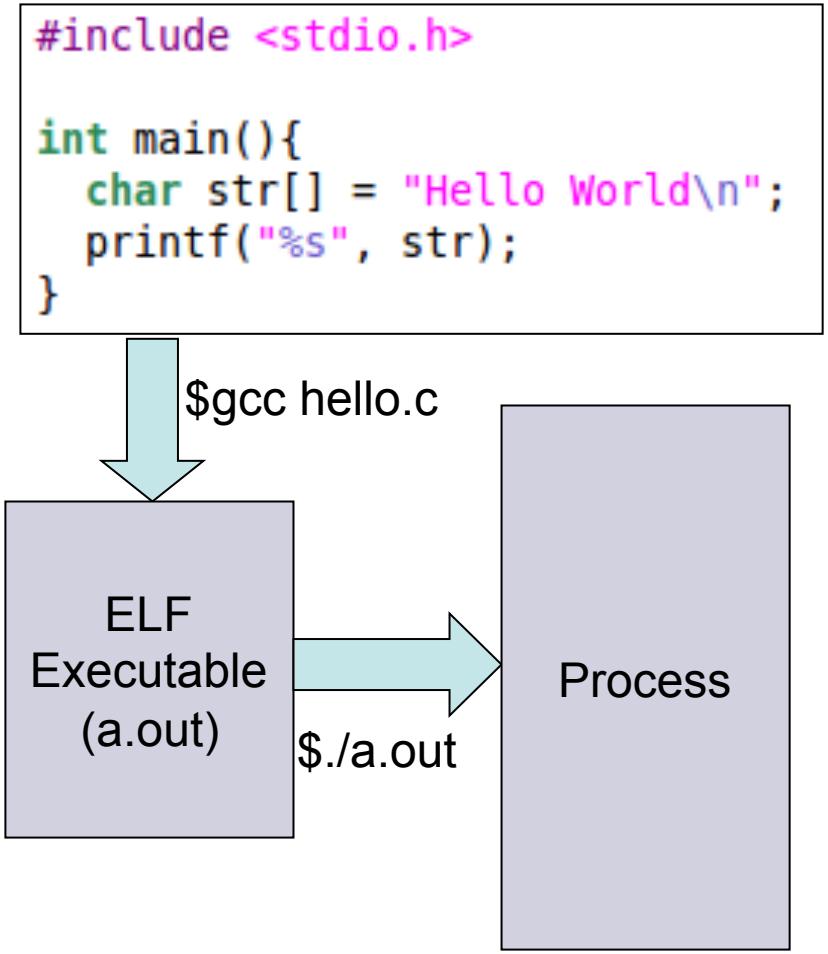
Processes

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

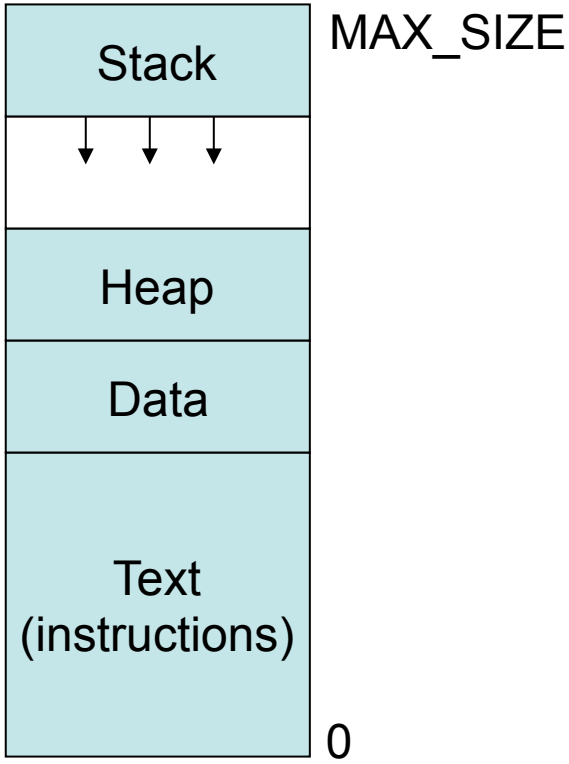
    (Code, Data: from ELF; Code, Data, Stack, Heap: In the user space of process; State in the OS, Kernel stack: In the kernel space)
  - State contains : registers, list of open files, related processes, etc.

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

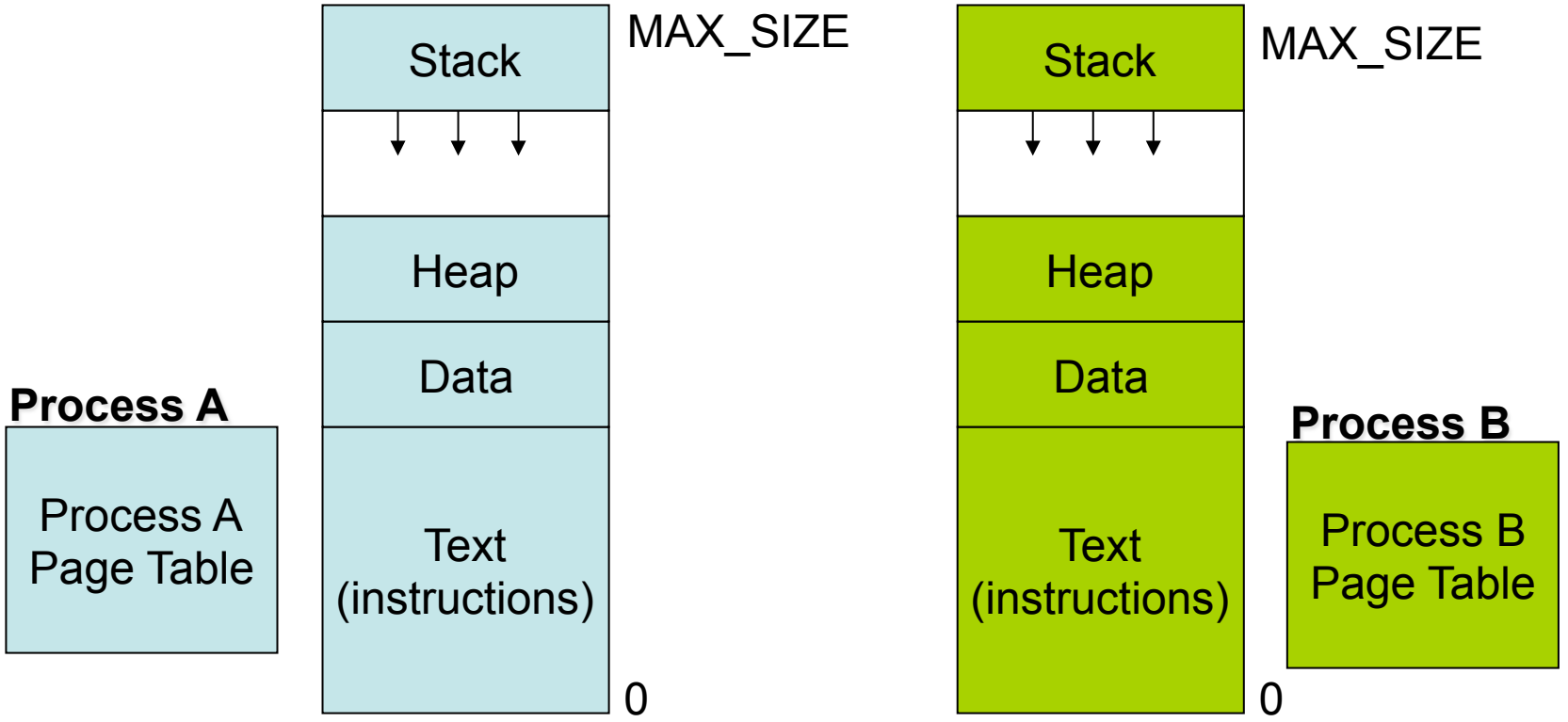
Process Address Space

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

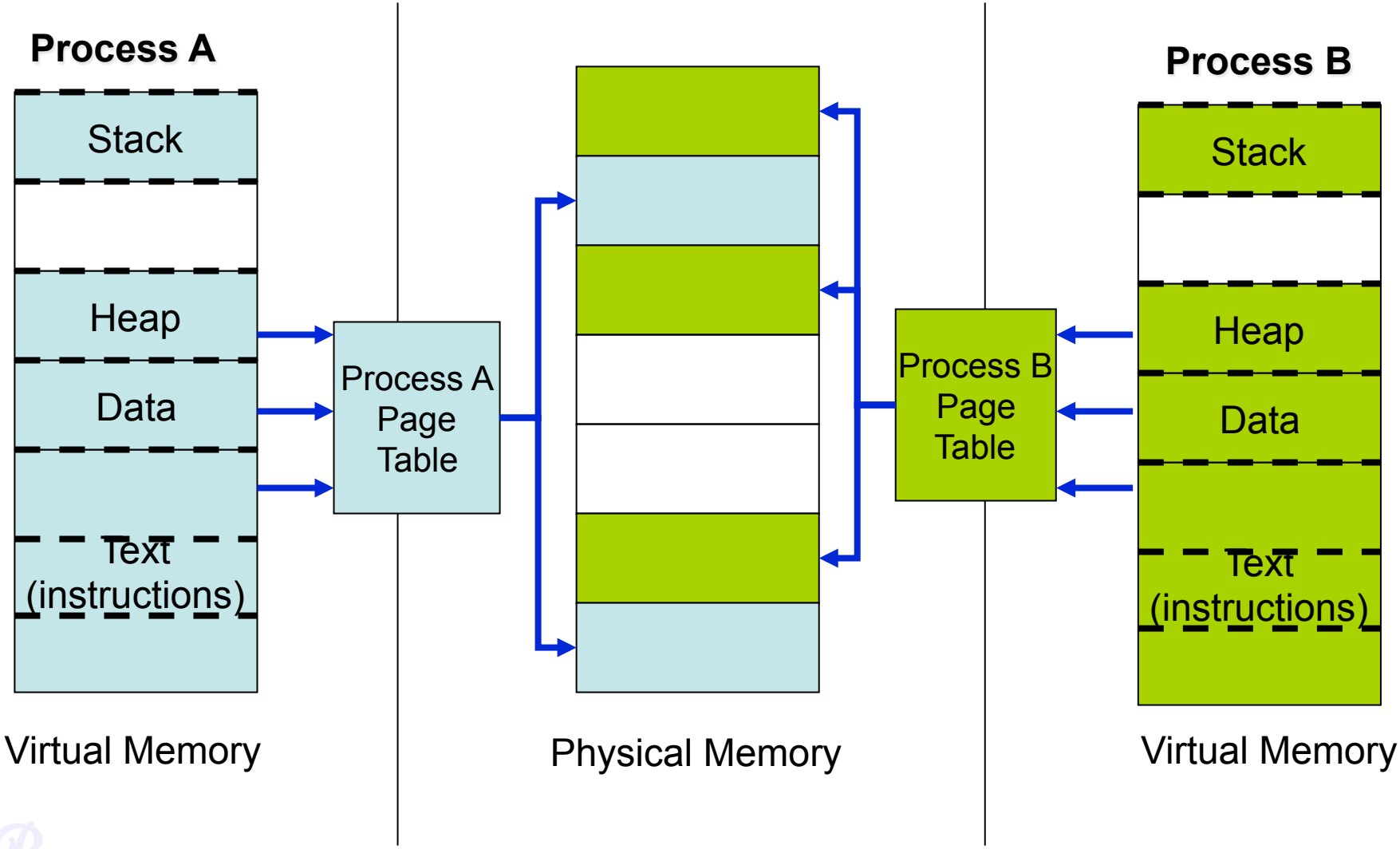


Process Address Space

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



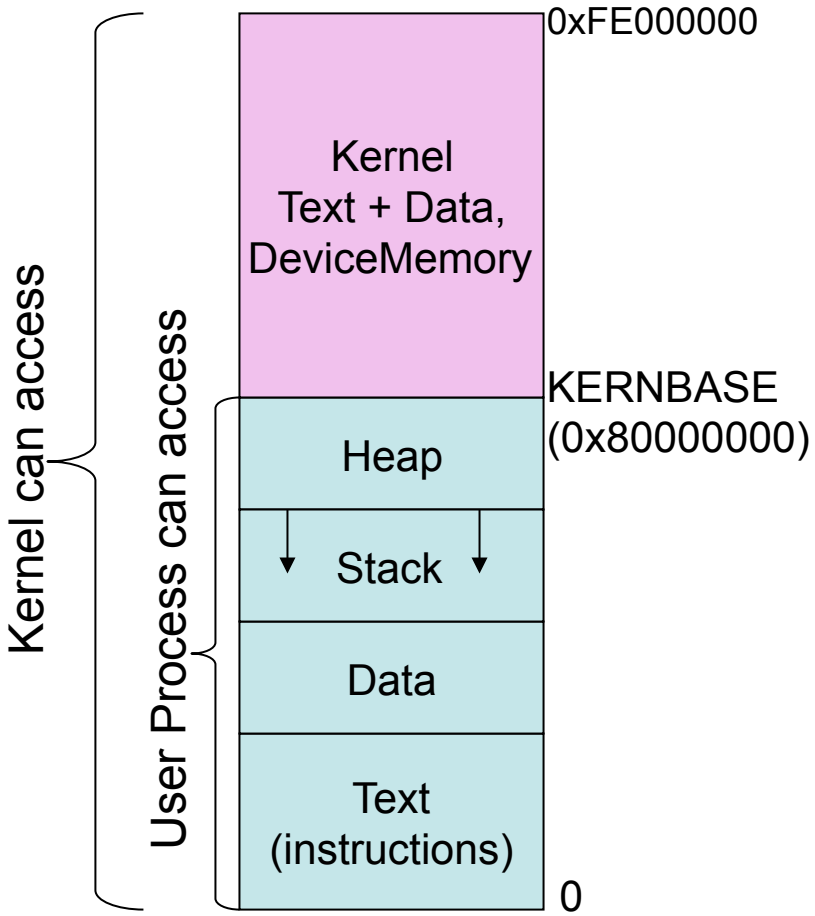
Virtual Address Mapping



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 Address Map in xv6

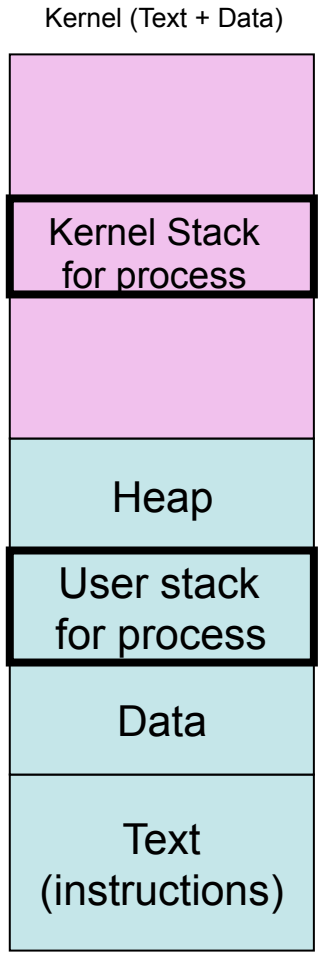


- Entire kernel mapped into every process address space
  - This allows easy switching from user code to kernel code (ie. during system calls)
    - No change of page tables needed
  - Easy access of user data from kernel space

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)



