Chapter 6
Limited Direct Execution

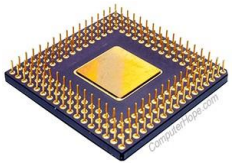
Dong Dai
CIS/UD
dai@udel.edu

Mindset

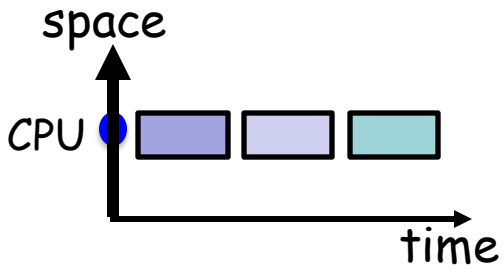
- Empty yourself (meditate 😌)
- Pretend that you were the first person being asked to design "**something**" to make a better use of a "bare" computing machine (CPU + memory)
- Let's **relive** the invention of Unix

How to virtualize CPU?

Introduction


single CPU

- **Share** one physical CPU among processes running seemingly at the same time
- Virtualize CPU via "**time**" sharing



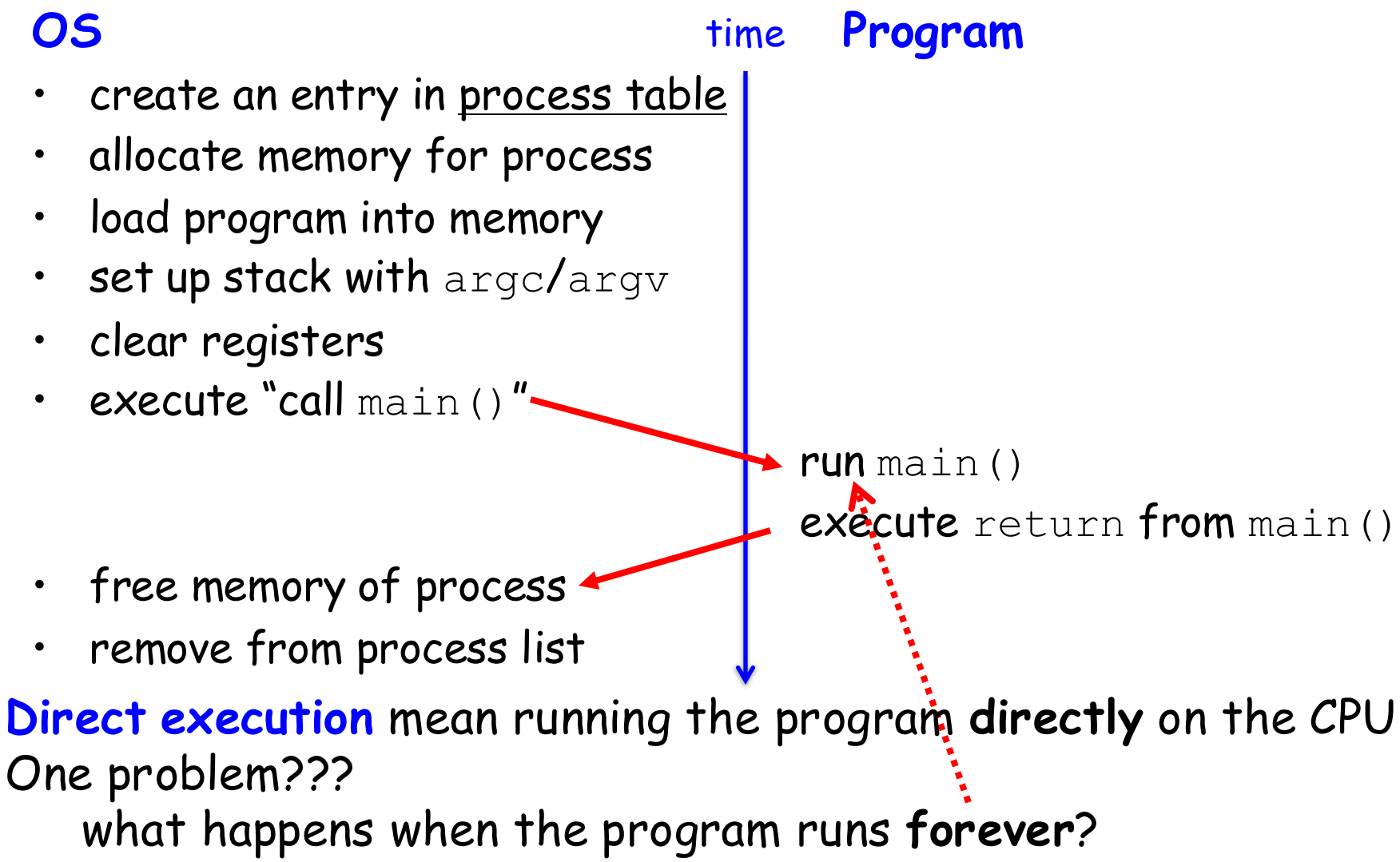
- **Two challenges** to **any** such virtualization mechanism
  - **performance** – with minimum overhead
  - **control** – **retain control of CPU** while running processes efficiently OS is a resource manager

➔ Obtaining **high performance while** maintaining **control** is the key challenge in building OS

How to obtain **best performance** of running your program?

Run programs (executables) **directly** (or **natively**) on CPU

Direction Execution



Direct execution mean running the program **directly** on the CPU

One problem???

what happens when the program runs **forever**?

Issues with Direct Execution

- "Direct Execution" – run programs **directly on CPU**
- Two issues (obvious advantage: fast)

  #1 if we just run a program, how can "we" make sure the program doesn't do anything that we don't want it to do, while still running it efficiently?

  #2 when we are running a process, how do "we" **stop** it from running and switch to another process (i.e., how do "we" implement **time sharing**)?
- How to deal with these issue?
- Put limits on running program – without **limits** on running programs, "we" would not be in control of anything

Limits on Running Programs?

What are the limits?

Two aspects

1. Can a running program do whatever it wants, such as performing I/O to/from disk, requesting more system resources, etc.?
   - these operations must be restricted
   - **how** should a user process perform these **"restricted" operations**?
2. **How** to **stop** a running program and **resume** running another program?

One Goal and Two Problems

Goal: efficiently **virtualize** the CPU with **control**

Two problems:

1. How a user process performs "**restricted operations**" (e.g., I/O)? (note that **"calculation"** is no problem!)
2. How to switch between processes?

Solution technique: **Limited Direct Execution (LDE)**

for best performance

for maintaining control

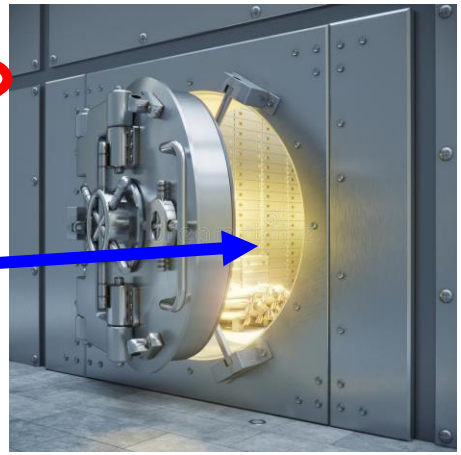
#1 Restricted Operations

- **Advantage** of direct execution:
  - **fast** - program runs **natively** on hardware CPU and thus executes as quickly as one would expect
- **Issue**
  - what if the process wishes to perform "**restricted**" operations (e.g., I/O requests or accessing more resources [e.g., memory])?
    - **how** can a user process be able to perform I/O and other restricted operations, but **without being given** complete control over the system?

How to Perform Restricted Operations?

- A **user process** must be able to perform **restricted operations**, but **without** being given complete control over the system

vault

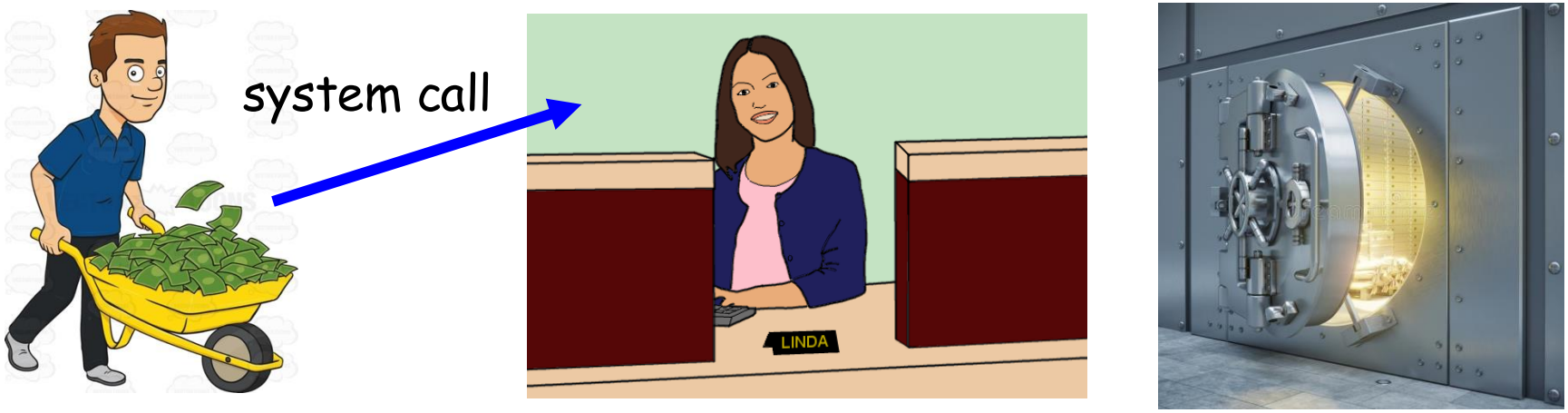


- **Customers ask the teller to perform "restricted operations" on their behalf**
- Both **customer** (user program) and **teller** (system calls) are **software** programs; how could **CPU** tell the difference?
  - **hardware "mode" bit**

Processor Mode

- **User mode:** program that runs in user mode is restricted in what it can do
  - when running in user mode, a process **cannot** issue I/O requests; doing so would result in the processor raising an exception; the OS would then likely kill the process
- **Kernel mode:** which operating system (kernel) runs in
  - in this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions
- One remaining question?
  - what should a user process do when it wishes to perform restricted (privileged) operations, e.g., I/O?

