proc *p = myproc();
246
247     // Allocate process.
248     if((np = allocproc()) == 0){
249         return -1;
250     }
251
252     // Copy user memory from parent to child.
253     if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
254         freeproc(np);
255         release(&np->lock);
256         return -1;
257     }
258     np->sz = p->sz;
259     np->parent = p;
260
261     // copy saved user registers.
262     *(np->tf) = *(p->tf);
263
264     // Cause fork to return 0 in the child.
265     np->tf->a0 = 0;
266
267     // increment reference counts on open file descriptors.
268     for(i = 0; i < NOFILE; i++)
269         if(p->ofile[i])
270             np->ofile[i] = filedup(p->ofile[i]);
271     np->cwd = idup(p->cwd);
272
273     safestrcpy(np->name, p->name, sizeof(p->name));
274
275     pid = np->pid;
276     np->state = RUNNABLE;
277     release(&np->lock);
278     return pid;
279 }

```

The diagram illustrates the flow of the xv6 fork function through various code segments and associated annotations:

- Initial Allocation (Line 248-250):** An annotation points to the first if-block: "(1) Pick an UNUSED proc. (2) allocate pid. and (3) allocate trapframe (4) create empty page directory; (5) setup context (p->context->ra = forkret); (p->context->sp = p->kstack + PGSIZE)".
- User Memory Copy (Line 253-257):** An annotation points to the if-block: "Copy page directory from the parent process (p->pagetable) to the child (np->pagetable)".
- Process State Initialization (Line 258-263):** A large brace groups these lines, with an annotation pointing to it: "Set size of np same as that of parent Set parent of np Copy trapframe from parent to child".
- Trapframe Modification (Line 265):** An annotation points to this line: "In child process, set eax register in trapframe to 0. This is what fork returns in the child process".
- File Descriptor Management (Line 267-271):** A large brace groups these lines, with an annotation pointing to it: "Other things... copy file pointer from parent, cwd, executable name".
- Final Process Setup (Line 273-274):** An annotation points to the safestrcpy call: "Child process is finally made runnable".
- Return and Lock Release (Line 275-279):** An annotation points to the final return statement: "Parent process returns the pid of the child".

The xv6 fork

```

240 int
241 fork(void)
242 {
243     int i, pid;
244     struct proc *np;
245
246     static struct proc*
247     allocproc(void)
248     {
249         struct proc *p;
250
251         for(p = proc; p < &proc[NPROC]; p++) {
252             acquire(&p->lock);
253             if(p->state == UNUSED) {
254                 goto found;
255             } else {
256                 release(&p->lock);
257             }
258         }
259
260         return 0;
261     }
262
263     /* Pick an UNUSED proc. */
264     if((p->tf = (struct trapframe *)kalloc()) == 0){
265         release(&p->lock);
266         return 0;
267     }
268
269     /* An empty user page table. */
270     p->pagetable = proc_pagetable(p);
271
272     /* Set up new context to start executing at forkret,
273      * which returns to user space.
274     */
275     memset(&p->context, 0, sizeof p->context);
276     p->context.ra = (uint64)forkret;
277     p->context.sp = p->kstack + PGSIZE;
278
279     return p;
280 }

```

The xv6 fork

- (1) Pick an UNUSED proc.
- (2) allocate pid.
- (3) allocate trapframe
- (4) create empty page directory;
- (5) setup context
 - (p->context->ra = forkret);
 - (p->context->sp = p->kstack + PGSIZE)

forkret would ensure that the new child process gets 0 when fork returns

Copying Page Tables of Parent

- **uvmcopy** (in `vm.c`)
 - replicates parents memory pages
 - Constructs new table pointing to the new pages
 - Steps involved
 1. For each virtual page of the parent (starting from 0 to its sz)
 - i. Find its page table entry (function `walk`)
 - ii. Use `kalloc` to allocate a page (`mem`) in memory for the child
 - iii. Use `memmove` to copy the parent page to `mem`
 - iv. Use `mappages` to add a page table entry for `mem`

xv6 does not support COW

Register modifications w.r.t. parent

Registers modified in child process

- `%eax = 0` so that `pid = 0` in child process
- `%eip = forkret` so that child exclusively executes function `forkret`

Exit system call

```
int pid;

pid = fork();
if (pid > 0){
    printf("Parent : child PID = %d", pid);
    pid = wait();
    printf("Parent : child %d exited\n", pid);
} else{
    printf("In child process");
    exit();
}
```

exit internals

- **init**, the first process, can never exit
- For all other processes on exit,
 1. Decrement the usage count of all open files
 - If usage count is 0, close file
 - Drop reference to in-memory inode
 2. wakeup parent
 - If parent state is **sleeping**, make it **runnable**
 - Needed, cause parent may be sleeping due to a wait
 3. Make init adopt children of exited process
 4. Set process state to **ZOMBIE**
 5. Force context switch to scheduler

note : page directory, kernel stack, not deallocated here

