CS4375-13948 Fall 2023
Denisse Loya Villalobos
dloyavilla@miners.utep.edu

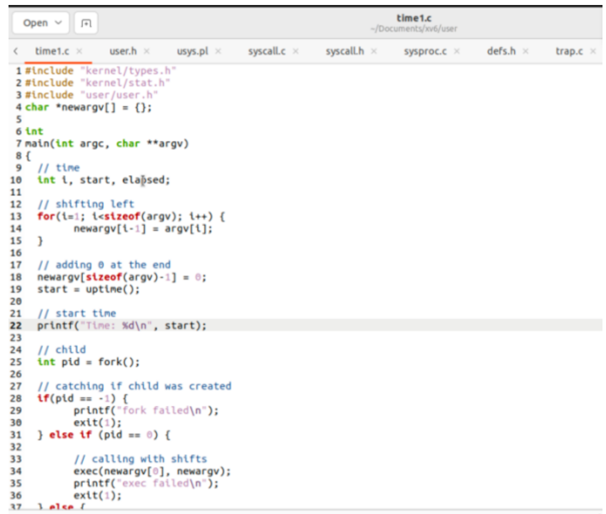
Homework Report
Submitted on 3-Oct-2023

HW 2: Time Command

Task 1. Implement a time1 command that reports elapsed time.

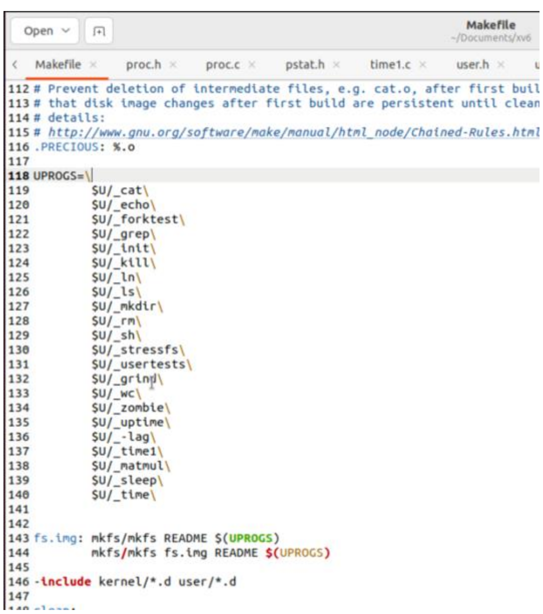
Figure 1: the code implemented that used the uptime to record the elapsed time of matmul.
Figure 2: Results from running my time1 command.
Figure 3: The time1, matmul, and sleep in UPROGS in Makefile.
I learned how to create a xv6 command, that can calculate the execution time elapse. I learned the correct integration into the build process, and now I comprehend the difference between working in the kernel or user.

The difficulties I had were debugging problems in my custom time1 command.



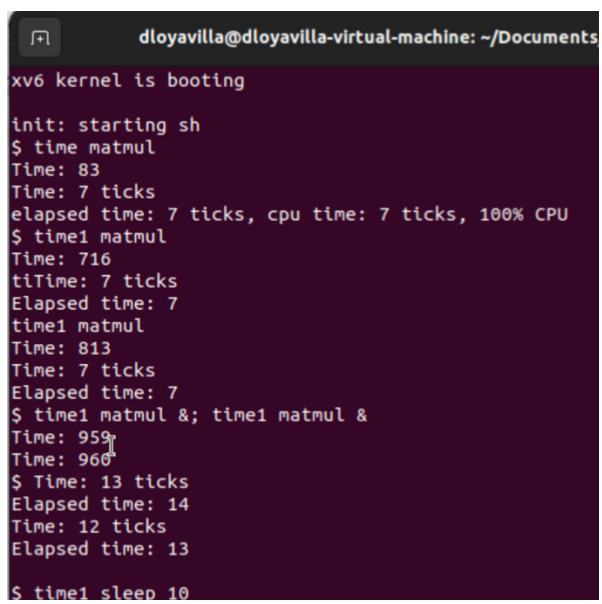
```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
char *newargv[] = {};

int
main(int argc, char **argv)
{
  // time
  int i, start, elapsed;

  // shifting left
  for(i=1; i<sizeof(argv); i++) {
        newargv[i-1] = argv[i];
  }

  // adding 0 at the end
  newargv[sizeof(argv)-1] = 0;
  start = uptime();

  // start time
  printf("Time: %d\n", start);

  // child
  int pid = fork();

  // catching if child was created
  if(pid == -1) {
        printf("fork failed\n");
        exit(1);
  } else if (pid == 0) {

        // calling with shifts
        exec(newargv[0], newargv);
        printf("exec failed\n");
        exit(1);
  } else {
```

