# Copy On Write Forks

CS3500 Operating Systems Jul-Nov 2025

Sanjeev Subrahmaniyan S B

EE23B102

## 1 Objective

The objective of this lab is to add the COW(Copy On Write) feature to the xv6 operating system. COW forking allows forking to be done with little memory overhead of copying pages from parent to child. Most importantly, a large amount of the pages that are copied from the parent when fork replicated the memory image are usually freed and copying them is not efficient use of system resources.

COW forking combats this problem by permitting the child and the parent to share the same physical memory as long as neither of them write to it. When one of the processes write to the memory, a new physical page is allocated, remapped and written to.

## 2 Code

The sections of code that needed to be modified were specified in the handout for the assignment. Specifically, changes are made in the following files and a brief explanation of the changes most significant changes are provided.

### 2.1 Makefile

```
diff --git a/Makefile b/Makefile
index 39a99d7..d47337d 100644
--- a/Makefile
+++ b/Makefile
@@ -132,6 +132,7 @@ UPROGS=\
        $U/_grind\
        $U/_wc\
        $U/_zombie\
+       $U/_cowtest\
```

In the makefile, an entry is made for the cowtest.c file to be executed as a user program under the UPROGS section.

### 2.2 kernel/kalloc.c

```
diff --git a/kernel/kalloc.c b/kernel/kalloc.c
index 0699e7e..e426272 100644
--- a/kernel/kalloc.c
+++ b/kernel/kalloc.c
@@ -14,6 +14,8 @@ void freerange(void *pa_start, void *pa_end);
 extern char end[]; // first address after kernel.
                    // defined by kernel.ld.

+// the reference counts are initialised to zero because of the global array
+
```

```
 struct run {
   struct run *next;
 };
@@ -21,6 +23,7 @@ struct run {
 struct {
   struct spinlock lock;
   struct run *freelist;
+  uint64 ref_counts[PHYSTOP/PGSIZE]; // to hold the reference counts to each of
   the pages
 } kmem;

 void
@@ -36,7 +39,12 @@ freerange(void *pa_start, void *pa_end)
   char *p;
   p = (char*)PGROUNDUP((uint64)pa_start);
   for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
+  {
+    acquire(&kmem.lock);
+    kmem.ref_counts[(uint64)p/PGSIZE]=1;
+    release(&kmem.lock);
     kfree(p);
+  }
 }

 // Free the page of physical memory pointed at by pa,
@@ -47,10 +55,18 @@ void
+  {
+  {
+    acquire(&kmem.lock);
+    kmem.ref_counts[(uint64)p/PGSIZE]=1;
+    release(&kmem.lock);
     kfree(p);
+  }
 }

 // Free the page of physical memory pointed at by pa,
@@ -47,10 +55,18 @@ void
 kfree(void *pa)
 {
   struct run *r;
+  uint64 refcount;

   if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
     panic("kfree");
-
+
+  // check if there are no references to this physical memory page
+  acquire(&kmem.lock);
+  refcount = --kmem.ref_counts[(uint64)pa/PGSIZE];
+  release(&kmem.lock);
+
+  // free the page when there are no references
+  if(refcount == 0){
   // Fill with junk to catch dangling refs.
   memset(pa, 1, PGSIZE);

@@ -60,6 +76,7 @@ kfree(void *pa)
   r->next = kmem.freelist;
   kmem.freelist = r;
```

```
   release(&kmem.lock);
+  }
 }

 // Allocate one 4096-byte page of physical memory.
@@ -78,5 +95,12 @@ kalloc(void)

   if(r)
     memset((char*)r, 5, PGSIZE); // fill with junk
+
+  // initialise the ref_count for the page to 1 if valid
+  if(r){
+    acquire(&kmem.lock);
+    kmem.ref_counts[(uint64)r/PGSIZE] = 1;
+    release(&kmem.lock);
+  }
   return (void*)r;
 }
```

In this file, changes are made to accomplish the following. The changes appear in the same order as they appear in the code extract given above.

1. An array of size PHYSTOP/PGSIZE is created to hold the reference counts for each page of physical memory. This is maintained inside the kmem struct to be able to use the kmem.lock effectively and prevent synchronisation issues.

2. Before the freerange() is called for the first time during boot to setup the freelist, all reference count entries are set to a value of 1. This is for the kfree() function to be able to decrement and bring them to 0, which is the desired value for a page that is yet to be allocated. The setting of this value is guarded by the spinlock to prevent races.

