

CS3500: Operating Systems

Lab 5: Traps and System Calls

September 18, 2020

Introduction

In the previous labs, we became familiar with system calls and traps. We also learnt the paging mechanism in xv6. This lab will put all those pieces together. Firstly, we will look at a debugger called **qemu-gdb** and get some insights into RISC-V assembly. Thereafter, we will design a **tracing and alert mechanism** in xv6.

Resources

Please go through the following resources before beginning this lab assignment:

1. The **xv6 book: Chapter 4 (Traps and System Calls)**: sections **4.1, 4.2, 4.5**
2. Source files: `kernel/trampoline.S` and `kernel/trap.c`

Note

As part of this assignment, we have provided a clean version of the xv6 repo, with the required files included in it. Please implement your solutions in this repo only. We have also attached the L^AT_EX template of this document. Please write your answers in this file and submit the generated PDF (NOT the `.tex`).

1 Avengers, Assemble! (20 points)

For this section, it will be important to understand a bit of RISC-V assembly.

There is a file named `user/call.c` as part of the provided xv6 repo. Modify the **Makefile** suitably to allow `user/call.c` to be compiled as a user program in xv6. Run the command `make fs.img`, which compiles `user/call.c` (among other files) and produces a readable assembly version of the program in `user/call.asm`. Read the assembly code in `user/call.asm` for the functions `g()`, `f()`, and `main()`. Here are some questions that you should answer:

1. (3 points) Which registers contain arguments to functions? For example, which register holds 13 in `main()`'s call to `printf()`?

Solution:

The argument registers which are a0, a1, a2, a3, a4, a5, a6, a7 are the 8 registers used to store arguments of a function in order. the argument 13 for printf is present in a2.

2. (2 points) Where is the function call to `f()` from `main()`? Where is the call to `g()`? (**HINT:** the compiler may inline functions.)

Solution:

Here `g()` is inlined into `f()` by the compiler as seen in line 14 of the code.

```
int f(int x) {
    e: 1141                addi sp,sp,-16
    10: e422                sd s0,8(sp)
    12: 0800                addi s0,sp,16
    return g(x);
}
    14: 250d                addiw a0,a0,3
    16: 6422                ld s0,8(sp)
    18: 0141                addi sp,sp,16
    1a: 8082                ret
```

The function `f` is not called from `main` because as all the arguments are constants the, compiler has optimised the code to the final value (12) of argument directly given to `printf` and so that also is not called from `main`.

3. (2 points) At what address is the function `printf()` located?

Solution:

`printf()` function is located at address 000000000000005b8.

4. (2 points) What value is in the register `ra` just after the `jalr` to `printf()` in `main()`?

Solution:

Just after `jalr`, the value in `ra` is the return address which is 0000000000000038.

5. (11 points) Run the following code.

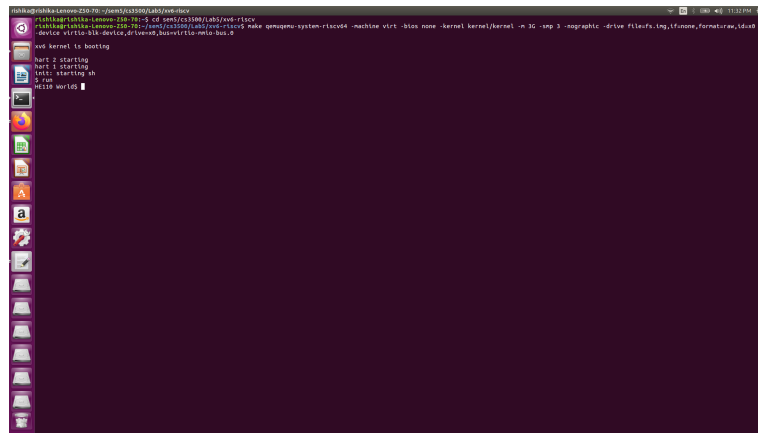
```
unsigned int i = 0x00646c72;
printf("H%x Wo%s", 57616, &i);
```

- (a) (3 points) What is the output? Here's an [ASCII table](#) that maps bytes to characters.

Solution: Result:

HE110 World

Obtained output:



- (b) (5 points) The above output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian, what would you set `i` to in order to yield the same output? Would you need to change 57616 to a different value? Here's a description of [little- and big-endian](#).

Solution: Here the value 57616 need not be changed because it is just getting converted to hexadecimal which is the same everywhere. The difference between little endian and big endian is the way each integer is stored in the form of binary numbers in the 4 bytes that are allocated. Little endian stores least significant 8 bits in the first byte and so on whereas big endian stores the most significant. Here 00, 64, 6c, 72 are stored in each byte in the reverse order and they correspond to string end character , d, l and r respectively. In case of big endian the same number prints these values in the opposite order. Therefore to get it in the right order the number should be changed such that the order of bytes is in the reverse order to that of little endian. So effectively `i` value should be 0x726c6400.

- (c) (3 points) In the following code, what is going to be printed after '`y=`'? (Note: the answer is not a specific value.) Why does this happen?

```
printf("x=%d y=%d", 3);
```

Solution: After '*y* =' random values get printed. This is because when the compiler sees a *%d* as part of the string it retrieves another argument using *va_arg()*. If there are no more arguments to be retrieved then the function exhibits random behaviour returning some unknown value which isn't specific. Thus it prints that random value.

2 The Retreat (30 points)

When something goes wrong, it is often helpful to look back and see what events led to the current predicament. In debugging terminology, we call this introspection a **backtrace**. Consider a code that dereferences a null pointer, which means it cannot execute any further due to the resulting kernel panic. While working with xv6, you may have encountered (or will encounter) such panics.

In each stack frame, the compiler puts a frame pointer that holds the address of the caller's frame pointer. We can design a **backtrace()** function using these frame pointers to walk the stack back up and print the saved return address in each stack frame. The GCC compiler, for instance, stores the frame pointer of the currently executing function in the register **s0**.

1. (30 points) In this section, you need to implement **backtrace()**. Feel free to refer to the hints provided at the end of this section.
 - (a) (20 points) Implement the **backtrace()** function in **kernel/printf.c**. Insert a call to this function in **sys_sleep()** in **kernel/sysproc.c** just before the **return** statement (you may comment out this line after you are done with this section). There is a user program **user/bttest.c** as part of the provided xv6 repo. Modify the **Makefile** accordingly and then run **bttest**, which calls **sys_sleep()**. Here is a sample output (you may get slightly different addresses):

```
$ bttest
backtrace:
0x0000000080002c1a
0x0000000080002a3e
0x00000000800026ba
```

What are the steps you followed? What is the output that you got?

Solution:

I added function definition of **backtrace** to **defs.h** and created a function *r_fp* in **riscv.h** which returns the value of **s0** (current frame pointer). I defined **back trace** in **printf.h** and in it I first got the value of frame pointer using *r_fp* and using that I iteratively found return addresses and previous fp values using the idea that return address of that frame is stored at address **fp-8** and previous fp value is stored at **fp-16** and the respective values can be found by dereferencing the addresses. As a stack is allocated only 1 page and we know that fp is in that page it is clear that the stack can only range from top

```
Terminal output:
$ bttest
backtrace:
0x0000000080002b38
0x000000008000299c
0x0000000080002686
```

```

root@kali:~/Documents# cat /usr/bin/ls
#!/usr/bin/perl
#ls: list directory contents.
#
#Usage: ls [-ldltFm] [-r] [-a] [-l] [-h] [-s] [-i] [-c] [-x] [-k] [-K] [-n] [-q] [-p] [-P] [-R] [-t] [-u] [-v] [-w] [-W] [-y] [-Z] [-z] [-1] [-2] [-3] [-4] [-5] [-6] [-7] [-8] [-9] [-0] [-10] [-11] [-12] [-13] [-14] [-15] [-16] [-17] [-18] [-19] [-20] [-21] [-22] [-23] [-24] [-25] [-26] [-27] [-28] [-29] [-30] [-31] [-32] [-33] [-34] [-35] [-36] [-37] [-38] [-39] [-40] [-41] [-42] [-43] [-44] [-45] [-46] [-47] [-48] [-49] [-50] [-51] [-52] [-53] [-54] [-55] [-56] [-57] [-58] [-59] [-60] [-61] [-62] [-63] [-64] [-65] [-66] [-67] [-68] [-69] [-70] [-71] [-72] [-73] [-74] [-75] [-76] [-77] [-78] [-79] [-80] [-81] [-82] [-83] [-84] [-85] [-86] [-87] [-88] [-89] [-90] [-91] [-92] [-93] [-94] [-95] [-96] [-97] [-98] [-99] [-100] [-101] [-102] [-103] [-104] [-105] [-106] [-107] [-108] [-109] [-110] [-111] [-112] [-113] [-114] [-115] [-116] [-117] [-118] [-119] [-120] [-121] [-122] [-123] [-124] [-125] [-126] [-127] [-128] [-129] [-130] [-131] [-132] [-133] [-134] [-135] [-136] [-137] [-138] [-139] [-140] [-141] [-142] [-143] [-144] [-145] [-146] [-147] [-148] [-149] [-150] [-151] [-152] [-153] [-154] [-155] [-156] [-157] [-158] [-159] [-160] [-161] [-162] [-163] [-164] [-165] [-166] [-167] [-168] [-169] [-170] [-171] [-172] [-173] [-174] [-175] [-176] [-177] [-178] [-179] [-180] [-181] [-182] [-183] [-184] [-185] [-186] [-187] [-188] [-189] [-190] [-191] [-192] [-193] [-194] [-195] [-196] [-197] [-198] [-199] [-200] [-201] [-202] [-203] [-204] [-205] [-206] [-207] [-208] [-209] [-210] [-211] [-212] [-213] [-214] [-215] [-216] [-217] [-218] [-219] [-220] [-221] [-222] [-223] [-224] [-225] [-226] [-227] [-228] [-229] [-230] [-231] [-232] [-233] [-234] [-235] [-236] [-237] [-238] [-239] [-240] [-241] [-242] [-243] [-244] [-245] [-246] [-247] [-248] [-249] [-250] [-251] [-252] [-253] [-254] [-255] [-256] [-257] [-258] [-259] [-260] [-261] [-262] [-263] [-264] [-265] [-266] [-267] [-268] [-269] [-270] [-271] [-272] [-273] [-274] [-275] [-276] [-277] [-278] [-279] [-280] [-281] [-282] [-283] [-284] [-285] [-286] [-287] [-288] [-289] [-290] [-291] [-292] [-293] [-294] [-295] [-296] [-297] [-298] [-299] [-300] [-301] [-302] [-303] [-304] [-305] [-306] [-307] [-308] [-309] [-310] [-311] [-312] [-313] [-314] [-315] [-316] [-317] [-318] [-319] [-320] [-321] [-322] [-323] [-324] [-325] [-326] [-327] [-328] [-329] [-330] [-331] [-332] [-333] [-334] [-335] [-336] [-337] [-338] [-339] [-340] [-341] [-342] [-343] [-344] [-345] [-346] [-347] [-348] [-349] [-350] [-351] [-352] [-353] [-354] [-355] [-356] [-357] [-358] [-359] [-360] [-361] [-362] [-363] [-364] [-365] [-366] [-367] [-368] [-369] [-370] [-371] [-372] [-373] [-374] [-375] [-376] [-377] [-378] [-379] [-380] [-381] [-382] [-383] [-384] [-385] [-386] [-387] [-388] [-389] [-390] [-391] [-392] [-393] [-394] [-395] [-396] [-397] [-398] [-399] [-400] [-401] [-402] [-403] [-404] [-405] [-406] [-407] [-408] [-409] [-410] [-411] [-412] [-413] [-414] [-415] [-416] [-417] [-418] [-419] [-420] [-421] [-422] [-423] [-424] [-425] [-426] [-427] [-428] [-429] [-430] [-431] [-432] [-433] [-434] [-435] [-436] [-437] [-438] [-439] [-440] [-441] [-442] [-443] [-444] [-445] [-446] [-447] [-448] [-449] [-450] [-451] [-452] [-453] [-454] [-455] [-456] [-457] [-458] [-459] [-460] [-461] [-462] [-463] [-464] [-465] [-466] [-467] [-468] [-469] [-470] [-471] [-472] [-473] [-474] [-475] [-476] [-477] [-478] [-479] [-480] [-481] [-482] [-483] [-484] [-485] [-486] [-487] [-488] [-489] [-490] [-491] [-492] [-493] [-494] [-495] [-496] [-497] [-498] [-499] [-500] [-501] [-502] [-503] [-504] [-505] [-506] [-507] [-508] [-509] [-510] [-511] [-512] [-513] [-514] [-515] [-516] [-517] [-518] [-519] [-520] [-521] [-522] [-523] [-524] [-525] [-526] [-527] [-528] [-529] [-530] [-531] [-532] [-533] [-534] [-535] [-536] [-537] [-538] [-539] [-540] [-541] [-542] [-543] [-544] [-545] [-546] [-547] [-548] [-549] [-550] [-551] [-552] [-553] [-554] [-555] [-556] [-557] [-558] [-559] [-560] [-561] [-562] [-563] [-564] [-565] [-566] [-567] [-568] [-569] [-570] [-571] [-572] [-573] [-574] [-575] [-576] [-577] [-578] [-579] [-580] [-581] [-582] [-583] [-584] [-585] [-586] [-587] [-588] [-589] [-590] [-591] [-592] [-593] [-594] [-595] [-596] [-597] [-598] [-599] [-600] [-601] [-602] [-603] [-604] [-605] [-606] [-607] [-608] [-609] [-610] [-611] [-612] [-613] [-614] [-615] [-616] [-617] [-618] [-619] [-620] [-621] [-622] [-623] [-624] [-625] [-626] [-627] [-628] [-629] [-630] [-631] [-632] [-633] [-634] [-635] [-636] [-637] [-638] [-639] [-640] [-641] [-642] [-643] [-644] [-645] [-646] [-647] [-648] [-649] [-650] [-651] [-652] [-653] [-654] [-655] [-656] [-657] [-658] [-659] [-660] [-661] [-662] [-663] [-664] [-665] [-666] [-667] [-668] [-669] [-670] [-671] [-672] [-673] [-674] [-675] [-676] [-677] [-678] [-679] [-680] [-681] [-682] [-683] [-684] [-685] [-686] [-687] [-688] [-689] [-690] [-691] [-692] [-693] [-694] [-695] [-696] [-697] [-698] [-699] [-700] [-701] [-702] [-703] [-704] [-705] [-706] [-707] [-708] [-709] [-710] [-711] [-712] [-713] [-714] [-715] [-716] [-717] [-718] [-719] [-720] [-721] [-722] [-723] [-724] [-725] [-726] [-727] [-728] [-729] [-730] [-731] [-732] [-733] [-734] [-735] [-736] [-737] [-738] [-739] [-740] [-741] [-742] [-743] [-744] [-745] [-746] [-747] [-748] [-749] [-750] [-751] [-752] [-753] [-754] [-755] [-756] [-757] [-758] [-759] [-760] [-761] [-762] [-763] [-764] [-765] [-766] [-767] [-768] [-769] [-770] [-771] [-772] [-773] [-774] [-775] [-776] [-777] [-778] [-779] [-780] [-781] [-782] [-783] [-784] [-785] [-786] [-787] [-788] [-789] [-790] [-791] [-792] [-793] [-794] [-795] [-796] [-797] [-798] [-799] [-800] [-801] [-802] [-803] [-804] [-805] [-806] [-807] [-808] [-809] [-810] [-811] [-812] [-813] [-814] [-815] [-816] [-817] [-818]
```

- Solution:** Command used: `addr2line -e kernel/kernel`
Here `kernel/kernel` is the executable.
Output:

[illegible]

·
·
·

- You may face some issues in terminating the backtrace. Note that xv6 allocates one page for each stack in the xv6 kernel at PAGE-aligned address. You can compute the top and bottom address of the stack page by using `PGROUNDUP(fp)` and `PGROUNDDOWN(fp)` (see `kernel/riscv.h`). These are helpful for terminating the loop in your `backtrace()`.
2. (30 points) [OPTIONAL] Print the names of the functions and line numbers in `backtrace()` instead of numerical addresses.

3 Wake me up when Sep ... (40 points)

From emails to WhatsApp notifications, we often rely on alerts for certain events. In this section, you will add such an alarm feature to xv6 that alerts a process as it uses CPU time.

1. (2 points) Think of scenarios where such a feature will be useful. Enumerate them.

Solution: This alarm can be used in features that involve process monitoring. For example if we want to measure the amount of memory or resources used by a process then we need an alarm to be sent whenever that particular process is being run by the cpu so that the resource can be kept track by the feature until there is a context switch. Process monitoring may involve time taken by the process as well.

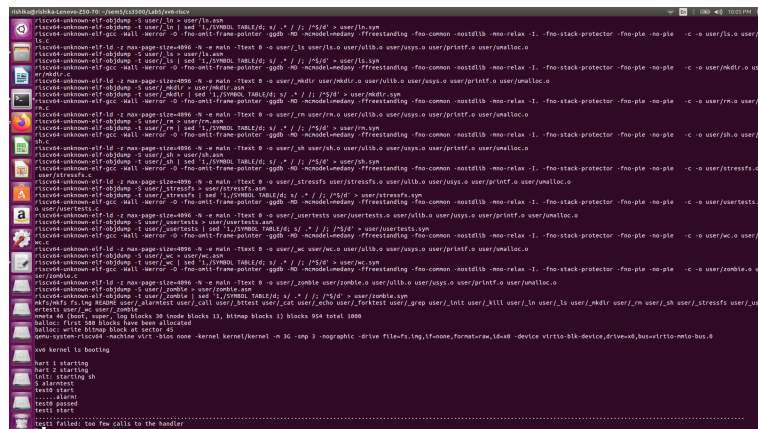
2. (38 points) More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers. You could use something similar to handle page faults in the application, for example. Feel free to refer to the hints at the end of this section.
 - (a) (10 points) Add a new `sigalarm(interval, handler)` system call. If an application calls `sigalarm(n, fn)`, then after every `n` "ticks" of CPU time that the program consumes, the kernel should cause the application function `fn` to be called. (A "tick" is a fairly arbitrary unit of time in xv6, determined by how often a hardware timer generates interrupts.) For the time being, create a simple `sigreturn()` system call with a `return 0;` statement.

HINT: You need to make sure that the handler is invoked when the process's alarm interval expires. You'll need to modify `usertrap()` in `kernel/trap.c` so that when a process's alarm interval expires, the process executes the handler. To this end, you will need to recall how system calls work from the previous labs (i.e., the code in `kernel/trampoline.S` and `kernel/trap.c`). Mention your approach as the answer below. Which register contains the user-space instruction address to which system calls return?

Solution:

I added 3 fields to the `proc.h`: `tick` which signifies count of ticks at present, `gap` which is the value `n` being the number of ticks after which the function handler is to be called, `fpoi` which is the function handler `fn` that is to be called. I made 2 system calls `sigalarm` and `sigreturn`. `sigreturn` just returns for now. In `sigalarm`, the arguments are `n`, `fn` and these values are allocated to the respective fields of that `proc`. Initially the values of `tick`, `gap` are 0 and -1 (so that it can't be equal to `tick` unless it gets a value through `sigreturn`) respectively and allocated by `allocproc`. When there is a timer interrupt and the control goes to the `usertrap`, I incremented the value of `tick` each time and if the value of `tick` is equal to `gap` value I changed the address to which it has to return (stored in `tf->epc`) to the function handler pointer. Thus it will jump to that function handler when this process is next executed by the cpu. The register that contains user-space instruction address to which system calls return is `epc`.

Output:



- (b) (8 points) Complete the `sigreturn()` system call, which ensures that when the function `fn` returns, the application resumes where it left off.

As a starting point: user alarm handlers are required to call the `sigreturn()` system call when they have finished. Have a look at the `periodic()` function in `user/alarmtest.c` for an example. You should add some code to `usertrap()` in `kernel/trap.c` and your implementation of `sys_sigreturn()` that cooperate to cause the user process to resume properly after it has handled the alarm.

Your solution will require you to save and restore registers. Mention your approach as the answer below. What registers do you need to save and restore to resume the interrupted code correctly? (**HINT**: it will be many).

