Traps and System Calls

Trap

- System call
  - e.g. ecall
- Exception
  - e.g. illegal instruction
  - Kernel kills the offending program
- Device interrupt
  - e.g. disk finishes read/write

1. Trap forces transfer of control into the kernel
2. Kernel executes appropriate handler code (save then restore previous state)
3. Return to user code where original code resumes (without needing to know anything special happened)

RISC-V Trap Machinery

- stvec : address of the trap handler
- sepc: program counter saved (pc is overwritten with value in stvec)
- scause: trap reason (number)
- sscratch: avoid overwriting user registers
- sstatus: control whether device interrupts are enabled

Traps from User Space

uservec ➡️ usertrap ➡️ ..trap handling.. ➡️ usertrapret ➡️ userret

- Trampoline page : uservec
- Uservec: save 32 register values (using sscratch register)
- Usertrap: determine case of the trap, process, and return
- Usertrapret: prepare for future trap from user space

Page Fault Exceptions

Exception handling in xv6:

- User space: kills the faulting process
- Kernel: kernel panics

Page Fault:

- Load page fault
- Store page fault
- Instruction page fault

RISC-V Assembly

```c
int g(int x) {
  return x+3;
}

int f(int x) {
  return g(x);
}

void main(void) {
  printf("%d %d\n", f(8)+1, 13);
  exit(0);
}
```

```
0000000000000000 <g>:
int g(int x) {
   0:    1141                    addi    sp,sp,-16
   2:    e406                    sd    ra,8(sp)
   4:    e022                    sd    s0,0(sp)
   6:    0800                    addi    s0,sp,16
  return x+3;
}
   8:    250d                    addiw    a0,a0,3
   a:    60a2                    ld    ra,8(sp)
   c:    6402                    ld    s0,0(sp)
   e:    0141                    addi    sp,sp,16
  10:    8082                    ret
0000000000000012 <f>:
int f(int x) {
  12:    1141                    addi    sp,sp,-16
  14:    e406                    sd    ra,8(sp)
  16:    e022                    sd    s0,0(sp)
  18:    0800                    addi    s0,sp,16
  return g(x);
}
  1a:    250d                    addiw    a0,a0,3
  1c:    60a2                    ld    ra,8(sp)
  1e:    6402                    ld    s0,0(sp)
  20:    0141                    addi    sp,sp,16
  22:    8082                    ret
```

```
0000000000000024 <main>:
void main(void) {
  24:    1141                    addi    sp,sp,-16
  26:    e406                    sd    ra,8(sp)
  28:    e022                    sd    s0,0(sp)
  2a:    0800                    addi    s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  2c:    4635                    li    a2,13
  2e:    45b1                    li    a1,12
  30:    00001517              auipc    a0,0x1
  34:    86050513              addi    a0,a0,-1952 # 890
<malloc+0xf6>
  38:    6aa000ef              jal    6e2 <printf>
  exit(0);
  3c:    4501                    li    a0,0
  3e:    2a2000ef              jal    2e0 <exit>
**
```

1. Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?
-> Registers a0-a7.

2. Where is the call to function f in the assembly code for main? Where is the call to g? (Hint: the compiler may inline functions.)
-> No explicit calls are made.

3. At what address is the function printf located?
-> 0x6e2

4. What value is in the register ra just after the jalr to printf in main?
-> address 0x3c -- the address of the next instruction.

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

5. What is the output of the given code?
-> HE110 World
57616 in hex is 0xE110
0x00646c72 in memory is stored as
72:(ASCII) r, 6c:l, 64:d, 00:\0 (ordered by lower to higher address)

For big endian, i has to be set differently.
i = 0x726c6400 would be
00:\0, 64:d, 6c:l, 72:r, and big endian will store bytes from higher address.

