



Marlan and Rosemary Bourns
College of Engineering

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.

Initially 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 `vm.c`. 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 parent's 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 `mmapages()`, 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

```

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;

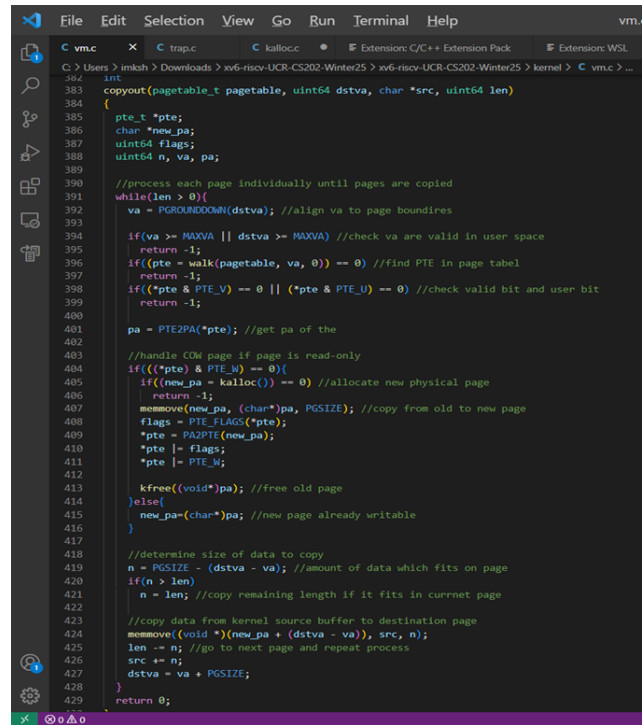
err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}

```

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.



```

383 copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
384 {
385     pte_t *pte;
386     char *new_pa;
387     uint64 flags;
388     uint64 n, va, pa;
389
390     //process each page individually until pages are copied
391     while(len > 0){
392         va = PGROUNDDOWN(dstva); //align va to page boundaries
393
394         if(va >= MAXVA || dstva >= MAXVA) //check va are valid in user space
395             return -1;
396         if((pte = walk(pagetable, va, 0)) == 0) //find PTE in page table
397             return -1;
398         if((*pte & PTE_V) == 0 || (*pte & PTE_U) == 0) //check valid bit and user bit
399             return -1;
400
401         pa = PTE2PA(*pte); //get pa of the
402
403         //handle COW page if page is read-only
404         if((*pte & PTE_W) == 0){
405             if((new_pa = kalloc()) == 0) //allocate new physical page
406                 return -1;
407             memmove(new_pa, (char*)pa, PGSIZE); //copy from old to new page
408             flags = PTE_FLAGS(*pte);
409             *pte = PA2PTE(new_pa);
410             *pte |= flags;
411             *pte |= PTE_W;
412
413             kfree((void*)pa); //free old page
414         }else{
415             new_pa = (char*)pa; //new page already writable
416         }
417
418         //determine size of data to copy
419         n = PGSIZE - (dstva - va); //amount of data which fits on page
420         if(n > len)
421             n = len; //copy remaining length if it fits in current page
422
423         //copy data from kernel source buffer to destination page
424         memmove((void *)new_pa + (dstva - va), src, n);
425         len -= n; //go to next page and repeat process
426         src += n;
427         dstva = va + PGSIZE;
428     }
429     return 0;

```

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 frame's 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 deallocated 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.
void *
kalloc(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.

```

    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

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 parent's pages with the child. Pages are marked read-only to trigger page faults on write attempts.