```

311 exit(int status)
312 {
313     struct proc *p = myproc();
314
315     if(p == initproc)
316         panic("init exiting");
317
318     // Close all open files.
319     for(int fd = 0; fd < NOFILE; fd++){
320         if(p->ofile[fd]){
321             struct file *f = p->ofile[fd];
322             fileclose(f);
323             p->ofile[fd] = 0;
324         }
325     }
326
327     begin_op();
328     input(p->cwd);
329     end_op();
330     p->cwd = 0;
331
332     acquire(&initproc->lock);
333     wakeup1(initproc);
334     release(&initproc->lock);
335
336     acquire(&p->lock);
337     struct proc *original_parent = p->parent;
338     release(&p->lock);
339
340     acquire(&original_parent->lock);
341
342     acquire(&p->lock);
343
344     // Give any children to init.
345     reparent(p);
346
347     // Parent might be sleeping in wait().
348     wakeup1(original_parent);
349
350     p->xstate = status;
351     p->state = ZOMBIE;
352
353     release(&original_parent->lock);
354
355     // Jump into the scheduler, never to return.
356     sched();
357     panic("zombie exit");
358 }

```

initproc can never exit

exit

Close all open files

Decrement in-memory inode usage

Wakeup parent of child.
The child may be an orphan, in
which case init must be woken up

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

Set exiting process state to zombie

invoke the scheduler, which
performs a context switch

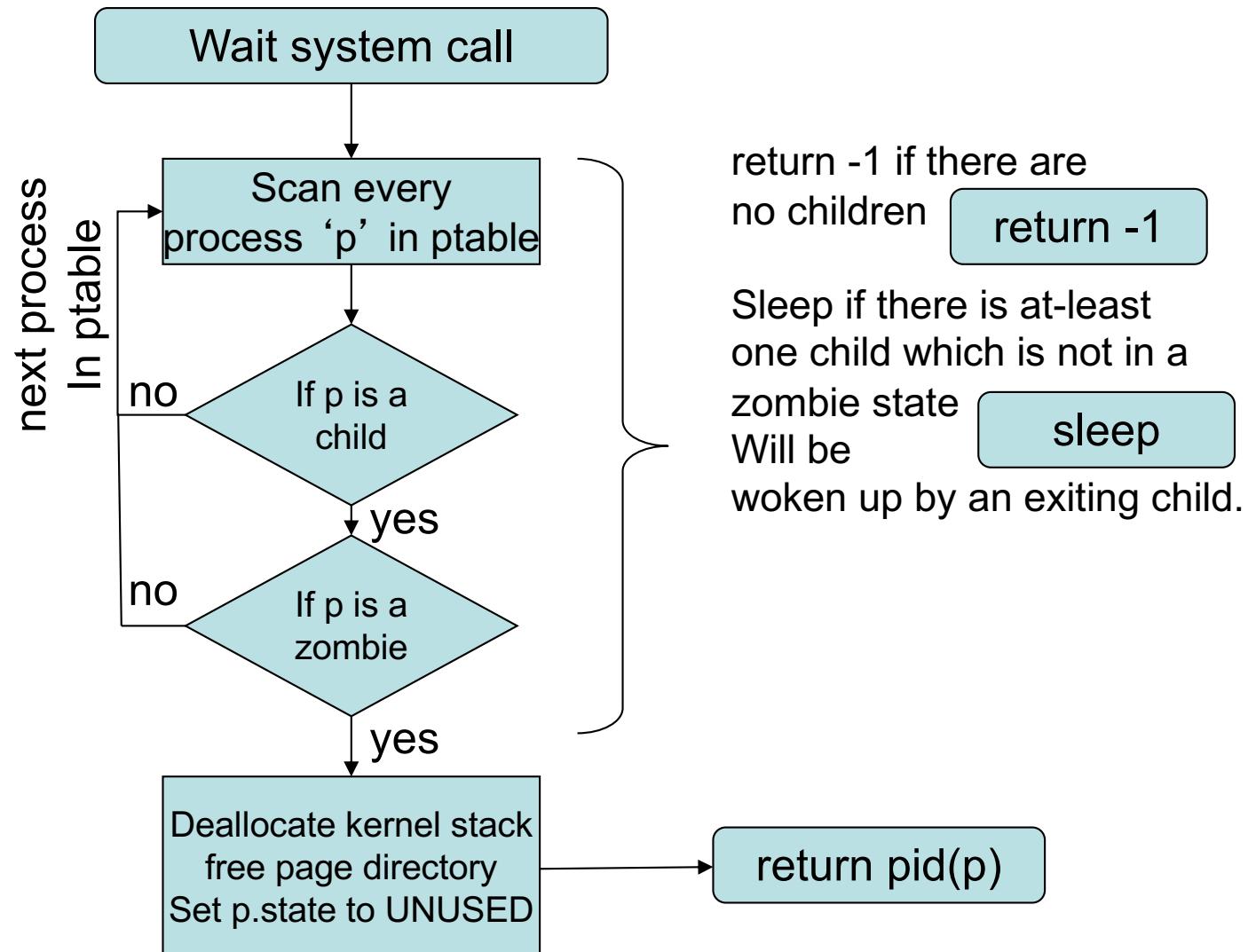
wait system call

- Invoked in parent parent
- Parent ‘waits’ until child exits

```
int pid;

pid = fork();
if (pid > 0){
    printf("Parent : child PID = %d", pid);
    pid = wait();
    printf("Parent : child %d exited\n", pid);
} else{
    printf("In child process");
    exit();
}
```

wait internals