3. The kfree() function is modified to decrement the refcount for that physical page everytime kfree() is called. This is done so because the free is only called by a process when it is going to remove the page from its page table and it needs to be unmapped. This requires decrementing the reference count for that page. After this, the usual kfree operations are performed if the last reference to that physical page was also removed. Note again that all of these actions are guarded by locks to prevent synchronisation issues.

### 2.3 kernel/riscv.h

```
diff --git a/kernel/riscv.h b/kernel/riscv.h
index 20a01db..27abf5e 100644
--- a/kernel/riscv.h
+++ b/kernel/riscv.h
@@ -343,6 +343,7 @@ typedef uint64 *pagetable_t; // 512 PTEs
 #define PTE_W (1L << 2)
 #define PTE_X (1L << 3)
 #define PTE_U (1L << 4) // user can access
+#define PTE_COW (1L << 9) // COW identifier
```

A macro is defined to specify the location of the COW bit in a page table entry. One of the RSW bits are used for this purpose.

### 2.4 kernel/trap.c

```
diff --git a/kernel/trap.c b/kernel/trap.c
index 512c850..5aec111 100644
--- a/kernel/trap.c
+++ b/kernel/trap.c
@@ -16,6 +16,12 @@ void kernelvec();

 extern int devintr();

+extern struct {
+  struct spinlock lock;
+  struct run *freelist;
+  uint64 refcounts[PHYSTOP/PGSIZE]; // to hold the reference counts to each of the
     pages
+} kmem;
+
 void
 trapinit(void)
 {
@@ -38,6 +44,11 @@ usertrap(void)
 {
   int which_dev = 0;

+  uint64 va, pa;
+  pte_t *pte;
+  char *mem;
+  uint flags;
+
   if((r_sstatus() & SSTATUS_SPP) != 0)
     panic("usertrap: not from user mode");

@@ -67,7 +78,50 @@ usertrap(void)
     syscall();
   } else if((which_dev = devintr()) != 0){
     // ok
+  } else if(r_scause() == 15){ // write page fault
+    // check if fault was COW page
+    va = r_stval(); // virtual address that caused page fault
+    va = PGROUNDDOWN(va); // find page of the corresponding page
+    if(va >= MAXVA){
+      setkilled(p);
+      exit(-1);
+    }
+    pte = walk(p->pagetable, va, 0); // find pte for faulting va
+    pa = PTE2PA(*pte); // save physical address of original page
+    if((*pte & PTE_COW) == 0){// write fault on non cow page
+      printf("write fault on non cow page\n");
+      goto err;
+    } else if((*pte & PTE_V) == 0){
+      printf("write fault on invalid pte entry\n");
+      goto err;
+    } else if((r_sstatus() == 0) && ((*pte & PTE_U) == 0) == 0){
+      printf("write fault with invalid permissions\n");
+      goto err;
+    } else {// fault on COW page
+      // allocate a new page
+      if((mem = kalloc()) == 0){
+        printf("write fault and kalloc failed\n");
+        goto err;
```

```
+      }
+      // copy the page on successful allocation
+      memmove((void *)mem, (void *)pa, PGSIZE);
+      // capture old flags
+      flags = PTE_FLAGS(*pte);
+      // set updated flags to have COW removed and W set
+      flags &= ~PTE_COW;
+      flags |= PTE_W;
+      *pte = *pte & ~PTE_V;
+      if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, flags) != 0){
+        printf("write fault and mapping failed\n");
+        goto err;
+      }
+      sfence_vma();
+
+      kfree((void*)pa);
+    }
+
   } else {
+  err:
     printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
     printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
     setkilled(p);
```

The trap handler code is modified to handle write page faults. The plan of action is to check if the page to be written to has appropriate permissions. Further, if the page is mapped copy on write, a new physical page is allocated for that virtual address(if more than one process are referring to the same physical memory). Otherwise, the same page is converted to a writable page by removing the COW flag and providing it the W flag.