Process Address Space

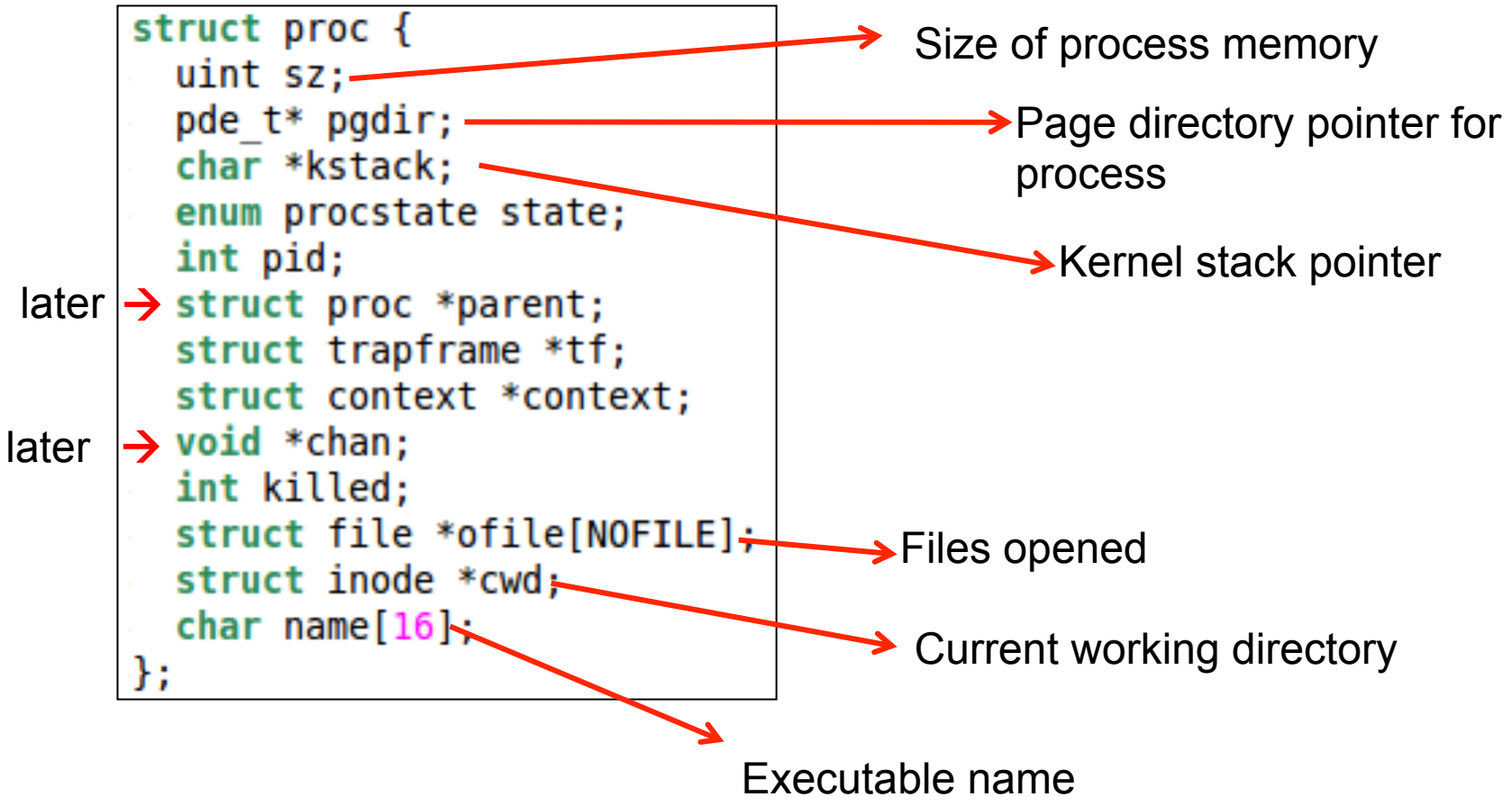
Process Management in xv6

- 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

ref : proc.h (struct proc) (2353)

Summary of entries in PCB

- More entries

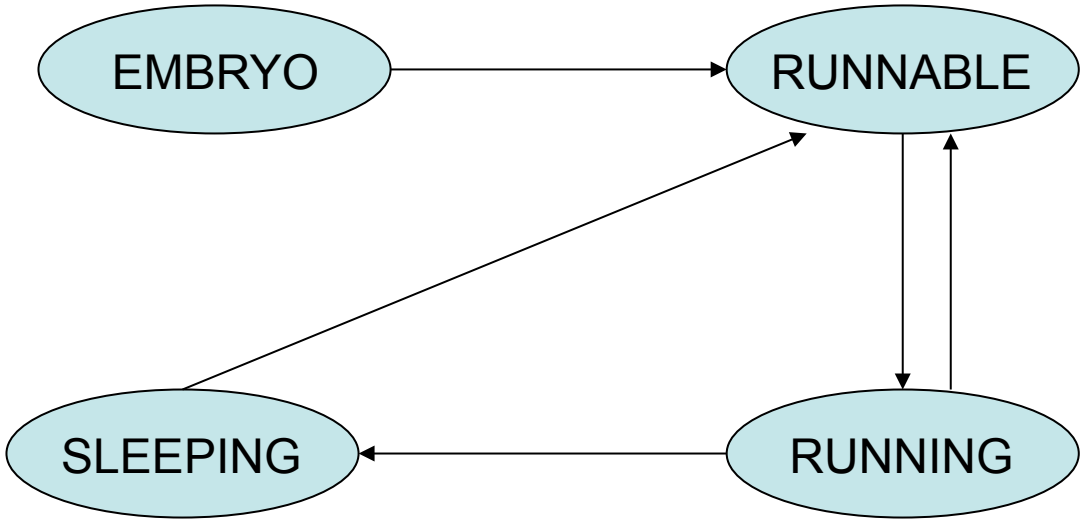


Entries in PCB

- PID
  - Process Identifier
  - Number incremented sequentially
    - When maximum is reached
    - Reset and continue to increment.
    - This time skip already allocated PID numbers

Process States

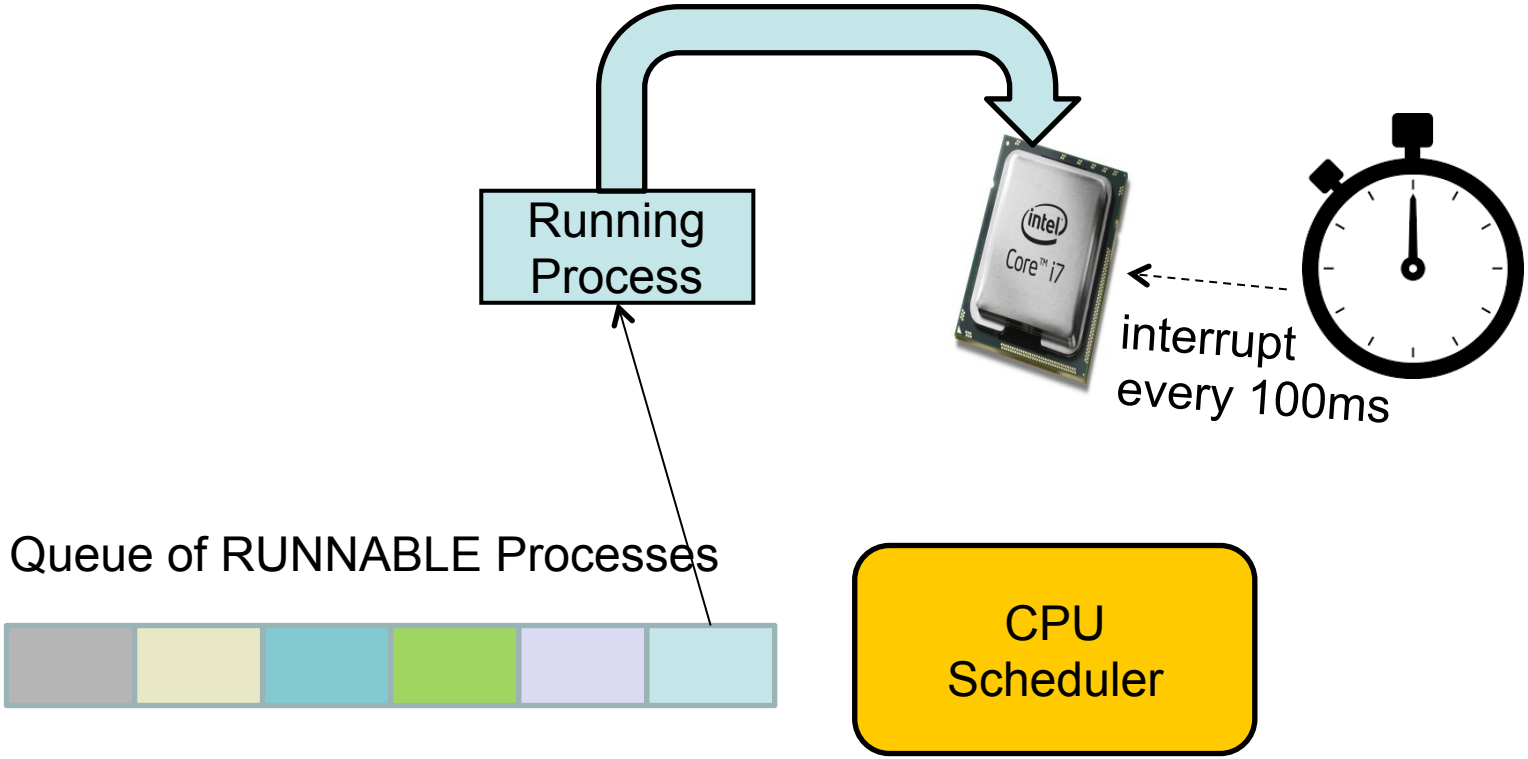
- Process State : specifies the state of the process



EMBRYO → The new process is currently being created
RUNNABLE → Ready to run
RUNNING → Currently executing
SLEEPING → Blocked for an I/O
Other states ZOMBIE (later)

ref : proc.h (struct proc) 2350

Scheduling Runnable Processes

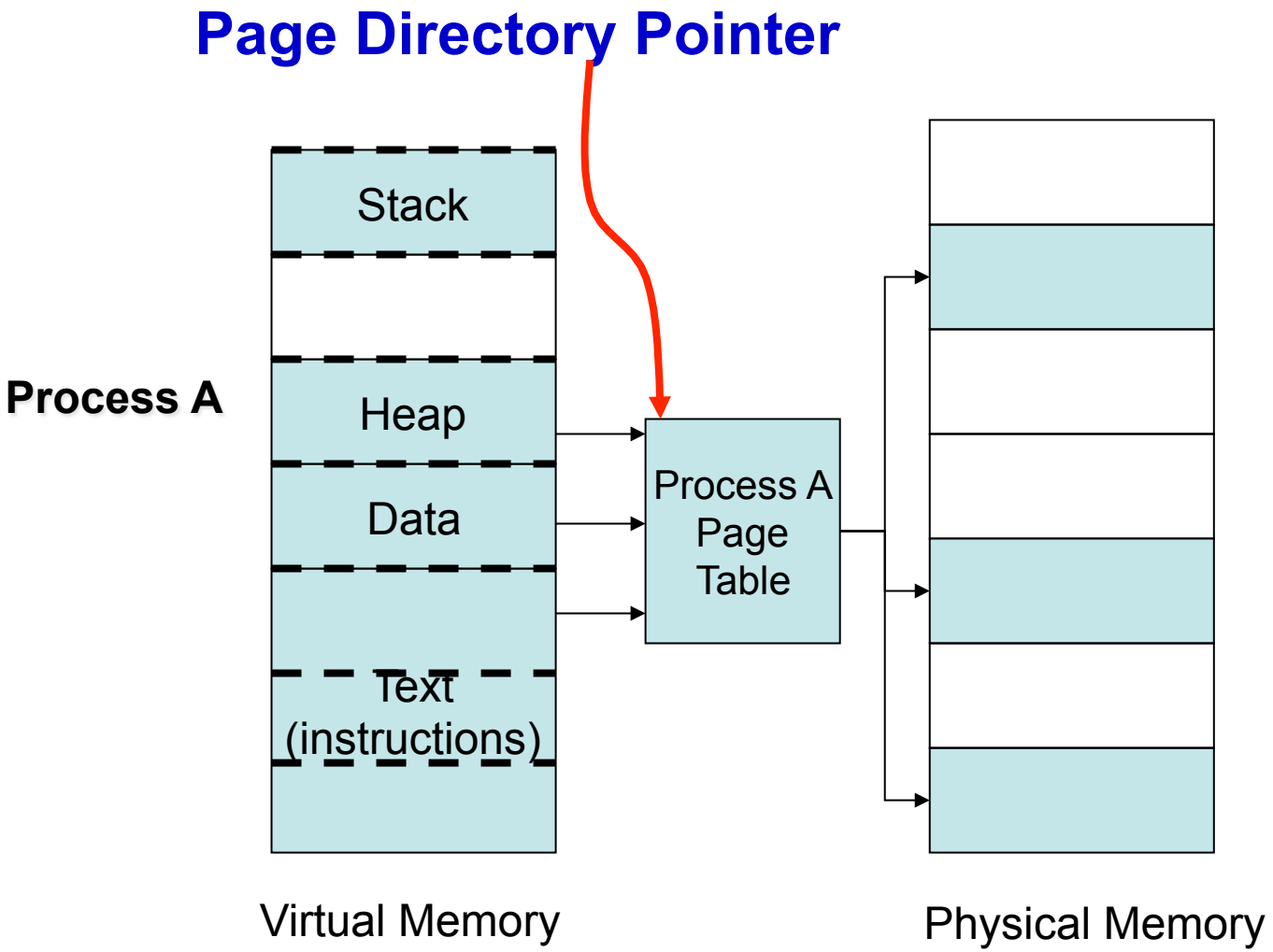


Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O

Scheduler picks another process from the ready queue

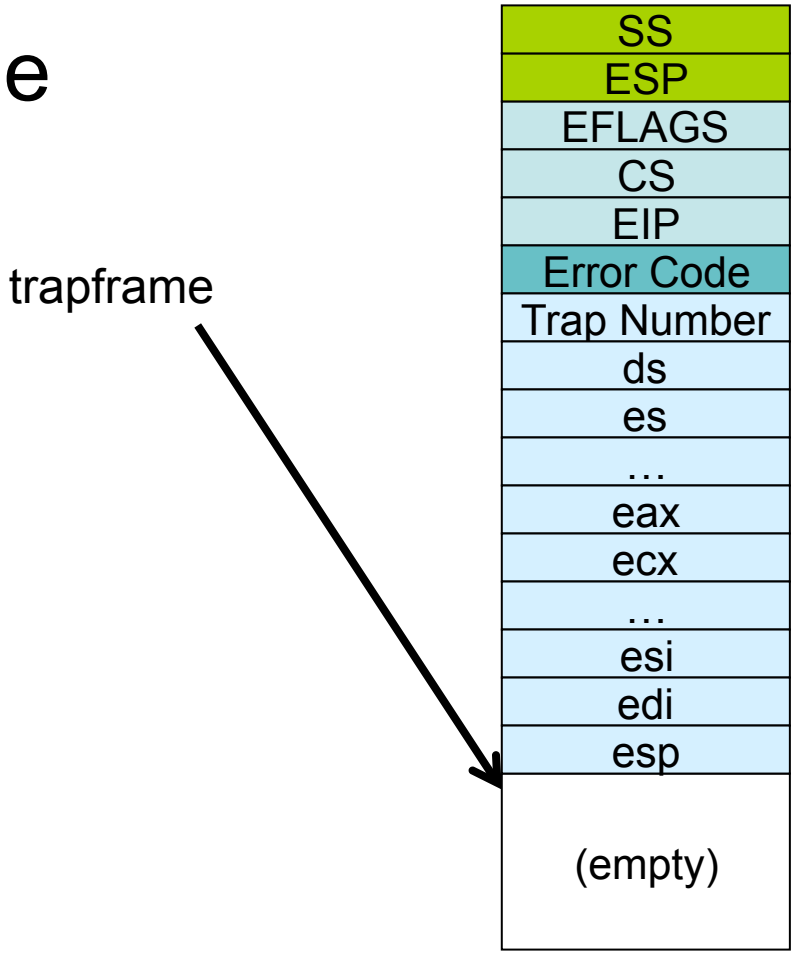
Performs a context switch

Page Directory Pointer



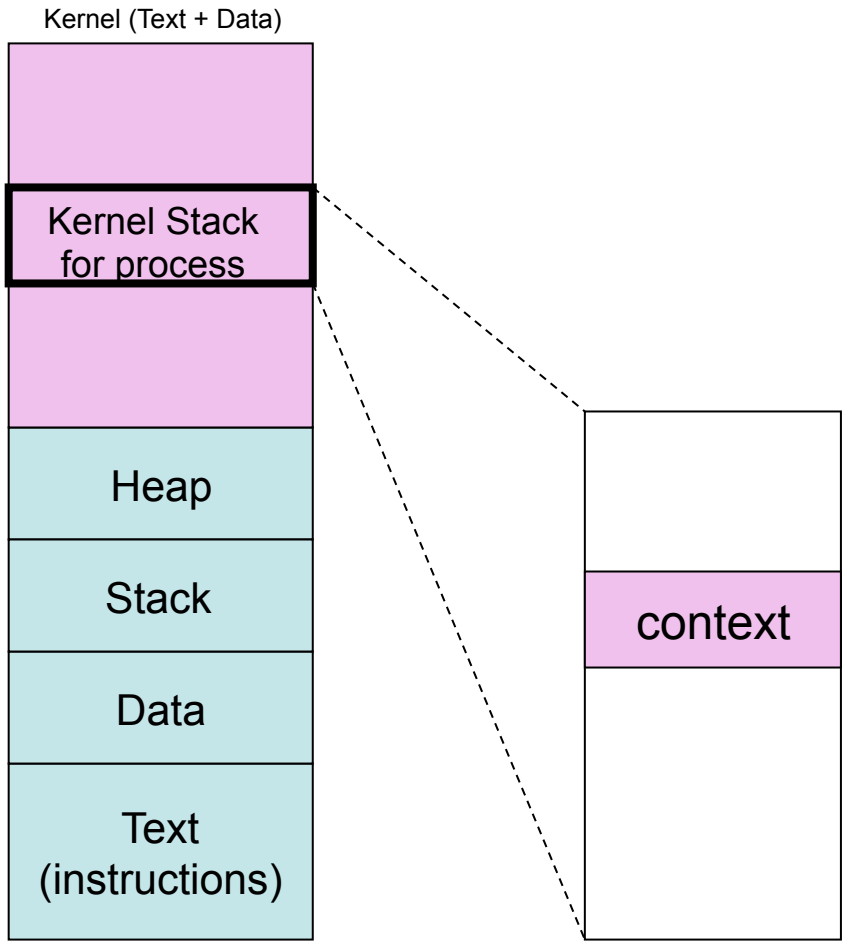
Entries in PCB

- Pointer to trapframe



Context Pointer

- Context pointer
  - Contains registers used for context switches.
  - Registers in context : %edi, %esi, %ebx, %ebp, %eip
  - Stored in the kernel stack space



