Operating System Design and Implementation

Lecture 6: Processes

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm

Classroom: ED-302

Acknowledgements and Disclaimer

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

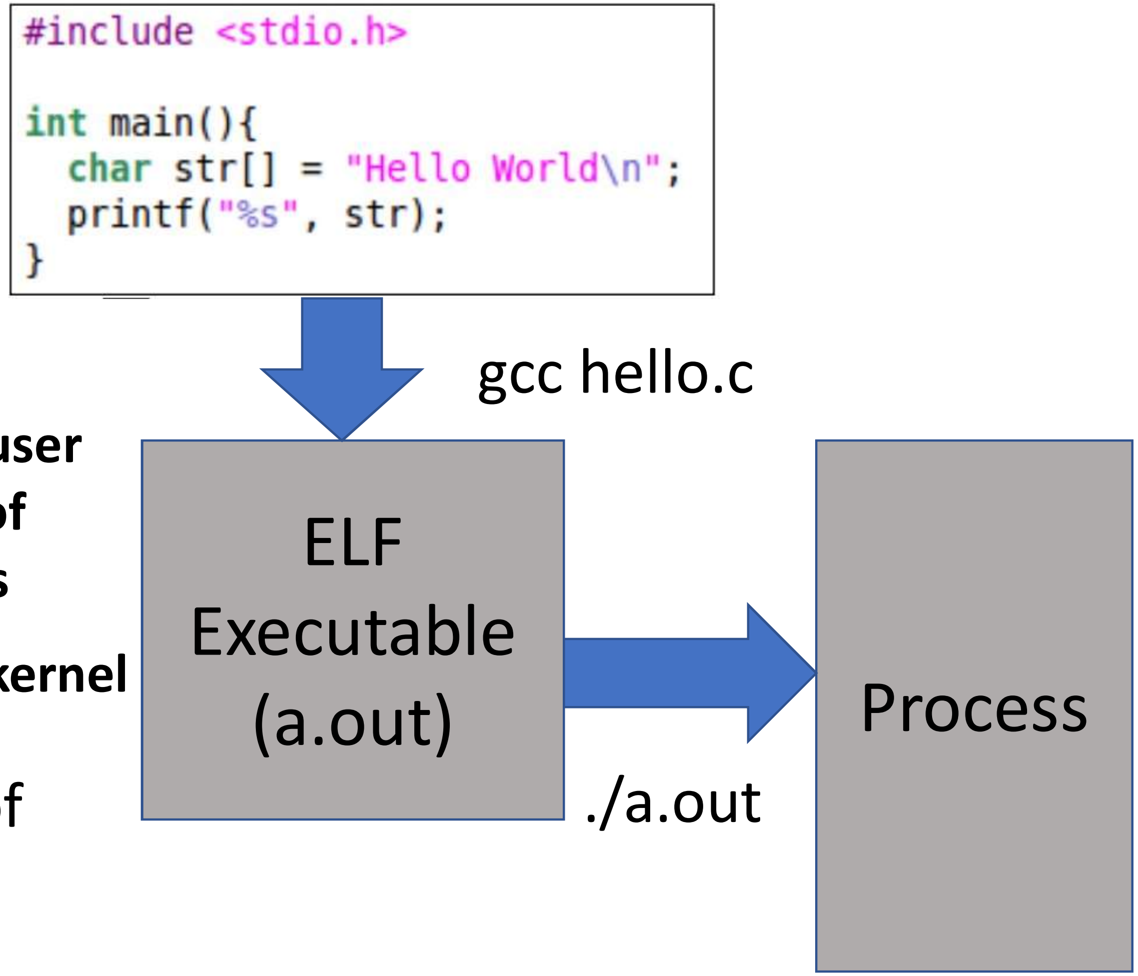
Outline

- Process
  - Process address space
  - Process stacks
  - Process control block
  - Creating the first process

Process

- Process
  - A program in execution
  - Include
    - Code
    - Data
    - Stack
    - Heap
    - State in the OS
    - Kernel stack
  - State contains: registers, list of open files etc.

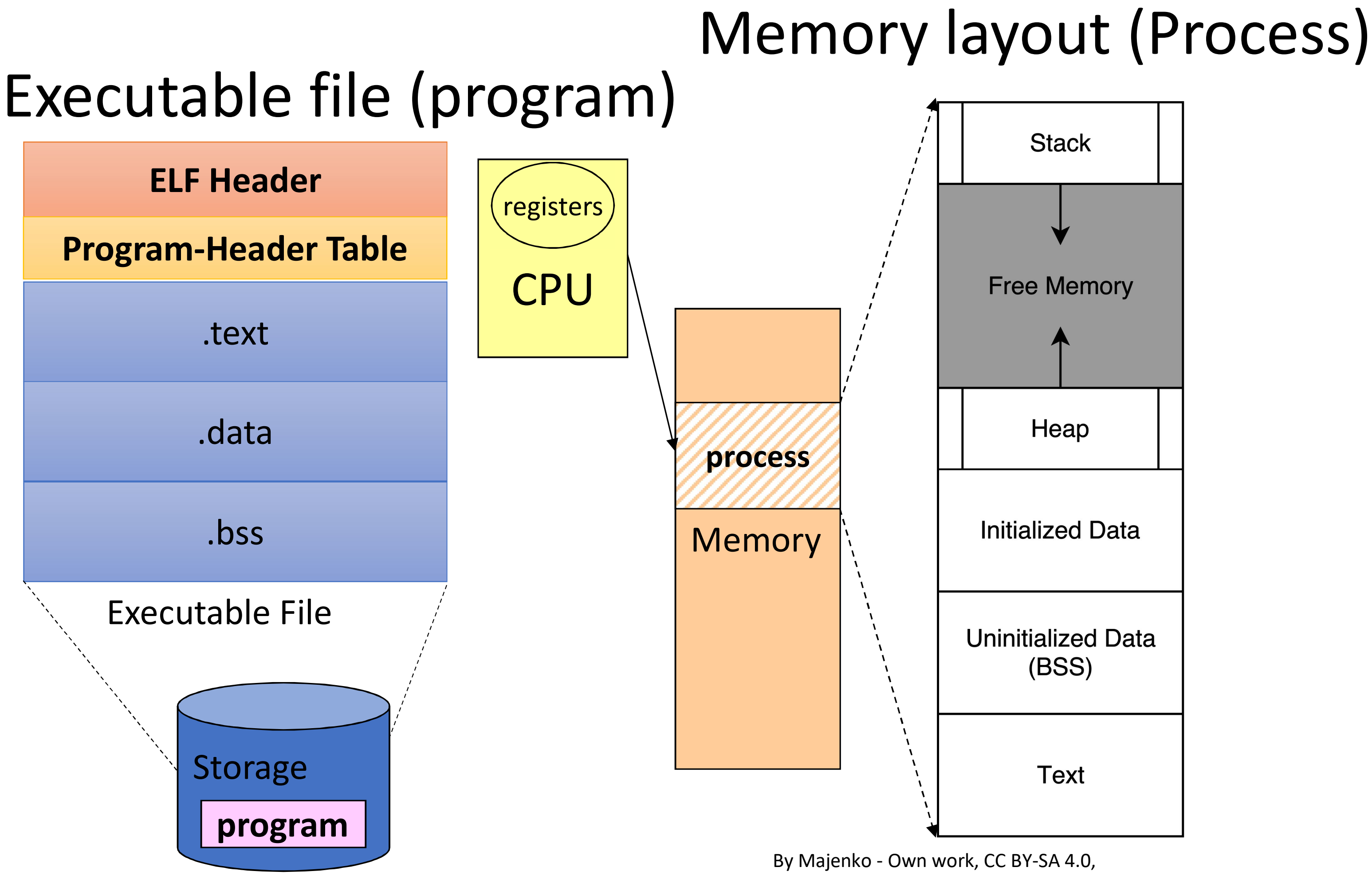
(Code, Data: **From ELF**; Code, Data, Stack, Heap: **In the user space of process**; State in the OS, Kernel stack: **In the kernel space**)



