Assignment 2
Copy on Write (COW)

AUTHORS

FNU Kshitij -SID -862548662- k002@ucr.edu
Vishnu Sujeetkumar Shukla -SID -862548226 - vshuk009@ucr.edu

2025-02-14

1 Step 1 - Installation

2 Step 1 - Modifications in Code

This section focuses on modifying the code to ensure compatibility with the provided test cases. We have made specific adjustments to the code flow and restructured the logic in certain files accordingly.

Intially we ran the code to see where were the error and what were the flaws in the code architecture , we finally lead with changes

2.1 Modifications

2.1.1 kernel/vm.c-Modifying fork() to Support Copy-On-Write:

Copy-on-Write (COW) was introduced by modifying the uvmcopy() function in vmċ. Instead of creating separate copies of memory pages during fork(), the system now allows both parent and child processes to initially share the same physical memory. This is achieved by iterating over the parentâs page table, extracting virtual addresses, and translating them into physical addresses. The key modification involves setting all writable pages to read-only by clearing the write permission bit. Additionally, a reserved software bit is set, marking these pages as COW-enabled.

With this change, both processes access shared pages in a read-only mode until either process attempts to modify a page as shown in Figure 1. At that moment, a page fault occurs, triggering the operating system to allocate a new physical page exclusively for the modifying process. This ensures that only the modified pages are duplicated, reducing overall memory consumption while maintaining process isolation.To complete the process, the modified page table entries are remapped using mappages(), ensuring that the child process references the same physical pages as the parent. This implementation significantly improves memory efficiency by delaying memory duplication until absolutely necessary, rather than preemptively copying all pages at the time of process creation.

After implementing the Copy-on-Write (COW) mechanism in the memory duplication process, further modifications were required to handle write operations correctly. The **copyout() function**, responsible for transferring data from kernel space to user space, needed adjustments to accommodate shared pages efficiently while preserving the COW behavior.To achieve this, the function first aligns the virtual address to the correct page boundary using a rounding macro. It then retrieves the corresponding page table entry to check if the memory is writable. If a write operation is attempted on a shared read-only page, a fault occurs, triggering the need for an exclusive copy.