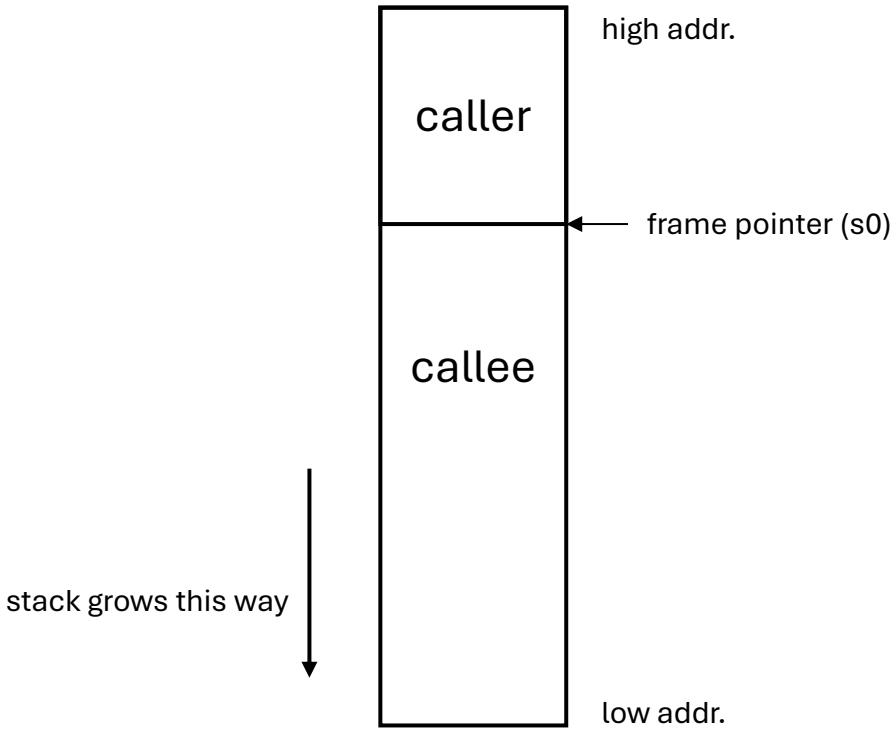
```
printf("x=%d y=%d", 3);
```

6. What is going to be printed after 'y='? Why does this happen?
-> some garbage value

Backtrace

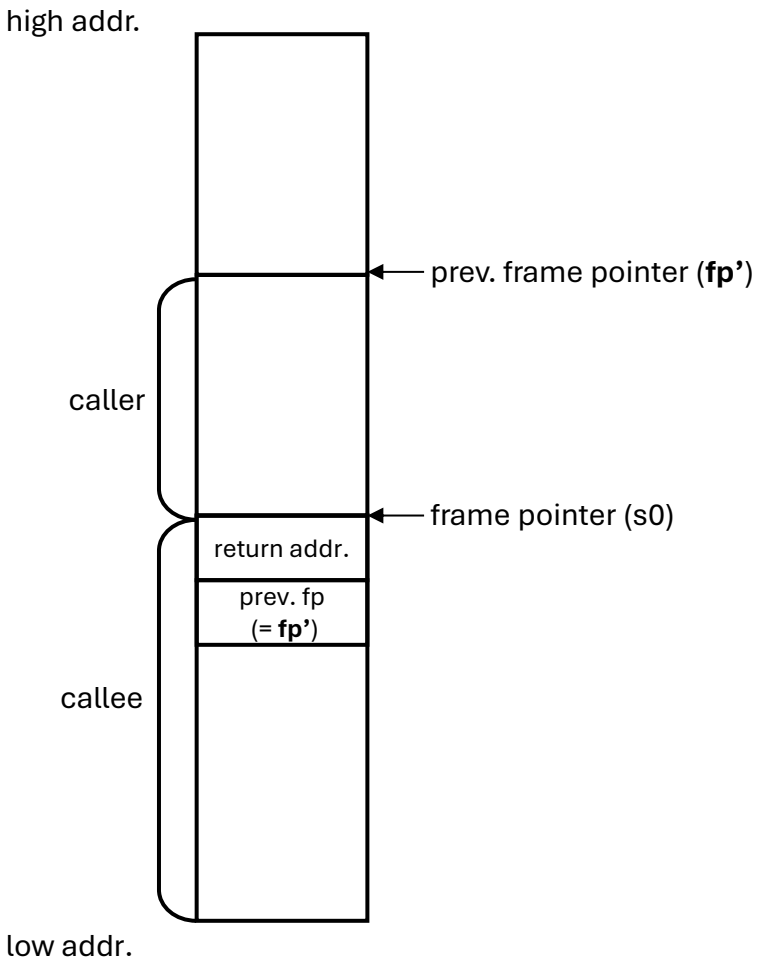
Reading from registers – read frame pointer