System Calls



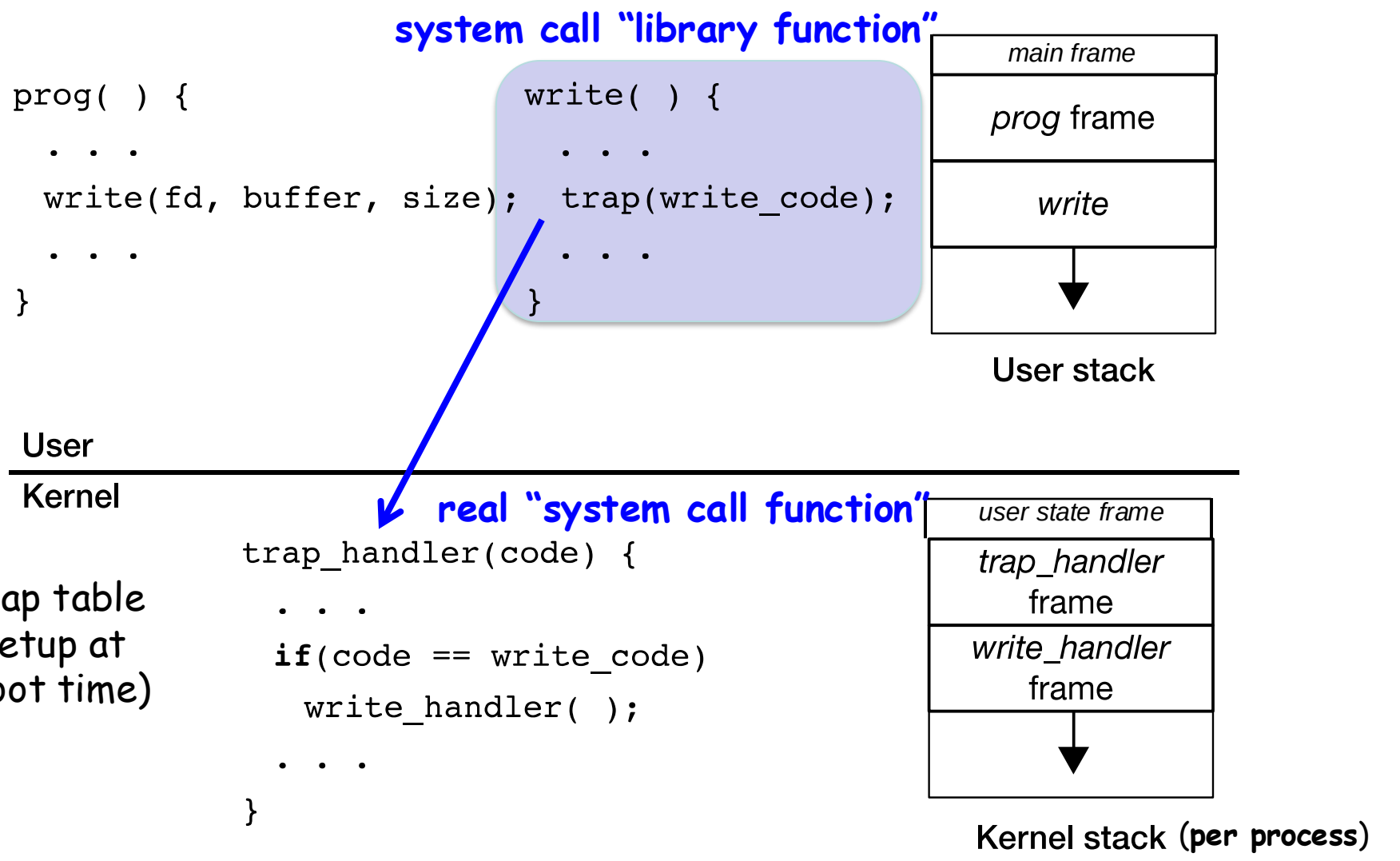
Customers (user process) invoke **system calls** to ask the teller (kernel) to perform restricted operations on their behalf

When and **how** does "mode bit" change its value?

Execution of System Calls

- To execute a system call, **system call library function** executes a special **trap** assembly instruction, which, **simultaneously**, jumps into kernel and raises privilege level to kernel mode (set mode bit to 1)
- Once in the kernel, the system can now perform whatever privileged operations are needed (if allowed permission-wise), and thus do the required work for the calling process [execute **real** system call function]
- When finished, kernel executes a special **return-from-trap** assembly instruction, which returns into the calling system call library function while **simultaneously** reducing the privilege level back to user mode (set mode bit to 0)

System Call


system call "library function"
prog( ) {
. . .
write(fd, buffer, size);
. . .
}
write( ) {
. . .
trap(write_code);
. . .
}
main frame
prog frame
write
User stack
User
Kernel
real "system call function"
trap_handler(code) {
. . .
if(code == write_code)
write_handler( );
. . .
}
trap table (setup at boot time)
user state frame
trap_handler frame
write_handler frame
Kernel stack (per process)

System Calls

Operating System
(pure software)

hardware **Mode Bit** in CPU +

- **trap**
- **return-from-trap**

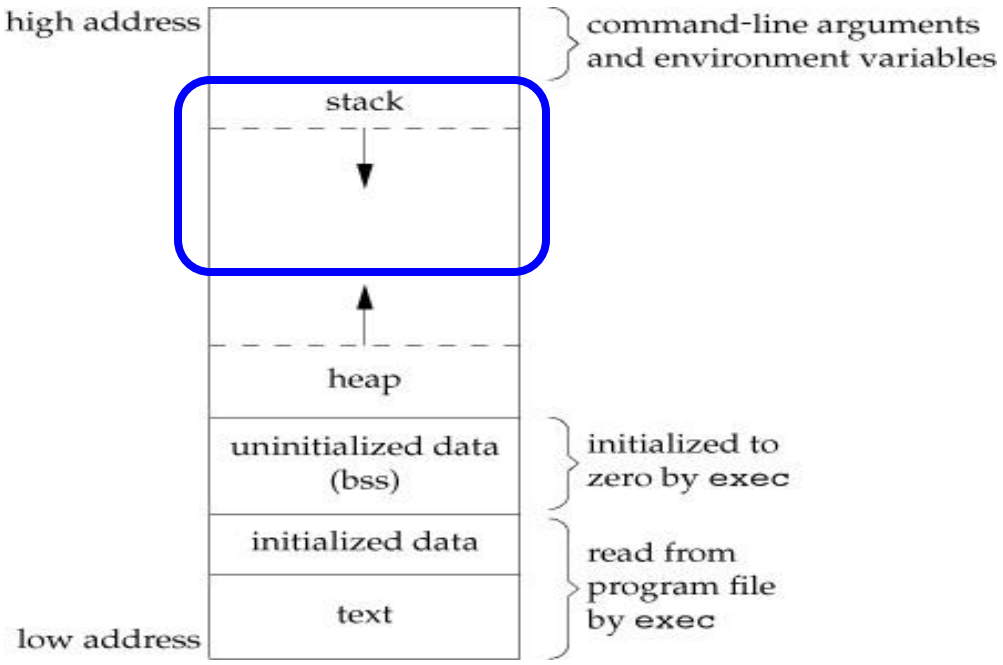
assembly instructions

Protected Control Transfer

Unix Process Memory Layout

Process has 4 segments

- **text** – lowest addresses, fixed size
- **data** – immediately after text for global variables `a.out`
- **stack** – store "**automatic** variables" with function calls
- **heap** - gap in the middle for dynamic memory allocation via `malloc()`



Stevens & Rago's Fig. 7.6

Stack

```
void A(int tmp) {
  if (tmp == 2) return;
  else B();
}

void B(){
  C();
}

void C(){
  A(2);
}

main(){A(1);}
```

- Stack

Stack

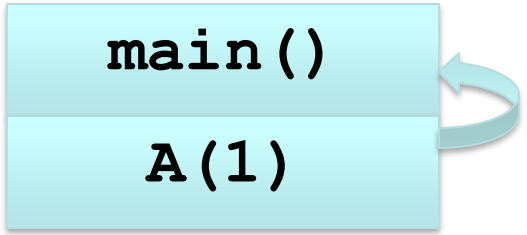
```
void A(int tmp) {
  if (tmp == 2) return;
  else B();
}

void B(){
  C();
}

void C(){
  A(2);
}

main(){A(1);}
```

- Stack



Stack

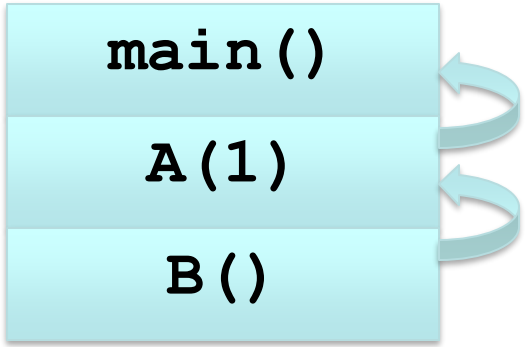
```
void A(int tmp) {
   if (tmp == 2) return;
   else B();
}

void B(){
   C();
}

void C(){
   A(2);
}

main(){A(1);}
```

- Stack



Stack

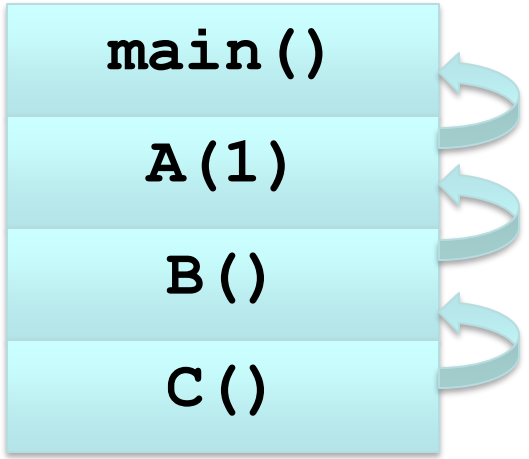
```
void A(int tmp) {
   if (tmp == 2) return;
   else B();
}

void B(){
   C();
}

void C(){
   A(2);
}

main(){A(1);}
```

- Stack



Stack

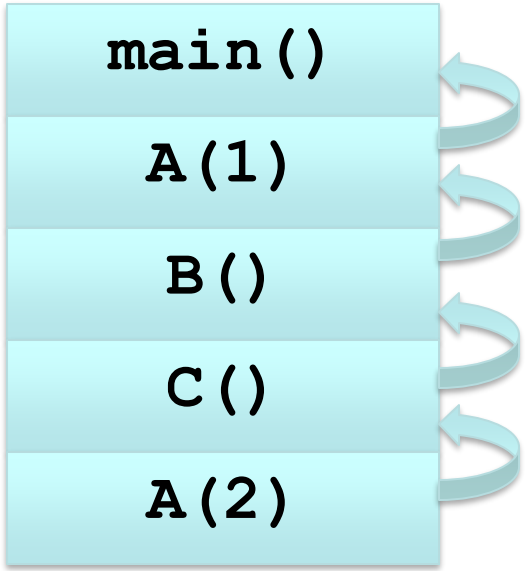
```
void A(int tmp) {
   if (tmp == 2) return;
   else B();
}

void B(){
   C();
}

void C(){
   A(2);
}

main(){A(1);}
```

- Stack



Stack

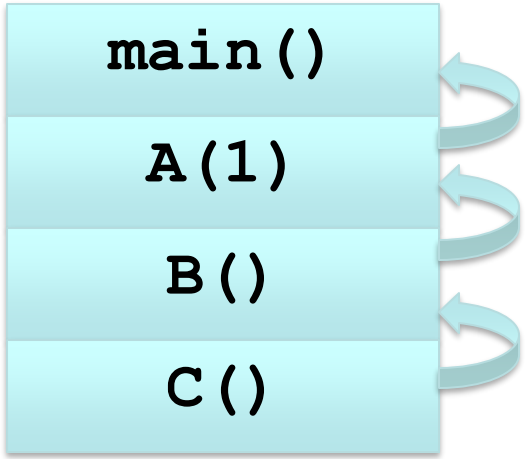
```
void A(int tmp) {
   if (tmp == 2) return;
   else B();
}

void B(){
   C();
}

void C(){
   A(2);
}

main(){A(1);}
```

- Stack



Stack

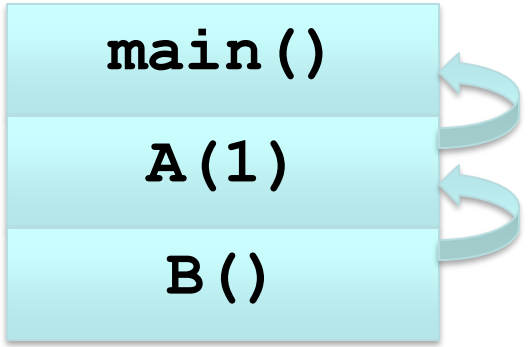
```
void A(int tmp) {
   if (tmp == 2) return;
   else B();
}

void B(){
   C();
}

void C(){
   A(2);
}

main(){A(1);}
```

