CS344 Assignment 4

Group-3, CSE

October 6, 2025

Team Members

- **230101024** (Baswa Dhanush Sai)
- **230101092** (Seella Mythresh Babu)
- **230101114** (Yeredla Koushik Reddy)
- **230101120** (Taes Padhihary)

Please find the Google Drive folder containing all the edited files here:

drive.google.com/drive/folders/1La-qcUDnzp5WVvYZ-ODVO9u200Yp8oiN?usp=sharing

Contents

---

1. Task 4.1: Copy-on-Write (COW) Fork

References used

R1 xiayingp.gitbook.io/build_a_os/labs/lab-5-copy-on-write-fork-for-xv6
R2 www.rose-hulman.edu/class/csse/csse332/2324b/labs/cow/

Synopsis of the problem The default xv6 `fork` system call copies all of the parent's user space memory into the child by creating the same number of new pages for the child as the parent. There are two major issues here: (1) if the parent is large, copying can be very time-consuming, and (2) copies often waste memory — in many cases neither the parent nor the child modifies a page, which means that *they could have shared the same physical memory all along*. A copy is only truly needed when one of the processes writes to a shared page. *Ref.:* [R1]

1.1. Implementation Strategy and Relevant Code Snippets

Our strategy will be to *not* allocate new pages until we actually need to write new stuff.

Mechanism

1. Copy-on-write fork creates just a pagetable for the child, with the PTEs pointing to the physical pages of the parent. We disable writing permission PTE_W for all PTEs in both the parent and child pagetable during fork (i.e. neither the parent nor the child can write to their pages after a fork system call). *Ref.:* [R2]

   To achieve this, we will make edits to the function uvmcopy in kernel/vm.c (fork uses it to allocate memory to the child).

   Another important place where CoW handling is required is the copyout function (defined in kernel/vm.c). This function is used whenever the kernel needs to copy data from kernel space into a user buffer (usually when returning results of some system calls). Under a copy-on-write fork, the user buffer's PTE may be marked read-only with PTE_COW (add this flag in kernel/riscv.h). If the kernel blindly writes to such a page through copyout, the process would crash because the page is not writable yet.

Listing 1. Add PTE_COW flag in kernel/riscv.h

```
// . . .
#define PTE_COW (1L << 8)
// . . .
```

Listing 2. Summary of edits in kernel/vm.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 (flags & PTE_W) {
      flags &= ~PTE_W; // clear write permission
        for both child and parent
      flags |= PTE_COW; // set the COW flag for
        both child and parent
      *pte = PA2PTE(pa) | flags;
    }

    // old strategy: always allocate new page
    // if((mem = kalloc()) == 0) goto err;
    // memmove(mem, (char*)pa, PGSIZE);

    // new strategy: don't allocate anything

    if(mappages(new, i, PGSIZE, pa, flags) != 0)
      goto err;
    inc_page_ref(pa);
  }
  return 0;

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

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

  while(len > 0){
    // . . .
    pte = walk(pagetable, va0, 0);

    // old logic:
    // // forbid copyout over read-only user text
       pages.
    // if((*pte & PTE_W) == 0) return -1;

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

    // new logic: handle COW pages separately
    if ((*pte & PTE_COW) != 0) {
      pa0 = PTE2PA(*pte);
      int flags = PTE_FLAGS(*pte);
      flags |= PTE_W; // add write permission
      flags &= ~PTE_COW; // clear COW flag