At this point, a new physical page is allocated dynamically, ensuring that the process making the modification receives a separate copy of the data. The contents of the original page are then transferred to this new page before updating the page table entry to reflect the new ownership. The old shared

```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 (flags & PTE_W) {
      *pte &= ~PTE_W;*pte |= PTE_R;
      flags &= ~PTE_W;flags |= PTE_R;
    }


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


    acquire(&pa_ref_lock);
    pa_ref_count[pa/PGSIZE]++;;
    release(&pa_ref_lock);
  }
  return 0;

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

Figure 1: vm.c

page remains unchanged for other processes still referencing it.Additionally, the function ensures data integrity by preventing writes beyond page boundaries. It carefully calculates the amount of data to be copied based on page size constraints and the requested length. The memory transfer itself is performed using an optimized copying routine, ensuring efficiency while maintaining correctness.

These changes in Figure 2 collectively preserve process isolation while significantly reducing memory duplication overhead. By only allocating new pages when absolutely necessary, the system optimizes resource usage while maintaining expected program behavior, seamlessly integrating COW into the memory management workflow.

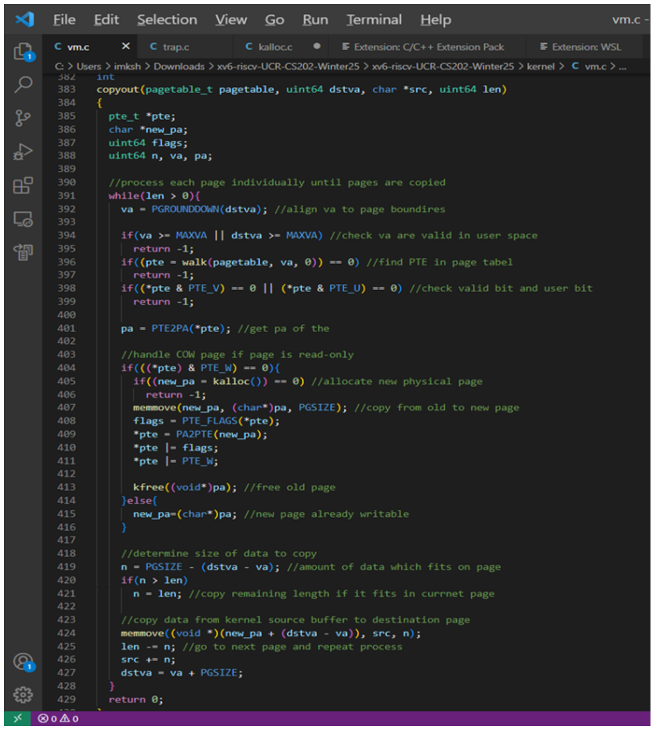


Figure 2: Copy Out Function

2.1.2 kernel/kalloc.c- Implementing Reference Counting

To efficiently manage shared physical memory among multiple processes, reference counting is implemented to track how many processes are using each memory frame. This prevents premature deallocation when a process exits while other processes still hold references to the same memory page. Each frameâs reference count is maintained in an array pa_ref_count[PHYSTOP/PGSIZE], ensuring accurate tracking.

In Figure 3 we talk about how During the fork() operation, when memory pages are shared between the parent and child process, the reference count is incremented to reflect the additional usage. Conversely, when a process terminates and releases its pages, the count is decremented. However, a page is only deallocated once its reference count drops to zero, ensuring no process loses access to shared data unexpectedly.

To maintain consistency in concurrent environments, a spinlock is used to protect the reference counter from race conditions . This spinlock ensures that reference updates occur atomically, preventing incorrect counts due to simultaneous modifications. The initialization of this mechanism occurs during system startup in kinit(), where all reference counters are set appropriately in freerange().Additionally, the kfree() function is modified to check the reference count before freeing a page. If the count is still above zero, the page remains in memory; otherwise, it is de allocated safely.

Similarly, kalloc() ensures that any newly allocated page starts with a reference count of 1, preventing immediate de allocation. These changes optimize memory usage, reducing redundant allocations while maintaining process isolation and efficient memory management.

```
pa_ref_count[(uint64)pa / PGSIZE]--;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");
 if (pa_ref_count[(uint64)pa / PGSIZE] == 0) {
   // Only free when count reaches 0
   release(&pa_ref_lock);

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

    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
} else
{
  release(&pa_ref_lock);
  }
}
}


/ Allocate one 4096-byte page of physical memory.
/ Returns a pointer that the kernel can use.
/ Returns 0 if the memory cannot be allocated.
oid *
alloc(void)
{
 struct run *r;

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

 if(r){
   memset((char*)r, 5, PGSIZE); // fill with junk

   //init counter to 1
   acquire(&pa_ref_lock);
   pa_ref_count[((uint64)r)/PGSIZE]=1;
   release(&pa_ref_lock);
 }

 return (void*)r;
}
```

Figure 3: Reference Counting

2.1.3 kernel/trap.c- Handling Page Faults in usertrap()

Handling Page Faults the changes is in Figure 4 When a process attempts to write to a read-only shared page, a page fault occurs, requiring proper handling to ensure data integrity and efficient memory management. By default, the usertraps() function in trap.c does not handle Copy-on-Write (CoW) page faults. To address this, additional logic was implemented to detect and process such faults dynamically.

The newly introduced mechanism checks whether the faulting page table entry (PTE) is valid and has the user-accessible bit set. If the page is marked as read-only (PTE_R but not PTE_W), it triggers a CoW fault, prompting the system to allocate a new writable physical page. The contents of the old read-only page are copied into this newly allocated page to maintain data consistency. The page table entry is then updated to reference the new page with write permissions enabled (PTE_W).

To prevent unnecessary deallocation of shared pages, the old physical page is freed only if it is no longer referenced by any process, leveraging the reference counting mechanism implemented in kfree(). If the page fault is unrelated to CoW, the system distinguishes between device-related faults and unexpected traps, ensuring proper exception handling. Unexpected faults are logged, and the process is terminated to maintain system stability.

This approach optimizes memory utilization by delaying page duplication until necessary, reducing redundant copies and improving performance in fork-heavy workloads. By integrating CoW handling into usertraps(), the system efficiently supports shared memory while maintaining isolation when modifications occur.