- Stack



Stack

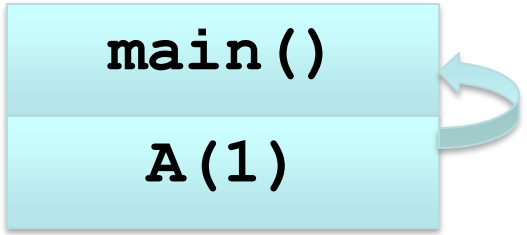
```
void A(int tmp) {
  if (tmp == 2) return;
  else B();
}

void B(){
  C();
}

void C(){
  A(2);
}

main(){A(1);}
```

- Stack



Stack

```
void A(int tmp) {
  if (tmp == 2) return;
  else B();
}

void B(){
  C();
}

void C(){
  A(2);
}

main(){A(1);}
```

- Stack

Kernel Stack

- When executing a **trap**, CPU makes sure to save enough of the caller's registers in order to be able to return correctly when the OS executes the **return-from-trap** instruction
- On x86, for example, **trap** will **push** the program counter, flags, and a few other registers onto a per-process **kernel stack**; **return-from-trap** will **pop** these values off the stack and resume execution of the user-mode program

Trap Table (Interrupt Vector)

- To execute a system call, **system call** library function executes **trap** instruction, which, **simultaneously**, jumps into kernel and raises privilege level to kernel mode
- How does the **trap** instruction know which **code** ("real" system call function) to run inside the kernel?
  - a calling user process can't specify an "address" to jump to (like making a function call); doing so would allow programs to jump anywhere into the kernel which is a **Very Bad Idea**
- kernel must control what code executes **upon a trap**
  - set up a **trap table** at **boot time** when kernel executes in kernel mode
    - at boot time, kernel tells hardware what code to run when certain **exceptional events** occur [e.g., hard-disk interrupt, keyboard interrupt, timer interrupt, or when a program makes a system call] → kernel informs hardware of the **locations** of trap handlers


```
3200 // x86 trap and interrupt constants.
3201
3202 // Processor-defined:
3203 #define T_DIVIDE         0      // divide error
3204 #define T_DEBUG          1      // debug exception
3205 #define T_NMI            2      // non-maskable interrupt
3206 #define T_BRKPT          3      // breakpoint
3207 #define T_OFLOW          4      // overflow
3208 #define T_BOUND          5      // bounds check
3209 #define T_ILLOP          6      // illegal opcode
3210 #define T_DEVICE         7      // device not available
3211 #define T_DBLFLT         8      // double fault
3212 // #define T_COPROC      9      // reserved (not used since 486)
3213 #define T_TSS           10      // invalid task switch segment
3214 #define T_SEGNP         11      // segment not present
3215 #define T_STACK         12      // stack exception
3216 #define T_GPFLT         13      // general protection fault
3217 #define T_PGFLT         14      // page fault
3218 // #define T_RES        15      // reserved
3219 #define T_FPERR         16      // floating point error
3220 #define T_ALIGN         17      // aligment check
3221 #define T_MCHK          18      // machine check
3222 #define T_SIMDERR       19      // SIMD floating point error
3223
3224 // These are arbitrarily chosen, but with care not to overlap
3225 // processor defined exceptions or interrupt vectors.
3226 #define T_SYSCALL       64      // system call
3227 #define T_DEFAULT      500      // catchall
3228
3229 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3230
3231 #define IRQ_TIMER        0
3232 #define IRQ_KBD          1
3233 #define IRQ_COM1         4
3234 #define IRQ_IDE         14
3235 #define IRQ_ERROR       19
3236 #define IRQ_SPURIOUS    31
```



```
3500 // System call numbers
3501 #define SYS_fork    1
3502 #define SYS_exit    2
3503 #define SYS_wait    3
3504 #define SYS_pipe    4
3505 #define SYS_read    5
3506 #define SYS_kill    6
3507 #define SYS_exec    7
3508 #define SYS_fstat   8
3509 #define SYS_chdir   9
3510 #define SYS_dup    10
3511 #define SYS_getpid 11
3512 #define SYS_sbrk   12
3513 #define SYS_sleep  13
3514 #define SYS_uptime 14
3515 #define SYS_open   15
3516 #define SYS_write  16
3517 #define SYS_mknod  17
3518 #define SYS_unlink 18
3519 #define SYS_link   19
3520 #define SYS_mkdir  20
3521 #define SYS_close  21
```

trap (software interrupt)
vs.
(hardware) **interrupt**

synchronous

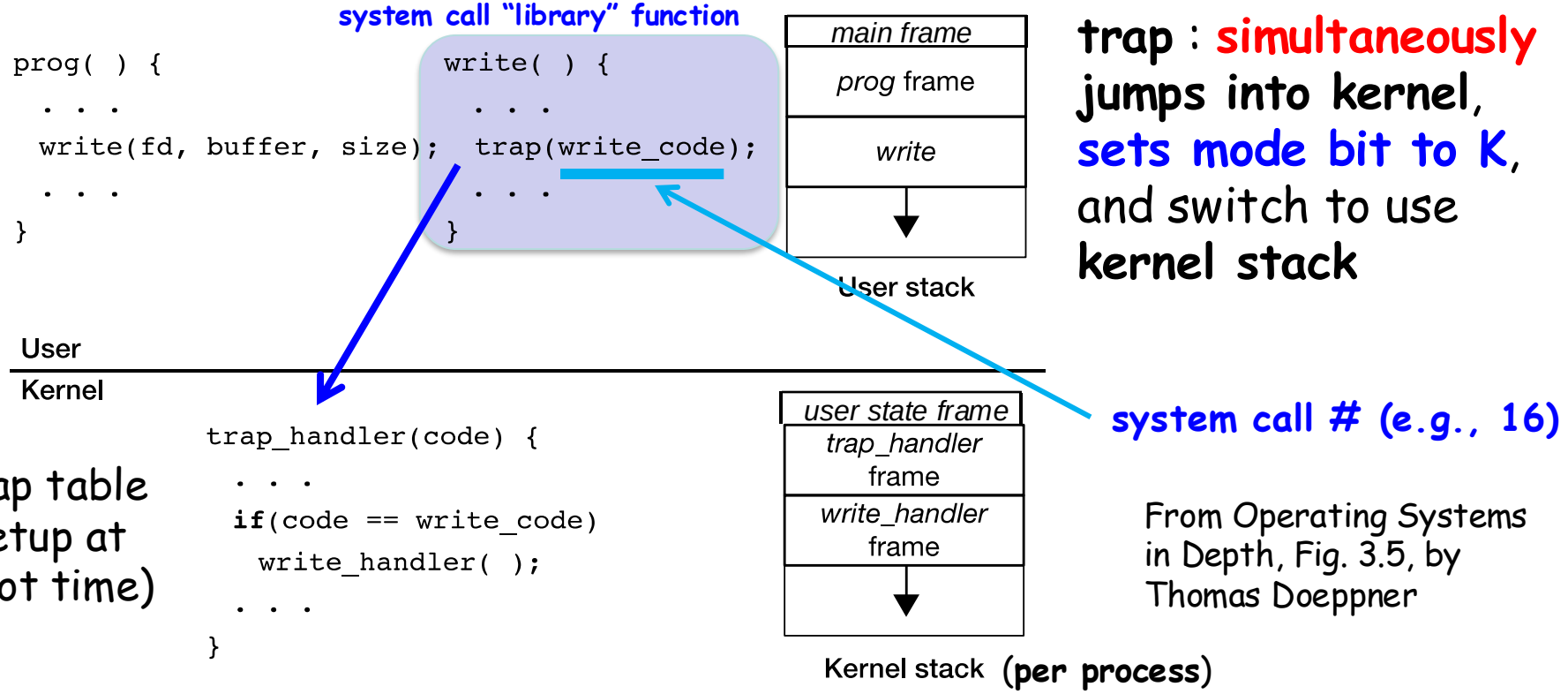
asynchronous

```
#define T_SYSCALL 64
#define SYS_write 16
```

System Calls

```
.globl write;
write:
  movl $16, %eax;
  int $64;
  ret
```

- **At boot time**, OS uses special instructions to tell the hardware where the **trap table** resides in memory
- "**trap**" into the kernel [`int $64`] and "**return-from-trap**" back to user-mode programs [`ret`]



trap : **simultaneously jumps into kernel, sets mode bit to K**, and switch to use **kernel stack**

From Operating Systems in Depth, Fig. 3.5, by Thomas Doeppner

Why System Calls Look Like Function Calls?

- From programmer's (or user's) perspective, it is a (normal) function call, but hidden inside that function call is the **trap** instruction (**`int $64`**)
- E.g., when you call `open()`, you are calling a function in the C library
- C library uses an agreed-upon **calling convention** with kernel to put the arguments to `open()` in well-known locations (e.g., on the stack, or in specific registers), puts the **system-call number** into a well-known location (onto the stack or a register [**`%eax`**]), and then executes **trap**
- The code in the library after the trap unpacks return values and returns control to the program that issued the "system call"
- The parts of the C library that make system calls are **hand-coded in assembly**, as they need to carefully follow convention in order to process arguments and return values correctly, as well as execute the hardware-specific trap instruction
- Now you know why you personally don't have to write assembly code to trap into an OS; somebody has already written that assembly for you.

```
void main(int argc, *argv[])
{
    int pid = atoi(argv[1]);
    kill_proc(pid);
}

void kill_proc(int pid)
{
  prog(pid);
}

void prog(int pid)
{
  kill(pid, 9);
}
```

mykill.c

```
printf(), scanf(),
fgets(), ……
```

```
strcpy(), strcmp(),
strlen(), ……
```

```
fork(), write(),
kill(), read(), …
```

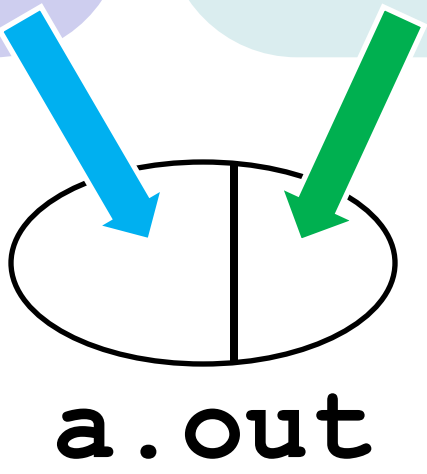
system call "library functions"

```
malloc(), free(),
calloc(), ……
```

• • •

C standard library (libc or glibc)

```
$ gcc mykill.c
```



C standard library is **implicitly** linked with `mykill.o` to create `a.out`

```
void main(int argc, *argv[])
{
    int pid = atoi(argv[1]);
    kill_proc(pid);
}

void kill_proc(int pid)
{
  prog(pid);
}

void prog(int pid)
{
  kill(pid, 9);
}
```

mykill.c

```
$ gcc mykill.c \
usys.S ulib.c \
printf.c umalloc.c
```

```
.globl kill;
kill:
  movl $6, %eax;
  int $64;
  ret
```

```
.globl read;
read:
  movl $5, %eax;
  int $64;
  ret
```

```
.globl write;
write:
  movl $16, %eax;
  int $64;
  ret
```

```
.globl exec;
exec:
  movl $7, %eax;
  int $64;
  ret
```