Figure 1. Time 1 Code



```makefile
# Prevent deletion of intermediate files, e.g. cat.o, after first buil
# that disk image changes after first build are persistent until clean
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

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/_uptime\
	$U/_-lag\
	$U/_time1\
	$U/_matmul\
	$U/_sleep\
	$U/_time\


fs.img: mkfs/mkfs README $(UPROGS)
	mkfs/mkfs fs.img README $(UPROGS)

-include kernel/*.d user/*.d
```

Figure 4. UPROGRS changes



```
xv6 kernel is booting

init: starting sh
$ time matmul
Time: 83
Time: 7 ticks
elapsed time: 7 ticks, cpu time: 7 ticks, 100% CPU
$ time1 matmul
Time: 716
tiTime: 7 ticks
Elapsed time: 7
time1 matmul
Time: 813
Time: 7 ticks
Elapsed time: 7
$ time1 matmul &; time1 matmul &
Time: 959
Time: 960
$ Time: 13 ticks
Elapsed time: 14
Time: 12 ticks
Elapsed time: 13

$ time1 sleep 10
```

Figure 2. Test Cases

Task 2. Keep track of how much cputime a process has used.

Figure 5. Added int for cputime so we can store the cputime in proc.h
Figure 7. Initialized to 0 cpu time, in allocproc, in proc.c so each time is called we start the counter to 0
Figure 6 & 8. Incremented cputime in file trap.c, for user and kernel trap, so it increments everytime it goes into the user or kernel traps
By completing this job, I have gained knowledge on how to alter the xv6 operating system to monitor CPU usage by particular programs. The resource utilization of processes can be tracked and profiled using this.
My difficultly was how to increment the cputime was confusing but after changing the code a little I figured it out.

```c
  /* 280 */ uint64 t6;
};

enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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
  int cputime;                 // cputime

  // 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
```

Figure 7. proc.h

```c
static struct proc*
allocproc(void)
{
  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;

found:
  p->pid = allocpid();
  p->state = USED;
  p->cputime = 0;
```

Figure 6. proc.c

```c
pstat.h    time1.c    user.h    usys.pl    syscall.c    syscall.h    sysproc.c
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2){
    yield();
    p->cputime++;

  }
  usertrapret();
}
```

Figure 5. User trap

```c
void
kerneltrap()
{
  int which_dev = 0;
  uint64 sepc = r_sepc();
  uint64 sstatus = r_sstatus();
  uint64 scause = r_scause();

  if((sstatus & SSTATUS_SPP) == 0)
    panic("kerneltrap: not from supervisor mode");
  if(intr_get() != 0)
    panic("kerneltrap: interrupts enabled");

  if((which_dev = devintr()) == 0){
    printf("scause %p\n", scause);
    printf("sepc=%p stval=%p\n", r_sepc(), r_stval());
    panic("kerneltrap");
  }

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING){
    myproc()->cputime++;
    yield();
  }

  // the yield() may have caused some traps to occur,
  // so restore trap registers for use by kernelvec.S's sepc instruction.
  w_sepc(sepc);
  w_sstatus(sstatus);
}
```

Figure 8. Kernel trap

Task 3. Implement a wait2() system call that waits for a child to exit and returns the child's status and rusage.

Figure 10. added rusage structure in a new file called pstat.h
Figure 11. in user.h added structure so the user can use it.
Figure 12. usys.pl adding the identifier for the system call.
Figure 9. syscall.h, adding number corresponding to systemcall
Figure 13. adding signature to make calls in user level.
Figure 14. syscall.c this file is to request resources from kernel to user level, so we add wait2.
Figure 16. sysproc.c we added wait 2, and validated the arguments passed.
Figure 15. code to implement the new process, it is based on the existing wait, just adding the use of the structure rusage, and copying the status.
I learned about the functionality of each file, and how to code an alternative version of the wait system call, but this time also being able to calculate the cputime.
This was the most difficult task for this lab, because we had to understand the general system call infrastructure, mapping system call numbers, validating arguments, and ensuring that user-level and kernel-level code operate together. These difficulties can be overcome by carefully following the offered instructions, consulting documentation, and undertaking comprehensive testing to confirm that the implementation is right.

