Copy on Write

OS Study Session #5

Some Reviews

Why OS?

- It's all about ***sharing resources between processes …***
- Key Requirements
  - Isolation
  - Multiplexing
  - Interaction

- What resources to share?
  - Memory -> This is what we mostly addressed so far
    - Virtual memory, Paging, …
  - CPU
    - Scheduling ..
  - I/O Device

Abstraction of Physical Resources

- **Virtual Address**
  - Giving an illusion that each processes has its own space
  - Should be mapped to valid **Physical Address**

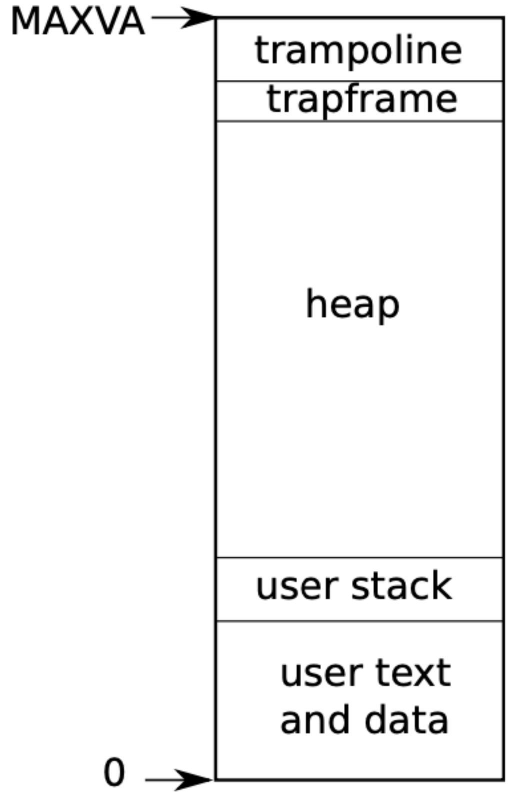
- **Address Translation**
  - OS translates VA to PA by its own rule
  - Uses Page Table (cf. Translation Lookaside Buffer)

- **Paging / Segmentation**
  - VA is broken into fixed-size pages or variable-size segments

Xv6 Processes and Virtual Address Space

- Process
  - A fundamental unit of job in OS
  - A unit of isolation

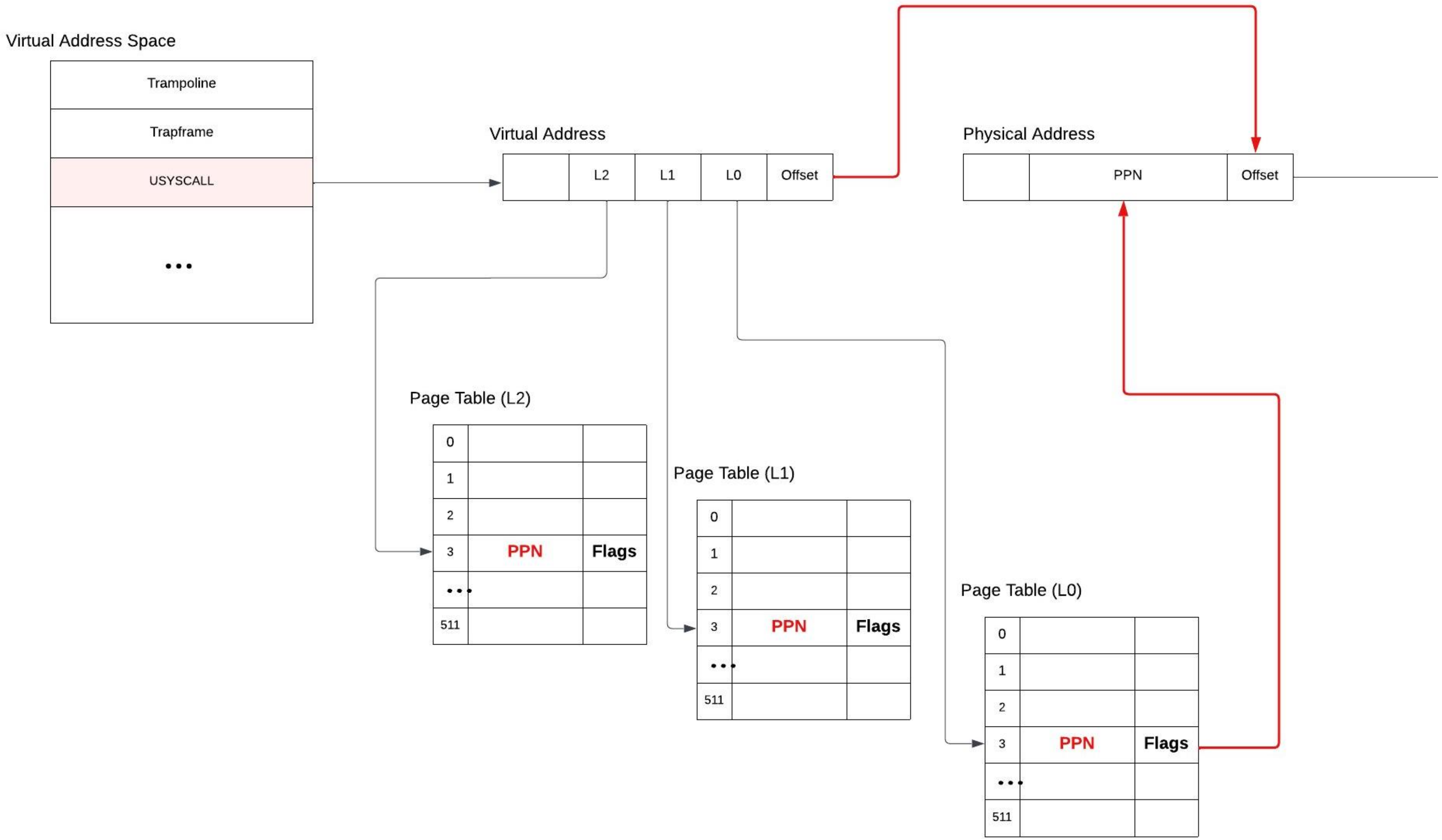
- Components
  - Code (Text Section): The program's executable instructions
  - Data Section: Global and static variables
  - Heap: Dynamic memory allocation
  - Stack: Function calls, local variables, return addresses.



