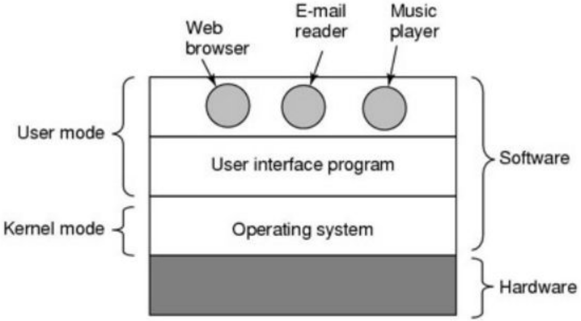
January 2025 CSE 314 - OS Sessional

Xv6 Introduction
Nafis Tahmid
May 07, 2025

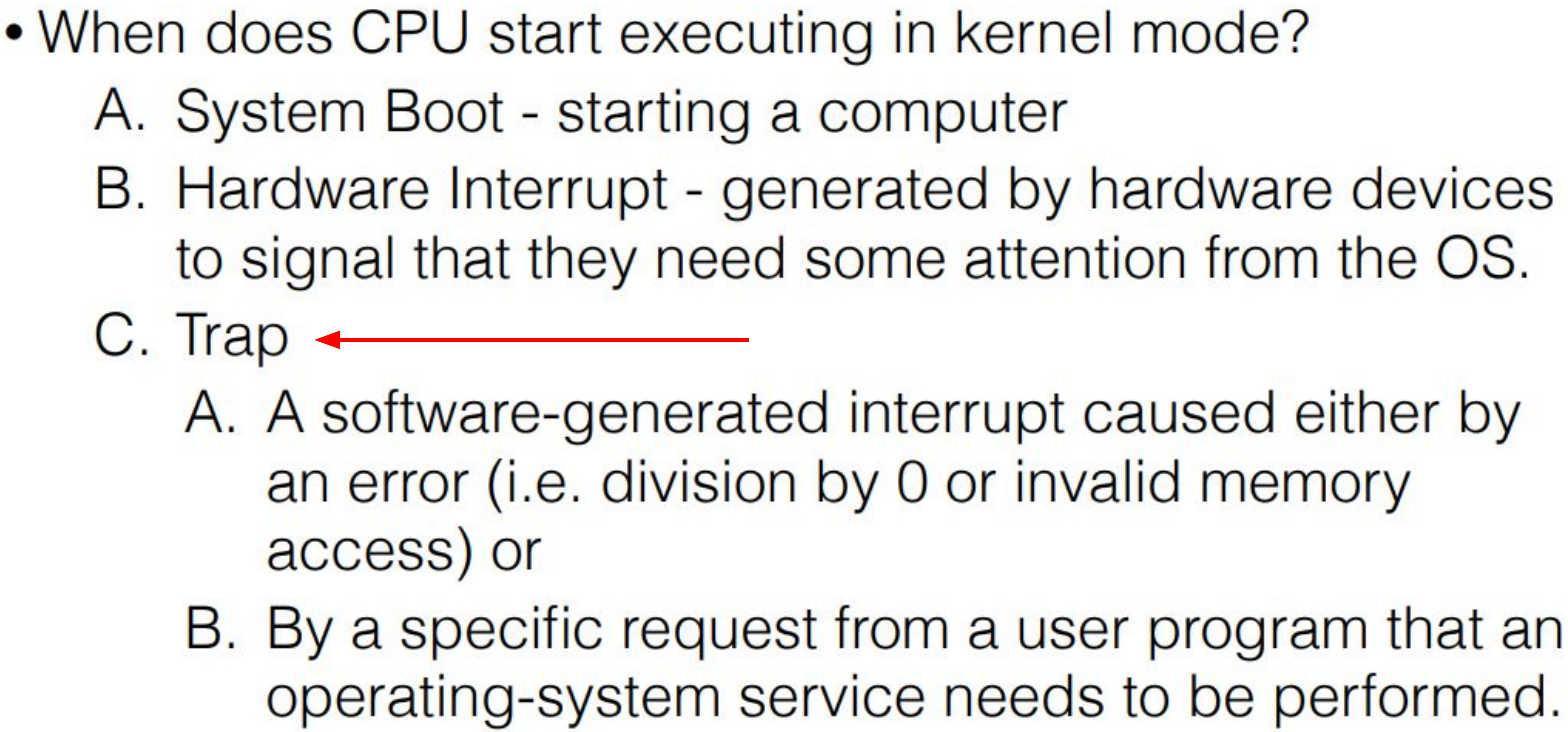
Dual Mode Operation

CPU executes in 2 Modes

- **Kernel mode:** CPU can execute all machine instructions CPU can use every hardware feature
- **User mode:** permits only a subset of the instructions and a subset of the hardware features. Allows OS to protect itself and other system components.



Kernel Mode Execution



- When does CPU start executing in kernel mode?
  - A. System Boot - starting a computer
  - B. Hardware Interrupt - generated by hardware devices to signal that they need some attention from the OS.
  - C. Trap
    - A. A software-generated interrupt caused either by an error (i.e. division by 0 or invalid memory access) or
    - B. By a specific request from a user program that an operating-system service needs to be performed.

Switching Modes

- To obtain services from the operating system,
  - an user program must make a ***system call***, which ***traps*** into the kernel and invokes the operating system.

- The **TRAP** instruction switches from user mode to kernel mode and starts the operating system.

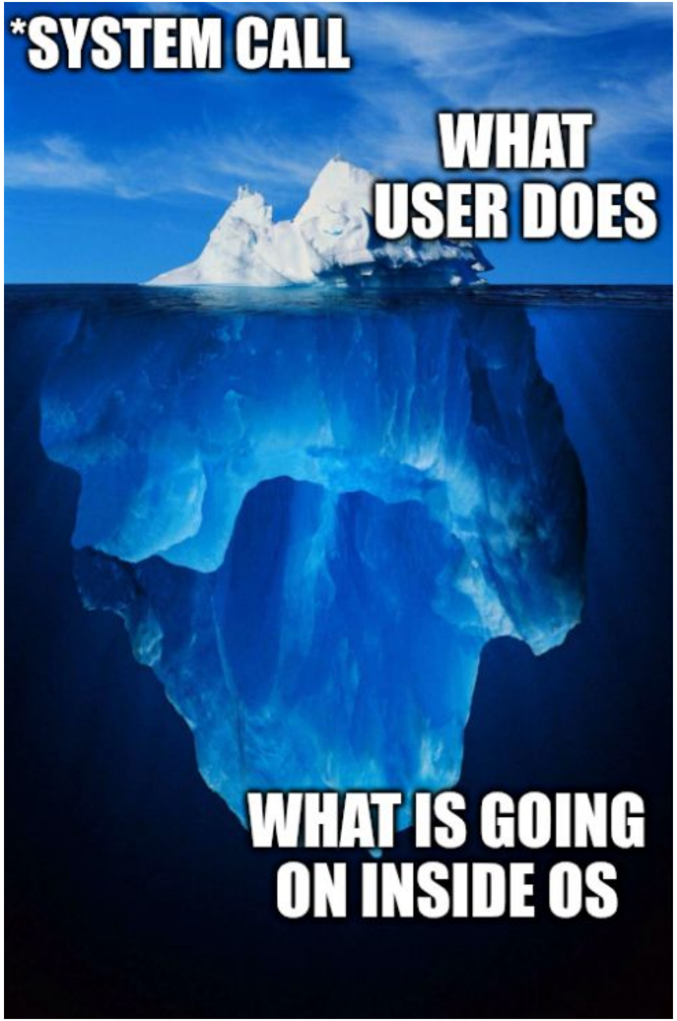
- When the work has been completed, control is returned to the user program at the instruction following the system call.

System Calls

- Programming interface to the services provided by the OS

- Typically used from a high-level language (C or C++) program.

- Typically a number is associated with each system call, and the OS maintains a table indexed according to these numbers.


*SYSTEM CALL
WHAT USER DOES
WHAT IS GOING ON INSIDE OS

Let's see xv6

Prerequisite tools: https://pdos.csail.mit.edu/6.828/2022/tools.html

Press Ctrl+A then X to quit xv6

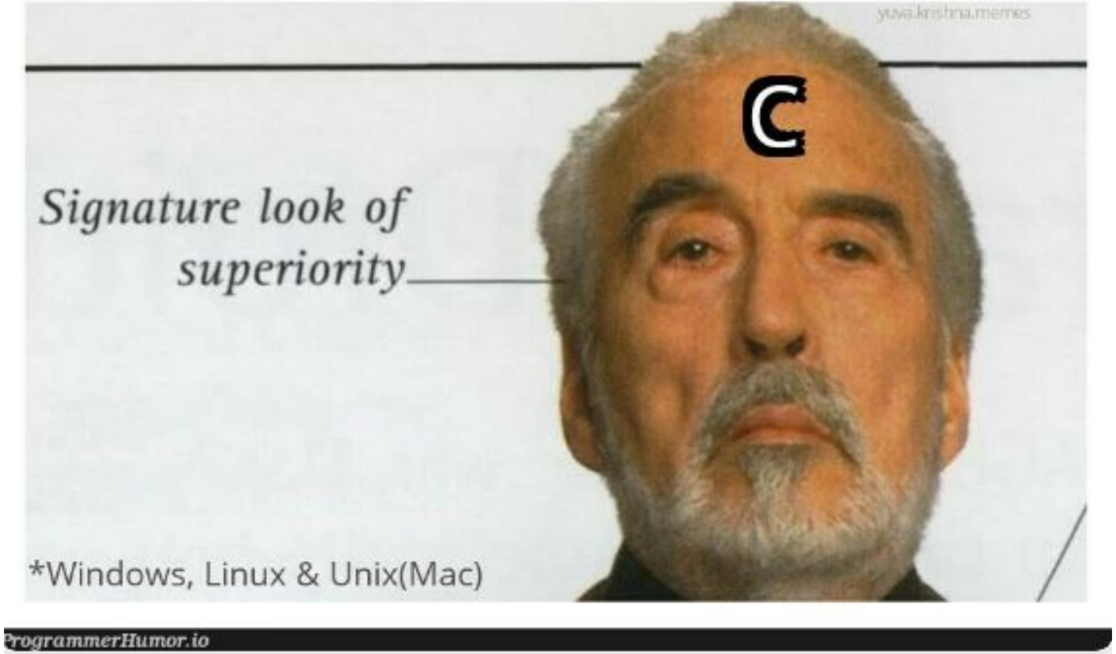
Challenges you will face in this assignment

Too many files

Use

- grep <pattern> *
- grep <pattern> kernel/*
- grep <pattern> user/*

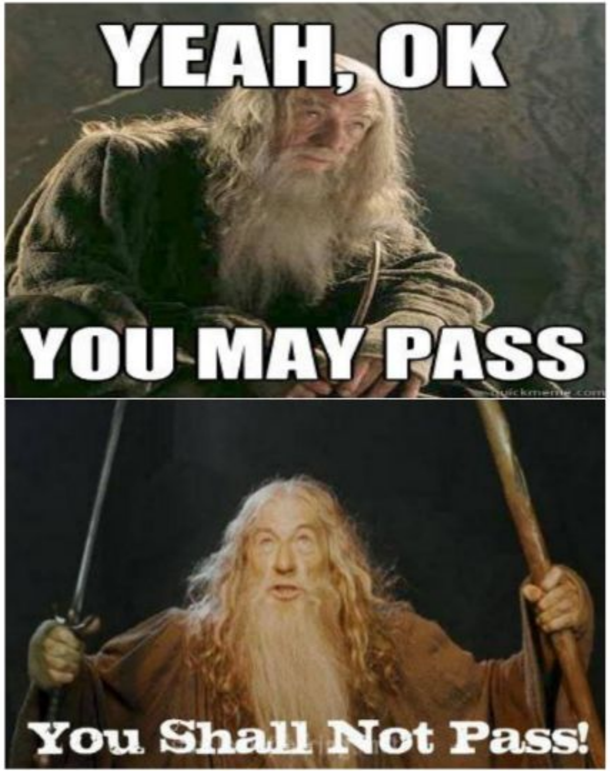
When you are the only language people choose to write a kernel*..

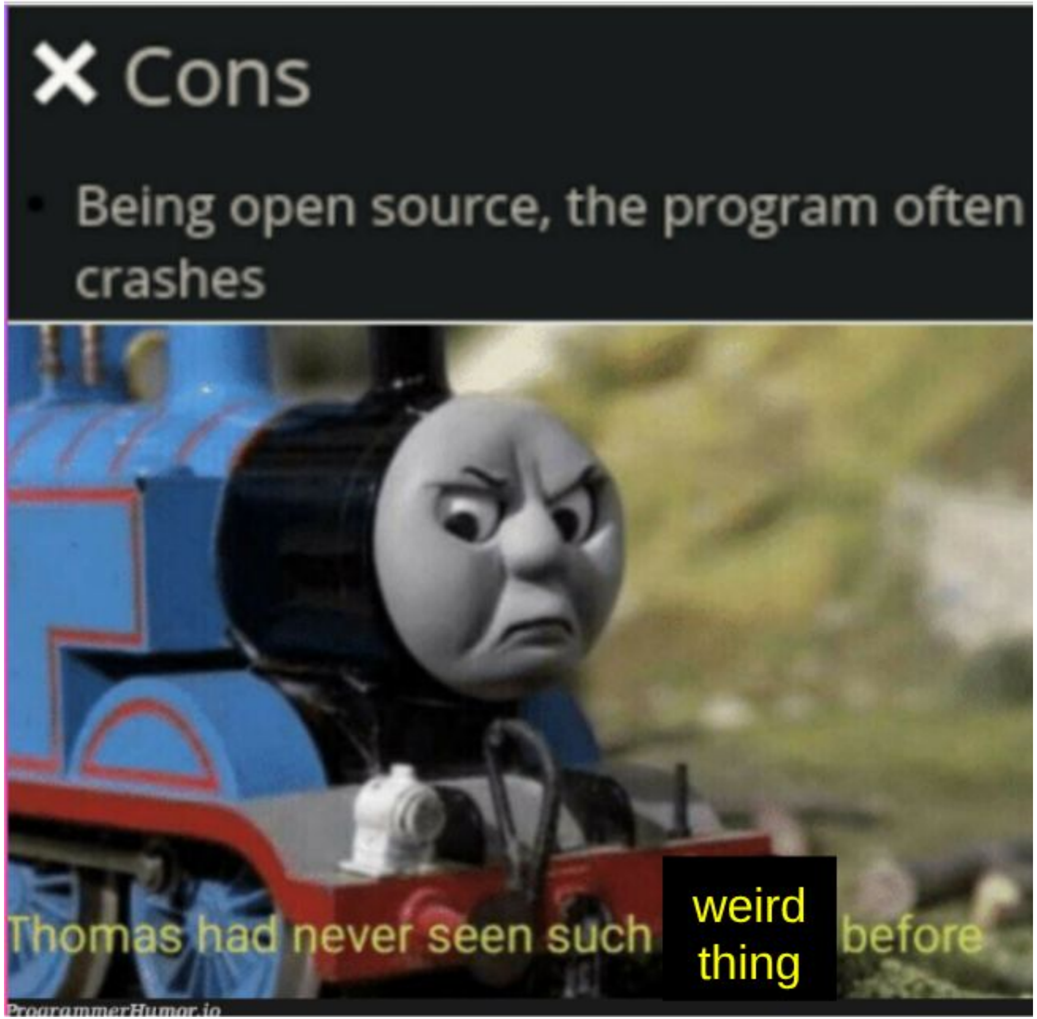


Warnings

me

xv6



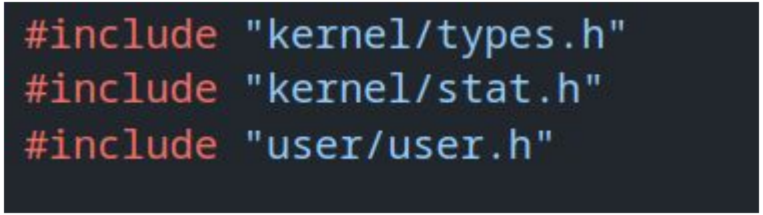


✕ Cons

- Being open source, the program often crashes

Adding a User program

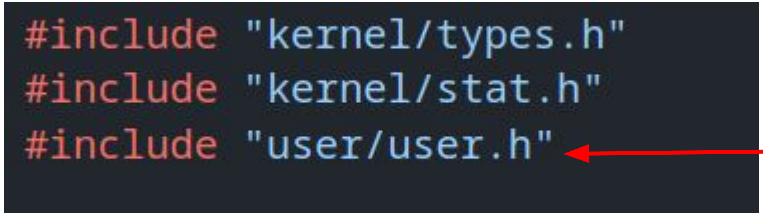
- Create a file under user folder
- These three lines must be added on top strictly following order



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

Adding a User program

- Create a file under user folder
- These three lines must be added on top **strictly following order**

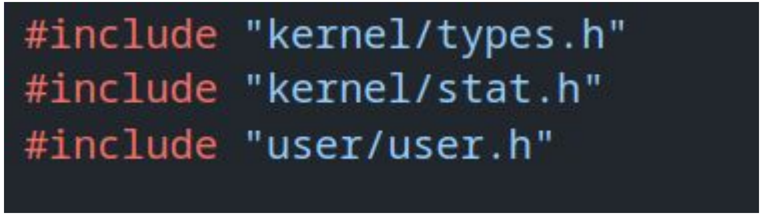


```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
```

User can call these functions **“declared”** here

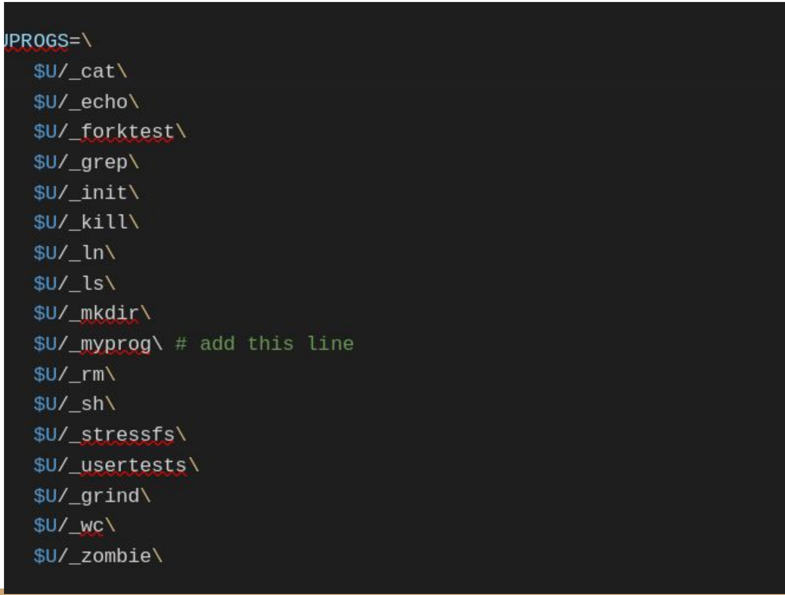
Adding a User program

- Create a file under user folder
- These three lines must be added on top strictly following order



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

- Then add prog_name in Makefile.
- Then run make clean; make qemu



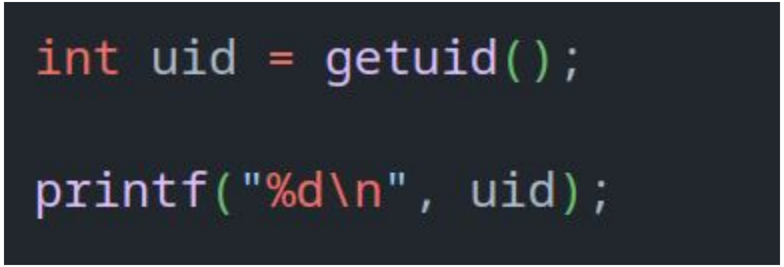
```
UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_myprog\ # add this line
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
```

Adding a system call

First let’s understand what happens during a system call.

