# Operating Systems Practice

Virtual Memory – Copy-on-Write

Eunji Lee

(ejlee@ssu.ac.kr)

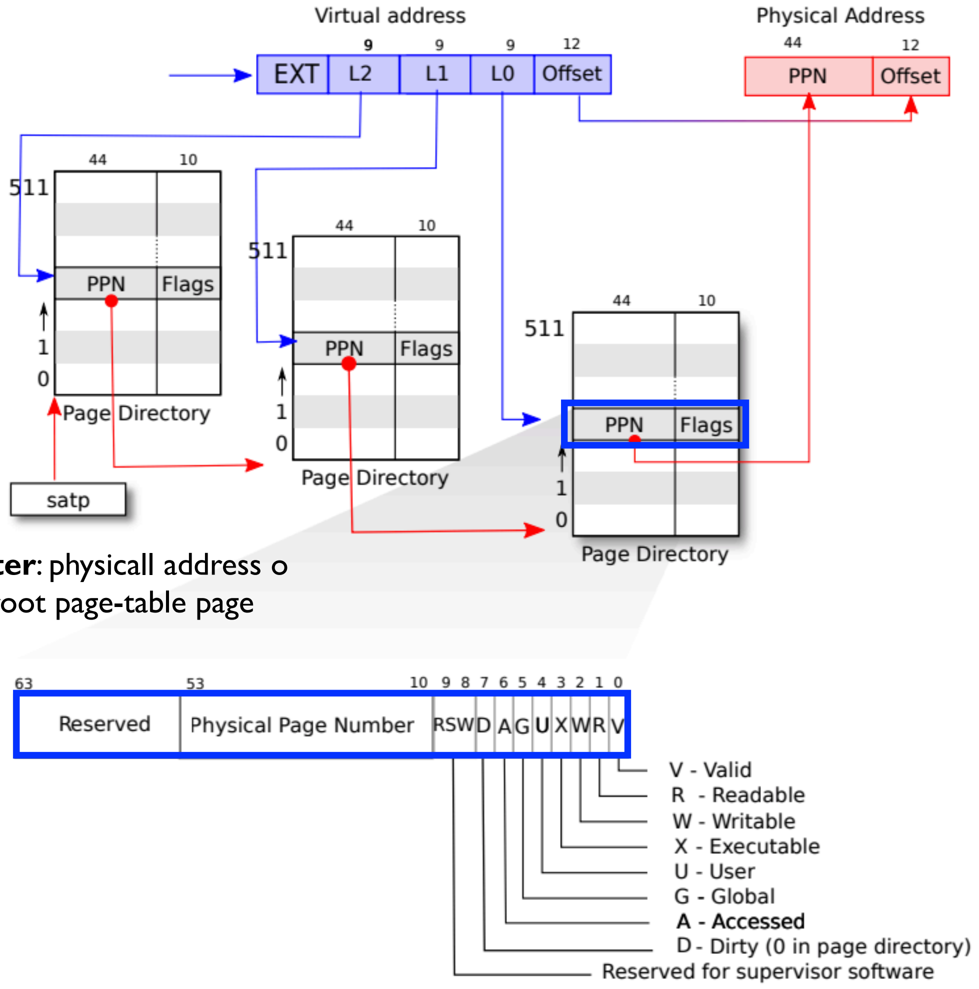
# Reference

- “xv6: a simple, Unix-like teaching operating system,” Chapter 3. Page tables

# Paging Hardware

- RISC-V uses 39-bit virtual address
- A page table is stored in physical memory as a three-level tree
- Each pte has 44-bit PPN and 10-bit flag

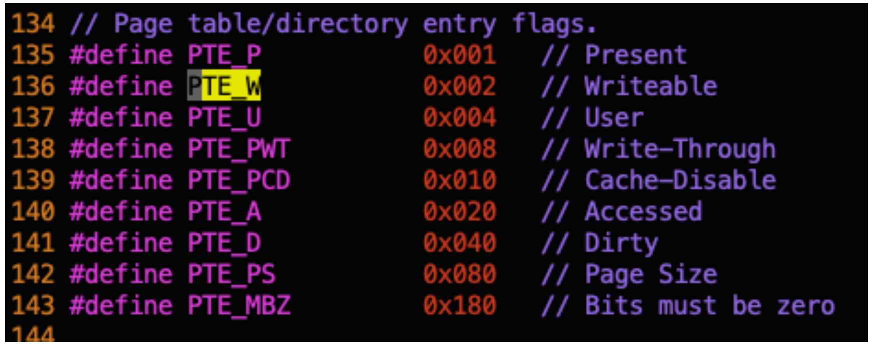
**root**
- 4KB (1page)
- 512 entries



**register**: physicall address o f the root page-table page

Figure 3.2: RISC-V page table hardware.

mmu.h



```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // Accessed
#define PTE_D           0x040   // Dirty
#define PTE_PS          0x080   // Page Size
#define PTE_MBZ         0x180   // Bits must be zero
```

# Kernel Address Space

- Direct mapped
- Only kernel stack pages are not direct-mapped
- guard page
  - Prevent problem caused by kernel stack overflow
  - guard page's PTE is invalid

