Sharbani Yagneshwar 8374360162

Os project – phase 1:

Creation of the first process in XV6:

In the main.c file, the userinit() function is called, which creates the first user process.

<u>userinit() function:</u>

Init process is the first process created by xv6 after boot up.

First, there's a pointer defined to proc struct, which is defined in proc.h, showing our process structure.

This proc structure involves several concepts like synchronization, memory management, trap/interrupt, file systems.

```
struct proc *p;


// Per-process state
struct proc {
 struct spinlock lock;

 // p->lock must be held when using these:
 enum procstate state;        // Process state
 void *chan;                  // If non-zero, sleeping on chan
 int killed;                  // If non-zero, have been killed
 int xstate;                  // Exit status to be returned to parent's
wait
 int pid;                     // Process ID

 // wait_lock must be held when using this:
 struct proc *parent;         // Parent process

 // these are private to the process, so p->lock need not be held.
 uint64 kstack;               // Virtual address of kernel stack
 uint64 sz;                   // Size of process memory (bytes)
 pagetable_t pagetable;       // User page table
 struct trapframe *trapframe; // data page for trampoline.S
 struct context context;      // swtch() here to run process
 struct file *ofile[NOFILE];  // Open files
 struct inode *cwd;           // Current directory
 char name[16];               // Process name (debugging)
};
```

Next, allocproc() is called.

```
p = allocproc();
initproc = p;
```

allocproc():

The function allocproc() is called during both init process creation and in fork system call.

The job of allocproc is to allocate a slot (a struct proc) in the process table and to initialize the parts of the process's state required for its kernel thread to execute.

Next, it tries to allocate a kernel stack of the process so that it's ready to be context switched by the scheduler. If the memory allocation fails, allocproc changes the state back to UNUSED and returns zero to signal failure.

allocproc sets up the new process with a specially prepared kernel stack and set of kernel registers that cause it to "return" to user space when it first runs.

Allocproc scans the proc table for a slot with state UNUSED.

```
 struct proc *p;

for(p = proc; p < &proc[NPROC]; p++) {
  acquire(&p->lock);
  if(p->state == UNUSED) {
    goto found;
  } else {
    release(&p->lock);
  }
}
return 0;
```

When it finds an unused slot, allocproc marks it as used and gives the process a unique PID.

```
found:
 p->pid = allocpid();
 p->state = USED;
```

Then we allocate the trap frame page, if there's any problem freeproc() would be called.

```
// Allocate a trapframe page.
if((p->trapframe = (struct trapframe *)kalloc()) == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
```

```
}
```

Next, we create the page table with proc_pagetable() function.

```
// An empty user page table.
p->pagetable = proc_pagetable(p);
if(p->pagetable == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
```

We need to have a stack pointer and return address for initial scheduling.
Memset writes zero to all the registers in the register save area. And then we initial the ra register to point to the forkret function. And we initial the sp register to point to the stack page.
When the process is scheduled, it will begin executing at the address of forkret with a stack.
And we finally release the lock

```
// Set up new context to start executing at forkret,
// which returns to user space.
memset(&p->context, 0, sizeof(p->context));
p->context.ra = (uint64)forkret;
p->context.sp = p->kstack + PGSIZE;
```

Next uvmfirst() will allocate a single page and copy the user init code.

Uvmfirst will load userinit code into the address 0 of the virtual address space that is being pointed to by this pagetable.

allocproc() sets up the pagetable, but doesn't fill it in with any code or data.

```
// allocate one user page and copy initcode's instructions
// and data into it.
uvmfirst(p->pagetable, initcode, sizeof(initcode));
```

We also set up the size field in proc struct, which tells us how big the virtual address space is.

```
p->sz = PGSIZE;
```

In trapframe, epc is where we saved the program counter. When a trap occurs in a running user process, we save the program counter in epc. We save all the general registers as well, including the stack pointer. And when we're ready to return to the user's process, we return to executing at the program counter stored in epc, after restoring all the registers.