```

363 wait(uint64 addr)
364 {
365     struct proc *np;
366     int havekids, pid;
367     struct proc *p = myproc();
368
369     // hold p->lock for the whole time to avoid lost
370     // wakeups from a child's exit().
371     acquire(&p->lock);
372     for(;;){
373         // Scan through table looking for exited children.
374         havekids = 0;
375         for(np = proc; np < &proc[NPROC]; np++){
376             // this code uses np->parent without holding np->lock.
377             // acquiring the lock first would cause a deadlock,
378             // since np might be an ancestor, and we already hold p->lock.
379             if(np->parent == p){
380                 // np->parent can't change between the check and the acquire()
381                 // because only the parent changes it, and we're the parent.
382                 acquire(&np->lock);
383                 havekids = 1;
384                 if(np->state == ZOMBIE){
385                     // Found one.
386                     pid = np->pid;
387                     if(addr != 0 && copyout(p->pagetable, addr, (char *)&np->xstate,
388                                         sizeof(np->xstate)) < 0) {
389                         release(&np->lock);
390                         release(&p->lock);
391                         return -1;
392                     }
393                     freeproc(np);
394                     release(&np->lock);
395                     release(&p->lock);
396                     return pid;
397                 }
398                 release(&np->lock);
399             }
400         }
401         // No point waiting if we don't have any children.
402         if(!havekids || p->killed){
403             release(&p->lock);
404             return -1;
405         }
406         // Wait for a child to exit.
407         sleep(p, &p->lock); //DOC: wait-sleep
408     }
409 }
410

```

wait

For every process in table, find if (a) p is a parent and (b) if the process is a zombie.

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

Copy the exit status of the child process from kernel to user space.

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

Process will be set to SLEEPING state and wakeup only when the child exits

```

363 wait(uint64 addr)
364 {
365     struct proc *np;
366     int havekids, pid;
367     struct proc *p = myproc();
368
369     // hold p->lock for the whole time to avoid lost
370     // wakeups from a child's exit().
371     acquire(&p->lock);
372     for(;;){
373         // Scan through table looking for exited children.
374         havekids = 0;
375         for(np = proc; np < &proc[NPROC]; np++){
376             // this code uses np->parent without holding np->lock.
377             // acquiring the lock first would cause a deadlock,
378             // since np might be an ancestor, and we already hold p->lock.
379             if(np->parent == p){
380                 // np->parent can't change between the check and the acquire()
381                 // because only the parent changes it, and we're the parent.
382                 acquire(&np->lock);
383                 havekids = 1;
384                 if(np->state == ZOMBIE){
385                     // Found one.
386                     pid = np->pid;
387                     if(addr != 0 && copyout(p->pagetable,
388                                         np->tf, sizeof(np->tf)) < 0)
389                         release(&np->lock);
390                         release(&p->lock);
391                         return -1;
392                     }
393                     freeproc(np);
394                     release(&np->lock);
395                     release(&p->lock);
396                     return pid;
397                 }
398                 release(&np->lock);
399             }
400         }
401         // No point waiting if we don't have any children.
402         if(!havekids || p->killed){
403             release(&p->lock);
404             return -1;
405         }
406         // Wait for a child to exit.
407         sleep(p, &p->lock); //DOC: wait-sleep
408     }
409 }
410

```

wait

For every process in table, find if (a) p is a parent and (b) if the process is a zombie.

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

```

static void
freeproc(struct proc *p)
{
    if(p->tf)
        kfree((void*)p->tf);
    p->tf = 0;
    if(p->pagetable)
        proc_freepagetable(p->pagetable, p->sz);
    p->pagetable = 0;
    p->sz = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->chan = 0;
    p->killed = 0;
    p->xstate = 0;
    p->state = UNUSED;
}

```

status of the child process user space.

emory, kernel stack, located here nt to peek into exited 's process

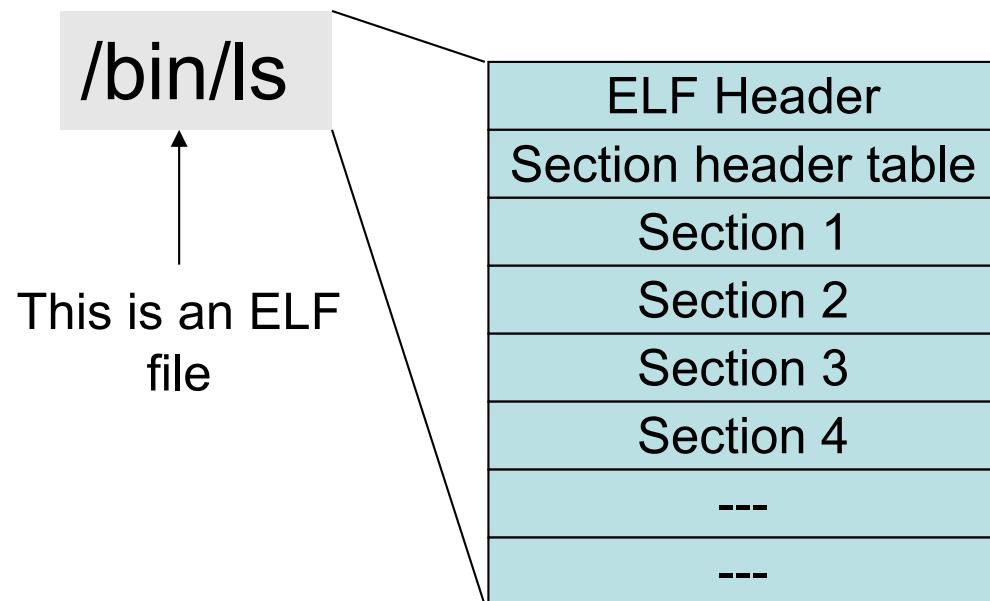
Process will be set to SLEEPING state and wakeup only when the child exits

Executing a Program (exec system call)

- exec system call
 - Load a program into memory and then execute it
 - Here ‘ls’ executed.

```
int pid;  
  
pid = fork();  
if (pid > 0){  
    pid = wait();  
} else{  
    execvp("ls", "", NULL);  
    exit(0);  
}
```

ELF Executables (linker view)

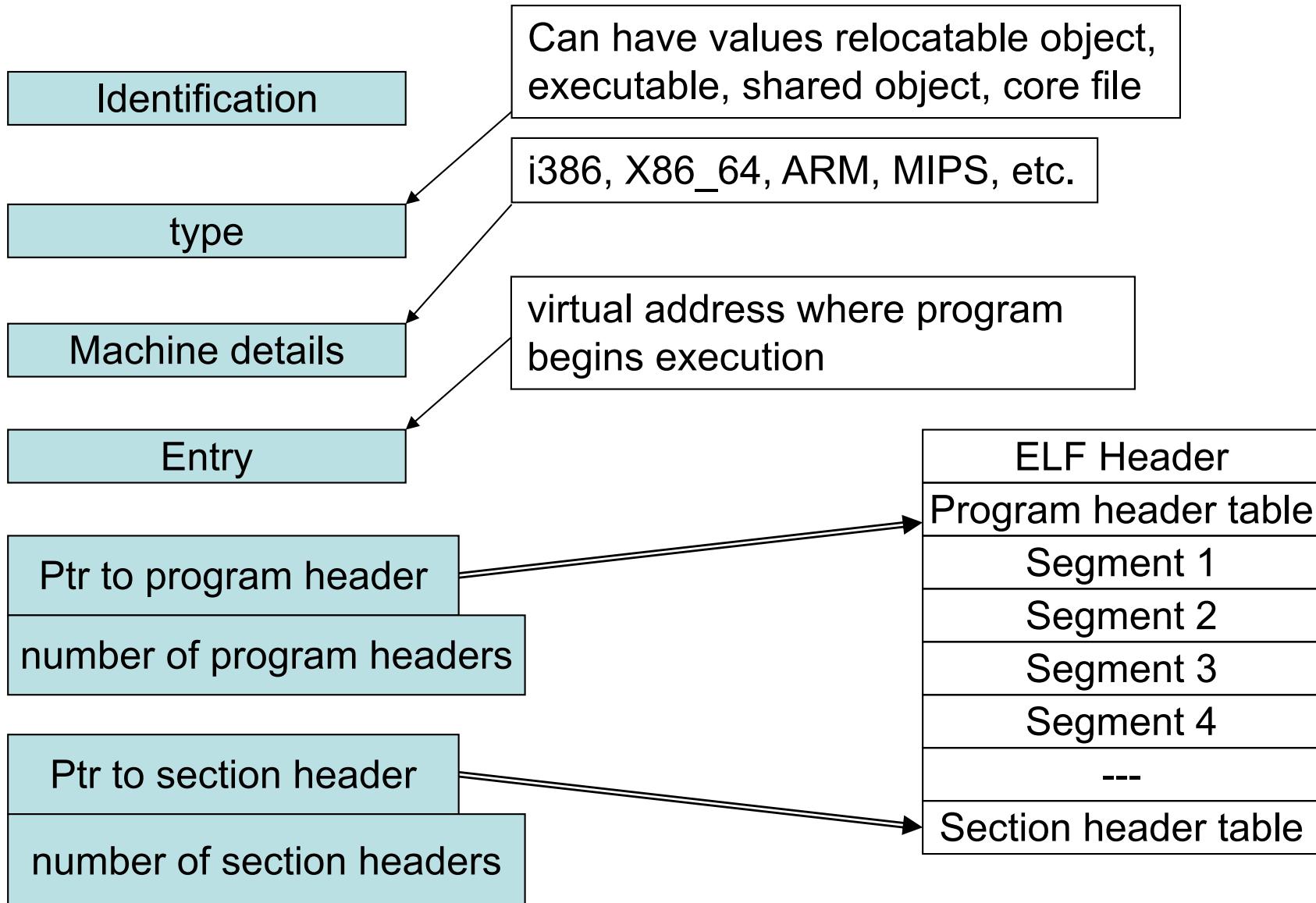


ELF format of executable

ref :www.skyfree.org/linux/references/ELF_Format.pdf

ref :[man elf](#)

ELF Header



Hello World's ELF Header

```
#include <stdio.h>

int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

```
$ gcc hello.c -c
$ readelf -h hello.o
```

```
optiplex:~/tmp$ readelf -h hello.o
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: REL (Relocatable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 368 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 64 (bytes)
  Number of section headers: 13
  Section header string table index: 10
```

Section Headers

- Contains information about the various sections
- \$ readelf –S hello.o

There are 13 section headers, starting at offset 0x170:										
Section Headers:										
[Nr]	Name	Type	Address	Offset	Size	EntSize	Flags	Link	Info	Align
[0]	NULL	PROGBITS	0000000000000000	00000000	0000000000000000	0000000000000000		0	0	0
[1]	.text	PROGBITS	0000000000000000	00000040	0000000000000005c	0000000000000000	AX	0	0	1
[2]	.rela.text	RELA	0000000000000000	000005f8	0000000000000048	0000000000000018		11	1	8
[3]	.data	PROGBITS	0000000000000000	0000009c	0000000000000000	0000000000000000	WA	0	0	1
[4]	.bss	NOBITS	0000000000000000	0000009c	0000000000000000	0000000000000000	WA	0	0	1
[5]	.rodata	PROGBITS	0000000000000000	0000009c	0000000000000003	0000000000000000	A	0	0	1
[6]	.comment	PROGBITS	0000000000000000	0000009f	000000000000002a	0000000000000001	MS	0	0	1
[7]	.note.GNU-stack	PROGBITS	0000000000000000	000000c9	0000000000000000	0000000000000000		0	0	1
[8]	.eh_frame	PROGBITS	0000000000000000	000000d0	0000000000000038	0000000000000000	A	0	0	8
[9]	.rela.eh_frame	RELA	0000000000000000	00000640	0000000000000018	0000000000000000		11	8	8
[10]	.shstrtab	STRTAB	0000000000000000	00000108	0000000000000061	0000000000000000		0	0	1
[11]	.symtab	SYMTAB	0000000000000000	000004b0	0000000000000120	0000000000000018		12	9	8
[12]	.strtab	STRTAB	0000000000000000	000005d0	0000000000000026	0000000000000000		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