Program ≠ Process

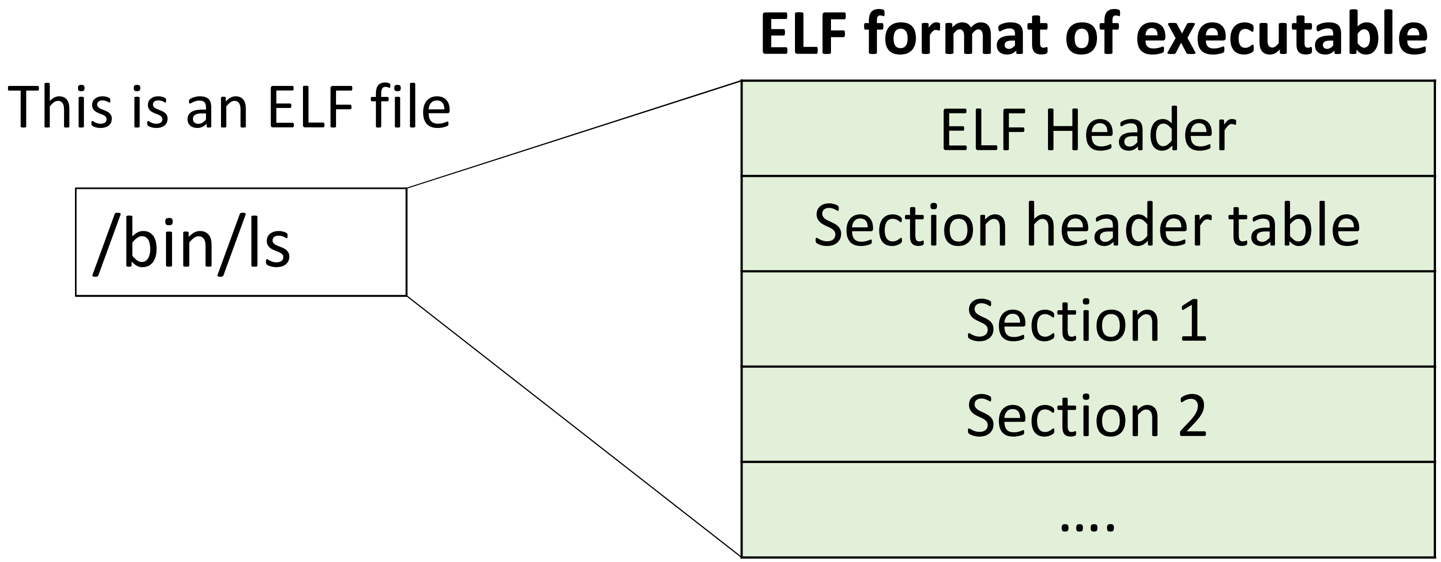
Program	Process
Code + static and global data	Dynamic instruction of code + data + heap + stack + process state
One program can create several processes	A process is unique isolated entity

Program vs. Process

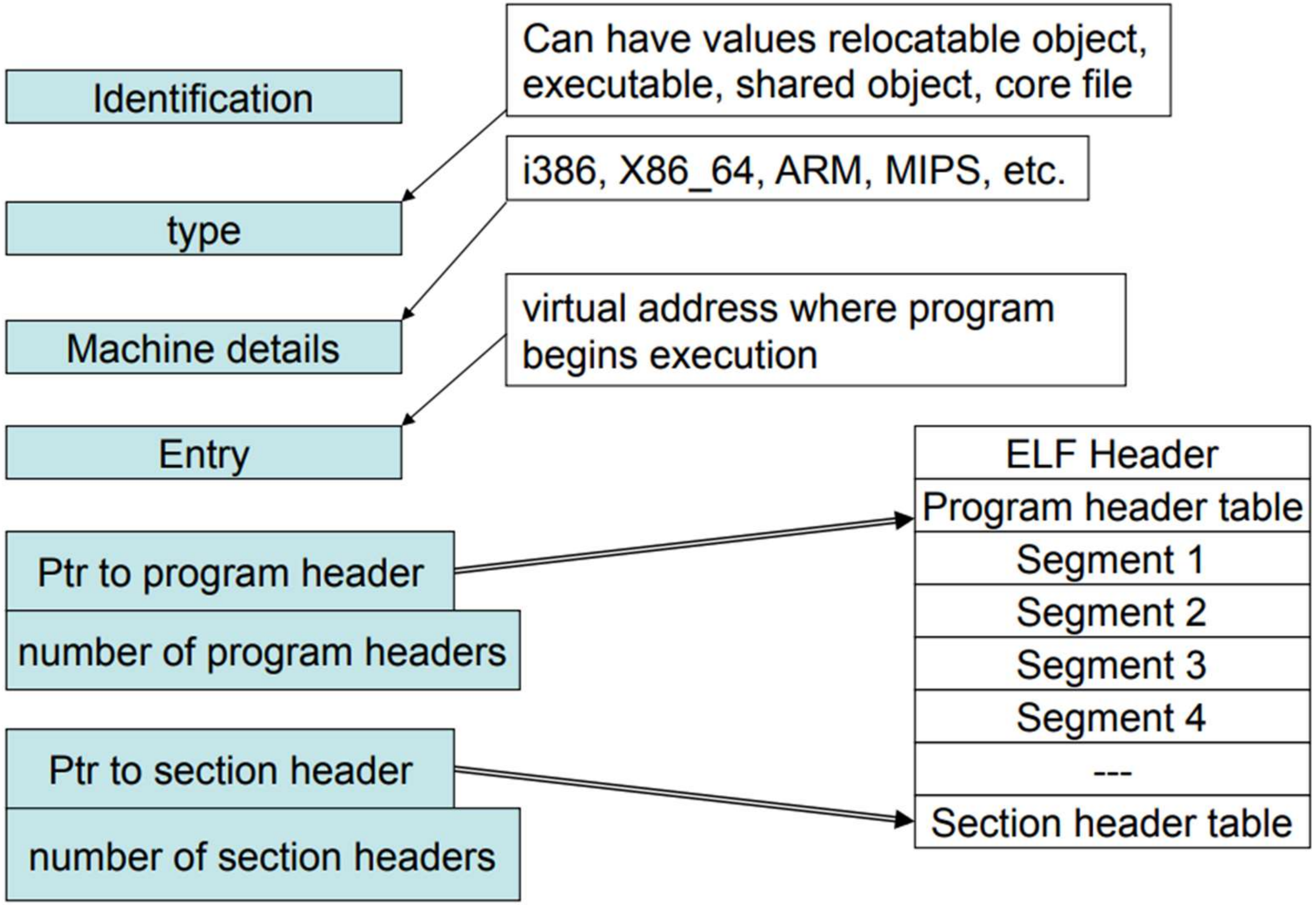


ELF executables (linker view)

- Section comprises all information needed for linking a target object file to build an executable
  - E.g. .text, .data, .rodata, .bss, .plt, .got …



