CS179F: Projects in Operating System

Lab 3: Copy-on-Write

Emiliano De Cristofaro and Lian Gao

Memory

What is memory

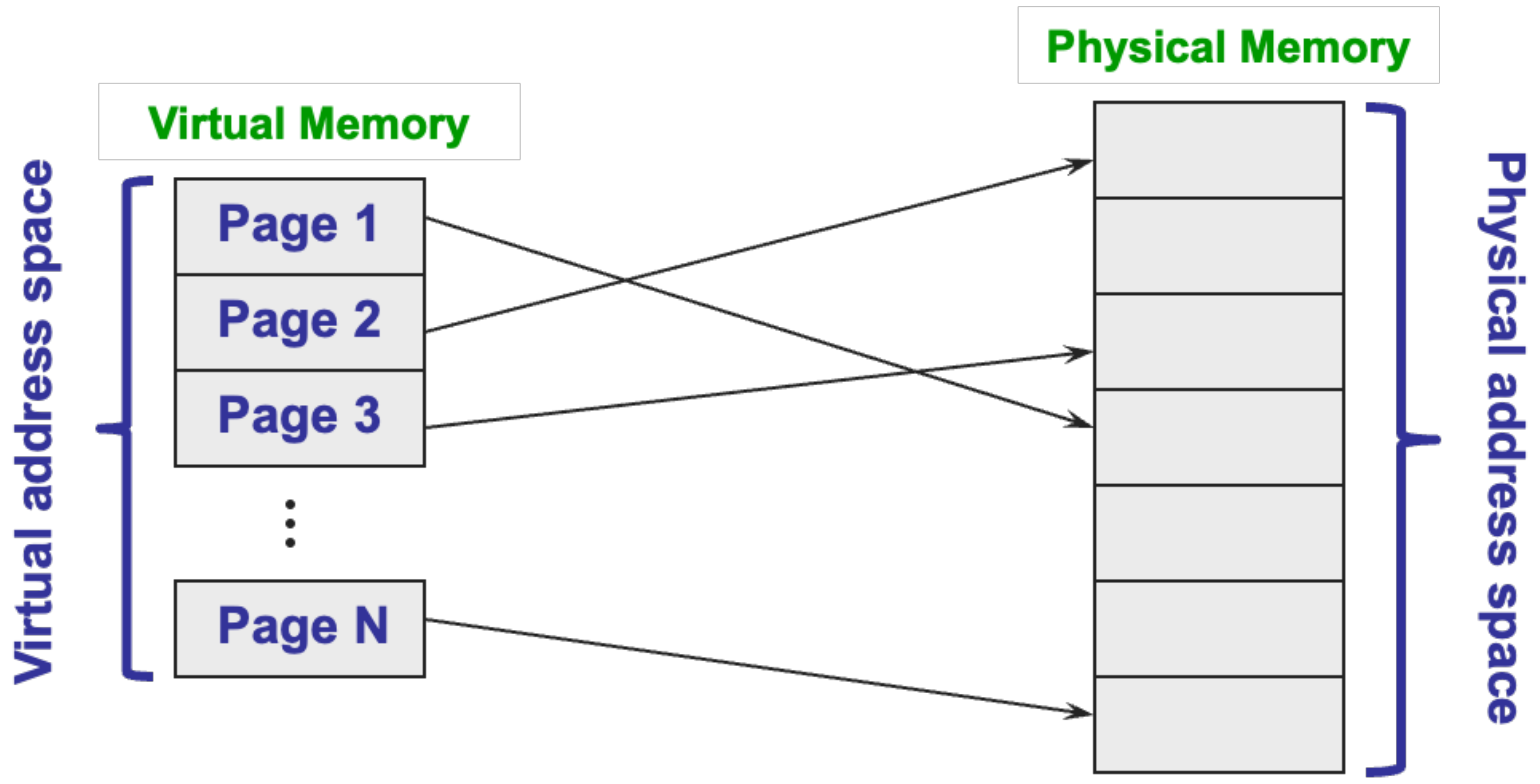
- From programmers’ perspective
  - A “place” to store data
- How to access data in memory?
  - Variables?
  - Names?
  - Addresses?
- Memory can be viewed as a big array
  - content = memory[address]

Address Spaces

- Address space: ordered set of non-negative integer addresses that can be used to access memory: {0, 1, 2, 3 … }
    - Addresses could be contiguous or segmented
- Virtual address space: set of virtual addresses
- Physical address space: set of physical addresses

Paging

- Split the virtual and physical address space into multiple fixed size partitions (i.e., pages), each virtual page can be translated to any physical page



Page Table Entry

When page fault can happen?

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)	P=0

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 2 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