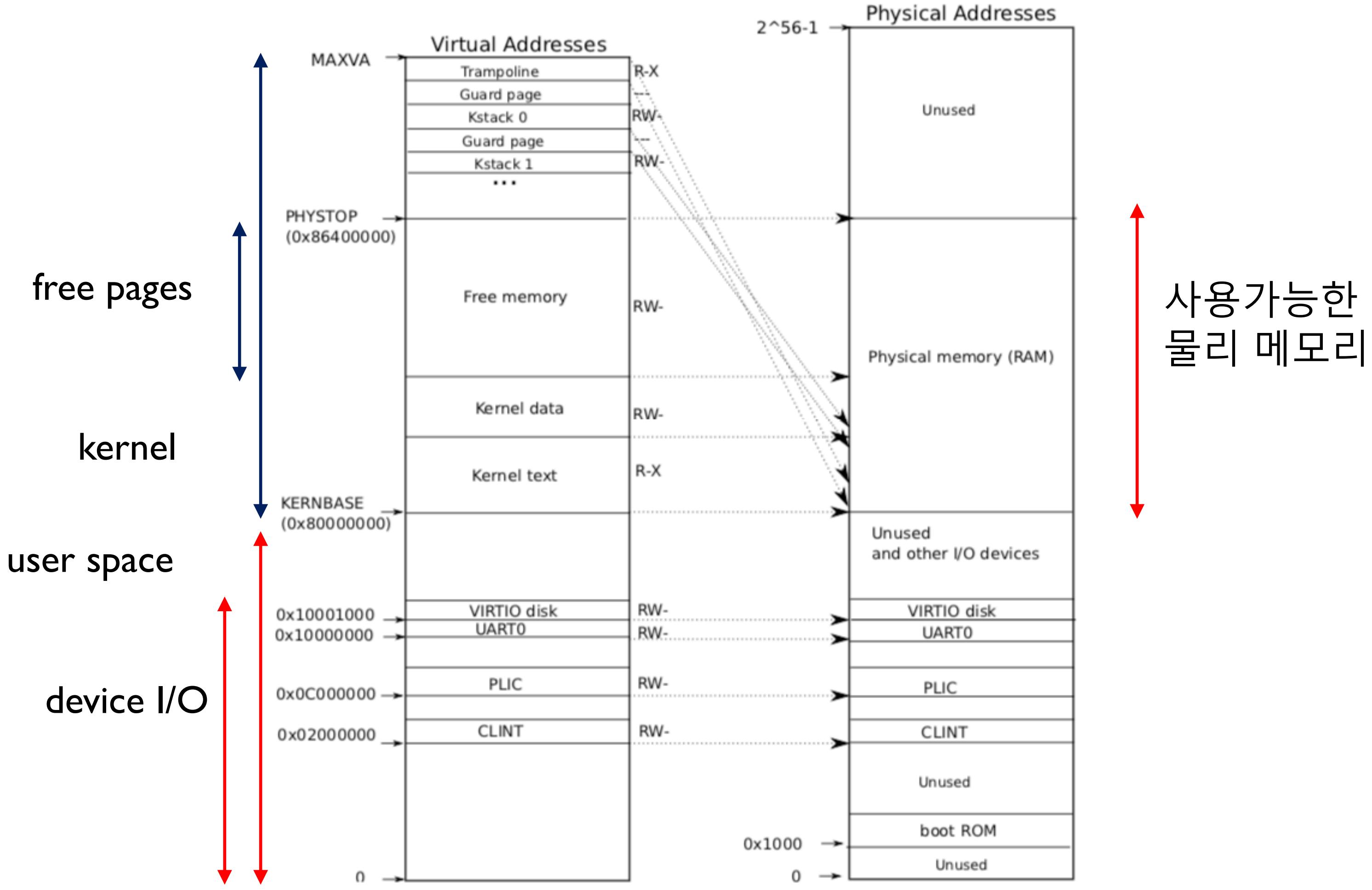
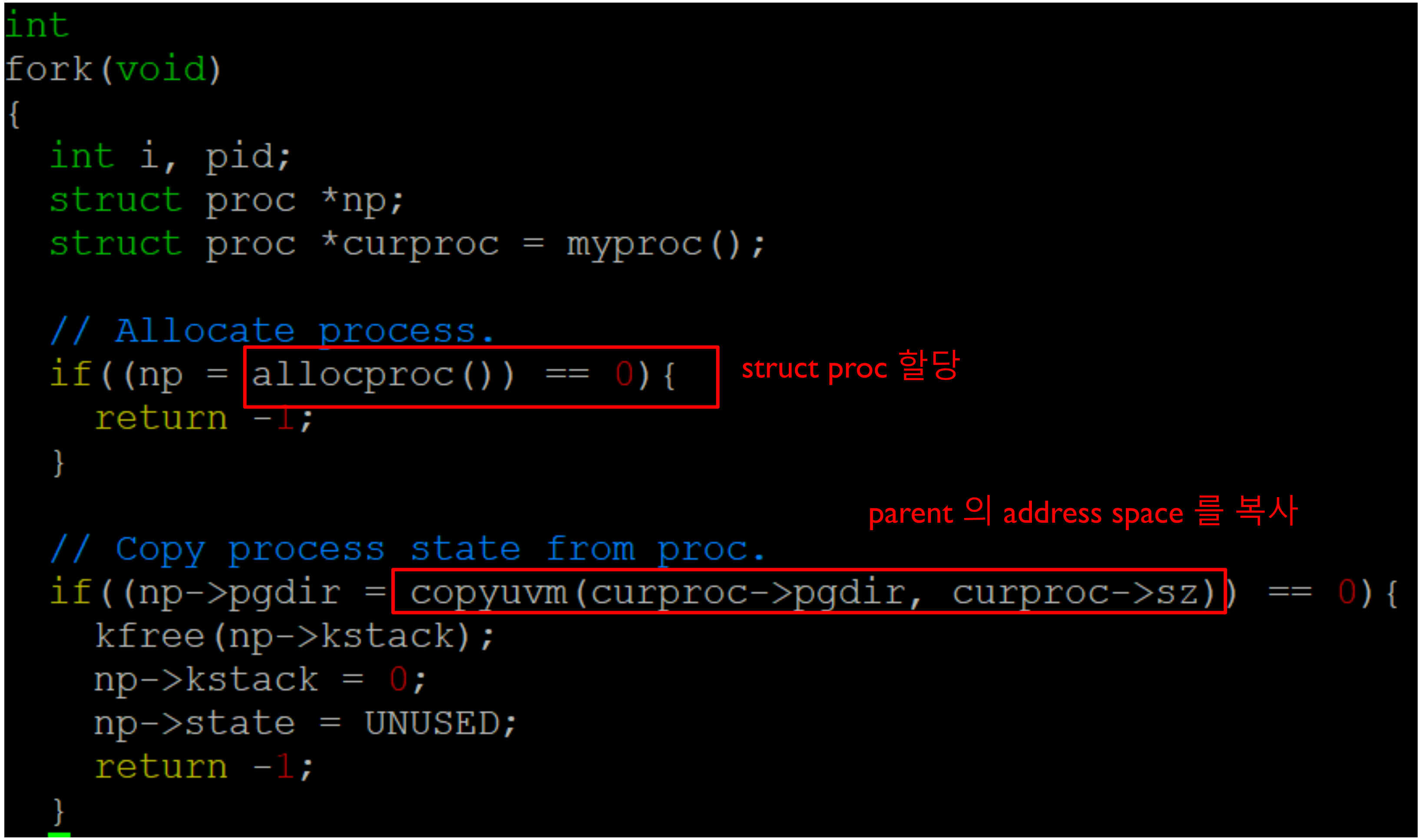


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

# Creating an address space

- vm.c : code for manipulate address spaces and page table

proc.c



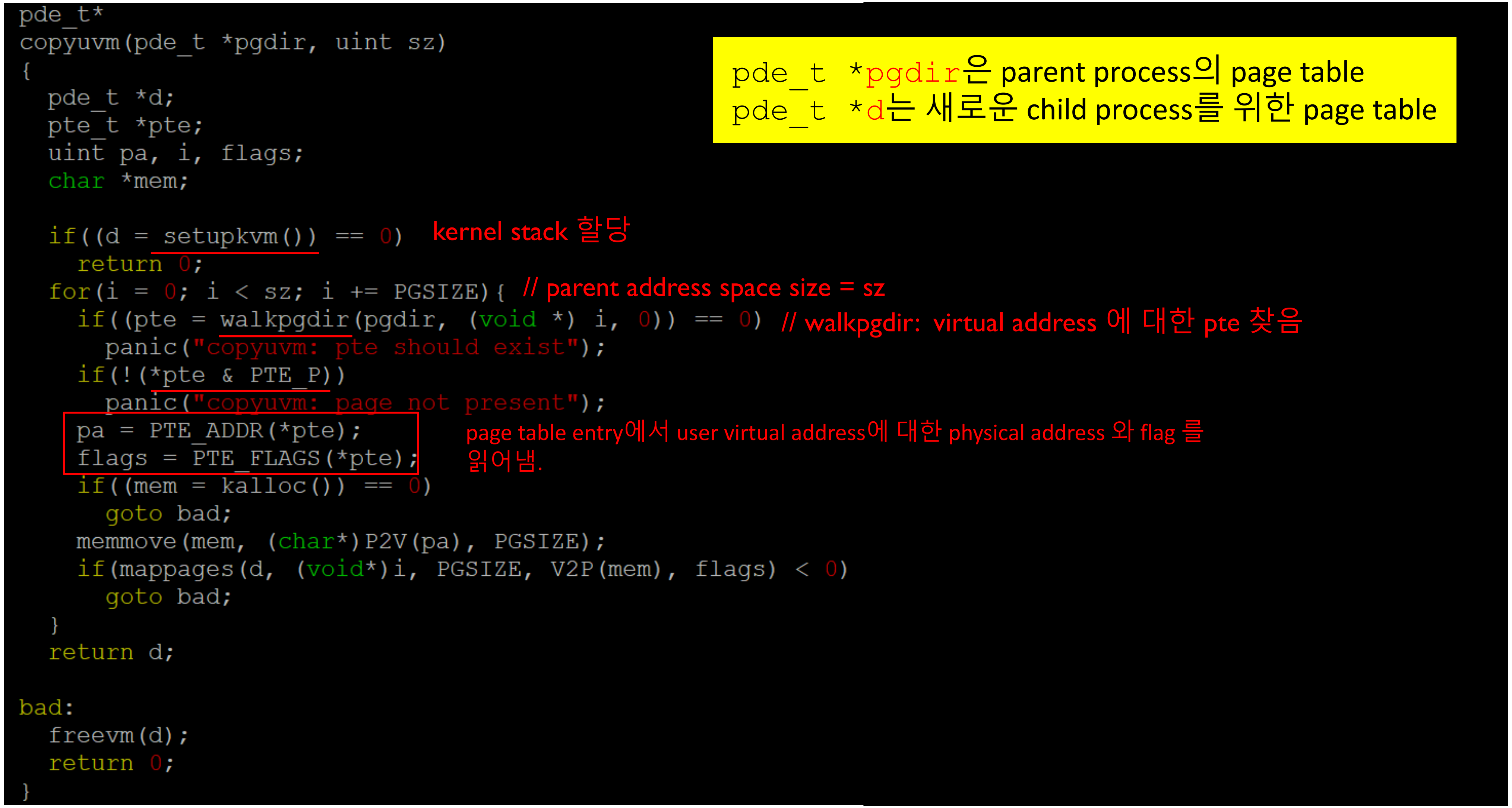
```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

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

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
```