Type of the section

PROGBITS : information defined by program

SYMTAB : symbol table

NULL : inactive section

NOBITS : Section that occupies no bits

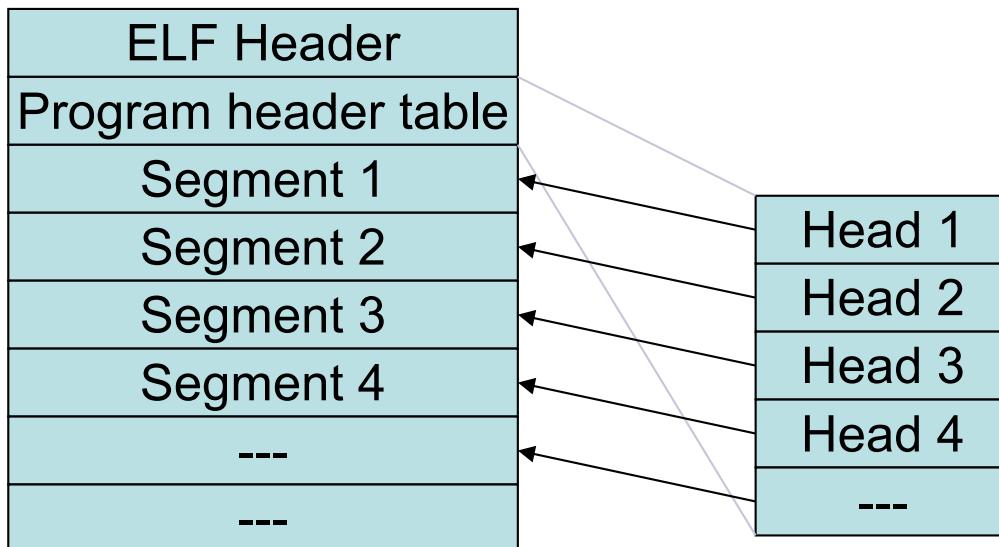
RELA : Relocation table

Offset and size of the section

Size of the table if present else 0

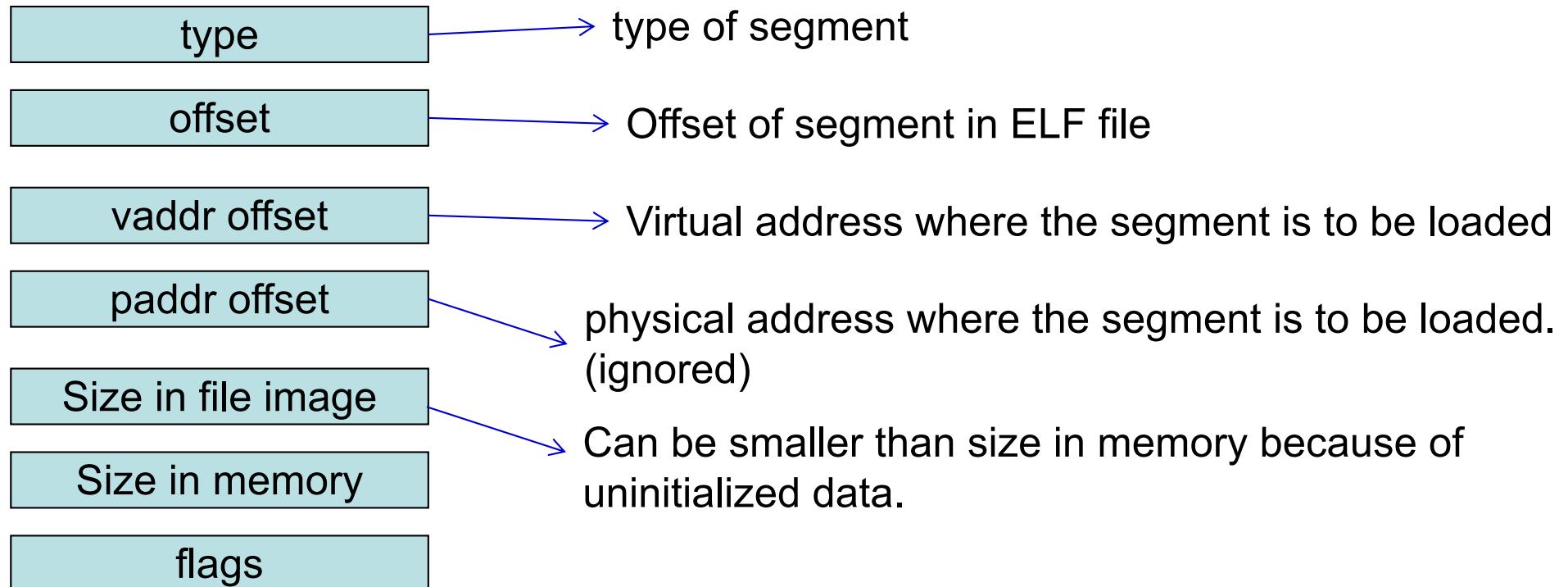
Virtual address where the
Section should be loaded
(* all 0s because this is a .o file)

Program Header (executable view)