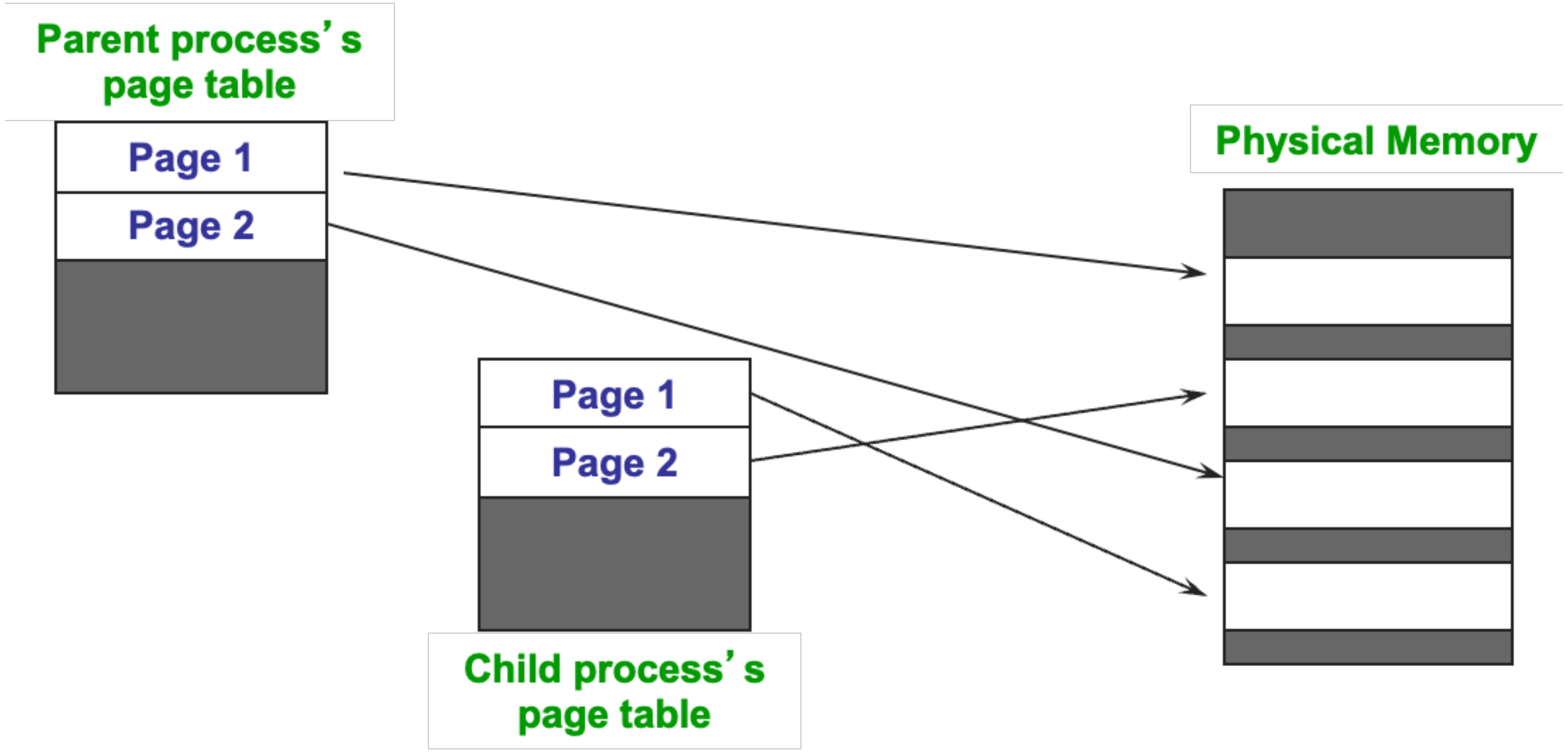
XD: Non-executable pages

Page Fault Handling

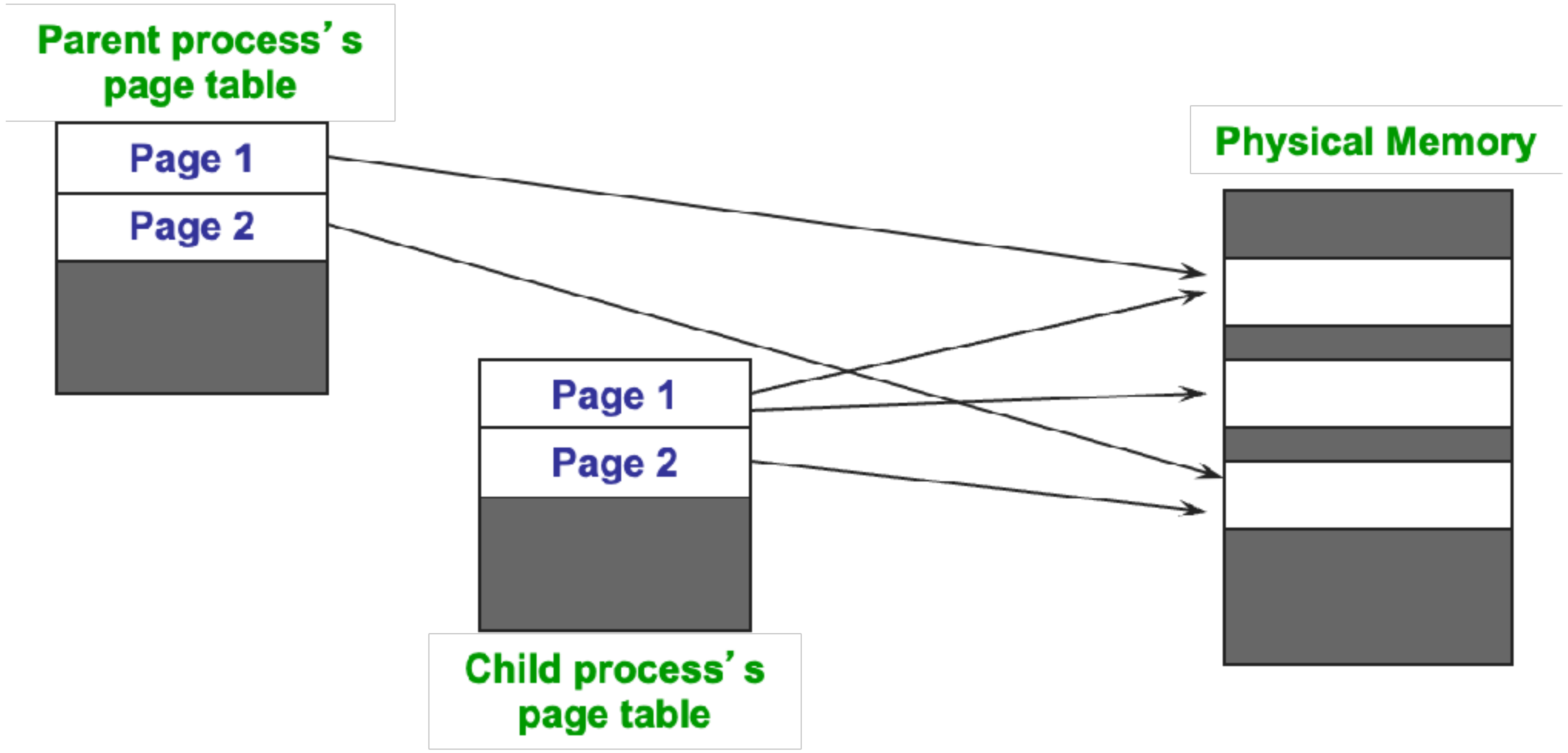
OS-induced page faults

- Lazy allocation
  - Aggressive lazy allocation
- Copy-on-Write
- Virtual memory

fork() without CoW



fork() with CoW



The problem with fork() in xv6

- fork() copies all of the parent process's user-space memory into the child…
  - If the parent is large, copying can take a long time
  - Copies often waste memory; in many cases, neither the parent nor the child modifies a page, so they could share the same physical memory?
  - The inefficiency is particularly clear if the child calls exec(), since exec() will throw away the copied pages, probably without using most of them
  - On the other hand, if both parent and child use a page, and one or both writes it, a copy is truly needed.

The solution? Copy-on-Write (CoW) fork

- COW fork() defers allocating and copying physical memory pages for the child until the copies are actually needed
  - Creates just a page table for the child, with Page Table Entries (PTEs) for user memory pointing to the parent's physical pages
  - Marks all the user PTEs in both parent and child as not writable; when either process tries to write one of these COW pages, the CPU will force a page fault
  - Page-fault handler detects this case, allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to refer to the new page, this time with the PTE marked writeable.