      char *new_page = kalloc();
      if (new_page == 0) return -1;
      memmove(new_page, (char*)pa0, PGSIZE);
        // move from old to new
      uvmunmap(pagetable, va0, 1, 0);
      dec_page_ref(pa0); // decrease reference
        count of the physical page
      if (mappages(pagetable, va0, PGSIZE, (
        uint64)new_page, flags | PTE_V) != 0) {
        panic("something is wrong in mappages.\n
        ");
      }
    // remap the old pa0 to new_page
    pa0 = walkaddr(pagetable, va0);
    if (pa0 == 0) return -1;
   }
   // . . .
  }
  return 0;
}
```

2. So, when either the parent or the child attempt to write a page, a page fault occurs. The kernel page fault handler (`scause==15` in `kernel/trap.c`) then needs to detect a CoW protected page (we add a new flag `PTE_COW` in `kernel/riscv.h`), allocate a page of physical memory for the faulting process, copy the original page into the new page, and modify the relevant PTE in the faulting process to refer to the new page, **this time with the PTE marked writeable**. When the page fault handler returns, the user process will be able to write its copy of the page. *Ref.:* [R2]

Listing 3. Summary of edits in `kernel/trap.c`

```
// . . .
if (r_scause() == 15) {
  // Task 4.1
  // page fault when we try to assign a page to
     the child
  // when a page fault occurs on COW page, use
     kalloc() to allocate new page,
  // copy the old page to the new page
  // and add the new page in the PTE with
     PTE_W permission

  uint64 va = PGROUNDDOWN(r_stval());
  pte_t *pte = walk(p->pagetable, va, 0);

  if (pte == 0) {
    printf("page not found\n");
    setkilled(p);
  } else if ((*pte & PTE_V) == 0 || (*pte &
     PTE_U) == 0 || (*pte & PTE_COW) == 0)
     setkilled(p);
  else {
    uint64 pa = PTE2PA(*pte);
    int flags = PTE_FLAGS(*pte);

    flags |= PTE_W;
    flags &= ~PTE_COW;

    int count = get_page_ref(pa);
    if (count > 1) {
      // more than one process is using the
      physical page
      // allocate a new page for the process
      char *new_page = kalloc();
      if (new_page == 0) {
        printf("kalloc failed\n");
        setkilled(p);
      }

      memmove(new_page, (char*)pa, PGSIZE);
      // move from old to new

      *pte = PA2PTE((uint64)new_page) | flags;
      dec_page_ref(pa); // decrease reference
      count of the old page
    } else {
      // only one process is using the physical
      page
      // just change the permission of the page to
      PTE_W
      *pte = PA2PTE(PTE2PA(*pte)) | flags;
    }
  }
}
// . . .
```

3. COW `fork` makes freeing of the physical pages that implement user memory a little trickier. A given physical page may be referred to by multiple processes' page tables and should be freed only when the last reference disappears. *Ref.:* [R1]

   To implement this correctly, we add the helper functions `void inc_page_ref (uint64 pa)` (a new page is now referencing page `pa`), `int dec_page_ref (uint64 pa)` (one less page is referencing `pa`; returns the updated count of referencing pages) and `int get_page_ref (uint64 pa)` (number of pages that are referencing page `pa`).

Now, in kernel/kalloc.c we modify the function kalloc to call inc_page_ref and kfree now decreases reference count of a page with dec_page_ref and marks a page as "free" only when we have get_page_ref(pa) == 0.

Listing 4. Summary of edits in kernel/kalloc.c

```
struct {
  struct spinlock lock;
  int ref_count[PHYSTOP/PGSIZE];
} page_refs;

void init_page_refs() {
  initlock(&page_refs.lock, "page_refs");
  for(int i = 0; i < PHYSTOP/PGSIZE; i++)
      page_refs.ref_count[i] = 0;
}

void inc_page_ref(uint64 pa) {
  if (pa>=PHYSTOP) return;
  acquire(&page_refs.lock);
  page_refs.ref_count[pa/PGSIZE]++;
  release(&page_refs.lock);
}

int dec_page_ref(uint64 pa) {
  if (pa>=PHYSTOP) return -1;
  int count;
  acquire(&page_refs.lock);
  if(page_refs.ref_count[pa/PGSIZE] > 0)
    page_refs.ref_count[pa/PGSIZE]--;
  count = page_refs.ref_count[pa/PGSIZE];
  release(&page_refs.lock);
  return count;
}

int get_page_ref(uint64 pa) {
  if (pa>=PHYSTOP) return -1;
  int count;
  acquire(&page_refs.lock);
  count = page_refs.ref_count[pa/PGSIZE];
  release(&page_refs.lock);
  return count;
}

