OS project 3 report

Pooya sharifi 9831321
ALi babaloo 9831322

******important:before starting this project, we took some measurements in order to improve upon our previous version of the scheduler of the second part of the project.
This is the results for the previous part of the project:

```
found the child with stime of 100
going to wait - found the child with rutime of 1
found the child with ttime of 384
found the child with stime of 100
going to wait - found the child with rutime of 4
found the child with ttime of 385
found the child with stime of 100
going to wait - found the child with rutime of 3
found the child with ttime of 387
found the child with stime of 100
going to wait - found the child with rutime of 2
found the child with ttime of 388
found the child with stime of 100
going to wait - found the child with rutime of 4
found the child with ttime of 390
found the child with stime of 100
going to wait - found the child with rutime of 2
found the child with ttime of 390
found the child with stime of 100
going to wait - found the child with rutime of 3
found the child with ttime of 392
found the child with stime of 100
going to wait - found the child with rutime of 3
found the child with ttime of 392
found the child with stime of 100
going to wait - found the child with rutime of 3
found the child with ttime of 395
found the child with stime of 100
going to wait - found the child with rutime of 3
found the child with ttime of 395
found the child with stime of 100
going to wait - found the child with rutime of 2
found the child with ttime of 395
found the child with stime of 100
going to wait - found the child with rutime of 2
found the child with ttime of 397
found the child with stime of 100
going to wait - found the child with rutime of 2
found the child with ttime of 398
found the child with stime of 100
going to wait - found the child with rutime of 4
found the child with ttime of 398
found the child with stime of 100
going to wait - found the child with rutime of 1
found the child with ttime of 400
found the child with stime of 100
going to wait - found the child with rutime of 4
found the child with ttime of 401
found the child with stime of 100
going to wait - found the child with rutime of 2
found the child with ttime of 401
found the child with stime of 100
going to wait - found the child with rutime of 2
found the child with ttime of 402
found the child with stime of 100
```

Third part of the project:

So in this project we are trying to implement the “Copy on write” feature while forking. First to do so we have to add a variable to check how many processes are pointing to a certain page.So we don’t accidentally remove a page while deallocating one process while another process is using that page.

Thus, we add this struct to kalloc.c file :

```
static struct{
//a lock and an array for each page reference count
 struct spinlock lock;
 int cnt[REF_CNT_IDX(PHYSTOP)];
```

```
}kref;
```

And then we define a couple of functions to make our coding easier:

```
void kref_lock(){
 acquire(&kref.lock);
}

void kref_unlock(){
 release(&kref.lock);
}

static void set_refcnt (uint64 pa, int cnt){
 kref.cnt[REF_CNT_IDX(pa)] = cnt;
}

uint64 inc_refcnt (uint64 pa){
 return ++kref.cnt [REF_CNT_IDX(pa)];
}

uint64 dec_refcnt(uint64 pa){
 return --kref.cnt [REF_CNT_IDX(pa)];
}

void ref_init_cnt() {

 // Initialize the reference count array
 for (int i = 0; i < REF_CNT_IDX(PHYSTOP); i++) {
     kref.cnt[i] = 0;
 }
}
```

Then while initializing we need also initialize this variable and its locks:

```
// // Initialize the lock
 initlock(&kref.lock, "kref");
 ref_init_cnt();
```

acquire kref.lock,decrement kref.cnt for *pa and the if not zero release kref.lock and return.

```
 kref_lock();
 dec_refcnt((uint64)pa);
 //if not zero, release kref.lock and return
 if(kref.cnt[REF_CNT_IDX((uint64)pa)]>0){
   kref_unlock();
   // printf("%d",kref.cnt[REF_CNT_IDX((uint64)pa)]);
   return ;
 }
 kref_unlock();
```

Then we update kalloc so that when a page is allocated we set kref as 1 because the page is only being used by one process.

```
void *
kalloc(void)
{
 struct run *r;

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

 if(r){
 //if r != NULL then set reference count to one
   kref_lock();
   set_refcnt((uint64)r,1);
   // printf("allocated %d",kref.cnt[REF_CNT_IDX((uint64)r)]);
   kref_unlock();
   memset((char*)r, 5, PGSIZE); // fill with junk
 }
 return (void*)r;
}
```

The next thing we have to do is to add a COW header in the page table In RISCV.h file

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

Okay so the most important part when we fork we do the correct thing for COW.

But the fork code itself doesn't need to be changed, it's the uvmcopy() function in vm.c file we need to fix:
We first check that the page is writable, if yes,then it means that the page hasn't been allocated by any other process thus we need to set the COW flag to true and make the page unwritable. This is so that no process can write on this page.Because they are using the same page.

```
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((pte = walk(old, i, 0)) == 0)
     panic("uvmcopy: pte should exist");
   if((*pte & PTE_V) == 0)
     panic("uvmcopy: page not present");

 // if page is writable
 // make it unwritable (so we get pagefaults)
 // set the PTE_COW flag
 //don't copy the pages, just map them

   pa = PTE2PA(*pte);
   *pte &= ~PTE_W;
   flags = PTE_FLAGS(*pte);
   // if((mem = kalloc()) == 0)
   //   goto err;
   // memmove(mem, (char*)pa, PGSIZE);

   if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
     // kfree(mem);
     goto err;
   }

   kref_lock();
   inc_refcnt(pa);
   kref_unlock();
```

```
 }
 return 0;

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

Now imagine if two process are using a page and they want to write on it, how can we manage this?
We manage this by changing the page fault code so that it knows to allocate new page when there is a page fault and the page isn't writable and the flag of cow is on.

The page fault error is 0x00000000f, thats why we need to change the if(r_cause=15)

```
else if(r_scause() == 15){ //page fault
   if(killed(p))
     exit(-1);

   if(cowtrap(r_stval())<0)
     setkilled(p);
```

We need to implement cowTrap function:
First we check that va bigger than max virtual address check
Then we get the process page table entry
After checking the flags so that we know that this page fault is definitely from a fork COW
Then we make a new page and copy the content, and set the flags to the correct amounts.