```c
    if((*pte & PTE_V) == 0 || (*pte & PTE_U) == 0){
      p->killed = 1;
      exit(-1);
    }

    if (!(*pte & PTE_R)) {
      printf("usertrap: page fault on non-COW page at %p\n", va);
      p->killed = 1;
      exit(-1);
    }


    oldPA = PTE2PA(*pte);
    if((newPage= kalloc())==0){
      p->killed = 1;
      exit(-1);
    }

    memmove(newPage, (char *)oldPA, PGSIZE);
    flags = PTE_FLAGS(*pte);
    *pte = PA2PTE((uint64)newPage);
    *pte |= flags;
    *pte |= PTE_W;

    kfree((void*)oldPA);

  }else if((which_dev = devintr()) != 0){
    // ok
  } 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;
  }

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


  if(which_dev == 2)
    yield();

  usertrapret();
}
```

Figure 4: Trap.c

UC RIVERSIDE UNIVERSITY OF CALIFORNIA
Marlan and Rosemary Bourns College of Engineering

2.1.4 kernel/proc.c (Modifying fork())

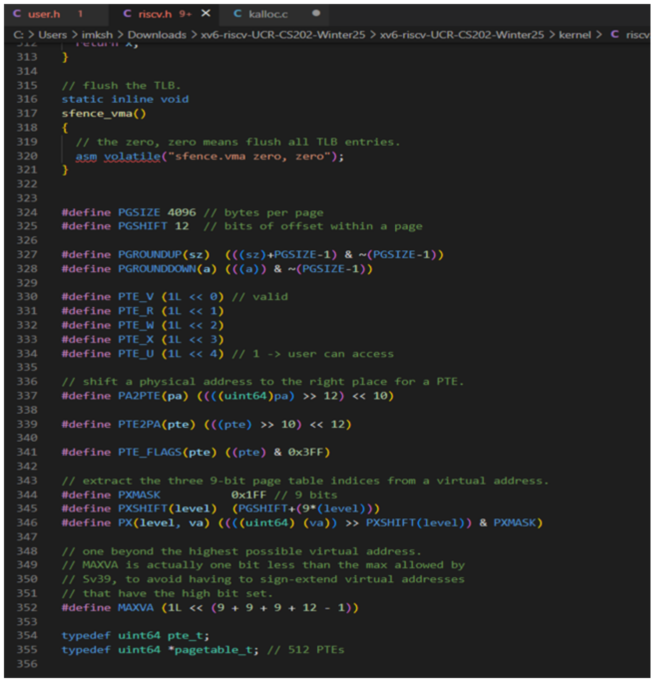
This file manages process creation and execution. The fork() system call, which previously copied all memory pages of the parent, was modified to implement COW instead. Figure 5 shows how we Instead of allocating new memory pages, we now share the parentâs pages with the child.Pages are marked read-only to trigger page faults on write attempts.

```c
// 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;

  np->parent = p;

  // copy saved user registers.
  *(np->tf) = *(p->tf);

  // Cause fork to return 0 in the child.
  np->tf->a0 = 0;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  np->state = RUNNABLE;

  release(&np->lock);

  return pid;
}

// Pass p's abandoned children to init.
```

Figure 5: proc.c

2.1.5 kernel/riscv.h (Defining COW Flags)

These were the changes that was done with riscv.h Figure 6



```
user.h 1    riscv.h 9+    kalloc.c
C: > Users > imksh > Downloads > xv6-riscv-UCR-CS202-Winter25 > xv6-riscv-UCR-CS202-Winter25 > kernel > riscv
  return x;
}

// flush the TLB.
static inline void
sfence_vma()
{
  // the zero, zero means flush all TLB entries.
  asm volatile("sfence.vma zero, zero");
}


#define PGSIZE 4096 // bytes per page
#define PGSHIFT 12  // bits of offset within a page

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access

// shift a physical address to the right place for a PTE.
#define PA2PTE(pa) ((((uint64)pa) >> 12) << 10)