struct proc 할당

parent 의 address space 를 복사

# Creating an address space

vm.c



```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
      goto bad;
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

pde_t *pgdir은 parent process의 page table
pde_t *d는 새로운 child process를 위한 page table

kernel stack 할당

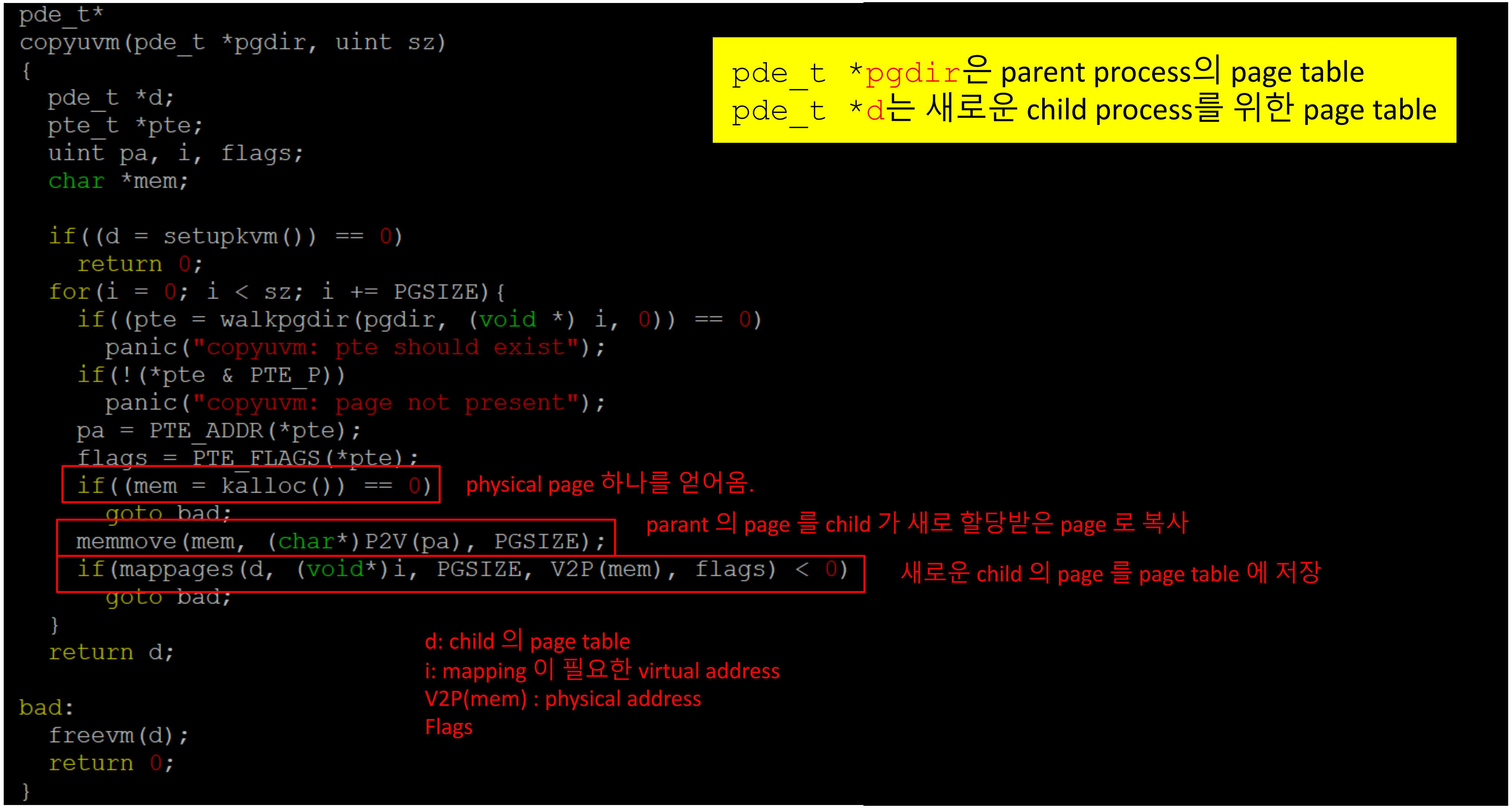
// parent address space size = sz

// walkpgdir: virtual address 에 대한 pte 찾음

page table entry에서 user virtual address에 대한 physical address 와 flag 를 읽어냄.

# Creating an address space

vm.c



```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
      goto bad;
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

pde_t *pgdir은 parent process의 page table
pde_t *d는 새로운 child process를 위한 page table

physical page 하나를 얻어옴.

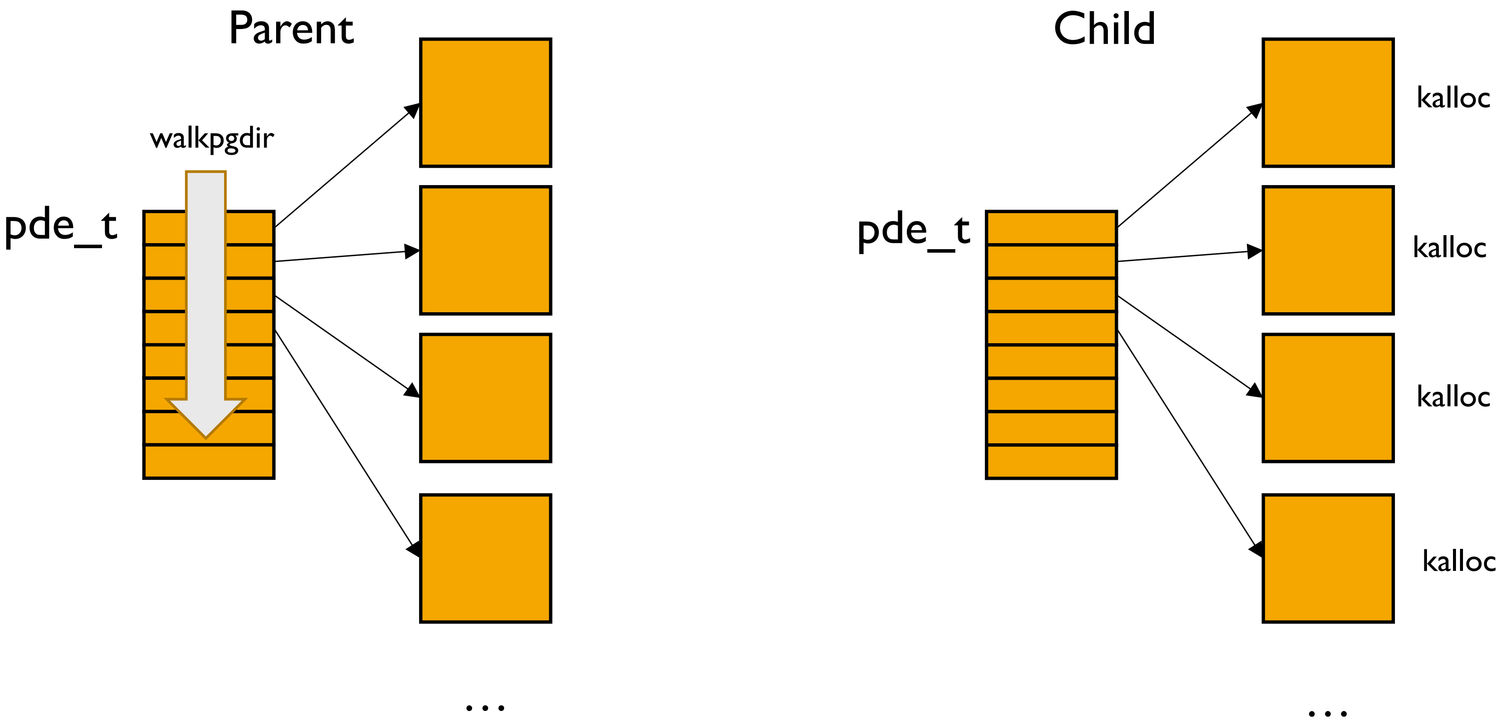
parant 의 page 를 child 가 새로 할당받은 page 로 복사