- Contains information about each segment
- One program header for each segment
- A program header entry contains (among others)
 - Offset of segment in ELF file
 - Virtual address of segment
 - Segment size in file (filesz)
 - Segment size in memory (memsz)
 - Segment type
 - Loadable segment
 - Shared library
 - etc

Program Header Contents



Program headers for Hello World

- readelf -l hello

```
Elf file type is EXEC (Executable file)
Entry point 0x4004b0
There are 9 program headers, starting at offset 64

Program Headers:
Type          Offset        VirtAddr       PhysAddr  FileSiz      MemSiz      Flags Align
PHDR          0x0000000000000040 0x0000000000400040 0x000000000000400040 0x000000000000001f8 0x000000000000001f8 R E 8
INTERP        0x00000000000000238 0x0000000000400238 0x00000000000000400238 0x000000000000001c 0x000000000000001c R 1
              [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD          0x0000000000000000 0x0000000000400000 0x0000000000400000 0x000000000000007b4 0x000000000000007b4 R E 200000
LOAD          0x00000000000000e10 0x0000000000600e10 0x0000000000600e10 0x00000000000000238 0x00000000000000240 RW 200000
DYNAMIC       0x00000000000000e28 0x0000000000600e28 0x0000000000600e28 0x000000000000001d0 0x000000000000001d0 RW 8
NOTE          0x00000000000000254 0x0000000000400254 0x0000000000400254 0x00000000000000044 0x00000000000000044 R 4
GNU_EH_FRAME  0x00000000000000688 0x0000000000400688 0x0000000000400688 0x0000000000000034 0x0000000000000034 R 4
GNU_STACK     0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000 RW 10
GNU_RELRO    0x000000000000e10 0x0000000000600e10 0x0000000000600e10 0x000000000000001f0 0x000000000000001f0 R 1

Section to Segment mapping:
Segment Sections...
00
01  .interp
02  .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
03  .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04  .dynamic
05  .note.ABI-tag .note.gnu.build-id
06  .eh_frame_hdr
07
08  .init_array .fini_array .jcr .dynamic .got
```

Mapping between segments and sections

exec

exec.c

```
12 int
13 exec(char *path, char **argv)
14 {
15     char *s, *last;
16     int i, off;
17     uint64 argc, sz, sp, ustack[MAXARG+1], stackbase;
18     struct elfhdr elf;
19     struct inode *ip;
20     struct proghdr ph;
21     pagetable_t pagetable = 0, oldpagetable;
22     struct proc *p = myproc();
23
24     begin_op();
25
26     if((ip = namei(path)) == 0){
27         end_op();
28         return -1;
29     }
30     ilock(ip);
31
32     // Check ELF header
33     if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
34         goto bad;
35     if(elf.magic != ELF_MAGIC)
36         goto bad;
37
38     if((pagetable = proc_pagetable(p)) == 0)
39         goto bad;
...
•
```

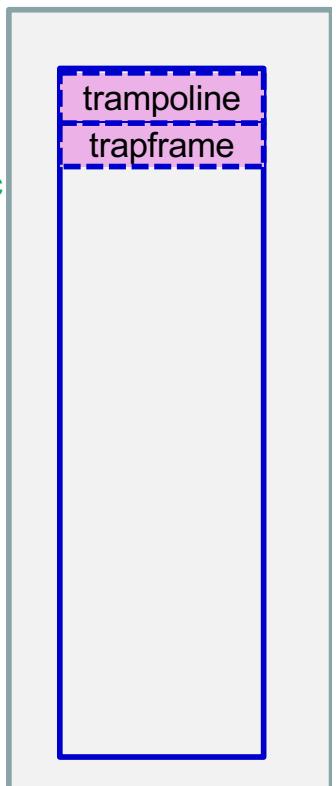
Parameters are the path of executable and command line arguments

Get pointer to the inode for the executable

Process Memory Map

- Executable files begin with a signature.

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



Set up a new set of page tables.

Do we really need to do this?



exec contd. (load segments into memory)

```
...
41 // Load program into memory.
42 sz = 0;
43 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
44     if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
45         goto bad;
46     if(ph.type != ELF_PROG_LOAD)
47         continue;
48     if(ph.memsz < ph.filesz)
49         goto bad;
50     if(ph.vaddr + ph.memsz < ph.vaddr)
51         goto bad;
52     if((sz = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
53         goto bad;
54     if(ph.vaddr % PGSIZE != 0)
55         goto bad;
56     if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
57         goto bad;
58 }
59 unlockout(ip);
...

```

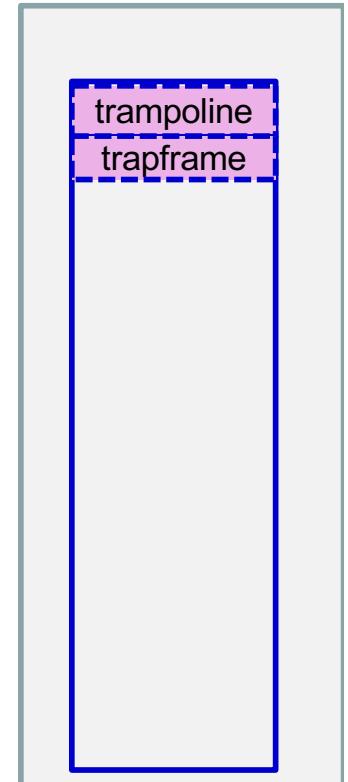
Parse through all the elf program headers.