ELF header



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

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)
```

Offset and size of the section

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)

Size of the table if present else 0

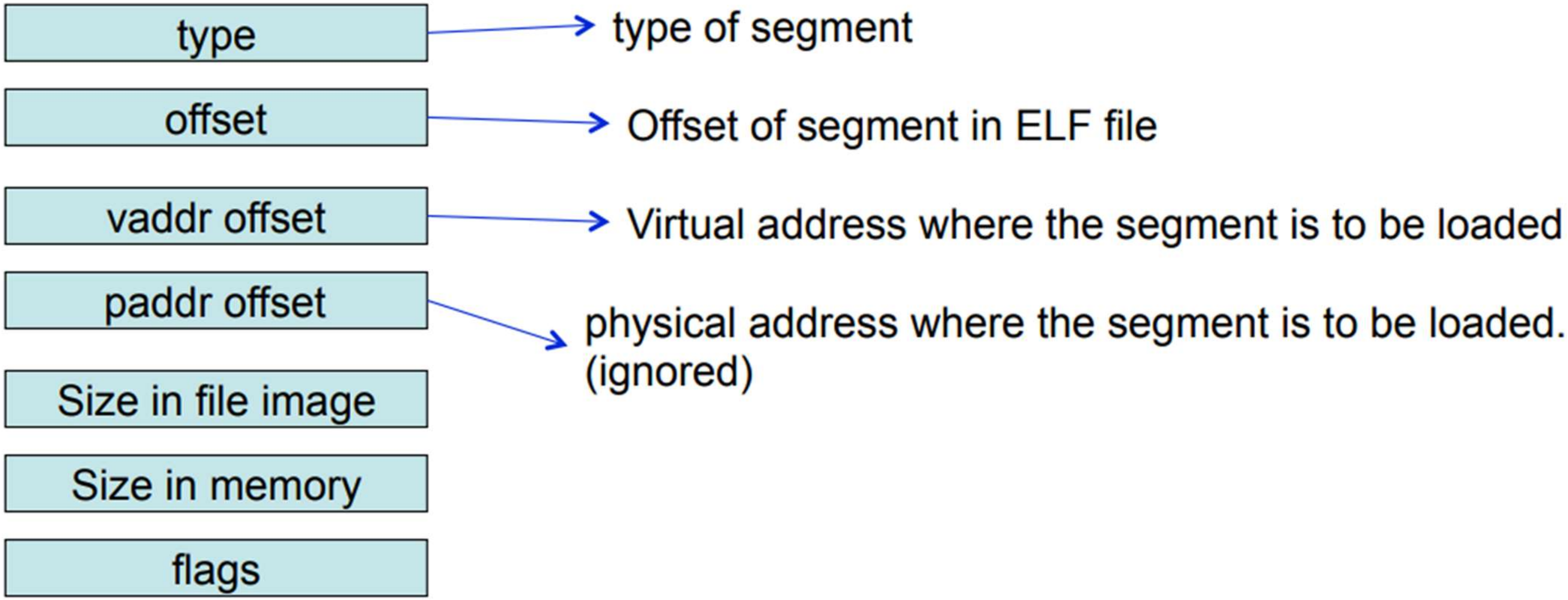
http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Program header (executable view)

- Program headers split the executable into segments with different attributes, which will be loaded into memory
- No need on link time
- A program header entry contains
  - 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.

ELF Header
Program header table
Segment 1
Segment 2
....

Program header contents



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

Program headers for hello world executable

- readelf –l hello

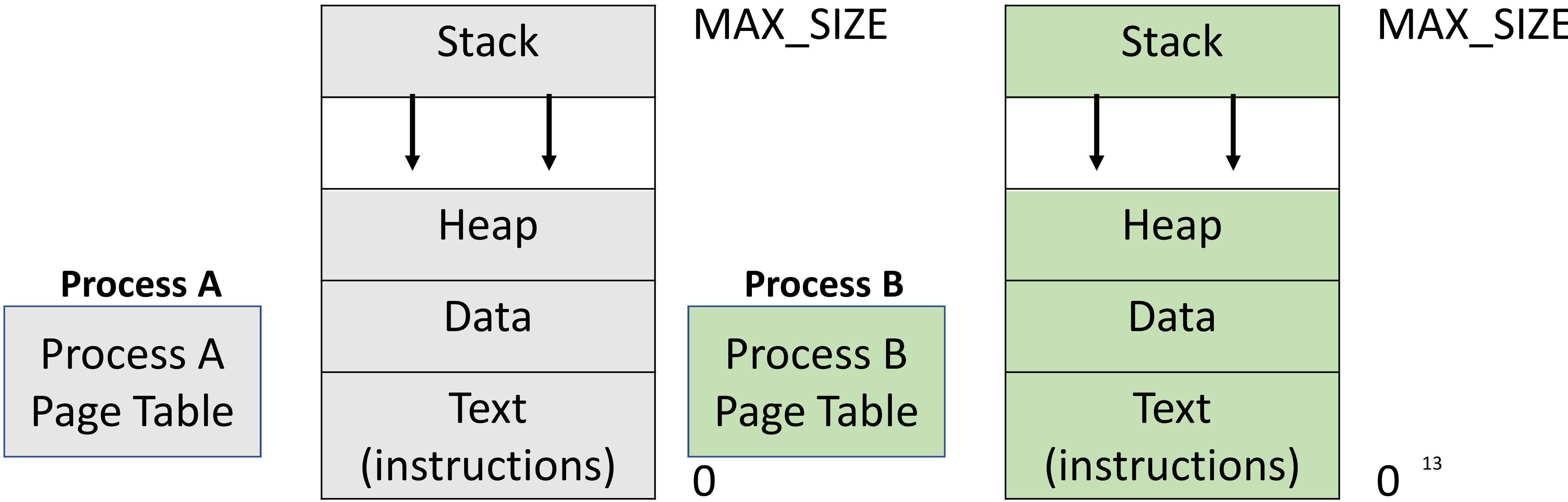
```
Elf file type is EXEC (Executable file)
Entry point 0x4004b0
There are 9 program headers, starting at offset 64

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.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

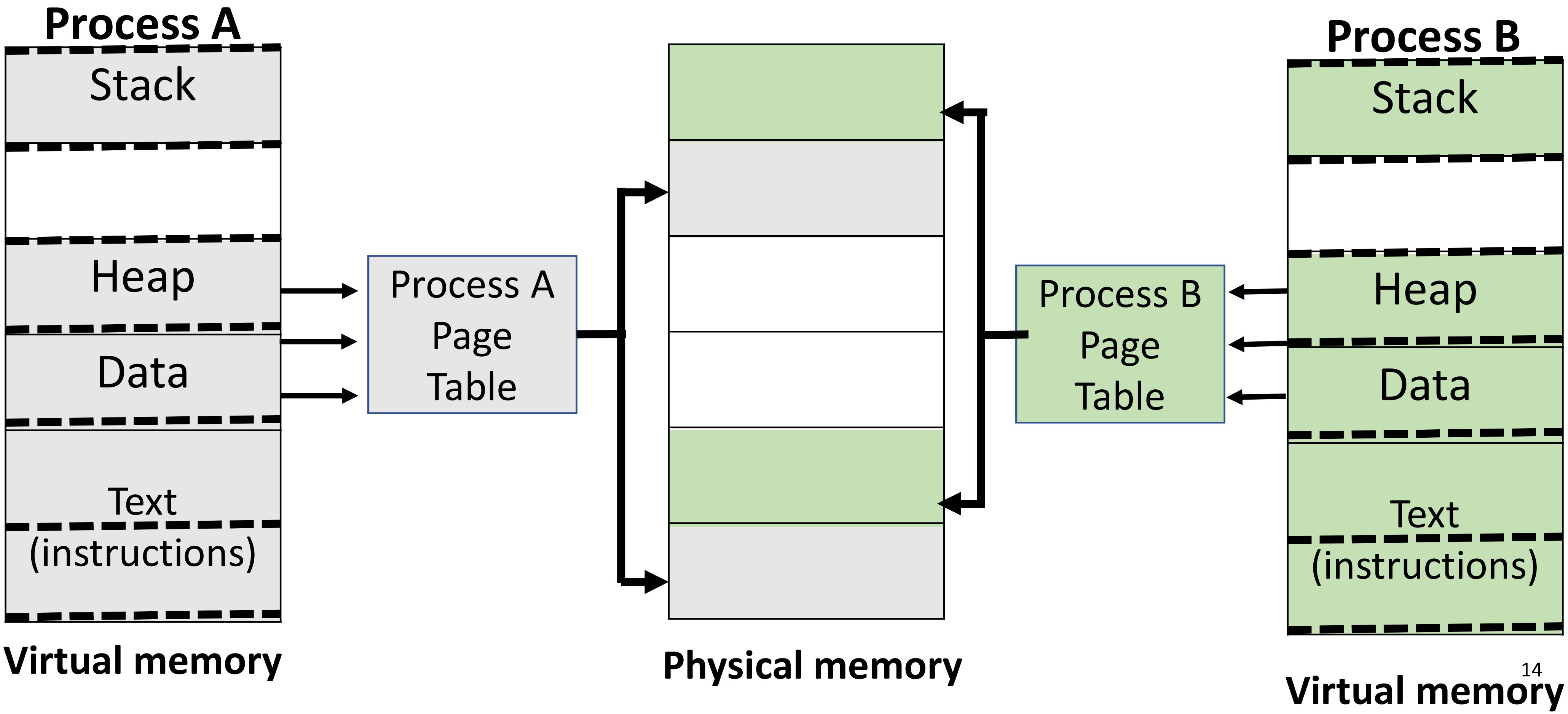
http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Process address space

