Assignment 1

Group 11

Tanmay Mittal	220101099
Sushant Kumar	220101096
Tanay Goenka	220101098
Sarthak Kapoor	220101116

Our Changed Files Link:
https://github.com/Tanmay7404/CS344_OsLab_2024/tree/main/Assignment%201

Exercise 1.1

```
atoi(const char *s)
{
  int n;

  n = 0;
  while('0' <= *s && *s <= '9')
    n = n*10 + *s++ - '0';
  return n;
}
```

- **atoi** function takes input as a string and converts to an integer in C. This function returns 0 if the given input is <0 and returns the floor of the floating point numbers.

```c
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

- The `sleep` function in xv6 is a synchronization mechanism that puts the current process to sleep, making it inactive until a specific event occurs. The function takes two arguments: `chan`, a pointer that represents the reason or event the process is waiting for (such as a resource becoming available or an I/O operation completing), and `lk`, a spinlock that protects the data associated with that event. Upon invocation, `sleep` first checks that the current process (p) and the lock (`lk`) are valid; if not, it triggers a kernel panic. If the process is valid, `sleep` ensures mutual exclusion by acquiring the `ptable.lock` (if `lk` is not already `ptable.lock`), then releases the original lock (`lk`) to allow other processes to access the shared data while this process is sleeping. The function sets the process's `chan` field to the specified channel and changes its state to `SLEEPING`. Afterward, it calls the `sched()` function to yield the CPU, allowing the scheduler to switch to another process. Upon being woken up, the process clears its `chan` field, reacquires the original lock (`lk`), and resumes execution. This implementation is crucial for process synchronization, enabling xv6 to handle concurrent processes efficiently by putting processes to sleep and waking them up based on specific events or conditions.

sleep.c > main(int, char * [])

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[]){
    if(argc!=2){
        printf(2,"ERROR 404 : The program expects 2 arguments only. Received %d which is != 2 obviously\n",arg
        exit();
    }
    for(int i=0;i<strlen(argv[1]);i++){
        if((argv[1][i]<'0')||(argv[1][i]>'9')){
            printf(2,"ERROR 405 : The argument passed is not a positive int. Received %s which is obviously no
            exit();
        }
    }
    int ticks=atoi(argv[1]);
    sleep(ticks);
    exit();


}
```

Made a file sleep.c which takes input arguments of the number of ticks to sleep. If invalid number or arguments is passed or the argument is not an int, gives an error. Else sleep for the given time, and then exit

QEMU

Machine View

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ sleep 12 12
ERROR 404 : The program expects 2 arguments only. Received 3 which is != 2 obvio
usly
$ sleep -23
ERROR 405 : The argument passed is not a positive int. Received -23 which is obv
iously not an pos int u dumbo
$ sleep 1.53
ERROR 405 : The argument passed is not a positive int. Received 1.53 which is ob
viously not an pos int u dumbo
$ sleep 200
$
```

Exercise 1.2

```
void printFrame(int frameNumber) {
    clearScreen();


    print(frameNumber);
}

int main(int argc, char *argv[]) {
    int i;
    while(1){
        for (i = 0; i <=100; i++) {
            printFrame(i);
            sleep(6);
        }
        for(int i=100;i>=0;i--){
            printFrame(i);
            sleep(6);
        }
    }

    exit();
}
```

Created a file animation.c which loops forever and in each loop, clears the screen, print ascii image (shifted by some space depending on the loop number) and then sleep for a small amount

```
void clearScreen() {
    printf(1, "\033[2J\033[H");
}
```

clearScreen function just prints empty spaces on screen until earlier values gets removed from screen

print(i) functions print a man with i spaces from the beginning

https://github.com/Tanmay7404/CS344_OsLab_2024/blob/main/Assignment%201/Ex%202/Screencast%20from%2027-08-24%2007%3A37%3A49%20PM%20IST.webm

Here is our animation. Hope u enjoy :)

Exercise 1.3 (Statistics of a process)

We added 4 variables to the proc structure (in proc.h) , which track the creation time, run time, ready time and the sleep time.

proc.h

```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  uint ctime,stime,retime,rutime; // vars added for finding out the stats
};
```

Ctime → creation time,

Stime → sleeping time (while the process is waiting for I/0)

Retime → while the process is in the ready queue.

Runtime → while the process is in the running state

We then added a function named : **update_process_times** which is responsible for updating the values of the above variables. This function is defined in proc.c and called from trap.c. The trap scheduler calls this function upon a timer interrupt. When there is a timer interrupt , the **ticks** variable is incremented and we leveraged this fact to increment our defined variables by using a switch-case statement that checks the state of the program and increments the corresponding variables.

proc.c

```c
void update_process_times() {
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        // printf(0,"process update %d\n",p->state);
        switch(p->state) {
            case SLEEPING:
                p->stime++;
                break;
            case RUNNABLE:
                p->retime++;
                break;
            case RUNNING:
                p->rutime++;
                break;
            default:
                break;
        }
    }
    release(&ptable.lock);
}
```

trap.c

```c
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
      update_process_times();
    }
```

When a process is created the function **allocproc()** is called that initializes the process, here we have initialized our defined variables.

proc.c

```
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  p->ctime = ticks;
  p->retime = 0;
  p->rutime = 0;
  p->stime = 0;
  return p;
}
```

We then implemented the system call **wait2(**int *,int*,int***)** this takes in 3 int pointers (retime ,rutime ,stime). The wait2 system call clears the values of these variables in addition to the functionality provided with the normal **wait()** system call.

proc.c

```c
int
wait2(int *retime, int *rutime, int *stime)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;


      if(p->state == ZOMBIE){

        *retime = p->retime;
        *rutime = p->rutime;
        *stime = p->stime;
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        p->ctime = 0;
        p->retime = 0;
        p->rutime = 0;
        p->stime = 0;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

sysproc.c

```c
int sys_wait2(void)
{
  int *retime, *rutime, *stime;

  // Correctly retrieve the pointers from user space arguments
  if (argptr(0, (void*)&retime, sizeof(int)) < 0) {
    return -1;
  }
  if (argptr(1, (void*)&rutime, sizeof(int)) < 0) {
    return -1;
  }
  if (argptr(2, (void*)&stime, sizeof(int)) < 0) {
    return -1;
  }

  // Call wait2 with the retrieved pointers
  return wait2(retime, rutime, stime);
}
```

After creating the wait2 system call we created a test file for implementing this system call.

test_ass1.c

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void)
{
    int retime, rutime, stime;
    int pid = fork();

    if(pid == 0) {
        for (volatile int i = 0; i < 100000000; i++) {}
        char buf[100];
        read(0, buf, 100);
        sleep(69);
        for (volatile int j = 0; j < 10000; j++) {}
        exit();
    } else {
        // Parent process
        if (wait2((void*)&retime, (void*)&rutime, (void*)&stime) >= 0) {
            printf(1,"retime: %d, rutime: %d, stime: %d\n", retime, rutime, stime);
        }
        else {
            printf(1,"wait2 failed \n ");
        }
    }

    exit();
}
```

Output

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test_ass1
asb
retime: 69, rutime: 36, stime: 301
$ sarthak@F5-RAZOR:~/Desktop/SEM 5/OS/xv6-public-master$
```

Output received on executing the file **test_ass1.c**

Test Cases:

1. We changed the loop limits in the child process that directly affects the runtime of the process
   1.1 i < 100000
   1.2 i < 1000000
   1.3 i < 10000000
2. We changed the sleep time in the child that directly affects the ready time of the child process
3. We added the read() call that simulates the I/O wait.