Suppose we have added **our own** system call with signature **int getuid();** which simply returns an **integer value** stored in a variable named **uid** in **kernel.**

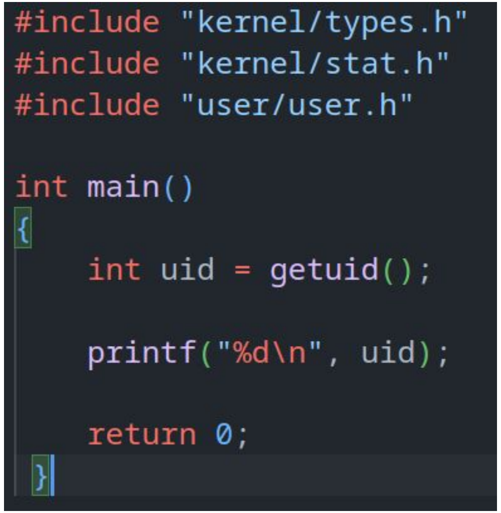
gibuid.c



```
int uid = getuid();

printf("%d\n", uid);
```

A series of things happen

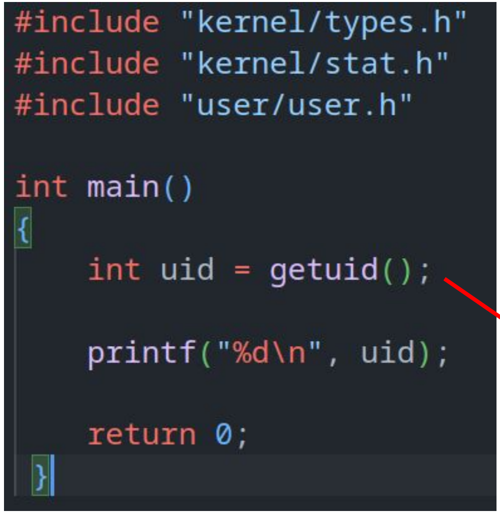
user/gibuid.c



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

int main()
{
    int uid = getuid();

    printf("%d\n", uid);

    return 0;
 }
```

A series of things happen

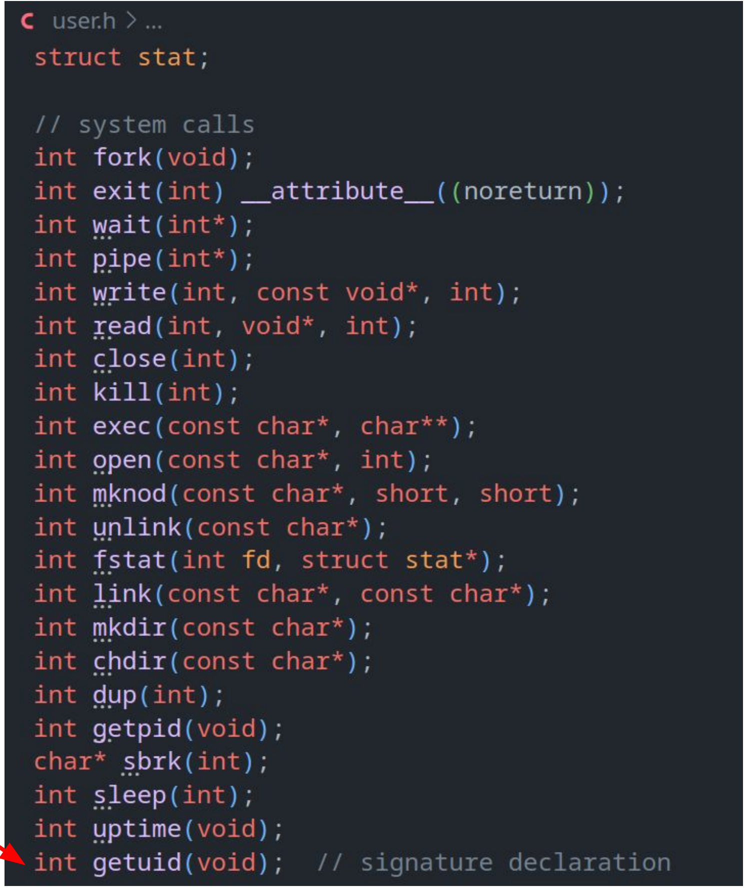
user/gibuid.c



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

int main()
{
    int uid = getuid();

    printf("%d\n", uid);

    return 0;
}
```

Signature is **declared** here

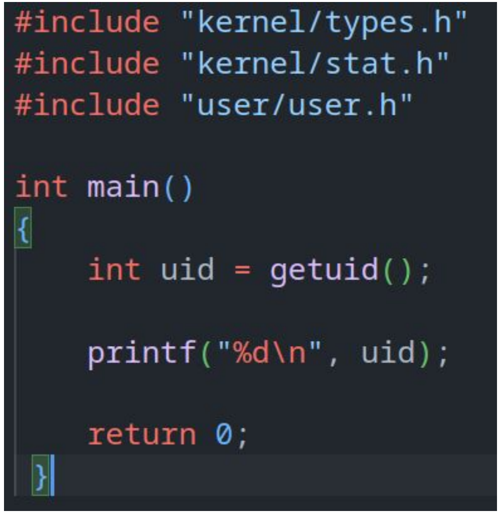
user/user.h



```
struct stat;

// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(const char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int getuid(void);  // signature declaration
```

A series of things happen

user/gibuid.c

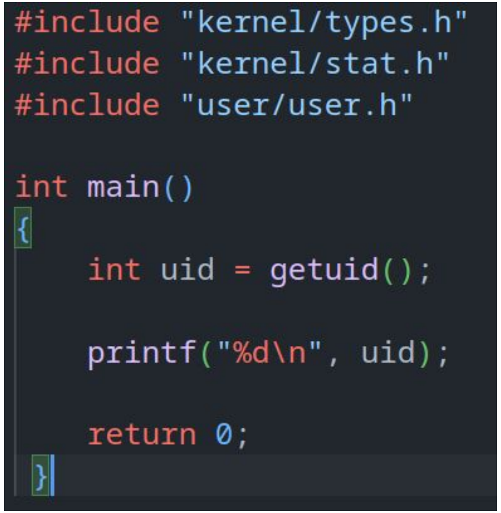


```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main()
{
    int uid = getuid();

    printf("%d\n", uid);

    return 0;
}
```

What about definition?

A series of things happen

user/gibuid.c



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

int main()
{
    int uid = getuid();

    printf("%d\n", uid);

    return 0;
 }
```

What about definition?

Available in user/usys.S (generated from user/usys.pl)

Some other files come into play also. Let’s see…

System Calls [Recap]

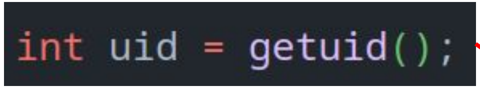
- Programming interface to the services provided by the OS

- Typically used from a high-level language (C or C++) program.

- Typically a number is associated with each system call, and the OS maintains a table indexed according to these numbers.

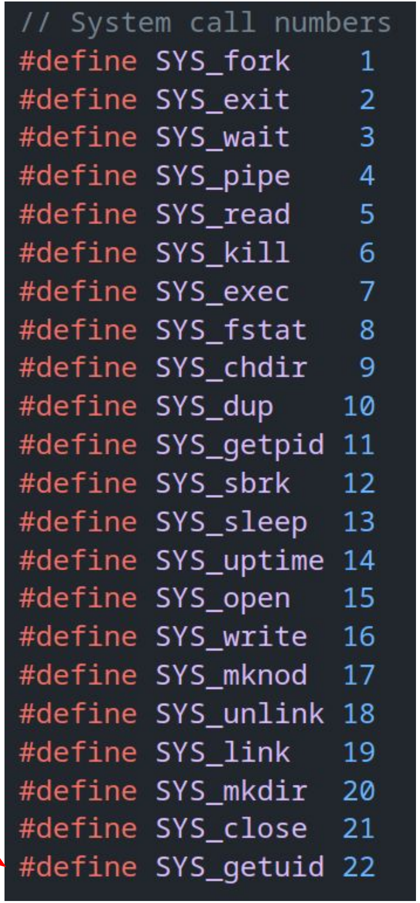
A series of things happen

user/gibuid.c



```c
int uid = getuid();
```

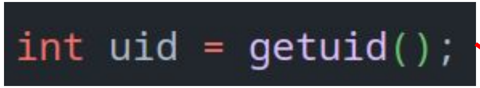
kernel/syscall.h



```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getuid 22
```

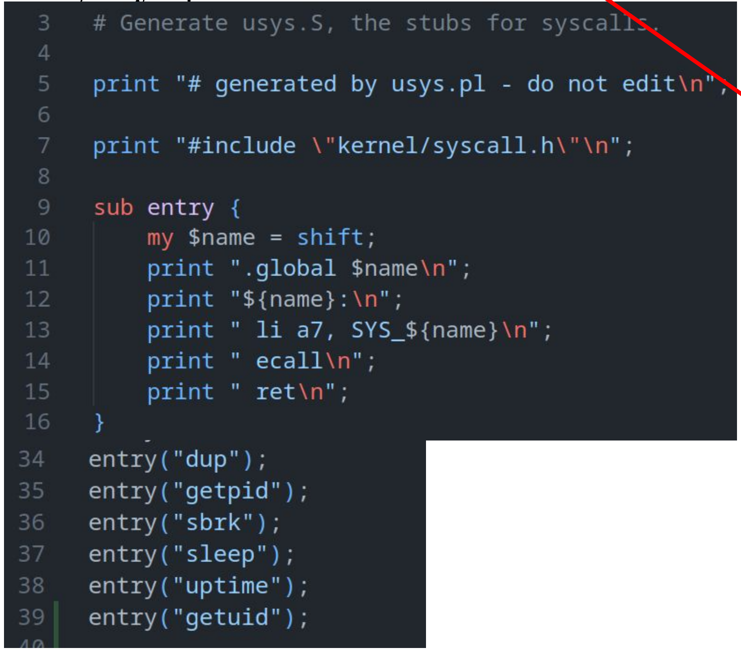
A series of things happen

user/gibuid.c



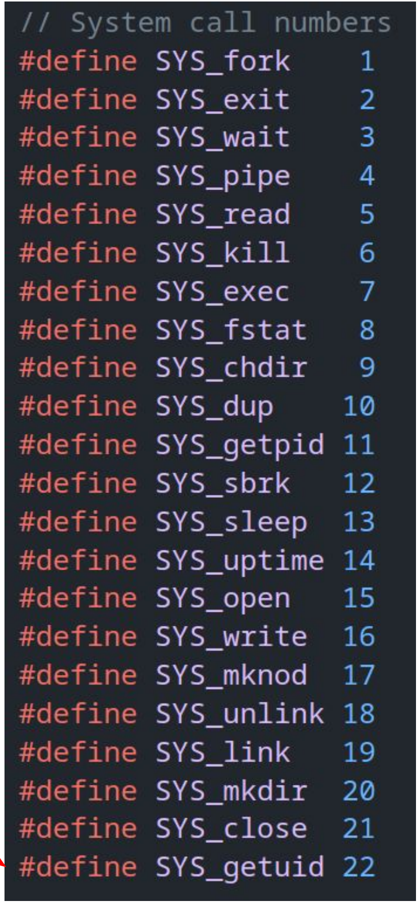
```c
int uid = getuid();
```

user/usys.pl



```perl
# Generate usys.S, the stubs for syscalls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("getuid");
```

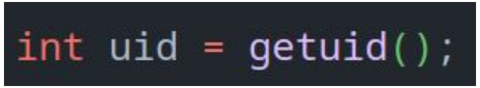
kernel/syscall.h



```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getuid 22
```

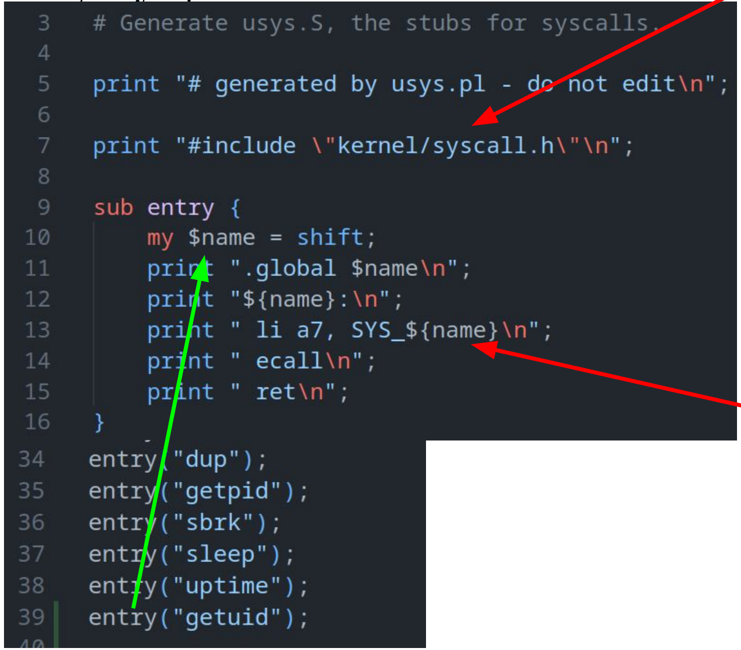
A series of things happen

user/gibuid.c



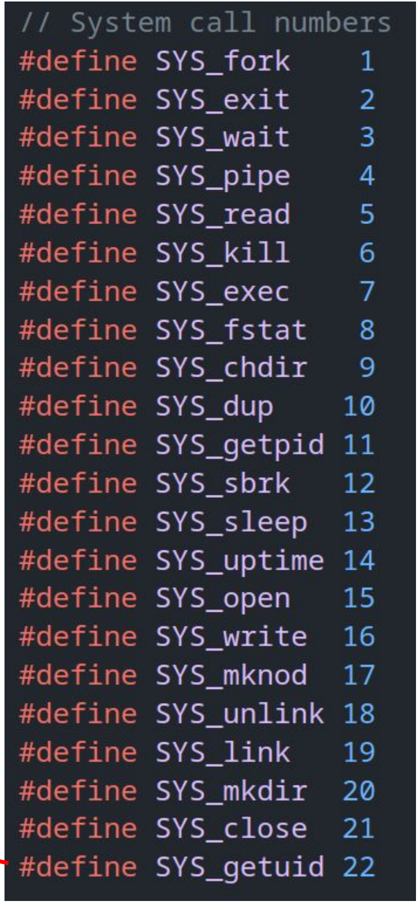
```c
int uid = getuid();
```

user/usys.pl



```perl
# Generate usys.S, the stubs for syscalls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("getuid");
```

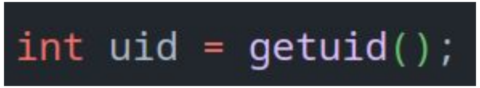
kernel/syscall.h



```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getuid 22
```

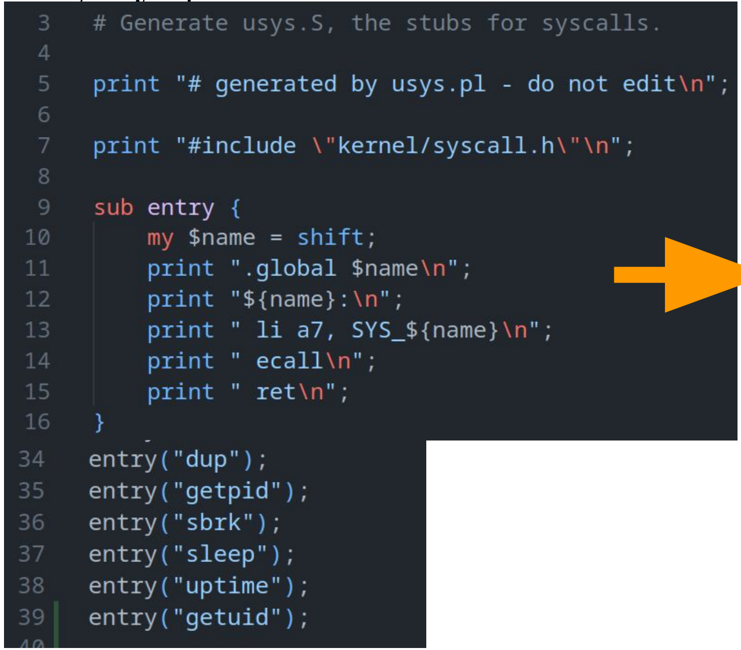
A series of things happen

user/gibuid.c



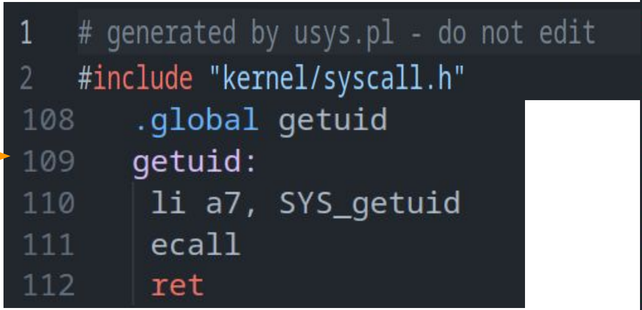
```
int uid = getuid();
```

user/usys.pl



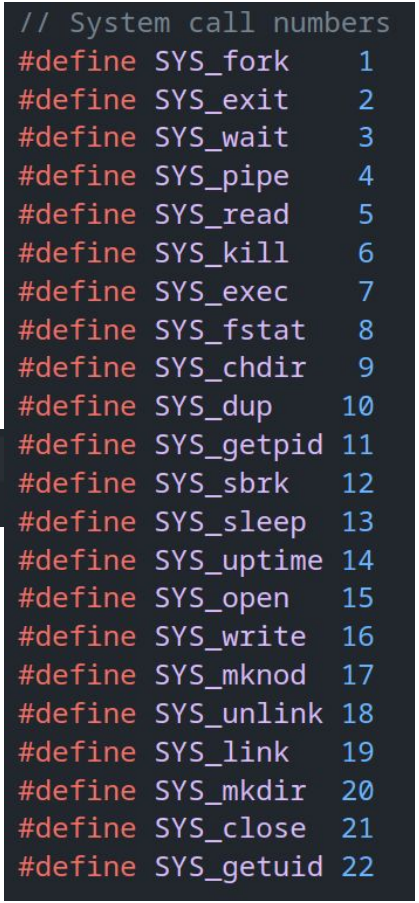
```
# Generate usys.S, the stubs for syscalls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("getuid");
```

user/usys.S



```
# generated by usys.pl - do not edit
#include "kernel/syscall.h"
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

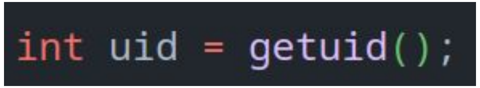
kernel/syscall.h



```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getuid 22
```

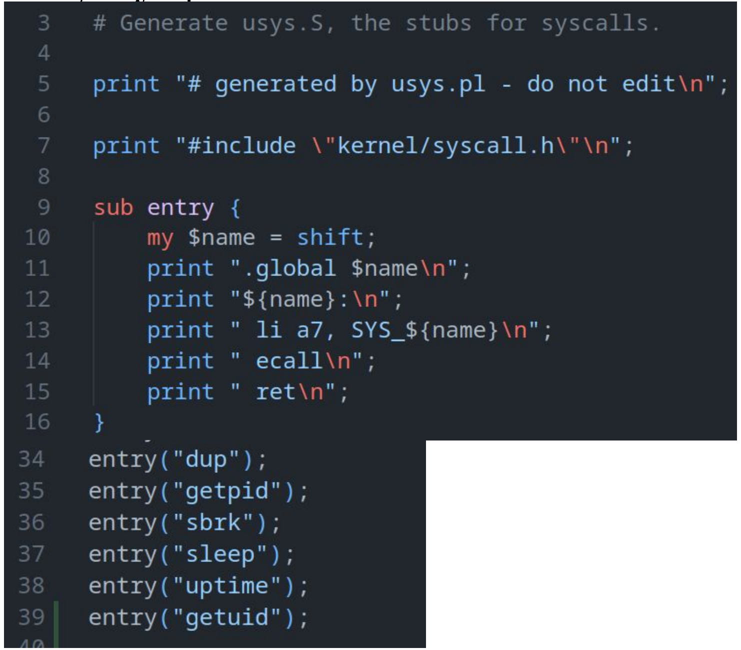
A series of things happen

user/gibuid.c



```
int uid = getuid();
```

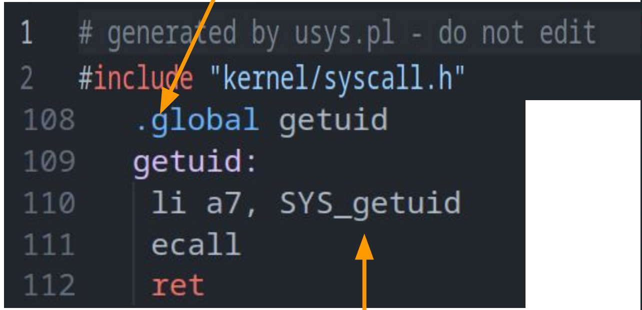
user/usys.pl



```
# Generate usys.S, the stubs for syscalls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("getuid");
```

Definition we are looking for..