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



Virtual address mapping

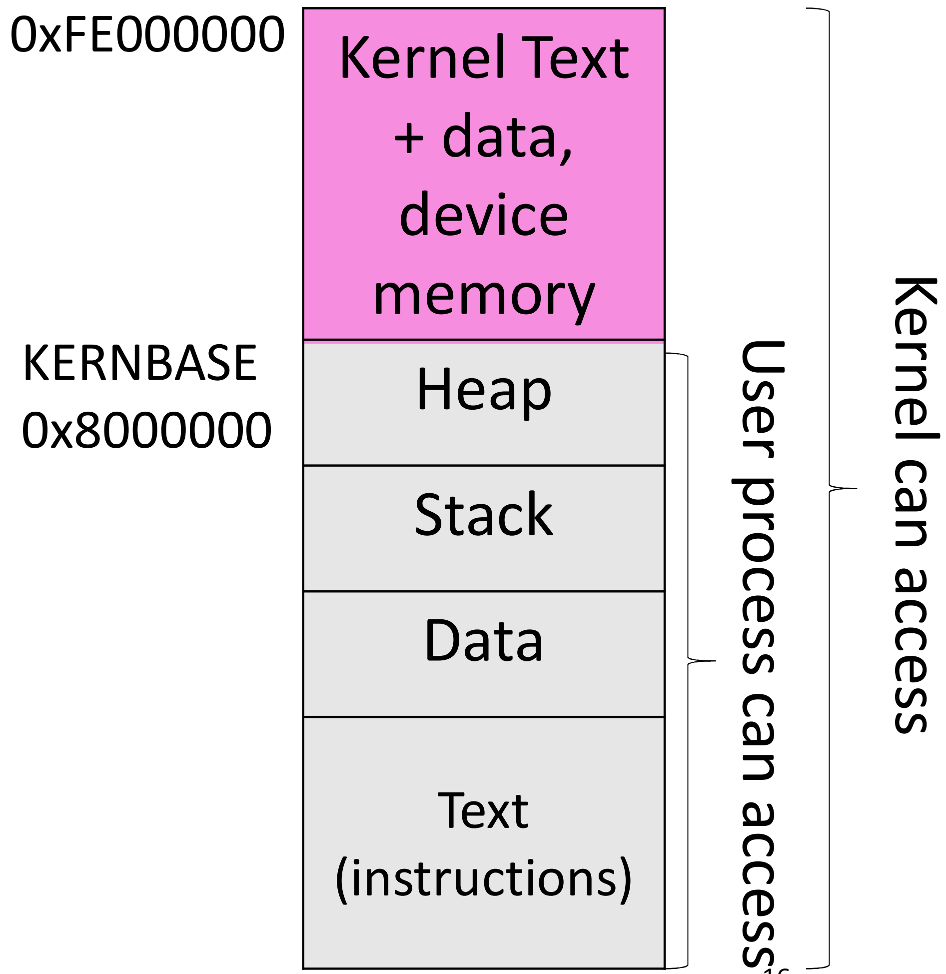


Advantage of virtual address map

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

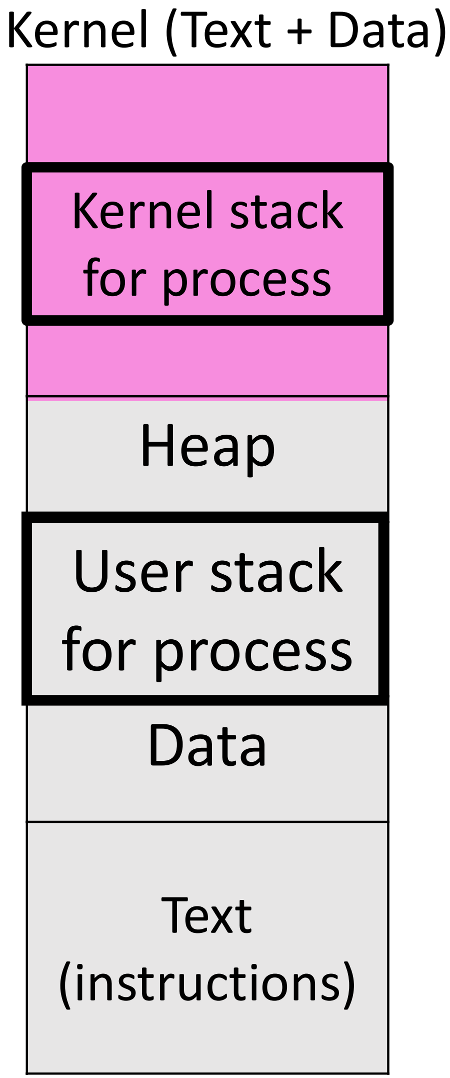
Process address map in xv6

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



Process stacks

- Each process has two stacks
  - **User space stack**
    - Used when executing user code
  - **Kernel space stack**
    - Used when executing kernel code (e.g. during system calls)
  - **Advantage**:
    - Kernel can execute even user stack is corrupted
    - For instance, buffer overflow attack in user stack won’t affect the kernel



Process management

- Each process has a **PCB** (process control block)
  - Holds important process specific information in PCB
- Why does a process need PCB ?
  - Allow process to resume execution after a while
  - Keep track of resources used
  - Track the process state