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)

Load segment into RAM at the specified virtual address (ph.vaddr)

Virtual Memory Map

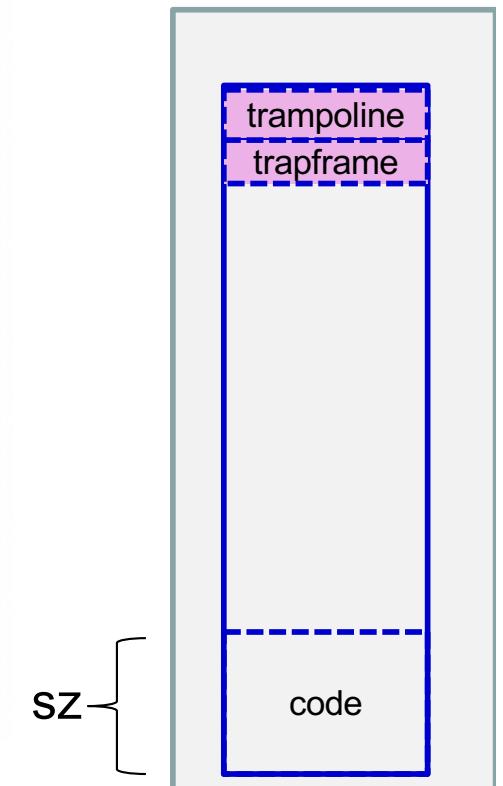


uvmalloc

Grow the process from oldsz
(initially 0) to newsz (vaddr+memsz)

```
235 uint64
236 uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
237 {
238     char *mem;
239     uint64 a;
240
241     if(newsz < oldsz)
242         return oldsz;
243
244     oldsz = PGROUNDUP(oldsz);
245     a = oldsz;
246     for(; a < newsz; a += PGSIZE){
247         mem = kalloc();
248         if(mem == 0){
249             uvmdealloc(pagetable, a, oldsz);
250             return 0;
251         }
252         memset(mem, 0, PGSIZE);
253         if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_W|PTE_X|PTE_R|PTE_U) != 0){
254             kfree(mem);
255             uvmdealloc(pagetable, a, oldsz);
256             return 0;
257         }
258     }
259     return newsz;
260 }
```

Virtual Memory Map



loadseg

Load segment pointed to by ip + offset
to the specified virtual address va

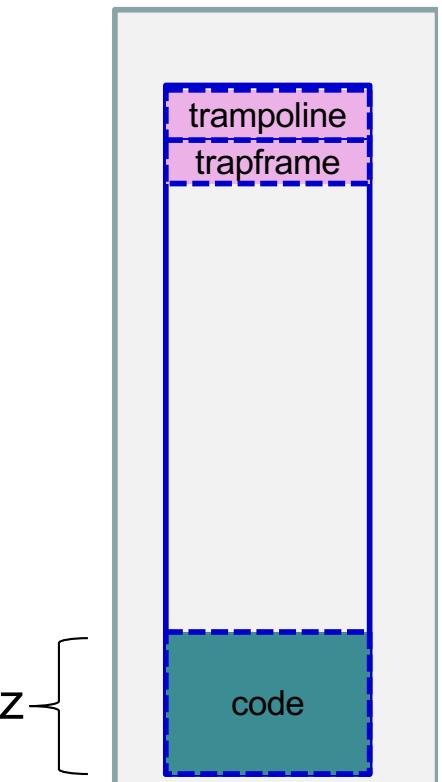
```
static int
loadseg(pagetable_t pagetable, uint64 va, struct inode *ip, uint offset, uint sz)
{
    uint i, n;
    uint64 pa;

    if((va % PGSIZE) != 0)
        panic("loadseg: va must be page aligned");

    for(i = 0; i < sz; i += PGSIZE){
        pa = walkaddr(pagetable, va + i);
        if(pa == 0)
            panic("loadseg: address should exist");
        if(sz - i < PGSIZE)
            n = sz - i;
        else
            n = PGSIZE;
        if(readi(ip, 0, (uint64)pa, offset+i, n) != n)
            return -1;
    }

    return 0;
}
```

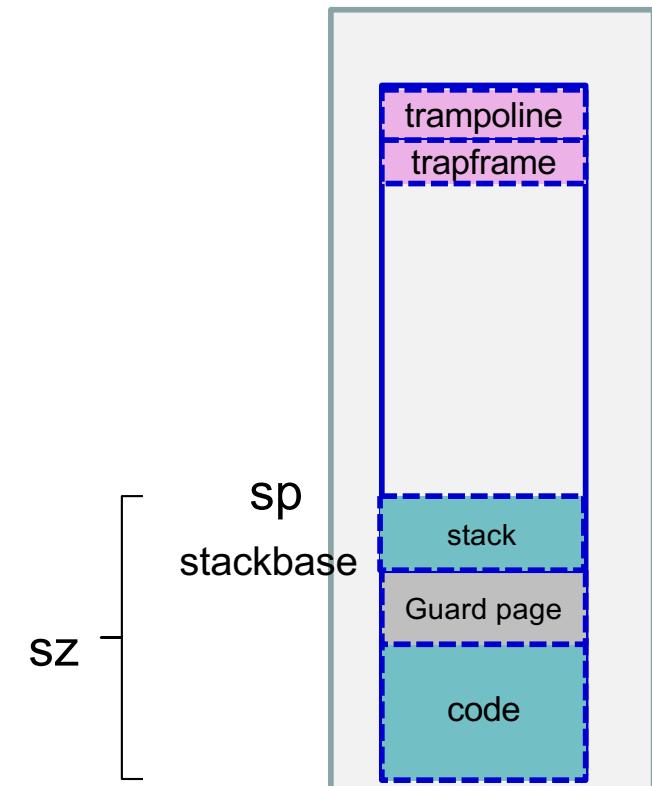
Virtual Memory Map



exec contd. (allocate user stacks)

```
•  
•  
•  
•  
66 // Allocate two pages at the next page boundary.  
67 // Use the second as the user stack.  
68 sz = PGROUNDUP(sz);  
69 if((sz = uvmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)  
70     goto bad;  
71 uvmclear(pagetable, sz-2*PGSIZE);  
72 sp = sz;  
73 stackbase = sp - PGSIZE;
```