Storing procs in xv6

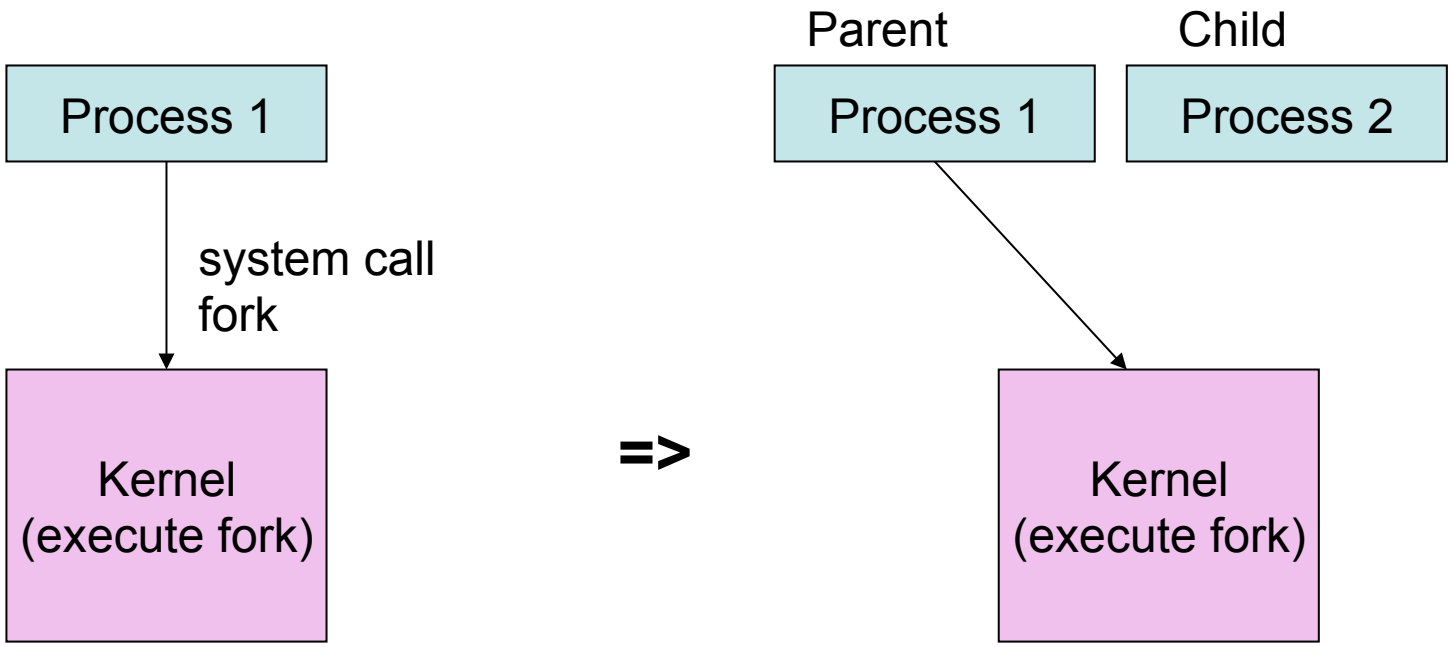
- In a globally defined array present in ptable
- NPROC is the maximum number of processes that can be present in the system (#define NPROC 64)
- Also present in ptable is a lock that seralizes access to the array.

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

ref : proc.c (struct ptable) 2409, params.h (NPROC) 0150

Creating a Process by Cloning

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



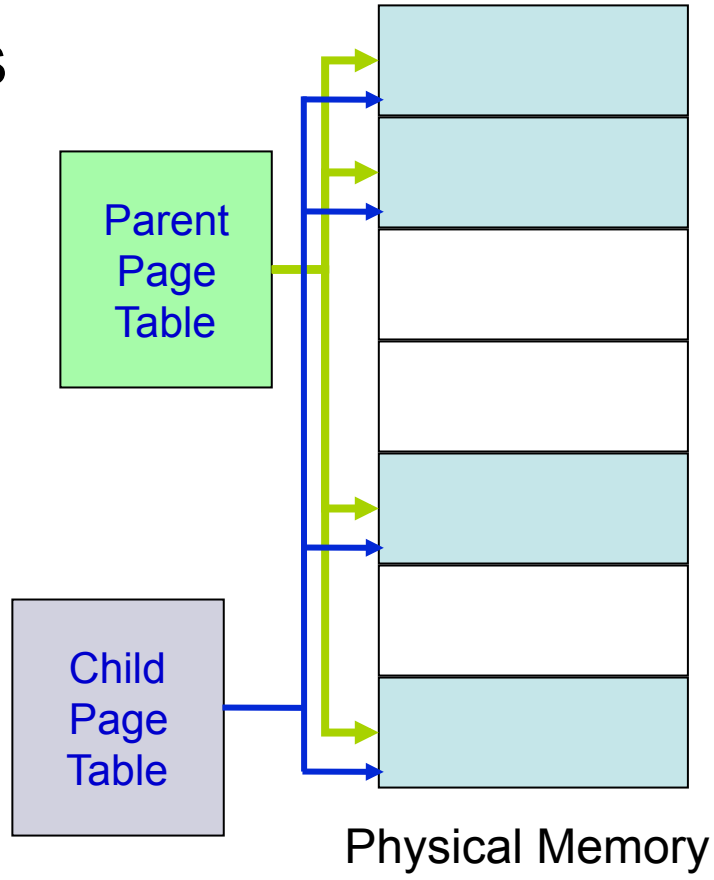
Creating a Process by Cloning (using fork system call)

- 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

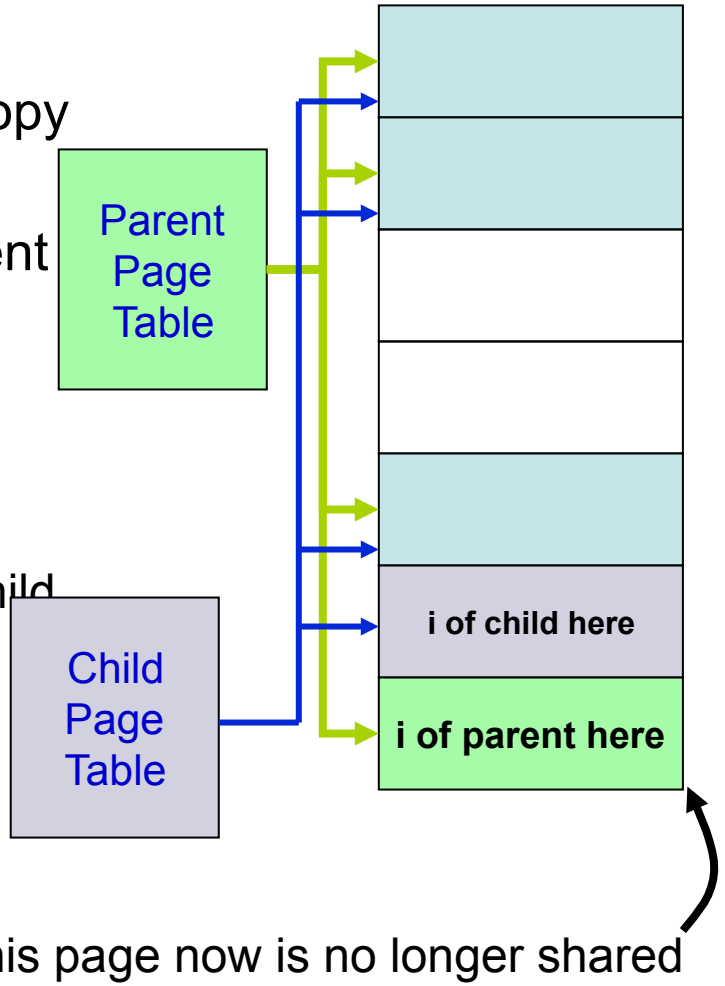
Output

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

```
int i=0, pid;
pid = fork();
if (pid > 0){
        sleep(1);
        printf("parent : %d\n", i);
        wait();
} else{
        i = i + 1;
        printf("child : %d\n", i);
}
```

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



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

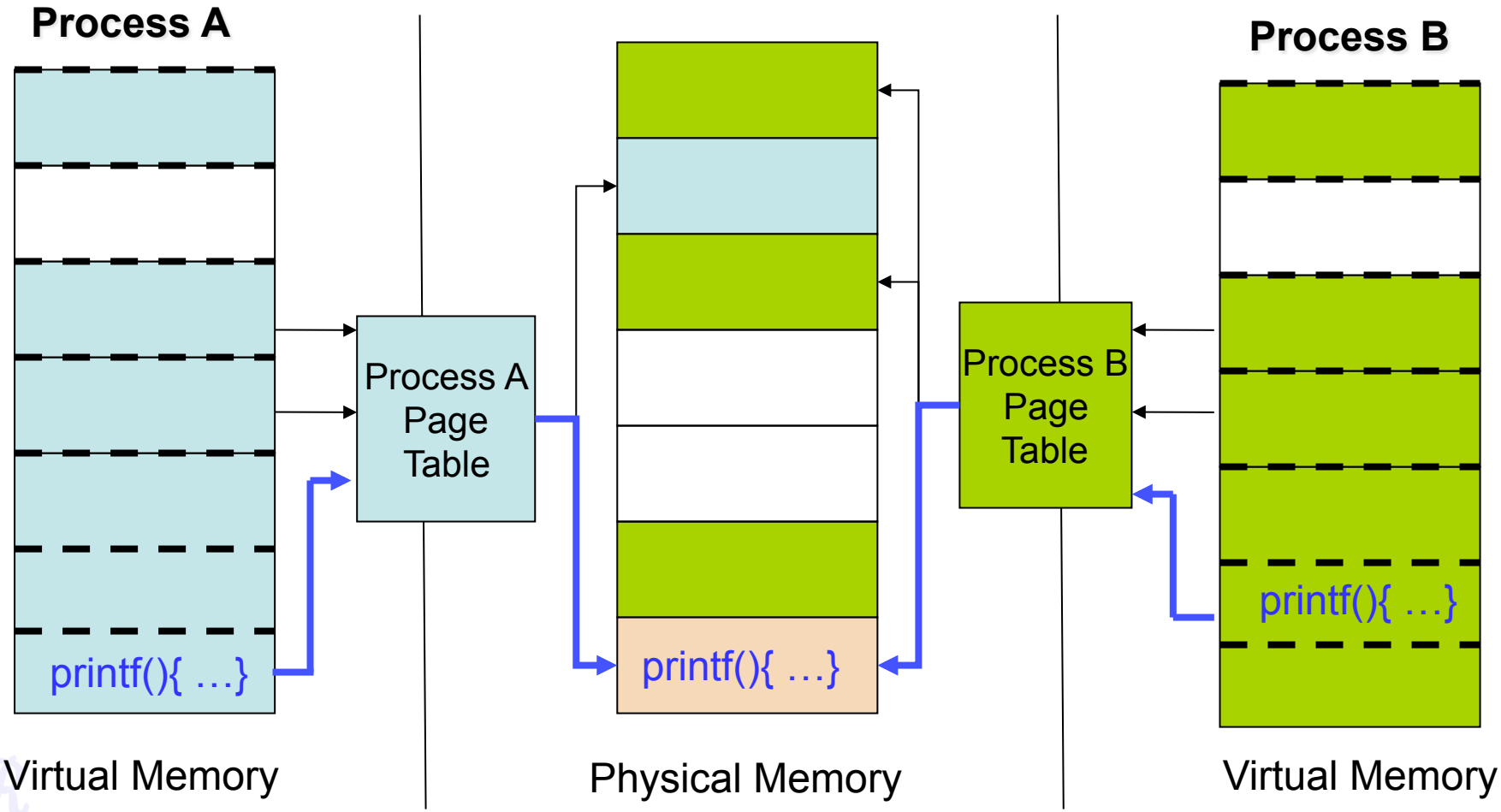
Executing a Program (exec system call)