  - When page fault handler returns, user process will be able to write its copy of the page.

But…

- COW fork() makes freeing physical pages that implement user memory a bit ***trickier***
- A given physical page may be referred to by multiple processes' page tables, and should be freed only when the last reference disappears

Example

Original fork():

Make a copy of the page table and the physical memory for child proc;

If a page is duplicated but not modified, it's not necessary to make a new copy;

COW fork():

Defer copy operation until the first write, when maintaining two copies is actually necessary

Example

<kernel/proc.c>

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;
```

Original fork():

Make a copy of the page table and the physical memory for child proc;

If a page is duplicated but not modified, it’s not necessary to make a new copy;

COW fork():

Defer copy operation until the first write, when maintaining two copies is actually necessary

Example

Original fork():

Make a copy of the page table and the physical memory for child proc;

If a page is duplicated but not modified, it's not necessary to make a new copy;

COW fork():

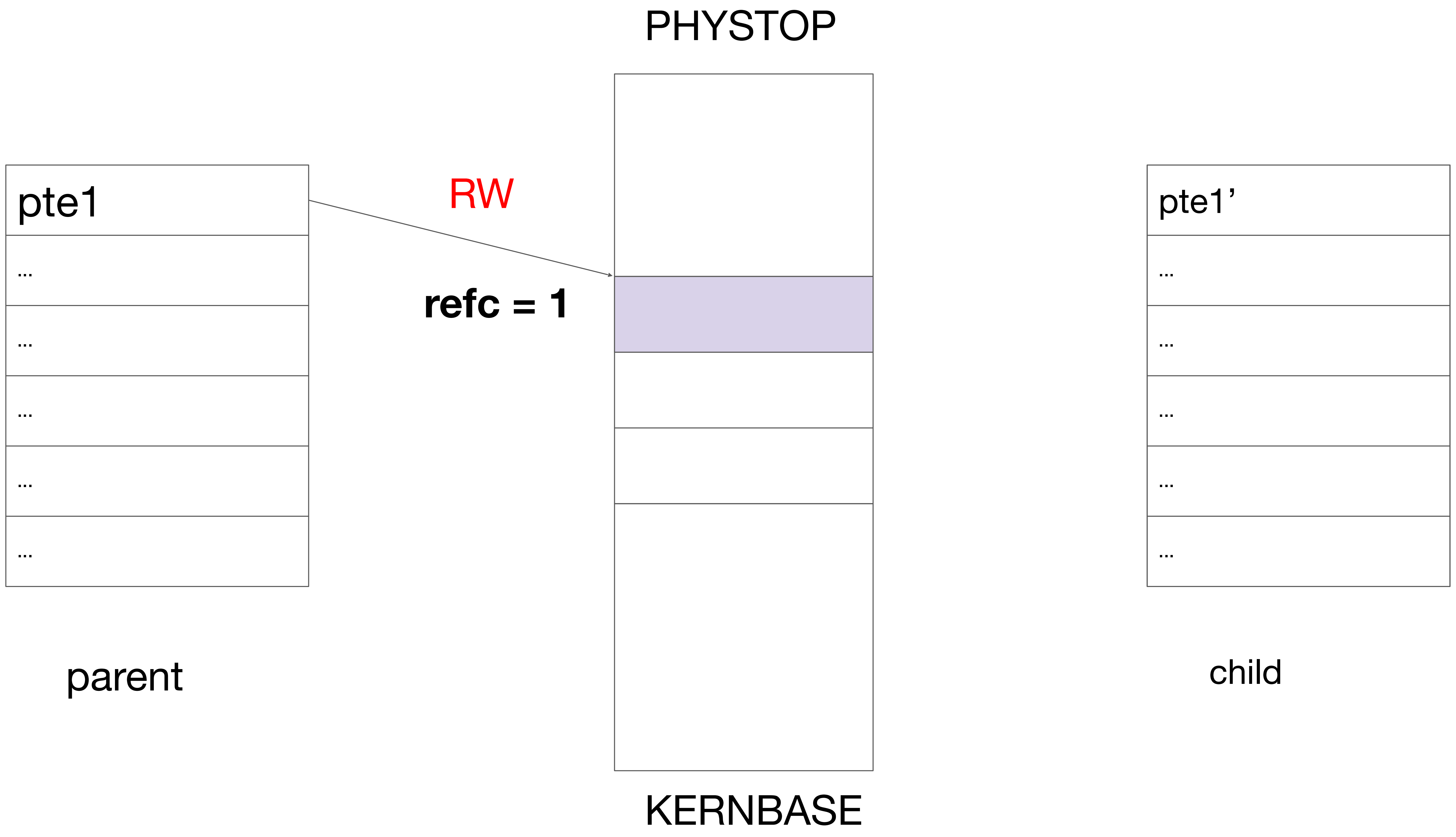
Defer copy operation until the first write, when maintaining two copies is actually necessary

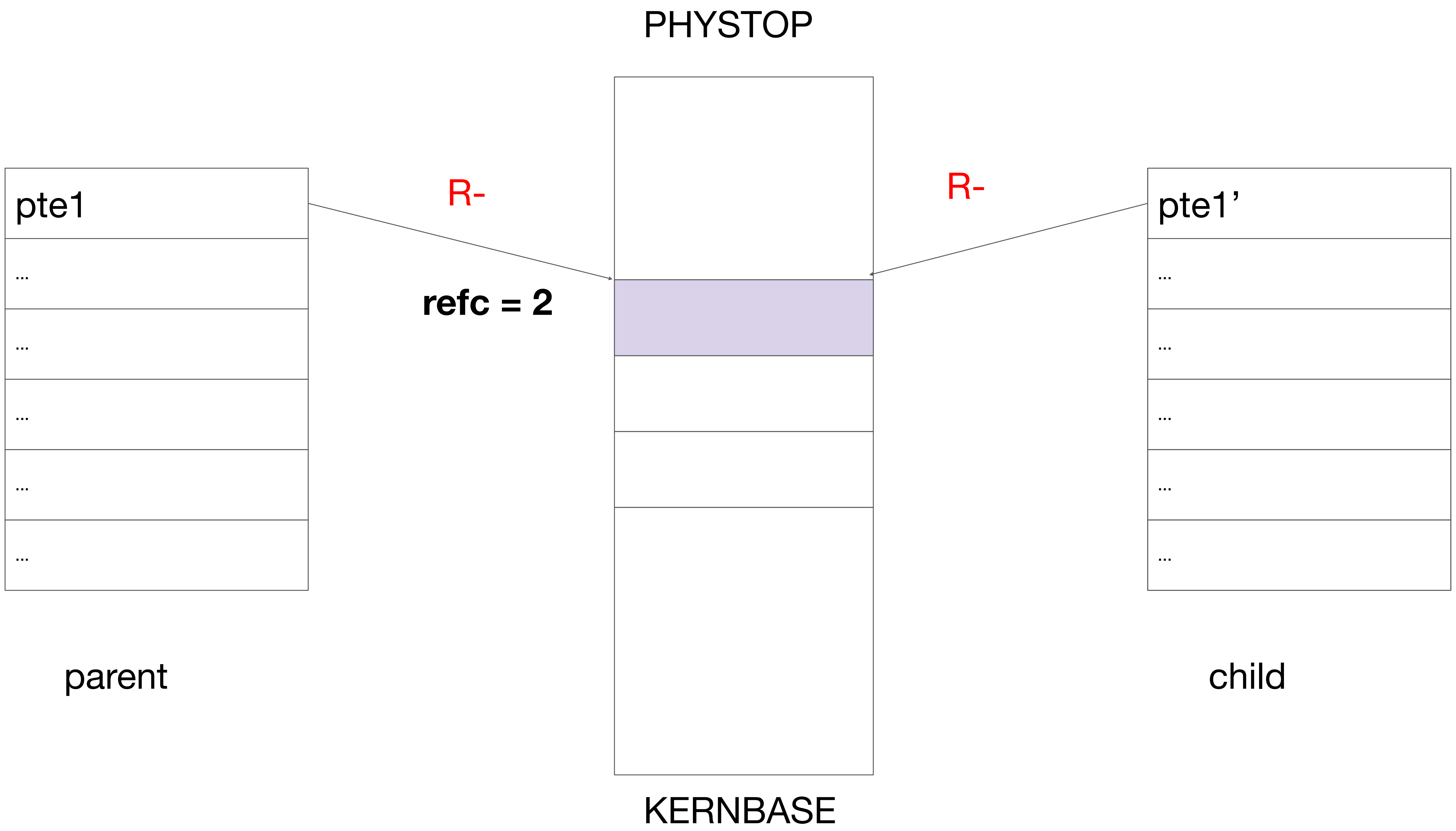
<kernel/proc.c>