### 2.5 kernel/vm.h

```
diff --git a/kernel/vm.c b/kernel/vm.c
index 5c31e87..a744a94 100644
--- a/kernel/vm.c
+++ b/kernel/vm.c
@@ -5,6 +5,7 @@
 #include "riscv.h"
 #include "defs.h"
 #include "fs.h"
+#include "spinlock.h"

 /*
  * the kernel's page table.
@@ -15,6 +16,12 @@ extern char etext[];  // kernel.ld sets this to end of kernel
   code.

 extern char trampoline[]; // trampoline.S

+extern struct {
+  struct spinlock lock;
+  struct run *freelist;
+  uint64 ref_counts[PHYSTOP/PGSIZE]; // to hold the reference counts to each of
   the pages
+} kmem;
+
 // Make a direct-map page table for the kernel.
```

```
 pagetable_t
 kvmmake(void)
@@ -315,7 +322,6 @@ 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((pte = walk(old, i, 0)) == 0)
@@ -324,12 +330,40 @@ uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
       panic("uvmcopy: page not present");
     pa = PTE2PA(*pte);
     flags = PTE_FLAGS(*pte);
-    if((mem = kalloc()) == 0)
-      goto err;
-    memmove(mem, (char*)pa, PGSIZE);
-    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
-      kfree(mem);
-      goto err;
+
+    // do not make a new page for child, just give it a reference
+    if((flags & PTE_U) == 0){// kernel pages, just copy and maintain reference
+      // initially i thought these need not hold reference counts
+      // but there could be multiple free problems
+      // if one process forks another and does not increment the ref count
+      // when the child exits, it will clean the kernel part and move on
+      // this would lead to corrupted values
+      if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0)
+        goto err;
+      acquire(&kmem.lock);
+      kmem.ref_counts[(uint64)pa/PGSIZE]++;
+      release(&kmem.lock);
+      continue;
+    } else if((flags & PTE_W) == 0){// non writable pages, just remap
+      if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0)
+        goto err;
+      acquire(&kmem.lock);
+      kmem.ref_counts[(uint64)pa/PGSIZE]++;
+      release(&kmem.lock);
+      continue;
+    } else {// writable user space pages, must be COW enabled
+      // the pages must have write perms removed and tagged COW
+      flags &= ~PTE_W;
+      flags |= PTE_COW;
+      // map for the child
+      if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0)
+        goto err;
+      // proceed only if child is successfully mapped COW
+      acquire(&kmem.lock);
+      kmem.ref_counts[(uint64)pa/PGSIZE]++;
+      release(&kmem.lock);
+      // update parent page table entry
+      *pte = PA2PTE(pa) | flags; // clear the old flags and set new ones
+        goto err;
+      // proceed only if child is successfully mapped COW
+      acquire(&kmem.lock);
+      kmem.ref_counts[(uint64)pa/PGSIZE]++;
+      release(&kmem.lock);
```

```
+      // update parent page table entry
+      *pte = PA2PTE(pa) | flags; // clear the old flags and set new ones
     }
   }
   return 0;
@@ -360,15 +394,36 @@ copyout(pagetable_t pagetable, uint64 dstva, char *src,
   uint64 len)
 {
   uint64 n, va0, pa0;
   pte_t *pte;
+  char *mem;
+  uint flags, new_flags;

   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 ||
-       (*pte & PTE_W) == 0)
-      return -1;
+    if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0)
+      return -1; // removed return when write is not enabled
+    if((*pte & PTE_COW) != 0){// recognise and handle COW page
+      // allocate a new page to store the contents of the target COW page
+      pa0 = PTE2PA(*pte); // preserve original physical address
+      if((mem = kalloc()) == 0)
+        return -1;
+      memmove((void *)mem, (void *)pa0, PGSIZE); // copy the contents to a new
   page
+      flags = PTE_FLAGS(*pte); // extract the flags of the COW page
+      // new page must be installed with PTE_W set and PTE_COW cleared
+      new_flags = flags;
+      new_flags &= ~PTE_COW;
+      new_flags |= PTE_W;
+      // call mappages to map the new physical page
+      // invalidate previous entry to prevent panic on remap
+      *pte = *pte & ~PTE_V;
+      if(mappages(pagetable, va0, PGSIZE, (uint64)mem, new_flags) != 0)
+        return -1;
+      // original page refcount must be decremented
+      sfence_vma();
+      kfree((void *)pa0);
+    } else if((*pte & PTE_W) == 0) return -1; // not COW and not writable
     pa0 = PTE2PA(*pte);
     n = PGSIZE - (dstva - va0);
     if(n > len)
```

The vm.c file contains two major changes: for the uvmcopy() and the copyout() function.

For the uvmcopy() function, the changes made are to set all pages to be copy on write and map the same physical memory in the child as well. At the same time, the parent's reference is also modified to copy on write. Note the appropriate handling of the reference counts with guarded locks.