#define PTE2PA(pte) (((pte) >> 10) << 12)

#define PTE_FLAGS(pte) ((pte) & 0x3FF)

// extract the three 9-bit page table indices from a virtual address.
#define PXMASK          0x1FF // 9 bits
#define PXSHIFT(level)  (PGSHIFT+(9*(level)))
#define PX(level, va) ((((uint64) (va)) >> PXSHIFT(level)) & PXMASK)

// one beyond the highest possible virtual address.
// MAXVA is actually one bit less than the max allowed by
// Sv39, to avoid having to sign-extend virtual addresses
// that have the high bit set.
#define MAXVA (1L << (9 + 9 + 9 + 12 - 1))

typedef uint64 pte_t;
typedef uint64 *pagetable_t; // 512 PTEs
```

Figure 6: riscv.h

2.1.6 kernel/defs.h (Adding Function Prototypes)

void incref(uint64); void decref(uint64);

2.1.7 user/user.h and user/usys.pl (Registering System Calls)

Added prototype for the modified system call Figure 7 and 8

Figure 7: usys.pl

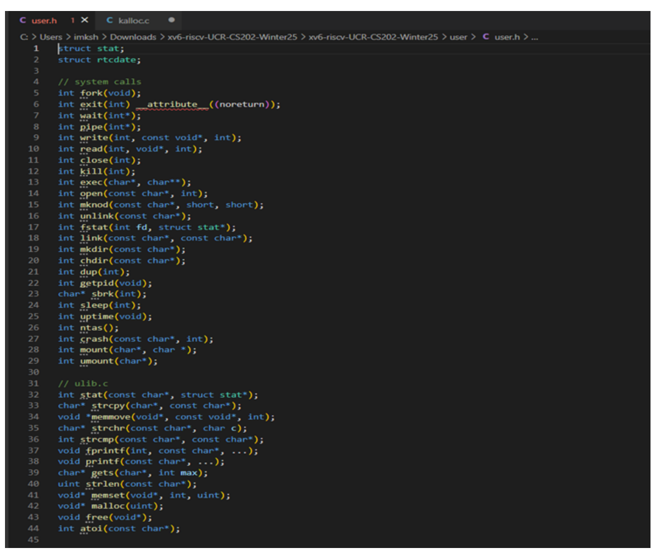


Figure 8: user.h

Marlan and Rosemary Bourns College of Engineering

3 Compiling and Results

Environment Setup and Execution Results To ensure proper compilation and testing, the environment was configured by updating the ḃashrc file to source cs202. Without this setup, the compilation process and the auto-grading script failed due to missing configurations.

During compilation, an issue arose in sh.c, where the runcmd() function was mistakenly flagged as causing infinite recursion. This was resolved by adding the __attribute__((noreturn)) directive, which informs the compiler that the function does not return, preventing misinterpretation of its execution flow.

Additionally, crucial debugging files,ġdbinit and .gdbinit.tmpl-riscv, were lost due to improper GitHub commits. Their absence caused failures when running make qemu-gdb and make grade. These files were restored to ensure smooth debugging and successful grading.

