Operating Systems

Introduction to Lab 5

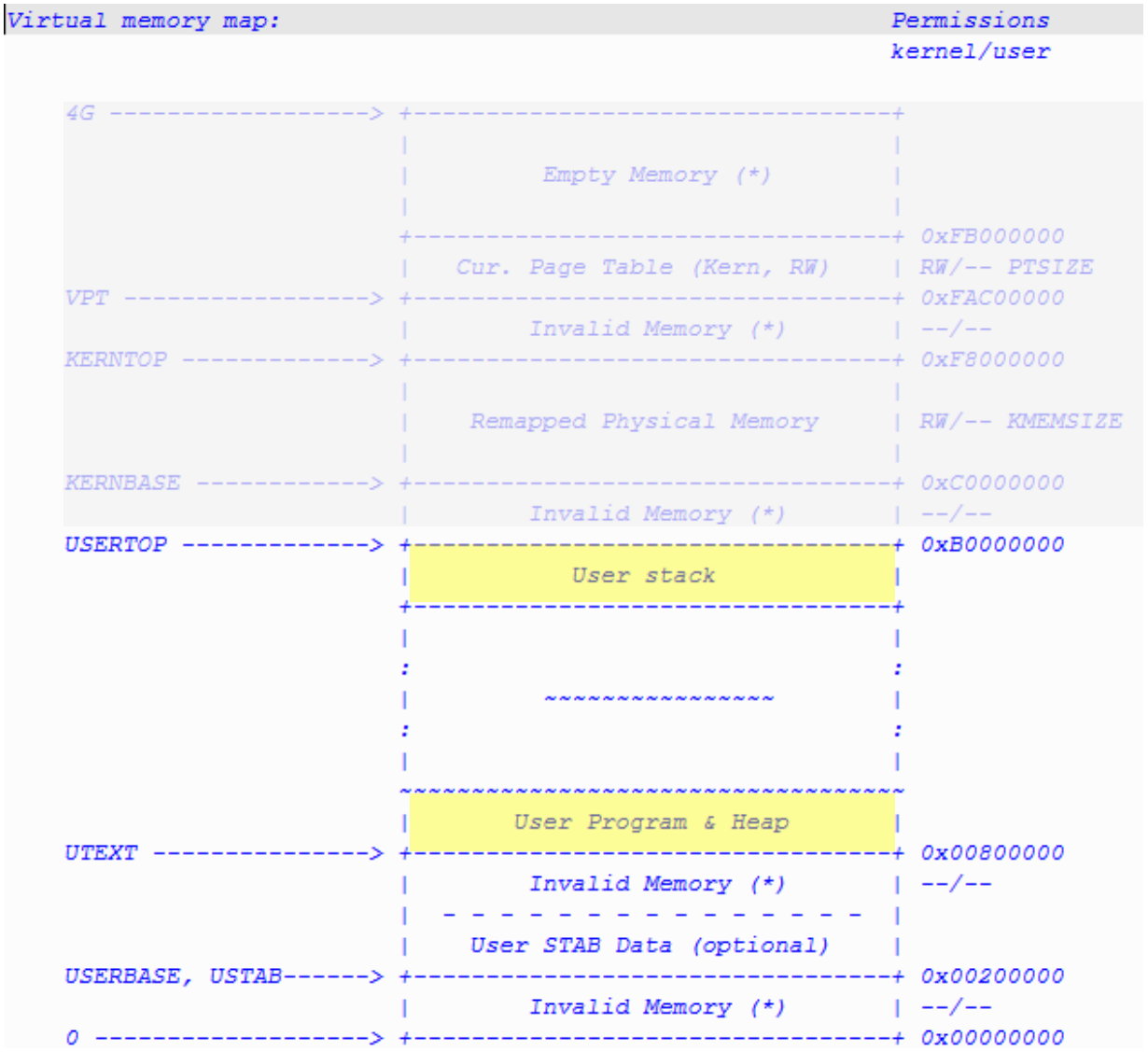
Department of Computer Science & Technology
Tsinghua University

Outline

- Memory layout of processes
- Execute an ELF binary in userspace
- Process initialization in uCore
- Process duplication
- Copy-on-write memory management