- 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);
```

Virtual Addressing Advantages (Shared libraries)

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

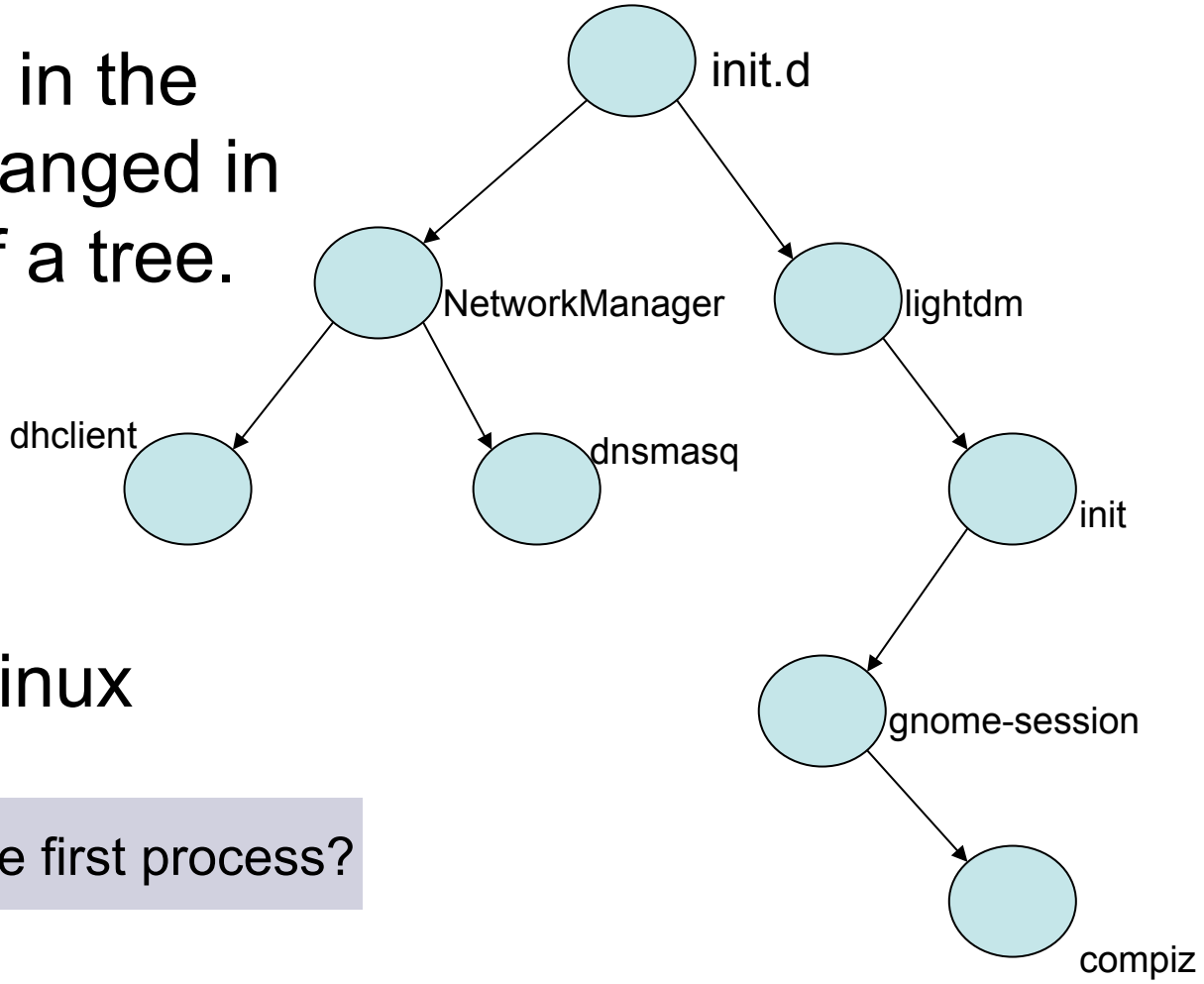


The first process

- Unix : **/sbin/init** (xv6 initcode.S)
  - 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

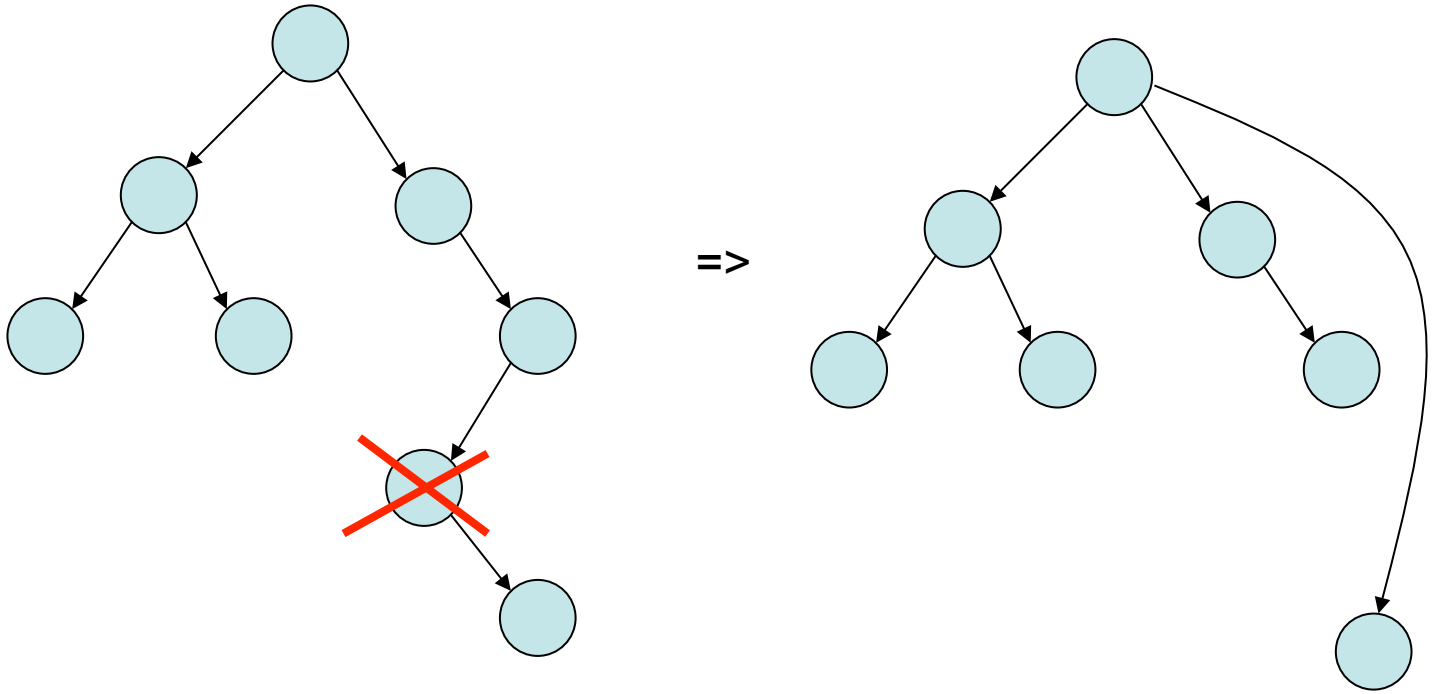
- Involuntary : kill(pid, signal)
  - Signal can be sent by another process or by OS
  - pid is for the process to be killed
  - signal a signal that the process needs to be killed
    - Examples : SIGTERM, SIGQUIT (ctrl+\), SIGINT (ctrl+c), SIGHUP

Zombies

- 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

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.S
- Creating the first process
  - main (1239) invokes userinit (2503)
  - userinit
    - allocate a process id, kernel stack, fill in the proc entries
    - Setup kernel page tables
    - copy initcode.S to 0x0
    - create a user stack
    - set process to runnable
      - the scheduler would then execute the process

allocproc (2455)

```
2454 static struct proc*
2455 allocproc(void)
2456 {
2457   struct proc *p;
2458   char *sp;
2459
2460   acquire(&ptable.lock);
2461   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2462     if(p->state == UNUSED)
2463       goto found;
2464   release(&ptable.lock);
2465   return 0;
2466
2467 found:
2468   p->state = EMBRYO;
2469   p->pid = nextpid++;
2470   release(&ptable.lock);
2471
2472   // Allocate kernel stack.
2473   if((p->kstack = kalloc()) == 0){
2474     p->state = UNUSED;
2475     return 0;
2476   }
2477   sp = p->kstack + KSTACKSIZE;
2478
2479   // Leave room for trap frame.
2480   sp -= sizeof *p->tf;
2481   p->tf = (struct trapframe*)sp;
2482
2483   // Set up new context to start executing at forkret,
2484   // which returns to trapret.
2485   sp -= 4;
2486   *(uint*)sp = (uint)trapret;
2487
2488   sp -= sizeof *p->context;
2489   p->context = (struct context*)sp;
2490   memset(p->context, 0, sizeof *p->context);
2491   p->context->eip = (uint)forkret;
2492
2493   return p;
2494 }
```

find an unused proc entry in the PCB table

set the state to EMBRYO (neither RUNNING nor UNUSED)

set the pid (in real systems.. Need to ensure that the pid is unused)

1

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

2

3

4

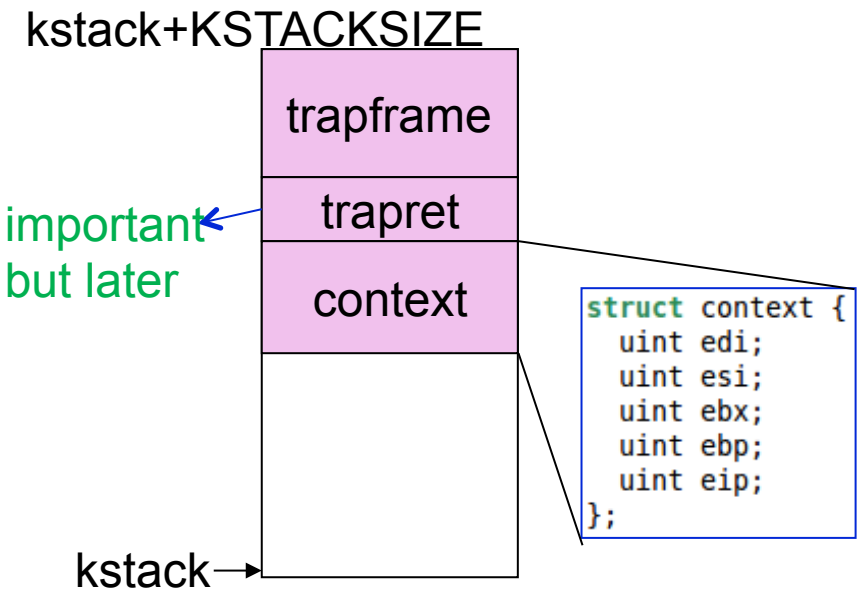
```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

allocproc (2455)

```
2454 static struct proc*
2455 allocproc(void)
2456 {
2457   struct proc *p;
2458   char *sp;
2459
2460   acquire(&ptable.lock);
2461   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2462     if(p->state == UNUSED)
2463       goto found;
2464   release(&ptable.lock);
2465   return 0;
2466
2467 found:
2468   p->state = EMBRYO;
2469   p->pid = nextpid++;
2470   release(&ptable.lock);
2471
2472   // Allocate kernel stack.
2473   if((p->kstack = kalloc()) == 0){
2474     p->state = UNUSED;
2475     return 0;
2476   }
2477   sp = p->kstack + KSTACKSIZE;
2478
2479   // Leave room for trap frame.
2480   sp -= sizeof *p->tf;
2481   p->tf = (struct trapframe*)sp;
2482
2483   // Set up new context to start executing at forkret,
2484   // which returns to trapret.
2485   sp -= 4;
2486   *(uint*)sp = (uint)trapret;
2487
2488   sp -= sizeof *p->context;
2489   p->context = (struct context*)sp;
2490   memset(p->context, 0, sizeof *p->context);
2491   p->context->eip = (uint)forkret;
2492
2493   return p;
2494 }
```

allocate kernel stack of size 4KB.
We next need to allocate space on to kernel stack for

1. the trapframe
2. trapret
3. context



Process's stack in kernel space

forkret: this is important, but we'll look at it later

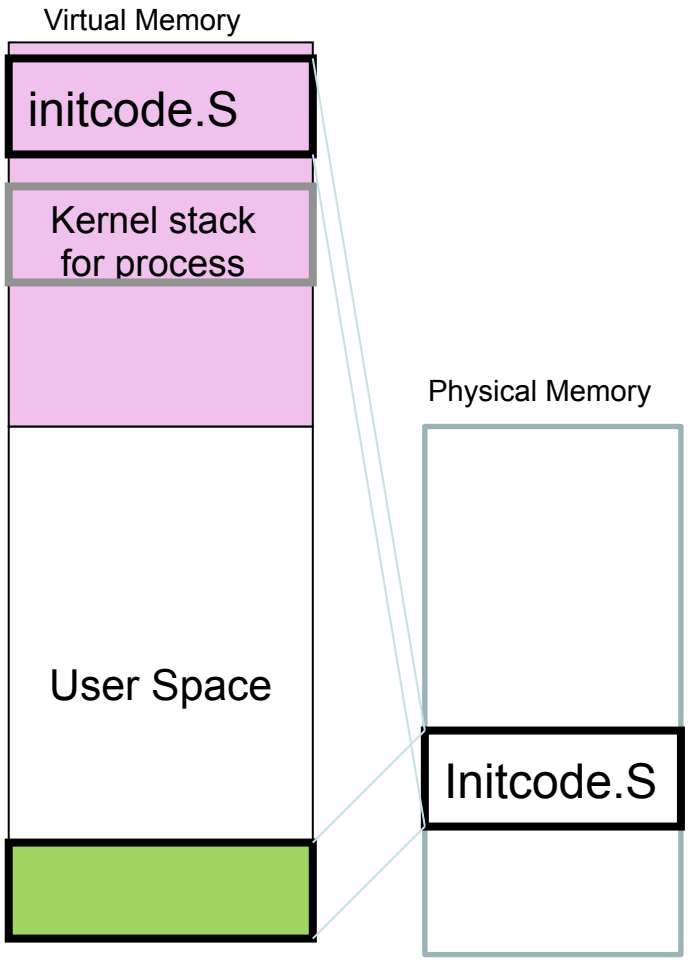
Setup pagetables

- Kernel page tables
  - Invoked by setupkvm(1837)
- User page tables
  - Setup in inituvm (1903)

```
2511   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
```

```
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```

Create PTEs in page directory
VA = 0 →PA (v2p(mem))
Size 1 page (4KB)



…do the rest

```
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[], _binary_initcode_size[];
2506
2507   p = allocproc();
2508   initproc = p;
2509   if((p->pgdir = setupkvm()) == 0)
2510     panic("userinit: out of memory?");
2511   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2512   p->sz = PGSIZE;
2513   memset(p->tf, 0, sizeof(*p->tf));
2514   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2515   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2516   p->tf->es = p->tf->ds;
2517   p->tf->ss = p->tf->ds;
2518   p->tf->eflags = FL_IF;
2519   p->tf->esp = PGSIZE;
2520   p->tf->eip = 0;  // beginning of initcode.S
2521
2522   safestrcpy(p->name, "initcode", sizeof(p->name));
2523   p->cwd = namei("/");
2524
2525   p->state = RUNNABLE;
2526 }
```

Set size to 4KB

Fill trapframe

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

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 →mpmain (1241) →scheduler (1257)

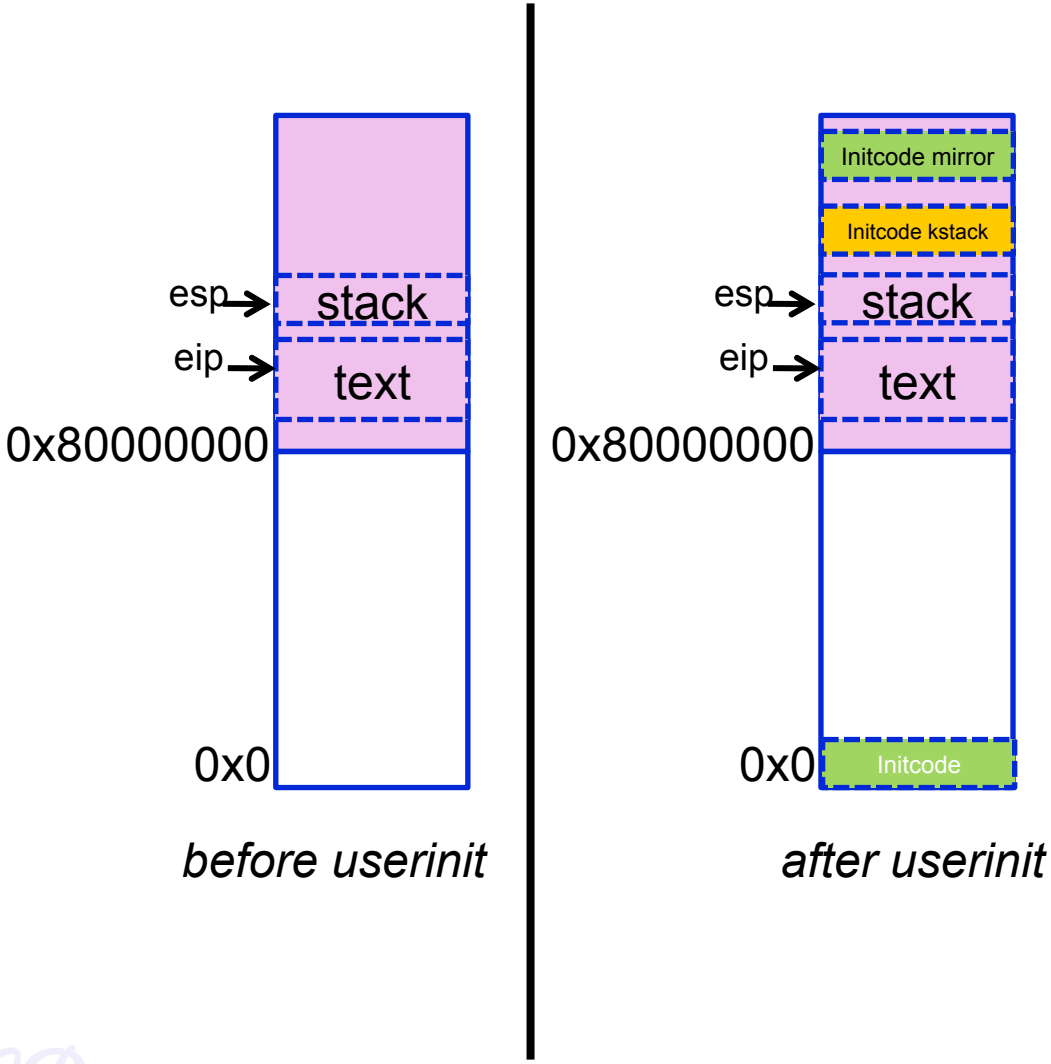
  – The initcode process is selected

    (as it is the only process runnable)

  – …and is then executed

Scheduling the first process

Recall : the virtual memory map

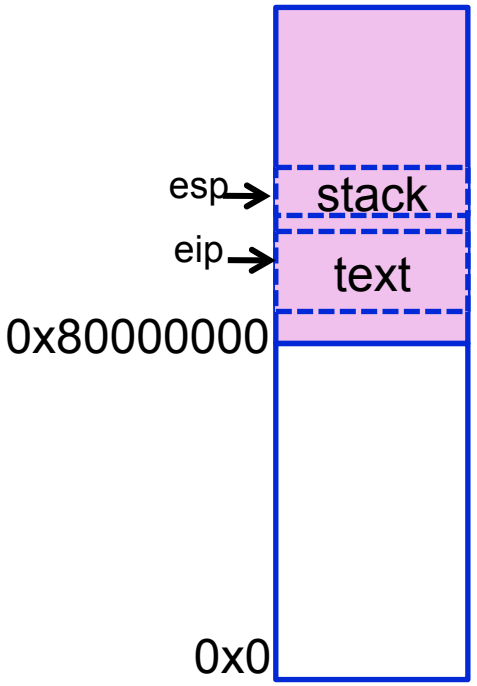


The code and stack for Initcode has been setup.

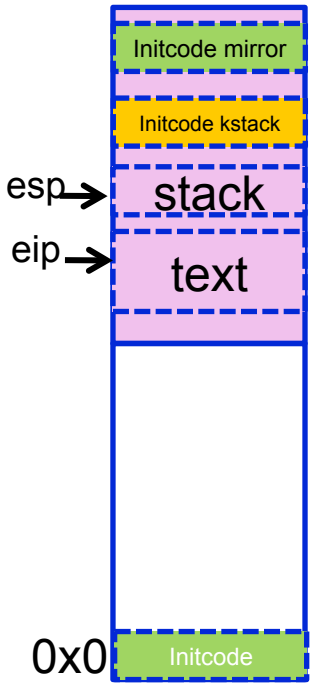
But we are still executing kernel code with the kernel stack.

scheduler() changes this to get Initcode to execute

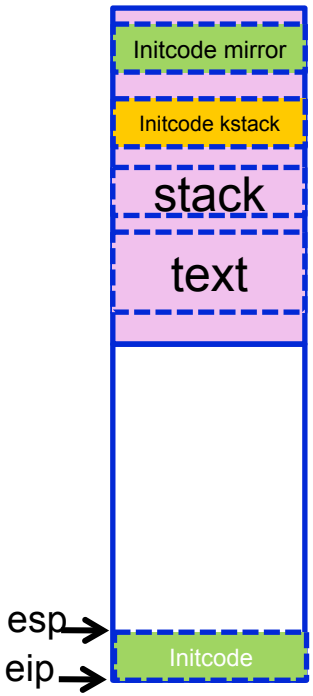
What we need!



before userinit



after userinit



Need to get here (stack starts at 4KB and grows downwards)

Scheduler ()

- main →mpmain (1241) →scheduler (1257)