```c
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;
```

```c
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err:
  uvmunmap(new, 0, i, 1);
  return -1;
}
```

What will happen in COW fork()?





To achieve this, finish step 1 & 2 in next 2 pages.

Step 1. Implement page reference counter.

In kalloc.c file:

1. Create a new struct that allows you to record the reference count of each physical page;
    a. You can use linked list, or fixed length array, etc.
    b. Modification to the reference count should be guarded by a lock;
2. After calling kalloc() function, the newly allocated physical page’s ref count should be 1;
3. In kfree() function, decrement the ref count, release the physical page when it reaches 0;
4. Create a new function to increment the ref count;

You can refer to file.c for more hint on how a ref counter is used during resource allocation and release. (Hint: filealloc(), filedup(), fileclose())

Step 2. Fix uvmcopy() function

Modify uvmcopy function:

1> remove the new page allocation;

2> map parent's physical page to child's virtual page;

3> clear PTE_W from both proc's PTEs;

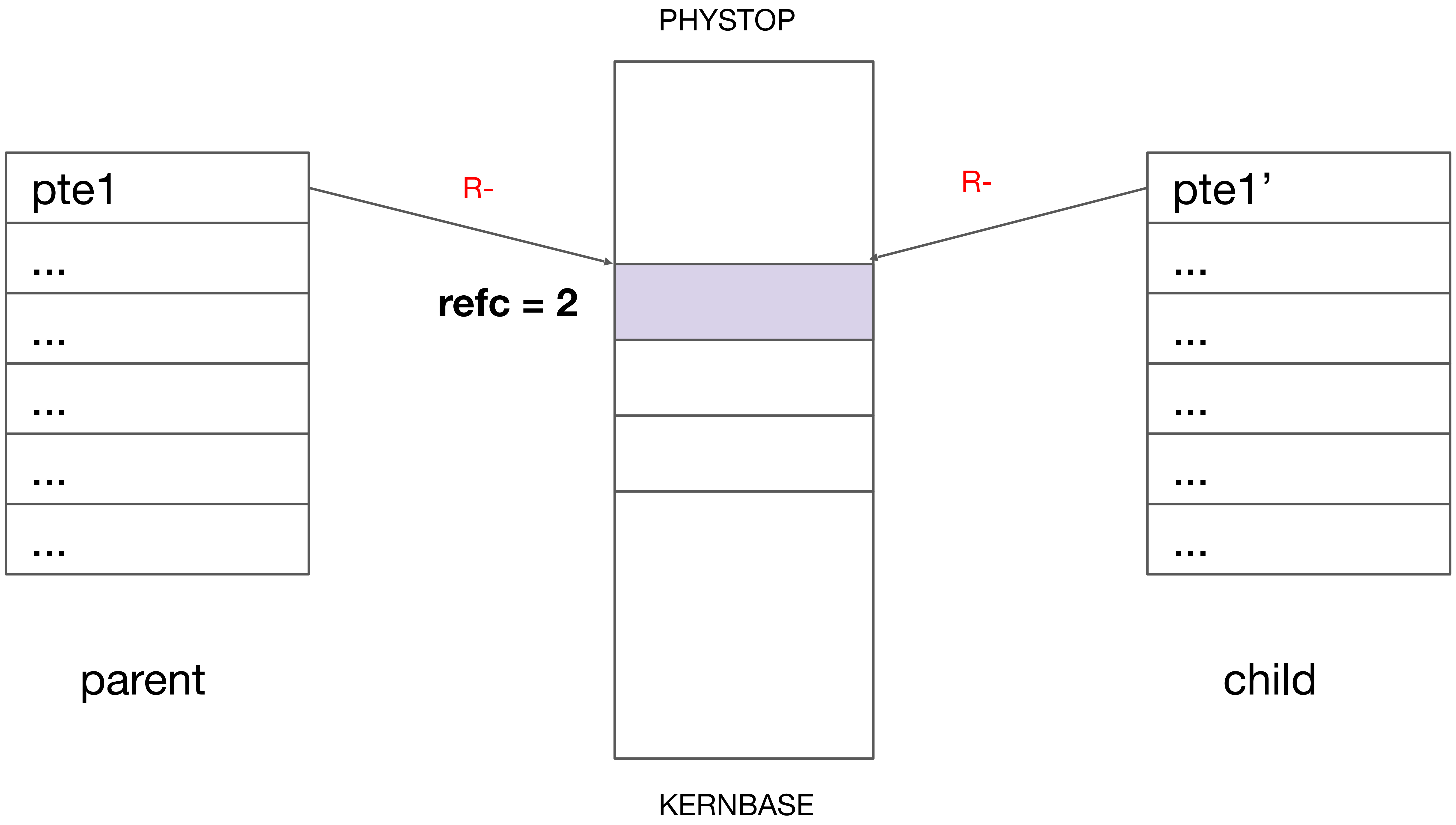
4> define your own privilege flag to record whether a PTE is COW mapping (hint: riscv.h; xv6 book chapter 3.1);

5> increment the page reference counter of this physical page;

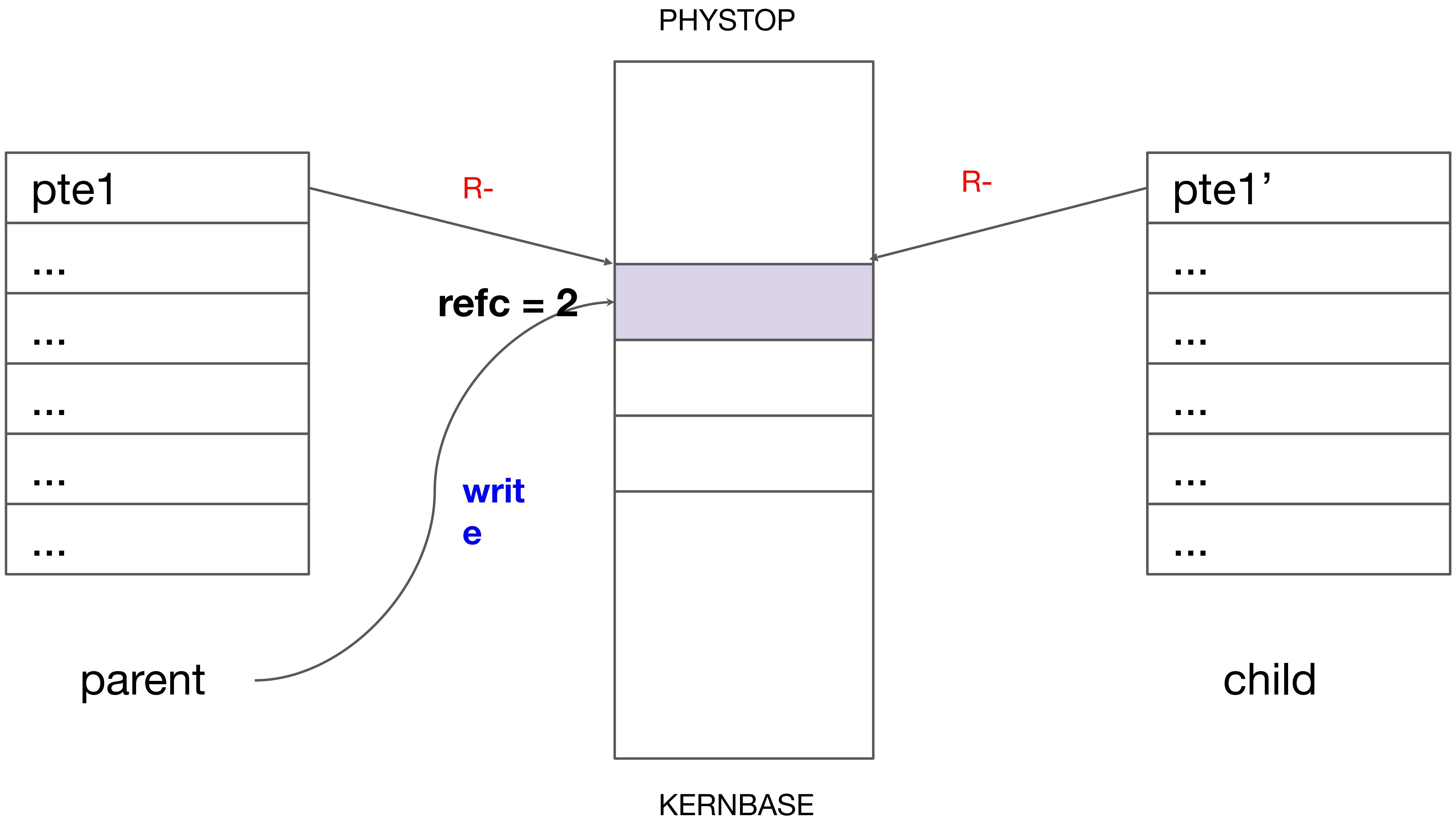
<kernel/vm.c>

```c
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err:
  uvmunmap(new, 0, i, 1);
  return -1;
}
```

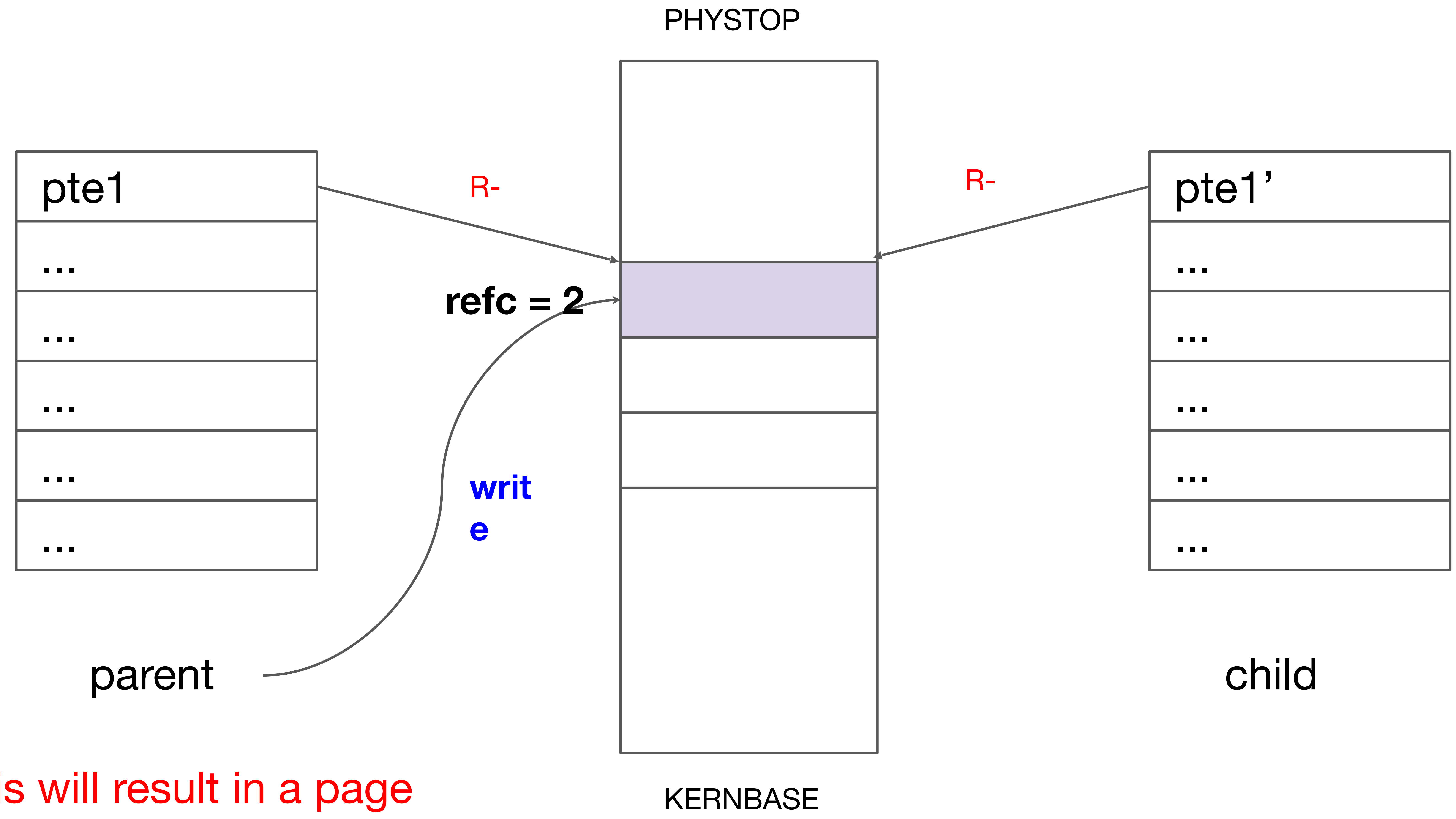
What will happen in COW fork()?



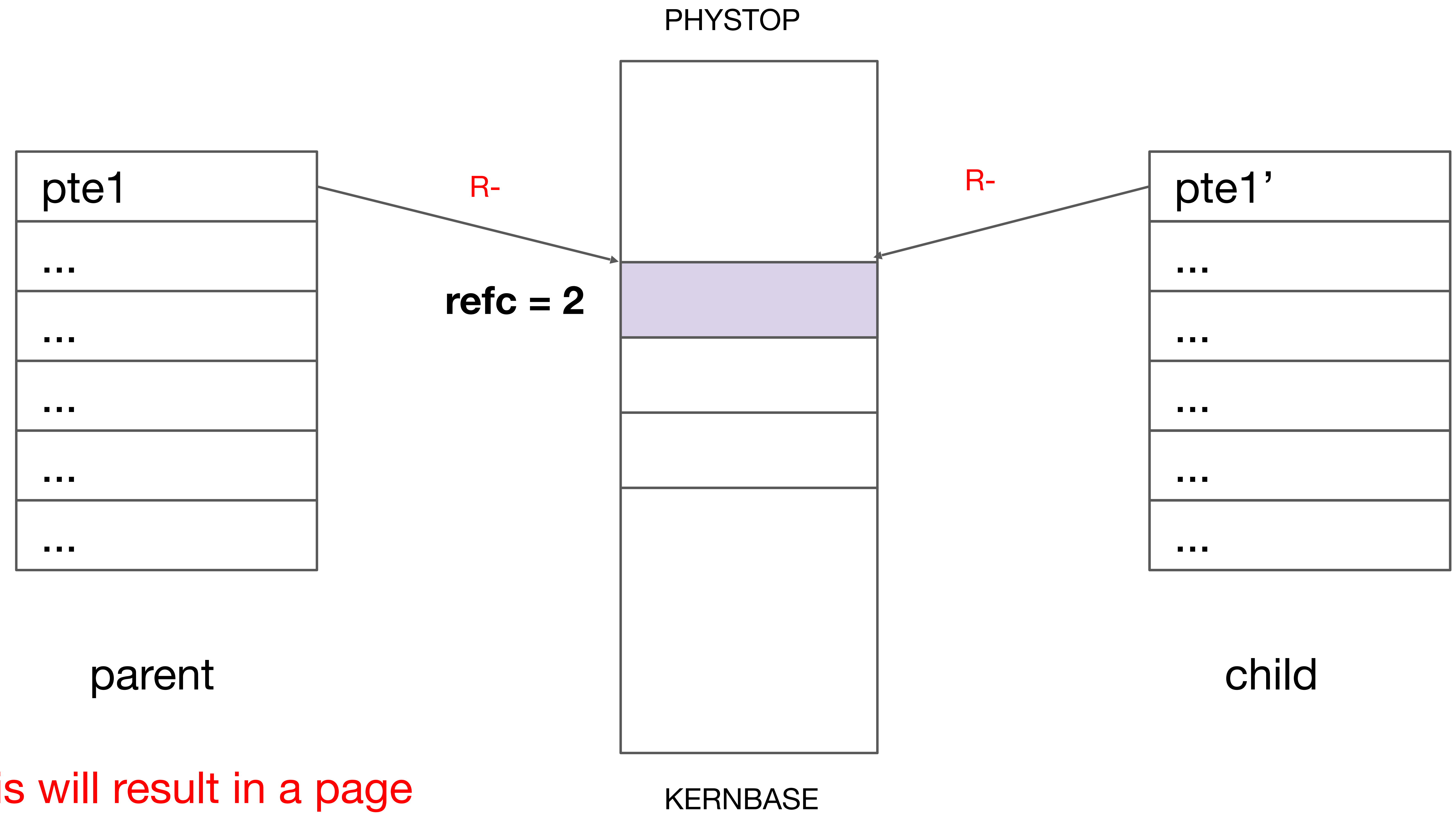
What will happen in COW fork()?



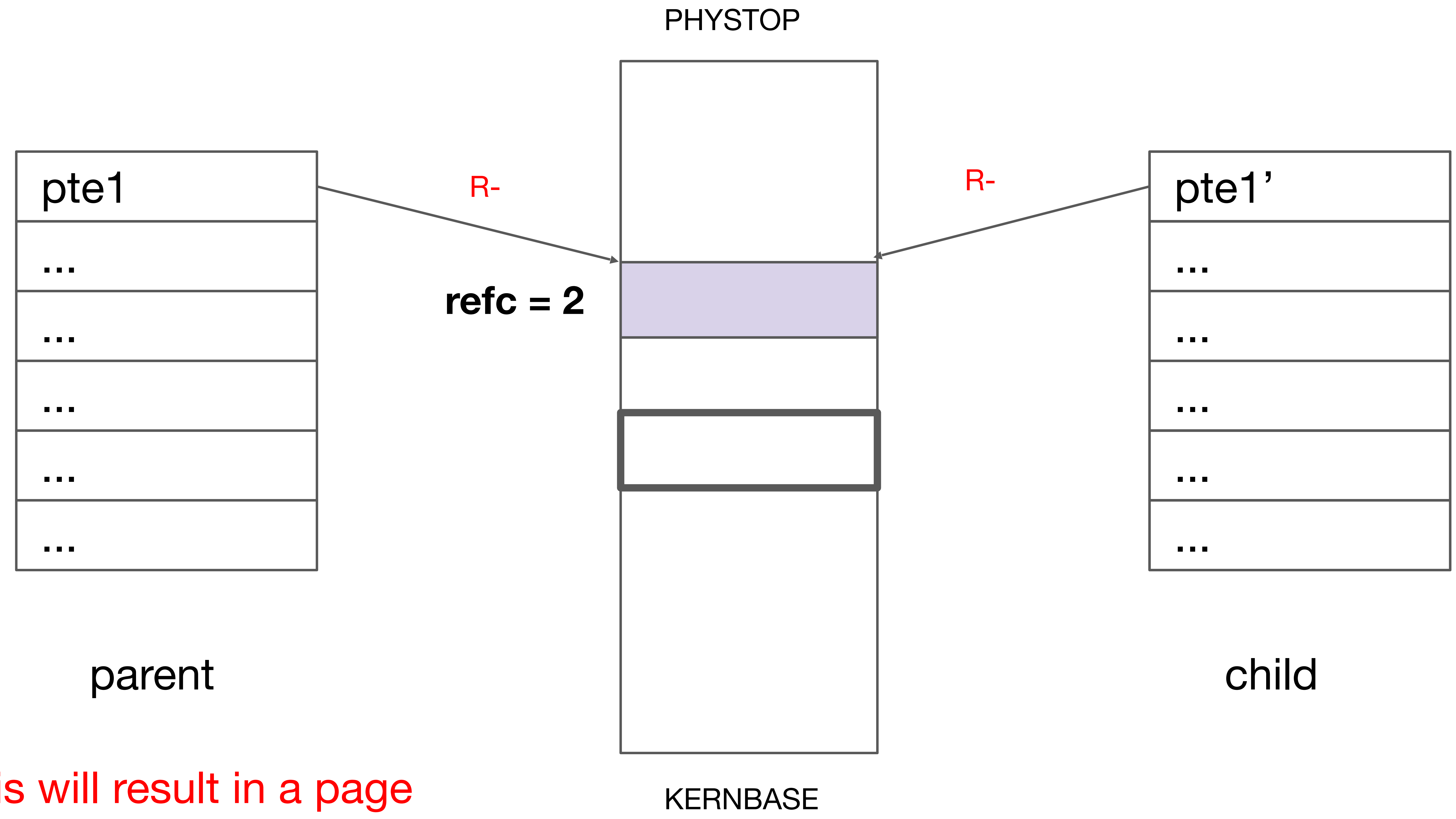
What will happen in COW fork()?



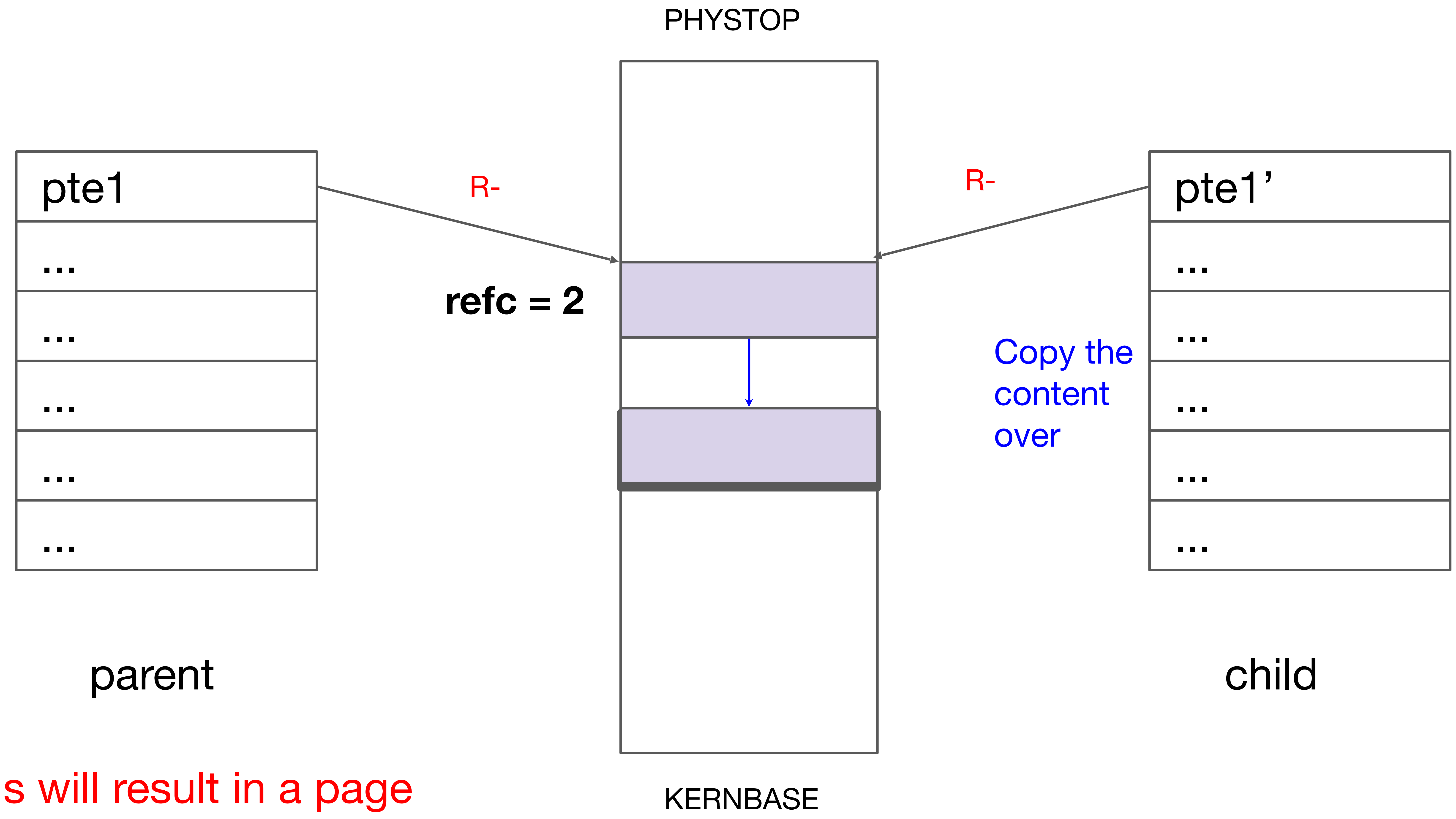
This will result in a page fault, which is handled in usertrap().



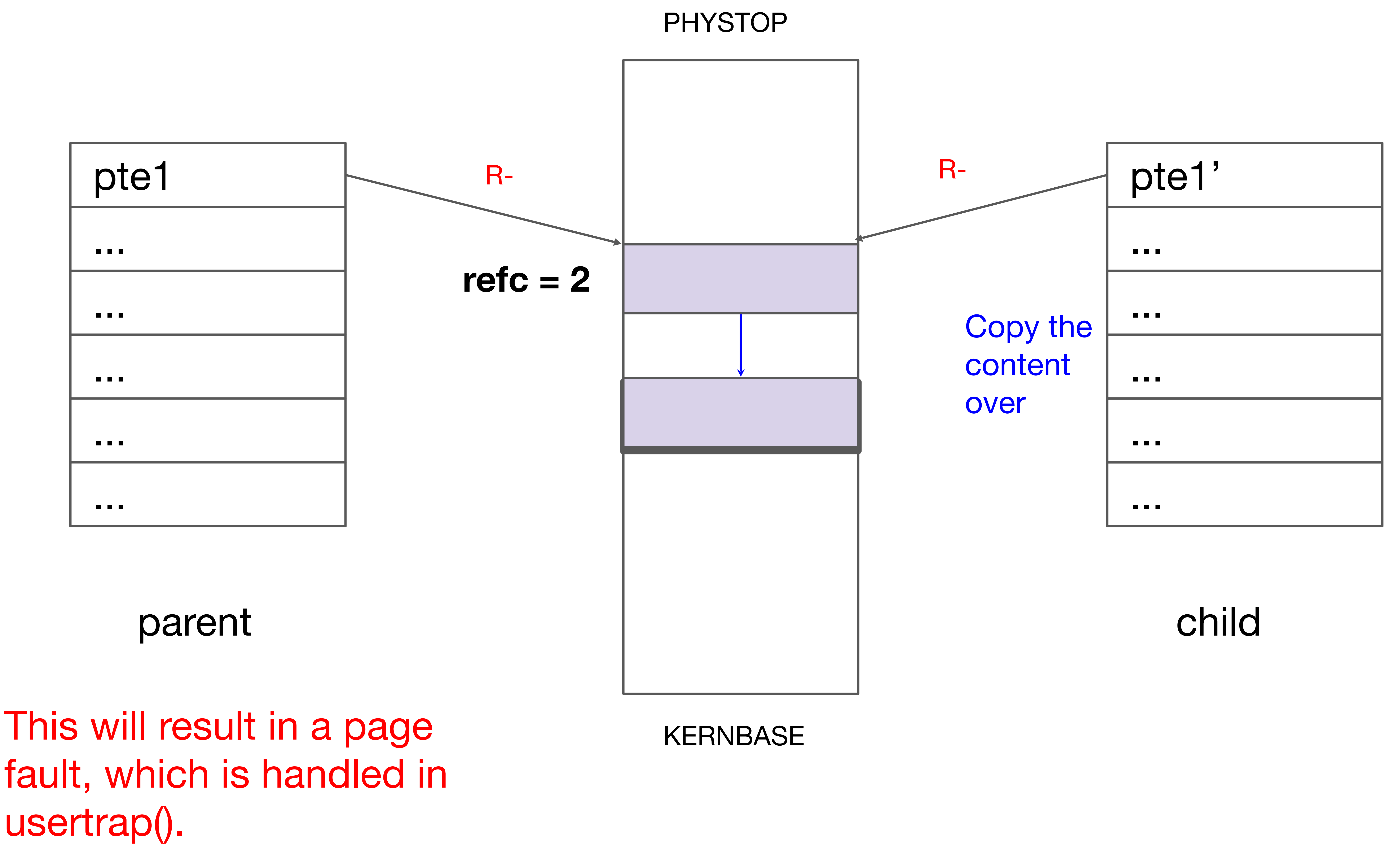
This will result in a page fault, which is handled in usertrap().

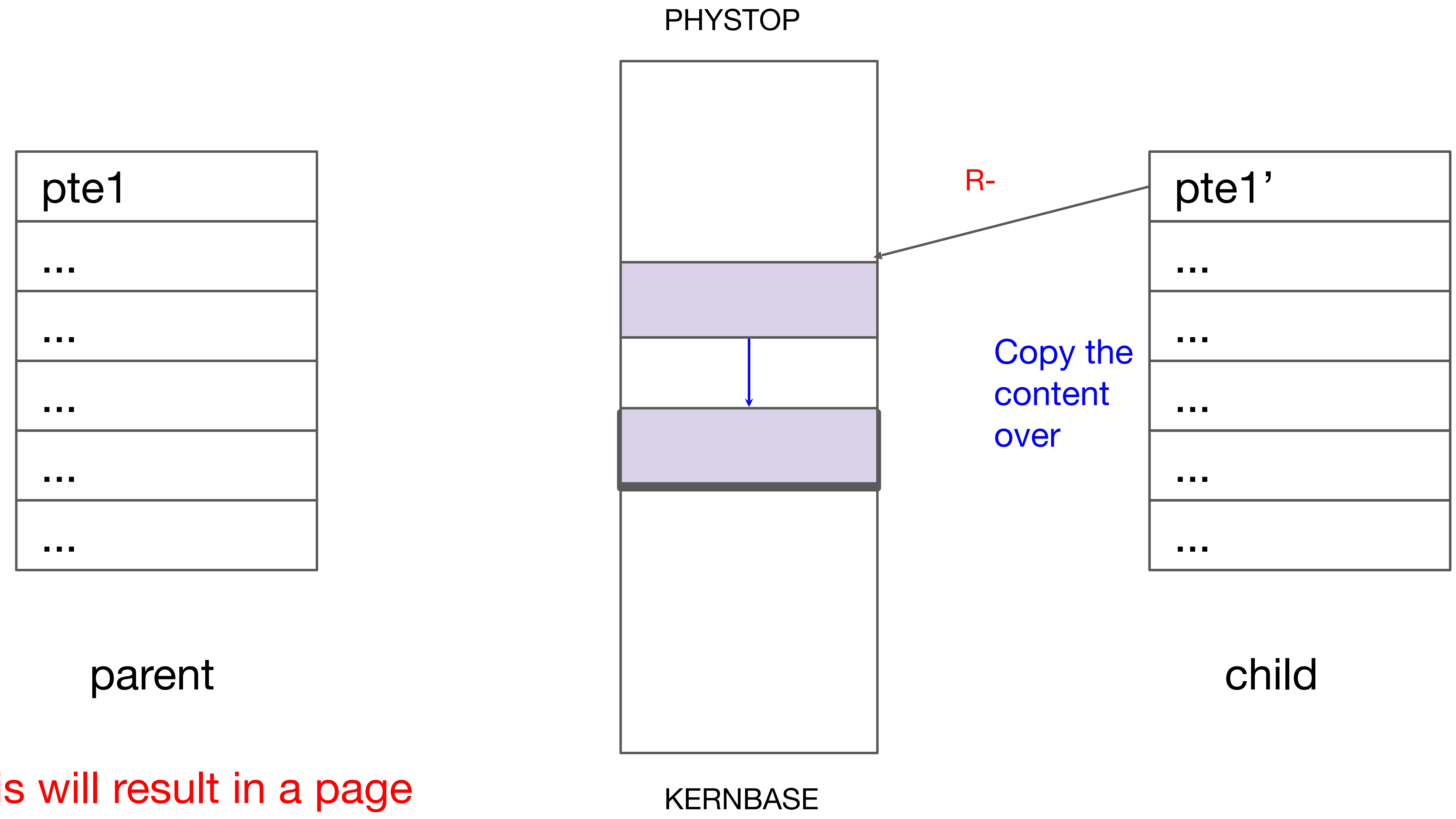


This will result in a page fault, which is handled in usertrap().

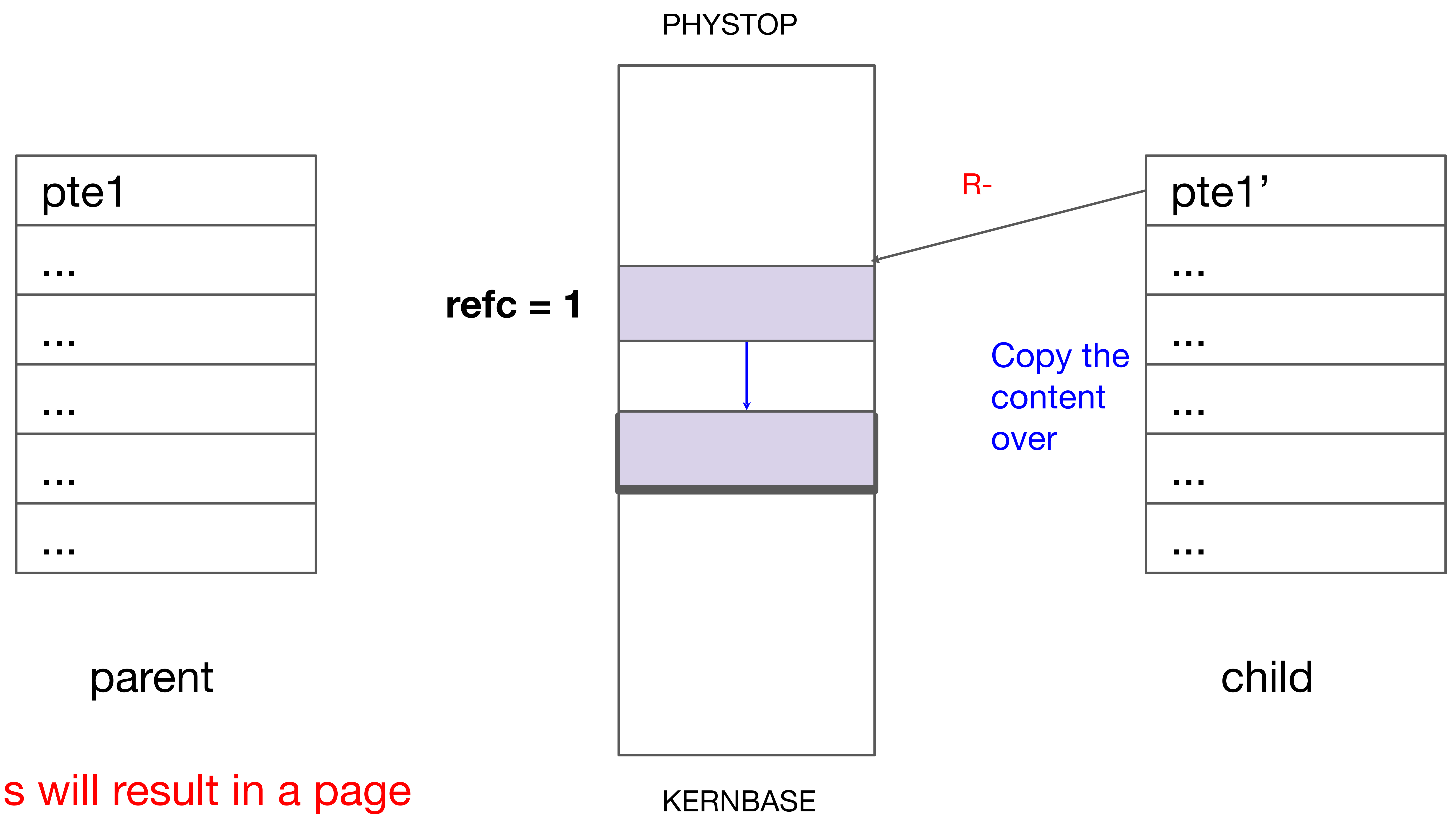


This will result in a page fault, which is handled in usertrap().

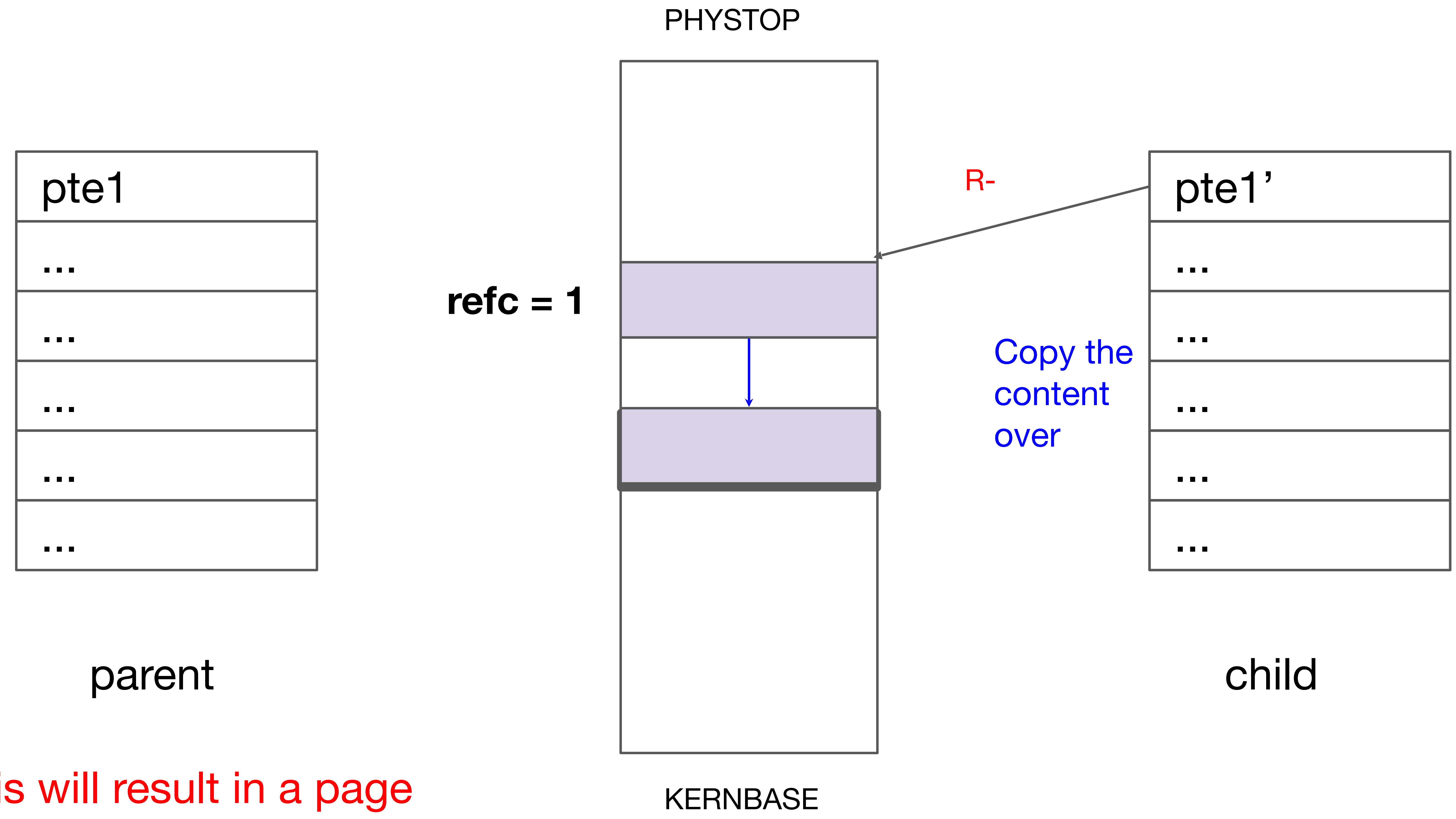

PHYSTOP