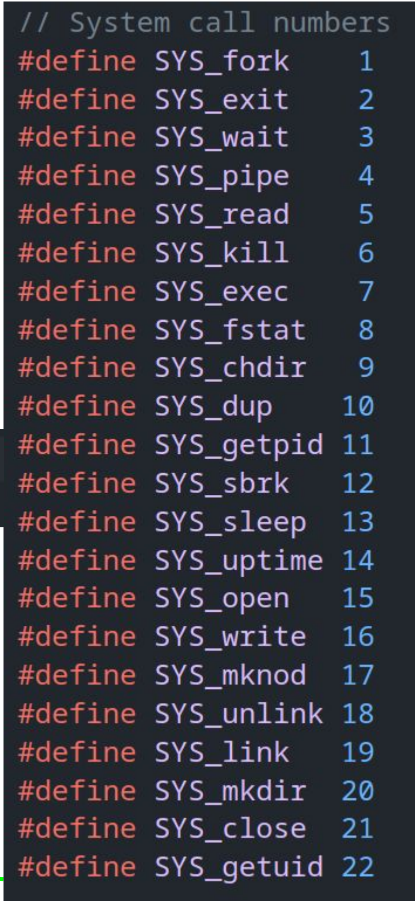
user/usys.S



```
# generated by usys.pl - do not edit
#include "kernel/syscall.h"
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

Loads 22 at a7 register

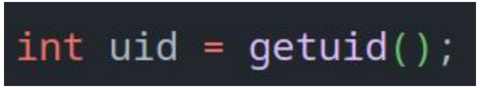
kernel/syscall.h



```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getuid 22
```

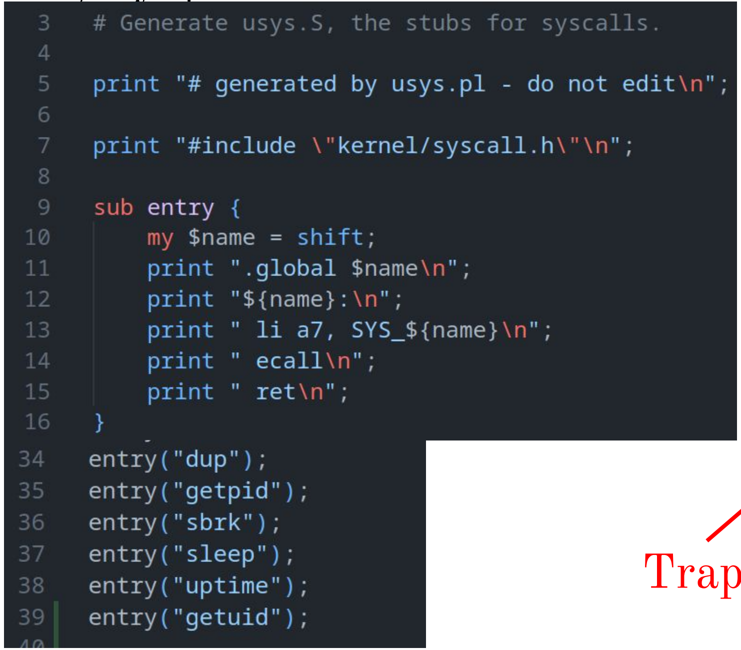
A series of things happen

user/gibuid.c



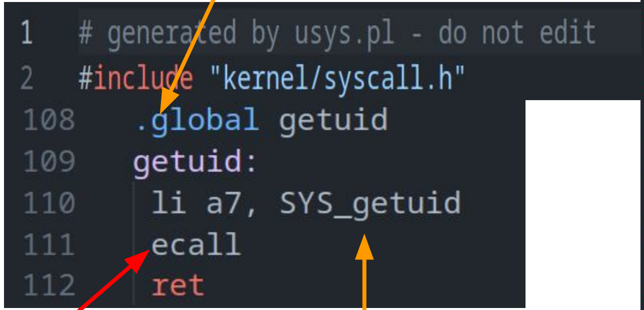
```c
int uid = getuid();
```

user/usys.pl



```perl
# Generate usys.S, the stubs for syscalls.

print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("getuid");
```

user/usys.S



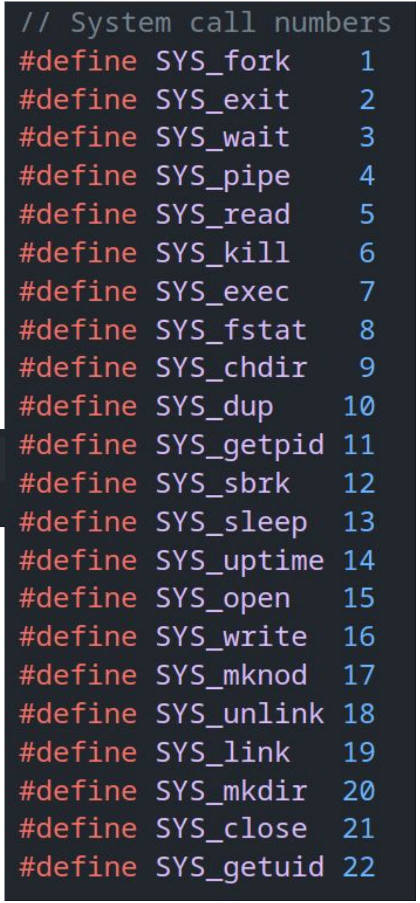
```asm
# generated by usys.pl - do not edit
#include "kernel/syscall.h"
  .global getuid
  getuid:
   li a7, SYS_getuid
   ecall
   ret
```

Definition we are looking for..

Loads 22 at a7 register

Traps

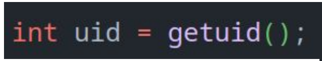
kernel/syscall.h



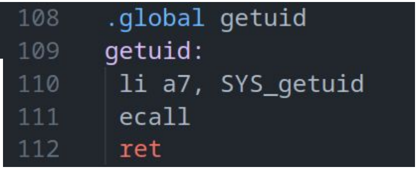
```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getuid 22
```

A series of things happen

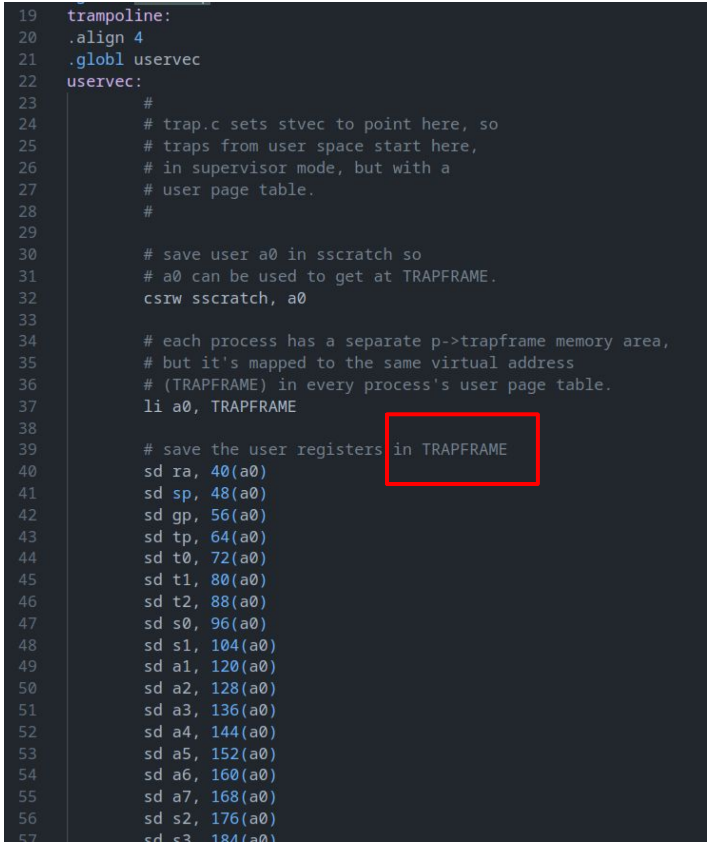
user/gibuid.c -> user/Usys.s -> kernel/trampoline.S/uservec()



```c
int uid = getuid();
```



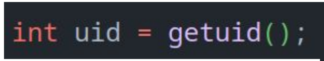
```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```



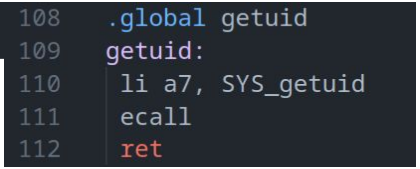
```asm
trampoline:
.align 4
.globl uservec
uservec:
        #
        # trap.c sets stvec to point here, so
        # traps from user space start here,
        # in supervisor mode, but with a
        # user page table.
        #

        # save user a0 in sscratch so
        # a0 can be used to get at TRAPFRAME.
        csrw sscratch, a0

        # each process has a separate p->trapframe memory area,
        # but it's mapped to the same virtual address
        # (TRAPFRAME) in every process's user page table.
        li a0, TRAPFRAME

        # save the user registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
        sd tp, 64(a0)
        sd t0, 72(a0)
        sd t1, 80(a0)
        sd t2, 88(a0)
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
```

A series of things happen

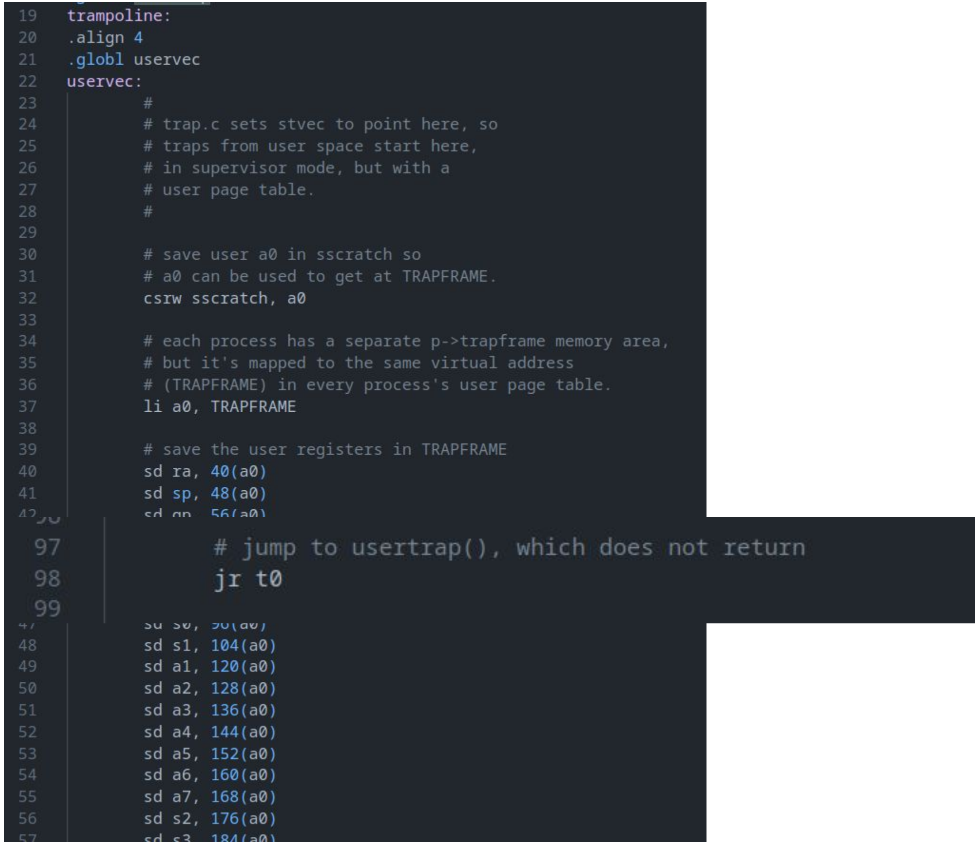
user/gibuid.c -> user/Usys.s -> kernel/trampoline.S/uservec()



```
int uid = getuid();
```



```
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```



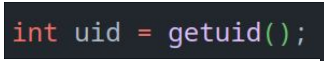
```
trampoline:
.align 4
.globl uservec
uservec:
        #
        # trap.c sets stvec to point here, so
        # traps from user space start here,
        # in supervisor mode, but with a
        # user page table.
        #

        # save user a0 in sscratch so
        # a0 can be used to get at TRAPFRAME.
        csrw sscratch, a0

        # each process has a separate p->trapframe memory area,
        # but it's mapped to the same virtual address
        # (TRAPFRAME) in every process's user page table.
        li a0, TRAPFRAME

        # save the user registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
```

```
        # jump to usertrap(), which does not return
        jr t0
```

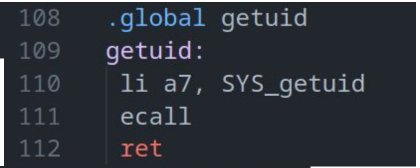
```
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
```

A series of things happen

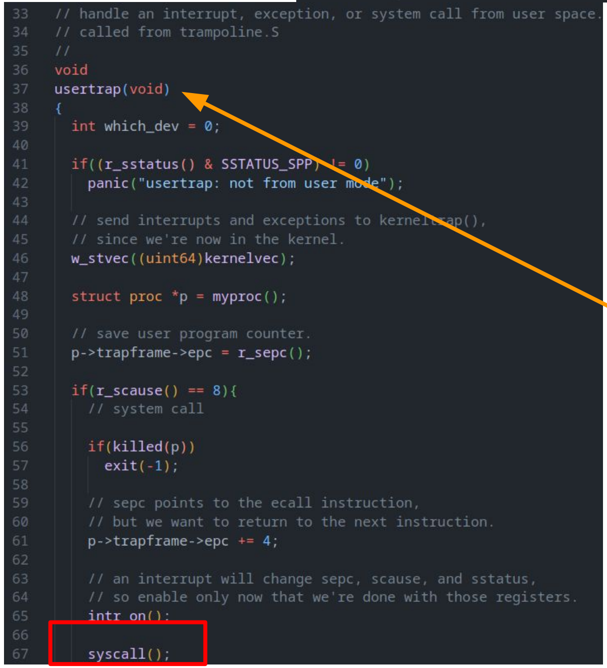
user/gibuid.c -> user/Usys.s -> kernel/trampoline.S/uservec() -> kernel/trap.c/usertrap()



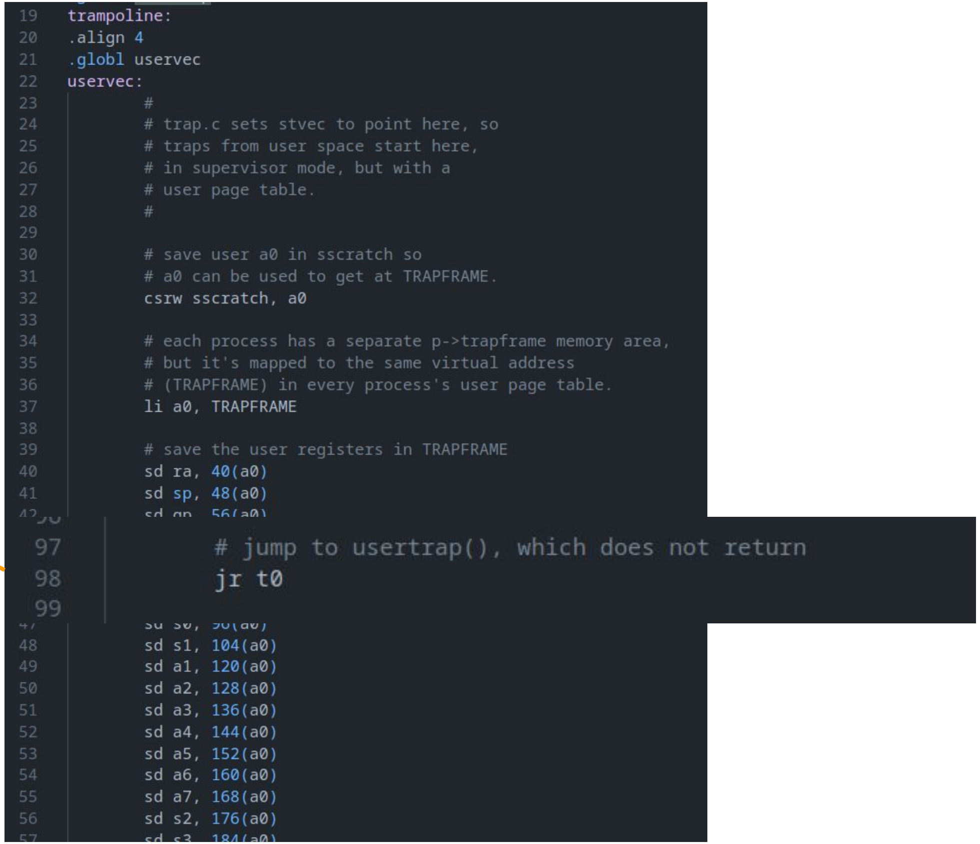
```
int uid = getuid();
```



```
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```



```
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
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();

    syscall();
```



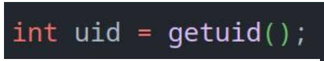
```
trampoline:
.align 4
.globl uservec
uservec:
        #
        # trap.c sets stvec to point here, so
        # traps from user space start here,
        # in supervisor mode, but with a
        # user page table.
        #

        # save user a0 in sscratch so
        # a0 can be used to get at TRAPFRAME.
        csrw sscratch, a0

        # each process has a separate p->trapframe memory area,
        # but it's mapped to the same virtual address
        # (TRAPFRAME) in every process's user page table.
        li a0, TRAPFRAME

        # save the user registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
```

```
        # jump to usertrap(), which does not return
        jr t0

```

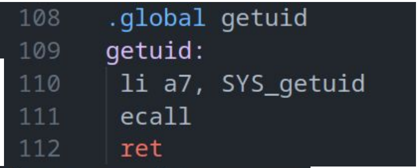
```
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
```

A series of things happen

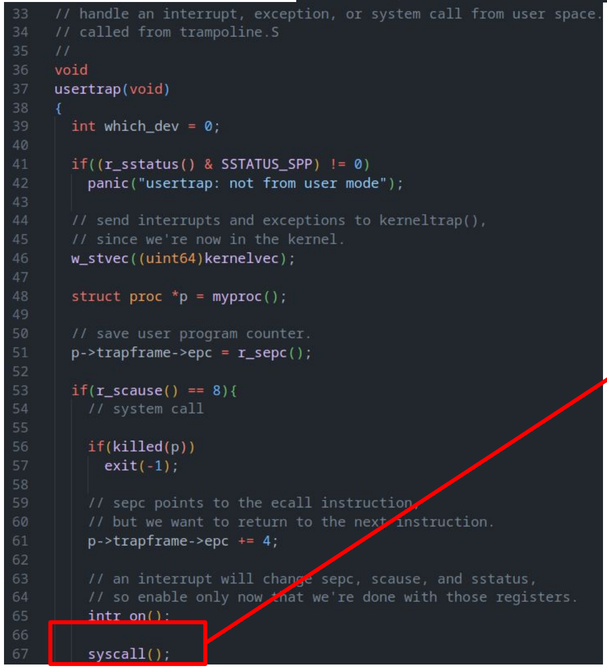
user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()



```c
int uid = getuid();
```

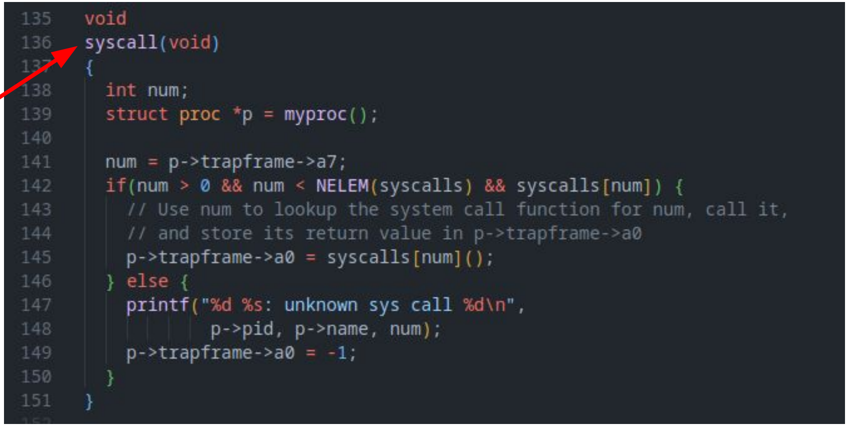


```
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```



```c
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
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();

    syscall();
```

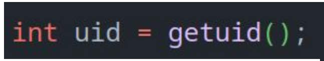
kernel/syscall.c



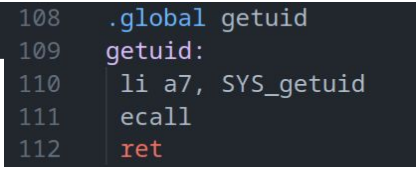
```c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()

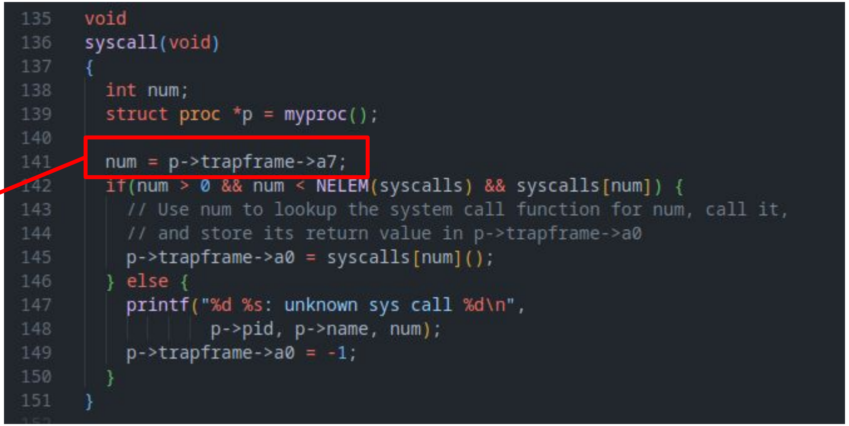


```c
int uid = getuid();
```



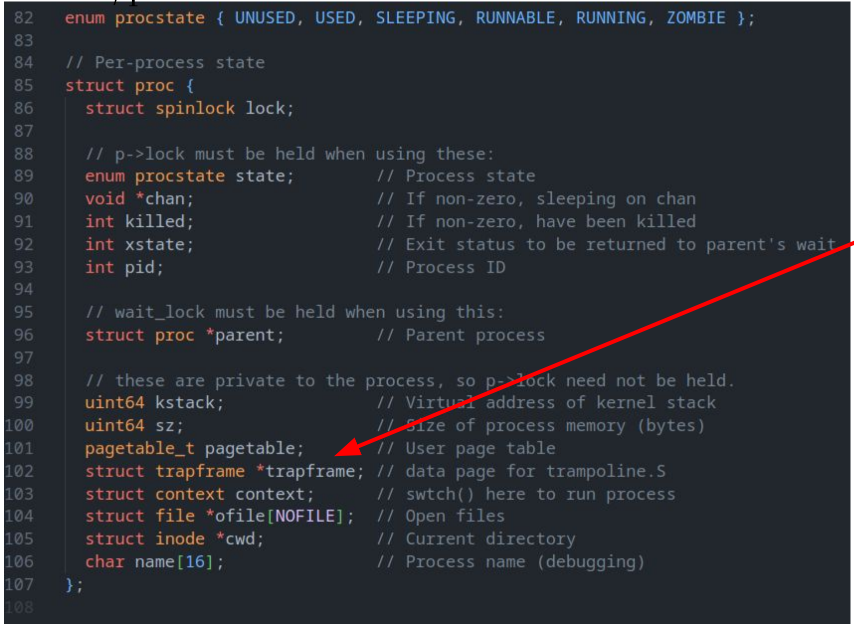
```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

kernel/syscall.c



```c
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

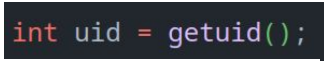
kernel/proc.h



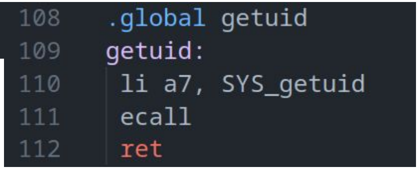
```c
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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

  // 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)
};
```

A series of things happen

user/gibuid.c->user/Usys.s ->**kernel/trampoline.S/uservec()**->kernel/t

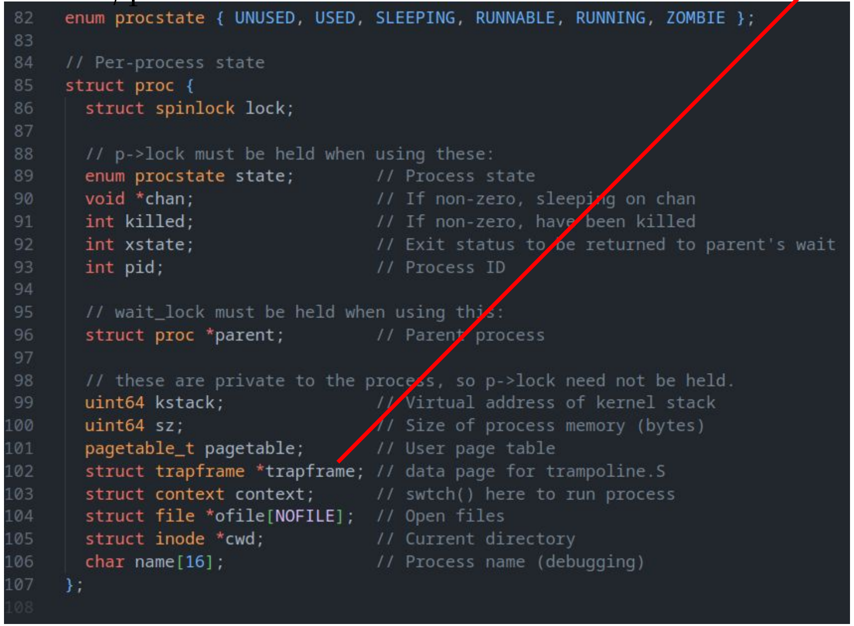


```
int uid = getuid();
```

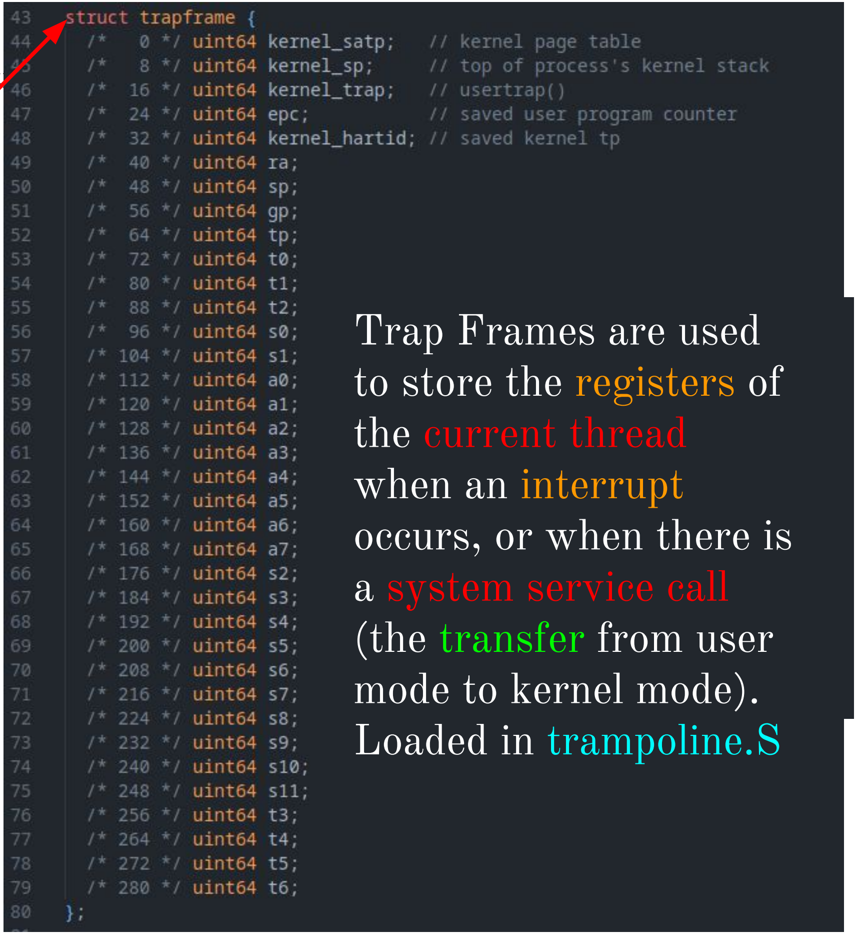


```
.global getuid
getuid:
  li a7, SYS_getuid
  ecall
  ret
```

kernel/proc.h



```
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

  // 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)
};
```

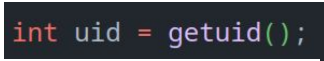


```
struct trapframe {
  /*   0 */ uint64 kernel_satp;   // kernel page table
  /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
  /*  16 */ uint64 kernel_trap;   // usertrap()
  /*  24 */ uint64 epc;           // saved user program counter
  /*  32 */ uint64 kernel_hartid; // saved kernel tp
  /*  40 */ uint64 ra;
  /*  48 */ uint64 sp;
  /*  56 */ uint64 gp;
  /*  64 */ uint64 tp;
  /*  72 */ uint64 t0;
  /*  80 */ uint64 t1;
  /*  88 */ uint64 t2;
  /*  96 */ uint64 s0;
  /* 104 */ uint64 s1;
  /* 112 */ uint64 a0;
  /* 120 */ uint64 a1;
  /* 128 */ uint64 a2;
  /* 136 */ uint64 a3;
  /* 144 */ uint64 a4;
  /* 152 */ uint64 a5;
  /* 160 */ uint64 a6;
  /* 168 */ uint64 a7;
  /* 176 */ uint64 s2;
  /* 184 */ uint64 s3;
  /* 192 */ uint64 s4;
  /* 200 */ uint64 s5;
  /* 208 */ uint64 s6;
  /* 216 */ uint64 s7;
  /* 224 */ uint64 s8;
  /* 232 */ uint64 s9;
  /* 240 */ uint64 s10;
  /* 248 */ uint64 s11;
  /* 256 */ uint64 t3;
  /* 264 */ uint64 t4;
  /* 272 */ uint64 t5;
  /* 280 */ uint64 t6;
};
```

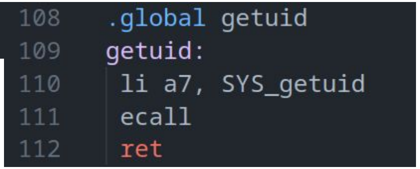
Trap Frames are used to store the registers of the current thread when an interrupt occurs, or when there is a system service call (the transfer from user mode to kernel mode). Loaded in trampoline.S

A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()

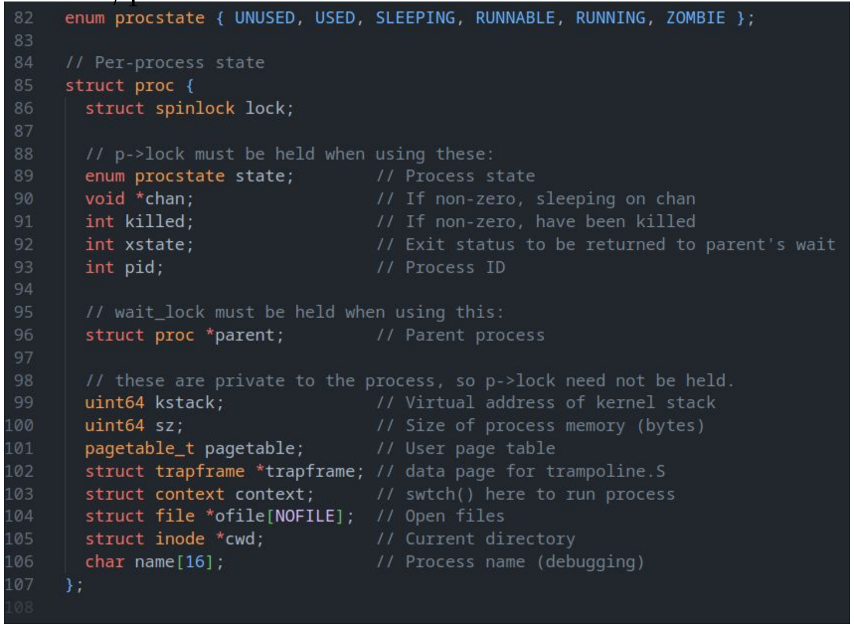


```c
int uid = getuid();
```



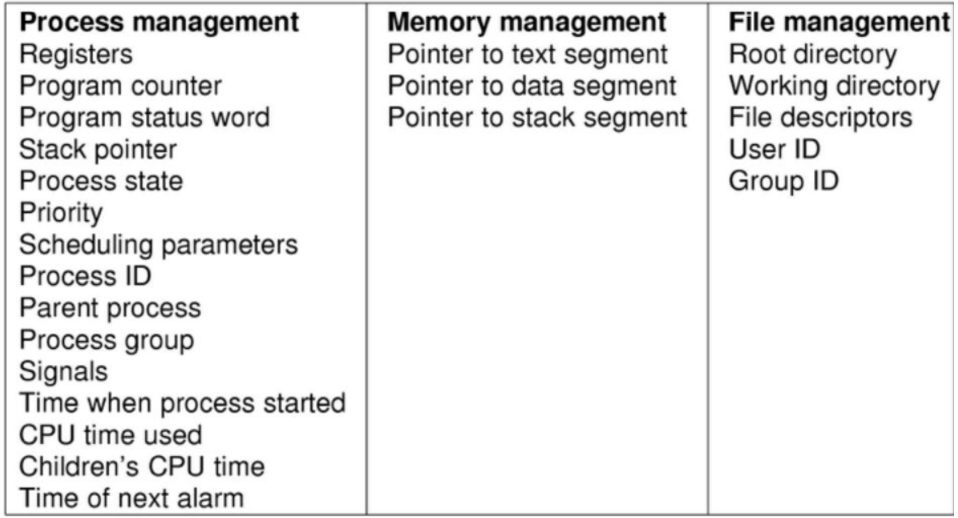
```
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

kernel/proc.h



```c
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

  // 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)
};
```

Process Control Block (PCB)

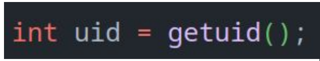


Process management	**Memory management**	**File management**
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

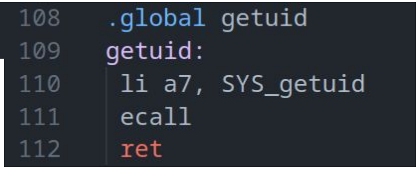
Figure: Fields of a PCB

A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()

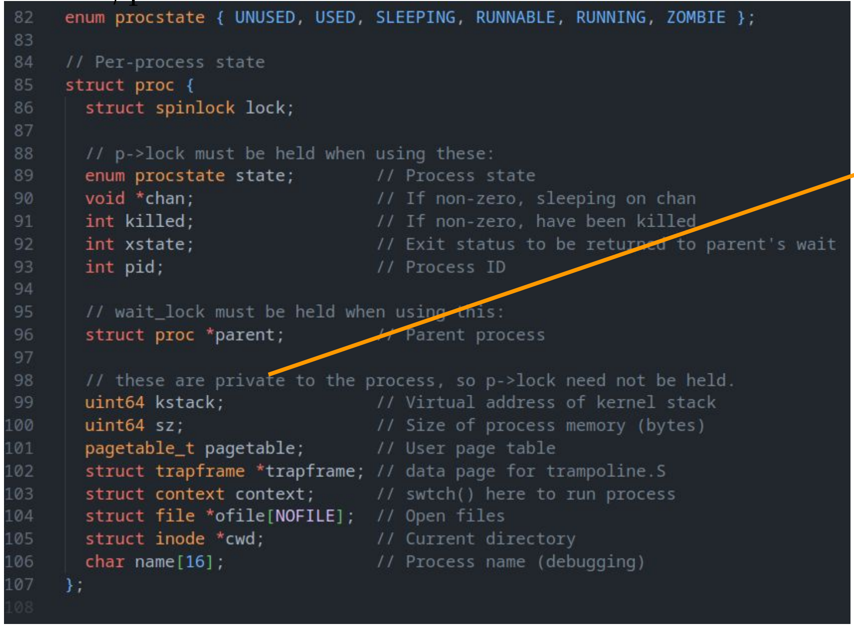


```c
int uid = getuid();
```



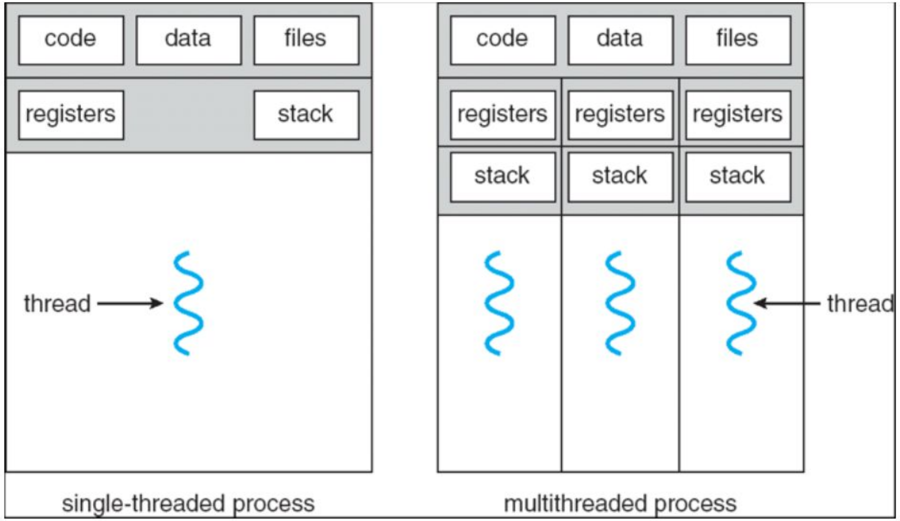
```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

kernel/proc.h



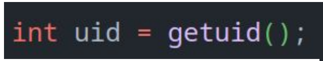
```c
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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

  // 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)
};
```

Multithreaded Processes

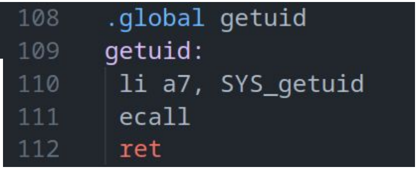


A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()

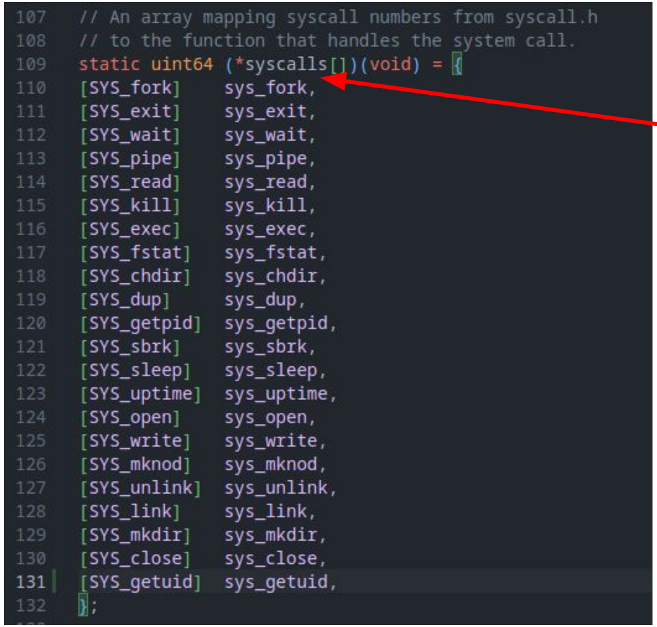


```c
int uid = getuid();
```



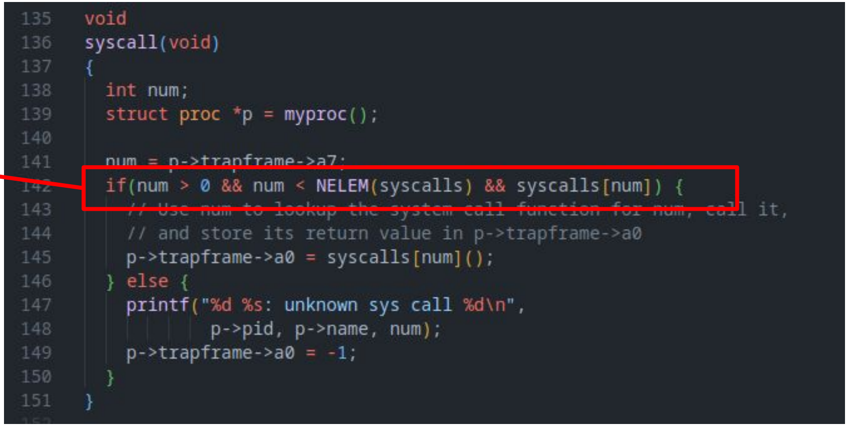
```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

kernel/syscall.c



```c
// 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_getuid]  sys_getuid,
};
```

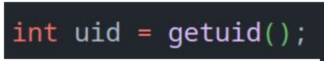
kernel/syscall.c



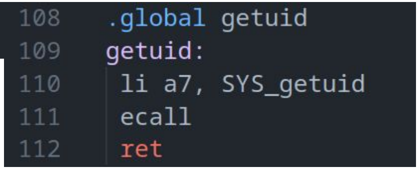
```c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()

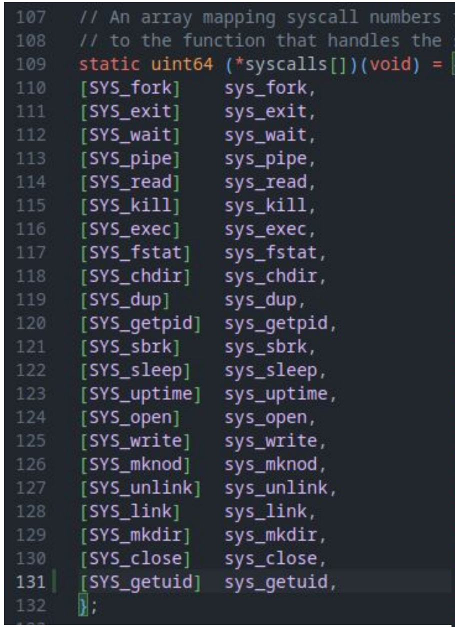


```
int uid = getuid();
```



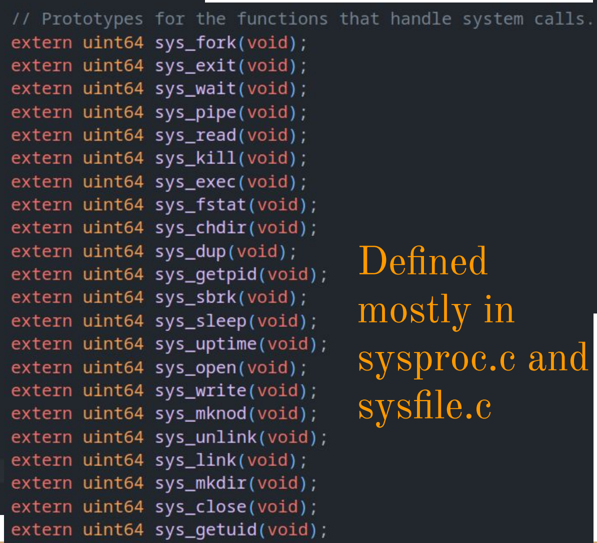
```
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

kernel/syscall.c



```
// 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_getuid]  sys_getuid,
};
```

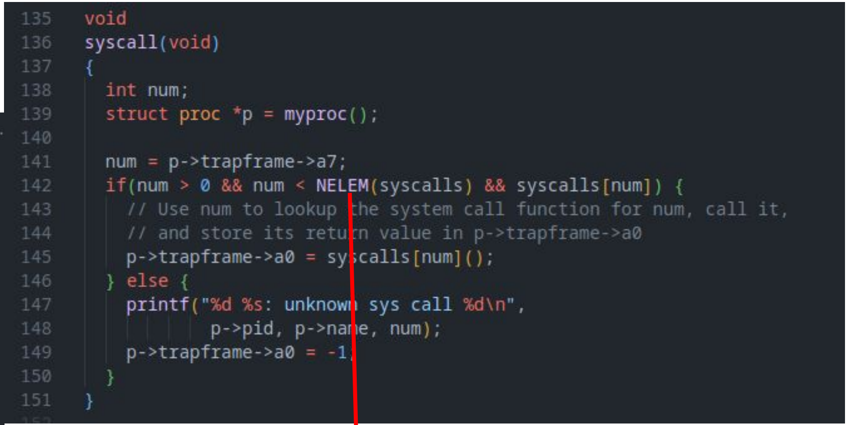
kernel/syscall.c



```
// Prototypes for the functions that handle system calls.
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
extern uint64 sys_kill(void);
extern uint64 sys_exec(void);
extern uint64 sys_fstat(void);
extern uint64 sys_chdir(void);
extern uint64 sys_dup(void);
extern uint64 sys_getpid(void);
extern uint64 sys_sbrk(void);
extern uint64 sys_sleep(void);
extern uint64 sys_uptime(void);
extern uint64 sys_open(void);
extern uint64 sys_write(void);
extern uint64 sys_mknod(void);
extern uint64 sys_unlink(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_getuid(void);
```

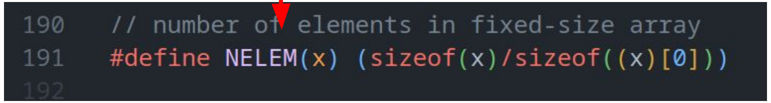
Defined mostly in sysproc.c and sysfile.c

kernel/syscall.c



```
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

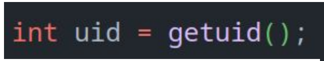
kernel/defs.h



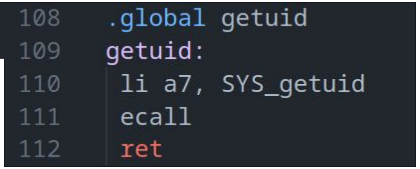
```
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()



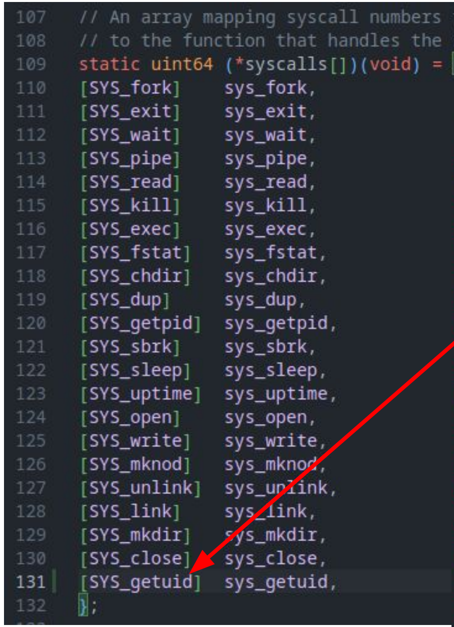
```c
int uid = getuid();
```



```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

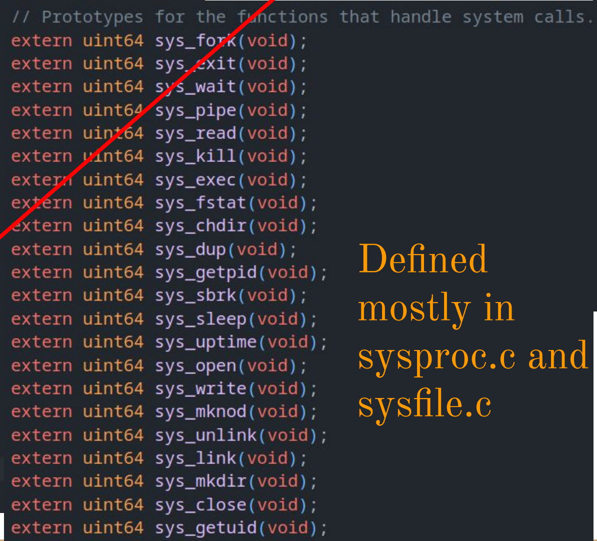
These were defined in syscall.h (loaded in a7). So, line 145 will call sys_getuid() [defined in sysproc.c]

kernel/syscall.c



```c
// 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_getuid]  sys_getuid,
};
```

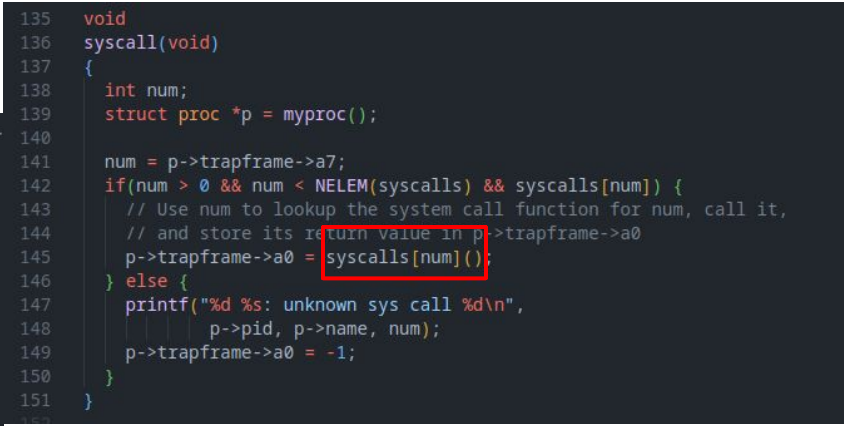
kernel/syscall.c



```c
// Prototypes for the functions that handle system calls.
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
extern uint64 sys_kill(void);
extern uint64 sys_exec(void);
extern uint64 sys_fstat(void);
extern uint64 sys_chdir(void);
extern uint64 sys_dup(void);
extern uint64 sys_getpid(void);
extern uint64 sys_sbrk(void);
extern uint64 sys_sleep(void);
extern uint64 sys_uptime(void);
extern uint64 sys_open(void);
extern uint64 sys_write(void);
extern uint64 sys_mknod(void);
extern uint64 sys_unlink(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_getuid(void);
```

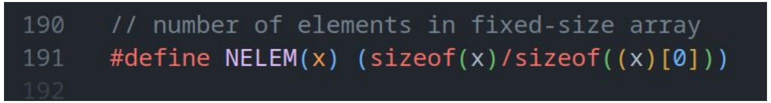
Defined mostly in sysproc.c and sysfile.c

kernel/syscall.c



```c
135  void
136  syscall(void)
137  {
138    int num;
139    struct proc *p = myproc();
140
141    num = p->trapframe->a7;
142    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
143      // Use num to lookup the system call function for num, call it,
144      // and store its return value in p->trapframe->a0
145      p->trapframe->a0 = syscalls[num]();
146    } else {
147      printf("%d %s: unknown sys call %d\n",
148              p->pid, p->name, num);
149      p->trapframe->a0 = -1;
150    }
151  }
```

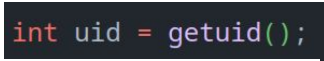
kernel/defs.h



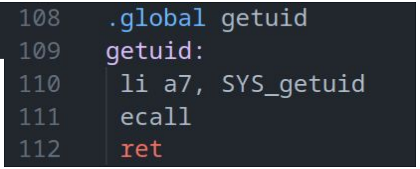
```c
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()->kernel/sysproc.c



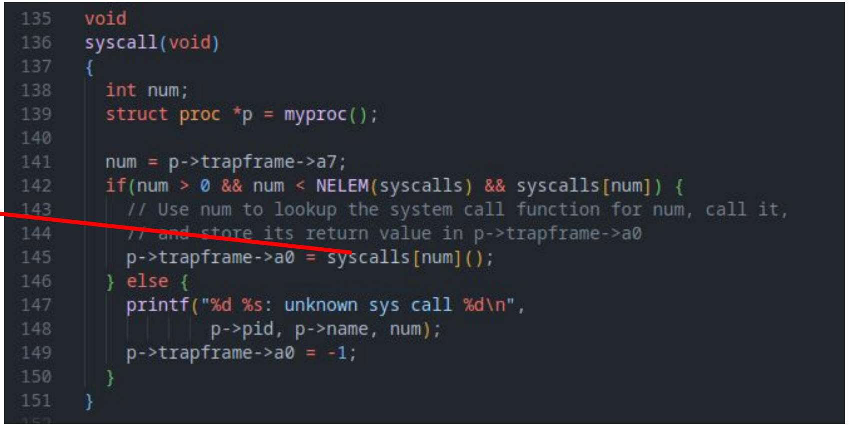
```c
int uid = getuid();
```



```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

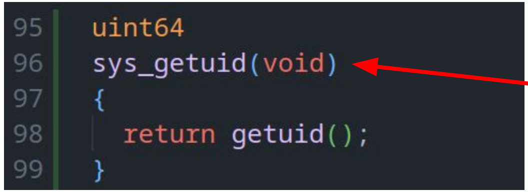
These were defined in syscall.h (loaded in a7). So, line 145 will call sys_getuid() [defined in sysproc.c]

kernel/syscall.c



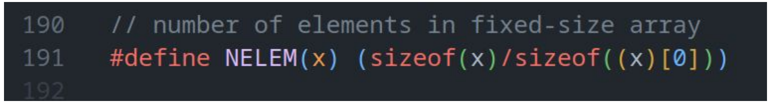
```c
135  void
136  syscall(void)
137  {
138    int num;
139    struct proc *p = myproc();
140
141    num = p->trapframe->a7;
142    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
143      // Use num to lookup the system call function for num, call it,
144      // and store its return value in p->trapframe->a0
145      p->trapframe->a0 = syscalls[num]();
146    } else {
147      printf("%d %s: unknown sys call %d\n",
148              p->pid, p->name, num);
149      p->trapframe->a0 = -1;
150    }
151  }
```

kernel/sysproc.c



```c
uint64
sys_getuid(void)
{
  return getuid();
}
```

kernel/defs.h

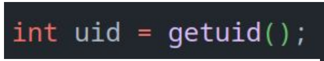


```c
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

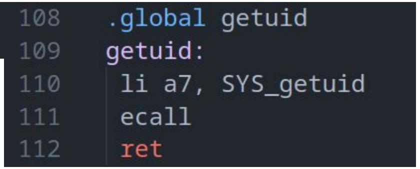
A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()->kernel/sysproc.c

->kernel/proc.c

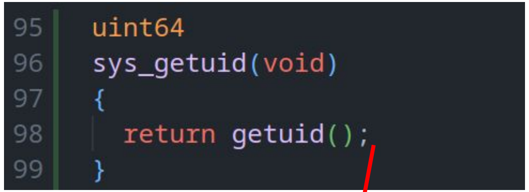


```c
int uid = getuid();
```



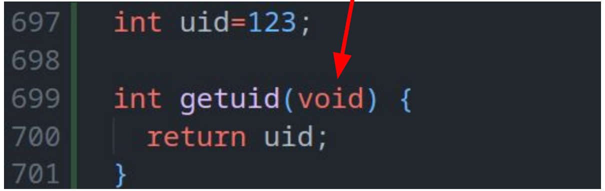
```asm
.global getuid
getuid:
  li a7, SYS_getuid
  ecall
  ret
```

kernel/sysproc.c



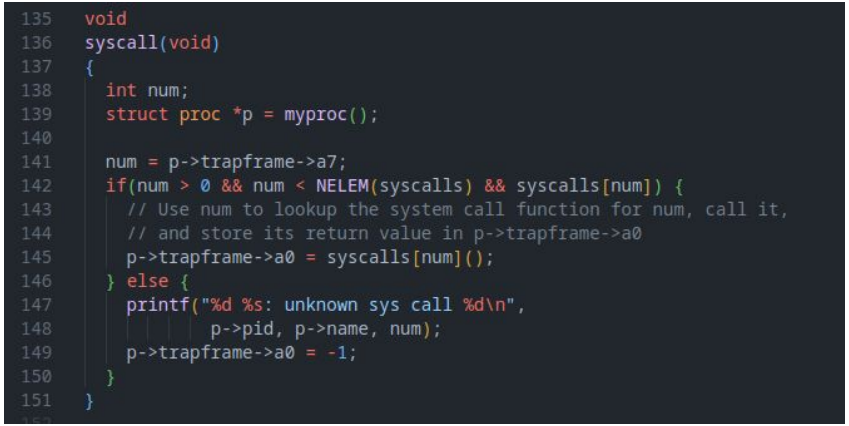
```c
uint64
sys_getuid(void)
{
  return getuid();
}
```

kernel/proc.c



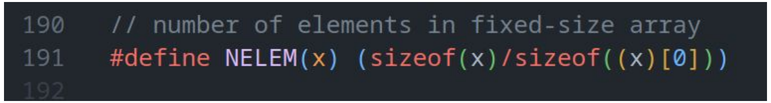
```c
int uid=123;

int getuid(void) {
  return uid;
}
```

kernel/syscall.c



```c
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

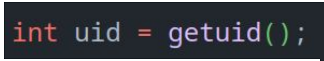
kernel/defs.h



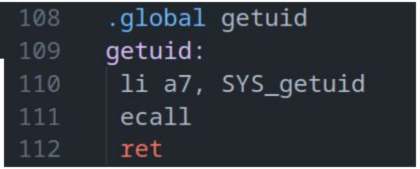
```c
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()->kernel/sysproc.c ->kernel/proc.c

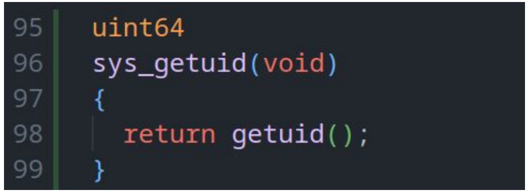


```
int uid = getuid();
```



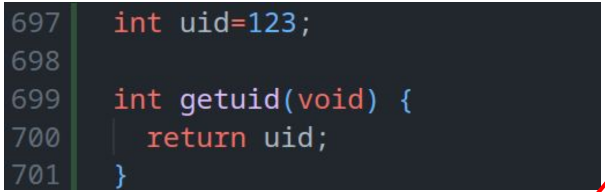
```
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

kernel/sysproc.c



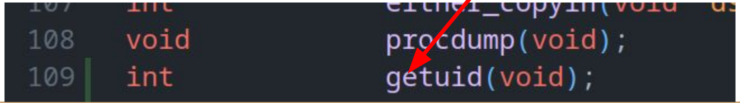
```
uint64
sys_getuid(void)
{
  return getuid();
}
```

kernel/proc.c



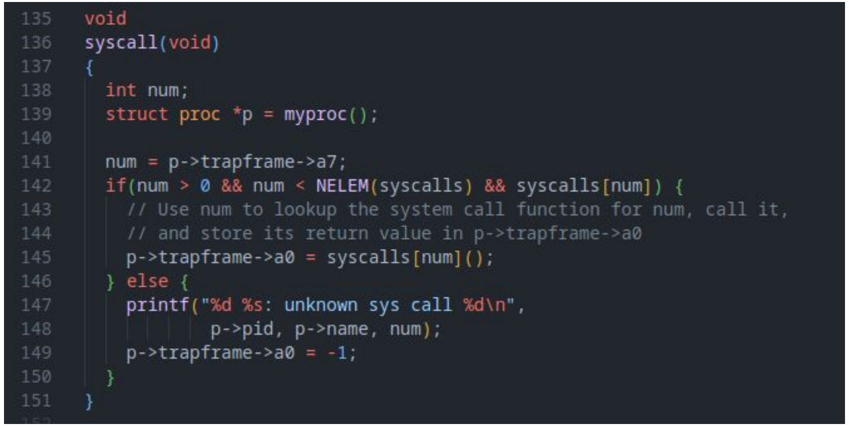
```
int uid=123;

int getuid(void) {
  return uid;
}
```

to make getuid() accessible from kernel/sysproc.c, add this **declaration** in **kernel/defs.h**. If no new function was defined, this line wouldn't have been required.



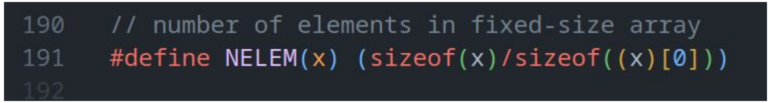
```
int             either_copyin(void
void            procdump(void);
int             getuid(void);
```

kernel/syscall.c



```
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

kernel/defs.h

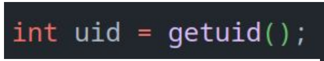


```
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

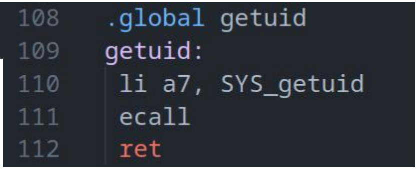
A series of things happen

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()->kernel/sysproc.c

->kernel/proc.c

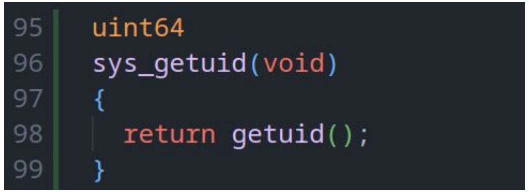


```c
int uid = getuid();
```



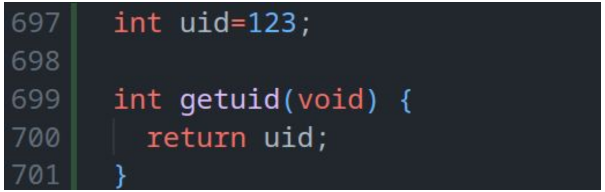
```
    .global getuid
    getuid:
     li a7, SYS_getuid
     ecall
     ret
```

kernel/sysproc.c

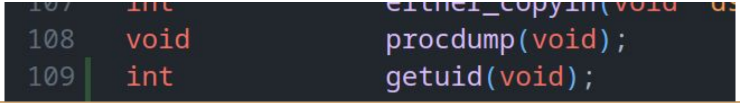


```c
uint64
sys_getuid(void)
{
  return getuid();
}
```

kernel/proc.c



```c
int uid=123;

int getuid(void) {
  return uid;
}
```

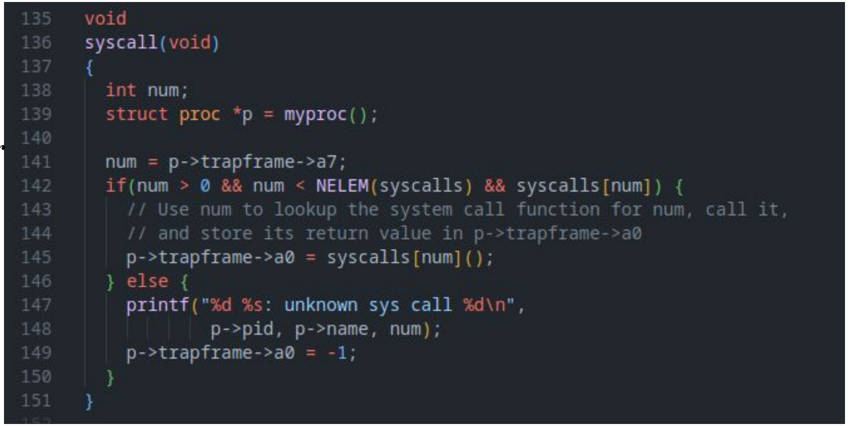


```c
int             either_copyin(void  us
void            procdump(void);
int             getuid(void);
```

Why bother adding another function in proc.c?