```
int cowtrap(uint64 va){
 // va bigger than max virtual address check
 if(va > MAXVA)
   // exit(-1);
   return -1;
 // get the process and pte
 struct proc *p = myproc();
 pte_t *pte = walk(p->pagetable, va, 0);
 if (pte == 0)
   // exit(-1);
   return -1;
```

```
 // Check if it's valid?
 if ((*pte & PTE_U) == 0 || (*pte & PTE_V) == 0)
   return -1;

 // get pte flag and check if PTE_COW is set or not (not sure the code is
correct or not)
 // Unset PTE_COW and set PTE_W
 uint flags = PTE_FLAGS(*pte);
 if ((flags &= ~PTE_COW) == 0){
   printf("\ntermal_____________");
   exit(-1);
   return -1;
 }
 flags |= PTE_W;
 // make new page
 uint64 pa1 = PTE2PA(*pte);
 uint64 pa2 = (uint64)kalloc();
 if (pa2 == 0)
   return -1;

 // move older to new
 memmove((void *)pa2, (void *)pa1, PGSIZE);
 *pte = PA2PTE(pa2) | PTE_U | PTE_V | PTE_W | PTE_X | PTE_R;

 //free old pte - not sure about dec_refcnt part
 // dec_refcnt(pa1);
 kfree((void *)pa1);
 // printf("Check \n ");
 return 0;
}
```

Then we make the test file:

```
#include "../kernel/types.h"
#include "../kernel/memlayout.h"
#include "../user/user.h"
#include "../kernel/riscv.h"

void
simple_test()
```

```
{
int total_pages=(PHYSTOP - KERNBASE) / PGSIZE;

int num_allocated_pages = (2 * total_pages) / 3;

char *ptr = sbrk(0);  // Get the current break (end of the data segment)
for (int i = 0; i < num_allocated_pages; i++) {
   if (sbrk(PGSIZE) == (char*)-1) {
       // Error handling if sbrk fails
       printf("sbrk failed\n");
       // Additional cleanup code, if necessary
       break;
   }

   // Write something to the allocated page
   strcpy(ptr, "Hello, page!");

   // Move the pointer to the next page
   ptr += PGSIZE;
}

int pid=fork();
if (pid<0){
   printf("fork failed\n");
   exit(-1);
}
if(pid == 0){
   exit(0);
}
wait(0);


char* current_brk = sbrk(0);  // Get the current break (end of the data
segment)

for (int i = 0; i < num_allocated_pages; i++) {
   if (sbrk(-PGSIZE) == (char*)-1) {
       // Error handling if sbrk fails
       printf("sbrk failed\n");
       // Additional cleanup code, if necessary
```

```
        break;
    }
}

// Verify if the program break pointer has moved back
char* new_brk = sbrk(0);
if (new_brk < current_brk) {
    printf("Memory deallocated successfully\n");
} else {
    // Handle deallocation failure, if necessary
    printf("Memory deallocation failed\n");
}


printf("okay\n");
}

void main(void){

    simple_test();
}
```

Using sbrk() function we allocate ⅔ of the memory.
We also write something on these pages so we make sure its being used.
Then we fork()
Normally this should not have been possible since xv6 doesn't let you allocate that much memory.
But since we have **correctly** implemented the COW then it allows us to do so.

“Memory deallocated
Okay”

Conclusion

In conclusion, the implementation of Copy-On-Write (COW) in the xv6 operating system has proven to be a valuable addition, providing numerous benefits in terms of memory efficiency and process management. COW allows multiple processes to share memory pages until a modification occurs, reducing memory duplication and improving system performance.

By leveraging the COW mechanism, xv6 optimizes the creation and forking of processes. Instead of immediately duplicating the entire memory space of a parent process during a fork, COW allows the child process to share memory pages with its parent. This results in significant savings in terms of memory consumption and execution time.

Additionally, COW ensures that modifications made by one process do not affect the memory shared by other processes until necessary. When a write operation is attempted on a shared memory page, a copy of that page is created, allowing each process to have its own writable copy. This copy-on-write strategy minimizes unnecessary memory copying and enhances efficiency.

The implementation of COW in xv6 involved modifying the page table entries and introducing new flags, such as `PTE_COW`, to track the status of shared pages. By utilizing appropriate page fault handling mechanisms, the operating system can seamlessly manage shared memory and facilitate on-demand copying when modifications occur.

Throughout the development and testing process, the COW feature has demonstrated improved memory utilization and process management. It has proven effective in scenarios where multiple processes need to share memory but have limited write requirements. The integration of COW in xv6 enhances the overall stability, scalability, and resource utilization of the operating system.

Moving forward, further optimizations and enhancements can be explored to refine the COW implementation in xv6. This may include fine-tuning the page fault handling mechanisms, evaluating the performance impact under various workloads, and conducting thorough testing to ensure the robustness and reliability of the COW feature.

Overall, the addition of COW to xv6 marks a significant milestone in the evolution of the operating system, enabling efficient memory sharing and enhancing the overall system performance. The integration of COW opens up possibilities for more advanced memory management techniques and sets the foundation for future enhancements in xv6 and similar operating systems.