pte1
...
...
...
...
...
R-
refc = 2
R-
pte1’
...
...
...
...
...
Copy the content over
parent
child
KERNBASE
This will result in a page fault, which is handled in usertrap().



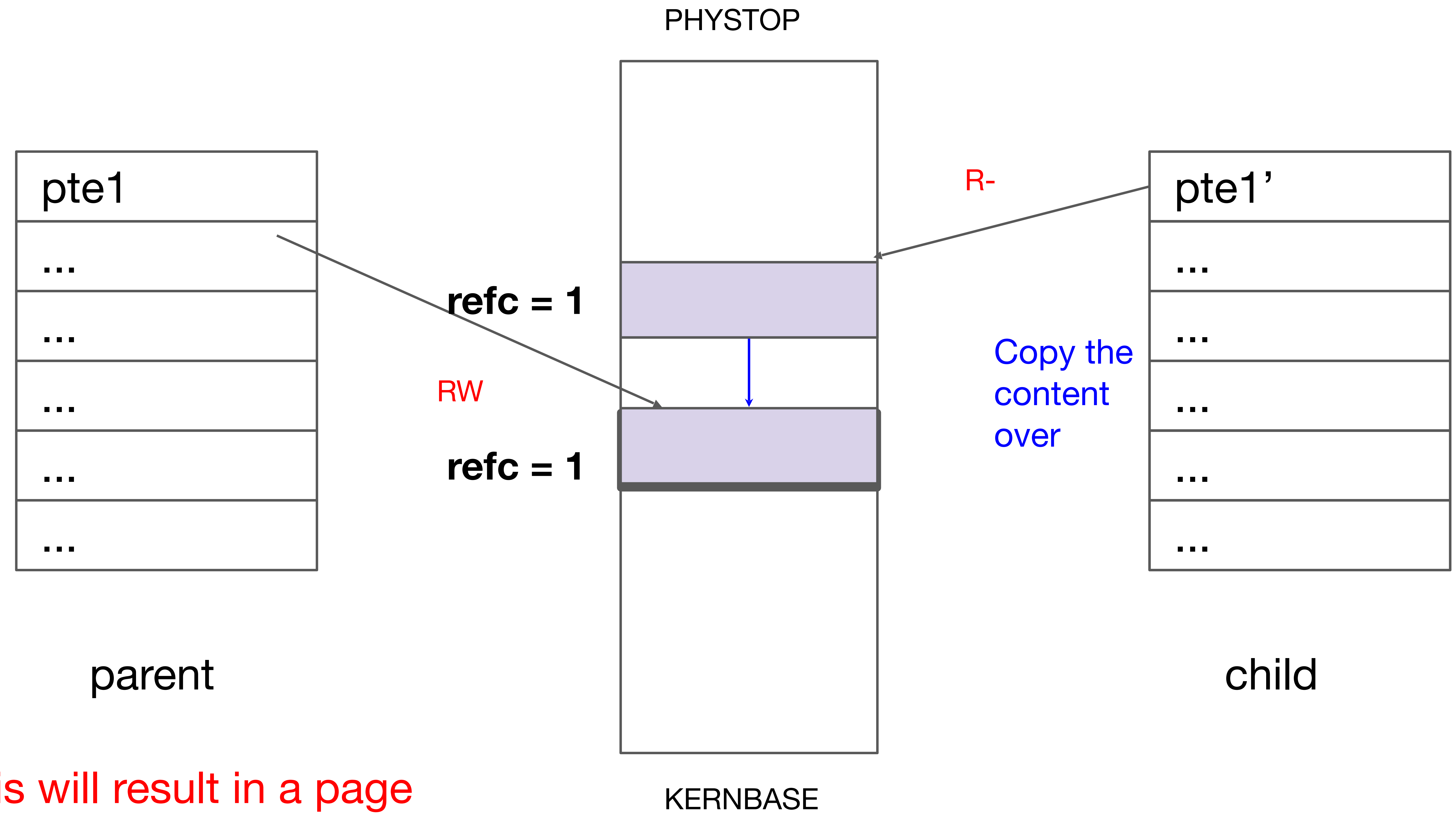
This will result in a page fault, which is handled in usertrap().



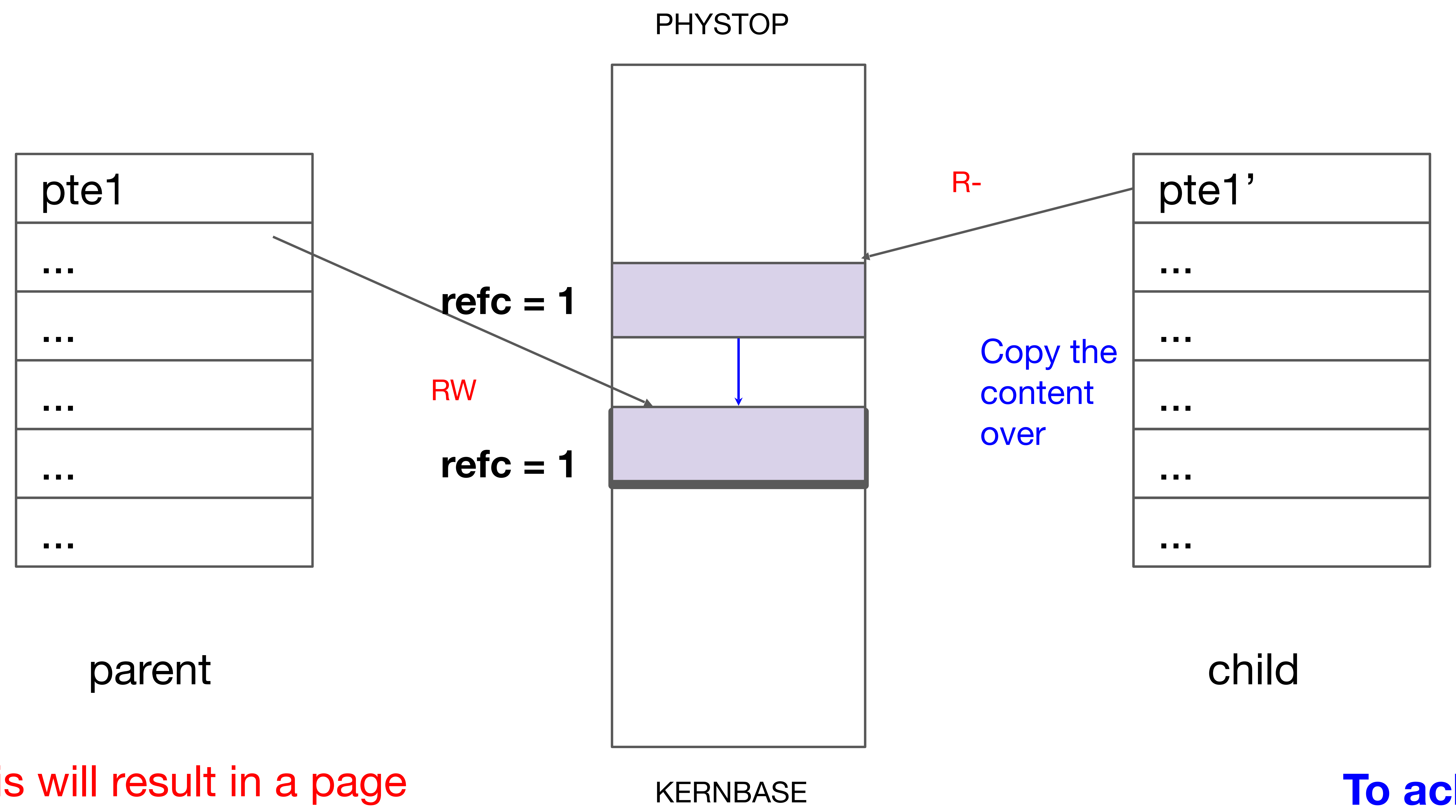
This will result in a page fault, which is handled in usertrap().



This will result in a page fault, which is handled in usertrap().



This will result in a page fault, which is handled in usertrap().



This will result in a page fault, which is handled in usertrap().

To achieve this, finish step 3 in the next page.

Step 3. Fix usertrap() function

Modify usertrap() function to:

1> Check if this is a writing (r_scause() returns 15) page fault to a COW page, you can use your newly defined privilege flag to validate;