●●● usys.S system call library functions

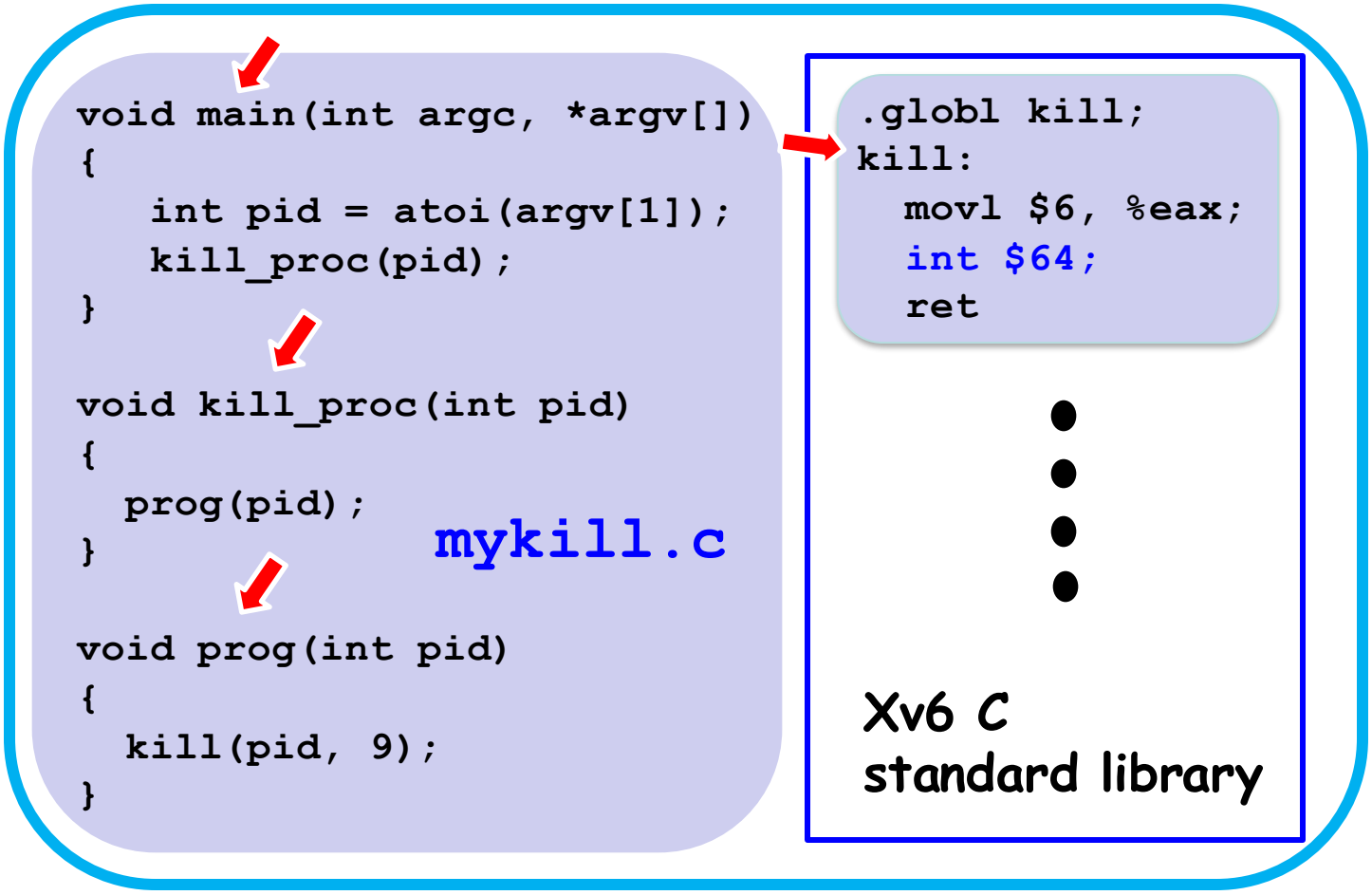
strcpy(), strcmp(), strlen(), atoi(), ... ulib.c

putc(), printint(), printf() printf.c

malloc(), free(), morecore() umalloc.c

Xv6 C standard library

C standard library (libc or glibc)



User stack

```
main: argc,
argv, pid
```

```
kill_proc:
pid
```

```
prog: pid
```

```
kill:
```

```
$ gcc mykill.c \
usys.S ulib.c \
printf.c umalloc.c
$ ./a.out
```

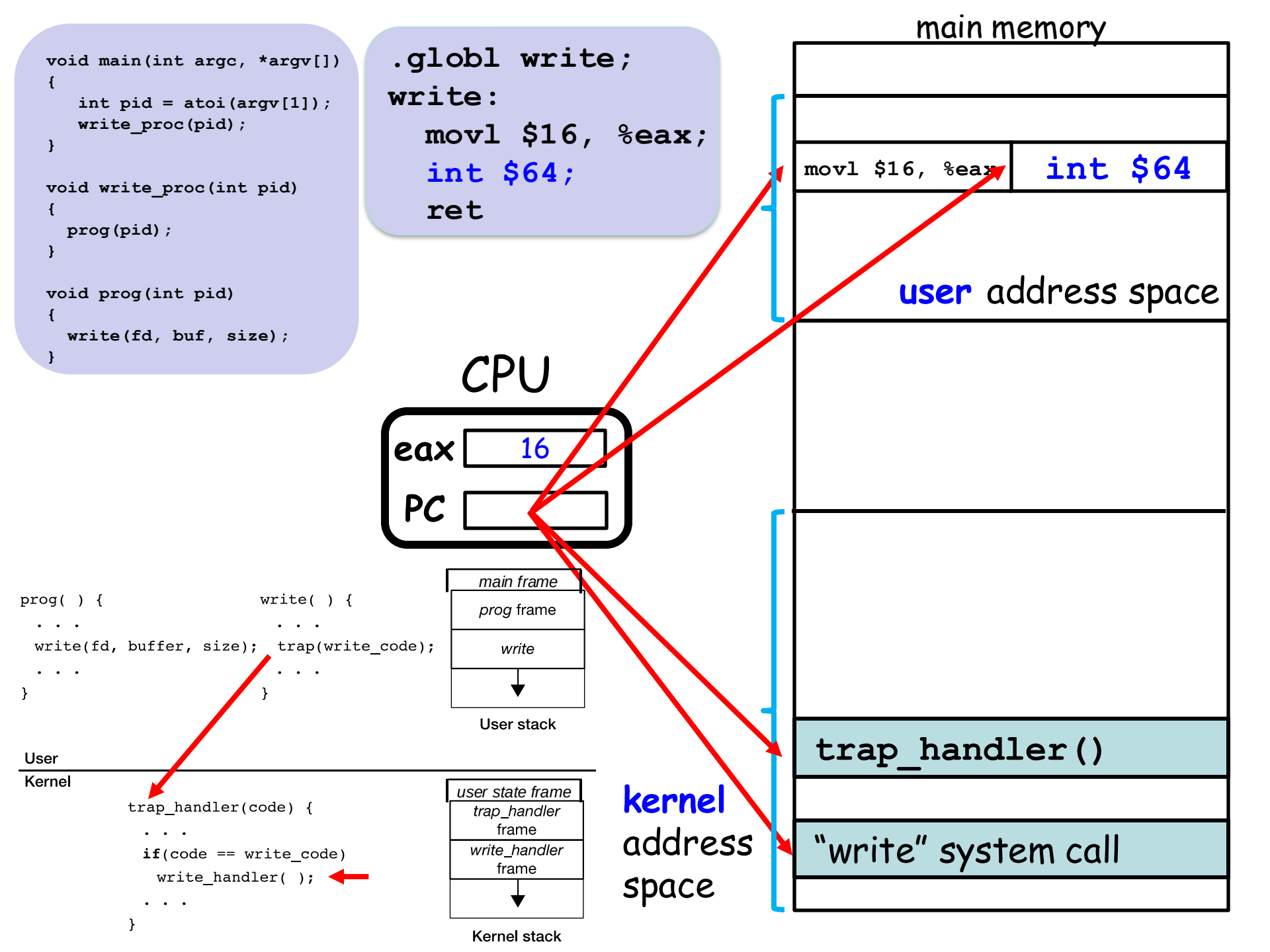
Trap [`int $64`] loads PC with a **predefined memory address** in the kernel address space

```
void main(int argc, *argv[])
{
    int pid = atoi(argv[1]);
    write_proc(pid);
}

void write_proc(int pid)
{
  prog(pid);
}

void prog(int pid)
{
  write(fd, buf, size);
}
```

```
.globl write;
write:
  movl $16, %eax;
  int $64;
  ret
```



```
3200 // x86 trap and interrupt constants.
3201
3202 // Processor-defined:
3203 #define T_DIVIDE         0      // divide error
3204 #define T_DEBUG          1      // debug exception
3205 #define T_NMI            2      // non-maskable interrupt
3206 #define T_BRKPT          3      // breakpoint
3207 #define T_OFLOW          4      // overflow
3208 #define T_BOUND          5      // bounds check
3209 #define T_ILLOP          6      // illegal opcode
3210 #define T_DEVICE         7      // device not available
3211 #define T_DBLFLT         8      // double fault
3212 // #define T_COPROC      9      // reserved (not used since 486)
3213 #define T_TSS           10      // invalid task switch segment
3214 #define T_SEGNP         11      // segment not present
3215 #define T_STACK         12      // stack exception
3216 #define T_GPFLT         13      // general protection fault
3217 #define T_PGFLT         14      // page fault
3218 // #define T_RES        15      // reserved
3219 #define T_FPERR         16      // floating point error
3220 #define T_ALIGN         17      // aligment check
3221 #define T_MCHK          18      // machine check
3222 #define T_SIMDERR       19      // SIMD floating point error
3223
3224 // These are arbitrarily chosen, but with care not to overlap
3225 // processor defined exceptions or interrupt vectors.
3226 #define T_SYSCALL       64      // system call
3227 #define T_DEFAULT      500      // catchall
3228
3229 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3230
3231 #define IRQ_TIMER        0
3232 #define IRQ_KBD          1
3233 #define IRQ_COM1         4
3234 #define IRQ_IDE         14
3235 #define IRQ_ERROR       19
3236 #define IRQ_SPURIOUS    31
```



```
3500 // System call numbers
3501 #define SYS_fork    1
3502 #define SYS_exit    2
3503 #define SYS_wait    3
3504 #define SYS_pipe    4
3505 #define SYS_read    5
3506 #define SYS_kill    6
3507 #define SYS_exec    7
3508 #define SYS_fstat   8
3509 #define SYS_chdir   9
3510 #define SYS_dup    10
3511 #define SYS_getpid 11
3512 #define SYS_sbrk   12
3513 #define SYS_sleep  13
3514 #define SYS_uptime 14
3515 #define SYS_open   15
3516 #define SYS_write  16
3517 #define SYS_mknod  17
3518 #define SYS_unlink 18
3519 #define SYS_link   19
3520 #define SYS_mkdir  20
3521 #define SYS_close  21
```

```
.globl write;
write:
  movl $16, %eax;
  int $64; // trap
  ret
```

System Call

```
.globl write;
write:
  movl $16, %eax;
  int $64;
  ret
```

- A **system-call number** is assigned to each system call
- The "**user-facing" system call library function** is responsible for placing the desired system-call number in a register (**%eax**) or at a specified location on the stack, and then "**traps" into kernel** (via **`int $64`**)
- Kernel, when handling the system call inside the **trap handler**, examines this number, ensures it is valid, and, if it is, executes the corresponding **code of the system call**