Solution:

The control goes to `sigreturn` at the end of the function handler and so the control must return to the original return user instruction of the application from which the interrupt happened. To do this, I created a new field `kcontext`

in proc to temporarily store trap frame and before changing the $tf \rightarrow epc$ I stored the original trapframe struct values in kcontext and restored these values to tf in sigreturn. I also changed value of tick back to 0 in sigreturn.

Output:

- (c) (20 points) There is a file named `user/alarmtest.c` in the xv6 repository we have provided. This program checks your solution against three test cases. `test0` checks your `sigalarm()` implementation to see whether the alarm handler is called at all. `test1` and `test2` check your `sigreturn()` implementation to see whether the handler correctly returns to the point in the application program where the timer interrupt occurred, with all registers holding the same values they held when the interrupt occurred. You can see the assembly code for `alarmtest` in `user/alarmtest.asm`, which may be handy for debugging.

Once you have implemented your solution, modify `Makefile` accordingly and then run `alarmtest`. If it passes `test0`, `test1` and `test2`, run `usertests` to make sure you didn't break any other parts of the kernel. Following is a sample output of `alarmtest` and `usertests` if the alarm invocation and return have been handled correctly.

```
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
```

```

test1 passed
test2 start
.....alarm!
test2 passed
$ usertests
...
ALL TESTS PASSED
$

```

3.1 Additional hints for test cases

test0: Invoking the handler

Get started by modifying the kernel to jump to the alarm handler in user space, which will cause `test0` to print “alarm!”. At this stage, ignore if the program crashes after this. Following are some hints:

- The right declarations to put in `user/user.h` are:

```

int sigalarm(int ticks, void (*handler)());
int sigreturn(void);

```

- Recall from your previous labs the changes that need to be made for system calls.
- `sys_sigalarm()` should store the alarm interval and the pointer to the handler function in new fields in `struct proc` (in `kernel/proc.h`).
- To keep track of the number of ticks passed since the last call (or are left until the next call) to a process’s alarm handler, add a new field in `struct proc` for this too. You can initialize `proc` fields in `allocproc()` in `kernel/proc.c`.
- Every tick, the hardware clock forces an interrupt, which is handled in `usertrap()` in `kernel/trap.c`. You should add some code there to modify a process’s alarm ticks, but only in the case of a timer interrupt, something like:

```

if(which_dev == 2) ...

```

- It will be easier to look at traps with `gdb` if you configure `QEMU` to use only one CPU, which you can do by running:

```

make CPUS=1 qemu-gdb

```

test1/test2: Resuming interrupted code

Most probably, your `alarmtest` crashes in `test0` or `test1` after it prints “alarm!”, or `alarmtest` (eventually) prints “test1 failed”, or `alarmtest` exits without printing “test1 passed”. To fix this, you must ensure that, when the alarm handler is done, control returns to the instruction at which the user program was originally interrupted by the timer interrupt. You must ensure that the register contents are restored to the

values they held at the time of the interrupt, so that the user program can continue undisturbed after the alarm. Finally, you should “re-arm” the alarm counter after each time it goes off, so that the handler is called periodically. Here are some hints:

- Have `usertrap()` save enough state in `struct proc` when the timer goes off, so that `sigreturn()` can correctly return to the interrupted user code.
- Prevent re-entrant calls to the handler: if a handler hasn’t returned yet, the kernel shouldn’t call it again. `test2` tests this.

Submission Guidelines

1. Implement your solutions in the provided xv6 folder. Write your answers in the attached L^AT_EX template, convert it to PDF and name it as `YOUR.ROLL_NO.pdf`. This will serve as a report for the assignment.
2. Put your entire solution xv6 folder, and the `YOUR.ROLL_NO.pdf` in a common folder named `YOUR.ROLL_NO.LAB5`.
3. Compress the folder `YOUR.ROLL_NO.LAB5` into `YOUR.ROLL_NO.LAB5.tar.gz` and submit the compressed folder on Moodle.
4. NOTE: Make sure to run `make clean`, delete any additional manual and the `.git` folder from the xv6 folder before submitting.