MEMORY LAYOUT OF USER PROCESSES

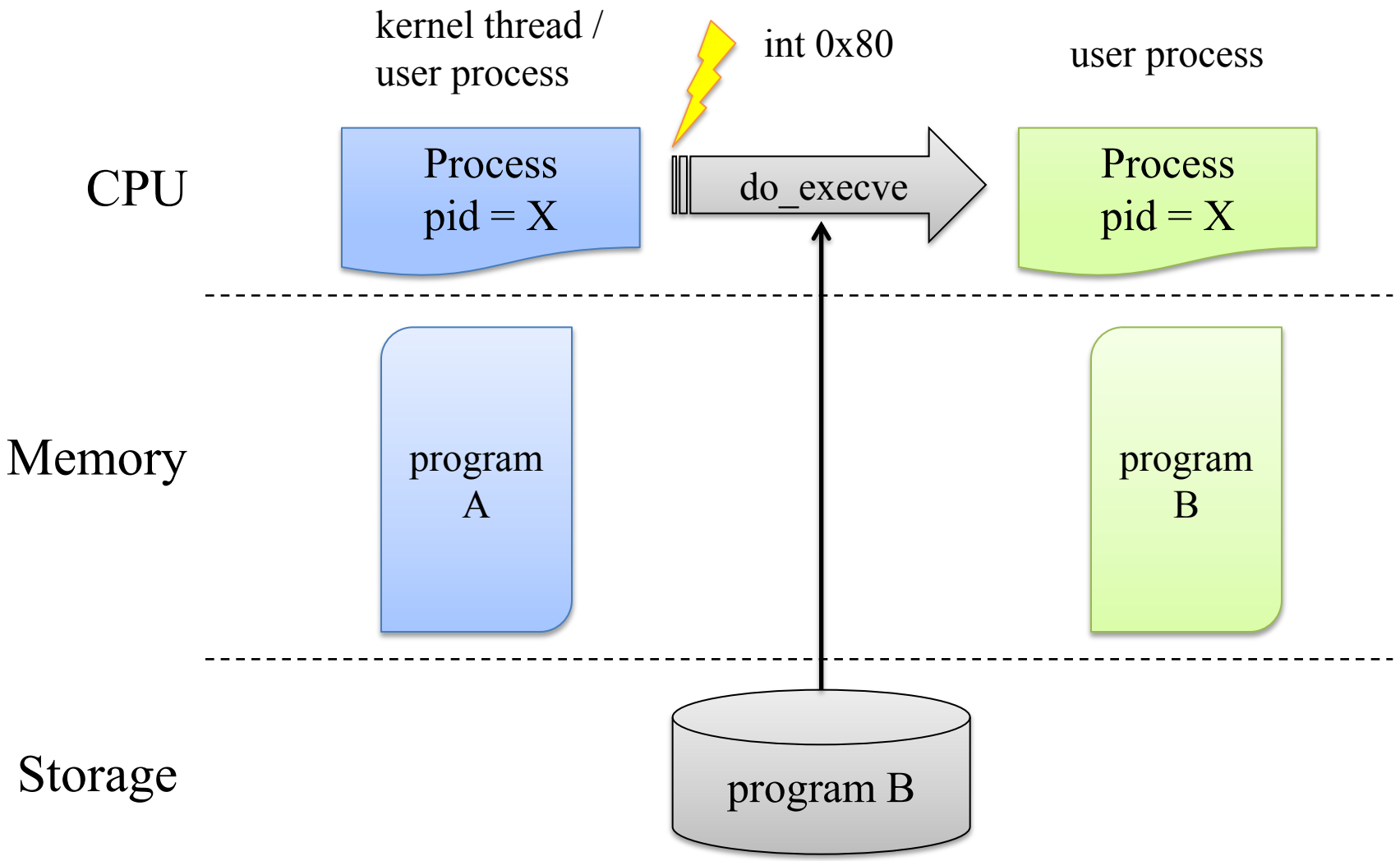
Memory layout of user processes



- Understand the steps to load & run an ELF binary in userspace

EXECUTE AN ELF BINARY IN USERSPACE

Execute an ELF binary in userspace – overview



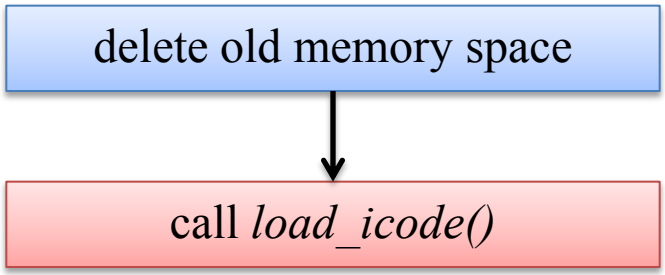
Execute an ELF binary in userspace – Steps (do_execve)

delete old memory space

proc.c : 634 - 642

```
if (mm != NULL) {
    lcr3(boot_cr3);
    if (mm_count_dec(mm) == 0) {
        exit_mmap(mm);
        put_pgdir(mm);
        mm_destroy(mm);
    }
    current->mm = NULL;
}
```

Execute an ELF binary in userspace – Steps (do_execve)

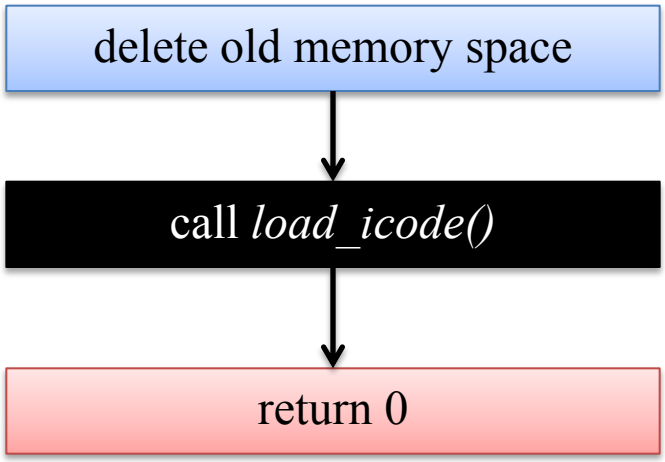


proc.c : 634 - 642

proc.c : 644

```
if ((ret = load_icode(binary, size)) != 0) {
    goto execve_exit;
}
```

Execute an ELF binary in userspace – Steps (do_execve)



proc.c : 634 - 642

proc.c : 644

proc.c : 648

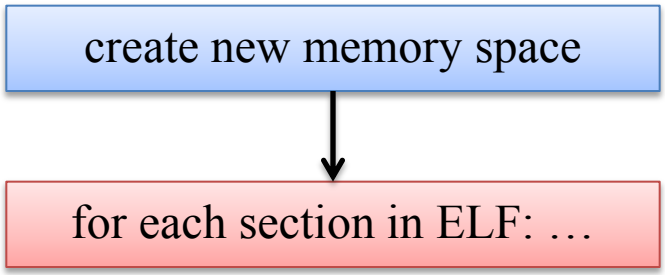
Execute an ELF binary in userspace – Steps (load_icode)

create new memory space

proc.c : 487-493

```
if ((mm = mm_create()) == NULL) {
    goto bad_mm;
}
if (setup_pgdir(mm) != 0) {
    goto bad_pgdir_cleanup_mm;
}
```

Execute an ELF binary in userspace – Steps (load_icode)



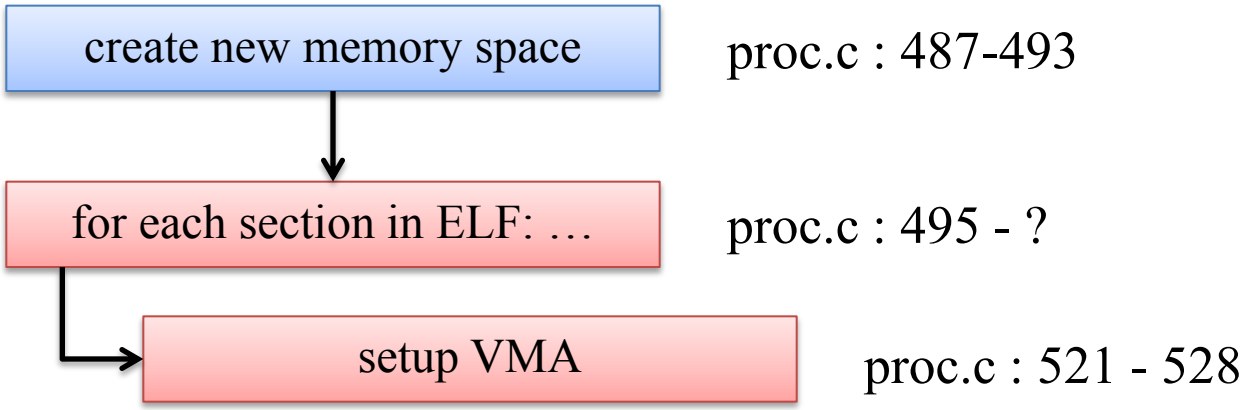
proc.c : 487-493

proc.c : 495 - ?

```
struct elfhdr *elf = (struct elfhdr *)binary;
struct proghdr *ph =
    (struct proghdr *)(binary + elf->e_phoff);
if (elf->e_magic != ELF_MAGIC) {
    ret = -E_INVAL_ELF;
    goto bad_elf_cleanup_pgdir;
}

uint32_t vm_flags, perm;
struct proghdr *ph_end = ph + elf->e_phnum;
for (; ph < ph_end; ph ++) {
    ……
}
```

Execute an ELF binary in userspace – Steps (load_icode)



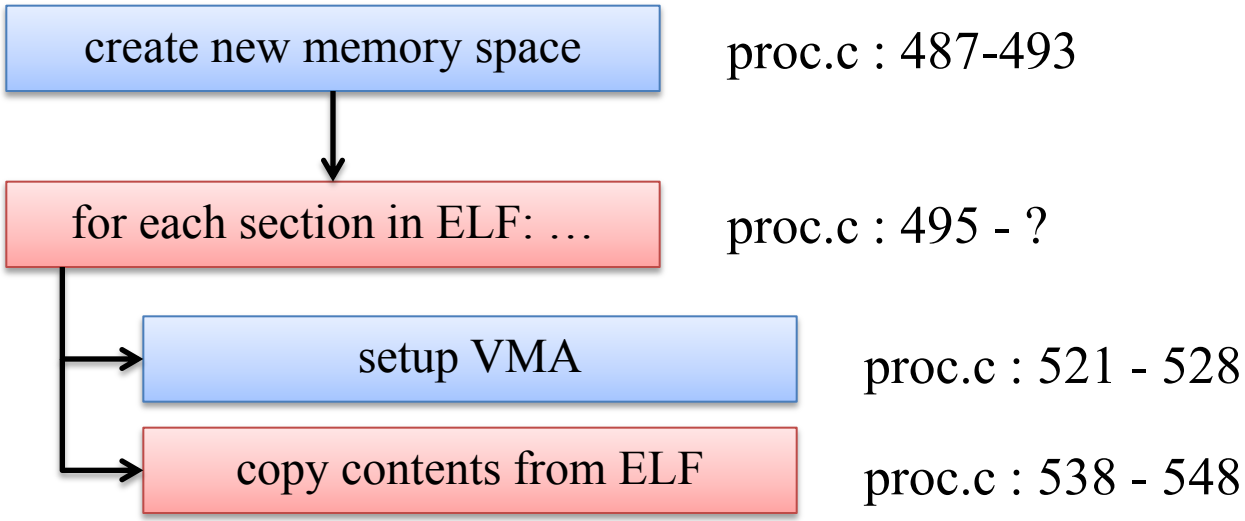
proc.c : 487-493

proc.c : 495 - ?