```

245 // Sets up child kernel stack to return as if from fork() system call.
246 int
247 fork(void)
248 {
249     int i, pid;
250     struct proc *np;
251     struct proc *p = myproc();
252
253     // Allocate process.
254     if((np = allocproc()) == 0){
255         return -1;
256     }
257
258     // Copy user memory from parent to child.
259     if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
260         freeproc(np);
261         release(&np->lock);
262         return -1;
263     }
264     np->sz = p->sz;
265
266     np->parent = p;
267
268     // copy saved user registers.
269     *(np->tf) = *(p->tf);
270
271     // Cause fork to return 0 in the child.
272     np->tf->a0 = 0;
273
274     // increment reference counts on open file descriptors.
275     for(i = 0; i < NOFILE; i++){
276         if(p->ofile[i]){
277             np->ofile[i] = filedup(p->ofile[i]);
278             np->cwd = idup(p->cwd);
279
280             safestrcpy(np->name, p->name, sizeof(p->name));
281
282             pid = np->pid;
283
284             np->state = RUNNABLE;
285
286             release(&np->lock);
287
288             return pid;
289         }
290     }
291     // 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

```

C user.h 1 C riscv.h 9+ X C kalloc
C > Users > imksh > Downloads > xv6-riscv-UCR-CS202-Winter25 > xv6-riscv-UCR-CS202-Winter25 > kernel > C riscv
313 }
314
315 // flush the TLB.
316 static inline void
317 sfence_vma()
318 {
319     // the zero, zero means flush all TLB entries.
320     asm volatile("sfence.vma zero, zero");
321 }
322
323
324 #define PGSIZE 4096 // bytes per page
325 #define PGSHIFT 12 // bits of offset within a page
326
327 #define PROUNDUP(x) (((x)+PGSIZE-1) & ~(PGSIZE-1))
328 #define PGROUNDDOWN(x) (((x)) & ~(PGSIZE-1))
329
330 #define PTE_V (1L << 0) // valid
331 #define PTE_R (1L << 1)
332 #define PTE_W (1L << 2)
333 #define PTE_X (1L << 3)
334 #define PTE_U (1L << 4) // 1 -> user can access
335
336 // shift a physical address to the right place for a PTE.
337 #define PA2PTE(pa) (((uint64)pa) >> 12) << 10)
338
339 #define PTE2PA(pte) (((pte) >> 10) << 12)
340
341 #define PTE_FLAGS(pte) ((pte) & 0x3FF)
342
343 // extract the three 9-bit page table indices from a virtual address.
344 #define PXMASK 0x1FF // 9 bits
345 #define PXSHIFT(level) (PGSHIFT+(9*(level)))
346 #define PX(level, va) (((uint64)(va)) >> PXSHIFT(level)) & PXMASK
347
348 // one beyond the highest possible virtual address.
349 // MAXVA is actually one bit less than the max allowed by
350 // Sv39, to avoid having to sign-extend virtual addresses
351 // that have the high bit set.
352 #define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
353
354 typedef uint64 pte_t;
355 typedef uint64 *pagetable_t; // 512 PTEs
356

```

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


```

    print "${name}:\n";
    print "  li a7, SYS_${name}\n";
    print "  ecall\n";
    print "  ret\n";
}

entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
entry("kill");
entry("exec");
entry("open");
entry("mknod");
entry("unlink");
entry("fstat");
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("ntas");
entry("crash");
entry("mount");
entry("umount");

```

Figure 7: usys.pl

```

C: userh 1 X C: kalloc
C: > userh > inish > Downloads > xv6-riscv-UCR-CS202-Winter25 > xv6-riscv-UCR-CS202-Winter25 > user > C: userh > ...
1 struct stat;
2 struct rtcdate;
3
4 // system calls
5 int fork(void);
6 int exit(int) __attribute__((noreturn));
7 int wait(int*);
8 int pipe(int*);
9 int write(int, const void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(const char*, int);
15 int mknod(const char*, short, short);
16 int unlink(const char*);
17 int fstat(int fd, struct stat*);
18 int link(const char*, const char*);
19 int mkdir(const char*);
20 int chdir(const char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int ntas();
27 int crash(const char*, int);
28 int mount(char*, char*);
29 int umount(char*);
30
31 // ulib.c
32 int stat(const char*, struct stat*);
33 char* strcpy(char*, const char*);
34 void *memcpy(void*, const void*, int);
35 char* strchr(const char*, char c);
36 int strcmp(const char*, const char*);
37 void printf(int, const char*, ...);
38 void printf(const char*, ...);
39 char* gets(char*, int max);
40 uint strlen(const char*);
41 void* memset(void*, int, uint);
42 void* malloc(uint);
43 void free(void*);
44 int atoi(const char*);
45

```

Figure 8: user.h

3 Compiling and Results

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

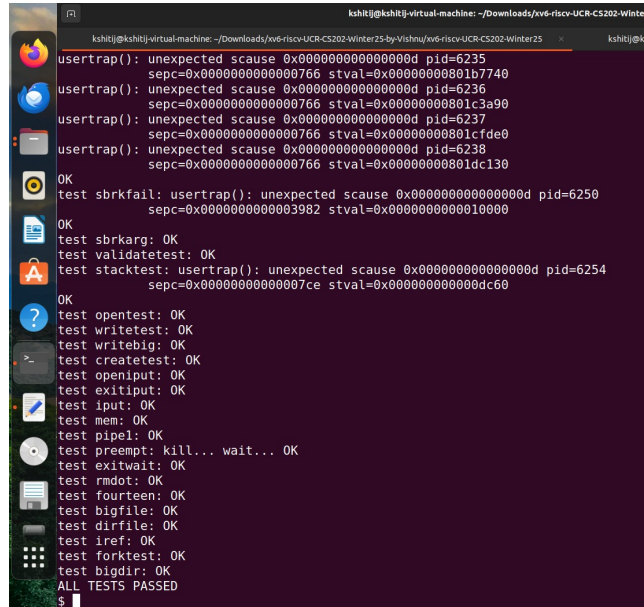
During compilation, an issue arose in `sh.c`, where the `runcmd()` function was mistakenly flagged as causing infinite recursion. This was resolved by adding the `__attribute__((noreturn))` directive, which informs the compiler that the function does not return, preventing misinterpretation of its execution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

Additionally, crucial debugging files, `.gdbinit` 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.