새로운 child 의 page 를 page table 에 저장

d: child 의 page table
i: mapping 이 필요한 virtual address
V2P(mem) : physical address
Flags

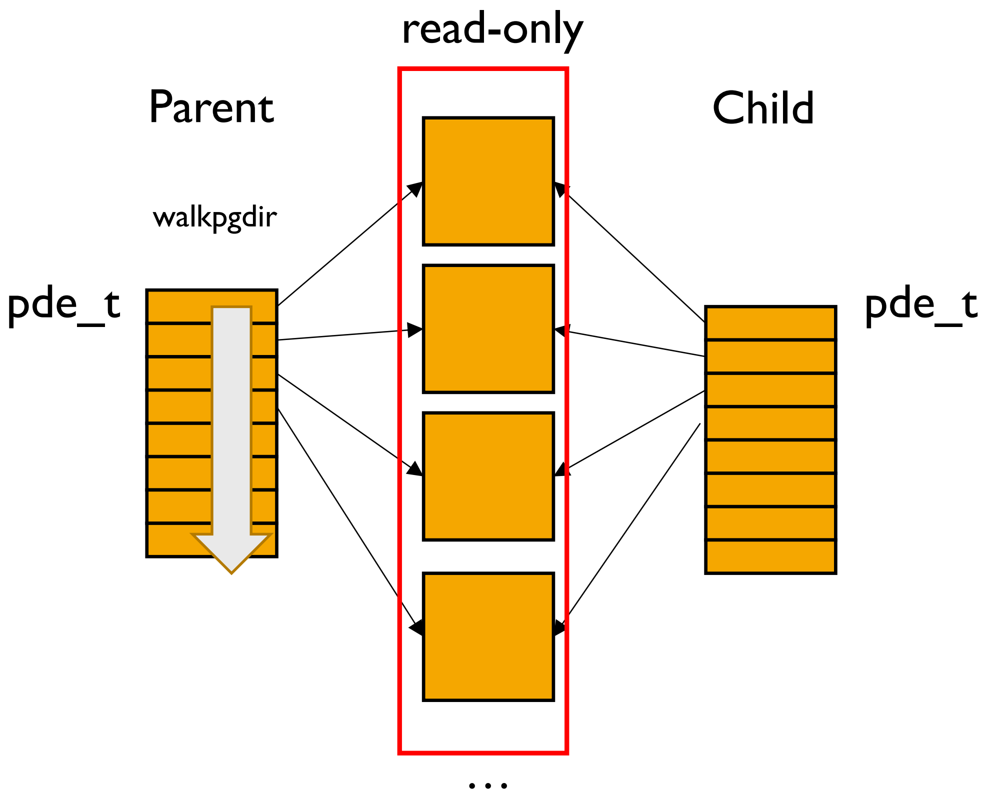
# Creating an address space

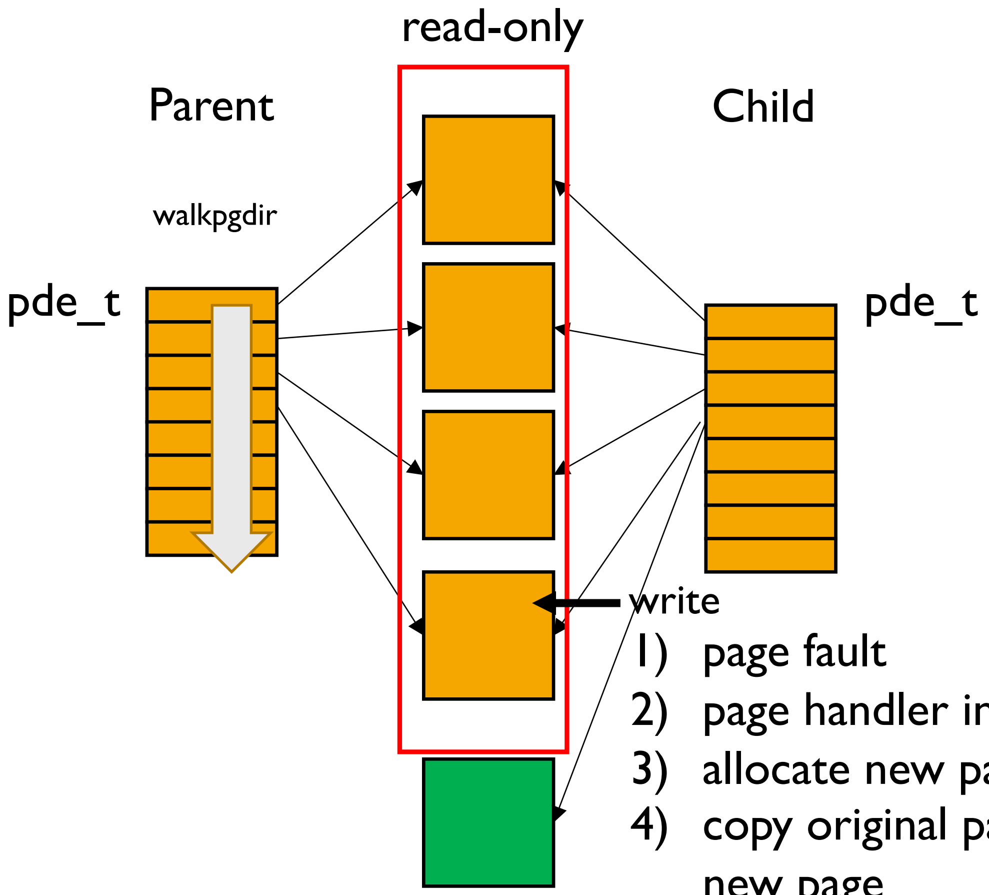
- Duplication (copyuvm() in vm.c)



# Creating an address space

- Copy-on-write (copyuvm() in vm.c)