Entries of PCB in xv6

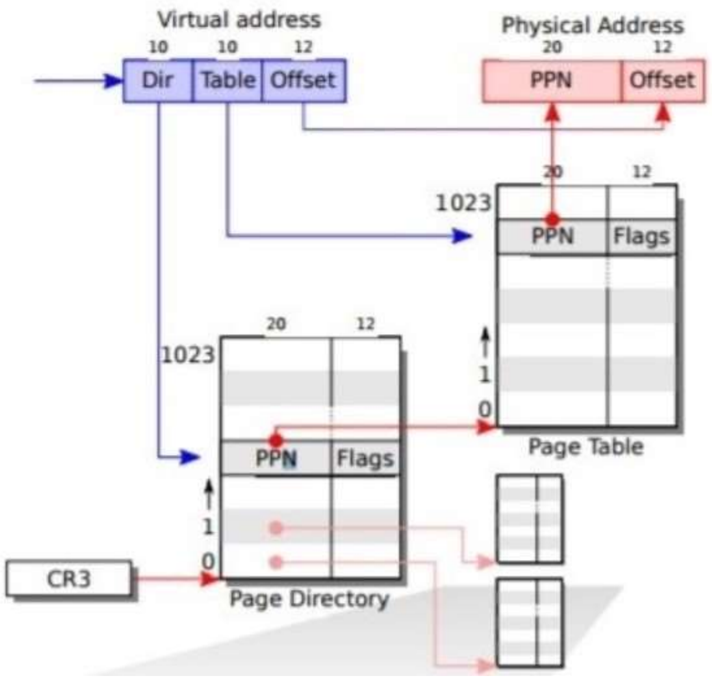
proc.h

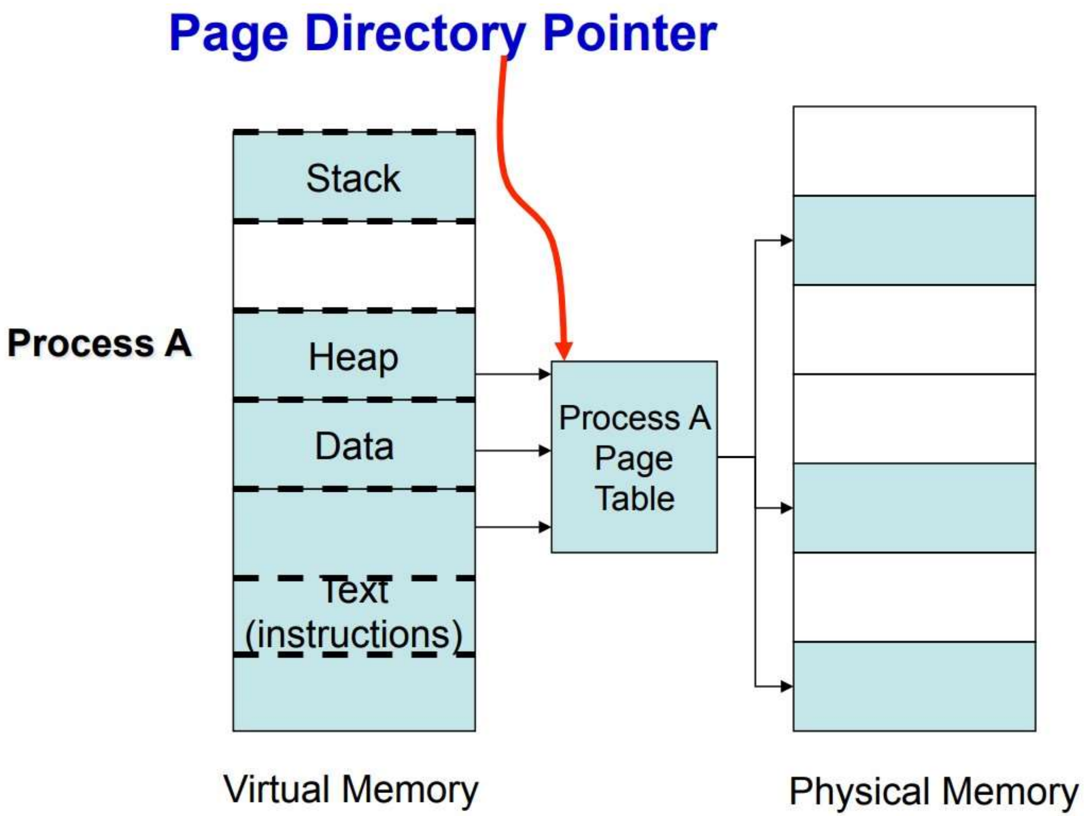
```
struct proc {
        uint                    sz;
        pde_t*                  pgdir;
        char                    *kstack;
        enum procstate          state;
        int                     pid;
        struct proc             *parent;
        struct trapframe        *tf;
        struct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

- sz → Size of process memory
- pgdir → Page directory pointer for process
- *kstack → Kernel stack pointer
- *ofile[NOFILE] → Files opened
- *cwd → Current working directory
- name[16] → Executable name

Page directory pointer

- Page directory pointer
  - Point to the page directory

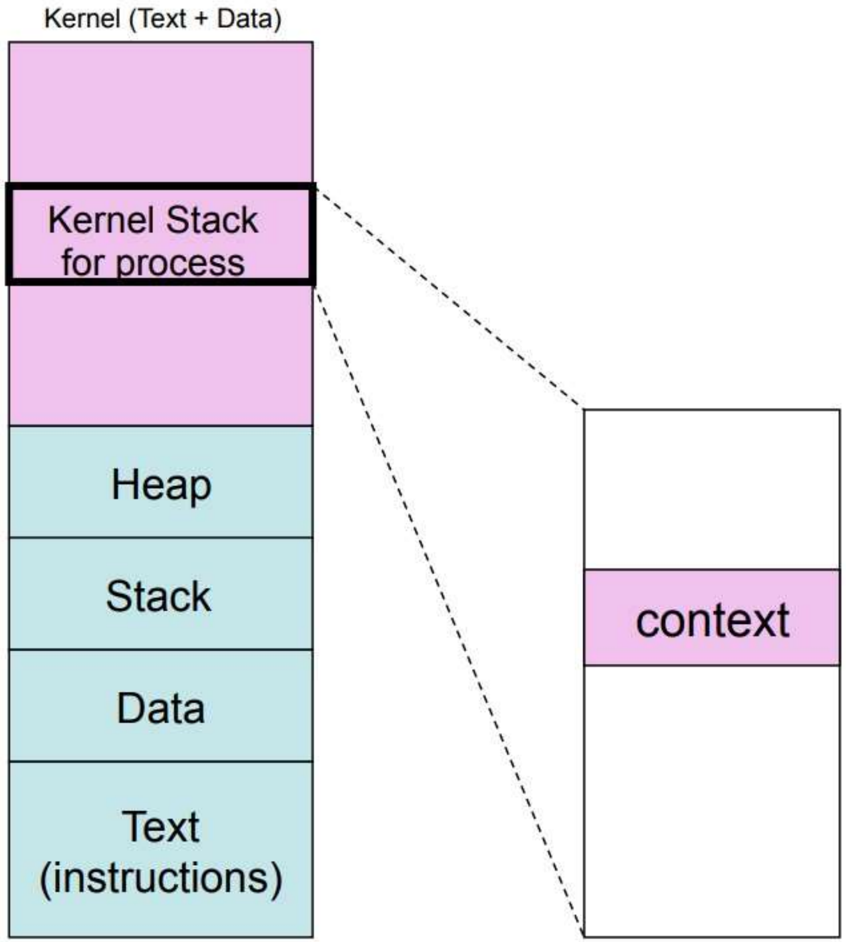




http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Context pointer

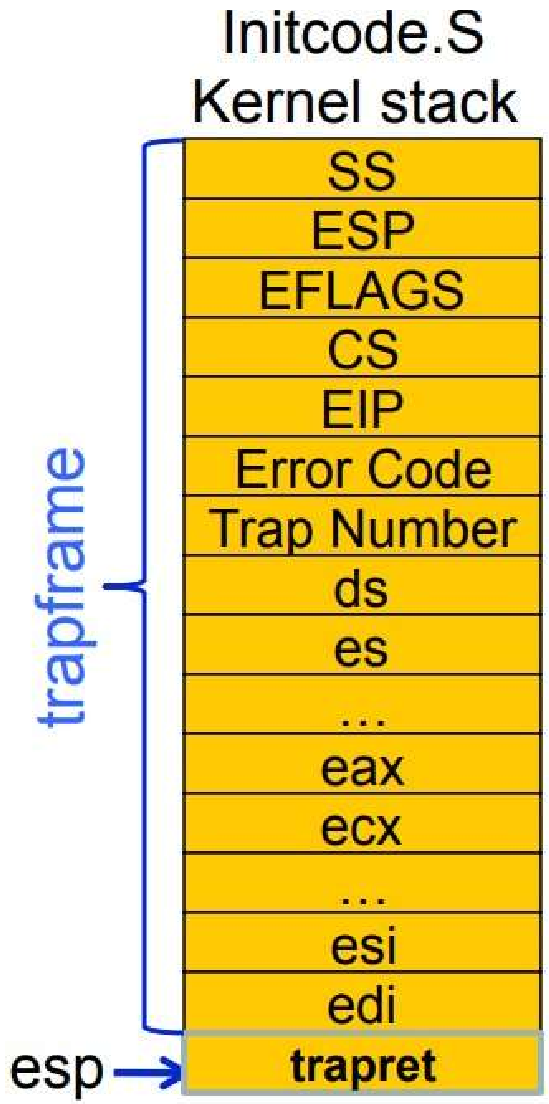
- **Context pointer**
  - Contains registers used for context switches
  - Registers in context
    - %edi, %esi, %ebx, %ebp, %eip
  - Stored in the kernel stack space



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

Trapframe

- **Trapframe**
  - Process state is pushed on the kernel stack during trap handling
  - CPU context of where execution stopped is saved, so that it can resume after trap
  - Some extra information needed by trap handler is also saved



Process table

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

- **The process table**
  - An array of PCB in Linux kernel
  - Contains PCB's for all of the current processes in the system
  - Includes **Process ID, Process priority, process state, process resource usage**
- **Storing process in xv6**
  - NPROC is the maximum number of processes that can be present in the system (#define NPROC 64)
  - Also present in process table is a lock that series access to the array

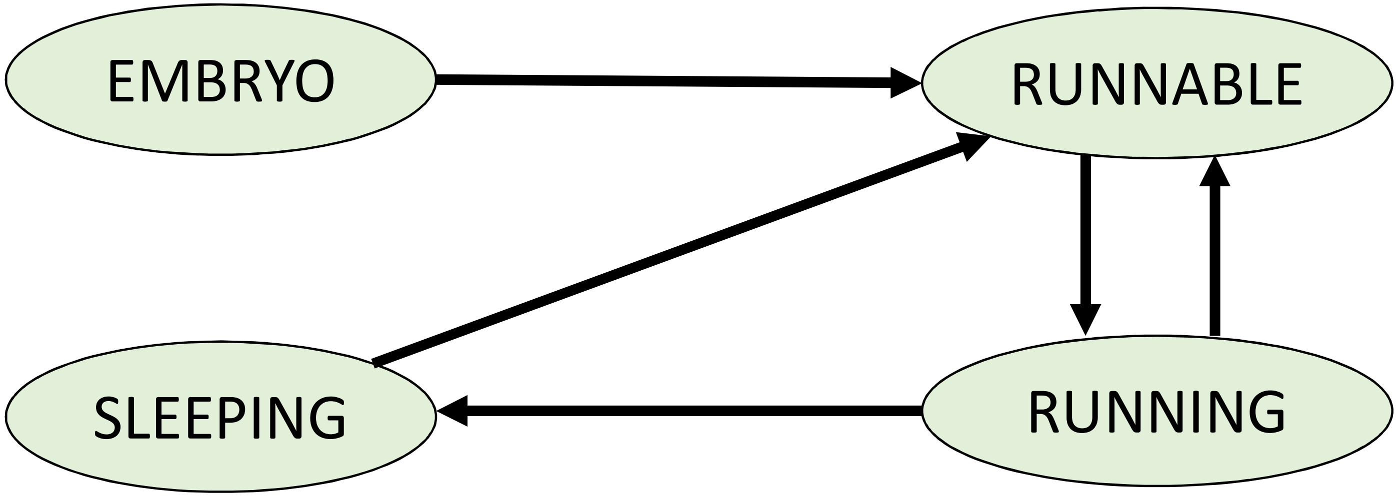
http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Process identifier (PID)

- **Process identifier (PID)**
  - Number incremented sequentially
  - Reset and continue to increment when maximum is reached
  - This time skip already allocated PID numbers

Process state

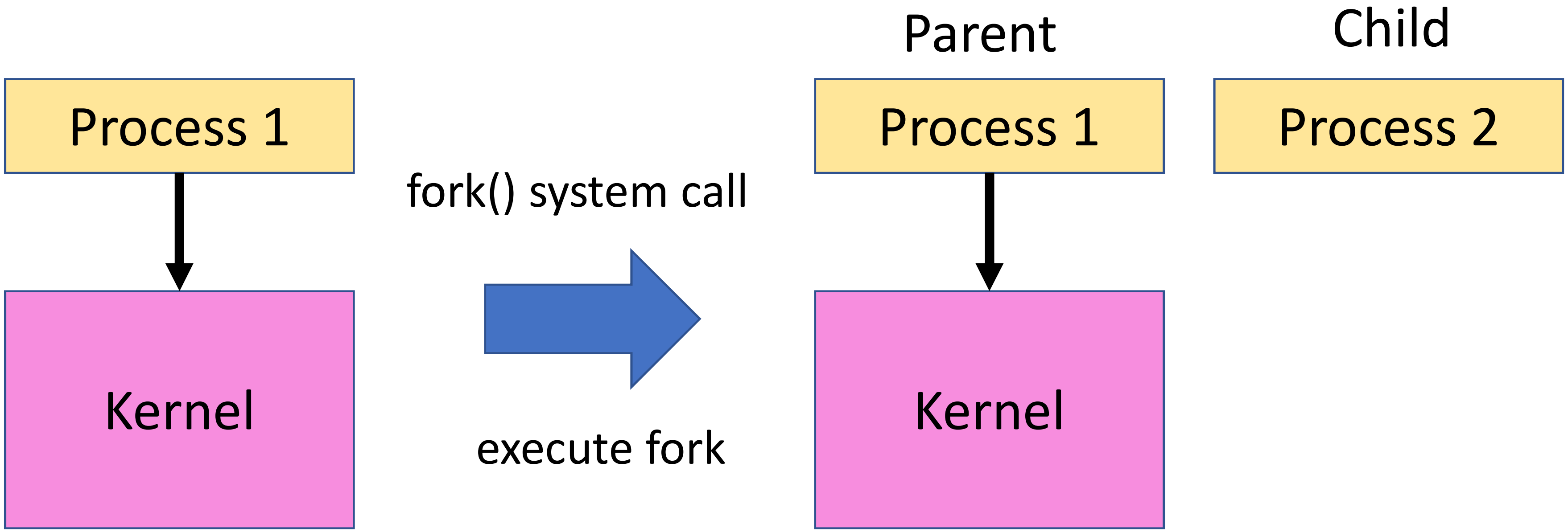
- **Process state**: specifies the state of the process



1. **EMBRYO**: The new process is currently being created
2. **RUNNABLE**: Ready to run
3. **RUNNING**: Currently executing
4. **SLEEPING**: Blocked for an I/O

Create a process by cloning

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



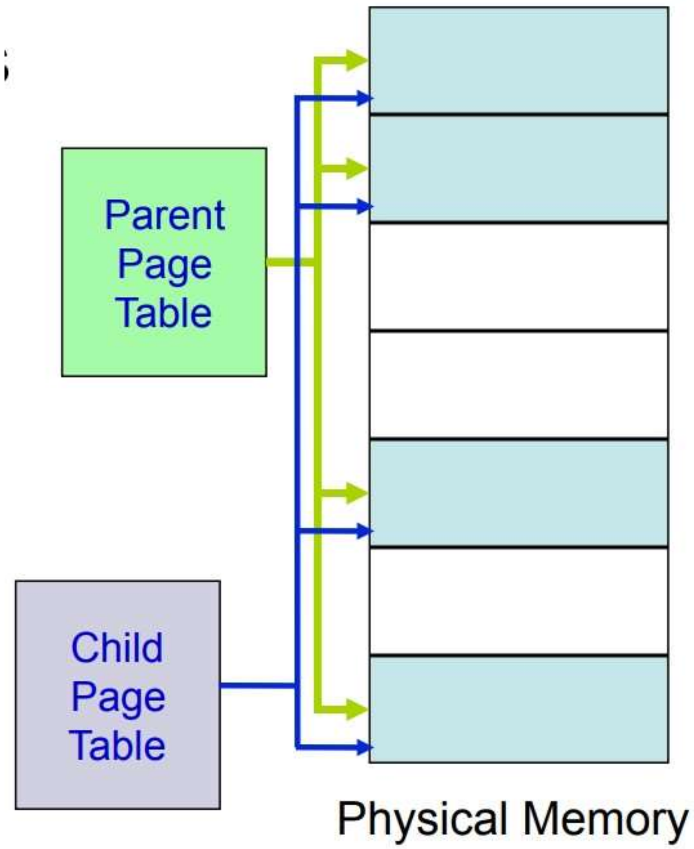
Creating a process by fork system call

- **In parent**
  - fork returns child pid
- **In child process**
  - fork return 0
- **pid = wait()**
  - Return pid of an exiting child