proc.c : 521 - 528

```
vm_flags = 0, perm = PTE_U;
if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
if (vm_flags & VM_WRITE) perm |= PTE_W;
if ((ret = mm_map(mm, ph->p_va, ph->p_memsz,
vm_flags, NULL)) != 0) {
    goto bad_cleanup_mmap;
}
```

Execute an ELF binary in userspace – Steps (load_icode)



proc.c : 487-493

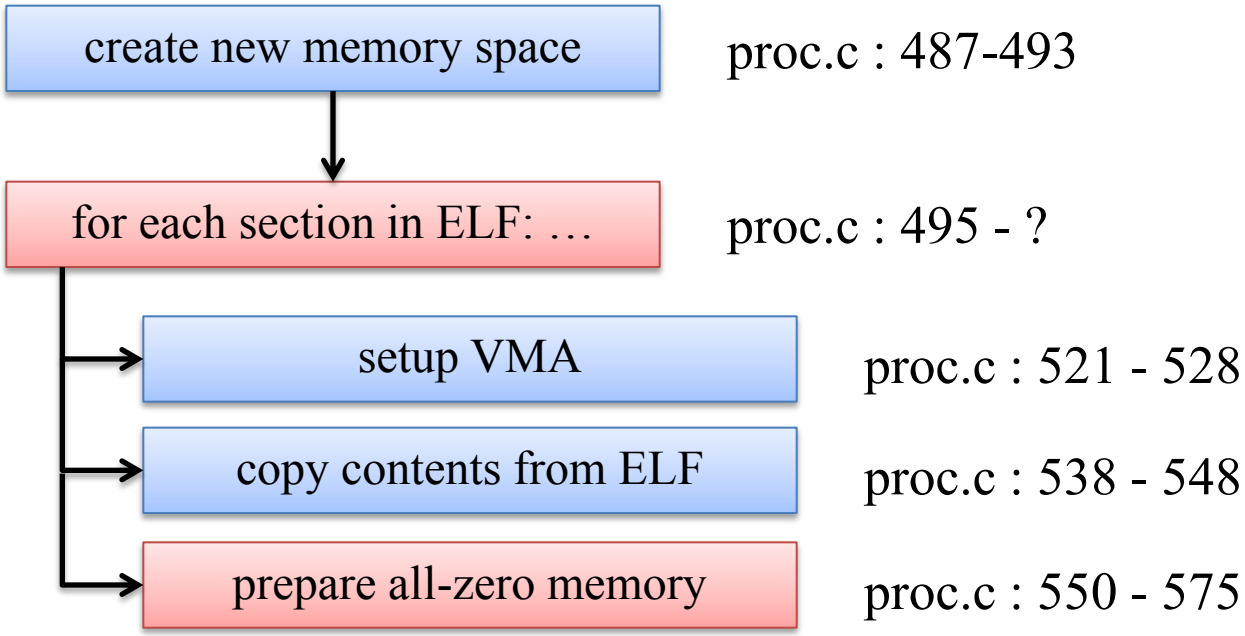
proc.c : 495 - ?

proc.c : 521 - 528

proc.c : 538 - 548

```
while (start < end) {
    if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) ==
NULL)
        goto bad_cleanup_mmap;
    off = start - la, size = PGSIZE - off, la += PGSIZE;
    if (end < la)
        size -= la - end;
    memcpy(page2kva(page) + off, from, size);
    start += size, from += size;
}
```

Execute an ELF binary in userspace – Steps (load_icode)



proc.c : 487-493

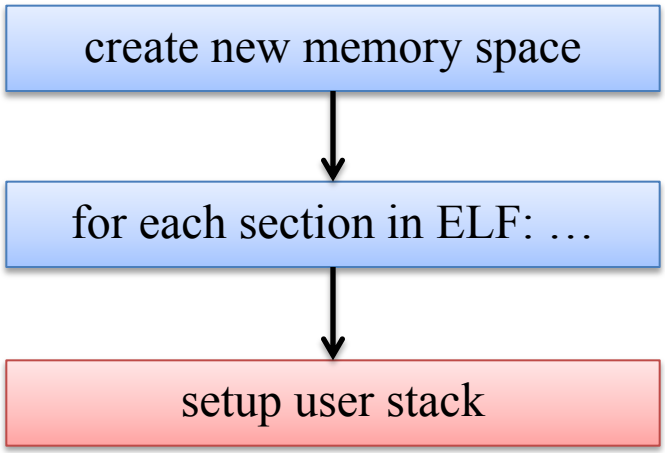
proc.c : 495 - ?

proc.c : 521 - 528

proc.c : 538 - 548

proc.c : 550 - 575

Execute an ELF binary in userspace – Steps (load_icode)



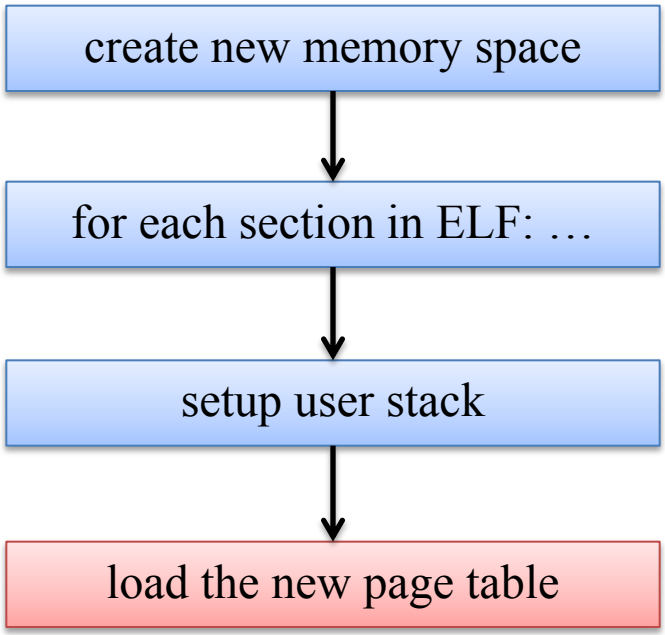
proc.c : 487-493

proc.c : 495 - 576

proc.c : 578 - 585

```
vm_flags = VM_READ | VM_WRITE | VM_STACK;
if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE,
vm_flags, NULL)) != 0) {
    goto bad_cleanup_mmap;
}
pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE, PTE_USER;
pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE, PTE_USER);
pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE, PTE_USER);
pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE, PTE_USER);
```