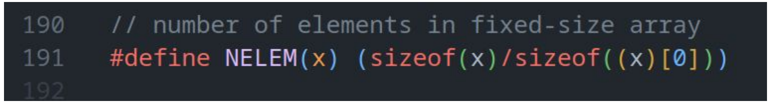
<u>Ans:</u> It's the style of the coding in xv6, treating sysproc.c as a mediator having **sys call handlers** who verifies input and then delegates. We should adhere to the practice.

kernel/syscall.c



```c
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

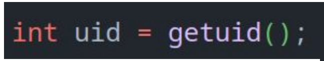
kernel/defs.h



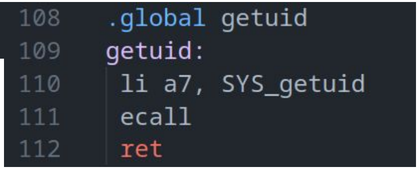
```c
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

A series of things happened(now returning)

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()->kernel/sysproc.c

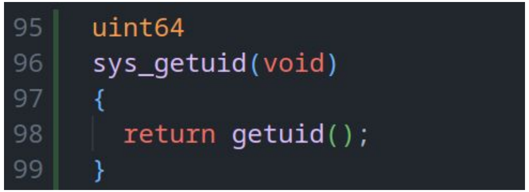


```c
int uid = getuid();
```



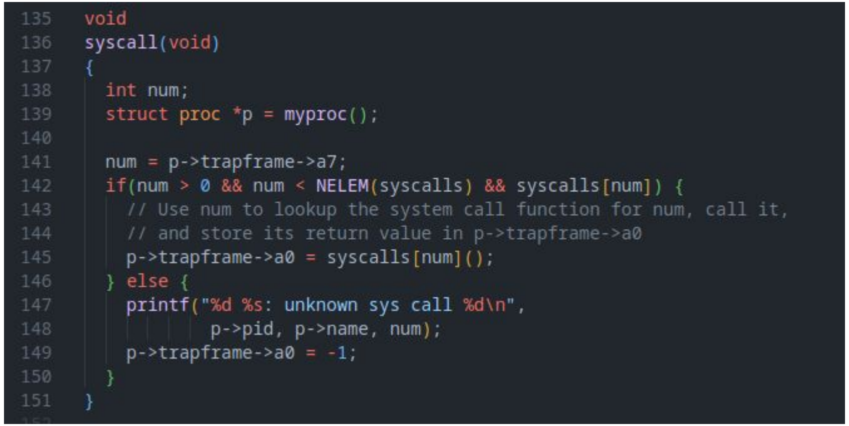
```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

kernel/sysproc.c



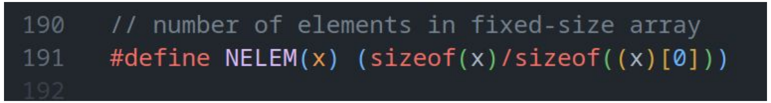
```c
uint64
sys_getuid(void)
{
  return getuid();
}
```

kernel/syscall.c



```c
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

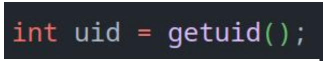
kernel/defs.h



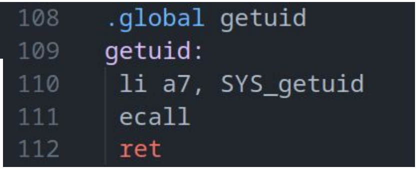
```c
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

A series of things happened(now returning)

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->syscall.c/syscall()

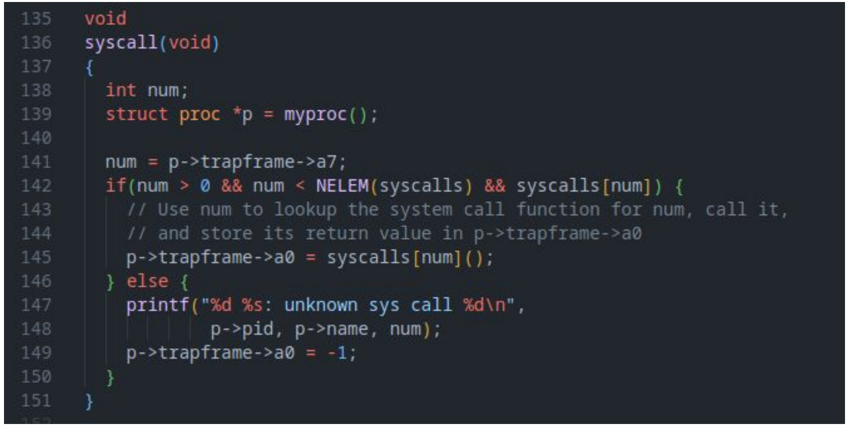


```c
int uid = getuid();
```



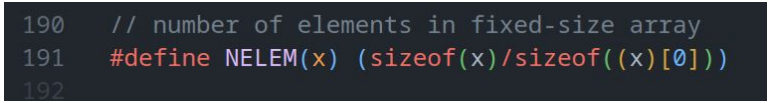
```
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

kernel/syscall.c



```c
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

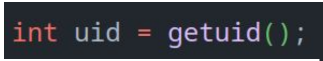
kernel/defs.h



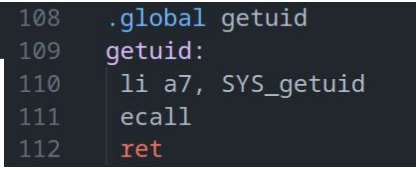
```c
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

A series of things happenIng

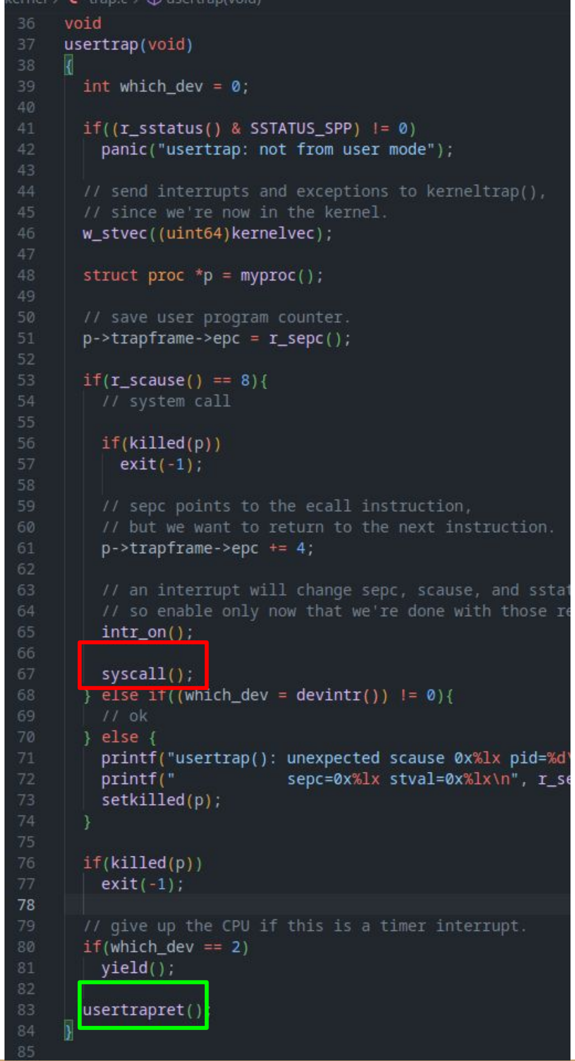
user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()



```c
int uid = getuid();
```



```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

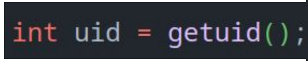


```c
void
usertrap(void)
{
  int which_dev = 0;

  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 sstat
    // so enable only now that we're done with those re
    intr_on();

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

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

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret()
}
```

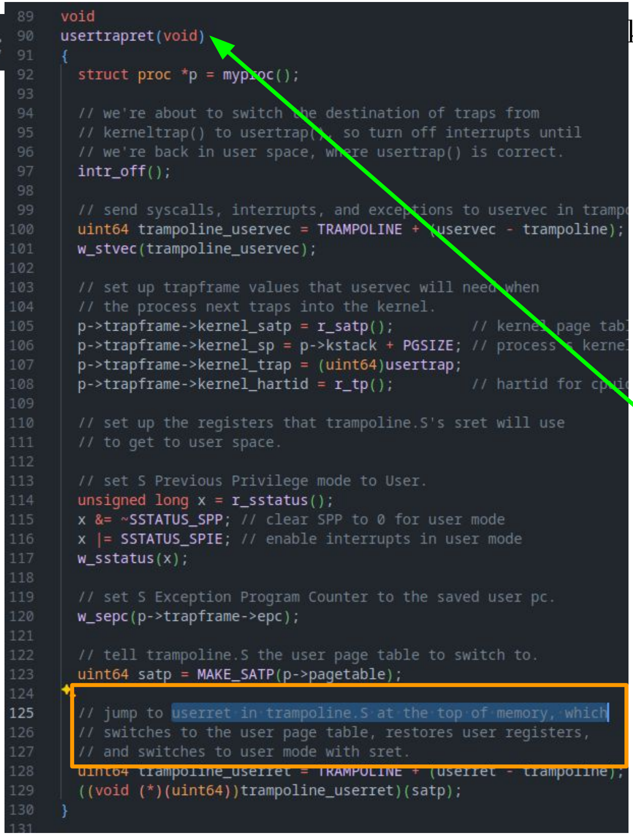
A series of things happenIng

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->

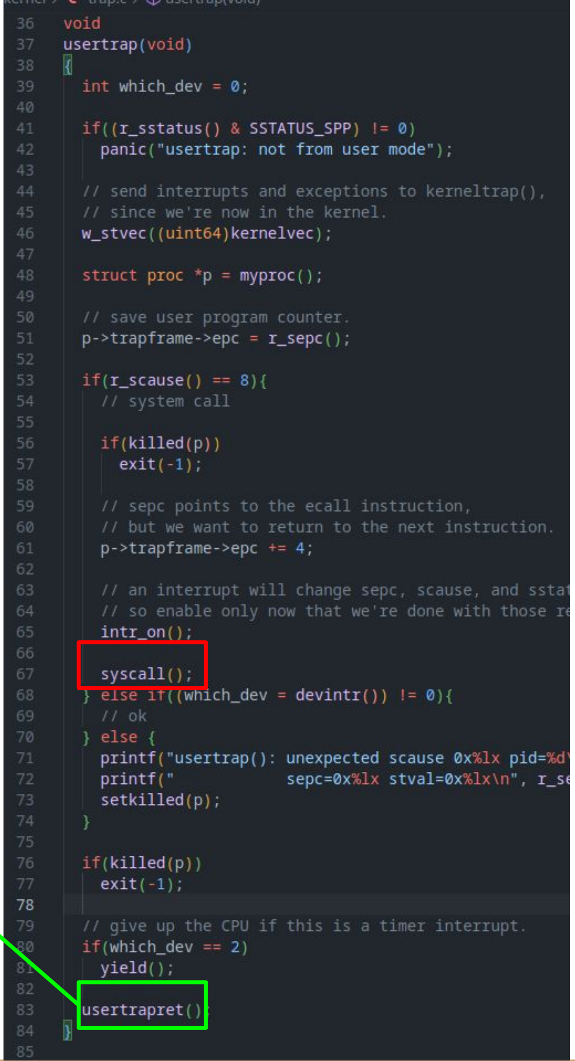
kernel/trap.c/usertrapret()



```c
int uid = getuid();
```



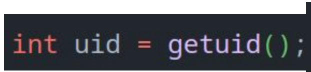
```c
void
usertrapret(void)
{
  struct proc *p = myproc();

  // we're about to switch the destination of traps from
  // kerneltrap() to usertrap(), so turn off interrupts until
  // we're back in user space, where usertrap() is correct.
  intr_off();

  // send syscalls, interrupts, and exceptions to uservec in trampo
  uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
  w_stvec(trampoline_uservec);

  // set up trapframe values that uservec will need when
  // the process next traps into the kernel.
  p->trapframe->kernel_satp = r_satp();         // kernel page tabl
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel
  p->trapframe->kernel_trap = (uint64)usertrap;
  p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid

  // set up the registers that trampoline.S's sret will use
  // to get to user space.

  // set S Previous Privilege mode to User.
  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
  x |= SSTATUS_SPIE; // enable interrupts in user mode
  w_sstatus(x);

  // set S Exception Program Counter to the saved user pc.
  w_sepc(p->trapframe->epc);

  // tell trampoline.S the user page table to switch to.
  uint64 satp = MAKE_SATP(p->pagetable);

  // jump to userret in trampoline.S at the top of memory, which
  // switches to the user page table, restores user registers,
  // and switches to user mode with sret.
  uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
  ((void (*)(uint64))trampoline_userret)(satp);
}
```



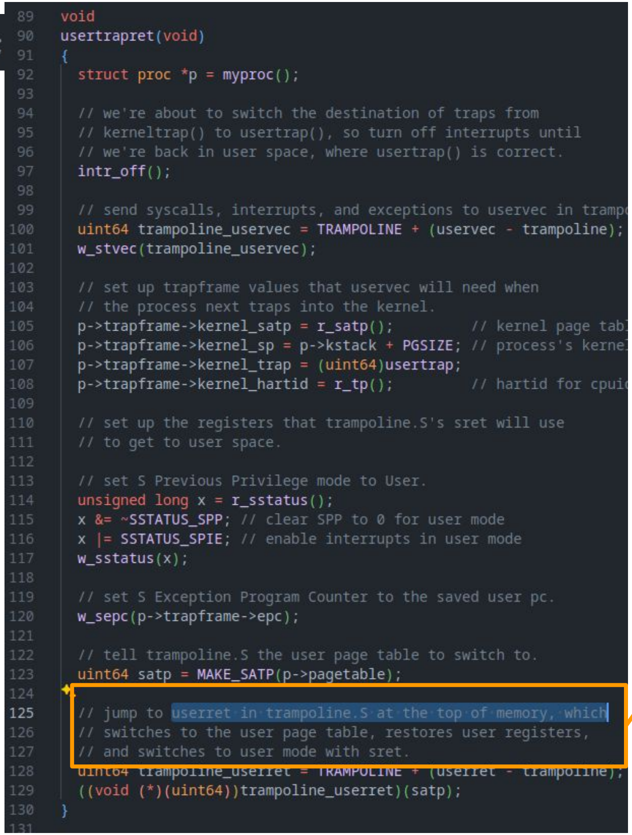
```c
void
usertrap(void)
{
  int which_dev = 0;

  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 sstat
    // so enable only now that we're done with those re
    intr_on();

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

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

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

A series of things happenIng

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->kernel/trampoline.S/userret()

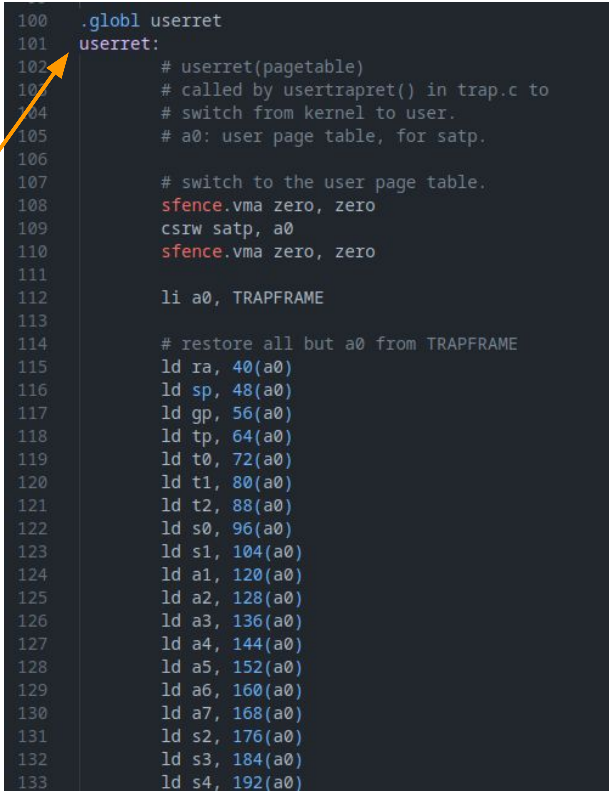


```
int uid = getuid();
```



```
void
usertrapret(void)
{
  struct proc *p = myproc();

  // we're about to switch the destination of traps from
  // kerneltrap() to usertrap(), so turn off interrupts until
  // we're back in user space, where usertrap() is correct.
  intr_off();

  // send syscalls, interrupts, and exceptions to uservec in trampo
  uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
  w_stvec(trampoline_uservec);

  // set up trapframe values that uservec will need when
  // the process next traps into the kernel.
  p->trapframe->kernel_satp = r_satp();         // kernel page tabl
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kerne
  p->trapframe->kernel_trap = (uint64)usertrap;
  p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid

  // set up the registers that trampoline.S's sret will use
  // to get to user space.

  // set S Previous Privilege mode to User.
  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
  x |= SSTATUS_SPIE; // enable interrupts in user mode
  w_sstatus(x);

  // set S Exception Program Counter to the saved user pc.
  w_sepc(p->trapframe->epc);

  // tell trampoline.S the user page table to switch to.
  uint64 satp = MAKE_SATP(p->pagetable);

  // jump to userret in trampoline.S at the top of memory, which
  // switches to the user page table, restores user registers,
  // and switches to user mode with sret.
  uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
  ((void (*)(uint64))trampoline_userret)(satp);
}
```

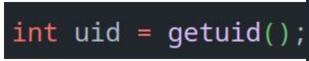
kernel/trampoline.S



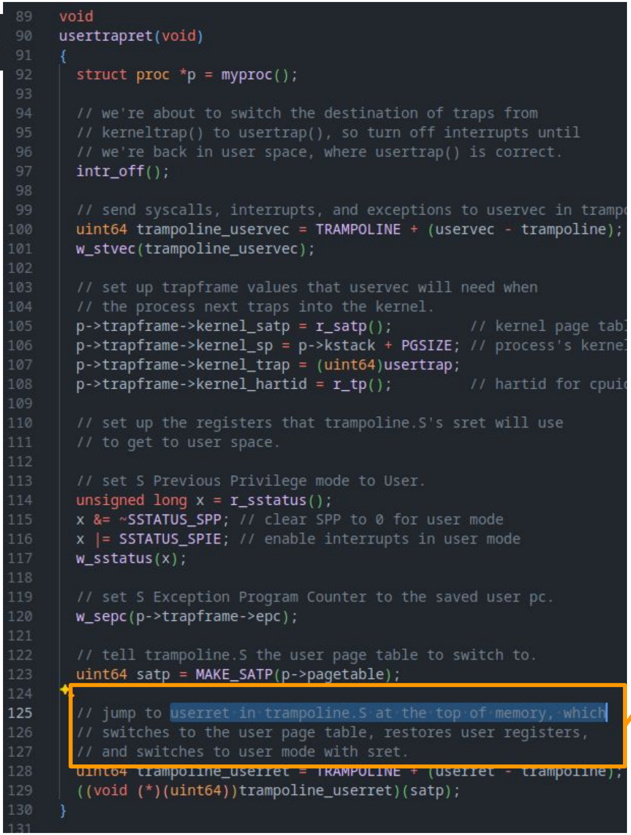
```
.globl userret
userret:
        # userret(pagetable)
        # called by usertrapret() in trap.c to
        # switch from kernel to user.
        # a0: user page table, for satp.

        # switch to the user page table.
        sfence.vma zero, zero
        csrw satp, a0
        sfence.vma zero, zero

        li a0, TRAPFRAME

        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
        ld s1, 104(a0)
        ld a1, 120(a0)
        ld a2, 128(a0)
        ld a3, 136(a0)
        ld a4, 144(a0)
        ld a5, 152(a0)
        ld a6, 160(a0)
        ld a7, 168(a0)
        ld s2, 176(a0)
        ld s3, 184(a0)
        ld s4, 192(a0)
```

A series of things happenIng

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->kernel/trampoline.S/userret()

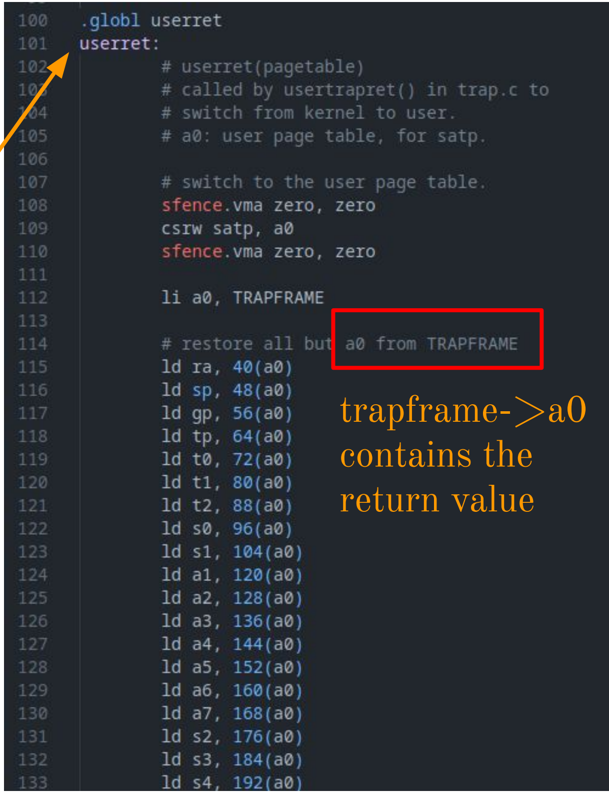


```c
int uid = getuid();
```



```c
void
usertrapret(void)
{
  struct proc *p = myproc();

  // we're about to switch the destination of traps from
  // kerneltrap() to usertrap(), so turn off interrupts until
  // we're back in user space, where usertrap() is correct.
  intr_off();

  // send syscalls, interrupts, and exceptions to uservec in trampo
  uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
  w_stvec(trampoline_uservec);

  // set up trapframe values that uservec will need when
  // the process next traps into the kernel.
  p->trapframe->kernel_satp = r_satp();         // kernel page tabl
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel
  p->trapframe->kernel_trap = (uint64)usertrap;
  p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid

  // set up the registers that trampoline.S's sret will use
  // to get to user space.

  // set S Previous Privilege mode to User.
  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
  x |= SSTATUS_SPIE; // enable interrupts in user mode
  w_sstatus(x);

  // set S Exception Program Counter to the saved user pc.
  w_sepc(p->trapframe->epc);

  // tell trampoline.S the user page table to switch to.
  uint64 satp = MAKE_SATP(p->pagetable);

  // jump to userret in trampoline.S at the top of memory, which
  // switches to the user page table, restores user registers,
  // and switches to user mode with sret.
  uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
  ((void (*)(uint64))trampoline_userret)(satp);
}
```

kernel/trampoline.S

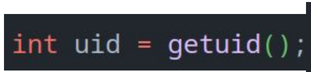


```asm
.globl userret
userret:
        # userret(pagetable)
        # called by usertrapret() in trap.c to
        # switch from kernel to user.
        # a0: user page table, for satp.

        # switch to the user page table.
        sfence.vma zero, zero
        csrw satp, a0
        sfence.vma zero, zero

        li a0, TRAPFRAME

        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
        ld s1, 104(a0)
        ld a1, 120(a0)
        ld a2, 128(a0)
        ld a3, 136(a0)
        ld a4, 144(a0)
        ld a5, 152(a0)
        ld a6, 160(a0)
        ld a7, 168(a0)
        ld s2, 176(a0)
        ld s3, 184(a0)
        ld s4, 192(a0)
```