```
int pid;

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

How to make a copy of a process in memory ?

- Making a copy of a process is calling **forking**
  - Parent (is the original)
  - Child (is the new process)
  - Child is an exact copy of parent
- **When fork is invoked**
  - **All pages are shared between parent and child**
  - Easily done by copying the parent's page table



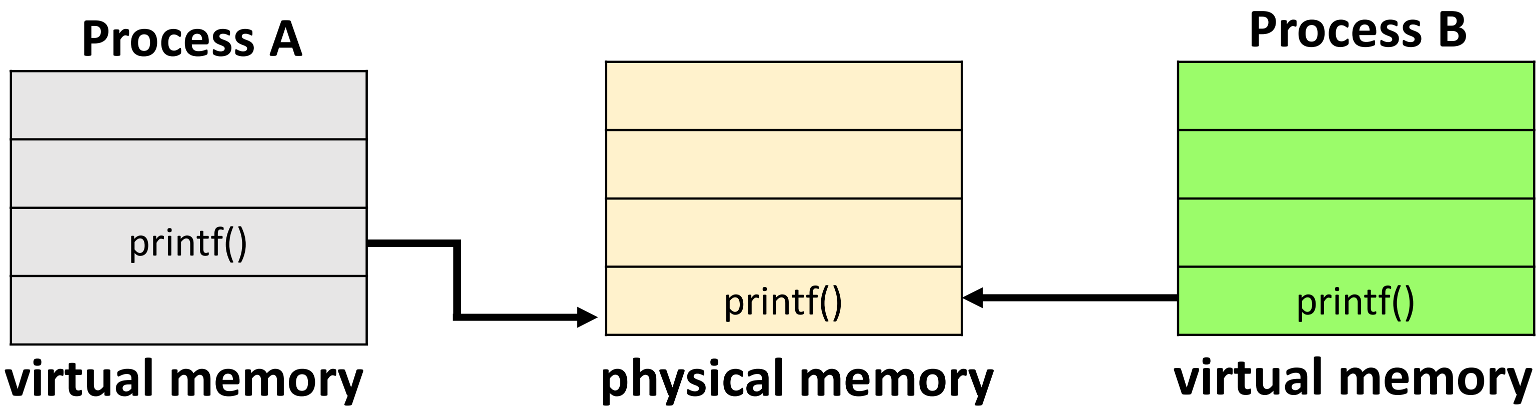
http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

How to reduce the process cloning overhead ?