Execute an ELF binary in userspace – Steps (load_icode)



proc.c : 487-493

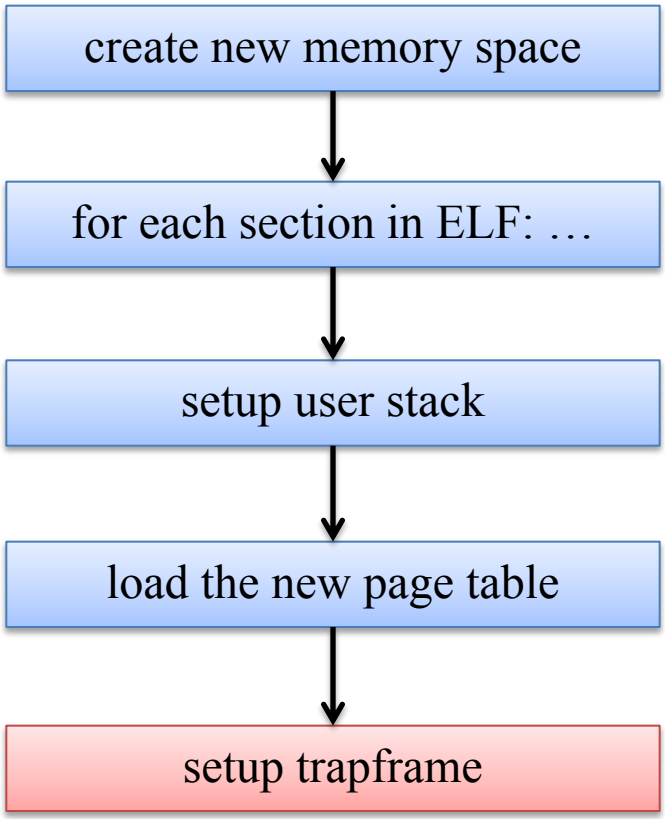
proc.c : 495 - 576

proc.c : 578 - 585

proc.c : 588 - 591

```
mm_count_inc(mm);
current->mm = mm;
current->cr3 = PADDR(mm->pgdir);
lcr3(PADDR(mm->pgdir));
```

Execute an ELF binary in userspace – Steps (load_icode)



proc.c : 487-493

proc.c : 495 - 576

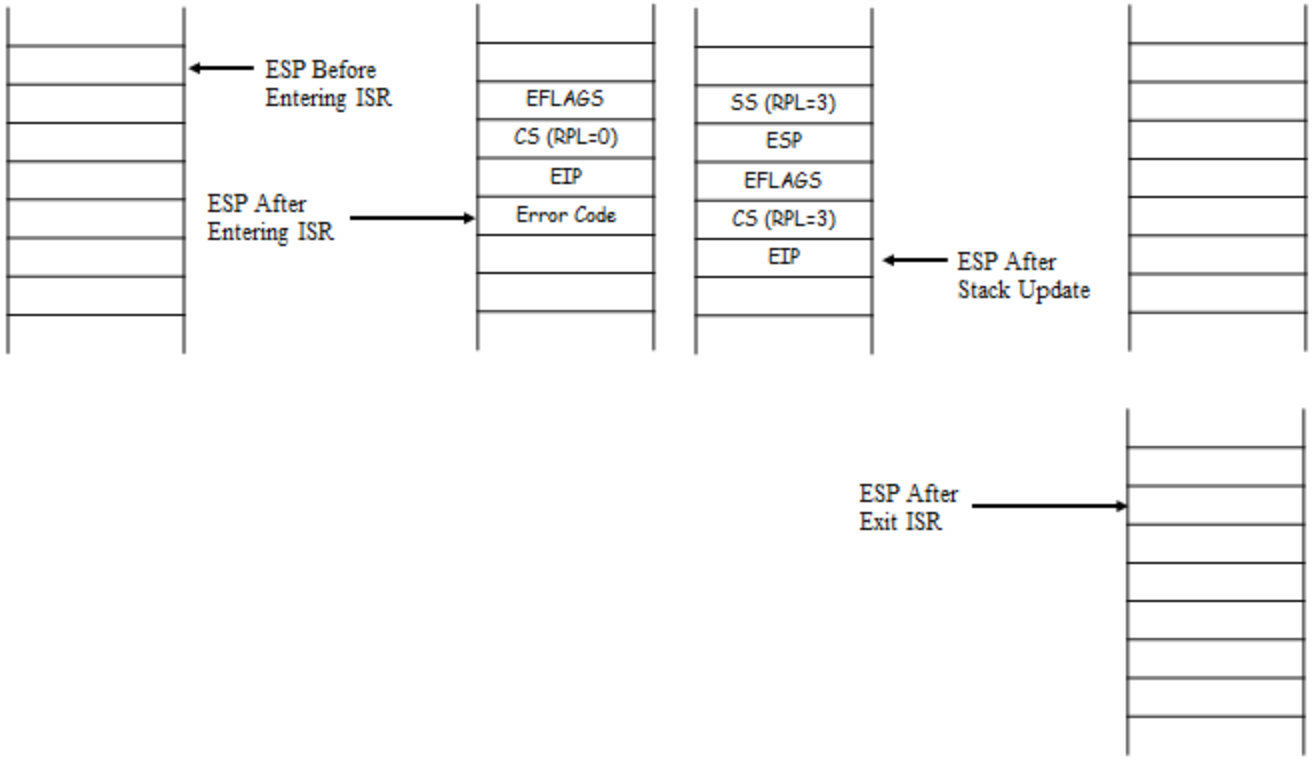
proc.c : 578 - 585

proc.c : 588 - 591

proc.c : YOUR WORK!!!

Execute an ELF binary in userspace – Steps (load_icode)

x86 privilege levels – switch privilege levels (0 to 3)