```
2708 scheduler(void)
2709 {
2710   struct proc *p;
2711
2712   for(;;){
2713     // Enable interrupts on this processor.
2714     sti();
2715
2716     // Loop over process table looking for process to run.
2717     acquire(&ptable.lock);
2718     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2719       if(p->state != RUNNABLE)
2720         continue;
2721
2722       // Switch to chosen process.  It is the process's job
2723       // to release ptable.lock and then reacquire it
2724       // before jumping back to us.
2725       proc = p;
2726       switchuvm(p);
2727       p->state = RUNNING;
2728       swtch(&cpu->scheduler, proc->context);
2729       switchkvm();
2730
2731       // Process is done running for now.
2732       // It should have changed its p->state before coming back.
2733       proc = 0;
2734     }
2735     release(&ptable.lock);
2736
2737   }
2738 }
```

Find the process which is RUNNABLE.
In this case initcode is selected

extern struct proc *proc asm("%gs:4"); //
cpus[cpunum()].proc

switchuvm

```
1873 switchuvm(struct proc *p)
1874 {
1875   pushcli();
1876   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1877   cpu->gdt[SEG_TSS].s = 0;
1878   cpu->ts.ss0 = SEG_KDATA << 3;
1879   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1880   ltr(SEG_TSS << 3);
1881   if(p->pgdir == 0)
1882     panic("switchuvm: no pgdir");
1883   lcr3(v2p(p->pgdir));  // switch to new address space
1884   popcli();
1885 }
```

New TSS segment in GDT

Set the new stack (this is the kernel stack corresponding to initcode.S)

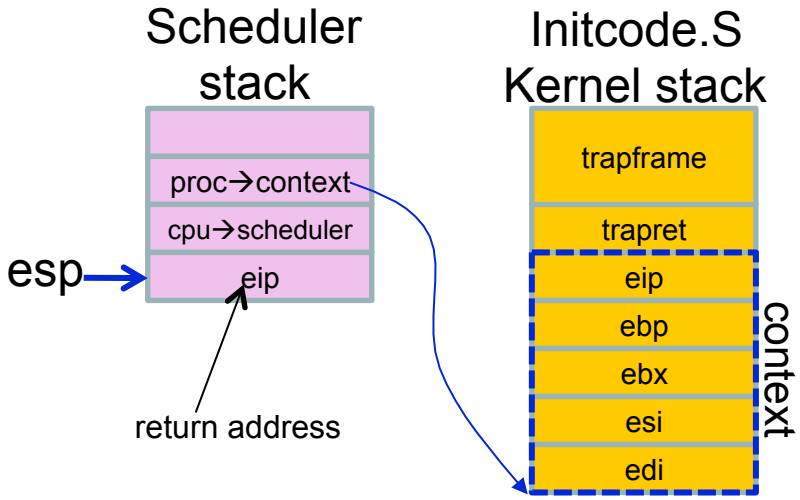
Set the new page tables (corresponding to initcode.S)

Load TSS offset

swtch(cpu→scheduler, proc→context) (1)

```
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```

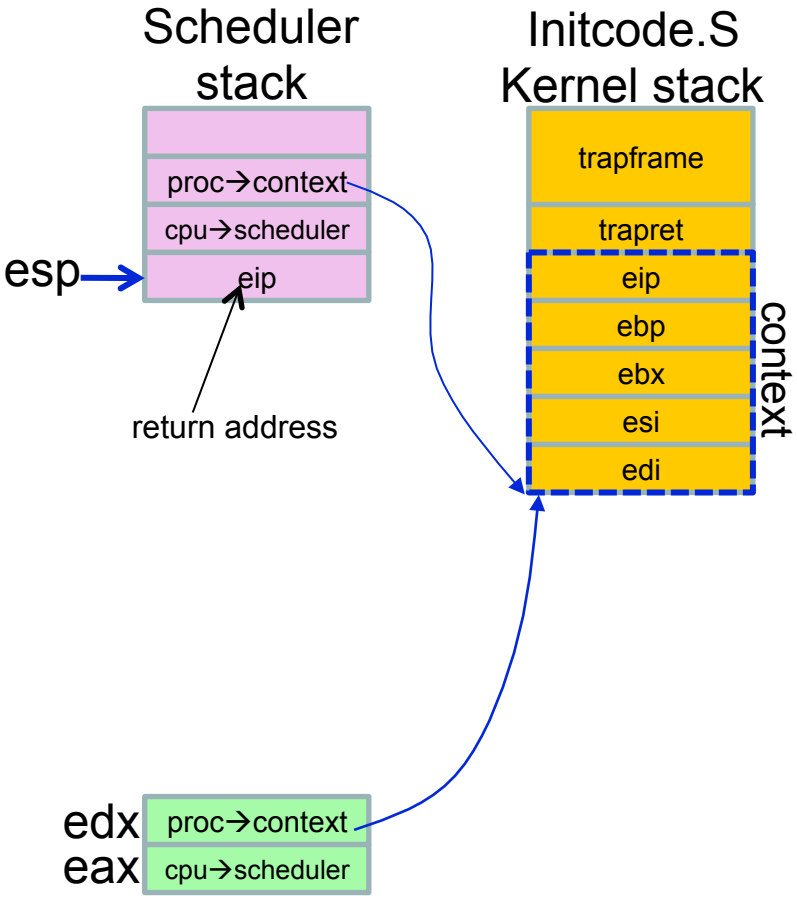
eip →



swtch(cpu→scheduler, proc→context) (2)

```
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```

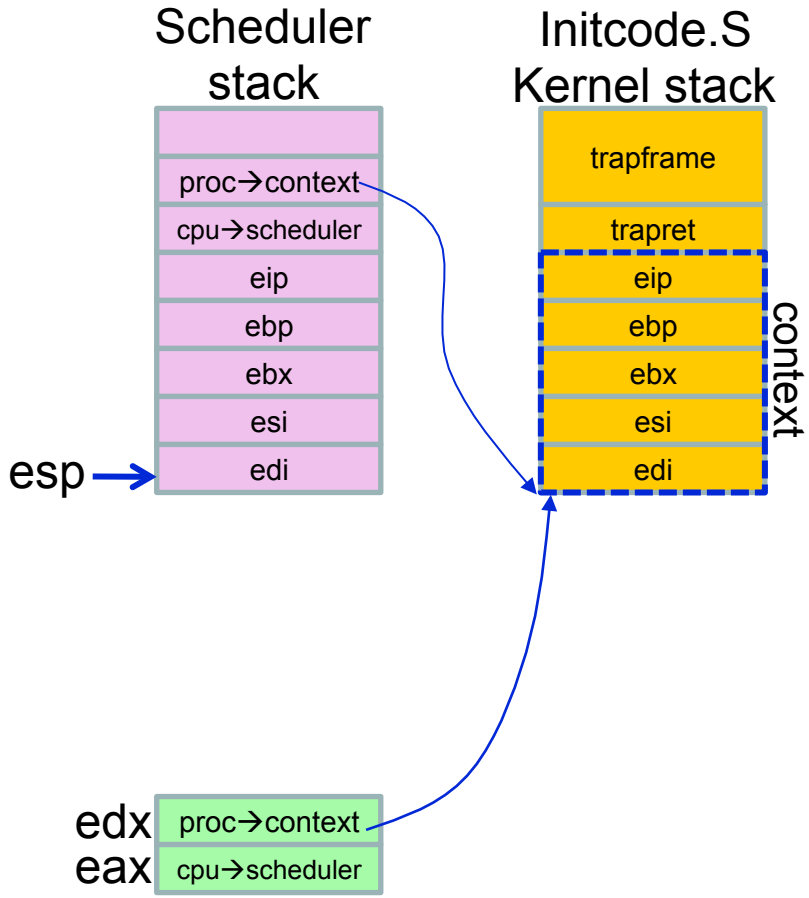
eip →



swtch(cpu→scheduler, proc→context) (3)

```
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```

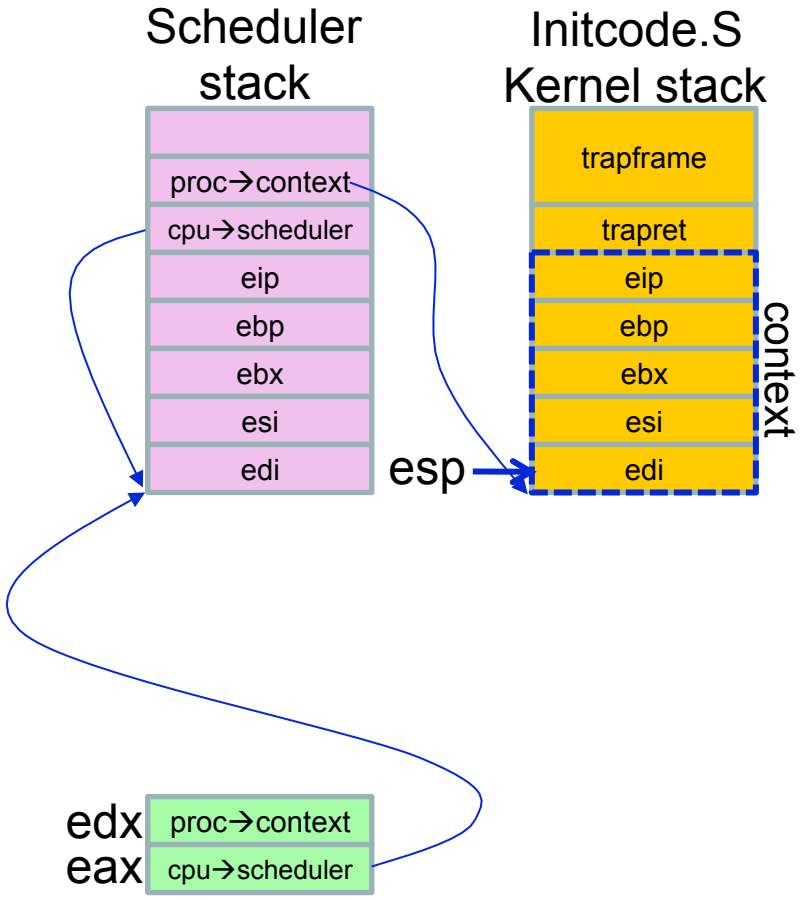
eip →



swtch(cpu→scheduler, proc→context) (4)

```
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```

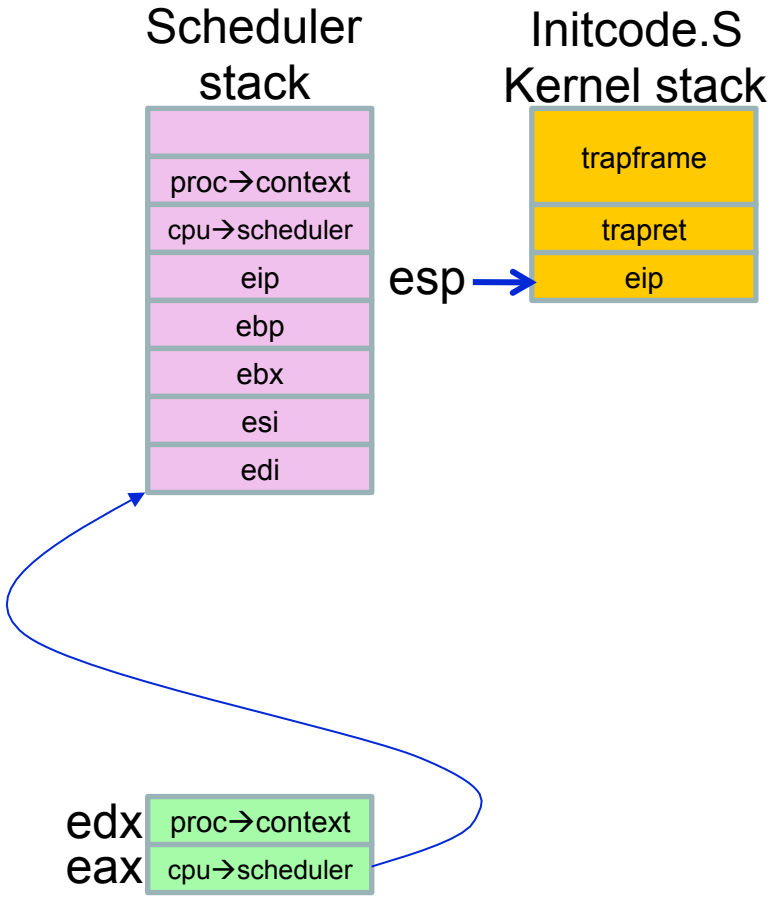
eip →



swtch(cpu→scheduler, proc→context) (5)

```
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```

eip →



So, swtch return corresponds to initcode's eip. Where can that be?

return from swtch

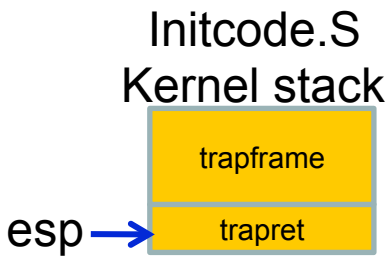
- recollect forkret (a couple of slide back)

  p →context→eip = (uint) forkret;

- So, swtch on return executes forkret

forkret

- Does nothing much.
  - Initilizes a log for the first process
- And then returns to trapret