- **Copy-on-write (COW)**
  - Common code (for example shared libraries) would continue to be shared
  - When data in any of the shared pages changed, OS intercepts and makes a copy of the page
  - Thus, parent and child will have different copies of this page
- **Why does COW work ?**
  - Copying each page from parent and child would incur significant disk swapping -> huge performance penalties
  - Postpone coping of pages as much as possible

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
- Pages from shared libraries, shared between processes
  - printf() implements in shared libraries

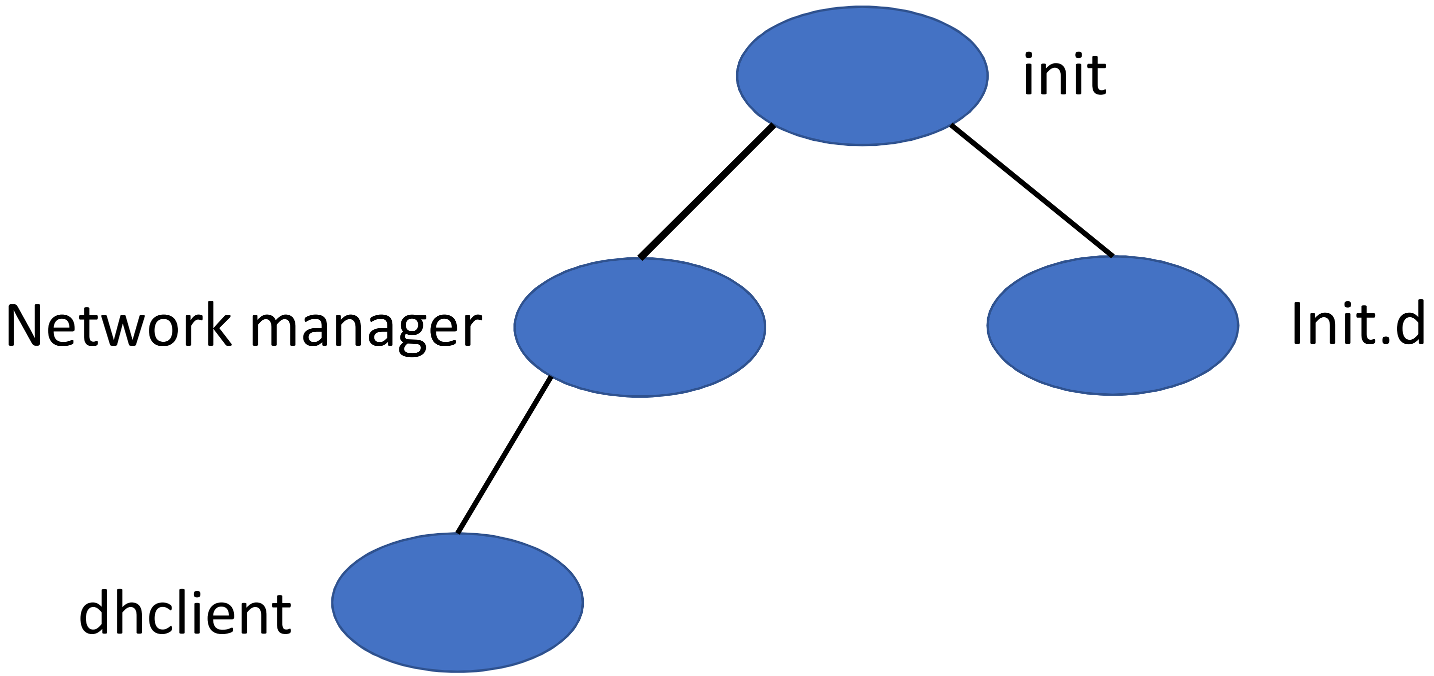


The first process

- 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
- Who create the first process ?
  - In Linux, **start_kernel()** first calls sched_init() to create first user space process init

Process tree

- Processes in the system arranged in the form of a tree
- pstree in Linux



Process termination

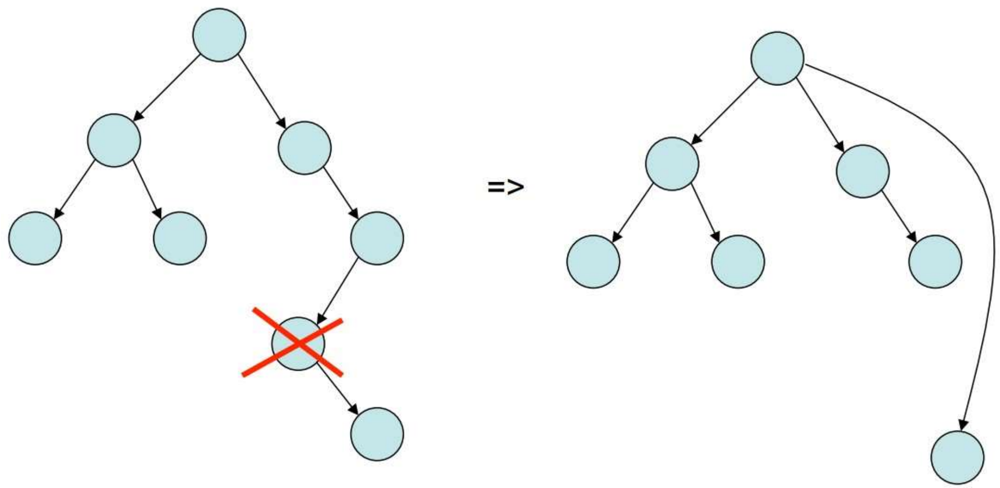
- Voluntary: exit(status)
  - OS passes exit status to parent via wait(&status)
  - OS frees process resources
- Involuntary: kill(pid, signal)
  - Signal can be sent by another process or by OS
  - pid is for the process to be killed
  - Signal enforces the process to be killed in different ways
    - E.g. SIGTERM, SIGQUIT(ctrl+\), SIGINT(ctrl+c), SIGHUP

Zombies

- What is a **zombie** (defunct) process ?
  - PCB in OS still exists even though program no longer executing
- When parent process reads child's status ?
  - Parent process can read the child's exit status through wait system call
  - Zombie entries removed from OS
- When parent doesn't read status
  - Zombie will continue to exist infinitely -> a resource leak

Orphans

- When a parent process terminates before its child
- Adopted by first process (/sbin/init)
- **Unintentional orphans**
  - When parent crashes
- **Intentional orphans**
  - Process becomes detached from user session and runs in the background



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

https://github.com/mit-pdos/xv6-public.git

The first process in xv6

- Creating the first process
  - main (main.c) invokes userinit()
- **userinit**
  - Allocate a process id, kernel stack, fill in the process 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

- Find an unused proc entry in the process table
  - proc.c

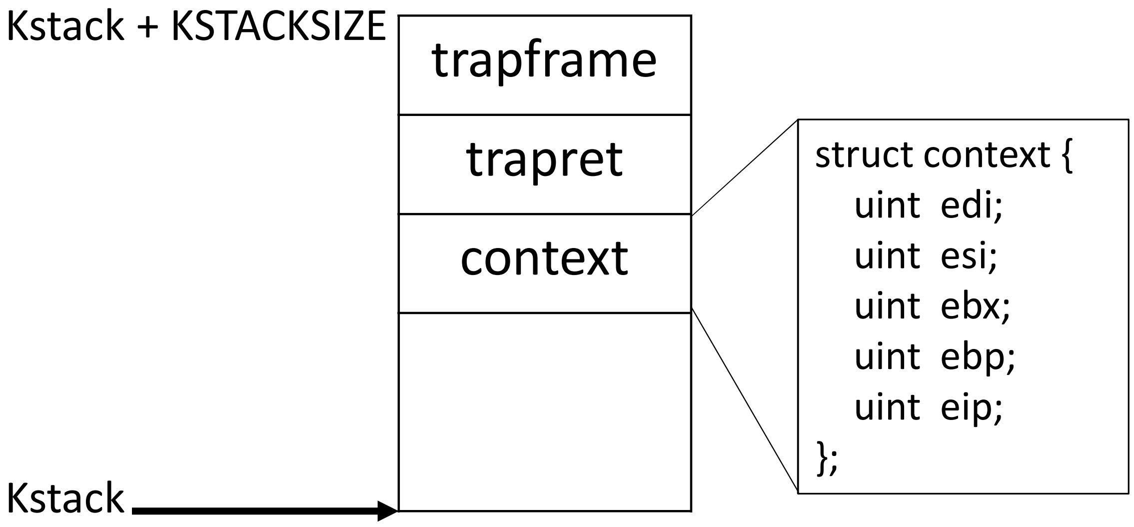
Set the state to EMBRYO (neither RUNNING nor UNUSED)