- s0 contains the current frame pointer (memory address)
- frame pointer points to the bottom of the callee stack




```
28 + // current value of the fp
29 + static inline uint64
30 + r_fp()
31 + {
32 +   uint64 x;
33 +   asm volatile("mv %0, s0" : "=r"(x));
34 +   return x;
35 + }
36 +
```

Restoring Return Address and prev. FP




```
62 + static inline uint64 get_return_addr(uint64 fp)
63 + {
64 +   return *((uint64*)(fp - 8));
65 + }
66 +
67 + static inline uint64 get_saved_fp(uint64 fp)
68 + {
69 +   return *((uint64*)(fp - 16));
70 + }
71 +
```

Walk up the stack until fp is out of the current page

```
72 + void backtrace()
73 + {
74 +   uint64 fp = r_fp();                // fp when backtrace() called
75 +   const uint64 pg = PGROUNDDOWN(fp); // page of the kernel stack
76 +   printf("backtrace:\n");
77 +   do {
78 +     printf("%p\n", (void*)get_return_addr(fp));
79 +     fp = get_saved_fp(fp); // restore saved fp
80 +   } while (pg == PGROUNDDOWN(fp));
81 + }
82 +
```

Alarm – Yi, Cherry

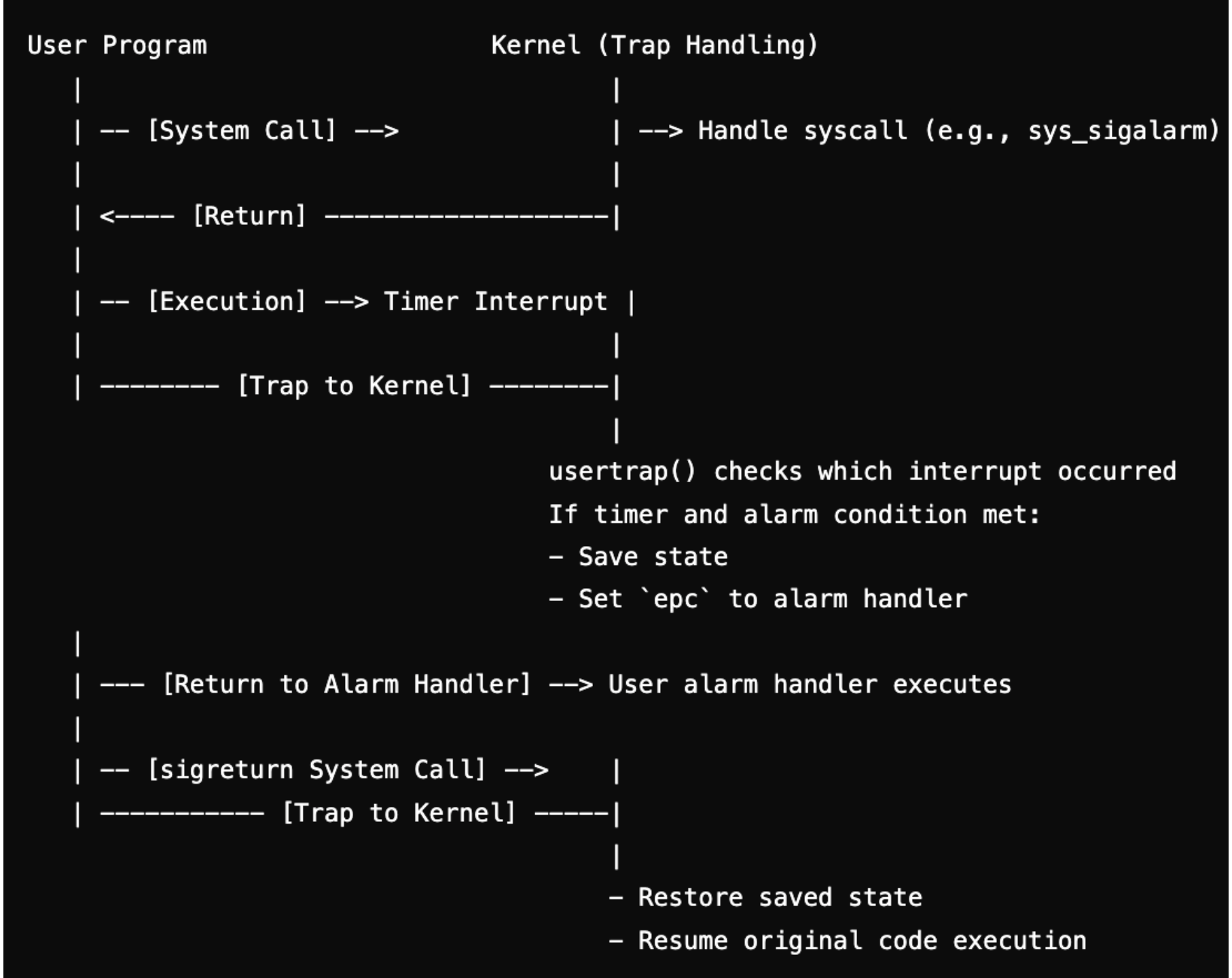
Requirements

- System Call Implementation
  - Sigalarm(ticks, handler)
    - Kernel should trigger the handler ftn every ticks