2> Respond to writing page fault to COW page, allocate a new physical page and duplicate the content of the COW page; (hint: memmove())

3> remap the faulting virtual page from the COW page to the new page, with PTE_W flag on; (hint: unmap from the COW page, then map to the new page);

```c
    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if(r_scause() == 15) { // write page fault
    /*
      your modification goes here
    */
  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }
```

Step 4. Fix copyout() function

Copy from kernel to user virtual address, which could belong to a COW page;

Handle the COW page similarly as in usertrap function;

Allocate a new physical page, copy COW page content to the new one;

Remap faulting virtual page to the new physical page;

```c
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    /*
      check if dstva belongs to a COW page with no writing priviledge
      use the same scheme as page faults if so.
    */
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

Lab 3 Grading

You need to pass cowtest and usertests to complete lab 3:

$ make qemu

$ cowtest

$ usertest

To grade lab 3:

$ echo “X” > time.txt
// X is the number of hours that you spent on this lab

$ make grade

```
running cowtest:
$ make qemu-gdb
(15.4s)
  simple: OK
  three: OK
  file: OK
usertests:
$ make qemu-gdb
OK (144.0s)
time: FAIL
    Cannot read time.txt
Score: 99/100
make: *** [grade] Error 1
```

```
running cowtest:
$ make qemu-gdb
(10.1s)
  simple: OK
  three: OK
  file: OK
usertests:
$ make qemu-gdb
OK (126.8s)
time: OK
Score: 100/100
```