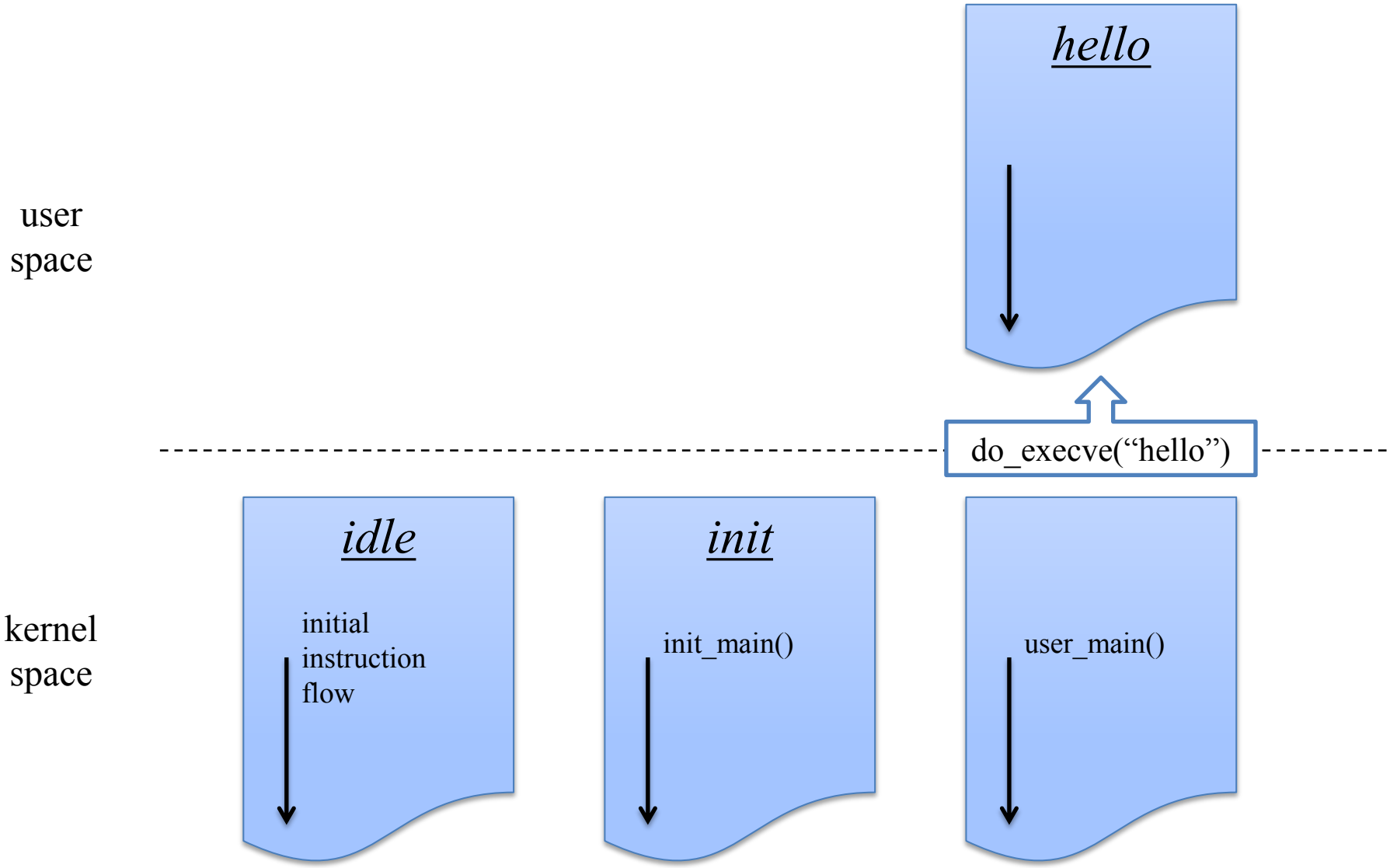


9

- Know how the ancestors of processes are created

PROCESS INITIALIZATION IN UCORE

Process initialization in uCore

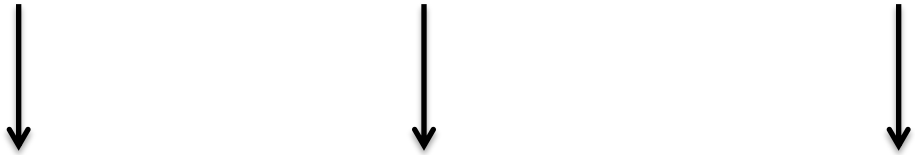


- Understand how processes can be duplicated from existing ones (i.e. forking)

PROCESS DUPLICATION

Process duplication – do_fork(): prototype

int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)



for copy_mm()

current esp position for copy_thread()

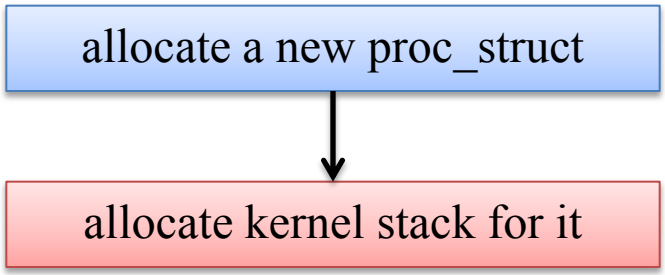
parent's trapframe for copy_thread()

Process duplication – do_fork(): steps (YOUR WORK!!!)

allocate a new proc_struct

use *alloc_proc()*
Note: may fail

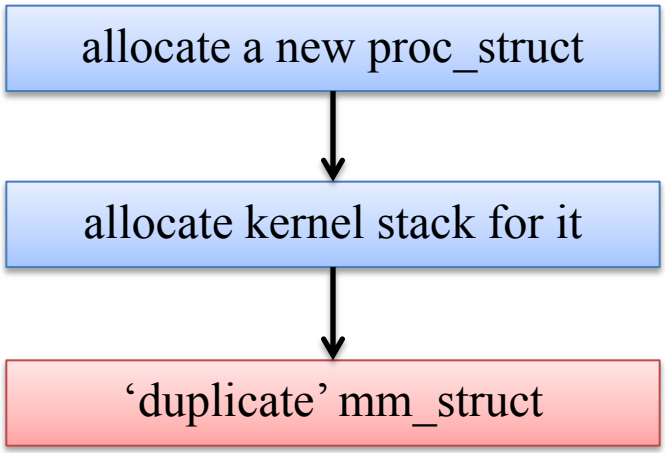
Process duplication – do_fork(): steps (YOUR WORK!!!)



use *setup_kstack()*

Note: may fail

Process duplication – do_fork(): steps (YOUR WORK!!!)



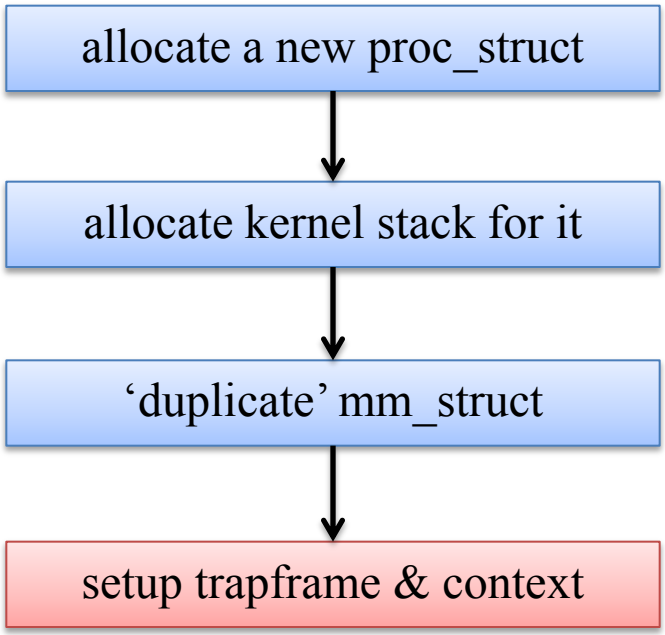
i.e. create a new virtual memory space for the newly created process

use *copy_mm()*

copy_range(): copy memory in the parent process to the new one YOUR WORK

Note: may fail

Process duplication – do_fork(): steps (YOUR WORK!!!)



copy the parent's trapframe (in the kernel stack)
to the new process

eax = 0 (return value of the system call)

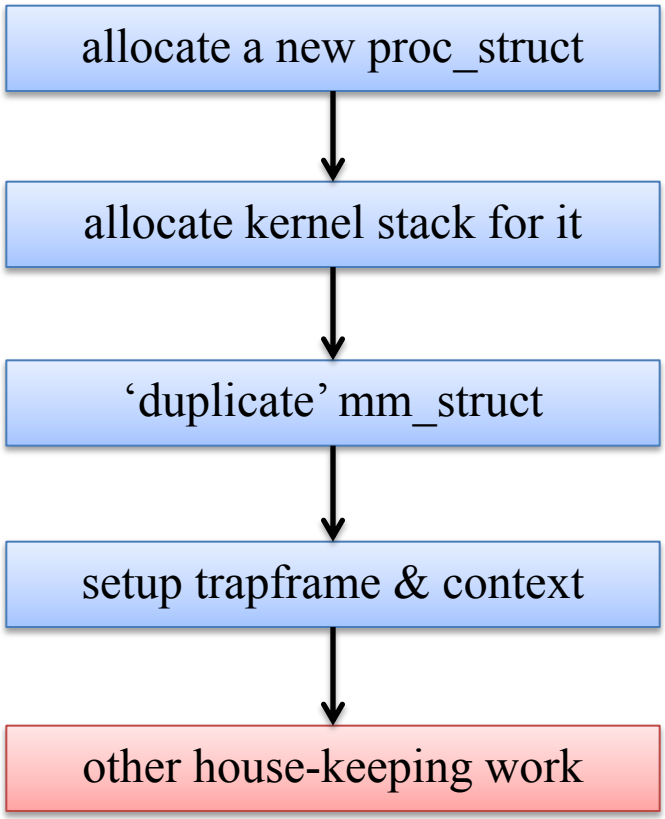
esp = (the parameter)