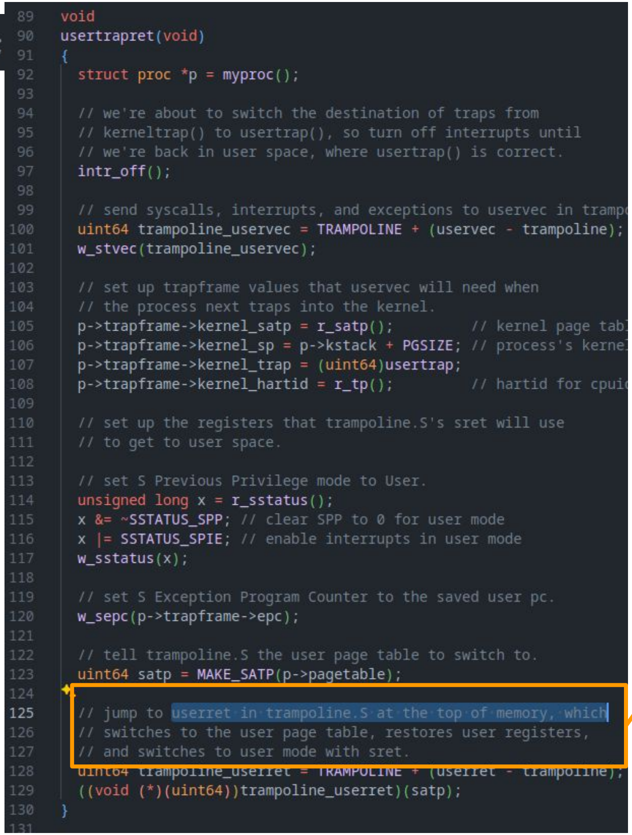
trapframe->a0 contains the return value

A series of things happenIng

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->kernel/trampoline.S/userret()

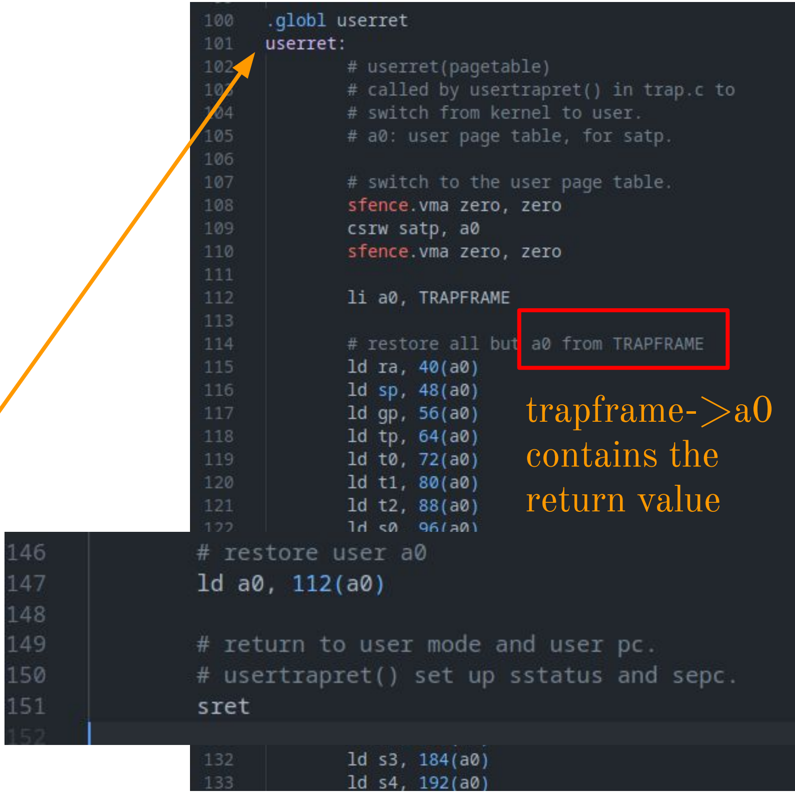


```
int uid = getuid();
```



```
void
usertrapret(void)
{
  struct proc *p = myproc();

  // we're about to switch the destination of traps from
  // kerneltrap() to usertrap(), so turn off interrupts until
  // we're back in user space, where usertrap() is correct.
  intr_off();

  // send syscalls, interrupts, and exceptions to uservec in trampo
  uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
  w_stvec(trampoline_uservec);

  // set up trapframe values that uservec will need when
  // the process next traps into the kernel.
  p->trapframe->kernel_satp = r_satp();         // kernel page tabl
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel
  p->trapframe->kernel_trap = (uint64)usertrap;
  p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid

  // set up the registers that trampoline.S's sret will use
  // to get to user space.

  // set S Previous Privilege mode to User.
  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
  x |= SSTATUS_SPIE; // enable interrupts in user mode
  w_sstatus(x);

  // set S Exception Program Counter to the saved user pc.
  w_sepc(p->trapframe->epc);

  // tell trampoline.S the user page table to switch to.
  uint64 satp = MAKE_SATP(p->pagetable);

  // jump to userret in trampoline.S at the top of memory, which
  // switches to the user page table, restores user registers,
  // and switches to user mode with sret.
  uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
  ((void (*)(uint64))trampoline_userret)(satp);
}
```

kernel/trampoline.S



```
.globl userret
userret:
        # userret(pagetable)
        # called by usertrapret() in trap.c to
        # switch from kernel to user.
        # a0: user page table, for satp.

        # switch to the user page table.
        sfence.vma zero, zero
        csrw satp, a0
        sfence.vma zero, zero

        li a0, TRAPFRAME

        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
```

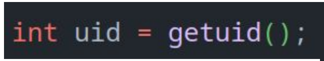
```
        # restore user a0
        ld a0, 112(a0)

        # return to user mode and user pc.
        # usertrapret() set up sstatus and sepc.
        sret
```

```
        ld s3, 184(a0)
        ld s4, 192(a0)
```

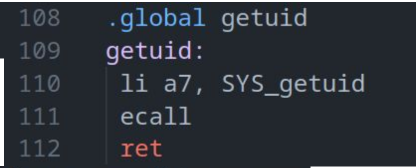
trapframe->a0 contains the return value

A series of things happen [RECAP & NOTE]

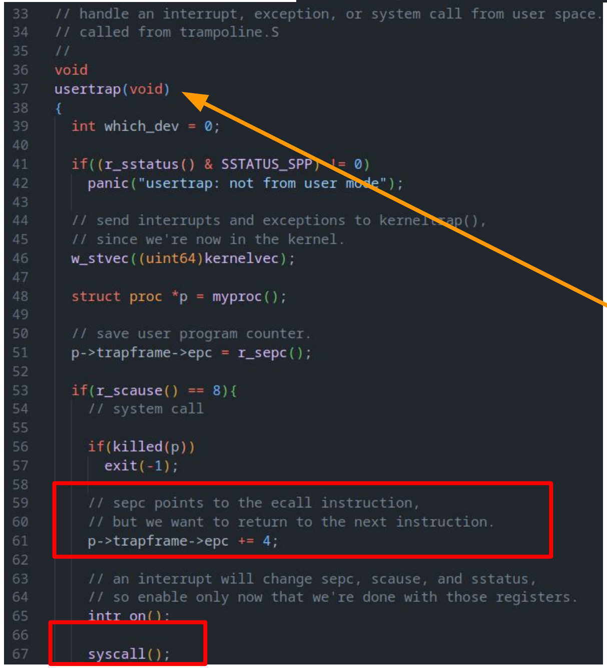
user/gibuid.c -> user/Usys.s -> kernel/trampoline.S/uservec() -> kernel/trap.c/usertrap()



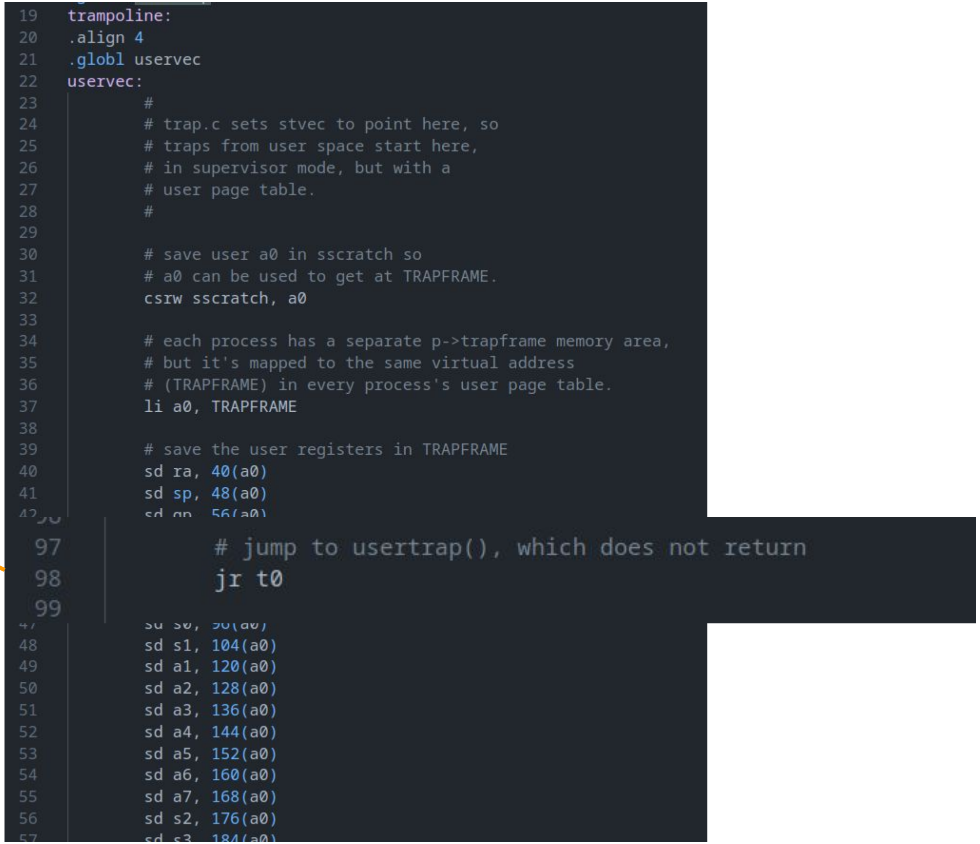
```c
int uid = getuid();
```



```asm
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```



```c
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
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();

    syscall();
```



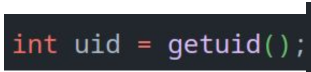
```asm
trampoline:
.align 4
.globl uservec
uservec:
        #
        # trap.c sets stvec to point here, so
        # traps from user space start here,
        # in supervisor mode, but with a
        # user page table.
        #

        # save user a0 in sscratch so
        # a0 can be used to get at TRAPFRAME.
        csrw sscratch, a0

        # each process has a separate p->trapframe memory area,
        # but it's mapped to the same virtual address
        # (TRAPFRAME) in every process's user page table.
        li a0, TRAPFRAME

        # save the user registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
```

```asm
97          # jump to usertrap(), which does not return
98          jr t0
99
```

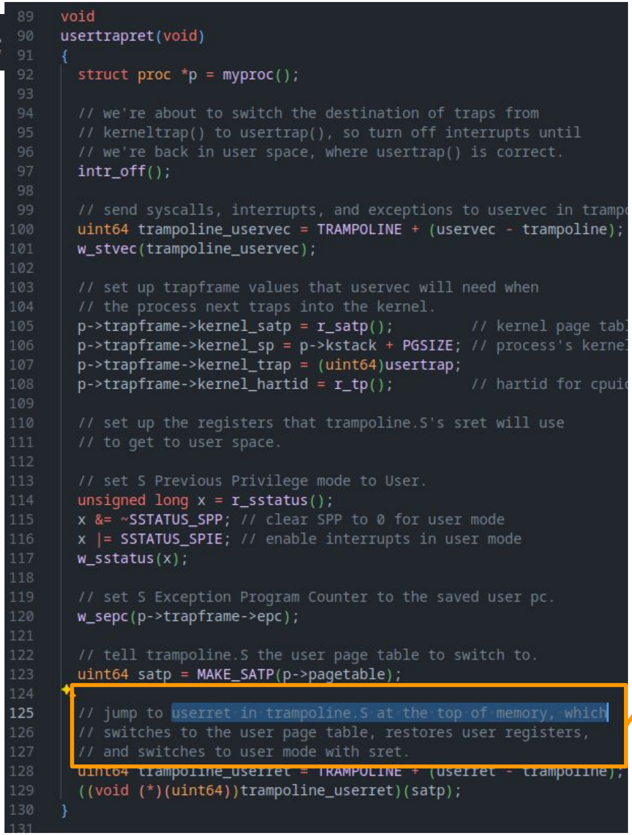
```asm
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
```

A series of things happenIng

user/gibuid.c->user/Usys.s ->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()->kernel/trampoline.S/userret()

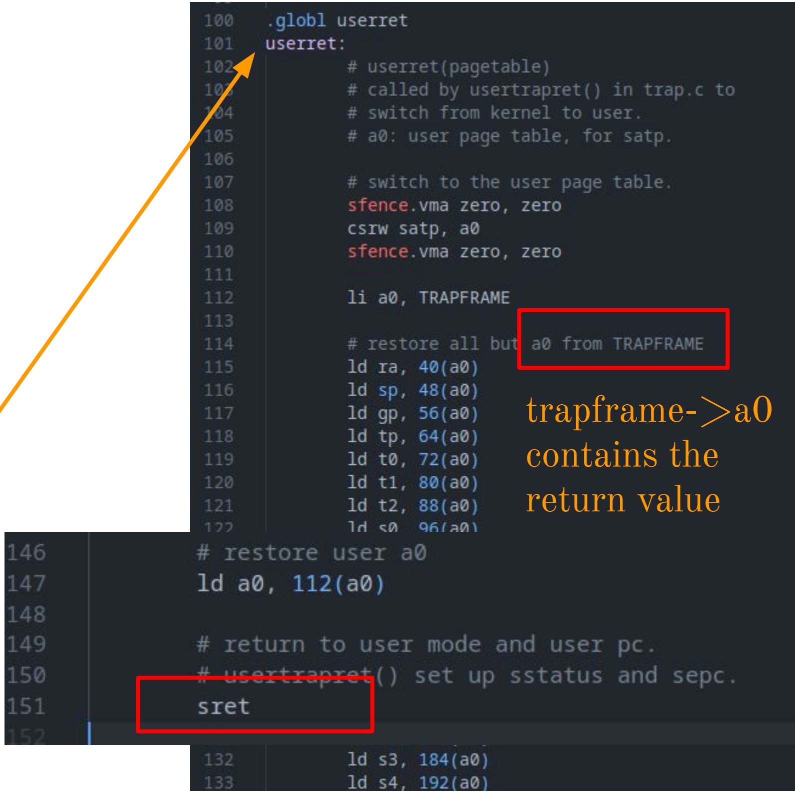


```c
int uid = getuid();
```



```c
void
usertrapret(void)
{
  struct proc *p = myproc();

  // we're about to switch the destination of traps from
  // kerneltrap() to usertrap(), so turn off interrupts until
  // we're back in user space, where usertrap() is correct.
  intr_off();

  // send syscalls, interrupts, and exceptions to uservec in trampo
  uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
  w_stvec(trampoline_uservec);

  // set up trapframe values that uservec will need when
  // the process next traps into the kernel.
  p->trapframe->kernel_satp = r_satp();         // kernel page tabl
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel
  p->trapframe->kernel_trap = (uint64)usertrap;
  p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid

  // set up the registers that trampoline.S's sret will use
  // to get to user space.

  // set S Previous Privilege mode to User.
  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
  x |= SSTATUS_SPIE; // enable interrupts in user mode
  w_sstatus(x);

  // set S Exception Program Counter to the saved user pc.
  w_sepc(p->trapframe->epc);

  // tell trampoline.S the user page table to switch to.
  uint64 satp = MAKE_SATP(p->pagetable);

  // jump to userret in trampoline.S at the top of memory, which
  // switches to the user page table, restores user registers,
  // and switches to user mode with sret.
  uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
  ((void (*)(uint64))trampoline_userret)(satp);
}
```

kernel/trampoline.S



```asm
.globl userret
userret:
        # userret(pagetable)
        # called by usertrapret() in trap.c to
        # switch from kernel to user.
        # a0: user page table, for satp.

        # switch to the user page table.
        sfence.vma zero, zero
        csrw satp, a0
        sfence.vma zero, zero

        li a0, TRAPFRAME

        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
```

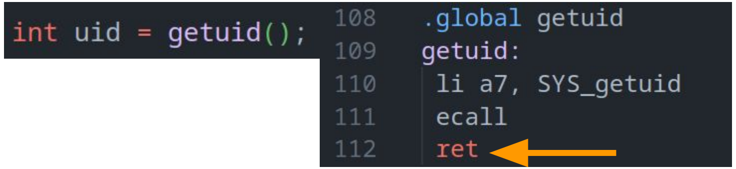
```asm
        # restore user a0
        ld a0, 112(a0)

        # return to user mode and user pc.
        # usertrapret() set up sstatus and sepc.
        sret
```

```asm
        ld s3, 184(a0)
        ld s4, 192(a0)
```

trapframe->a0 contains the return value

A series of things happened

user/gibuid.c->user/Usys.s

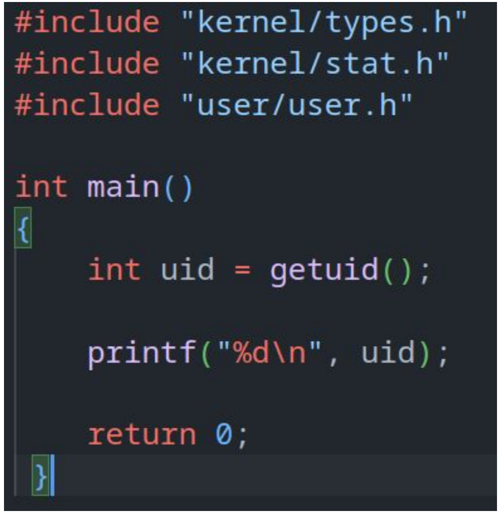


```
int uid = getuid();
```

```
.global getuid
getuid:
 li a7, SYS_getuid
 ecall
 ret
```

A series of things happened (We are back!)

user/gibuid.c

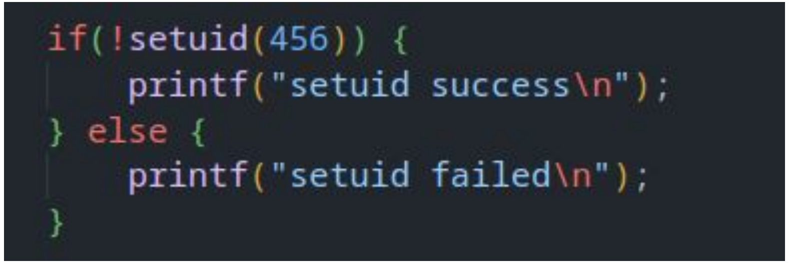


```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main()
{
    int uid = getuid();

    printf("%d\n", uid);

    return 0;
}
```

Okay, but how to pass arguments?

Say, we want to set the uid to 456! (As we know it’s simply a variable)

The call will be setuid(456)



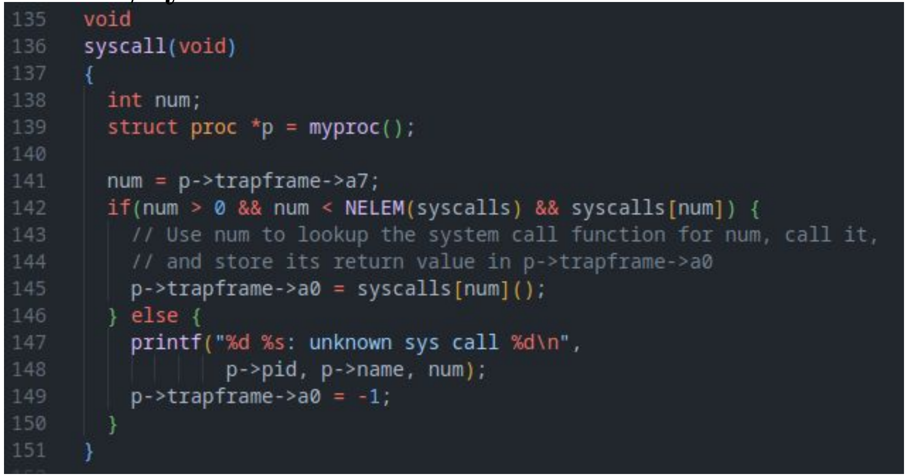
```
if(!setuid(456)) {
    printf("setuid success\n");
} else {
    printf("setuid failed\n");
}
```

xv6 stores the arguments passed in registers a0, a1, a2, a3, a4, a5

Yet another system call

user/gibuid.c->user/Usys.s->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()***[stores registers in trapframe]***->kernel/syscall.c/syscall()

kernel/syscall.c

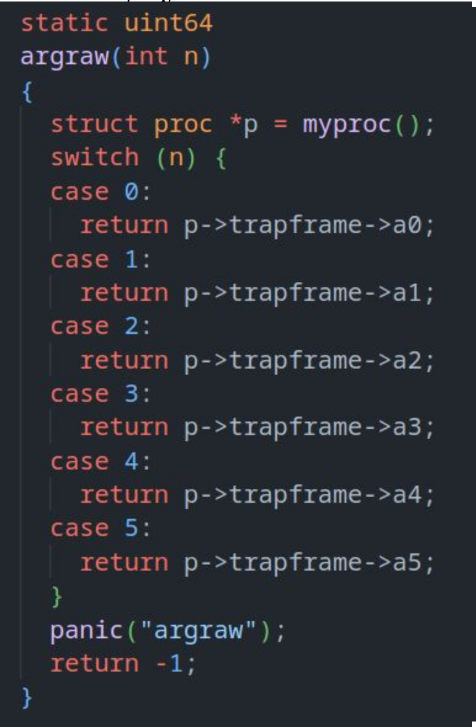


```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

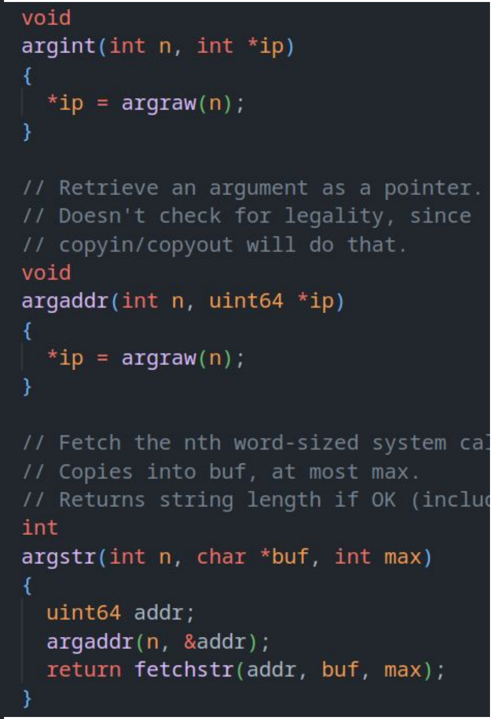
Yet another system call

user/gibuid.c->user/Usys.s->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()***[stores registers in trapframe]***->kernel/syscall.c/syscall()

kernel/syscall.c

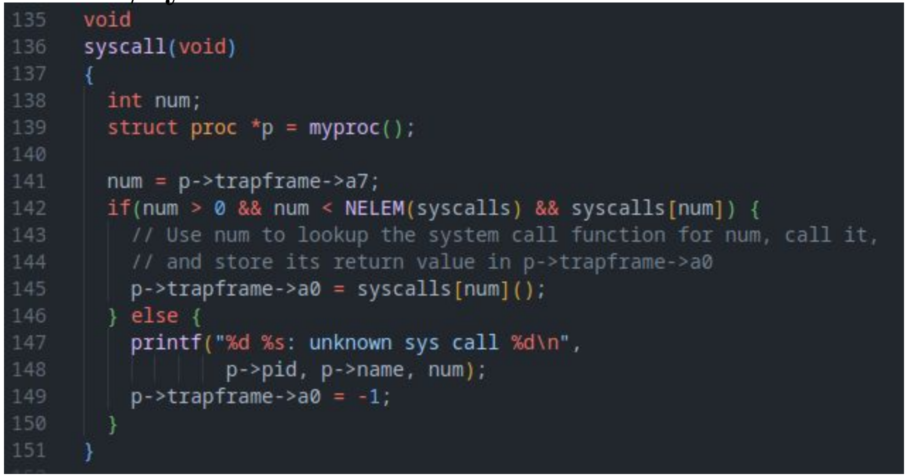


```
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}
```



```
void
argint(int n, int *ip)
{
  *ip = argraw(n);
}

// Retrieve an argument as a pointer.
// Doesn't check for legality, since
// copyin/copyout will do that.
void
argaddr(int n, uint64 *ip)
{
  *ip = argraw(n);
}

// Fetch the nth word-sized system ca
// Copies into buf, at most max.
// Returns string length if OK (inclu
int
argstr(int n, char *buf, int max)
{
  uint64 addr;
  argaddr(n, &addr);
  return fetchstr(addr, buf, max);
}
```

kernel/syscall.c

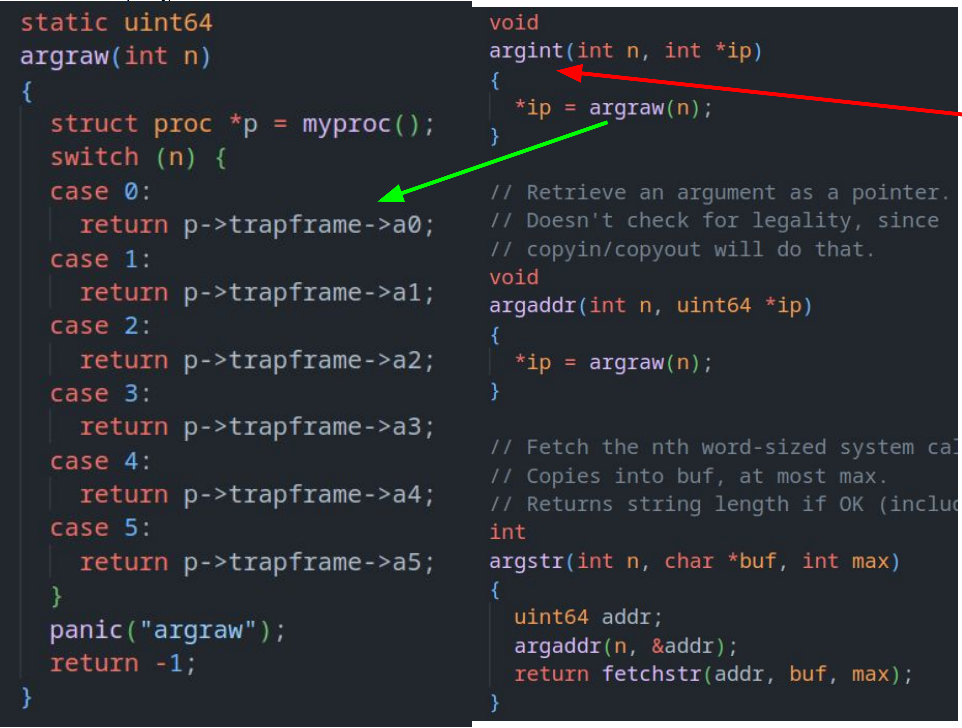


```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

Yet another system call

user/gibuid.c->user/Usys.s->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()*[stores registers in trapframe]*->
kernel/syscall.c/syscall() ***[say, it called sys_setuid() in sysproc.c]***

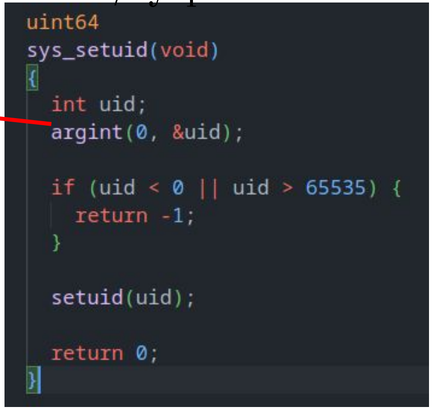
kernel/syscall.c



```
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}
```

```
void
argint(int n, int *ip)
{
  *ip = argraw(n);
}

// Retrieve an argument as a pointer.
// Doesn't check for legality, since
// copyin/copyout will do that.
void
argaddr(int n, uint64 *ip)
{
  *ip = argraw(n);
}

// Fetch the nth word-sized system ca
// Copies into buf, at most max.
// Returns string length if OK (incluc
int
argstr(int n, char *buf, int max)
{
  uint64 addr;
  argaddr(n, &addr);
  return fetchstr(addr, buf, max);
}
```

kernel/sysproc.c

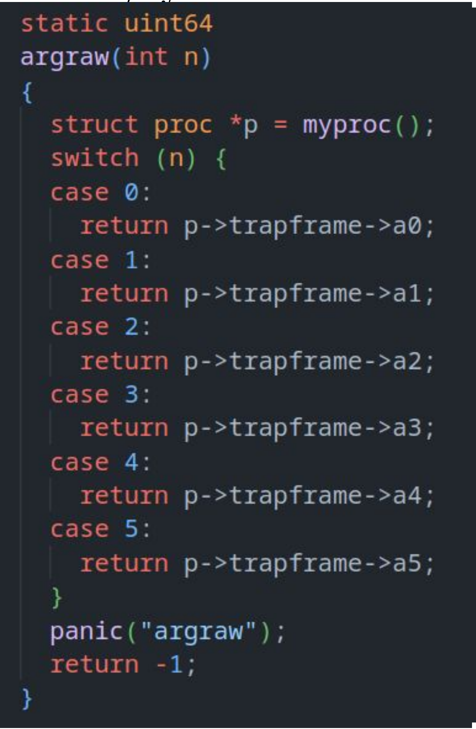


```
uint64
sys_setuid(void)
{
  int uid;
  argint(0, &uid);

  if (uid < 0 || uid > 65535) {
    return -1;
  }

  setuid(uid);

  return 0;
}
```

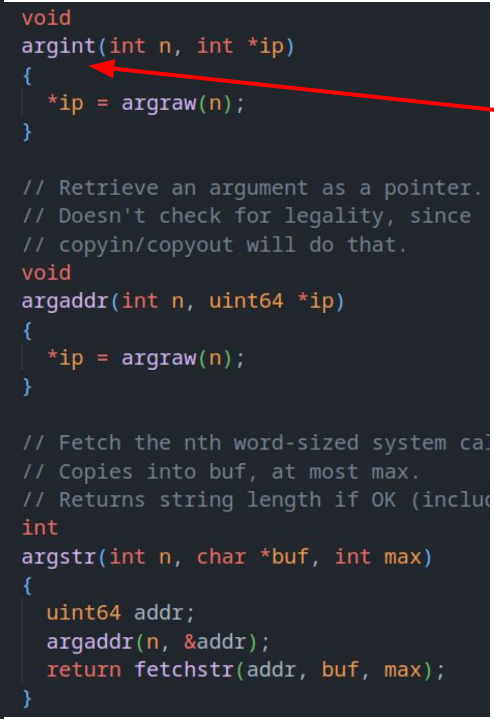
Yet another system call

user/gibuid.c->user/Usys.s->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()***[stores registers in trapframe]***-> kernel/syscall.c/syscall() ***[say, it called sys_setuid() in sysproc.c]***

kernel/syscall.c

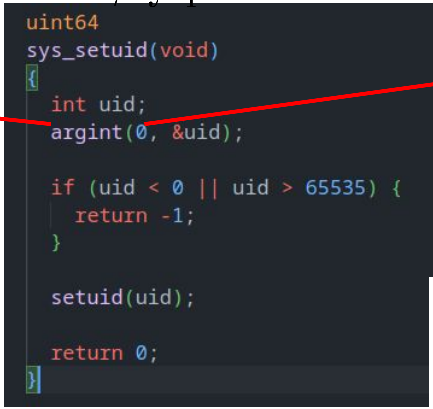


```c
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}
```



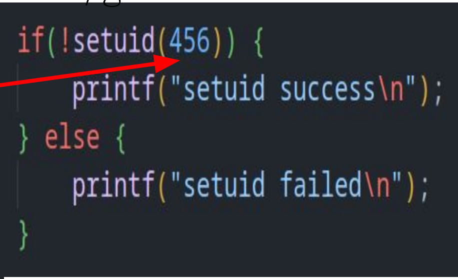
```c
void
argint(int n, int *ip)
{
  *ip = argraw(n);
}

// Retrieve an argument as a pointer.
// Doesn't check for legality, since
// copyin/copyout will do that.
void
argaddr(int n, uint64 *ip)
{
  *ip = argraw(n);
}

// Fetch the nth word-sized system ca
// Copies into buf, at most max.
// Returns string length if OK (incluc
int
argstr(int n, char *buf, int max)
{
  uint64 addr;
  argaddr(n, &addr);
  return fetchstr(addr, buf, max);
}
```

kernel/sysproc.c



```c
uint64
sys_setuid(void)
{
  int uid;
  argint(0, &uid);

  if (uid < 0 || uid > 65535) {
    return -1;
  }

  setuid(uid);

  return 0;
}
```

user/gibuid.c



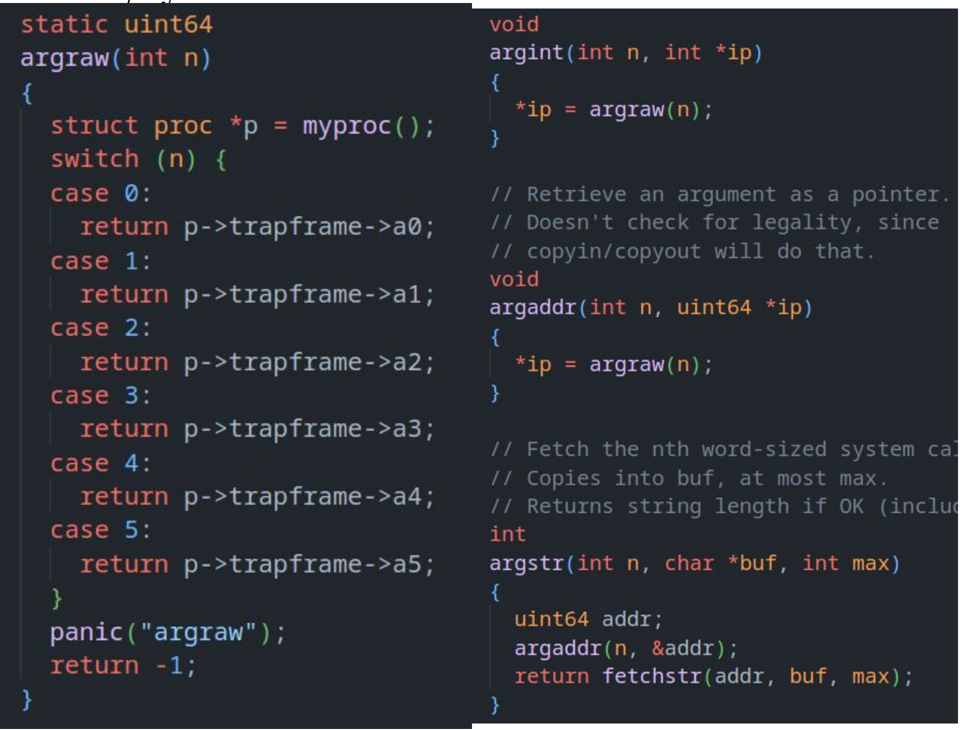
```c
if(!setuid(456)) {
    printf("setuid success\n");
} else {
    printf("setuid failed\n");
}
```

Only one argument. For others, n would be 1,2,3,..

Yet another system call

user/gibuid.c->user/Usys.s->kernel/trampoline.S/uservec()->kernel/trap.c/usertrap()***[stores registers in trapframe]***->
kernel/syscall.c/syscall() ***[say, it called sys_setuid() in sysproc.c]***

kernel/syscall.c

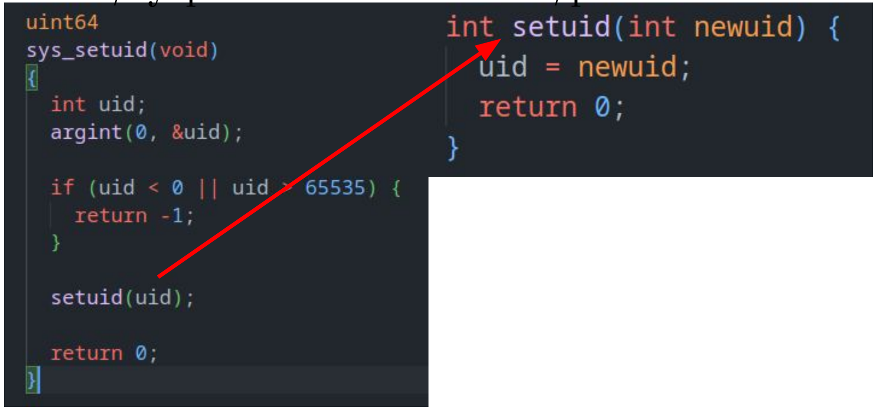


```c
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}
```

```c
void
argint(int n, int *ip)
{
  *ip = argraw(n);
}

// Retrieve an argument as a pointer.
// Doesn't check for legality, since
// copyin/copyout will do that.
void
argaddr(int n, uint64 *ip)
{
  *ip = argraw(n);
}

// Fetch the nth word-sized system ca
// Copies into buf, at most max.
// Returns string length if OK (incluc
int
argstr(int n, char *buf, int max)
{
  uint64 addr;
  argaddr(n, &addr);
  return fetchstr(addr, buf, max);
}
```

kernel/sysproc.c

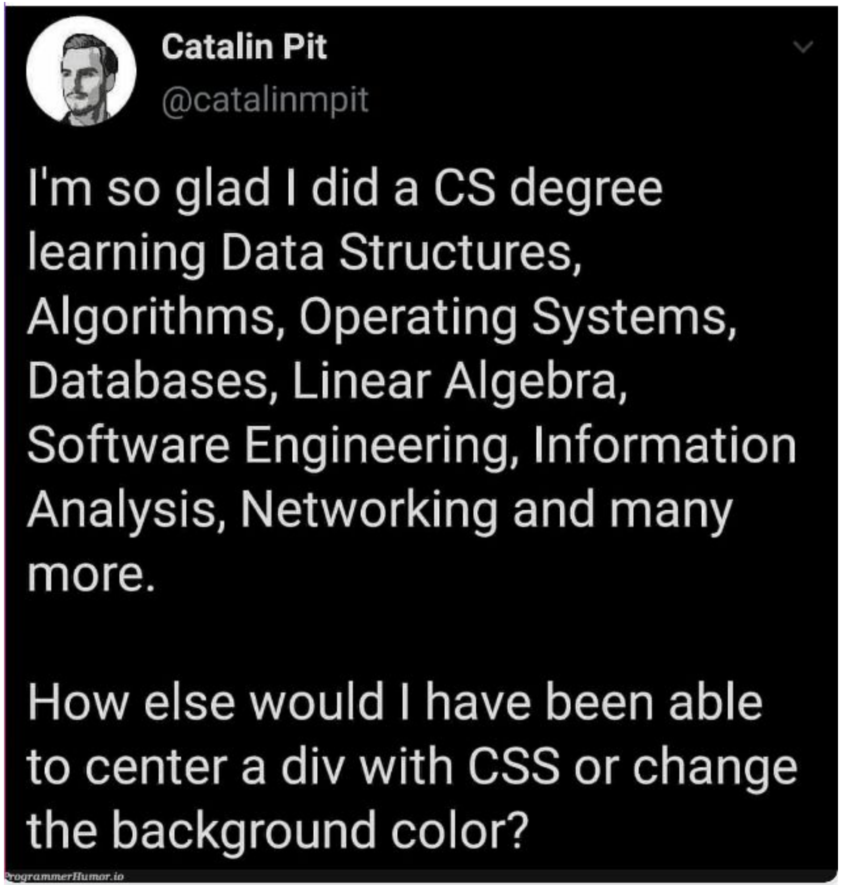
kernel/proc.c



```c
uint64
sys_setuid(void)
{
  int uid;
  argint(0, &uid);

  if (uid < 0 || uid > 65535) {
    return -1;
  }

  setuid(uid);

  return 0;
}
```

```c
int setuid(int newuid) {
  uid = newuid;
  return 0;
}
```

Thank You



Catalin Pit

@catalinmpit

I'm so glad I did a CS degree learning Data Structures, Algorithms, Operating Systems, Databases, Linear Algebra, Software Engineering, Information Analysis, Networking and many more.

How else would I have been able to center a div with CSS or change the background color?

ProgrammerHumor.io