    - If sigalarm(0,0) is called, periodic alarms should stop
  - Sigreturn( )
    - Handler ftn must call sigreturn to restore the process's state after execution

```
35 void
36 periodic()
37 {
38   count = count + 1;
39   printf("alarm!\n");
40   sigreturn();
41 }
42
```

```
146 void
147 slow_handler()
148 {
149   count++;
150   printf("alarm!\n");
151   if (count > 1) {
152     printf("test2 failed: alarm handler called more than once\n");
153     exit(1);
154   }
155   for (int i = 0; i < 1000*500000; i++) {
156     asm volatile("nop"); // avoid compiler optimizing away loop
157   }
158   sigalarm(0, 0);
159   sigreturn();
160 }
161
```

```
165 void
166 dummy_handler()
167 {
168   sigalarm(0, 0);
169   sigreturn();
170 }
171
```


User Program
Kernel (Trap Handling)
-- [System Call] -->
--> Handle syscall (e.g., sys_sigalarm)
<---- [Return] ----------
-- [Execution] --> Timer Interrupt
-------- [Trap to Kernel] --------
usertrap() checks which interrupt occurred
If timer and alarm condition met:
- Save state
- Set `epc` to alarm handler
--- [Return to Alarm Handler] --> User alarm handler executes
-- [sigreturn System Call] -->
---------- [Trap to Kernel] -----
- Restore saved state
- Resume original code execution