Layout of Virtual Address Space

Page Table Structure

Full Picture



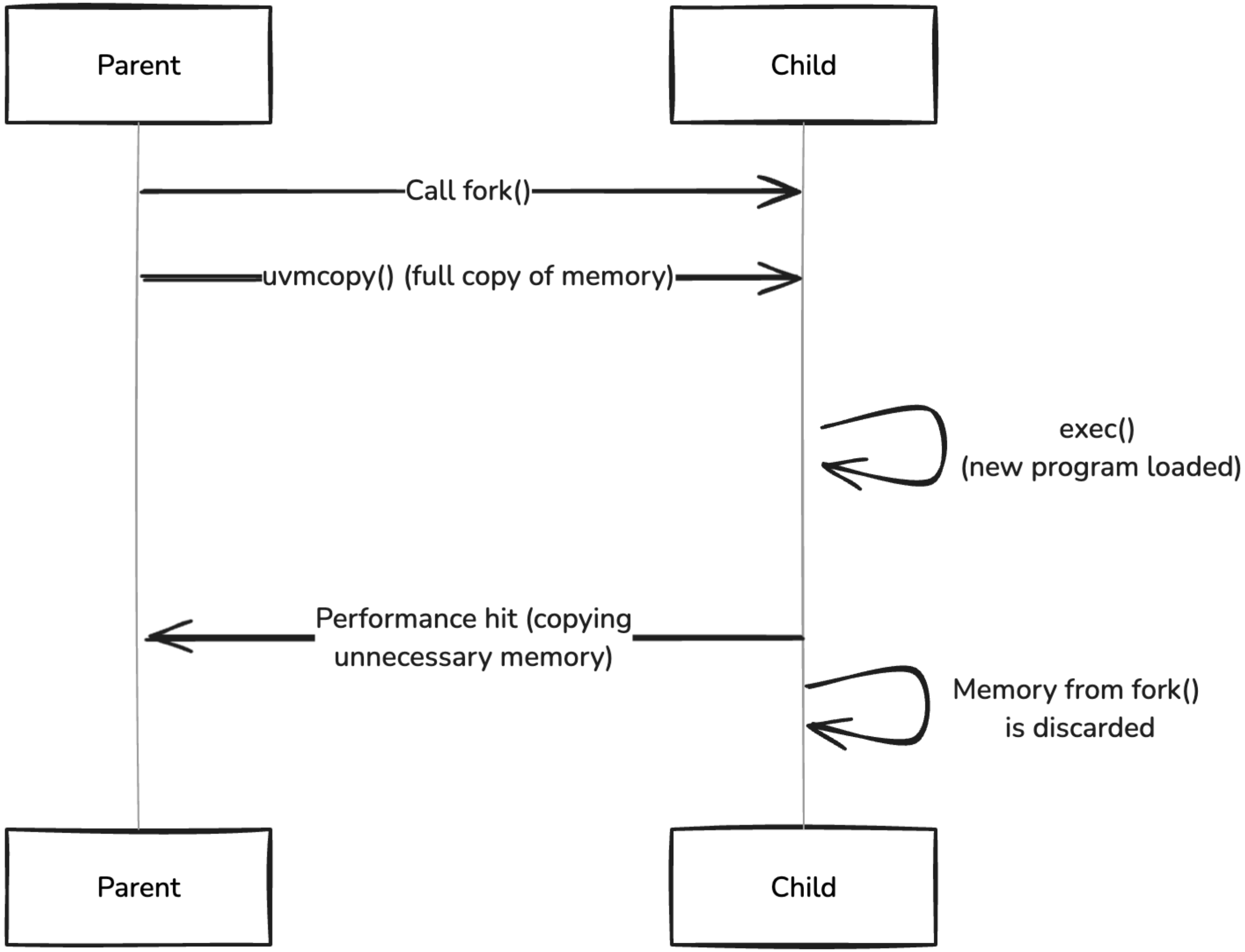
Copy-on-Write Concept

Basic fork() Behavior

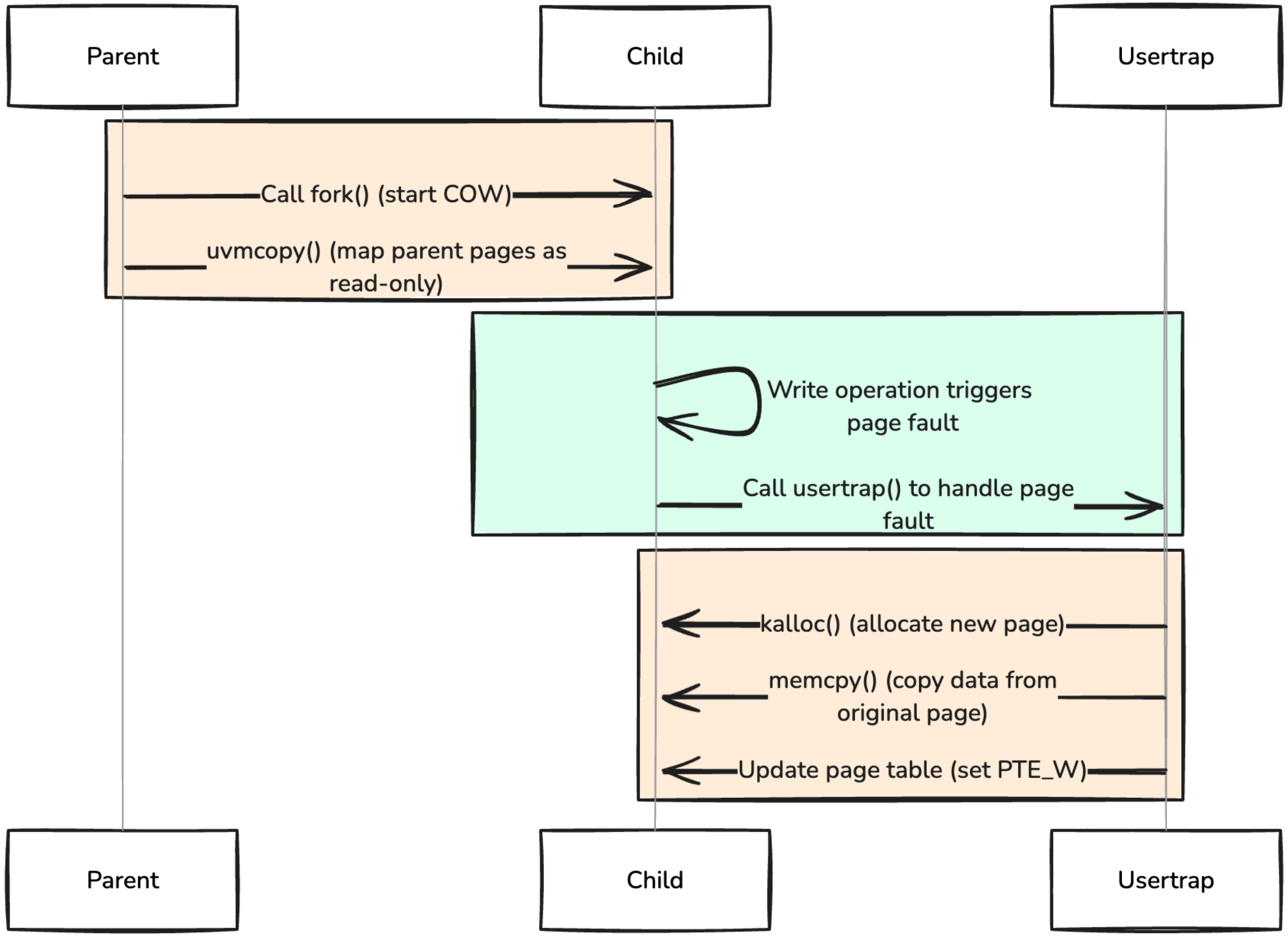
- Fork() creates a new process by duplicating the parent process's memory
- The entire memory of the parent is copied to the child process
- However, if the child immediately calls exec(), all that memory is discarded because the new program loaded by exec() does not use the memory that was copied.