Execution and Results After setting up the environment and resolving the compilation errors, the system was successfully booted. Several tests were conducted to validate the correctness of Copy-on-Write (CoW) implementation.

cowtest: This test suite verified different scenarios, including simple, three, and file tests, all of which passed successfully. These tests ensure that CoW pages are properly allocated, copied, and modified as expected. **usertests:** The execution of usertests in user/usertests.c confirmed that all standard user-level tests passed without errors, demonstrating system stability and correctness. Final Evaluation: The last step involved running make grade, which executes all tests automatically. The grading script, grade-lab-cow, reported a perfect score of 100/100 as shown in the Figure 9, 10, 11, indicating a successful implementation of the Copy-on-Write feature. Overall, the modifications and debugging efforts led to a fully functional CoW implementation, passing all required tests while maintaining system efficiency and stability.

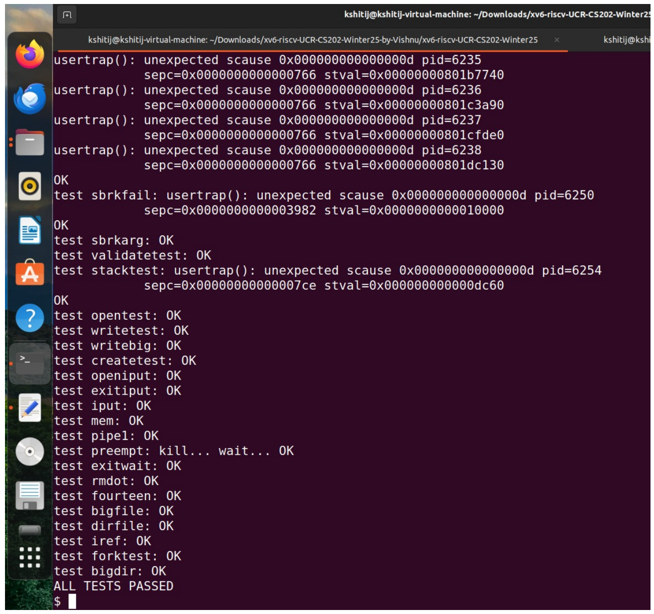


Figure 9: Usertests

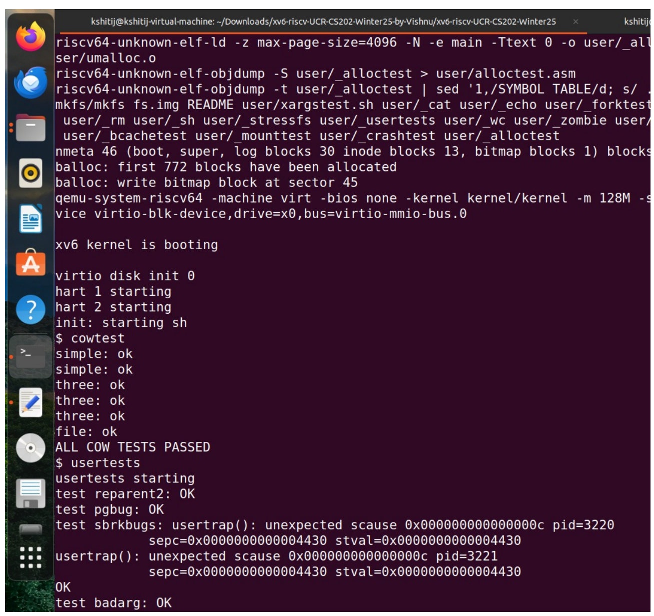


Figure 10: cowtest

UNIVERSITY OF CALIFORNIA
UC RIVERSIDE
Marlan and Rosemary Bourns
College of Engineering

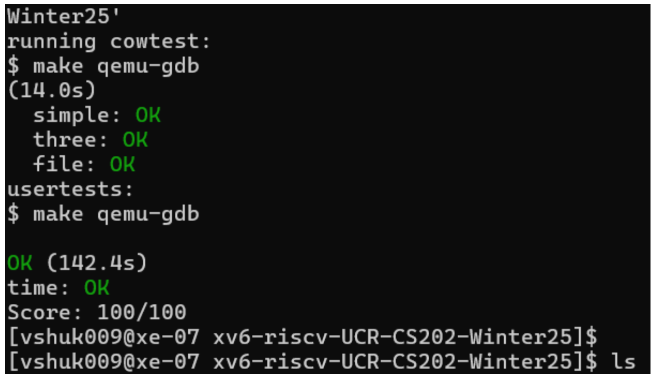


Figure 11: Make Grade File

4 Contribution Table

Contributor	File Name	Contribution Type
Kshitij	kernel/vm.c	Code Modification
	kernel/proc.c	Code Modification
	kernel/defs.h	Code Modification
	kernel/riscv.h	Code Modification
Vishnu	kernel/trap.c	Code Modification
	kernel/kalloc.c	Code Modification
	user/user.h	Code Modification
	user/usys.pl	Code Modification
Both	Report Compilation	Equal Contribution

Table 1: Contribution Table

5 Relevant Links

- Follow this link for the video: Click here
- Follow this link for the GitDiff and Git Repo: Click here for Video and Git Repo