System Call

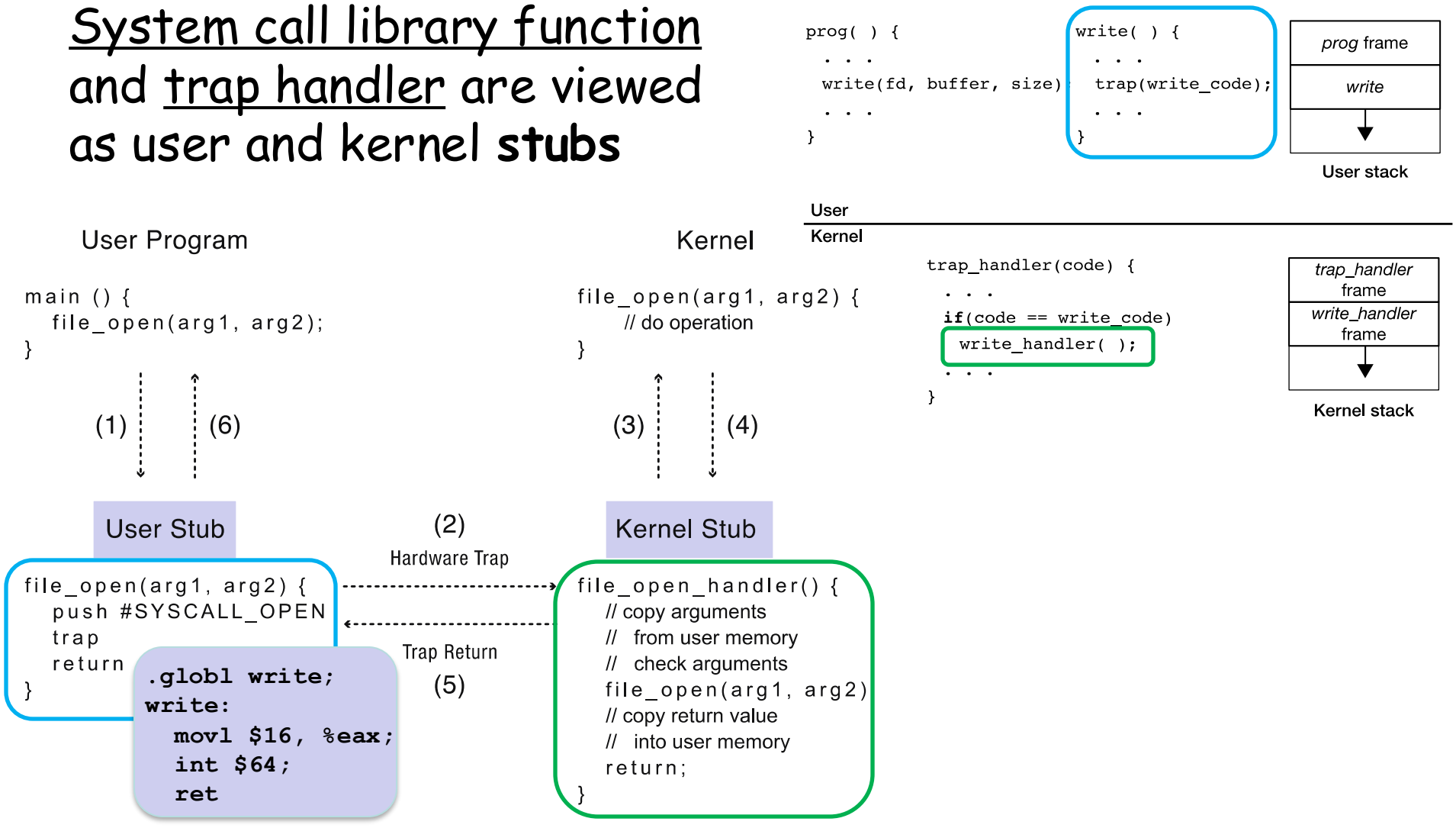
```
.globl write;
write:
  movl $16, %eax;
  int $64;
  ret
```

- Because the **trap table** is set up by a trusted entity (the kernel itself) and cannot be altered by the user, the operating system (kernel) is confident that program control goes to only well-defined points in the kernel
- This level of **indirection** serves as a form of **protection**; user code cannot specify an exact address to jump to, but rather must request a particular service via **system call number**

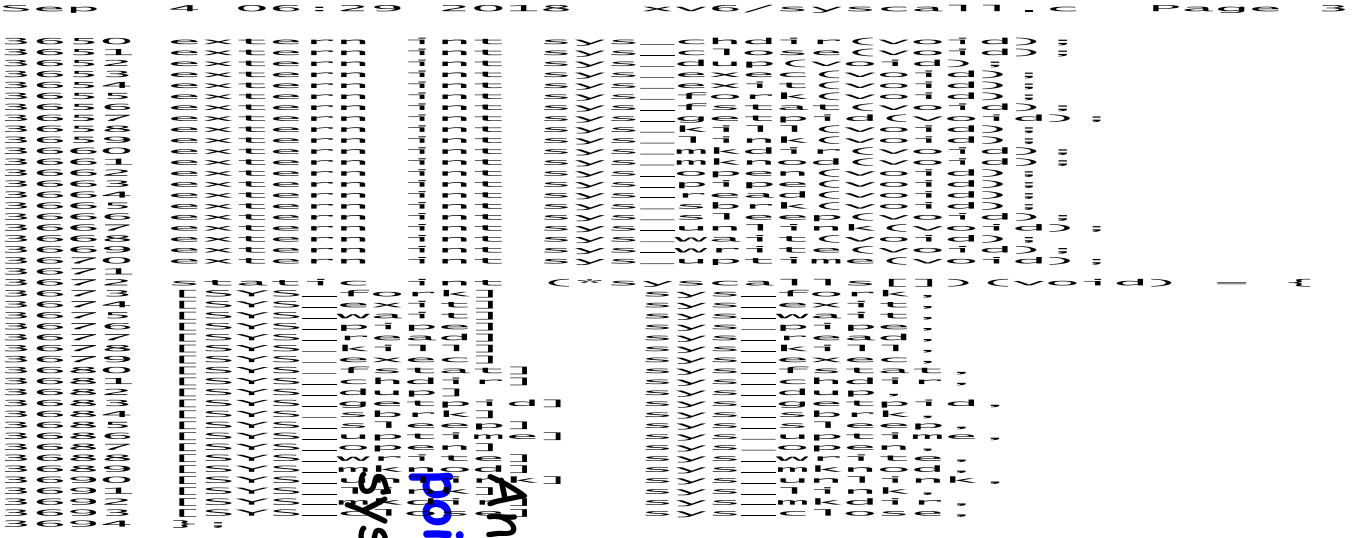
Protected Control Transfer

User/Kernel Stubs

System call library function and trap handler are viewed as user and kernel **stubs**



```
3500 // System call numbers
3501 #define SYS_fork    1
3502 #define SYS_exit    2
3503 #define SYS_wait    3
3504 #define SYS_pipe    4
3505 #define SYS_read    5
3506 #define SYS_kill    6
3507 #define SYS_exec    7
3508 #define SYS_fstat   8
3509 #define SYS_chdir   9
3510 #define SYS_dup    10
3511 #define SYS_getpid 11
3512 #define SYS_sbrk   12
3513 #define SYS_sleep  13
3514 #define SYS_uptime 14
3515 #define SYS_open   15
3516 #define SYS_write  16
3517 #define SYS_mknod  17
3518 #define SYS_unlink 18
3519 #define SYS_link   19
3520 #define SYS_mkdir  20
3521 #define SYS_close  21
```

An array of **function pointers** to (actual) system call functions

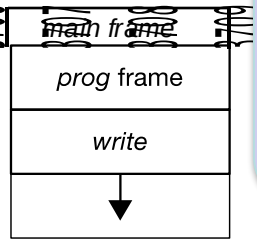
system call number saved in %eax

```
3672 static int (*syscalls[])(void) = {
3673 [SYS_fork]    sys_fork,
3674 [SYS_exit]    sys_exit,
3675 [SYS_wait]    sys_wait,
3676 [SYS_pipe]    sys_pipe,
3677 [SYS_read]    sys_read,
3678 [SYS_kill]    sys_kill,
3679 [SYS_exec]    sys_exec,
3680 [SYS_fstat]   sys_fstat,
3681 [SYS_chdir]   sys_chdir,
3682 [SYS_dup]     sys_dup,
3683 [SYS_getpid]  sys_getpid,
3684 [SYS_sbrk]    sys_sbrk,
3685 [SYS_sleep]   sys_sleep,
3686 [SYS_uptime]  sys_uptime,
3687 [SYS_open]    sys_open,
3688 [SYS_write]   sys_write,
3689 [SYS_mknod]   sys_mknod,
3690 [SYS_unlink]  sys_unlink,
3691 [SYS_link]    sys_link,
3692 [SYS_mkdir]   sys_mkdir,
3693 [SYS_close]   sys_close,
3694 };
```

```
3701 syscall(void)
3702 {
3703   int num;
3704   struct proc *curproc = myproc();
3705
3706   num = curproc->tf->eax;
3707   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3708     curproc->tf->eax = syscalls[num]();
3709   } else {
3710     cprintf("%d %s: unknown sys call %d\n",
3711             curproc->pid, curproc->name, num);
3712     curproc->tf->eax = -1;
```

```
prog( ) {
  . . .
  write(fd, buffer, size);
  . . .
}
```

```
write( ) {
  . . .
  trap(write_code);
  . . .
}
```



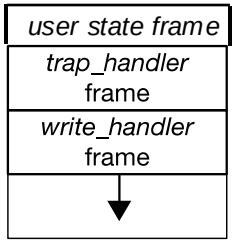
User stack

```
.globl write;
write:
  movl $16, %eax;
  int $64;
  ret
```

User

Kernel

```
trap_handler(code) {
  . . .
  if(code == write_code)
    write_handler( );
  . . .
}
```



Kernel stack

```
3600 // Fetch the nth 32-bit system call argument.
3601 int
3602 argint(int n, int *ip)
3603 {
3604   return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
3605 }
3606
3607 // Fetch the nth word-sized system call argument as a pointer
3608 // to a block of memory of size bytes.  Check that the pointer
3609 // lies within the process address space.
3610 int
3611 argptr(int n, char **pp, int size)
3612 {
3613   int i;
3614   struct proc *curproc = myproc();
3615
3616   if(argint(n, &i) < 0)
3617     return -1;
3618   if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
3619     return -1;
3620   *pp = (char*)i;
3621   return 0;
3622 }
3623
3624 // Fetch the nth word-sized system call argument as a string pointer.
3625 // Check that the pointer is valid and the string is nul-terminated.
3626 // (There is no shared writable memory, so the string can't change
3627 // between this check and being used by the kernel.)
3628 int
3629 argstr(int n, char **pp)
3630 {
3631   int addr;
3632   if(argint(n, &addr) < 0)
3633     return -1;
3634   return fetchstr(addr, pp);
3635 }
```



```
6150 int
6151 sys_write(void)
6152 {
6153   struct file *f;
6154   int n;
6155   char *p;
6156
6157   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6158     return -1;
6159   return filewrite(f, p, n);
6160 }
```