Test case & Key points to implement correctly

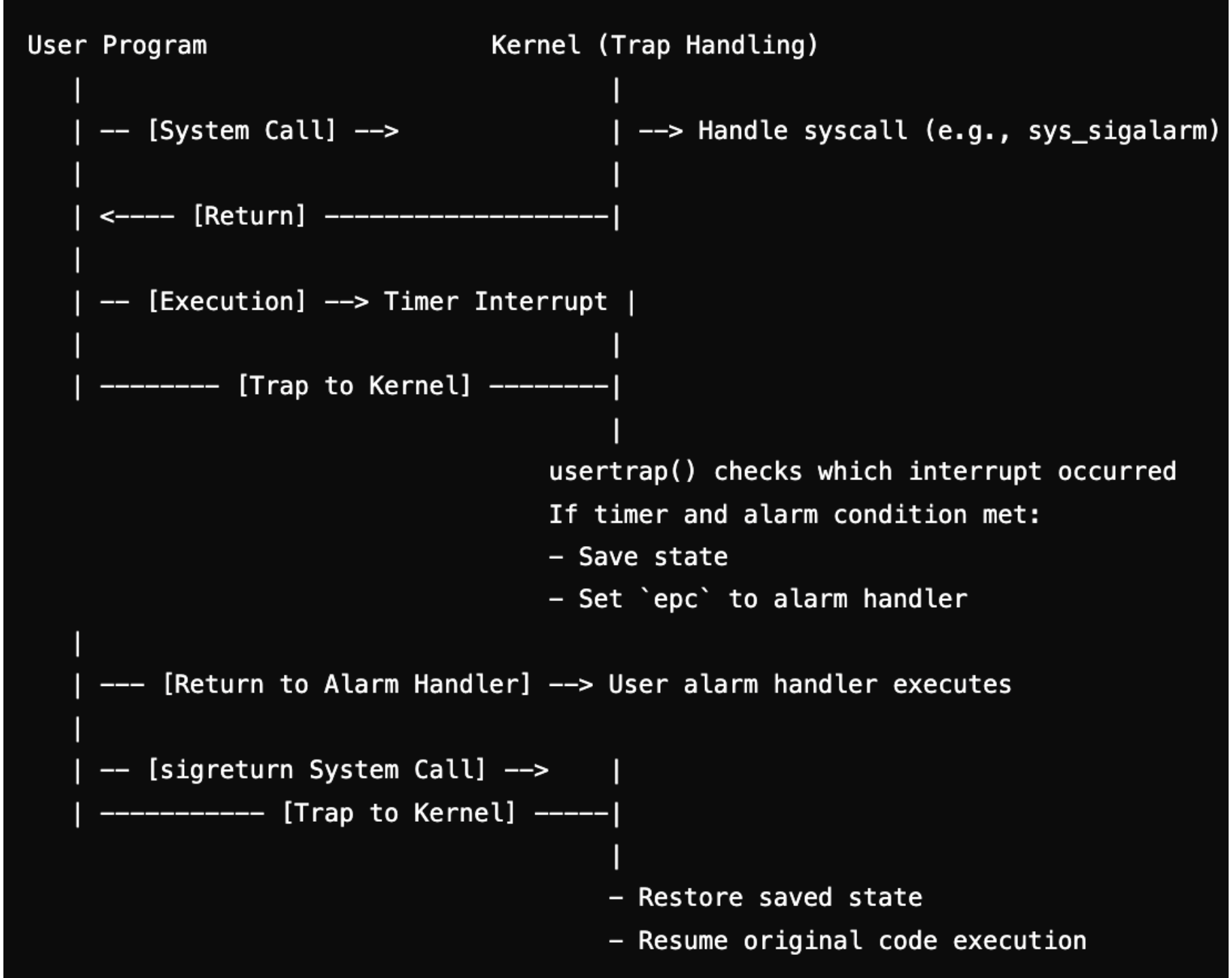
- Test0: Basic Alarm Handler Execution
- Test1: Multiple Alarm Handler Calls & Correct State Restoration
- Test2: Reentrant Alarm Prevention
  - `slow_handler()`는 오래 실행되며, 실행 중에 두 번째로 호출되면 안 됨.
- Test3: Register `a0` Preservation
  - `sigreturn()`이 호출될 때 레지스터 값이 변조되지 않으면 테스트 통과.

Test case & Key points to implement correctly

- **Timer Interrupt :** check if it is
- **State Management :** save the process state / restore when handler finishes
- **Re-entrant Handler Prevention :** use flag

Modified files & function

- Proc.c - sigalarm, sigreturn
  - 프로세스의 주요 동작 및 상태 관리 관련 기능을 제공.
- Sysproc.c - sys_sigalarm, sys_sigreturn
  - 사용자 프로그램에서 호출하는 시스템 콜 처리.
- Trap.c - usertrap
  - 예외, 인터럽트, 시스템 콜과 같은 트랩을 처리.