→ Ucecessary Memory Duplication

Basic fork() Behavior



Copy-on-Write Overview



Key Functions for Implementing COW
- uvmcopy()

[Before COW]

- uvmcopy() performs a full memory copy of the parent process's memory
- All Pages of the parent parent are copied to the child

➡️ Independent copies of memory for parent and child

[After COW]

- uvmcopy() maps parent's memory pages to the child's page table without copying
- Pages are shared, marked as read-only

➡️ Pages are only copied when either process attempts to write.

Key Functions for Implementing COW
- usertrap()

[Before COW]

- usertrap() handles page faults caued by invalid memory access (e.g. accessing unmapped memory)

[After COW]

- usertrap() handles COW page faults and allocates a new page only when a write occurs on a shared, read-only page

Key Functions for Implementing COW
- copyout()

[Before COW]

- copyout() simply copies data from kernel space to user space

[After COW]

- copyout() ensures that data from a COW page is correctly tranceferred from kernel space to user space

Reference counting for Implementing COW

[Before COW]

- No reference counting is implemented, meaning pages are freed immediately after a process finishes using them

[After COW]

- Reference counting is introduced to track how many processes are using a page.
- kalloc() and kfree() are modified to support reference counting

Labs:
Implement copy-on-write fork

(hard)

Problem Summary

- the `fork()` system call currently copies all of the parent process's memory into the child process - inefficient
- Inefficient when `fork()` is followed by `exec()` which discards most of that copied memory

Implementation Tasks

1. Instead of copying all memory during fork, just create a page table for the child with PTEs pointing to the parent's physical pages
2. Mark all user PTEs in both parent and child as read-only
3. When either process tries to write to a page, it causes a page fault
4. The kernel handler then:
   - Allocates a new physical page
   - Copies the content from the original page
   - Updates the PTE to point to the new page with write permission
5. Implement reference counting for physical pages to free them only when no process references them

Implementation Tasks

1. Use RSW bits in RISC-V PTE to mark COW pages
2. Modify `uvmcopy()` to map parent's pages without copying and mark pages as read-only
3. Modify `usertrap()` to handle COW page faults
4. Implement reference counting for physical pages in `kalloc.c`
5. Modify `copyout()` to handle COW pages
6. Modify `kfree()` to free page only when no process references them

1. Use RSW bits in RISC-V PTE to mark COW pages

```
  #define PTE_FLAGS(pte) ((pte) & 0x3FF)
+ #define PTE_COW (1L << 8) // Copy-on-write page
```

- Add COW bit to PTE (riscv.h)
- Flag for marking if this page is COW or not
- Enables recognition between COW page fault and normal faults

2. Modify uvmcopy( ) to map parent's pages without copying and mark pages as read-only

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

  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 the page is writeable, mark it COW
    if(flags & PTE_W) {
      flags &= ~PTE_W;  // Clear writable bit
      flags |= PTE_COW; // Set COW bit
      *pte = PA2PTE(pa) | flags;
    }

    // Map the page in child's page table
    if(mappages(new, i, PGSIZE, pa, flags) != 0)
      goto err;

    incref((void*)pa);
  }
  return 0;