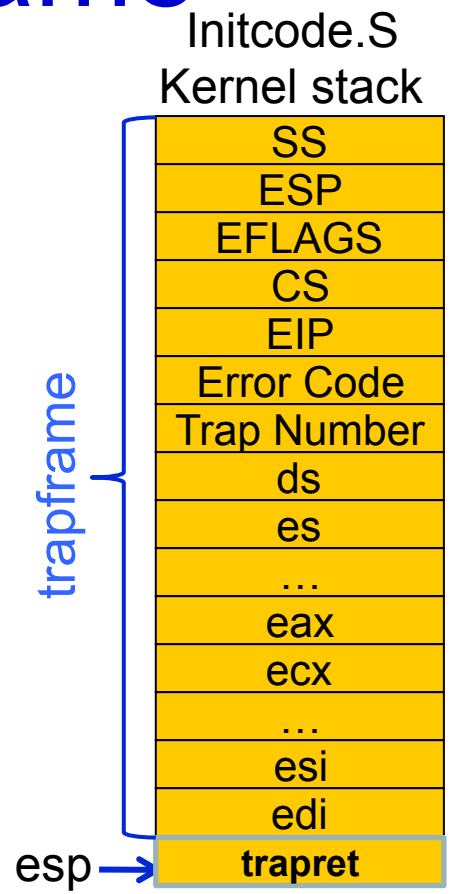



```
2783 forkret(void)
2784 {
2785   static int first = 1;
2786   // Still holding ptable.lock from scheduler.
2787   release(&ptable.lock);
2788
2789   if (first) {
2790     // Some initialization functions must be run in the context
2791     // of a regular process (e.g., they call sleep), and thus cannot
2792     // be run from main().
2793     first = 0;
2794     initlog();
2795   }
2796
2797   // Return to "caller", actually trapret (see allocproc).
2798 }
```

recall the trapframe

- Allocated in allproc.
- Filled in userinit

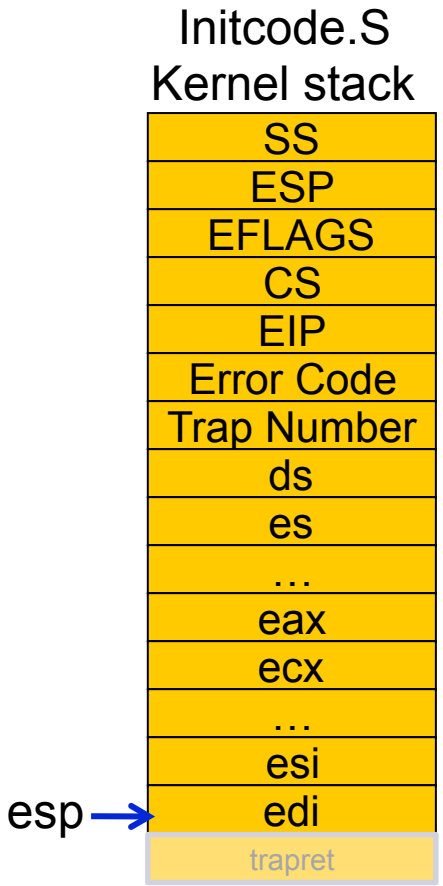
```
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[], _binary_initcode_size[];
2506
2507   p = allocproc();
2508   initproc = p;
2509   if((p->pgdir = setupkvm()) == 0)
2510     panic("userinit: out of memory?");
2511   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2512   p->sz = PGSIZE;
2513   memset(p->tf, 0, sizeof(*p->tf));
2514   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2515   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2516   p->tf->es = p->tf->ds;
2517   p->tf->ss = p->tf->ds;
2518   p->tf->eflags = FL_IF;
2519   p->tf->esp = PGSIZE;
2520   p->tf->eip = 0;  // beginning of initcode.S
2521
2522   safestrcpy(p->name, "initcode", sizeof(p->name));
2523   p->cwd = namei("/");
2524
2525   p->state = RUNNABLE;
2526 }
```



ref : struct trapframe in x86.h (0602 [06])

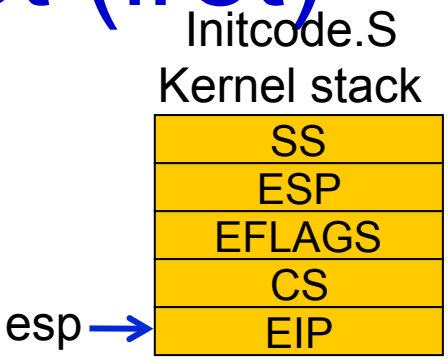
trapret

```
3254 alltraps:
3255   # Build trap frame.
3256   pushl %ds
3257   pushl %es
3258   pushl %fs
3259   pushl %gs
3260   pushal
3261
3262   # Set up data and per-cpu segments.
3263   movw $(SEG_KDATA<<3), %ax
3264   movw %ax, %ds
3265   movw %ax, %es
3266   movw $(SEG_KCPU<<3), %ax
3267   movw %ax, %fs
3268   movw %ax, %gs
3269
3270   # Call trap(tf), where tf=%esp
3271   pushl %esp
3272   call trap
3273   addl $4, %esp
3274
3275   # Return falls through to trapret...
3276 .globl trapret
3277 trapret:
3278   popal
3279   popl %gs
3280   popl %fs
3281   popl %es
3282   popl %ds
3283   addl $0x8, %esp  # trapno and errcode
3284   iret
```



Return from trapret (iret)

```
3254 alltraps:
3255   # Build trap frame.
3256   pushl %ds
3257   pushl %es
3258   pushl %fs
3259   pushl %gs
3260   pushal
3261
3262   # Set up data and per-cpu segments.
3263   movw $(SEG_KDATA<<3), %ax
3264   movw %ax, %ds
3265   movw %ax, %es
3266   movw $(SEG_KCPU<<3), %ax
3267   movw %ax, %fs
3268   movw %ax, %gs
3269
3270   # Call trap(tf), where tf=%esp
3271   pushl %esp
3272   call trap
3273   addl $4, %esp
3274
3275   # Return falls through to trapret...
3276 .globl trapret
3277 trapret:
3278   popal
3279   popl %gs
3280   popl %fs
3281   popl %es
3282   popl %ds
3283   addl $0x8, %esp  # trapno and errcode
3284   iret
```



Loads the new
%cs = SEG_UCODE | DPL_USER
%eip = 0
eflags = 0
%ss = SEG_UDATA | DPL_USER
%esp = 4096 (PGSZE)
…. there by starting initcode.S

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
```

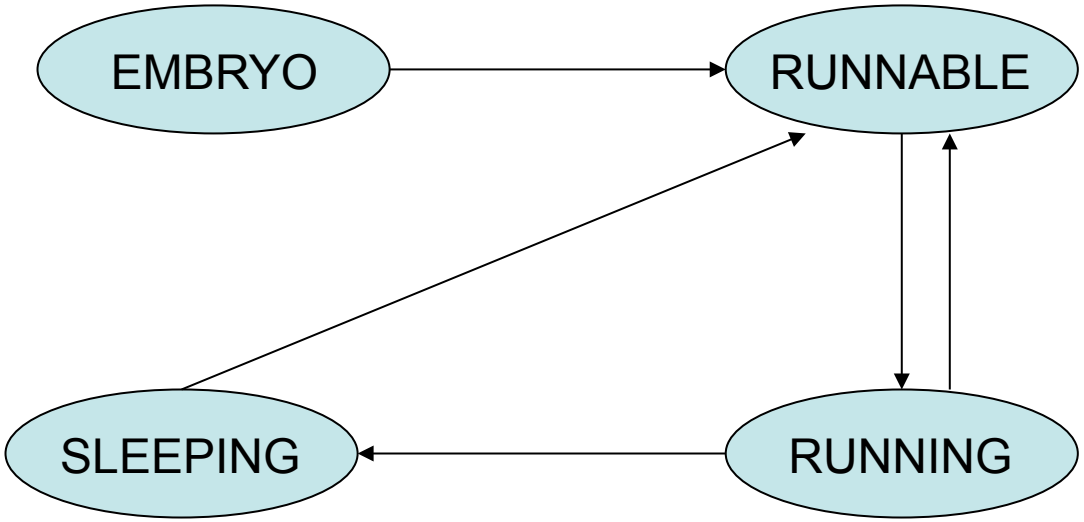
init.c

```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)

CPU Context Switching

Process States



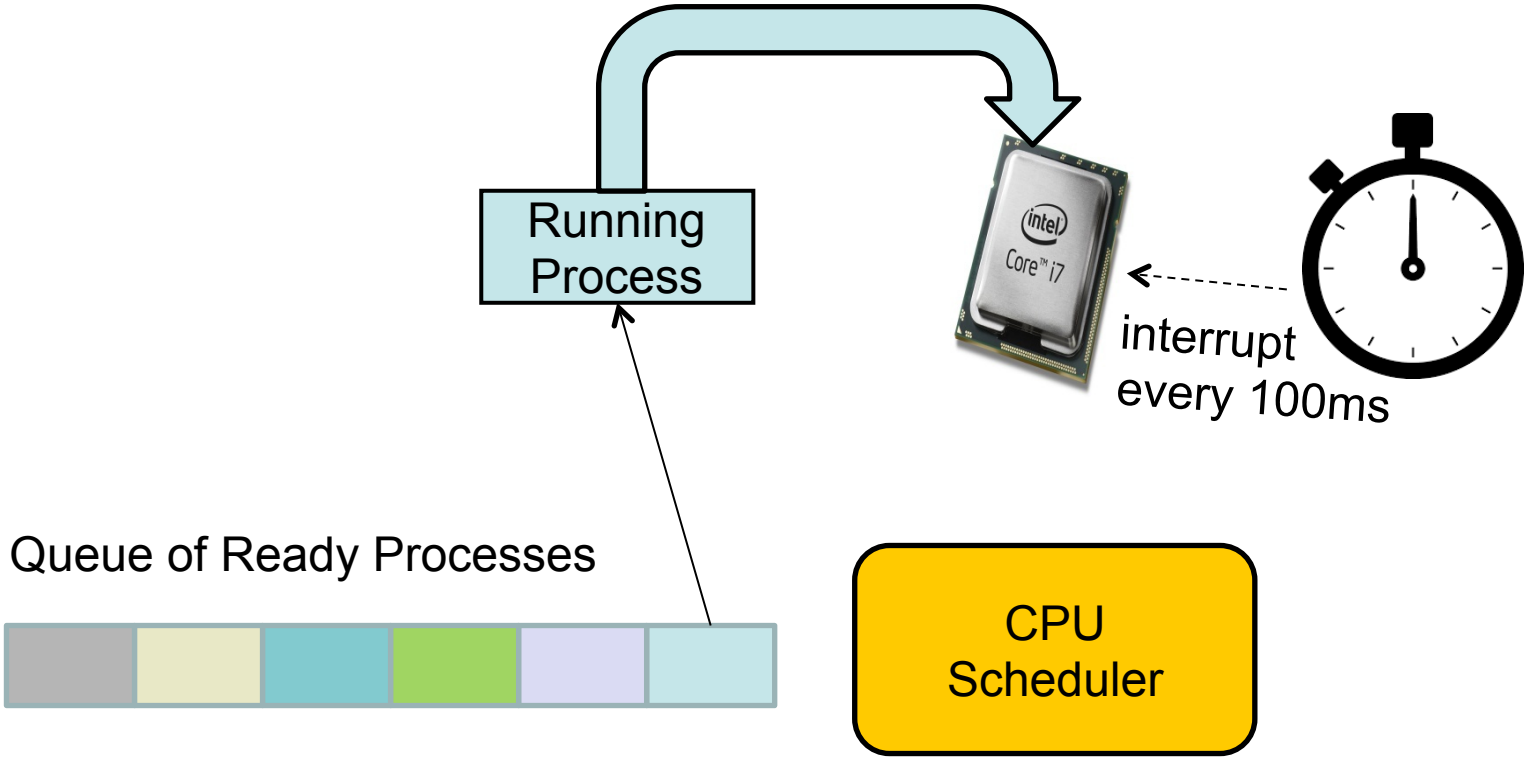
NEW (in xv6 EMBRYO) → The new process is currently being created
READY (in xv6 RUNNABLE) → Ready to run
RUNNING → Currently executing
WAITING (in xv6 SLEEPING) → Blocked for an I/O

ref : proc.h (struct proc) 2100

Context Switches

1. When a process switches from RUNNING to WAITING (eg. due to an I/O request)
2. When a process switches from RUNNING to READY (eg. when an interrupt occurs)
3. When a process switches from WAITING to READY (eg. Due to I/O completion)
4. When a process terminates

The full picture



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

Process Context

- The process context contains all information, which would allow the process to resume after a context switch

Process Contexts Revisited

- Segment registers not needed
  - Since they are constants across kernel contexts
- Caller has saved eax, ecx, edx
  - By x86 convention
- Context contain just 5 registers
  - edi, esi, ebx, ebp, eip
- Contexts always stored at the bottom of the process' kernel stack

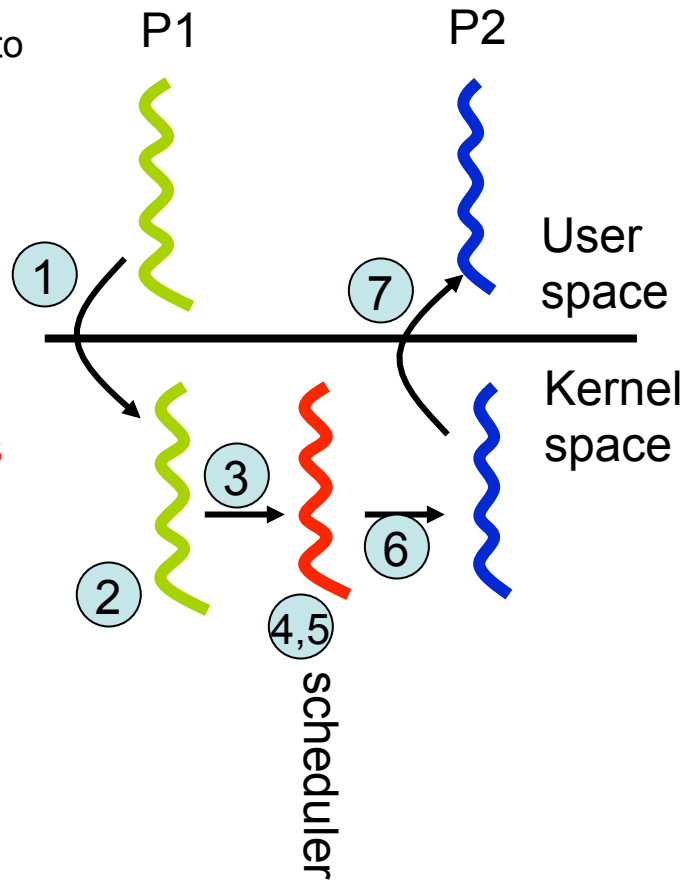
How to perform a context switch?

```
1. Save current process state
2. Load state of the next process
3. Continue execution of the next process
```

- Need to save current process registers without changing them
  - Not easy!! because saving state needs to execute code, which will modify registers
  - Solution : Use hardware + software … architecture dependent