User Program
Kernel (Trap Handling)
-- [System Call] -->
--> Handle syscall (e.g., sys_sigalarm)
<---- [Return] ----------
-- [Execution] --> Timer Interrupt
-------- [Trap to Kernel] --------
usertrap() checks which interrupt occurred
If timer and alarm condition met:
- Save state
- Set `epc` to alarm handler
--- [Return to Alarm Handler] --> User alarm handler executes
-- [sigreturn System Call] -->
---------- [Trap to Kernel] -----
- Restore saved state
- Resume original code execution

Proc.c - allocproc

```
149   // * Labs4 - #3
150   // Set up alarm fields
151   p->alarm_req = 0;        // alarm requred = 1, not = 0
152   p->alarm_ticks = 0;      // alarm unit(period)
153   p->ticks_cnt = 0;        // current tick cnt (to check it reaches the top(of tick))
154   p->alarm_handler = 0;    // handler address
155   p->alarm_active = 0;     // alarm active=1, inactive=0
156   memset(&p->saved_trapframe, 0, sizeof(p->saved_trapframe)); // initialize trapframe
```

Sysproc.c - handle syscall

```
// * labs4 - #3
uint64
sys_sigalarm(void)
{
  int ticks;
  uint64 handler;

  // get args from user - argraw checks for case
  argint(0, &ticks);
  argaddr(1,&handler);

  // handle disabling the alarm
  if(ticks == 0 && handler == 0)
  {
      myproc()->alarm_req=0;
      myproc()->alarm_active=0;
      myproc()->alarm_ticks=0;
      myproc()->ticks_cnt=0;
      myproc()->alarm_handler=0;
      return 0;
  }

  // call alarm
  sigalarm(ticks, handler);
  return 0;
}

// * labs4 - #3
uint64
sys_sigreturn(void)
{
  sigreturn();
  return 0;
}

```

Proc.h - proc struct

```
108   // * labs4 - #3
109   // alarm fields
110   int alarm_req;
111   int alarm_ticks;
112   int ticks_cnt;
113   uint64 alarm_handler;
114   int alarm_active;
115   struct trapframe saved_trapframe; // copy of registers before interrupt
116   uint64 prev_a0; // value a0 before execution (needed to restore)
117 };
```

Proc.c - sig (system call) ...

```
149   // * labs4 - #3
150   // Set up alarm fields
151   p->alarm_req = 0;       // alarm requred = 1, not = 0
152   p->alarm_ticks = 0;     // alarm unit(period)
153   p->ticks_cnt = 0;       // current tick cnt (to check it reaches the top(of tick))
154   p->alarm_handler = 0;   // handler address
155   p->alarm_active = 0;    // alarm active=1, inactive=0
156   memset(&p->saved_trapframe, 0, sizeof(p->saved_trapframe)); // initialize trapframe
```

```
231 // * labs4 - #3
232 int
233 sigalarm(int ticks, uint64 handler)
234 {
235   struct proc *p = myproc();
236
237   p->alarm_req = 1;
238   p->alarm_ticks = ticks;
239   p->ticks_cnt = 0;
240   p->alarm_handler = handler;
241   p->alarm_active = 0;
242   memset(&p->saved_trapframe, 0, sizeof(p->saved_trapframe));
243
244   return 0;
245 }
246
247 // * labs4 - #3
248 int
249 sigreturn(void)
250 {
251   struct proc *p = myproc();
252
253   p->alarm_active = 0;
254   memmove(p->trapframe, &p->saved_trapframe, sizeof(struct trapframe));
255   p->ticks_cnt = 0;
256
257   return p->trapframe->a0;
258 }
259
```

Trap.c - usertrap()

```c
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

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

    if(killed(p))
      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 sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
    intr_on();

    // * labs4 - #3
    // capture system call number
    uint64 syscall_num = p->trapframe->a7;
    syscall();

    // * if sigreturn is called, restore saved a0 value
    if (syscall_num == SYS_sigreturn)
    {
        p->trapframe->a0 = p->saved_trapframe.a0;
    }

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

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

  // give up the CPU if this is a timer interrupt.
  // * labs4 - #3
  if(which_dev == 2)
  {
    if(p->alarm_req && !p->alarm_active)
    {
        if (++p->ticks_cnt == p->alarm_ticks)
        {
            p->ticks_cnt = 0;
            p->alarm_active = 1;
            memmove(&p->saved_trapframe, p->trapframe, sizeof(struct trapframe));
            p->trapframe->epc = (uint64)p->alarm_handler; // jump to handler
        }
    }
    yield();
  }
  usertrapret();
}
```

Trap.c - usertrap()

```
72      // * labs4 - #3
73      // capture system call number which is called by user
74      uint64 syscall_num = p->trapframe->a7;
75      syscall();
76
77      // * if sigreturn is called, restore saved a0 value
78      if (syscall_num == SYS_sigreturn)
79      {
80          p->trapframe->a0 = p->saved_trapframe.a0;
81      }
```

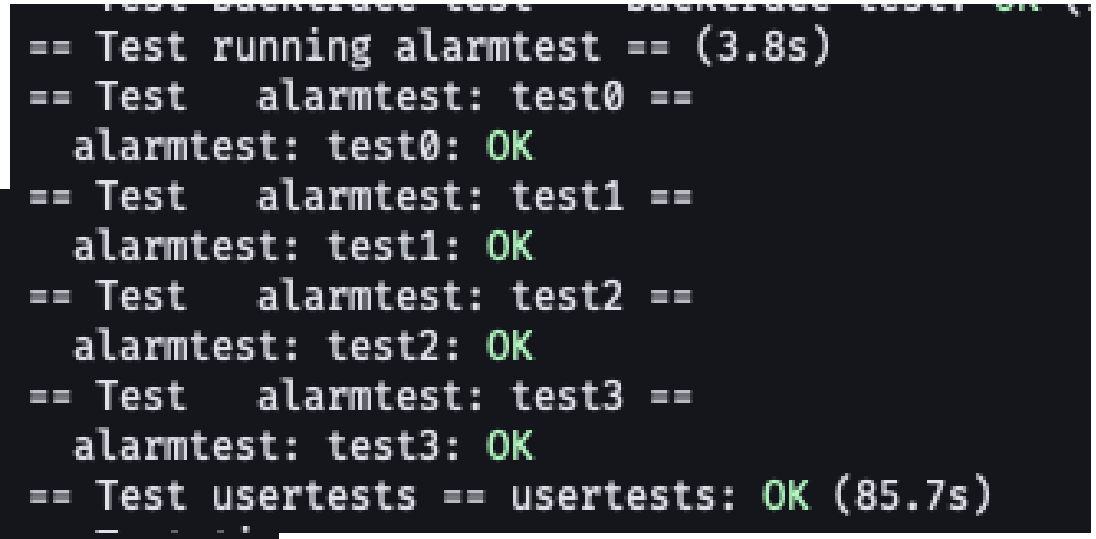
Trap.c - usertrap()

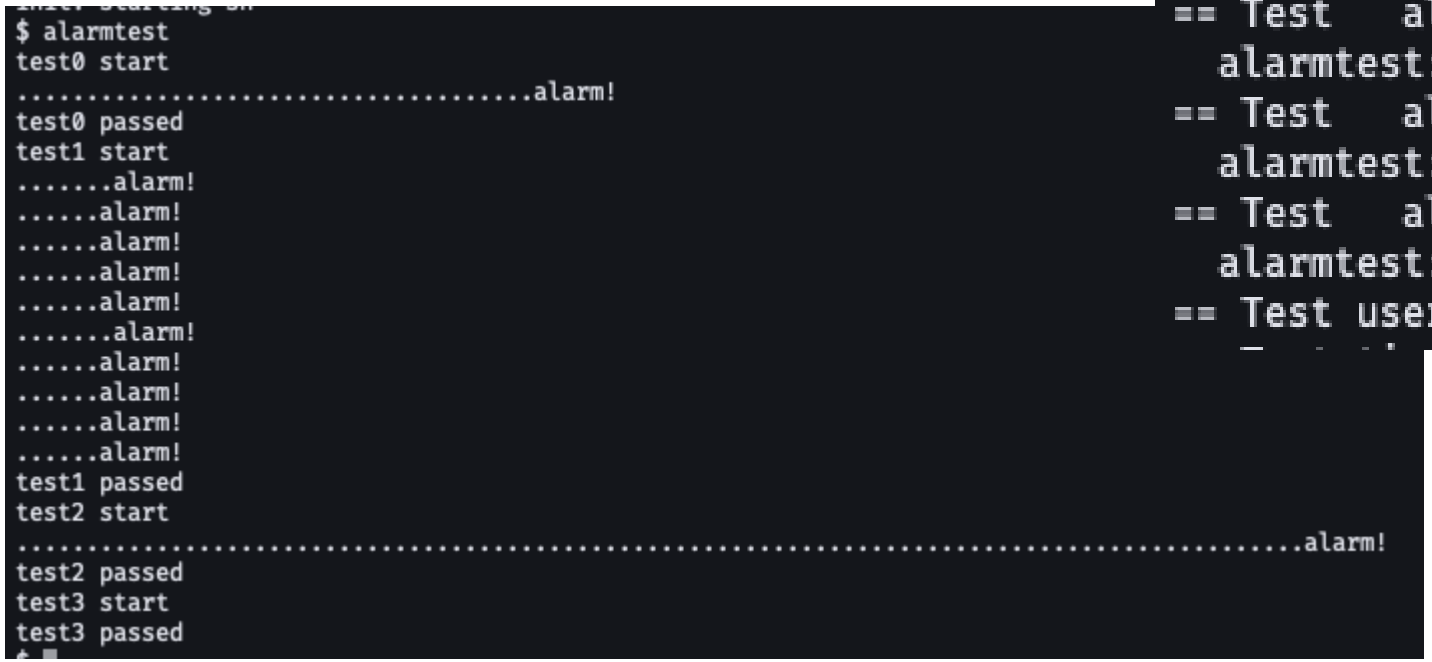
```
 94   // give up the CPU if this is a timer interrupt.
 95   // * labs4 - #3
 96   if(which_dev == 2)
 97   {
 98     if(p->alarm_req && !p->alarm_active) // alarm is requried && not in execution
 99     {
100         if (++p->ticks_cnt == p->alarm_ticks) // increase cnt, and check if it reaches the end of period
101         {
102             p->ticks_cnt = 0; // reset
103             p->alarm_active = 1; // active (prevent re-entrant)
104             memmove(&p->saved_trapframe, p->trapframe, sizeof(struct trapframe)); // save current status
105             p->trapframe->epc = (uint64)p->alarm_handler; // jump to handler
106         }
107     }
108     yield(); // let other process be executed
109   }
110   usertrapret(); // back to user mode (after interrupt or system call)
111 }
```

Test case & Key points to implement correctly

- Test0: Basic Alarm Handler Execution
- Test1: Multiple Alarm Handler Calls & Correct State Restoration
- Test2: Reentrant Alarm Prevention
  - `slow_handler()`는 오래 실행되며, 실행 중에 두 번째로 호출되면 안 됨.
- Test3: Register `a0` Preservation
  - `sigreturn()`이 호출될 때 레지스터 값이 변조되지 않으면 테스트 통과.

Results


== Test running alarmtest == (3.8s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test alarmtest: test3 ==
alarmtest: test3: OK
== Test usertests == usertests: OK (85.7s)


$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
test1 passed
test2 start
.....alarm!
test2 passed
test3 start
test3 passed