Virtual Memory Map



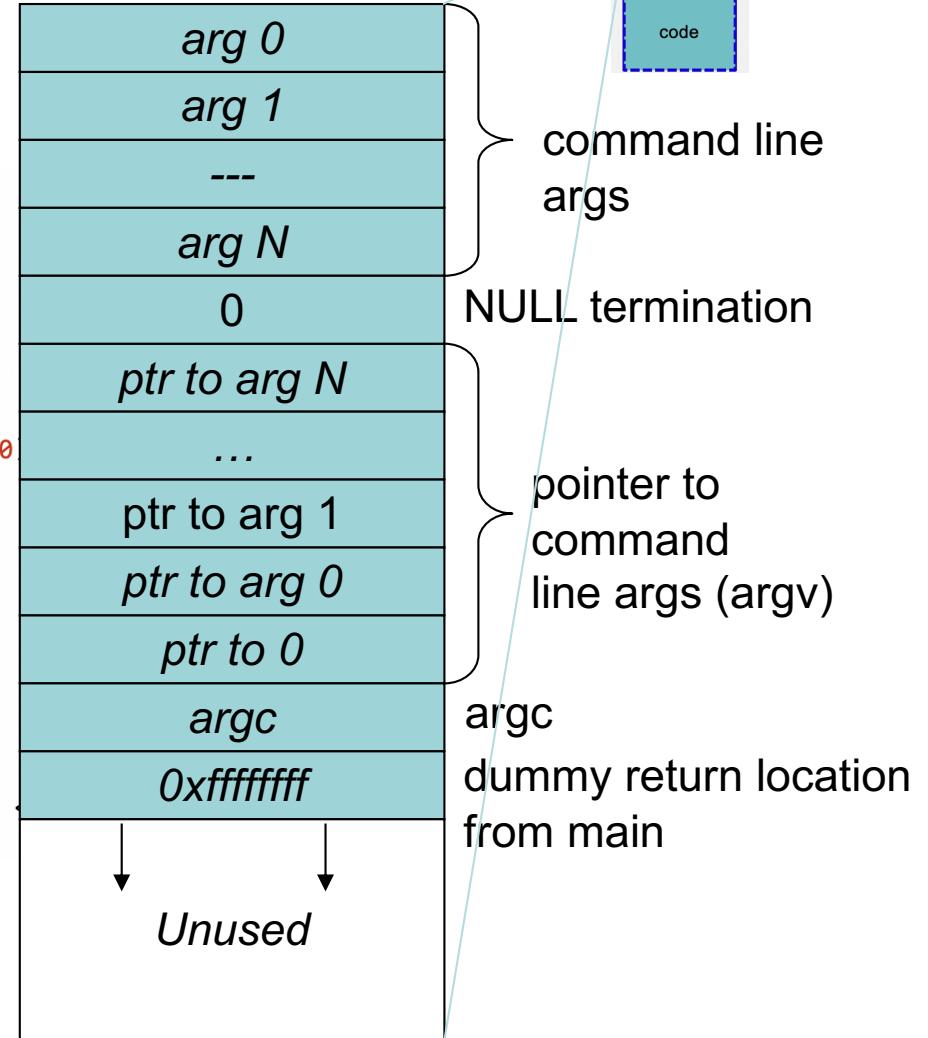
exec contd. (fill user stack with command line args)

```

    ●
    ●
    ●
    ●

75 // Push argument strings, prepare rest of stack in ustack.
76 for(argc = 0; argv[argc]; argc++) {
77     if(argc >= MAXARG)
78         goto bad;
79     sp -= strlen(argv[argc]) + 1;
80     sp -= sp % 16; // riscv sp must be 16-byte aligned
81     if(sp < stackbase)
82         goto bad;
83     if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
84         goto bad;
85     ustack[argc] = sp;
86 }
87 ustack[argc] = 0;
88
89 // push the array of argv[] pointers.
90 sp -= (argc+1) * sizeof(uint64);
91 sp -= sp % 16;
92 if(sp < stackbase)
93     goto bad;
94 if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)))
95     goto bad;
    ●
    ●
    ●
    ●

```



User stack

exec contd. (proc, trapframe, etc.)

```
•  
•  
•  
•  
97 // arguments to user main(argc, argv)  
98 // argc is returned via the system call return  
99 // value, which goes in a0.  
100 p->tf->a1 = sp;  
101  
102 // Save program name for debugging.  
103 for(last=s=path; *s; s++)  
104     if(*s == '/')  
105         last = s+1;  
106 safestrcpy(p->name, last, sizeof(p->name));  
107  
108 // Commit to the user image.  
109 oldpagetable = p->pagetable;  
110 p->pagetable = pagetable;  
111 p->sz = sz;  
112 p->tf->epc = elf.entry; // initial program counter = main  
113 p->tf->sp = sp; // initial stack pointer  
114 proc_freepagetable(oldpagetable, oldsiz);  
115 return argc; // this ends up in a0, the first argument to main(argc, argv)  
116
```

Set the executable file name in proc

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

Free the oldpagetable which was created during fork

Exercise

- How is the heap initialized in xv6?
see sys_sbrk and growproc