For the copyout() function, the handling is similar to how the page faults are handled in the trap function. The idea is to check if the page is mapped copy on write, and if so, a new physical page is allocated and the page table mappings are updated. The parent's page table entries are also appropriately modified and the reference counts for the page is decremented through a call to the kfree() function.

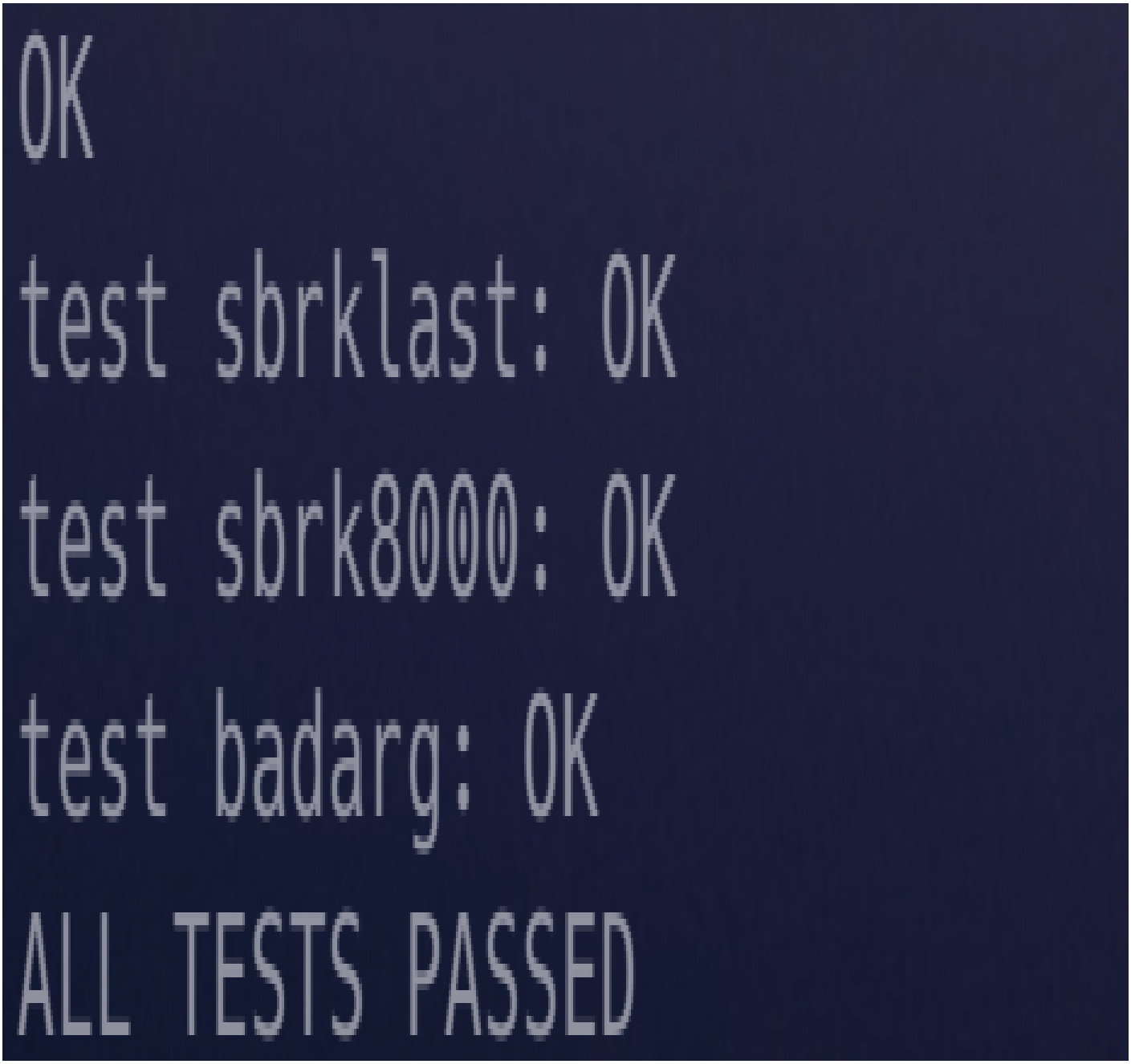
## 3 Execution and Output

The changes are tested with the cowtest.c file provided as part of the assignment along with usertests -q to see if the changes broke any other part of the OS. The output when cowtest is run in a docker container is:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
forkfork: ok
ALL COW TESTS PASSED
```

Similarly, when the usertests are run, the following output is seen, indicating correct functioning of the code.


OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED