Trace-Backtrace

CS3500 Operating Systems Jul-Nov 2025

Sanjeev Subrahmaniyan S B

EE23B102

1 Objective

The objective of this lab is to:

1. A trace utility to display the instances and return values of when system calls are made by the traced program. The output is formatted showing the pid of the calling process, the system call name and its return value. The system calls to be traced are specified using a mask argument to trace function.
2. A kernel backtrace utility to display how a certain point in execution was reached. Its use is demonstrated through a test code, backtracing the sleep system call.

2 Trace

2.1 Code

A number of changes are needed to be made to implement the trace system call. I followed the order prescribed in the handout for the assignment and present them in the same order, explaining their utility as well.

2.1.1 xv6-riscv/Makefile

The trace program must be exposed as a userprogram to be compiled. This is done by adding an entry ”$U/_trace” in the ”$UPROGS” section of the makefile.

2.1.2 xv6-riscv/user/user.h

Under the //systemcalls section of this file, I added the function prototype for the trace function.

```c
    int trace(void);
```

This is used to expose the function to the trace.c file, where the program is implemented and calls the trace systemcall.

2.1.3 xv6-riscv/user/trace.c

This is the implementation of the program to trace the system calls in the executed program, which is passed as arguments. Its functionality is explained briefly in the comments.

```c
    #include "kernel/types.h"
    #include "user/user.h"
    #include "kernel/fcntl.h"

    void main(int argc, char *argv[]){
      if(argc <= 2){
```

```
    fprintf(2, "Usage: trace <mask for calls to trace> <command to run>\n");
    exit(1);
  }

  int n;
  if((n = trace(atoi(argv[1]))) == 0){// trace mask set correctly
    // close(1); // close stdout to only display trace outputs
    exec(argv[2], argv+2); // run process to trace
    exit(0);
  }
}
```

This program simply sets the trace_mask of the proc and executes the program to be traced.

2.1.4 xv6-riscv/user/usys.pl

An entry is made in the user system calls perl script corresponding to the trace system call. This ensures that during compilation usys.S is generated, which generates the stub of assembly code. This stub puts the index for the SYS_trace into the a7 register and makes an ecall.

2.1.5 xv6-riscv/kernel/syscall.h

Here an index is assigned to the trace system call. This is for the trap handler to identify which system call to perform when an ecall is made.

```
#define SYS_trace 22
```

Note that I have chosen the number 22 as it is the next free number. Any other number can be chosen, but the function pointer array must be appropriately setup to call the correct system call.

2.1.6 xv6-riscv/kernel/proc.h

The proc struct is modified to have an extra entry corresponding to the trace mask. This is used to identify which system calls to trace and also in propagating the traces to children processes.

```
// 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 trace_mask;               // mask for which system calls must be traced

  // 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)
};
```

Note that the trace_mask entry is made as a part of those entries for which the locks must be held during modification. This is because the trace mask for one process may be set by another and there must be no races.

2.1.7 xv6-riscv/kernel/sysproc.c

A new function is made to take the argument of the trace call(the mask detailed which systemcalls to be traced) and write it into the trace_mask field of the struct proc for the process. This is done by acquiring the lock(whose importance is specified above) and storing the mask in the struct.

```
uint64
sys_trace(void)
{
  int mask;
  struct proc *p;

  argint(0, &mask);
  p = myproc();
  acquire(&p->lock);
  p->trace_mask = mask;
  release(&p->lock);
  return 0;
}
```

2.1.8 xv6-riscv/kernel/proc.c

Here, the trace mask is copied onto any children process spawning as a result of the fork() call, if the mask is set in the forking process. The line is added inside the fork() function definition.

```
  // copy tracing mask values
  np->trace_mask = p->trace_mask;
```

2.1.9 xv6-riscv/kernel/syscall.c

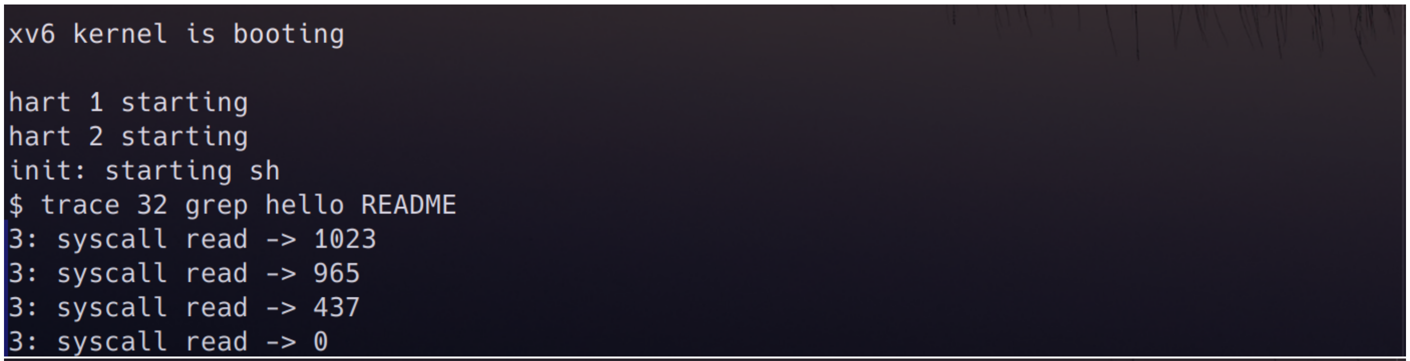
This is the main part of the code which checks if the trace mask corresponding to a system call is set, and prints the data if so. I have also exposed the sys_trace function for the compiler and added its entry to the fuction pointer array as well. Additionally, I have also added an array of strings to be used as the names when printing. The modified part alone is shown below:

```
extern uint64 sys_trace(void);
// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
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_trace]   sys_trace,
};

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_trace]   sys_trace,
};

char *syscallnames[] = {
  "fork",
  "exit",
  "wait",
  "pipe",
  "read",
  "kill",
  "exec",
  "fstat",
  "chdir",
  "dup",
  "getpid",
  "sbrk",
  "sleep",
  "uptime",
  "open",
```

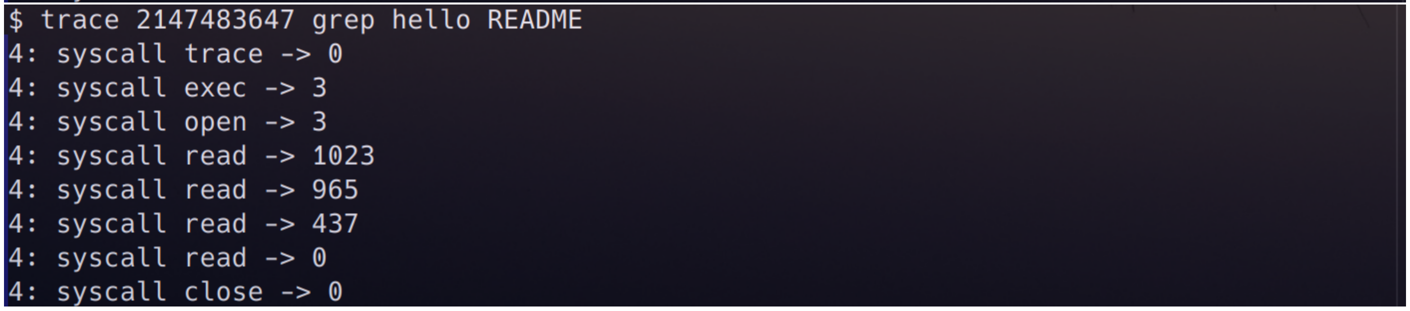
```c
  "write",
  "mknod",
  "unlink",
  "link",
  "mkdir",
  "close",
  "trace",
};

void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
    if((p->trace_mask & (1<<num)) == (1<<num)){
      printf("%d: syscall %s -> %ld\n", p->pid, syscallnames[num-1],
      (long int)p->trapframe->a0); // num-1 is used as syscall.h starts from 1
    }
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

2.2 Execution and Output

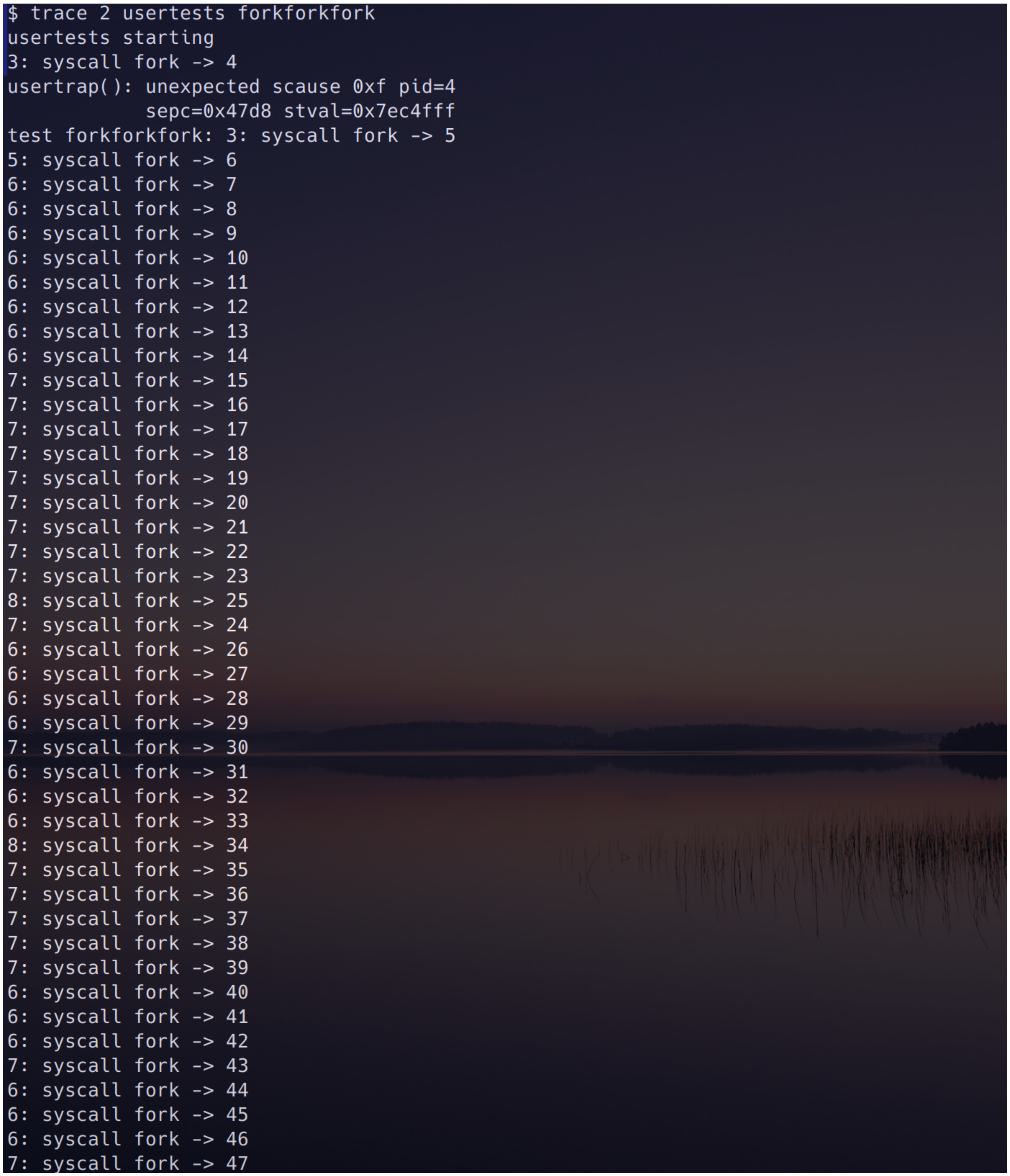
After running the make clean and make qemu commands, the commands exactly in the order in which is provided in the examples are run. The output is captured in these images:



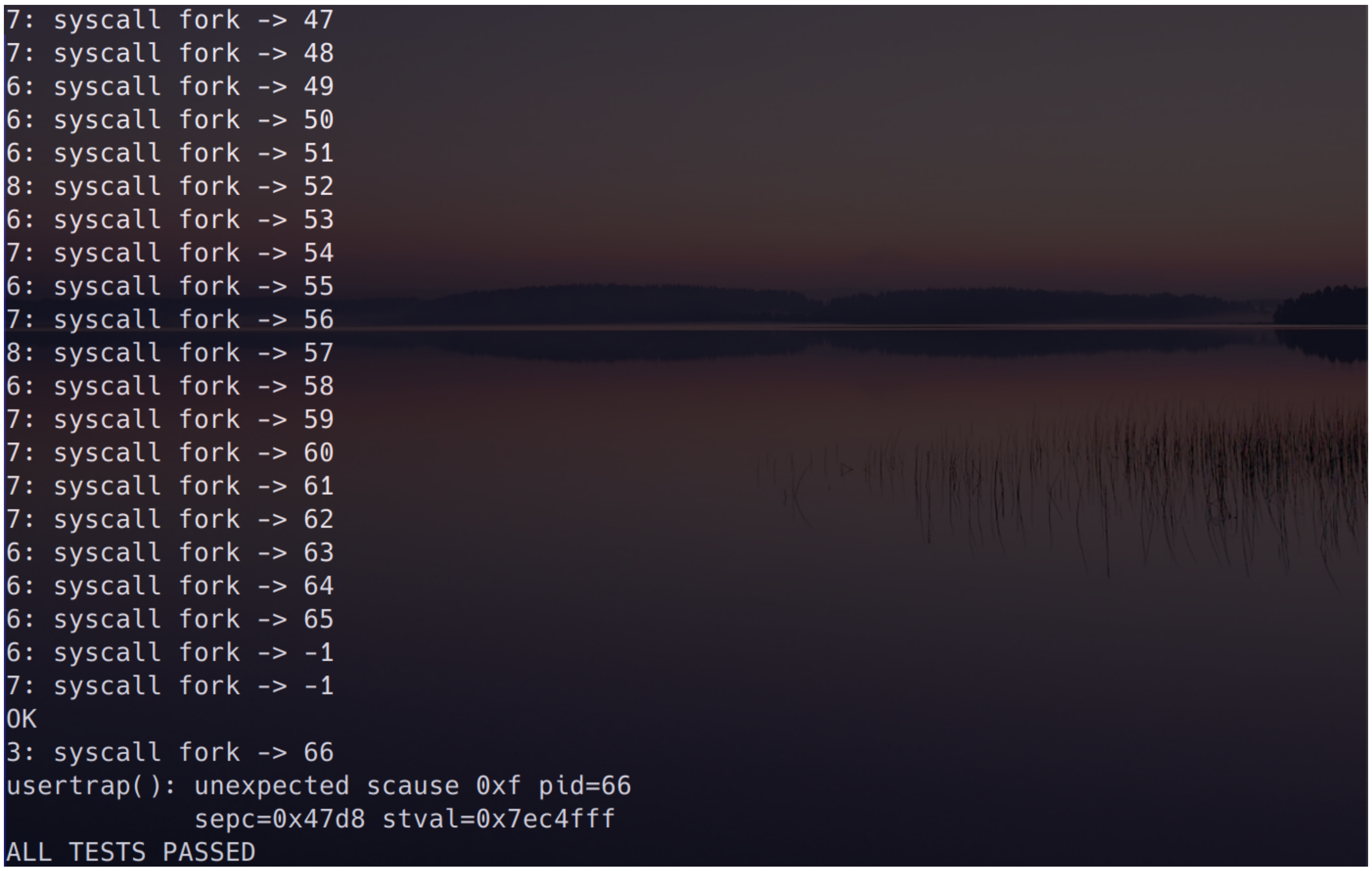
```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 965
3: syscall read -> 437
3: syscall read -> 0
```



```
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 965
4: syscall read -> 437
4: syscall read -> 0
4: syscall close -> 0
```



```
$ trace 2 usertests forkforkfork
usertests starting
3: syscall fork -> 4
usertrap(): unexpected scause 0xf pid=4
            sepc=0x47d8 stval=0x7ec4fff
test forkforkfork: 3: syscall fork -> 5
5: syscall fork -> 6
6: syscall fork -> 7
6: syscall fork -> 8
6: syscall fork -> 9
6: syscall fork -> 10
6: syscall fork -> 11
6: syscall fork -> 12
6: syscall fork -> 13
6: syscall fork -> 14
7: syscall fork -> 15
7: syscall fork -> 16
7: syscall fork -> 17
7: syscall fork -> 18
7: syscall fork -> 19
7: syscall fork -> 20
7: syscall fork -> 21
7: syscall fork -> 22
7: syscall fork -> 23
8: syscall fork -> 25
7: syscall fork -> 24
6: syscall fork -> 26
6: syscall fork -> 27
6: syscall fork -> 28
6: syscall fork -> 29
7: syscall fork -> 30
6: syscall fork -> 31
6: syscall fork -> 32
6: syscall fork -> 33
8: syscall fork -> 34
7: syscall fork -> 35
7: syscall fork -> 36
7: syscall fork -> 37
7: syscall fork -> 38
7: syscall fork -> 39
6: syscall fork -> 40
6: syscall fork -> 41
6: syscall fork -> 42
7: syscall fork -> 43
6: syscall fork -> 44
6: syscall fork -> 45
6: syscall fork -> 46
7: syscall fork -> 47
```