void kinit() {
  initlock(&kmem.lock, "kmem");
  // Task 4.1
  init_page_refs();
  freerange(end, (void*)PHYSTOP);
}

void kfree(void *pa) {
  // . . .
  if(dec_page_ref((uint64)pa) > 0) return;
  // proceed to free the page
}

void *kalloc(void) {
  // . . .
  if(r) {
    memset((char*)r, 5, PGSIZE); // fill with junk
    // Task 4.1
    inc_page_ref((uint64)r);
  }
  // . . .
}
```

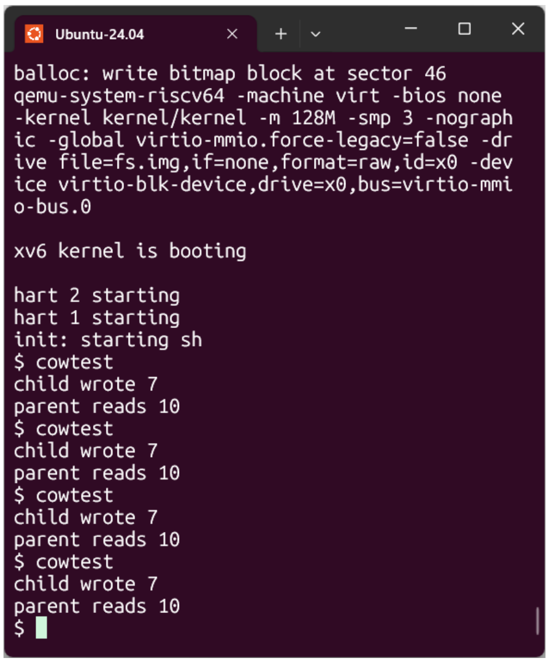
We will need to use the functions inc_page_ref, dec_page_ref and get_page_ref in other files and hence we will add their prototype to kernel/defs.h.

1.2. Testing Copy-on-write Fork

Listing 5 is a test program (whose format is as instructed in the task). Add $U/_cowtest to Makefile.

Listing 5. user/cowtest.c

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

#define PGSIZE 4096

int main(void){
  char *p = sbrk(PGSIZE);
  p[0] = 10;

  int pid = fork();
  if(pid < 0){ printf("fork failed\n"); exit(1); }
  if(pid == 0){
    // child: write should trigger COW copy
    p[0] = 7;
    printf("child wrote %d\n", p[0]); // 7
    exit(0);
  } else {
    wait(0);
    printf("parent reads %d\n", p[0]); // 10
    exit(0);
  }
}
```



```
balloc: write bitmap block at sector 46
qemu-system-riscv64 -machine virt -bios none
-kernel kernel/kernel -m 128M -smp 3 -nograph
ic -global virtio-mmio.force-legacy=false -dr
ive file=fs.img,if=none,format=raw,id=x0 -dev
ice virtio-blk-device,drive=x0,bus=virtio-mmi
o-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ cowtest
child wrote 7
parent reads 10
$ cowtest
child wrote 7
parent reads 10
$ cowtest
child wrote 7
parent reads 10
$ cowtest
child wrote 7
parent reads 10
$
```

2. Task 4.2: Demand Paging (MRU Page Replacement)

References used

R3 pages.cs.wisc.edu/ remzi/OSTEP/vm-beyondphys.pdf

2.1. Implementation Strategy and Relevant Code Snippets

Synopsis of the design We implement demand paging with a Most-Recently-Used (MRU) global replacement policy:

- xv6 on Sv39 exposes up to 2^{38} **bytes (256 GiB)** of virtual space for user.
- Typical xv6 QEMU config is **128 MiB**, i.e., **PHYSTOP/PGSIZE = 32,768** pages of **4 KiB** each.
- **MRU policy**: we maintain a global list of resident user pages ordered by last touch; we evict the head when needed.
- **Swapping**: a swapped page is tagged in software using PTE_S (bit 9) and a per-proc entry records where its data resides.