eip = *forkret*

copy_thread() does all those for you

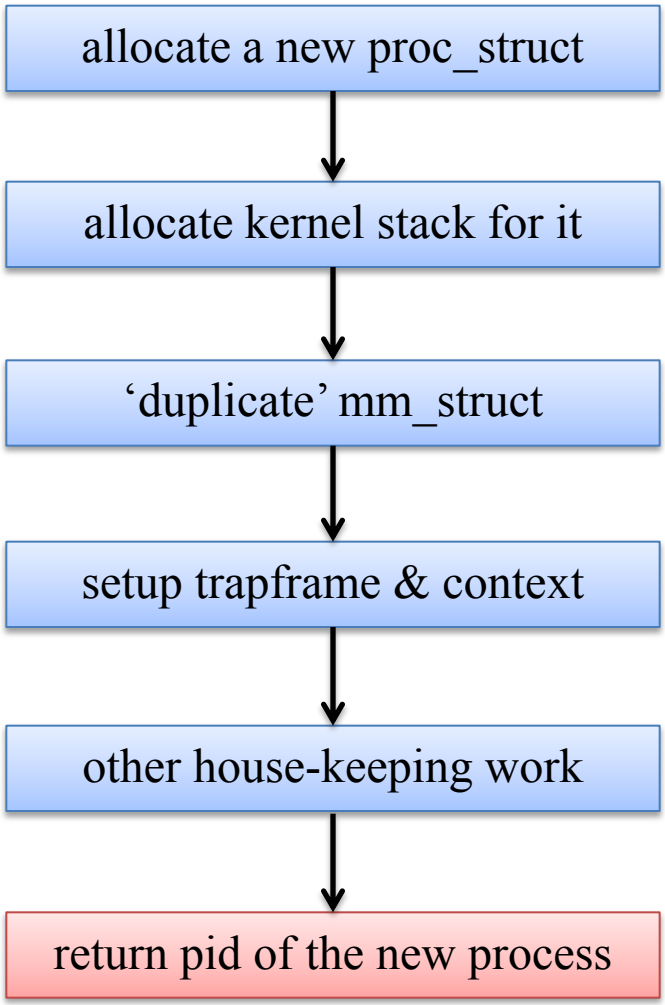
Note: won't fail

Process duplication – do_fork(): steps (YOUR WORK!!!)



Add the new proc_struct to *proc_list*
Wakeup the new process (use *wakeup_proc()*)
Note: There's a typo in comment of the source

Process duplication – do_fork(): steps (YOUR WORK!!!)

