CS 344 OS Lab

Assignment 1

Mathematics and Computing

Group 2

by
Aayush Suthar, 220123004
Aaditya Jalan, 220123001
Avanish Gadhikar, 220123075
Chaitanya Chhabra, 220123012

Link to patchfile.

Task 1.1: Sleep

Implement a user-level sleep program for xv6, along the lines of the UNIX sleep command. Your sleep should pause for a user-specified number of ticks. A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip. Your solution should be in the file user/sleep.c.

Created a user program ‘sleep.c’ to make use of the in-built sleep system call in xv6. The user program takes as input the number of ticks to delay the next operation.

```
aadityos@DESKTOP-TOLJKGE:~/xv6-public$ ls
BUGS         cat.c         file.c      kbd.c        mkfs.c     rm.c         spinlock.c  toc.hdr     vm.c
LICENSE      console.c     file.h      kbd.h        mmu.h      runoff       spinlock.h  trap.c      waittest.c
Makefile     cuth          forktest.c  kernel.ld    mp.c       runoff.list  spinp       trapasm.S   wc.c
Notes        date.h        fs.c        kill.c       mp.h       runoff.spec  stat.h      traps.h     x86.h
README       defs.h        fs.h        lapic.c      param.h    runoff1      stressfs.c  types.h     zombie.c
TRICKS       dot-bochsrc   gdbutil     ln.c         picirq.c   sh.c         string.c    uart.c
animate.c    echo.c        grep.c      log.c        pipe.c     show1        swtch.S     ulib.c
asm.h        elf.h         ide.c       ls.c         pr.pl      sign.pl      syscall.c   umalloc.c
bio.c        entry.S       init.c      main.c       printf.c   sleep.c      syscall.h   user.h
bootasm.S    entryother.S  initcode.S  memide.c     printpcs   sleep1.p     sysfile.c   usertests.c
bootmain.c   exec.c        ioapic.c    memlayout.h  proc.c     sleeplock.c  sysproc.c   usys.S
buf.h        fcntl.h       kalloc.c    mkdir.c      proc.h     sleeplock.h  toc.ftr     vectors.pl
```

Made changes to the Makefile and added sleep to UPROGS as well as extra

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _sleep\
```

```
EXTRA=\
    animate.c mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    sleep.c printf.c umalloc.c waittest.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

'Sleep.c' was implemented as shown below

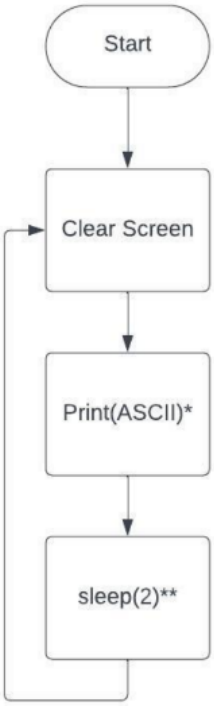
```c
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[]){
    if(argc != 2){
        printf(1, "Correct syntax: sleep <argument> \n");
            exit();
    }
    int cnt;
    cnt = atoi(argv[1]);
    sleep(cnt);
    exit();
}
```

sleep.c

Task 1.2: User Program to display animation

Created a C script “animate.c” which works as per this flowchart.
Added animate to UPROGS and EXTRAS.



Clear Screen

This was achieved by printing the escape sequence “\x1b]2J\x1b]H”

```
printf(1, "\x1b[2J\x1b[H");
```

Print ASCII

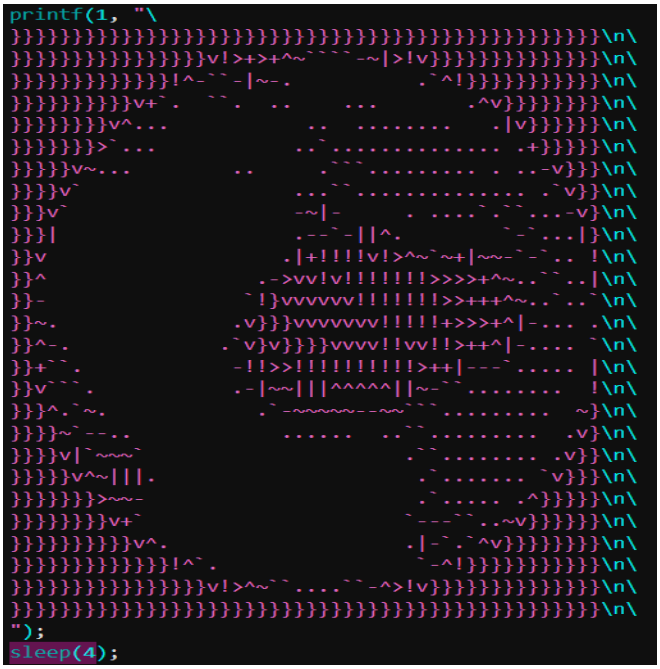
Our initial attempt to print full-screen Batman ASCII art was thwarted by the block size restriction on xv6. An error ‘fnb < MAXFILE’ was thrown!’

Our final animate.c script uses print statements to print a rotating globe.

Sleep(n)

Sleep(n) is a command to sleep for n ticks, which turns out to be about 10n milliseconds. By using “sleep(4)” we made the OS sleep for 40ms between each printf, which means the FPS of the animation is 25 Hz.

The final result was a rotating earth animation that was somewhat fluid and successful.