Relevant structures We note some structures that we use in this problem.

Listing 6. Structures used

```
#define SWAPFILE "/swapfile" // swap−space in
      xv6 that we created
#define MAX_SWAP_PAGES 512

struct mru_node { // node for doubly linked list
  struct spinlock lock;
  struct mru_node *next;
  struct mru_node *prev;
  uint64 va;
  pagetable_t pagetable;
  int pid;
};
```

```
struct { // double ended mru linked list
  struct spinlock lock;
  struct mru_node *head; // Head is Most Recently
      Used
  struct mru_node *tail;
  struct mru_node nodes[PHYSTOP / PGSIZE];
} mru_list;
```

```
struct {
  struct file *swap_file;
  struct spinlock lock;
  char bitmap[MAX_SWAP_PAGES / 8];
} swap_manager;
```

Mechanism

1. We track the number of faults, swap-ins, swap-outs inside `struct proc`, and wire two syscalls to observe this policy behavior at user-level.

Listing 7. Per-process counters and syscalls

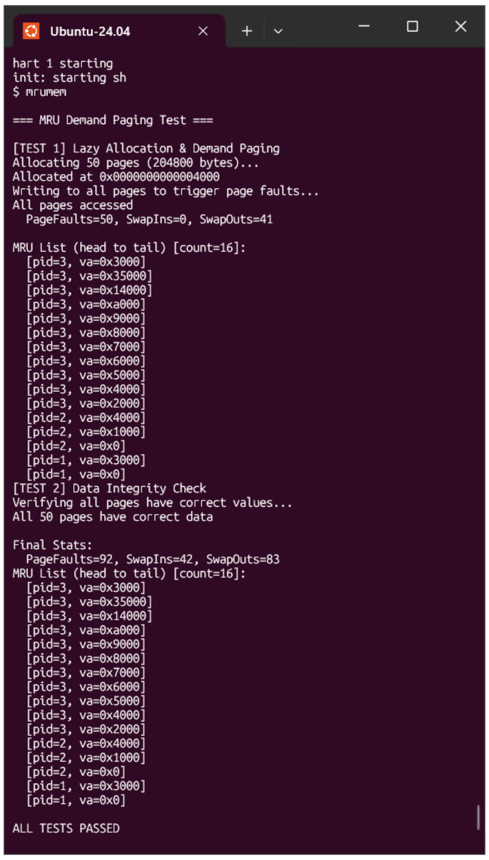
```
/* kernel/proc.h */
struct proc {
// . . .
// Task 4.2
  int page_faults; // Number of page faults
  int swap_ins; // Number of swap−ins
  int swap_outs; // Number of swap−outs
};

/* kernel/syscall.h */
#define SYS_getpagestat 22
#define SYS_dumpmru 23

/* kernel/syscall.c */
extern uint64 sys_getpagestat(void);
extern uint64 sys_dumpmru(void);
// . . .
[SYS_getpagestat] sys_getpagestat,
[SYS_dumpmru] sys_dumpmru,

/* kernel/sysproc.c */
struct pagestat { int page_faults, swap_ins,
      swap_outs; };

uint64 sys_getpagestat(void) {
  int pid; uint64 uaddr;
  argint(0,&pid); argaddr(1,&uaddr);
  struct proc p = findproc(pid); // helper function
      defined in proc.c to find the proc by pid
  if(!p) return −1;
  struct pagestat st = { p−>page_faults, p−>
      swap_ins, p−>swap_outs };
  release(&p−>lock);
  return copyout(myproc()−>pagetable, uaddr, (
      char)&st, sizeof st) ? −1 : 0;
}

uint64 sys_dumpmru(void) {
  dumpmru();
  return 0;
}

/* proc.c */
// helper function findproc
struct proc* findproc(int pid) {
  struct proc *p;
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p−>lock);
    if(p−>pid == pid && p−>state != UNUSED)
      return p;
    release(&p−>lock);
  }
  return 0;
}
```

- **`getpagestat`**: Report `page_faults`, `swap_ins`, `swap_outs` for a PID. Looks up the target proc under lock, reads the counter values and `copyout()`s the stats to user.
- **`dumpmru`**: Print the global MRU order (PID+VPN), head first. Traverses the list under lock; prints a small prefix for quick sanity checks.