Context switch in xv6

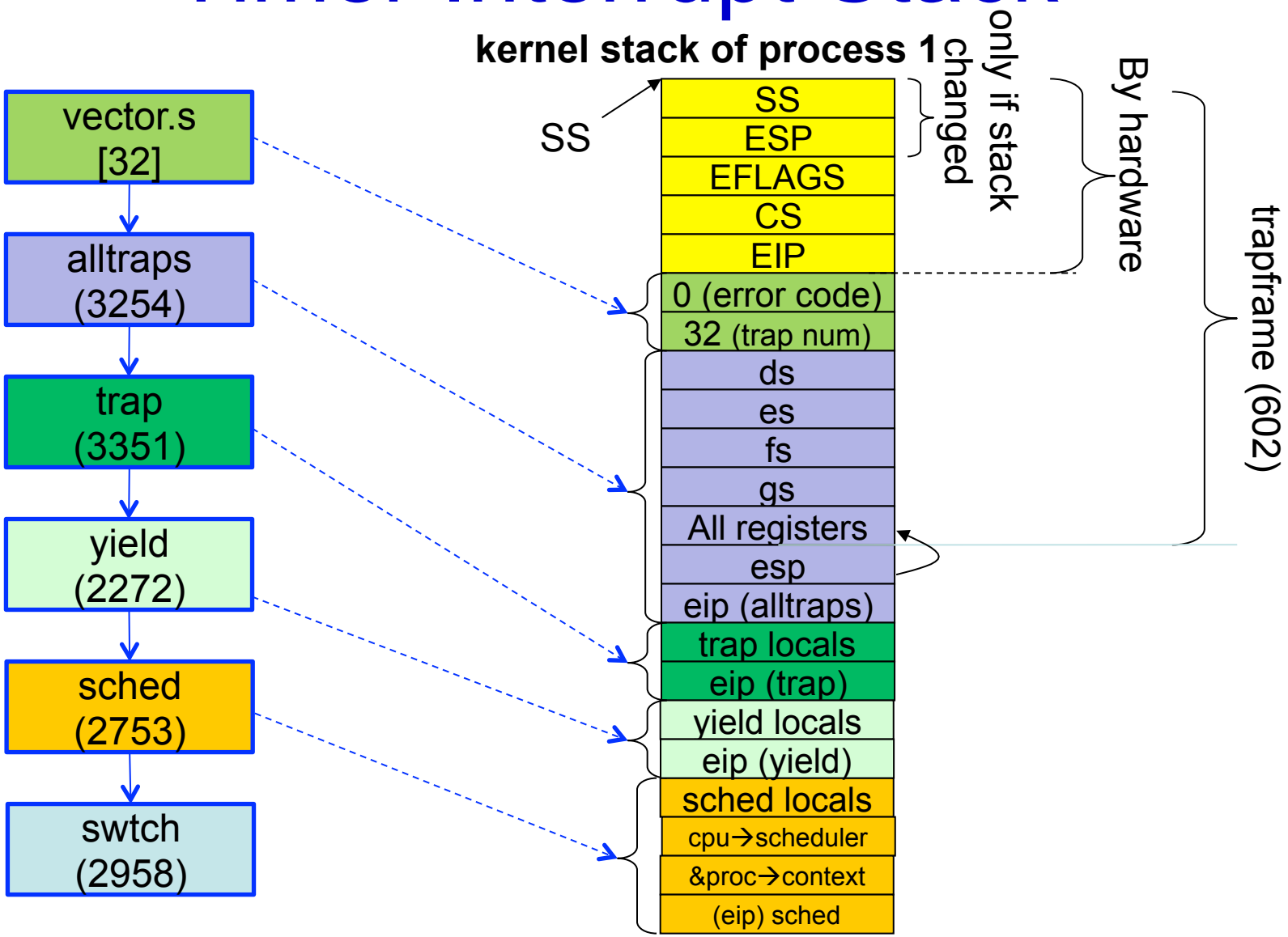
1. Gets triggered when any interrupt is invoked
   - Save P1s user-mode CPU context and switch from user to kernel mode
2. Handle system call or interrupt
3. Save P1's kernel CPU context and switch to scheduler CPU context
4. Select another process P2
5. Switch to P2's address space
6. Save scheduler CPU context and switch to P2's kernel CPU context
7. Switch from kernel to user modeand load P2's user-mode CPU context



Tracing Context Switch (The Timer Interrupts)

- Programming the Timer interval
  - Single Processor Systems : PIT ([80],8054)
  - Multi Processor Systems : LAPIC
- Programmed to interrupt processor every 10ms

Timer Interrupt Stack



trap, yield & sched

trap.c (3423)

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

(2772)

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

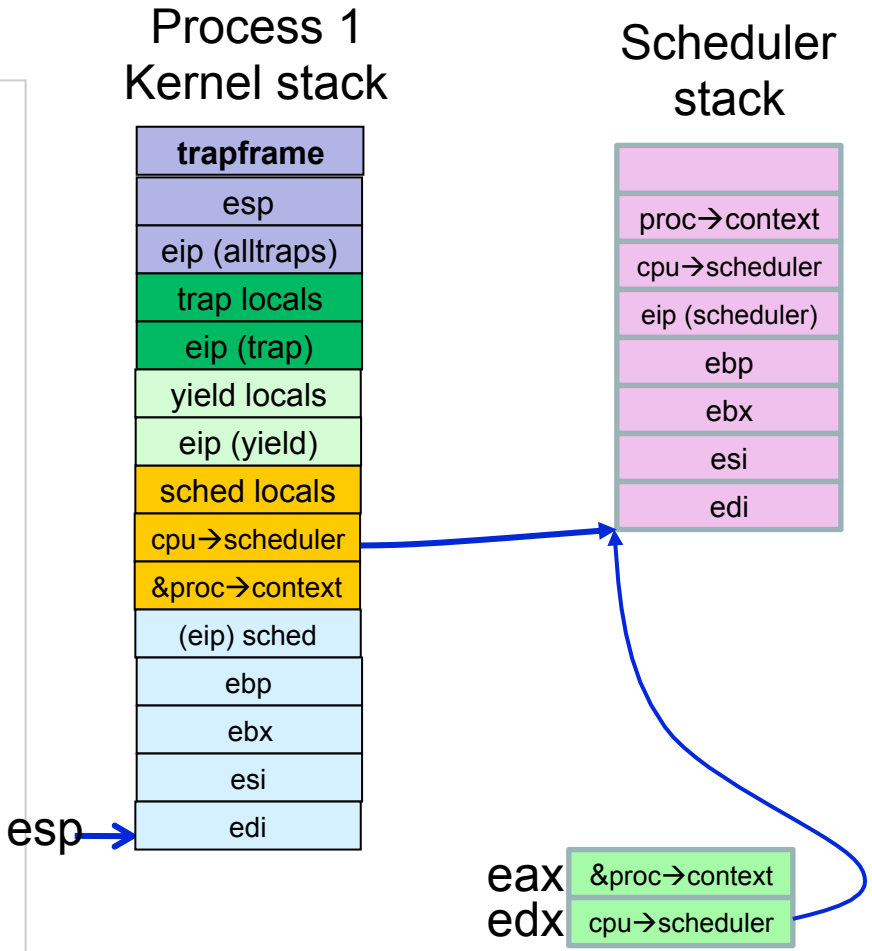
(2753)

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
```

swtch(&proc→context, cpu→scheduler)

```
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```

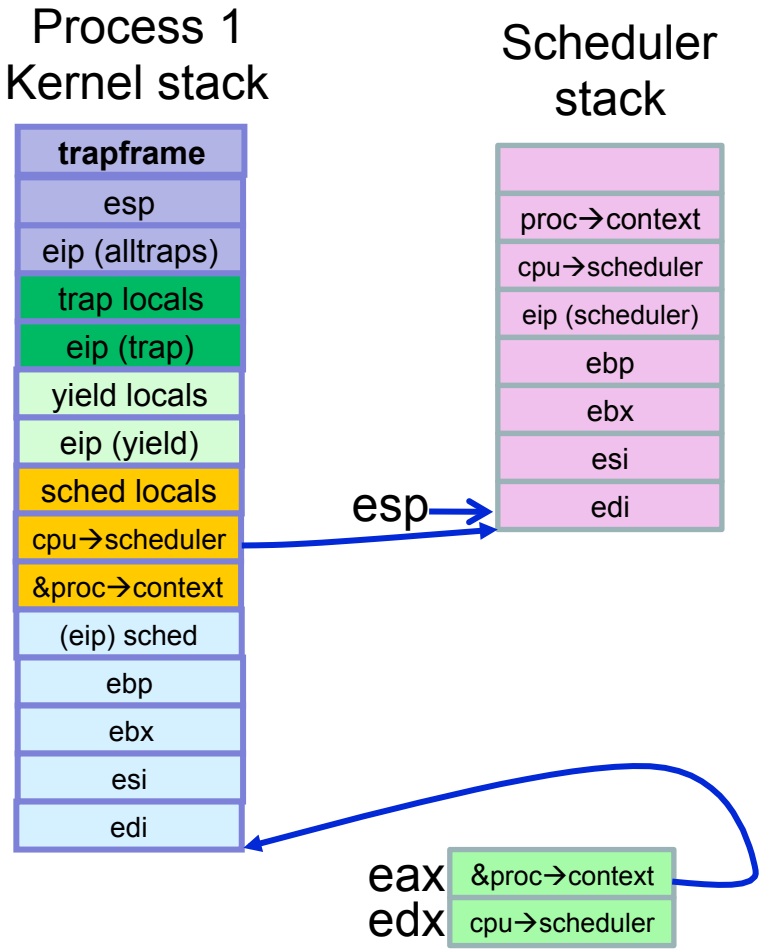
eip →



swtch(&proc→context, cpu→scheduler)

```
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```

eip →



Execution in Scheduler

```
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

eip →

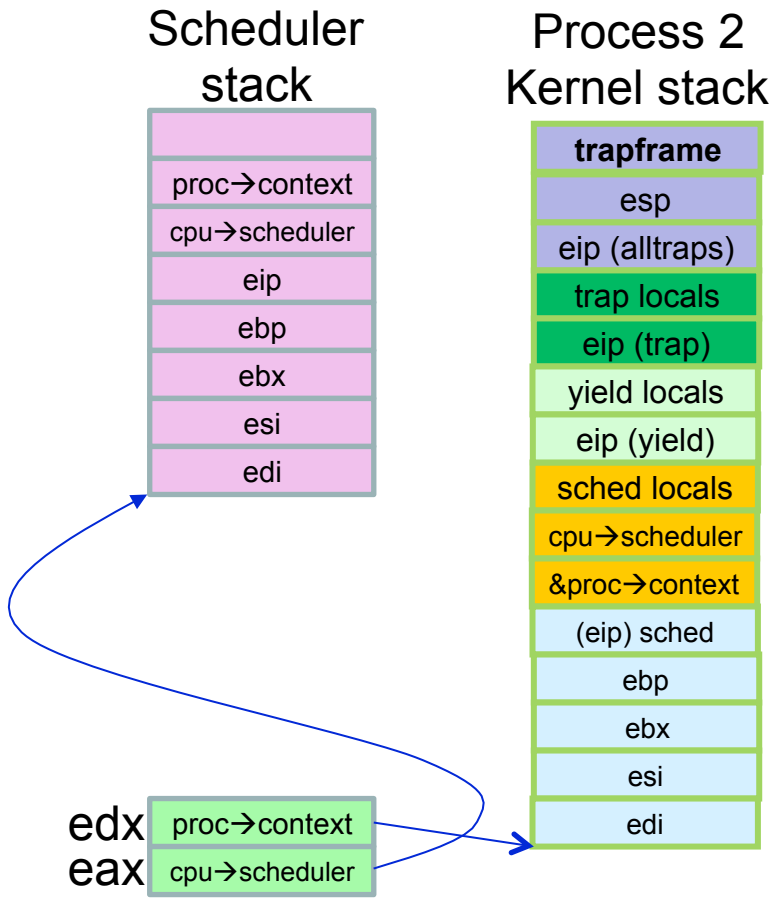
swtch returns to line 2729.

1. First switch to kvm pagetables
2. then select new runnable process
3. Switch to user process page tables
4. swtch(&cpu→scheduler, proc→conetxt)

swtch(&cpu→scheduler, proc→context)

```
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```

eip → 2970



Swtch returns to sched

sched in Process 2's context

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
```

eip →

swtch returns to line 2767.

1. Sched returns to yield
2. Yeild returns to trap
3. Trap returns to alltraps
4. Alltraps restores user space registers of process 2 and invokes IRET

Context Switching Overheads

- Direct Factors affecting context switching time
  - Timer Interrupt latency
  - Saving/restoring contexts
  - Finding the next process to execute
- Indirect factors
  - TLB needs to be reloaded
  - Loss of cache locality (therefore more cache misses)
  - Processor pipeline flush

Context Switch Quantum

- A short quantum
  - Good because, processes need not wait long before they are scheduled in.
  - Bad because, context switch overhead increase
- A long quantum
  - Bad because processes no longer appear to execute concurrently
  - May degrade system performance
- Typically kept between 10ms to 100ms
  - xv6 programs timers to interrupt every 10ms.

System Calls for Process Management

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);
}
```

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

Copying Page Tables of Parent

- copyuvm (in vm.c)
  - replicates parents memory pages
  - Constructs new table pointing to the new pages
  - Steps involved
    1. Call kalloc to allocate a page directory (pgdir)
    2. Set up kernel pages in pgdir
    3. For each virtual page of the parent (starting from 0 to its sz)
       i. Find its page table entry (function walkpgdir)
       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

done by setupkvm

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
  2. Drop reference to in-memory inode
  3. wakeup parent
     - If parent state is sleeping, make it runnable
     - Needed, cause parent may be sleeping due to a wait
  4. Make init adopt children of exited process
  5. Set process state to ZOMBIE
  6. Force context switch to scheduler

note : page directory, kernel stack, not deallocated here

ref : proc.c (exit) 2604

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

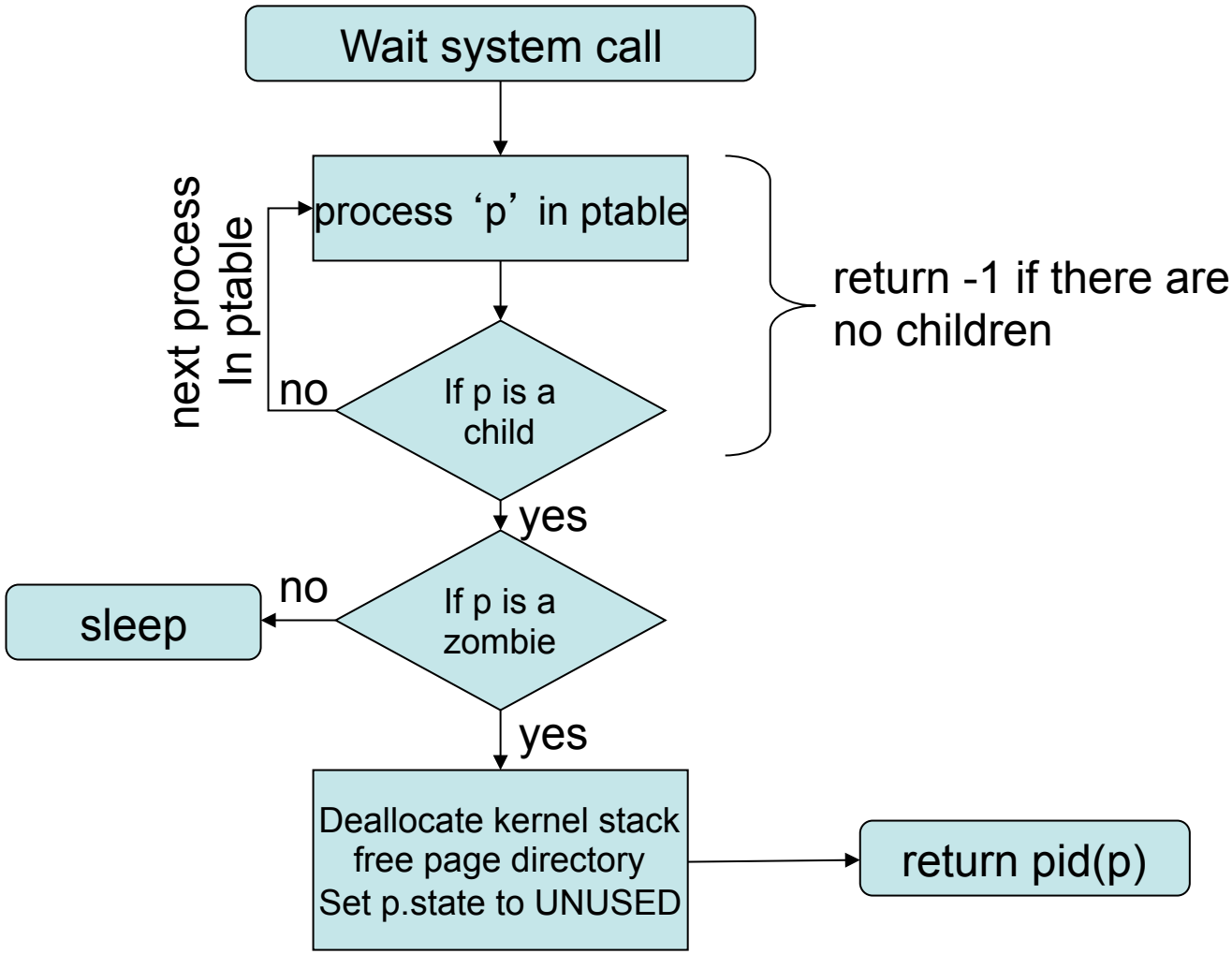
ref : proc.c

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



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

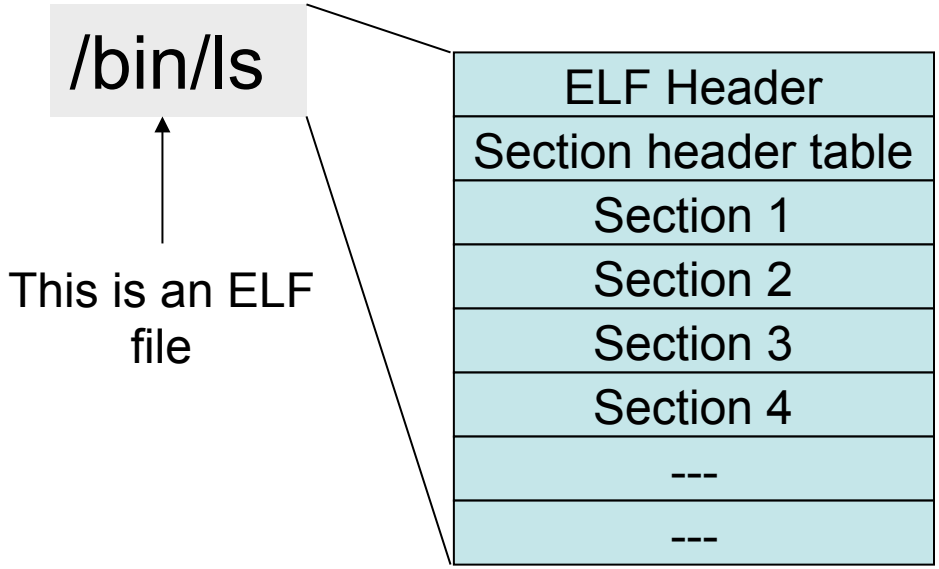
ref : proc.c

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{
  execlp("ls", "", NULL);
  exit(0);
```