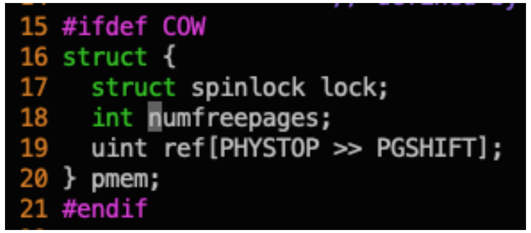




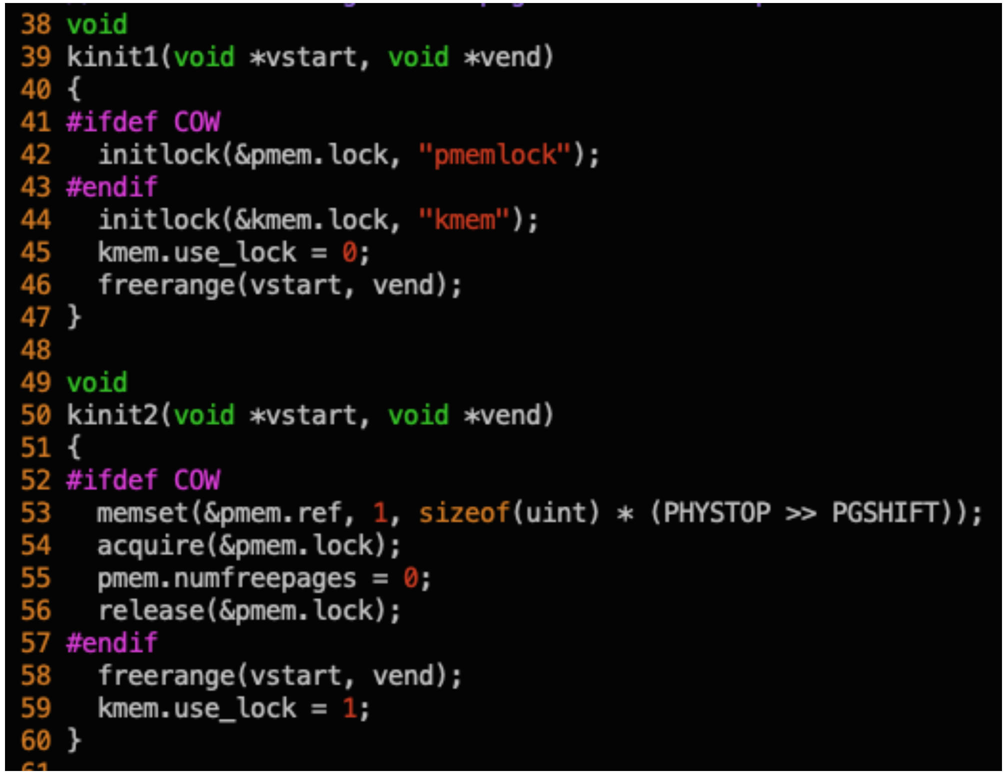
1) page fault
2) page handler invoked
3) allocate new page
4) copy original page into new page
5) update page table entry
6) TLB flush

# Modification – Reference Counter

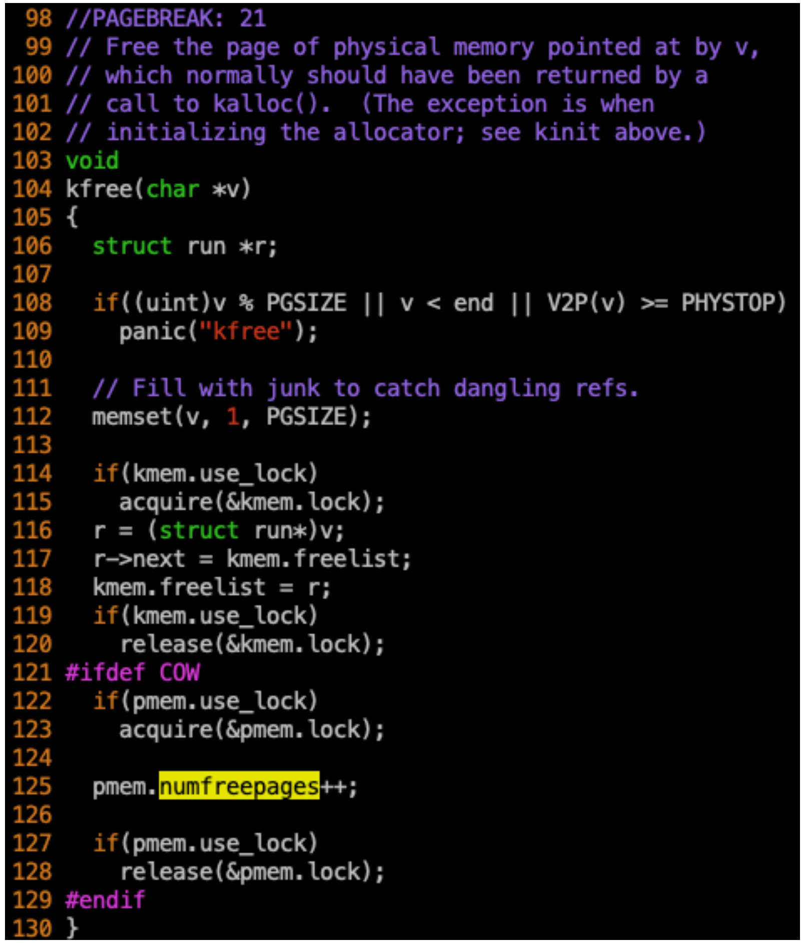
- Add reference counter and associated functions
  - kalloc.c



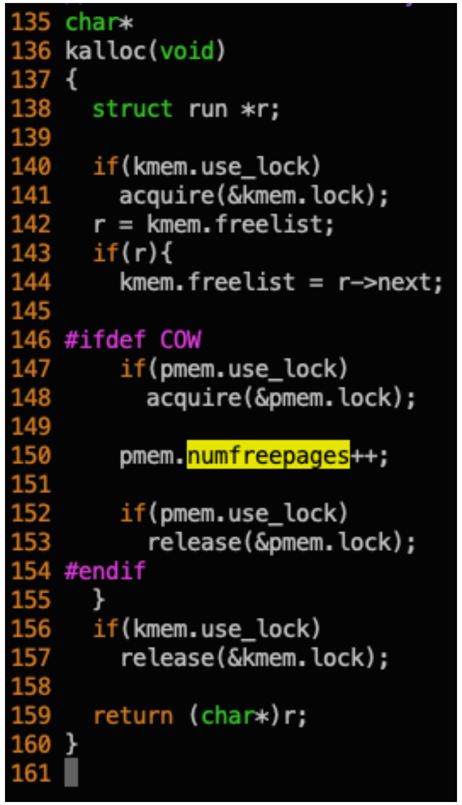
```
#ifdef COW
struct {
  struct spinlock lock;
  int numfreepages;
  uint ref[PHYSTOP >> PGSHIFT];
} pmem;
#endif
```



```
void
kinit1(void *vstart, void *vend)
{
#ifdef COW
  initlock(&pmem.lock, "pmemlock");
#endif
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}

void
kinit2(void *vstart, void *vend)
{
#ifdef COW
  memset(&pmem.ref, 1, sizeof(uint) * (PHYSTOP >> PGSHIFT));
  acquire(&pmem.lock);
  pmem.numfreepages = 0;
  release(&pmem.lock);
#endif
  freerange(vstart, vend);
  kmem.use_lock = 1;
}
```