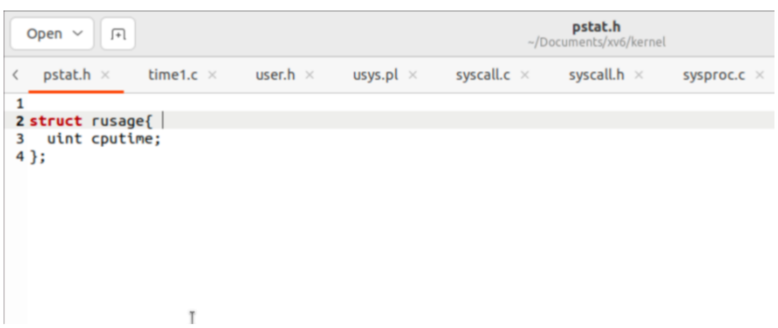


Figure 10. pstat.h

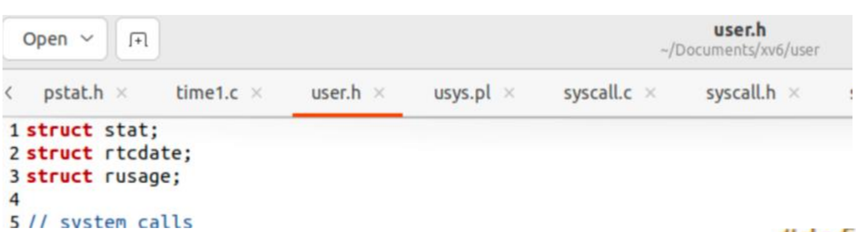


Figure 11. user.h

```
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_wait2  22
```

Figure 9. syscall.h

```
entry("sbrk");
entry("sleep");
entry("uptime");
entry("wait2");
```

Figure 12. usys.pl

```
char* sbrk(int);
int sleep(int);
int uptime(void);
int wait2(int*,struct rusage*);

// ulib.c
int stat(const char*, struct stat*);
```

Figure 13. user.h

```
extern uint64 sys_uptime(void);
extern uint64 sys_wait2(void);

static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_wait2]   sys_wait2,
```

Figure 14. syscall.c

```
uint64
sys_wait2(void)
{
  uint64 p1, p2;
  if(argaddr(0, &p1) < 0 || argaddr(1, &p2)<0){return -1;}
  return wait2(p1,p2);
}
```

Figure 15. sysproc.c

Open proc.c ~/Documents/xv6/kernel

proc.h × proc.c × pstat.h × time1.c × user.h × usys.pl × syscall.c ×

```c
//
int
wait2(uint64 addr, uint64 rusage)
{
  struct proc *np;
  int havekids, pid;
  struct proc *p = myproc();
  struct rusage ru;

  acquire(&wait_lock);

  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(np = proc; np < &proc[NPROC]; np++){

      if(np->parent == p){
        // make sure the child isn't still in exit() or swtch().
        acquire(&np->lock);

        havekids = 1;
        if(np->state == ZOMBIE){

          ru.cputime = np->cputime;
          copyout(p->pagetable, rusage, (char *)&ru, sizeof(ru));
          //copying data from kernel mode to user mode

          // Found one.
          pid = np->pid;
          if(addr != 0 && copyout(p->pagetable, addr, (char *)&np->xstate,
                                  sizeof(np->xstate)) < 0) {
            release(&np->lock);
            release(&wait_lock);
            return -1;
```

Figure 16. proc.c

Task 4. Implement a time command that runs the command given to it as an argument and outputs elapsed time, CPU time, and %CPU used.

Figure 17. Code to implement the time, similar to time 1, we check if we have enough arguments, then shift the input, fork, check if the child was created successfully and if the exec was also successful.