2. We then create files `kernel/mru.h` and `kernel/mru.c` that define all the demand paging functions. Description of the functions are given here:
   - **`mru_update(uint64 pa, pagetable_t pagetable, uint64 va, int pid)`**: Update page metadata and move it to the head of MRU list (mark as most recently used).
     - Maps the physical address to corresponding `mru_node` and updates the node's metadata.
     - Removes node from current position in list. Inserts at head.
     - Updates head/tail pointers as needed. Called on page fault handling, page access.

- **mru_remove_node(struct mru_node *node)**: Remove a node from the MRU list.
  - Checks if node is valid and actually in the list.
  - Updates `prev->next` and `next->prev` pointers to bypass this node.
  - Removes the head and clears its prev and next pointers.
- **mru_free(uint64 pa)**: Remove a page from MRU tracking when its physical memory is freed. Sets PID to -1 to mark as unused. Called on `kfree` when releasing physical pages.
- **swap_out(char *page_to_swap, int *slot_out)**: Write a page to swap-space on disk.
  - Scans bitmap to find an unused swap slot. Sets corresponding bit in bitmap.
  - Uses `writei` to write page data at `slot * PGSIZE` offset.
  - Returns slot number via `slot_out` parameter for later retrieval.
- **swap_in(int slot, char *page_to_fill)**: Brings a page back from the swap file into physical memory. It reads the page from the correct slot in the swap file, marks the slot as free in the bitmap, and loads the data into RAM.
- **mru_evict**: It selects the most recently used page from the MRU list, removes it from the list, and swaps it out to disk to free physical memory. It updates the page table entry to mark the page as swapped, records the swap slot, and releases the physical page for reuse. This function enforces the MRU (Most Recently Used) page replacement policy by always evicting the page that was accessed most recently.

2.2. Testing Demand Paging



```
hart 1 starting
init: starting sh
$ mrumem

=== MRU Demand Paging Test ===

[TEST 1] Lazy Allocation & Demand Paging
Allocating 50 pages (204800 bytes)...
Allocated at 0x0000000000004000
Writing to all pages to trigger page faults...
All pages accessed
  PageFaults=50, SwapIns=0, SwapOuts=41

MRU List (head to tail) [count=16]:
  [pid=3, va=0x3000]
  [pid=3, va=0x35000]
  [pid=3, va=0x14000]
  [pid=3, va=0xa000]
  [pid=3, va=0x9000]
  [pid=3, va=0x8000]
  [pid=3, va=0x7000]
  [pid=3, va=0x6000]
  [pid=3, va=0x5000]
  [pid=3, va=0x4000]
  [pid=3, va=0x2000]
  [pid=2, va=0x4000]
  [pid=2, va=0x1000]
  [pid=2, va=0x0]
  [pid=1, va=0x3000]
  [pid=1, va=0x0]
[TEST 2] Data Integrity Check
Verifying all pages have correct values...
All 50 pages have correct data

Final Stats:
  PageFaults=92, SwapIns=42, SwapOuts=83
MRU List (head to tail) [count=16]:
  [pid=3, va=0x3000]
  [pid=3, va=0x35000]
  [pid=3, va=0x14000]
  [pid=3, va=0xa000]
  [pid=3, va=0x9000]
  [pid=3, va=0x8000]
  [pid=3, va=0x7000]
  [pid=3, va=0x6000]
  [pid=3, va=0x5000]
  [pid=3, va=0x4000]
  [pid=3, va=0x2000]
  [pid=2, va=0x4000]
  [pid=2, va=0x1000]
  [pid=2, va=0x0]
  [pid=1, va=0x3000]
  [pid=1, va=0x0]

ALL TESTS PASSED
```

Figure 1. Testing mrumem: All successful