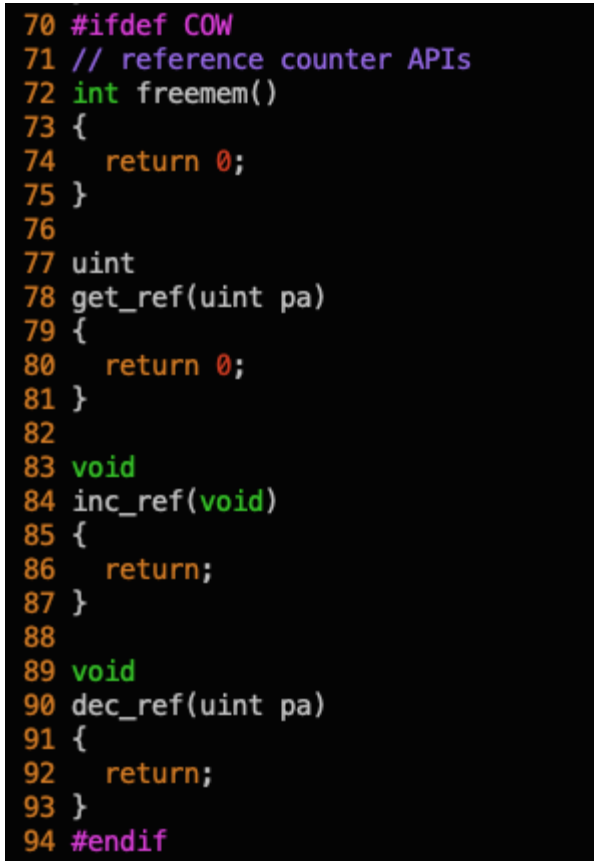
```
//PAGEBREAK: 21
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
#ifdef COW
  if(pmem.use_lock)
    acquire(&pmem.lock);

  pmem.numfreepages++;

  if(pmem.use_lock)
    release(&pmem.lock);
#endif
}
```



```
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;

#ifdef COW
    if(pmem.use_lock)
      acquire(&pmem.lock);

    pmem.numfreepages++;

    if(pmem.use_lock)
      release(&pmem.lock);
#endif
  }
  if(kmem.use_lock)
    release(&kmem.lock);

  return (char*)r;
}

```

# Modification – Reference Counter

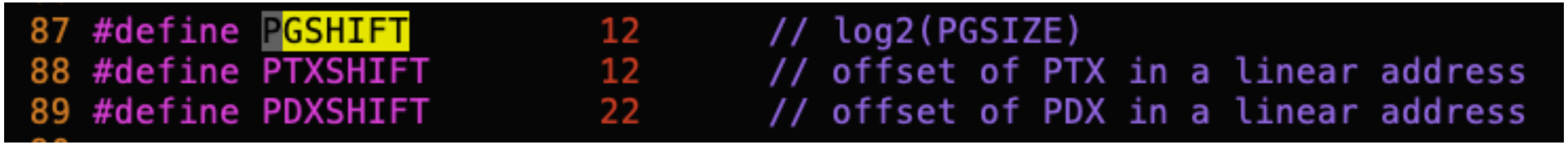
- Add reference counter and associated functions
  - kalloc.c



```
#ifdef COW
// reference counter APIs
int freemem()
{
  return 0;
}

uint
get_ref(uint pa)
{
  return 0;
}

void
inc_ref(void)
{
  return;
}

void
dec_ref(uint pa)
{
  return;
}
#endif
```

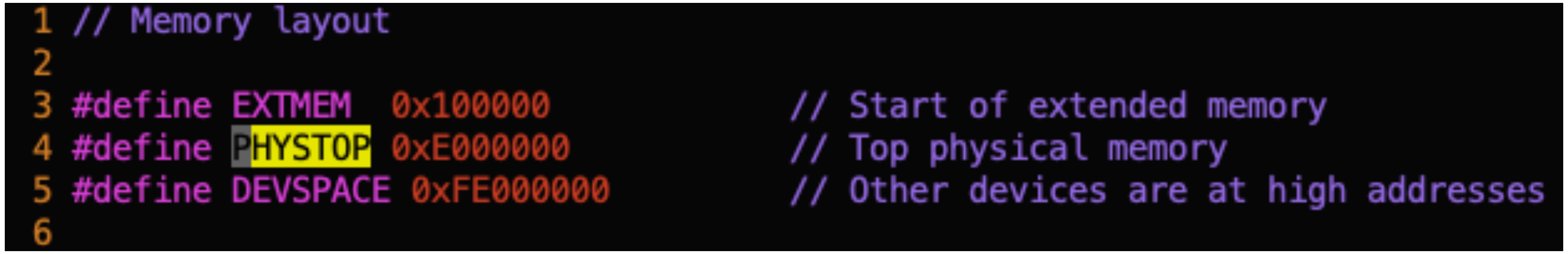
# Modification – Reference Counter

- Add reference counter and associated functions
  - `mmu.h`



```
#define PGSHIFT          12      // log2(PGSIZE)
#define PTXSHIFT         12      // offset of PTX in a linear address
#define PDXSHIFT         22      // offset of PDX in a linear address
```

  - `memlayout.h`



```
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

```

# Modification – Reference Counter

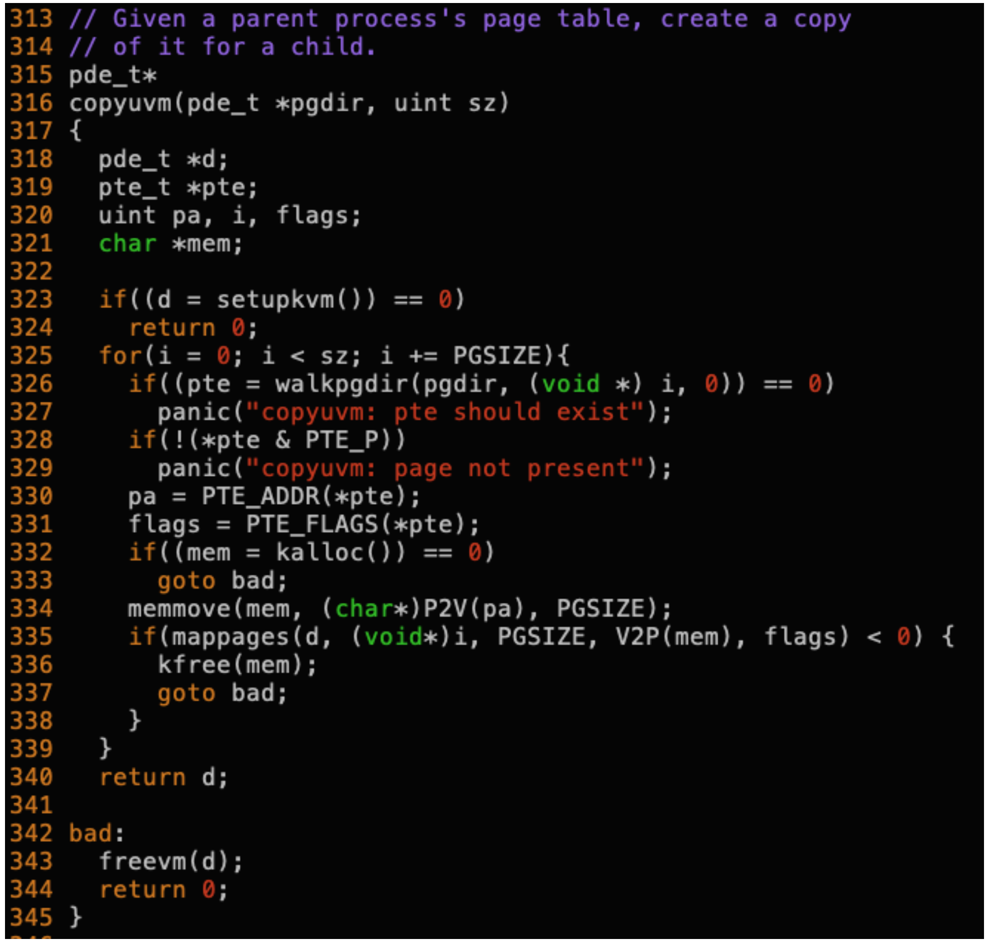
- Add reference counter and associated functions in kalloc.c
  - `ref[PHYSTOP >> PGSHIFT]`: reference counter for physical memory pages

  - `uint get_ref(uint pa):` read the reference count of pa
  - `void inc_ref(uint pa):` increase the reference count of pa
  - `void dec_ref(uint pa):` decrease the reference count of pa
- Increase reference counter
  - When allocating the physical page
  - When referencing the physical page
- Decrease reference counter
  - When de-allocating the physical page
  - When de-referencing the physical page
- Increase / decrease reference counter appropriately
  - kalloc / kfree