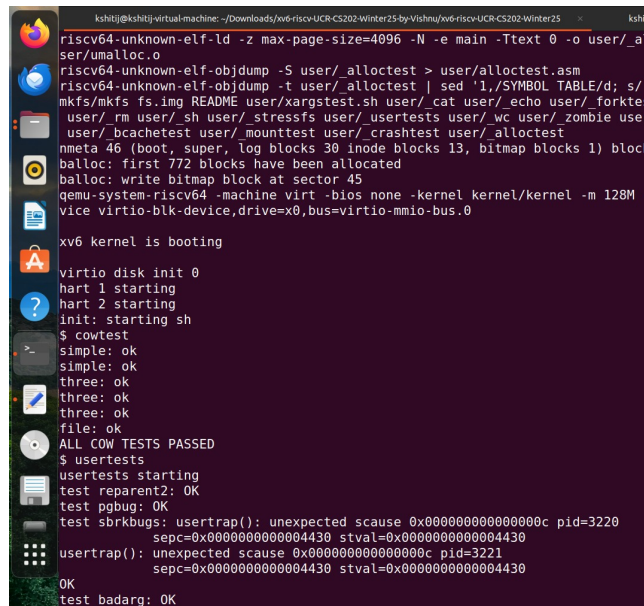
cowtest: This test suite verified different scenarios, including simple, three, and file tests, all of which passed successfully. These tests ensure that CoW pages are properly allocated, copied, and modified as expected. **usertests:** The execution of `usertests` in `user/usertests.c` confirmed that 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.

Assignment 1



```
kshltj@kshltj-virtual-machine: ~/Downloads/xv6-riscv-UCR-CS202-Winter25
usertrap(): unexpected scause 0x000000000000000d pid=6235
sepc=0x0000000000000766 stval=0x00000000001b7740
usertrap(): unexpected scause 0x000000000000000d pid=6236
sepc=0x0000000000000766 stval=0x00000000001c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6237
sepc=0x0000000000000766 stval=0x00000000001cfd0
usertrap(): unexpected scause 0x000000000000000d pid=6238
sepc=0x0000000000000766 stval=0x00000000001dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6250
sepc=0x0000000000003982 stval=0x000000000010000
OK
test sbrkarg: OK
test validate: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6254
sepc=0x00000000000007ce stval=0x000000000000dc60
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitinput: OK
test input: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test lref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

Figure 9: Usertests



```
kshltj@kshltj-virtual-machine: ~/Downloads/xv6-riscv-UCR-CS202-Winter25-by-Vishnu/xv6-riscv-UCR-CS202-Winter25
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_all
ser/umalloc.o
riscv64-unknown-elf-objdump -S user/_alloc.o > user/_alloc.o.asm
riscv64-unknown-elf-objdump -t user/_alloc.o | sed '1,/SYMBOL TABLE/d; s/ /
mkfs/mkfs fs.img README user/xargstest.sh user/_cat user/_echo user/_forktest
user/_rm user/_sh user/_stressfs user/_usertests user/_wc user/_zombie user/
user/_bcachetest user/_mounttest user/_crashtest user/_alloc.o
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks
ballocc: first 772 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -s
vice virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting
virtio disk init 0
hart 1 starting
hart 2 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
$ usertests
usertests starting
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3220
sepc=0x0000000000004430 stval=0x0000000000004430
usertrap(): unexpected scause 0x000000000000000c pid=3221
sepc=0x0000000000004430 stval=0x0000000000004430
OK
test badarg: OK
```

Figure 10: cowtest

```

Winter25'
running cowtest:
$ make qemu-gdb
(14.0s)
  simple: OK
  three: OK
  file: OK
usertests:
$ make qemu-gdb

OK (142.4s)
time: OK
Score: 100/100
[vshuk009@xe-07 xv6-riscv-UCR-CS202-Winter25]$
[vshuk009@xe-07 xv6-riscv-UCR-CS202-Winter25]$ ls

```

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](#)