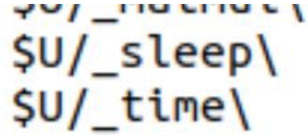
Figure 18. added time to the makefile

Figure 19. testing the code with the given input

I learned how to create a program that waits for child processes to exit while managing processes, and measure the elapsed time, CPU time, and %CPU utilization is calculated and displayed. It was difficult to understand how processes are created, run, and waited for in the xv6 operating system.

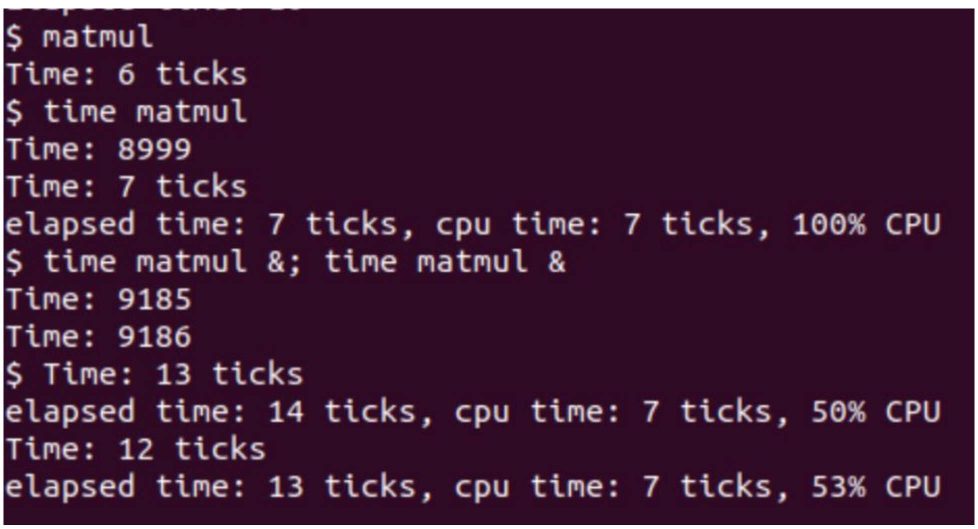
```c
#include "user/user.h"
#include "kernel/pstat.h"
char *newargv[] = {};

int
main(int argc, char **argv)
{
  int i, start, elapsed;
  struct rusage ru;

  if (argc < 2){
        printf("Usage: time <comm> [args...]");
        exit(-1);
   }

  for(i=1; i<sizeof(argv); i++) {
        newargv[i-1] = argv[i];
  }
  newargv[sizeof(argv)-1] = 0;
  start = uptime();
  printf("Time: %d\n", start);
  int pid = fork();
  if(pid == -1) {
        printf("fork failed\n");
        exit(1);
  } else if (pid == 0) {
        exec(newargv[0], newargv);
        printf("exec failed\n");
        exit(1);
  } else {
        int wpid = wait2(0, &ru);
        elapsed = uptime() - start;
        printf("elapsed time: %d ticks, cpu time: %d ticks, %d% CPU\n", elapsed, ru.cputime, ru.cputime*100/elapsed);
        if(wpid == pid){
        // the shell exited; restart it.
        exit(1);
```

Figure 17. time.c



```
$U/_sleep\
$U/_time\
```

Figure 18. Makefile



```
$ matmul
Time: 6 ticks
$ time matmul
Time: 8999
Time: 7 ticks
elapsed time: 7 ticks, cpu time: 7 ticks, 100% CPU
$ time matmul &; time matmul &
Time: 9185
Time: 9186
$ Time: 13 ticks
elapsed time: 14 ticks, cpu time: 7 ticks, 50% CPU
Time: 12 ticks
elapsed time: 13 ticks, cpu time: 7 ticks, 53% CPU
```

Figure 19. time test cases

Extra Credit (5 points). Discuss limitations of our time command. (Hint: For one limitation, consider what would happen if the command that is being timed forks child processes).

- Errors or inaccuracy with the cpu time because if we fork a process, the calculation will be only based on the parent, leaving all the child processes. Same with the rusage, it won't track each child separately.
- Also the code cannot be maintain if you want to measure complex commands, for example one that executes other commands.
- I'm not sure on how to solve this limitations but for sure we need to modify and make the code more complex, and I'm guessing with recursion we may be able to track each child process.