We are setting program counter to zero in here. Only for first process, we'll start executing it at location zero.

```
p->trapframe->epc = 0;      // user program counter
```

And we'll set the stack pointer to page size.

```
p->trapframe->sp = PGSIZE;  // user stack pointer
```

We use safestrcpy function to set the name of the process.

```
safestrcpy(p->name, "initcode", sizeof(p->name));
```

We also set the current working directory and finally set the process state to RUNNABLE, for the scheduler to find the process when it's looking for something to run and schedule it.

```
p->cwd = namei("/");

p->state = RUNNABLE;
```

After all these changes and setting the values of proc struct object, we can release the lock for this process.

```
release(&p->lock);
```

Adding kfreemem() system call:

Our goal is to implement a system call for returning free memory space. (number of free pages available in the kernel).

The xv6 kernel keeps track of free pages in a linked list.

The kernel uses the struct run to track a free page. This structure stores a pointer to the next free page and is stored within the page itself.

```
struct run {
 struct run *next;
};
```

The kernel keeps a pointer to the first page of this free list in the structure struct kmem. Pages are added to this list upon initialization, or on freeing them up.

```
struct {
 struct spinlock lock;
 struct run *freelist;
} kmem;
```

Functions related to page allocation can be found in kalloc.c file.

First, we will write a function to determine the number of free pages.
freepages() iterates through freelist linked list in order to find the number of free pages. (implemented in kalloc.c)
We should acquire the kmem.lock while counting to avoid other processes from modifying the freelist.

```
int
freepages()
{
   int pages = 0;
   struct run *r;

   acquire(&kmem.lock);
   r = kmem.freelist;

   //iterate over linked list
   while (r != 0) {
       pages += 1;
       r = r->next;
   }

   release(&kmem.lock);

   return pages;
}
```

After adding this function to kalloc.c we also need to add the freepages() prototype to def.s.h:

```
// kalloc.c

void*           kalloc(void);

void            kfree(void *);

void            kinit(void);

int             freepages(void);
```

After adding this function to kalloc.c we should add the freepages() function to kernel/main.c to print the number of free pages on boot, so we can make sure our function working.

```
void
main()
{
 if(cpuid() == 0){
   consoleinit();
   printfinit();
   printf("\n");
   printf("xv6 kernel is booting\n");
   printf("\n");
   kinit();         // physical page allocator
   printf("free memory pages: %d\n", freepages());
```

"make qemu" command result after these steps:

xv6 kernel is booting

free pages: 32734

Adding the kfreemem() system call:

In kernel directory, we should add sys_kfreemem() to sysproc.c:

```
uint64

sys_kfreemem(void)

{

  return freepages();
```

```
}
```

Then we need to add the sys_kfreemem syscall to syscall.h where a number is assigned to every system call.

```
#define SYS_kfreemem  22
```

Then, modify syscall.c to add sys_pages to the syscall table:

The function prototype which needs to be added to syscall.c file is as follows

```
extern uint64 sys_kfreemem(void);
```

we need to add pointer to system call in syscall.c file

```
[SYS_pages]   sys_kfreemem,
```

Adding the kfreemem() system call to the user directory

We should add entry kfreemem() at the end of usys.pl for the assembly functions that make the system calls to generate.

```
entry("kfreemem");
```

Then we should add the prototype for kfreemem() to user.h

```
int kfreemem(void);
```

Adding a pages user-level program

We should add the user program to call kfreemem() and print the result:

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

#include "kernel/stat.h"

#include "user/user.h"



Int
```

```c
main(int argc, char *argv[])

{

   printf("'running user program' free pages: %d\n", kfreemem());

   exit(0);

}
```

For the final step, we should add our user program to Makefile:

```makefile
UPROGS=\

  $U/_cat\

  $U/_echo\

  $U/_forktest\

  $U/_grep\

  $U/_init\

  $U/_kill\

  $U/_ln\

  $U/_ls\

  $U/_mkdir\

  $U/_rm\

  $U/_sh\

  $U/_stressfs\

  $U/_usertests\

  $U/_grind\

  $U/_wc\

  $U/_zombie\

   $U/_kfreemem\
```

Final result after running “make qemu”:

```
xv6 kernel is booting

free pages: 32734
hart 1 starting
hart 2 starting
init: starting sh
$ kfreemem
'running user program' free pages: 32564
$
```

Output analysis:

Qemu allocates 128Mb of memory. And every memory page is 4096 bytes. So there’s

4096 * 32734 = 134078464 bytes of memory left.

It’s close to 128 Mb and close to our expectation because there are no processes running.

The user program output shows fewer memory pages because the init process has been created, and memory page has been allocated.

---

https://github.com/syagneshwar44/XV6-Operating-System#