# Modification – Sharing pages

- copyuvm() in vm.c
  - DO NOT allocate a new page for child's address space
  - Install parent's address spaces to child's page table
  - Set the page NOT writable (flags) for both child and parent
  - Increase reference counter for shared page
  - TLB flush: `lcr3(V2P(pgdir))`

vm.c

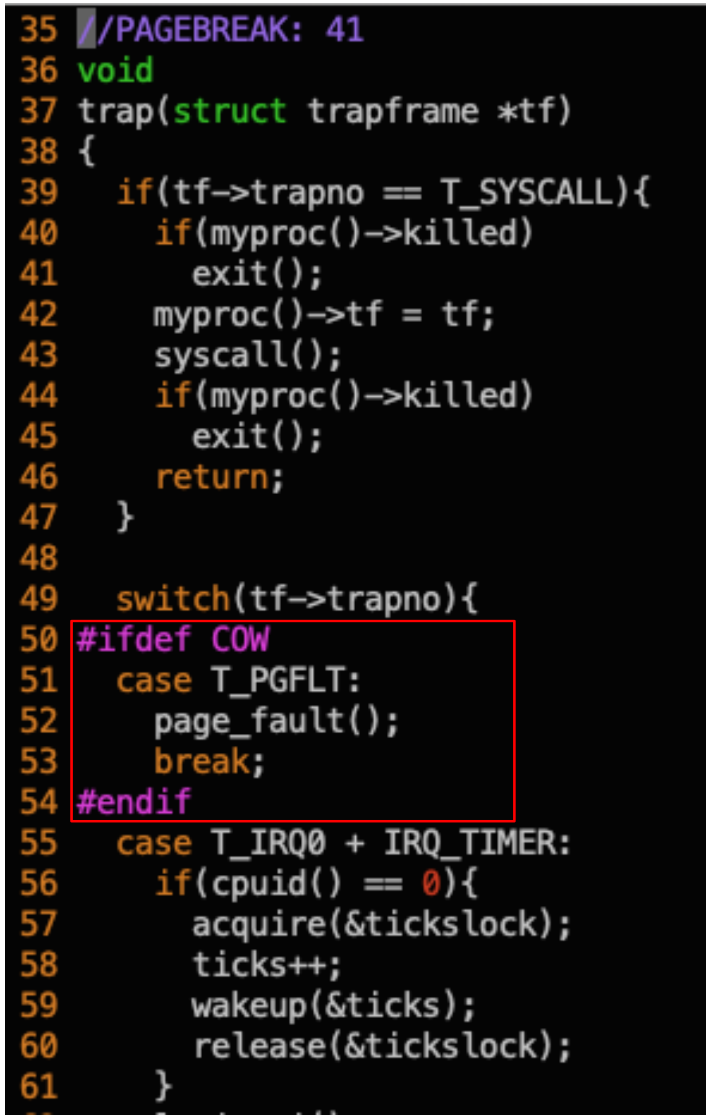


```c
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

# Modification – Page fault handler

- Add page fault handler in trap.c
- T_PGFLT occurs when memory access is invalid

trap.c

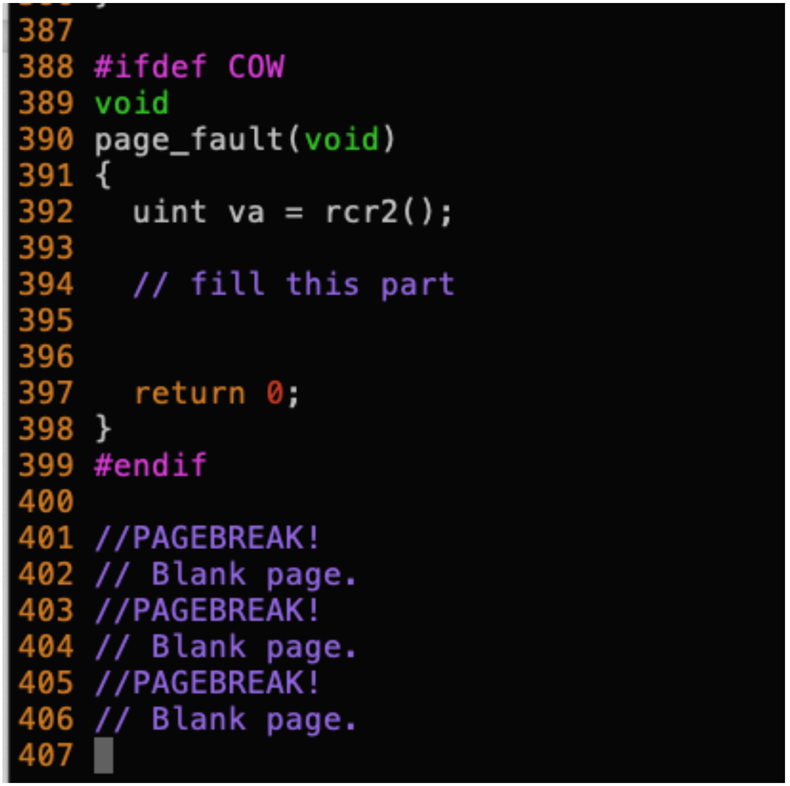


```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
#ifdef COW
  case T_PGFLT:
    page_fault();
    break;
#endif
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
```

# Modification – Page fault handler

- void page_fault(void)
  - `rcr2()`: return virtual address incurring page fault
  - check if the virtual address is valid
  - locate page table entry for the virtual address
    - `pte_t`
    - `walkpgdir()`
  - check the reference count of the physical address correspondin the virtual address
    - `get_refcounter()`
  - Perform copy-on-write
    - if reference counter > 2: allocate new page & copy
      - `kalloc()`, `memmove()`
      - decrease reference count for original page
    - Make pages writeable (update page table entry)
  - `lcr3(V2P(pgdir))`: TLB flush

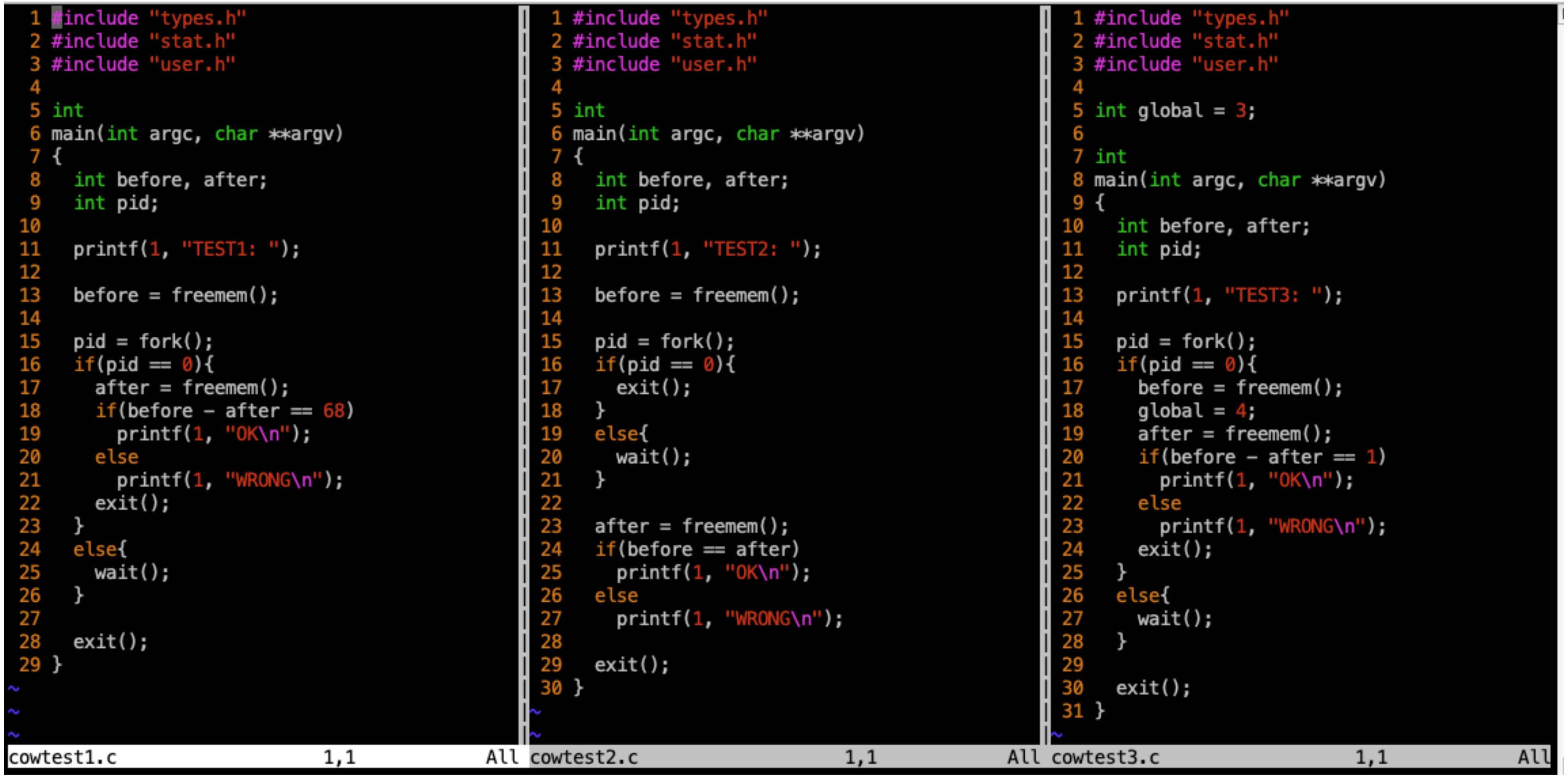
vm.c



```

#ifdef COW
void
page_fault(void)
{
  uint va = rcr2();

  // fill this part


  return 0;
}
#endif

//PAGEBREAK!
// Blank page.
//PAGEBREAK!
// Blank page.
//PAGEBREAK!
// Blank page.

```

# COW Test

- Three tests



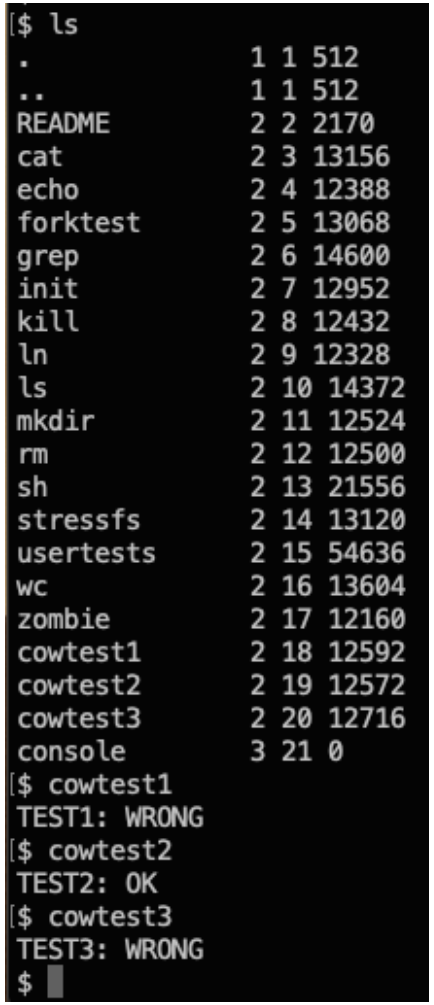
```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char **argv)
{
  int before, after;
  int pid;

  printf(1, "TEST1: ");

  before = freemem();

  pid = fork();
  if(pid == 0){
    after = freemem();
    if(before - after == 68)
      printf(1, "OK\n");
    else
      printf(1, "WRONG\n");
    exit();
  }
  else{
    wait();
  }

  exit();
}
~
~
~
cowtest1.c                1,1          All
```

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

int
main(int argc, char **argv)
{
  int before, after;
  int pid;

  printf(1, "TEST2: ");

  before = freemem();

  pid = fork();
  if(pid == 0){
    exit();
  }
  else{
    wait();
  }

  after = freemem();
  if(before == after)
    printf(1, "OK\n");
  else
    printf(1, "WRONG\n");

  exit();
}
~
~
cowtest2.c                1,1          All
```

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

int global = 3;

int
main(int argc, char **argv)
{
  int before, after;
  int pid;

  printf(1, "TEST3: ");

  pid = fork();
  if(pid == 0){
    before = freemem();
    global = 4;
    after = freemem();
    if(before - after == 1)
      printf(1, "OK\n");
    else
      printf(1, "WRONG\n");
    exit();
  }
  else{
    wait();
  }

  exit();
}
~
cowtest3.c                1,1          All
```

# COW Test

- All should be “OK”



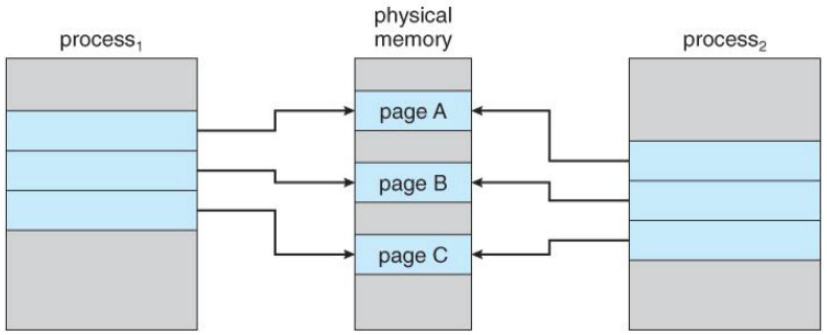
```
[$ ls
.              1 1 512
..             1 1 512
README         2 2 2170
cat            2 3 13156
echo           2 4 12388
forktest       2 5 13068
grep           2 6 14600
init           2 7 12952
kill           2 8 12432
ln             2 9 12328
ls             2 10 14372
mkdir          2 11 12524
rm             2 12 12500
sh             2 13 21556
stressfs       2 14 13120
usertests      2 15 54636
wc             2 16 13604
zombie         2 17 12160
cowtest1       2 18 12592
cowtest2       2 19 12572
cowtest3       2 20 12716
console        3 21 0
[$ cowtest1
TEST1: WRONG
[$ cowtest2
TEST2: OK
[$ cowtest3
TEST3: WRONG
$ 
```

# Copy-on-Write

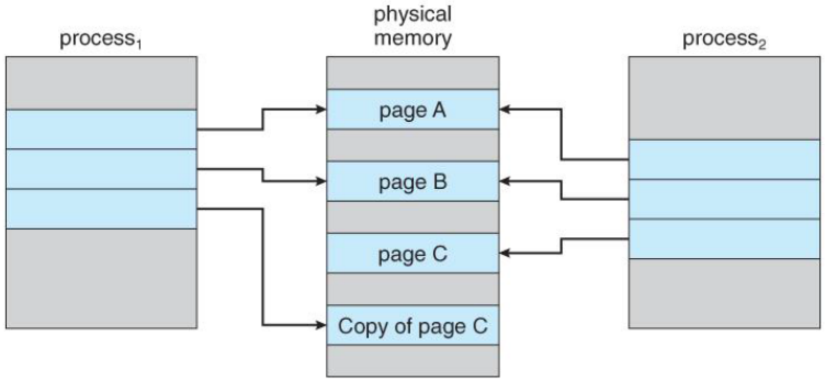
- When a process forks
  - Create shared mappings to the same page frames in physical page
  - Shared pages are protected as read-only

- When data is written to shared pages
  - Protection fault is generated
  - OS allocates new space in physical memory and directs the write to it

- Reference counter for physical pages is needed



Before process 1 modifies page C



After process 1 modifies page C