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

vm.c

Change the writable page to read-only
And set the COW bit

Point to same physical page
instead of copying

Increment page reference count

3. Modify `usertrap()` to handle COW page faults

```c
    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if(r_scause() == 15) { // Page fault on store
    uint64 va = r_stval();
    if(handle_cow_fault(p->pagetable, va) < 0) {
      printf("usertrap(): cow page fault failed\n");
      p->killed = 1;
    }
  } else {
    printf("usertrap(): unexpected scause %lx pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%lx stval=%lx\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(killed(p))
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

trap.c

If the page fault is store page fault (scause==15)

1. Check if the fault is from COW page
2. Call `handle_cow_fault()`

```c
int
handle_cow_fault(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  uint64 pa;
  uint flags;
  char *mem;

  if(va >= MAXVA) return -1;

  va = PGROUNDDOWN(va);
  pte = walk(pagetable, va, 0);

  if(pte == 0) return -1;
  if((*pte & PTE_V) == 0 || (*pte & PTE_U) == 0) return -1;
  if((*pte & PTE_COW) == 0) return -1;

  pa = PTE2PA(*pte);
  flags = PTE_FLAGS(*pte);

  // Allocate new page
  if((mem = kalloc()) == 0) return -1;

  // Copy old page to new page
  memmove(mem, (char*)pa, PGSIZE);

  // Map new page with write permission
  flags &= ~PTE_COW;  // Clear COW bit
  flags |= PTE_W;     // Set writable bit

  uvmunmap(pagetable, va, 1, 0);
  if(mappages(pagetable, va, PGSIZE, (uint64)mem, flags) != 0) {
    kfree(mem);
    return -1;
  }

  kfree((void*)pa);
  return 0;
}
```

- Check if va is valid
- Find PTE of the address in pagetable

- Check if PTE exists
- Check if PTE is valid & accessible from user mode
- Check if this page is actually COW

- Extract physical address, flag bits from PTE

- Allocate new physical memory
  - Fails if not enough memory
- Copy old page to new page

- Not a COW page anymore (writable)

- Remove old mapping
  - do_free=0 : don't release physical memory
- Map new page to va with updated flags
  - Release allocated memory and return error if mapping fails

4. Implement reference counting for physical pages in `kalloc.c`

```
+ // Reference count for each physical page
+ #define PA2IDX(pa) (((uint64)pa - KERNBASE) / PGSIZE)
+ #define MAX_PAGES ((PHYSTOP - KERNBASE) / PGSIZE)
+ struct { // Count reference for a page
+   struct spinlock lock;
+   int count[MAX_PAGES];
+ } ref_count;
```

```
  void
  kinit()
  {
    initlock(&kmem.lock, "kmem");
+   initlock(&ref_count.lock, "ref_count");
    freerange(end, (void*)PHYSTOP);
  }
```

```
+ // Increment reference count for a page
+ void
+ incref(void *pa)
+ {
+   acquire(&ref_count.lock);
+   ref_count.count[PA2IDX(pa)]++;
+   release(&ref_count.lock);
+ }
+
+ // Get reference count for a page
+ int
+ getref(void *pa)
+ {
+   int count;
+   acquire(&ref_count.lock);
+   count = ref_count.count[PA2IDX(pa)];
+   release(&ref_count.lock);
+   return count;
+ }
+
```

5. Modify copyout() to handle COW pages

Copy data from kernel space to user space.
Used in file-read or returning syscall result

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

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    if(va0 >= MAXVA)
      return -1;

    pte = walk(pagetable, va0, 0);
    if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0)
      return -1;

    // Handle COW page
    if((*pte & PTE_W) == 0 && (*pte & PTE_COW)) {
      if(handle_cow_fault(pagetable, va0) < 0)
        return -1;
      pte = walk(pagetable, va0, 0);
    }

    if((*pte & PTE_W) == 0)
      return -1;

    pa0 = PTE2PA(*pte);
    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;
}
```

- Find va in page table
- Return –1 if No PTE, invalid PTE, not user-accessible

If read-only COW page
- Call handle_cow_fault()
- Find the updated PTE pointing to a new page

- Get physical address from PTE
- Copy data

6. Modify `kfree()` to free page only when no process references them

```c
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  acquire(&ref_count.lock);
  if(--ref_count.count[PA2IDX(pa)] > 0) { // if (cnt == 0) release mem
    release(&ref_count.lock);
    return;
  }
  release(&ref_count.lock);

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

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

Release page if no process is referencing the page

- Cast to-be-released page into freelist struct
- Add released page to freelist