```
7: syscall fork -> 47
7: syscall fork -> 48
6: syscall fork -> 49
6: syscall fork -> 50
6: syscall fork -> 51
8: syscall fork -> 52
6: syscall fork -> 53
7: syscall fork -> 54
6: syscall fork -> 55
7: syscall fork -> 56
8: syscall fork -> 57
6: syscall fork -> 58
7: syscall fork -> 59
7: syscall fork -> 60
7: syscall fork -> 61
7: syscall fork -> 62
6: syscall fork -> 63
6: syscall fork -> 64
6: syscall fork -> 65
6: syscall fork -> -1
7: syscall fork -> -1
OK
3: syscall fork -> 66
usertrap(): unexpected scause 0xf pid=66
            sepc=0x47d8 stval=0x7ec4fff
ALL TESTS PASSED
```

3 Backtrace

3.1 Code

3.1.1 xv6-riscv/kernel/riscv.h

Inside the assembler directives, an inline assembly program is added to copy the value of the s0(frame pointer) register into a C language variable which can be used. This function is provided in the handout for the experiment.

```
static inline uint64
r_fp()
{
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x));
  return x;
}
```

3.1.2 xv6-riscv/kernel/defs.h

Here, the function prototype for backtrace() is added under the //printf.c section, which is where the code resides. This is required to use the functions in other programs by simply including the defs.h file.

```
void            backtrace(void);
```

3.1.3 xv6-riscv/kernel/printf.c

As mentioned in the handout, the implmentation of the backtrace function is contained in this file. Additionally, the backtrace function is also added inside the panic() function definition to call it when panicking.

```c
void
backtrace(void)
{
  uint64 fp = r_fp(); // read the current value of the frame pointer
  uint64 stack_low = PGROUNDDOWN(fp);
  uint64 stack_high = stack_low + PGSIZE;

  uint64 ret_addr, newfp;

  printf("backtrace:\n");

  while(fp >= stack_low +16 && fp < stack_high){
    ret_addr = *(uint64 *)(fp - 8);
    printf("0x%lx\n", ret_addr);
    newfp = *(uint64 *)(fp - 16);

    if(newfp <= fp || newfp < stack_low || newfp >= stack_high)break;