Task 1.3: Statistics

An exhaustive list of all the changes we made:

1. **proc.h**

Added **ctime** (current time), **stime** (sleep time), **retime** (ready time), and **rutime** (running time) to proc.h

```
// 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)
  int ctime;                   // Variable added to track creation time of process.
  int stime;                   // Variable added to track time spent sleeping.
  int retime;                  // Variable added to track time spent ready(runnable).
  int rutime;                  // Variable added to track time spent running.
};
```

2. **proc.c**

Modified function *allocproc* → Added the following initializations:

```
  p->ctime = ticks;               // Make creation time of process = current ticks
  p->stime = 0;                   // Initialised value of sleep time = 0
  p->retime = 0;                  // Initialised value of ready time = 0
  p->rutime = 0;                  // Initialised value of running time = 0
```

Modified function *scheduler* → Added increment instructions for all time variables

```
if(p->state == RUNNABLE){
  p->retime++;
}else if(p->state == RUNNING){
  p->rutime++;
}else if(p->state == SLEEPING){
  p->stime++;
}
if(p->state != RUNNABLE)
  continue;
```

Created function *int wait2*

```
// Add a function to implement wait2 functionality
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){
        // 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;

        // Assigning the values of retime, rutime, stime from those present in the kernel
        *retime = p->retime;
        *rutime = p->rutime;
        *stime = p->stime;

        release(&ptable.lock);
        return pid;
      }
    }
```

3. **usys.S**

Added SYCALL(wait2) at the end of declarations

```
SYSCALL(wait2)
```

4. **syscall.h**

Added #define SYS_wait2 at the end of definitions

```
#define SYS_wait2  22 // Added wait2 to the list of system calls.
```

5. **user.h**

Added forward declaration of wait2 to the list in this file

```
int wait2(int*, int*, int*); // Added wait2 to the list of system calls.
```

6. **sysproc.c**

Added sys_wait2 function with argument error-catching.

```
int
sys_wait2(void){
  int *retime, *rutime, *stime;
  if(argptr(0, (void*)&retime, sizeof(retime)<0))
    return -1;
  if(argptr(1, (void*)&rutime, sizeof(rutime)<0))
    return -1;
  if(argptr(2, (void*)&stime, sizeof(stime)<0))
    return -1;

  // OK
  return wait2(retime, rutime, stime);
}
```

7. **syscall.c**

Added extern forward declaration for sys_wait2

```
extern int sys_wait2(void);        // Added wait2 to the list of system calls.
```

8. **defs.h**

Added forward declaration for wait2 so that sysproc.c can recognize it

```
int             wait2(int*, int*, int*);            // Added sys_wait2
```

9. **[NEW] waittest.c**

Made this file to give different tasks to parent and child process and test the wait2 function

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

int
main(int argc, char *argv[]){
    int retime = 0, rutime = 0, stime = 0;
    int x = fork();
    if(x < 0){
        printf(1, "Error in creating child process\n");
    }else if (x == 0){
        // Inside child process.

        int g_child = fork();
        if(g_child < 0){
            printf(1, "Error in creating grandchild!\n");
        }else if(g_child == 0){
            // Inside grand child
            printf(1, "Inside grandchild\n");
            for(volatile int i = 0; i < 2000000; i++){
                int a = 2;
                a = a*4;
            }
            // sleep(4);
        }else{
            int a = 0, b = 0, c = 0;
            int w_g_child = wait2(&a, &b, &c);
            printf(1, "\x1b[2J\x1b[H");
            for(int i = 0; i < 250; i++){
                printf(1, "*");
            }
            printf(1, "\n\n");
            printf(1, "Stats for grand child\n");
            printf(1, "pid (Process ID) = %d\n", w_g_child);
            printf(1, "Ready/Runnable Time (retime) = %d\n", a);
            printf(1, "Running time (rutime) = %d\n", b);
            printf(1, "Sleeping Time (stime) = %d\n", c);
        }
    }else{
        // Inside Parent process.
        int y = wait2(&retime, &rutime, &stime);
        if(y < 0){
            printf(1, "Error generated in wait!\n");
        }
        printf(1, "Stats for child\n");
        printf(1, "pid (Process ID) = %d\n", y);
        printf(1, "Ready/Runnable Time (retime) = %d\n", retime);
        printf(1, "Running time (rutime) = %d\n", rutime);
        printf(1, "Sleeping Time (stime) = %d\n", stime);
    }
    exit();
}
```

10. Makefile

Added waittest.c to two places in this file: UPROGS and EXTRA

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _sleep\
    _animate\
    _waittest\
```

```
EXTRA=\
    animate.c mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    sleep.c printf.c umalloc.c waittest.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

Observation: After running waittest in QEMU, we saw the following screen:

```
**********
**********
**********

Stats for grand child
pid (Process ID) = 5
Ready/Runnable Time (retime) = 2
Running time (rutime) = 1
Sleeping Time (stime) = 0
Stats for child
pid (Process ID) = 4
Ready/Runnable Time (retime) = 11
Running time (rutime) = 9
Sleeping Time (stime) = 1
```

This picture demonstrates that:

Here the child process is forked to get a grandchild process in order to get a better understanding and analysis of the times that we are getting here.

The grandchild process which runs before the child process, runs for 1 unit of time, whereas the child process sleeps while the grandchild runs.

We have also inferred the ready time, running time, and sleep time from the PCB of the process using a system call.