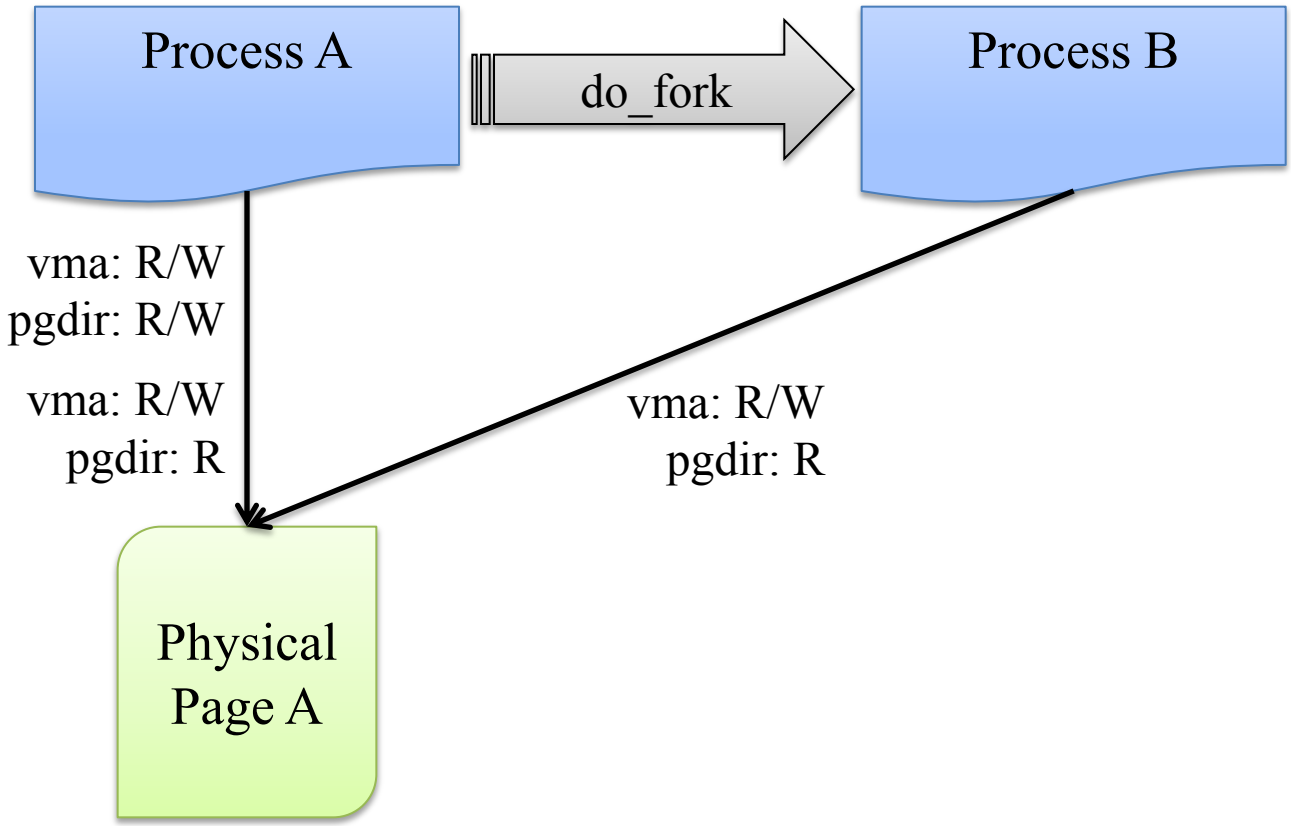


This will be the return value of the system call from the parent process

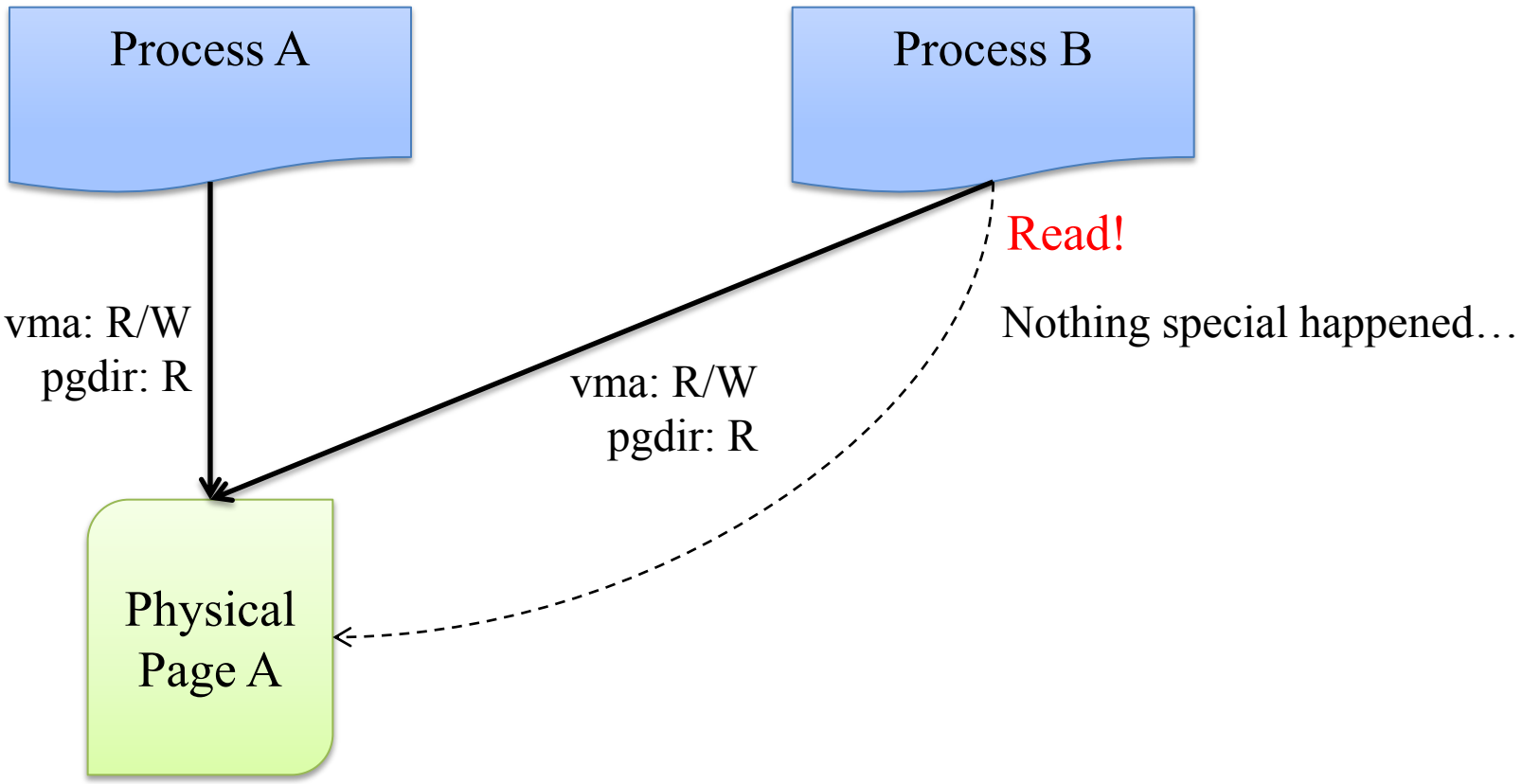
- Understand how to use COW to save memory

COPY-ON-WRITE MEMORY MANAGEMENT

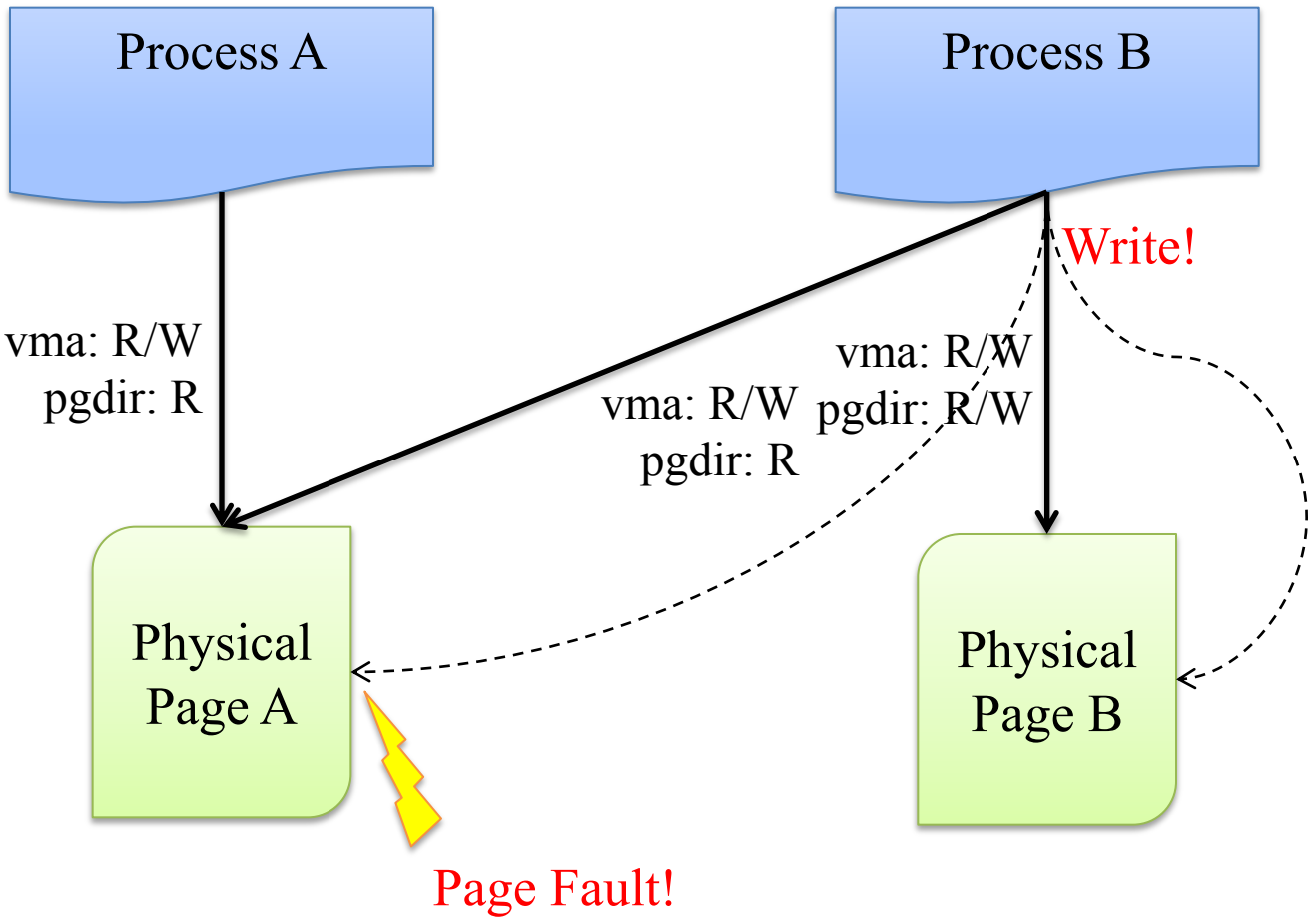
Copy-on-write memory management – What is it?



Copy-on-write memory management – What is it?