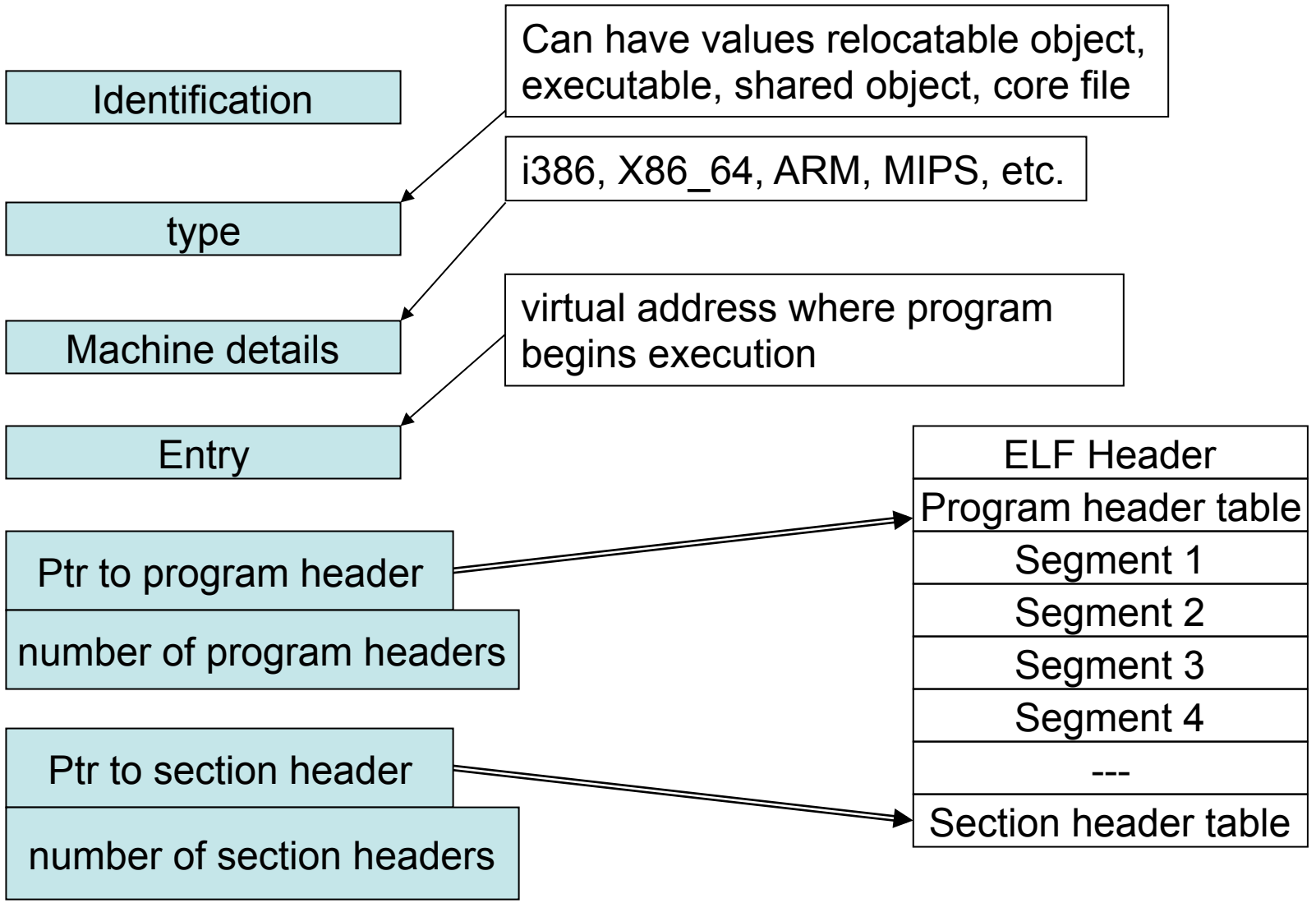
ELF Executables (linker view)



ELF format of executable

ref :www.skyfree.org/linux/references/ELF_Format.pdf
ref :see man elf

ELF Header


Identification
type
Machine details
Entry
Ptr to program header
number of program headers
Ptr to section header
number of section headers
Can have values relocatable object, executable, shared object, core file
i386, X86_64, ARM, MIPS, etc.
virtual address where program begins execution
ELF Header
Program header table
Segment 1
Segment 2
Segment 3
Segment 4

Section header table

Hello World's ELF Header

```c
#include <stdio.h>

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

```
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
```

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

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            0000000000000000 00000000 0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        0000000000000000 00000040 000000000000005c  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  0000000000000018          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)
```

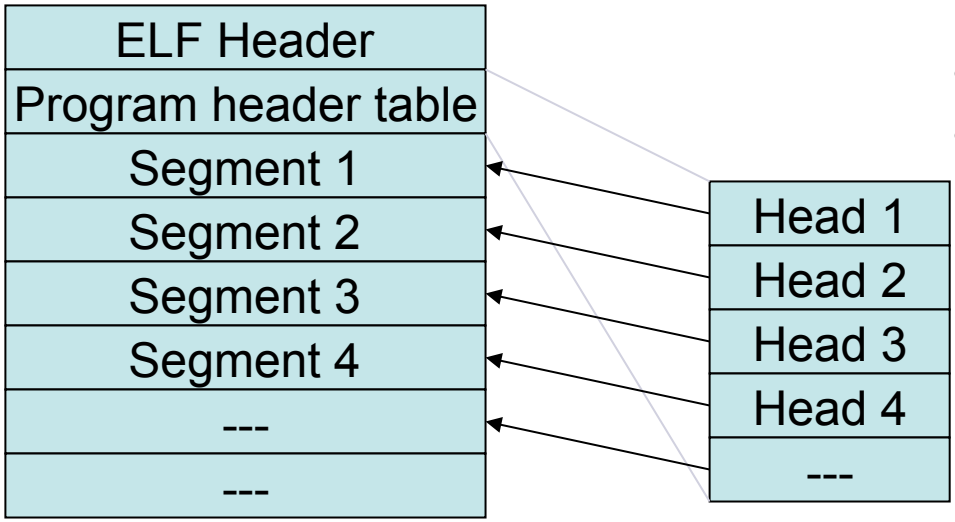
Type of the section
PROGBITS : information defined by program
SYMTAB : symbol table
NULL : inactive section
NOBITS : Section that occupies no bits
RELA : Relocation table

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

Offset and size of the section

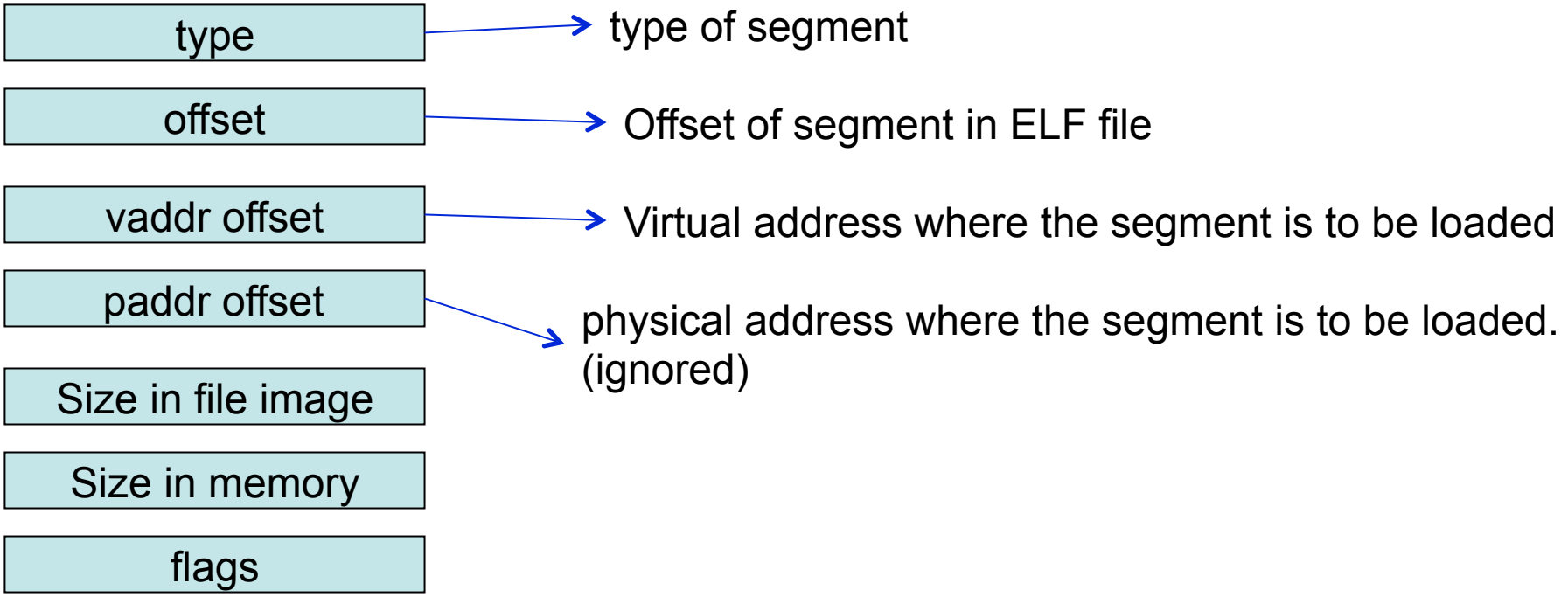
Size of the table if present else 0

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 0x0000000000400040 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000 0x00000000000007b4 0x00000000000007b4  R E    200000
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10 0x0000000000000238 0x0000000000000240  RW     200000
  DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28 0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000000688 0x0000000000400688 0x0000000000400688 0x0000000000000034 0x0000000000000034  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000e10 0x0000000000600e10 0x0000000000600e10 0x00000000000001f0 0x00000000000001f0  R      1

Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gu.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.gu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

Mapping between segments and sections

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

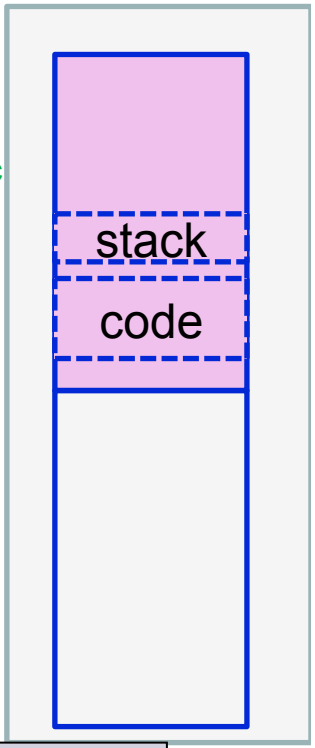
Executable files begin with a signature.

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?

Virtual Memory Map



ref : exec.c

exec contd. (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;
```

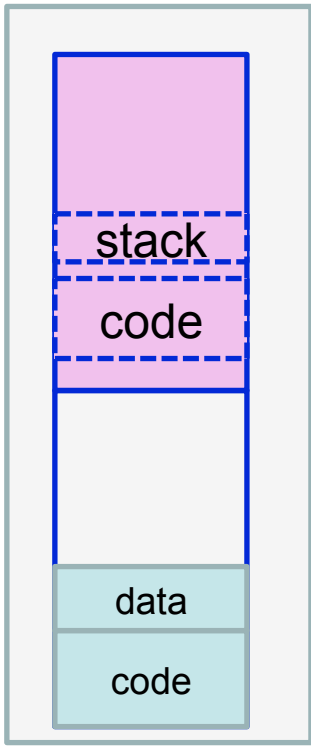
Parse through all the elf program headers.

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)

Virtual Memory Map

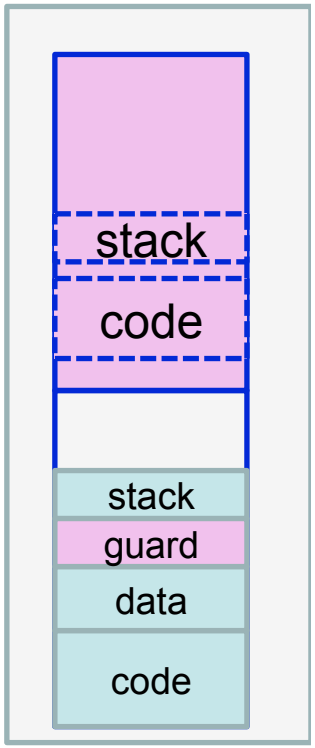


exec contd. (user stacks)

```
// 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;
```

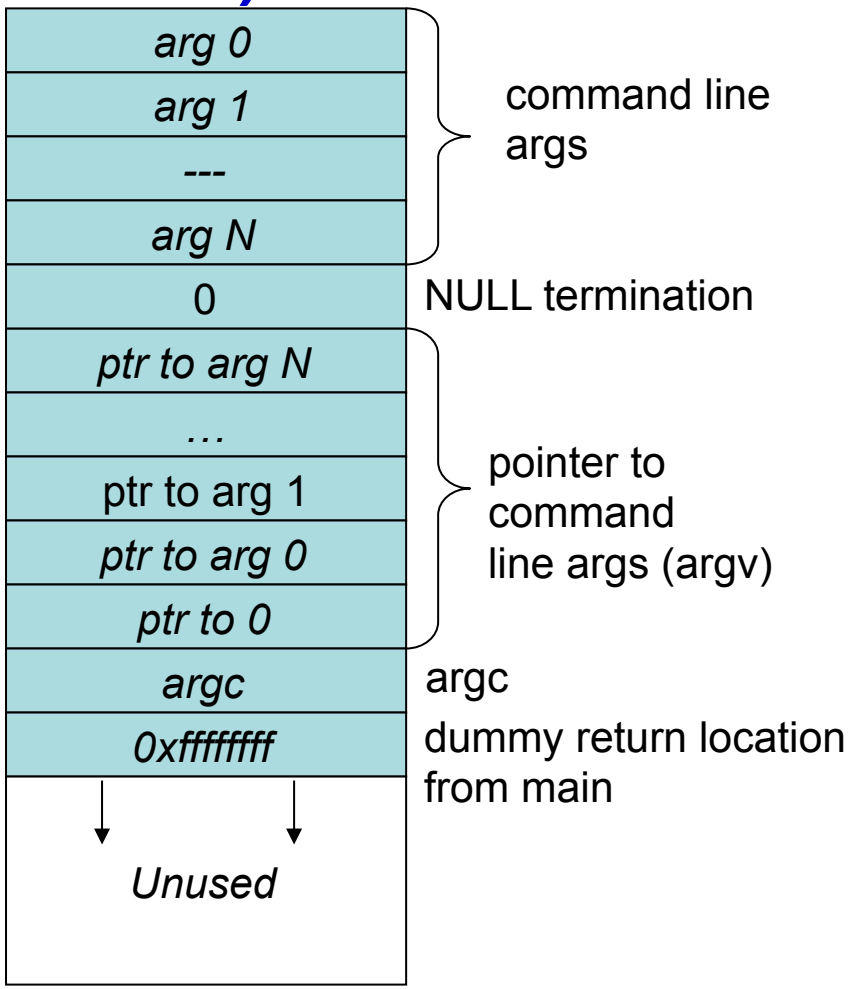
The first acts as a guard page protecting stack overflows

Virtual Memory Map



exec contd. (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;
```



exec contd. (proc, trapframe, etc.)

```
// 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.

Exercise

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