Set the pid (need to ensure that pid is unused)

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

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);
```

allocproc (cont.)

- Allocate kernel stack of size 4KB
- Allocate space on to kernel stack for
  - trapframe, trapret, context



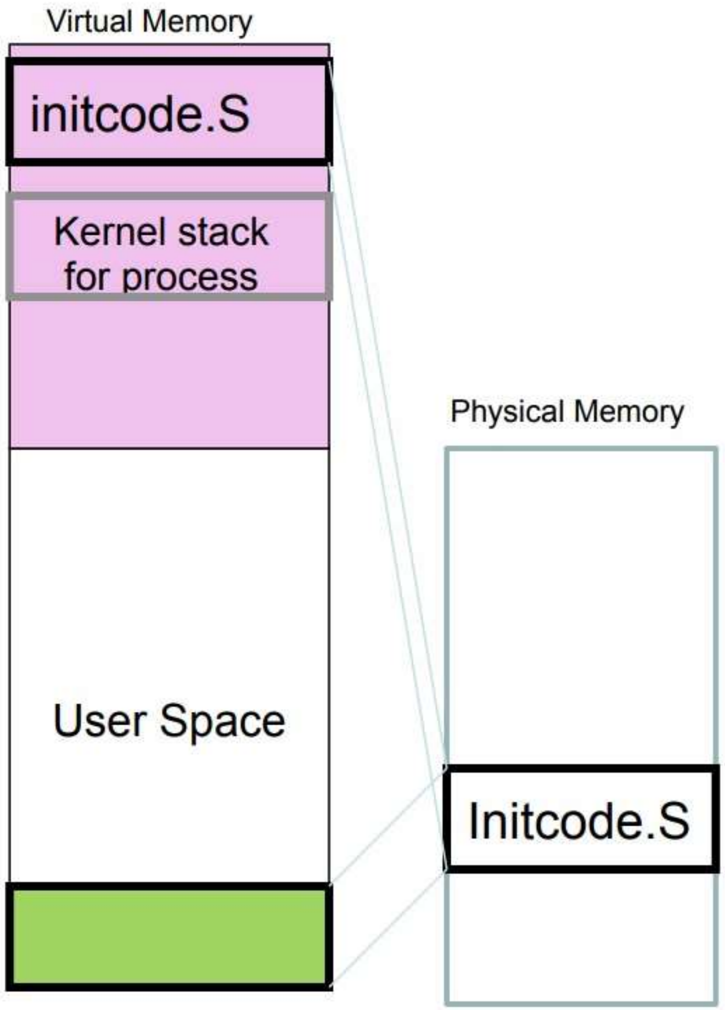
Process’s stack in kernel space

```c
// Allocate kernel stack.
if((p->kstack = kalloc()) == 0){
  p->state = UNUSED;
  return 0;
}
sp = p->kstack + KSTACKSIZE;

// Leave room for trap frame.
sp -= sizeof *p->tf;
p->tf = (struct trapframe*)sp;

// Set up new context to start executing at forkret,
// which returns to trapret.
sp -= 4;
*(uint*)sp = (uint)trapret;

sp -= sizeof *p->context;
p->context = (struct context*)sp;
memset(p->context, 0, sizeof *p->context);
p->context->eip = (uint)forkret;

return p;
```

Setup pagetables

- Kernel page tables
  - Invoked by setupkvm (vm.c)
- User page tables
  - Setup in inituvm (vm.c)

```
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);
}
```



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

Userinit (cont.)

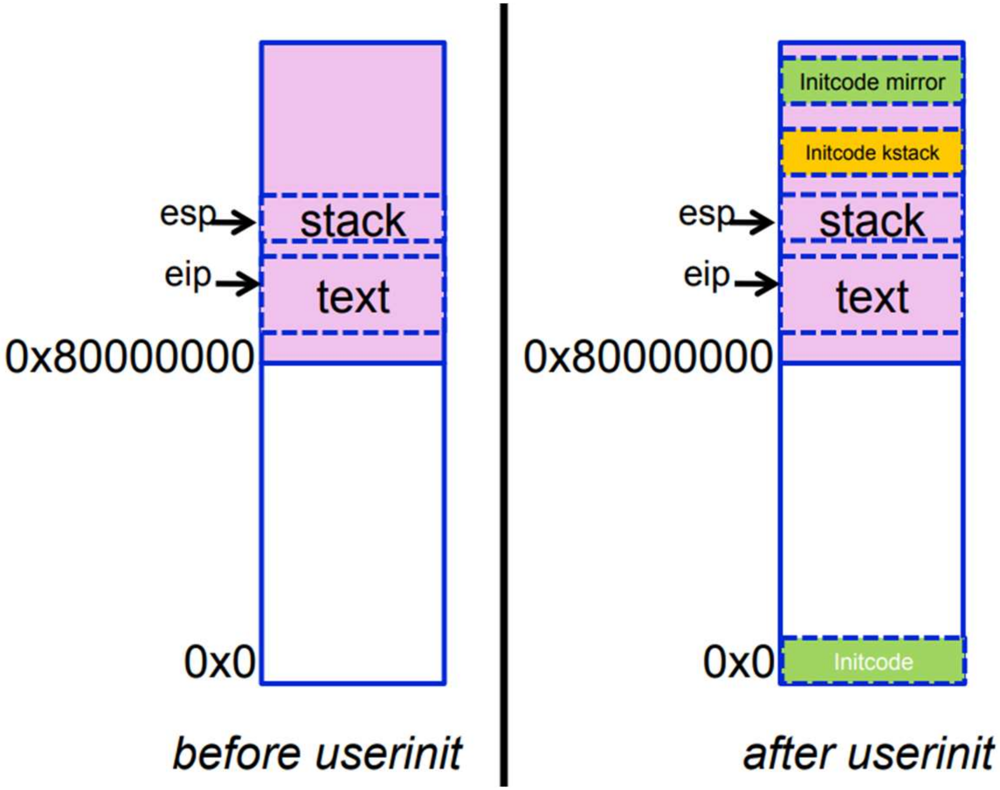
- userinit() (proc.c)
  - Fill the trapframe

```c
struct proc *p;
extern char _binary_initcode_start[], _binary_initcode_size[];

p = allocproc();

initproc = p;
if((p->pgdir = setupkvm()) == 0)
  panic("userinit: out of memory?");
inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
p->sz = PGSIZE;
memset(p->tf, 0, sizeof(*p->tf));
p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
p->tf->es = p->tf->ds;
p->tf->ss = p->tf->ds;
p->tf->eflags = FL_IF;
p->tf->esp = PGSIZE;
p->tf->eip = 0;  // beginning of initcode.S

safestrcpy(p->name, "initcode", sizeof(p->name));
p->cwd = namei("/");

// this assignment to p->state lets other cores
// run this process. the acquire forces the above
// writes to be visible, and the lock is also needed
// because the assignment might not be atomic.
acquire(&ptable.lock);

p->state = RUNNABLE;
```

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

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
  - The scheduler() gets initcode (in user space) to execute



Finally … initcode.S

- Invokes system call exec to invoke /init
  - Exec('/init')

```
# 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

- forks and creates a shell (sh)

```
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");
  }
```

Summary

- A process is different from the program
- Each process has its own address space
- Process kernel stack and user space stack
- Process control block (PCB) records the information for each process
- Creating the first process