    fp = newfp;
  }
}
```

The logic behind this snippet of code is very simple - we know that for every frame pointer, (fp - 8) contains the return address and (fp - 16) contains the previous frame's pointer. By continously traversing the line of calls(by moving to the previous frame pointer repeatedly) we can identify the calls leading up to a point in execution. At the same time, the return addresses are captured and printed. Note that we stop traversing this chain of calling when the file pointer moves out of the range of the stack of the process.

3.1.4 xv6-riscv/kernel/sysproc.c

The only modification in this file is to add a call to the backtrace() inside the sys_sleep() function for use in the testing program.

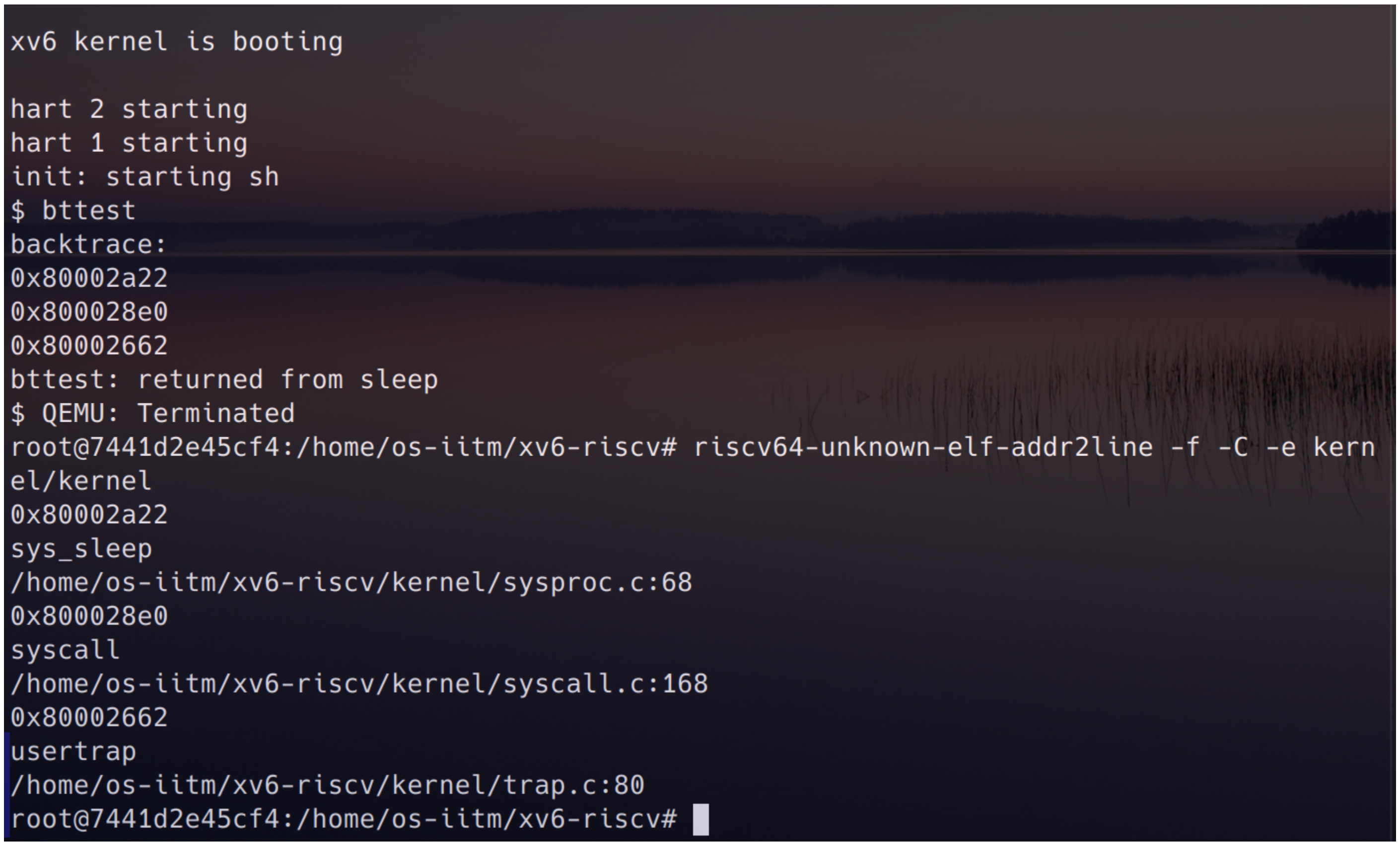
3.1.5 xv6-riscv/user/bttest.c and xv6-riscv/Makefile

This is the testing code given in the handout for the assignment. All that it performs is to call the sleep system call, which inturn calls the backtrace. The expected chain of events would then be sys_sleep -¿ syscall -¿ traphandler(usertrap), which is what is seen in the output. Additionally, an entry for bttest must also be made under the UPROGS section of the Makefile.

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

int
main(int argc, char *argv[])
{
  // Call sleep(1) so sys_sleep() runs in kernel and triggers
  sleep(1);
  printf("bttest: returned from sleep\n");
  exit(0);
}
```

3.2 Execution and Output

The code is executed just as illustrated in the examples and the return addresses decoded using the gdb utility. They can be seen in the screenshot below:



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bttest
backtrace:
0x80002a22
0x800028e0
0x80002662
bttest: returned from sleep
$ QEMU: Terminated
root@7441d2e45cf4:/home/os-iitm/xv6-riscv# riscv64-unknown-elf-addr2line -f -C -e kern
el/kernel
0x80002a22
sys_sleep
/home/os-iitm/xv6-riscv/kernel/sysproc.c:68
0x800028e0
syscall
/home/os-iitm/xv6-riscv/kernel/syscall.c:168
0x80002662
usertrap
/home/os-iitm/xv6-riscv/kernel/trap.c:80
root@7441d2e45cf4:/home/os-iitm/xv6-riscv#
```

Note that the output matches exactly with what is expected from the program and also the example given in the handout.