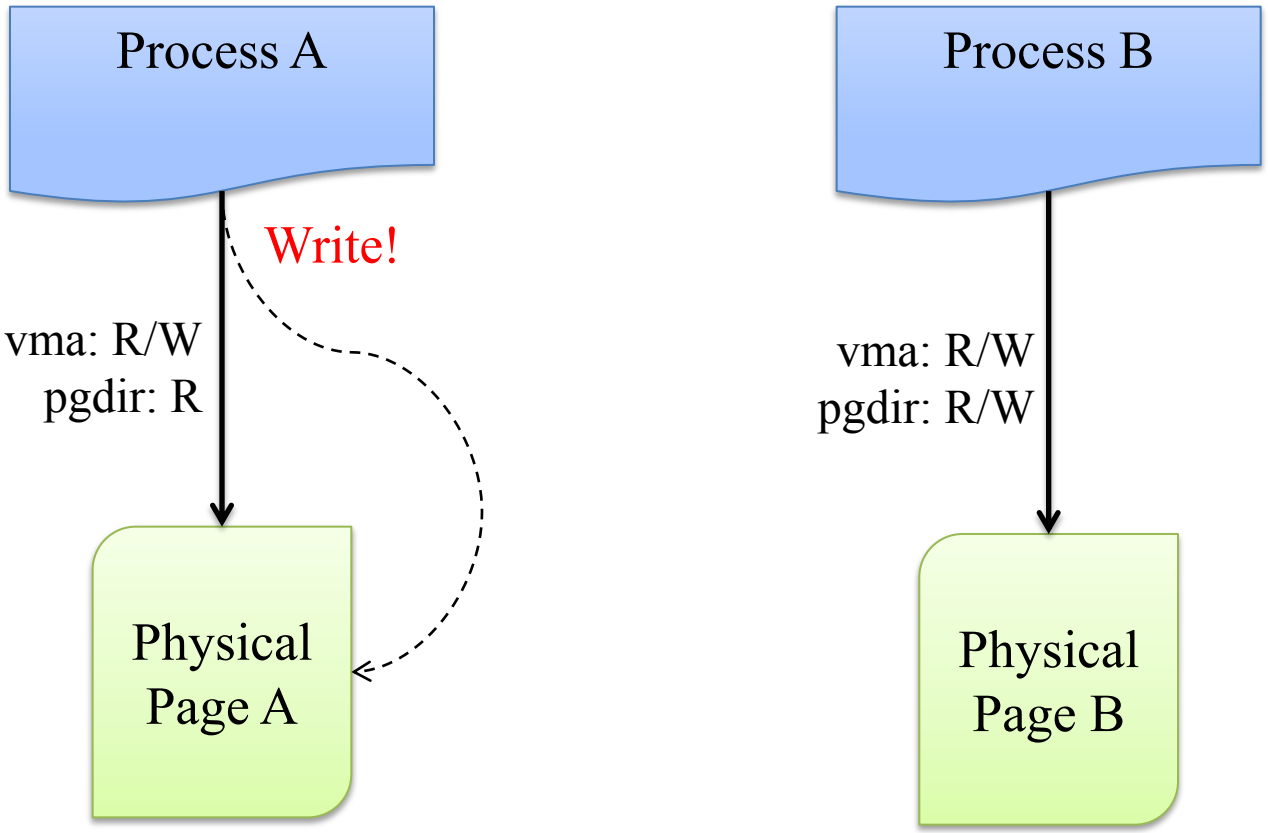


Copy-on-write memory management – What is it?



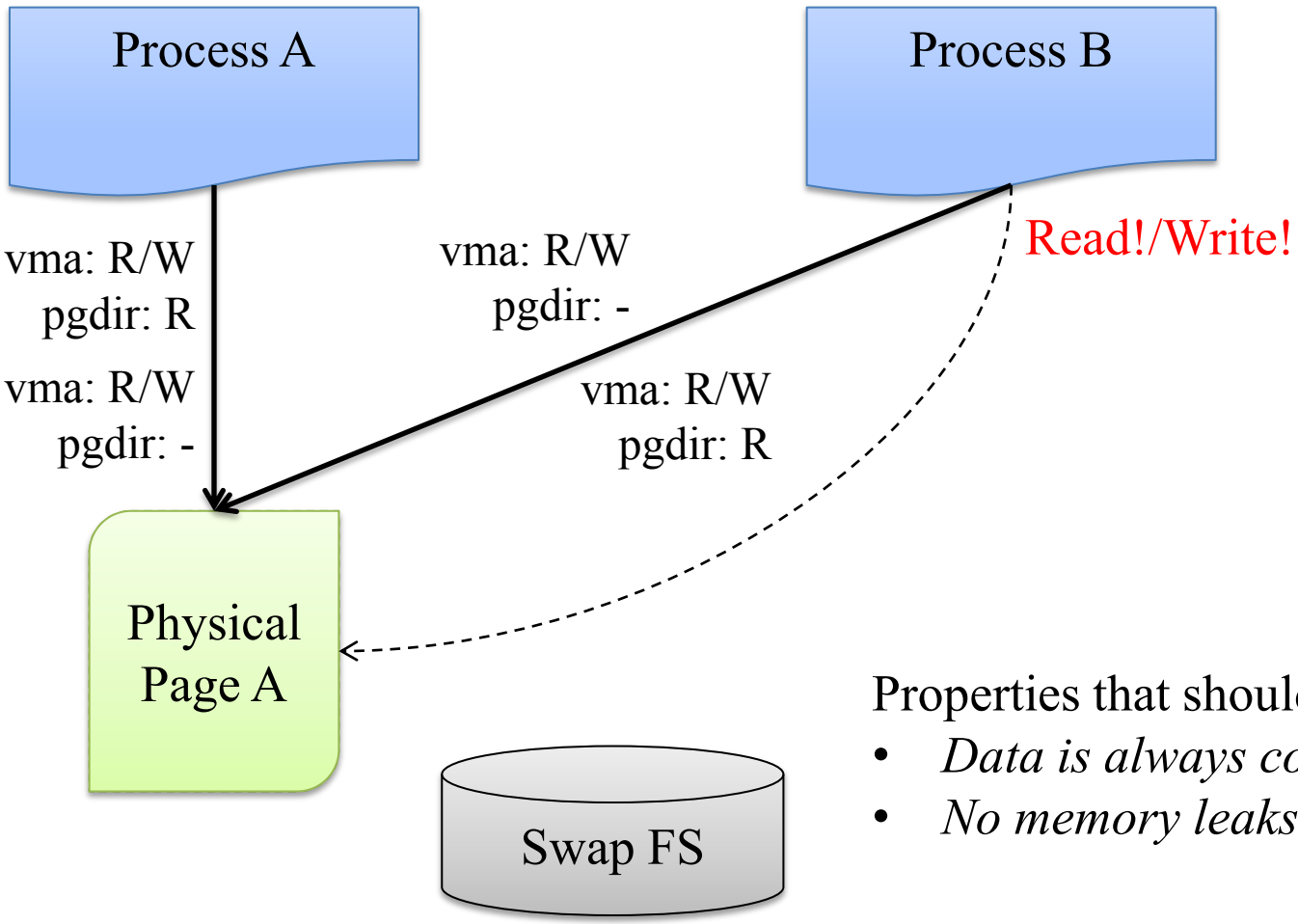
Is that all?

Copy-on-write memory management – Need more care



Need reference counting here… *page_ref()*

Copy-on-write memory management – Need more care



Properties that should hold:

- *Data is always correct*
- *No memory leaks*

Copy-on-write memory management – Steps

- *copy_range()* in pmm.c
  - *Do not copy pages when "share=true"*
- *do_pgfault()* in vmm.c
  - Detect COW case in the page fault handler
  - Handle page duplications and page table entry changes properly
- *dup_mmap()* in vmm.c
  - Change "*bool share=0*" to "*bool share=1*"

Copy-on-write memory management – Further Steps

- Take care of the corner cases properly
  - This may lack test cases. You can write some if needed.

Copy-on-write memory management – MM Summary

- MM states of a page?
  - Present? (invalid, valid, swapped-out)
  - User accessible?
  - Writable? (COW)
  - Accessed?
  - Dirty?
- Q: What is the state transition graph concerning these states? Formal proof of the model?

That's all. Thanks!