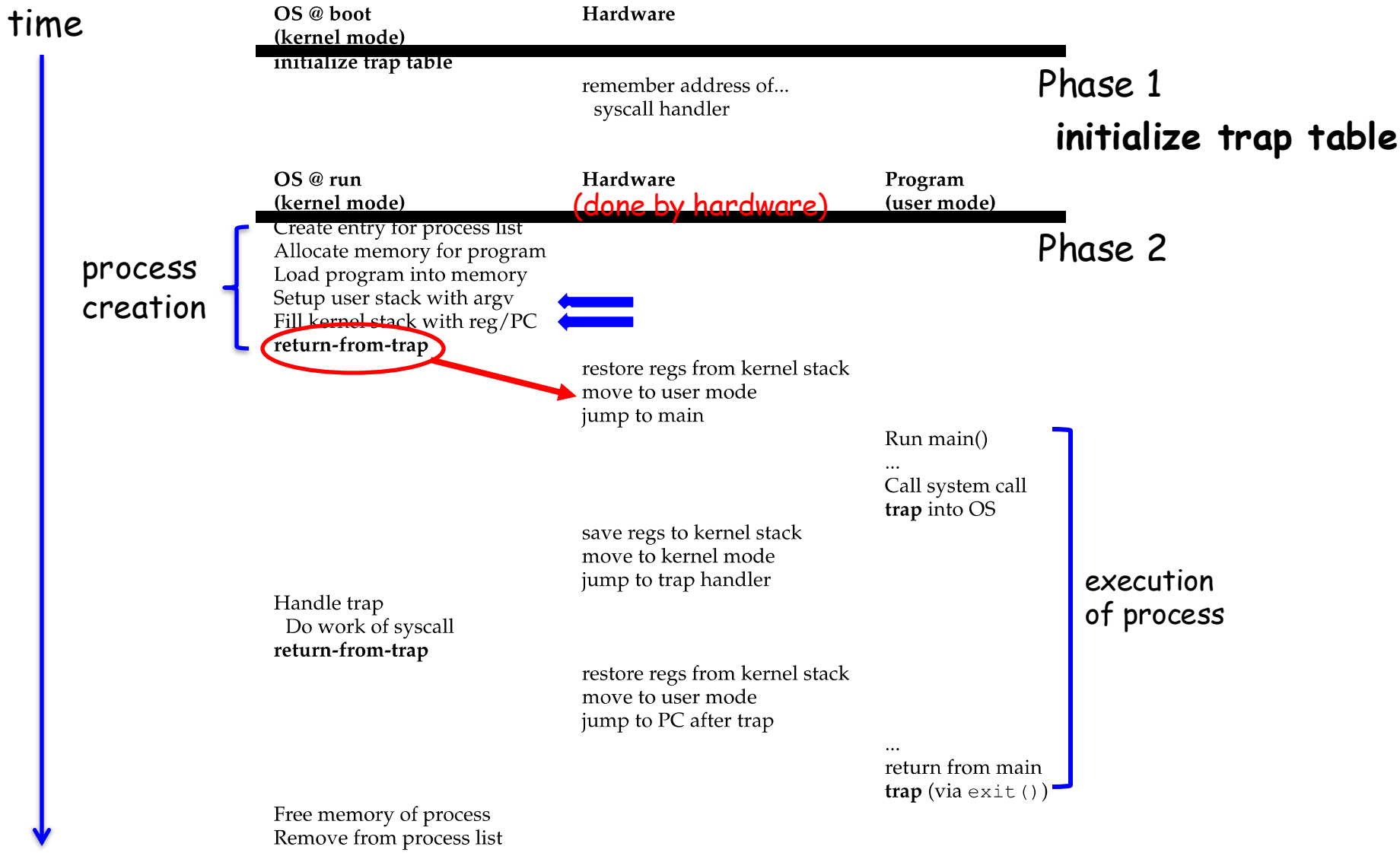
```
6000 // Write to file f.
6001 int
6002 filewrite(struct file *f, char *addr, int n)
6003 {
6004   int r;
6005
6006   if(f->writable == 0)
6007     return -1;
6008   if(f->type == FD_PIPE)
6009     return pipewrite(f->pipe, addr, n);
6010   if(f->type == FD_INODE){
6011     // write a few blocks at a time to avoid exceeding
6012     // the maximum log transaction size, including
6013     // i-node, indirect block, allocation blocks,
6014     // and 2 blocks of slop for non-aligned writes.
6015     // this really belongs lower down, since writei()
6016     // might be writing a device like the console.
6017     int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
6018     int i = 0;
6019     while(i < n){
6020       int n1 = n - i;
6021       if(n1 > max)
6022         n1 = max;
6023
6024       begin_op();
6025       ilock(f->ip);
6026       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6027         f->off += r;
6028       iunlock(f->ip);
6029       end_op();
6030
6031       if(r < 0)
6032         break;
6033       if(r != n1)
6034         panic("short filewrite");
6035       i += r;
6036     }
6037     return i == n ? n : -1;
6038   }
6039   panic("filewrite");
6040 }
```



```
6050 //
6051 // File-system system calls.
6052 // Mostly argument checking, since we don't trust
6053 // user code, and calls into file.c and fs.c.
6054 //
6055
6056 #include "types.h"
6057 #include "defs.h"
6058 #include "param.h"
6059 #include "stat.h"
6060 #include "mmu.h"
6061 #include "proc.h"
6062 #include "fs.h"
6063 #include "spinlock.h"
6064 #include "sleeplock.h"
6065 #include "file.h"
6066 #include "fcntl.h"
6067
6068 // Fetch the nth word-sized system call argument as a file descriptor
6069 // and return both the descriptor and the corresponding struct file.
6070 static int
6071 argfd(int n, int *pfd, struct file **pf)
6072 {
6073   int fd;
6074   struct file *f;
6075
6076   if(argint(n, &fd) < 0)
6077     return -1;
6078   if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
6079     return -1;
6080   if(pfd)
6081     *pfd = fd;
6082   if(pf)
6083     *pf = f;
6084   return 0;
6085 }
6086
6087
6088
6089
6090
```

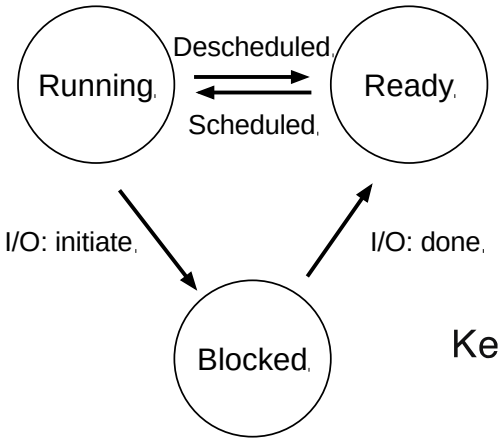
"read" "write()" system call

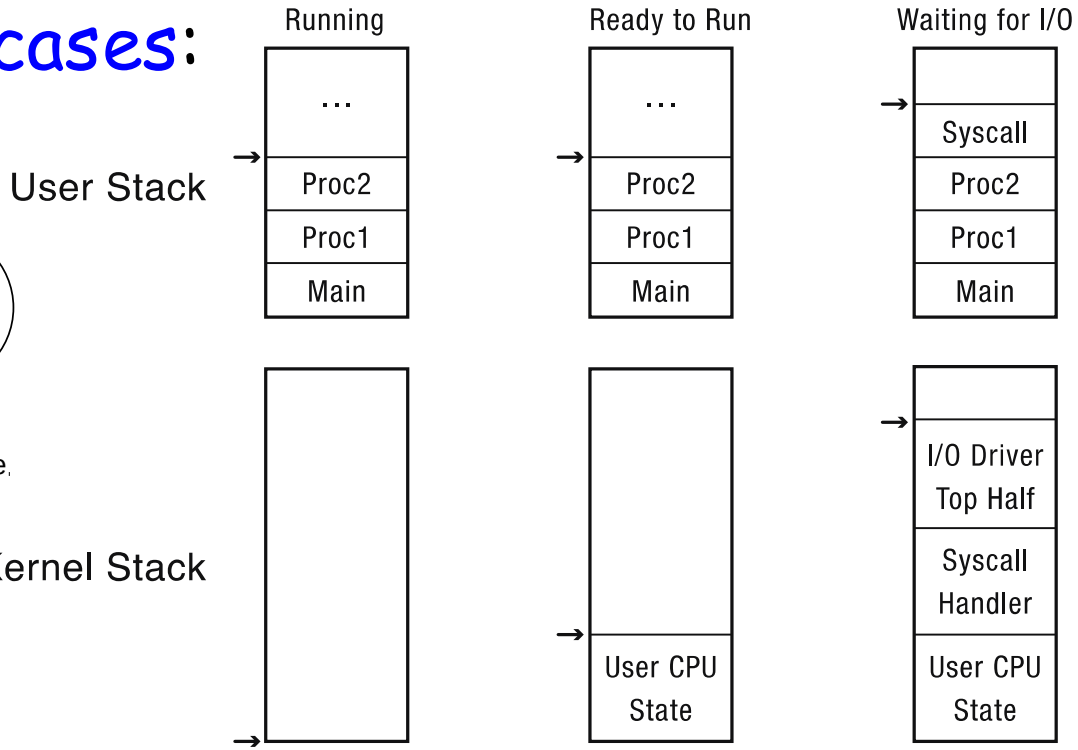
Limited Direct Execution



Two Stacks Per Process

Three cases:





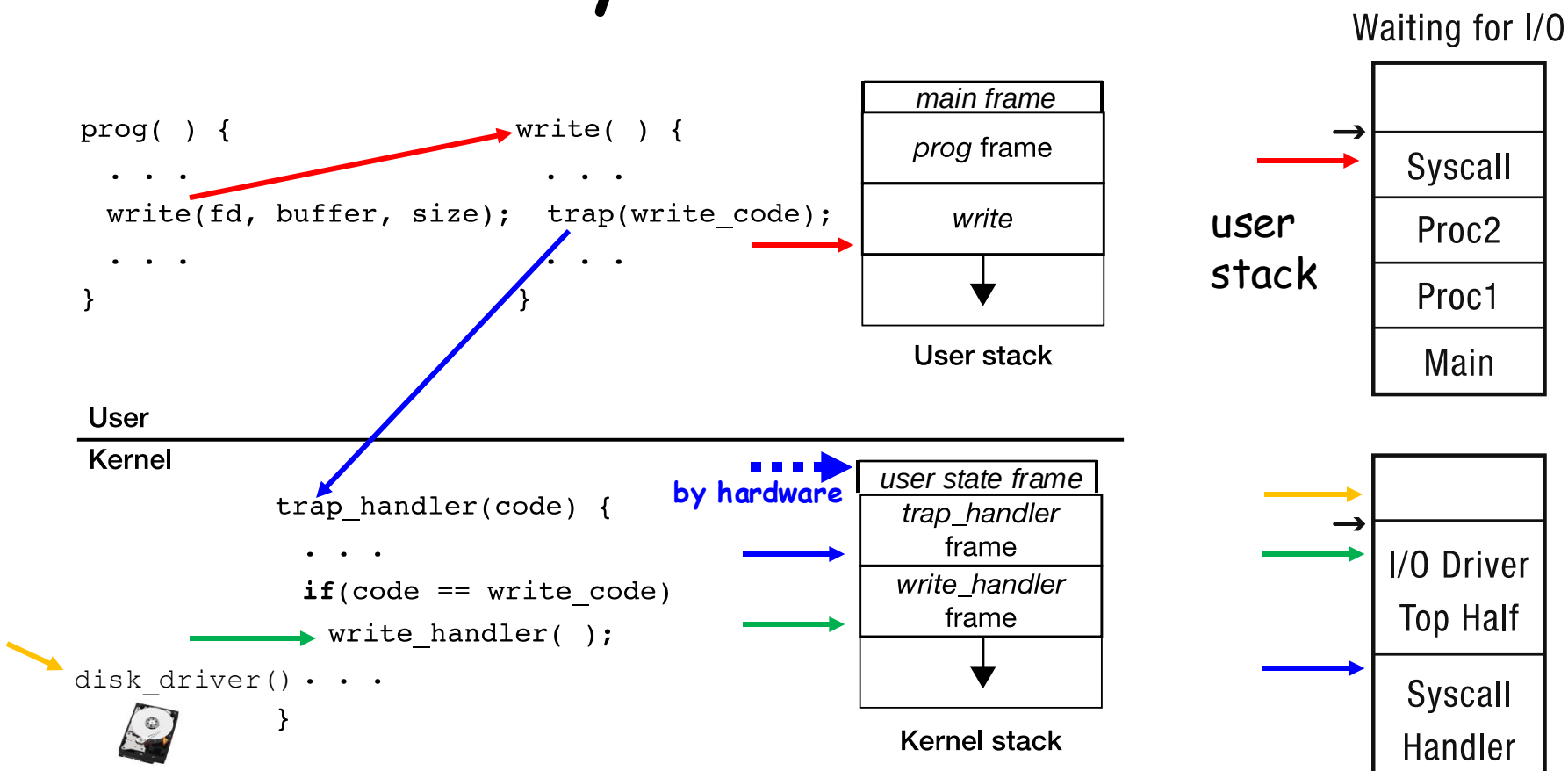
Three cases:

1. When a process is **running in user mode**, its kernel stack is **empty**
2. When a process has been preempted (by timer) [**ready but not running**], its kernel stack contains the **user-level processor state** at the point when the user process was interrupted (i.e., the state to be restored when the process is resumed)
3. When a process is **inside a system call** **waiting for "hardware" I/O to complete**
   - its **kernel stack** contains the context to be resumed when H/W I/O completes
   - its **user stack** contains the context to be resumed when the system call returns

Implementation of System Calls

- Setup **trap table (interrupt vector)** at boot time
  - tell hardware what code to run when **exceptional events** occur, e.g., hard-disk interrupt, keyboard interrupt, **system call**, etc.
- **trap** instruction
  - **simultaneously** jump into kernel; raise privilege level to kernel mode; push (save) caller's registers (e.g., PC, SP, etc.) onto per-process **kernel** stack
- **return-from-trap** instruction
  - **simultaneously** return to calling program; reduce privilege level to user mode; pop registers (e.g., PC, SP, etc.) off per-process **kernel** stack

System Call



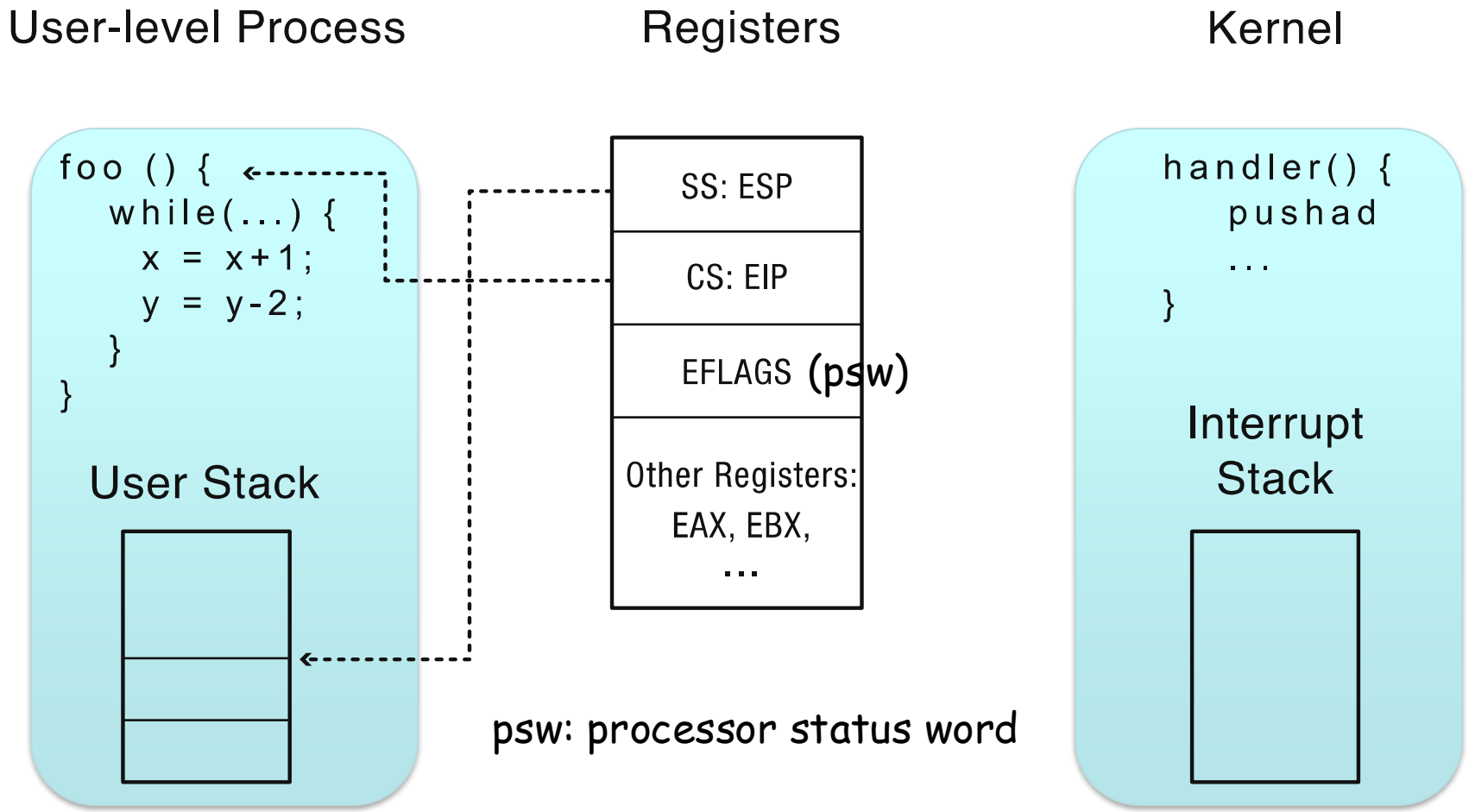
When a process is **inside a system call waiting for I/O**

- its **kernel stack** contains the context to be resumed when the "hardware" I/O completes
- its **user stack** contains the context to be resumed when the system call returns

```
2316 // Saved registers for kernel context switches.
2317 // Don't need to save all the segment registers (%cs, etc),
2318 // because they are constant across kernel contexts.
2319 // Don't need to save %eax, %ecx, %edx, because the
2320 // x86 convention is that the caller has saved them.
2321 // Contexts are stored at the bottom of the stack they
2322 // describe; the stack pointer is the address of the context.
2323 // The layout of the context matches the layout of the stack in swtch.S
2324 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2325 // but it is on the stack and allocproc() manipulates it.
2326 struct context {
2327   uint edi;
2328   uint esi;
2329   uint ebx;
2330   uint ebp;
2331   uint eip;
2332 };
2333
2334 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNING, ZOMBIE };
2335
2336 // Per-process state
2337 struct proc {
2338   uint sz;                     // Size of process memory (bytes)
2339   pde_t* pgdir;                // Page table
2340   char *kstack;                // Bottom of kernel stack for this process
2341   enum procstate state;        // Process state
2342   int pid;                     // Process ID
2343   struct proc *parent;         // Parent process
2344   struct trapframe *tf;        // Trap frame for current syscall
2345   struct context *context;     // swtch() here to run process
2346   void *chan;                  // If non-zero, sleeping on chan
2347   int killed;                  // If non-zero, have been killed
2348   struct file *ofile[NOFILE];  // Open files
2349   struct inode *cwd;           // Current directory
```

PCB

When Xv6 handles a system call, the current process's `proc->tf` is set to the interrupt trap frame, which contains the register set that will get restored to the processor when the system call returns. This way, system calls can interact with the register set and give return values back to user mode.



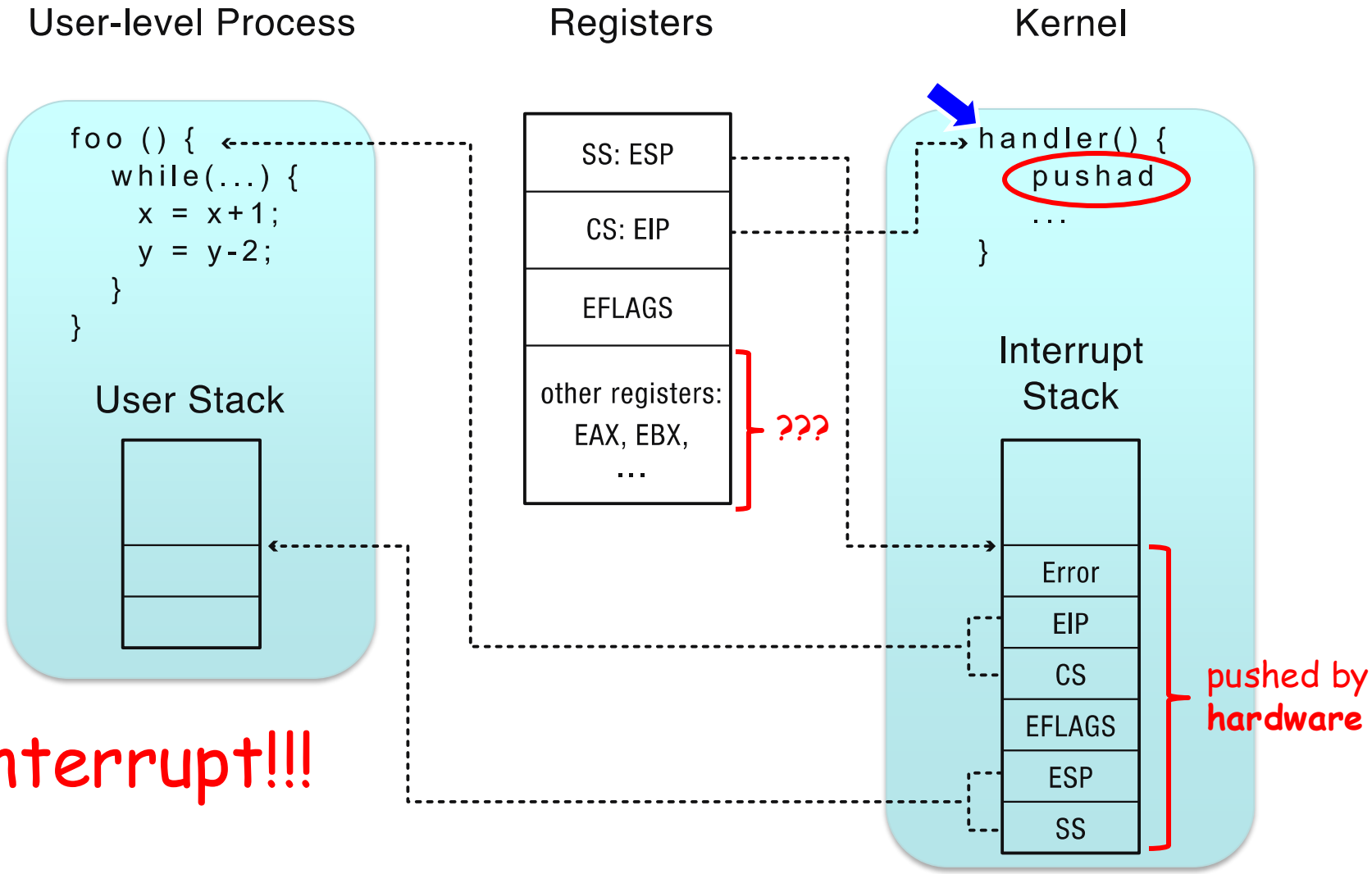
psw: processor status word

User process is running:

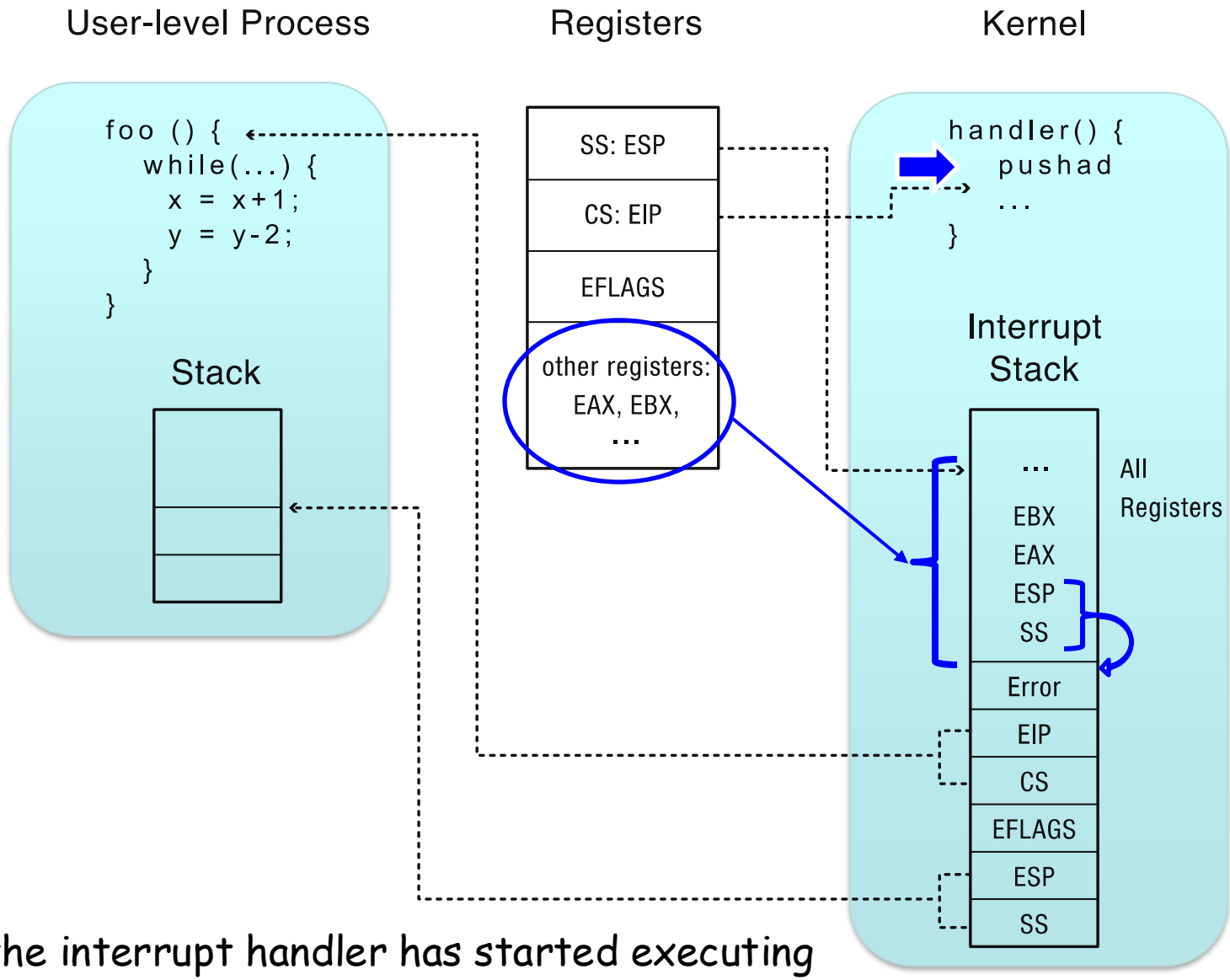
State, **before** an interrupt handler is invoked

- SS: stack segment; ESP: stack pointer
- CS: code segment; EIP: instruction pointer (PC)
- EIP and ESP refer to locations in **user process**
- interrupt (kernel) stack is **empty**

x86 is "segmented" so each segment (code, data, stack) has two parts: **base** + **offset**



- **x86 hardware** pushes the **user context** on the interrupt (kernel) stack and changes the program counter and stack pointer to locations in kernel memory (and push Error code if necessary)
- State, **after** x86 **hardware** has "**jumped to**" the interrupt handler



State, **after** the interrupt handler has started executing

- handler first saves (by **pushad**) the current state of the processor (general) registers, **since it may overwrite them**
  - saving stack pointer **twice**: first, the **user** stack pointer, then the **kernel** stack pointer

```
#!/usr/bin/perl -w

# Generate vectors.S, the trap/interrupt entry points.
# There has to be one entry point per interrupt number
# since otherwise there's no way for trap() to discover
# the interrupt number.

print "# generated by vectors.pl - do not edit\n";
print "# handlers\n";
print ".globl alltraps\n";
for(my $i=0; $i<256; $i++){
    print ".globl vector$i\n";
    if($i<10 && $i != 8 || $i >= 14 || $i == 17){
        print "vector$i:\n  pushl \$0\n";
    }
    print "  pushl \$$i\n";
    print "  jmp alltraps\n";
}

print "\n# vector table\n";
print ".data\n";
print ".globl vectors\n";
print "vectors:\n";
for(my $i=0; $i<256; $i++){
    print "  .long vector$i\n";
}

#  sample output:
#      # handlers
#      .globl alltraps
#      .globl vector0
#      vector0:
#        pushl $0
#        pushl $0
#        jmp alltraps
#      ...
#
#      # vector table
#      .data
#      .globl vectors
#      vectors:
#        .long vector0
#        .long vector1
#        .long vector2
#      ...
```

Sheet 32

```
# generated by vectors.pl - do not edit
# handlers
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
  jmp alltraps
.globl vector2
vector2:
  pushl $0
  pushl $2
  jmp alltraps
.globl vector3
vector3:
  pushl $0
  pushl $3
  jmp alltraps
```

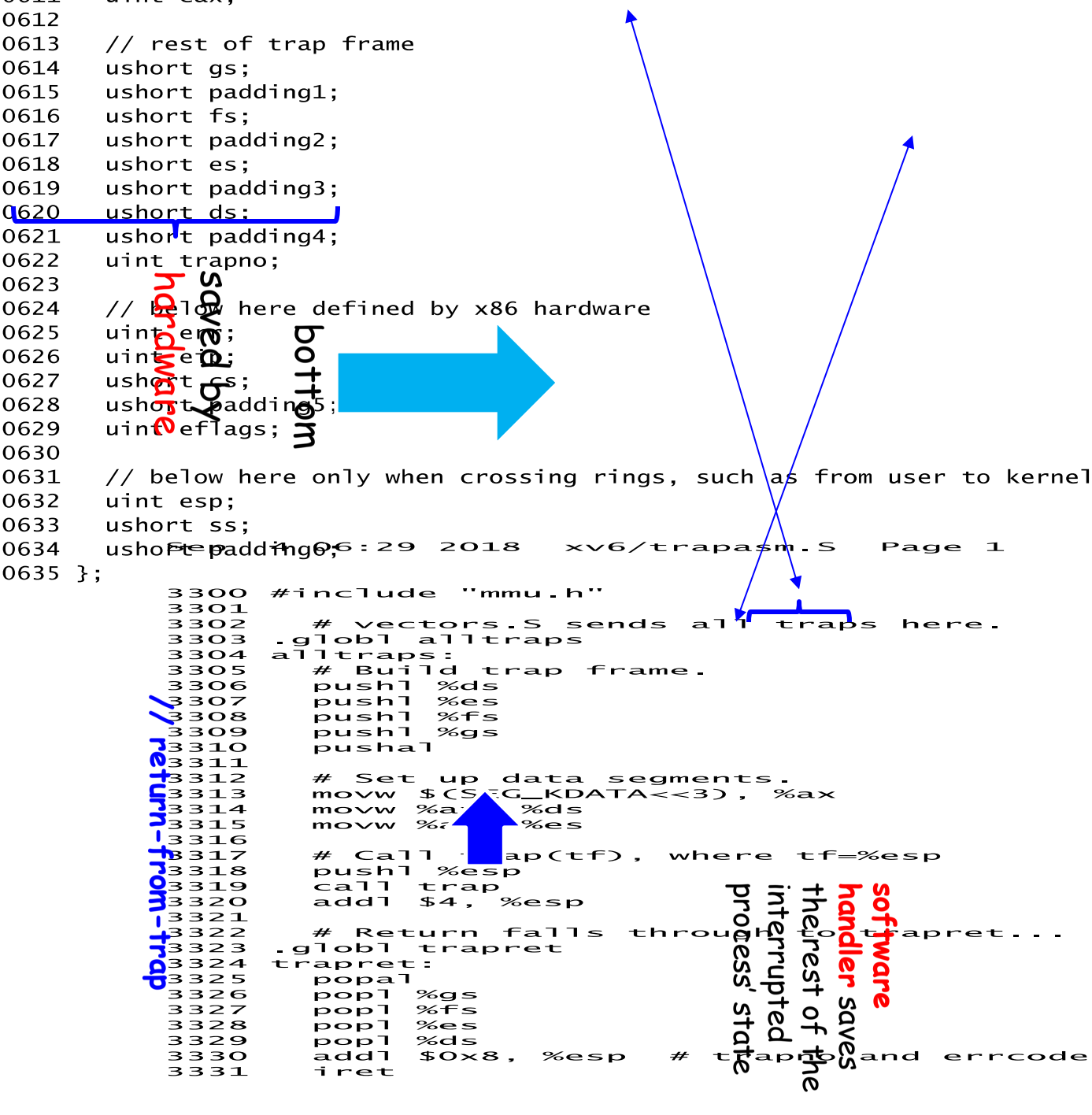


```
3200 // x86 trap and interrupt constants.
3201
3202 // Processor-defined:
3203 #define T_DIVIDE         0      // divide error
3204 #define T_DEBUG          1      // debug exception
3205 #define T_NMI            2      // non-maskable interrupt
3206 #define T_BRKPT          3      // breakpoint
3207 #define T_OFLOW          4      // overflow
3208 #define T_BOUND          5      // bounds check
3209 #define T_ILLOP          6      // illegal opcode
3210 #define T_DEVICE         7      // device not available
3211 #define T_DBLFLT         8      // double fault
3212 // #define T_COPROC      9      // reserved (not used since 486)
3213 #define T_TSS           10      // invalid task switch segment
3214 #define T_SEGNP         11      // segment not present
3215 #define T_STACK         12      // stack exception
3216 #define T_GPFLT         13      // general protection fault
3217 #define T_PGFLT         14      // page fault
3218 // #define T_RES        15      // reserved
3219 #define T_FPERR         16      // floating point error
3220 #define T_ALIGN         17      // aligment check
3221 #define T_MCHK          18      // machine check
3222 #define T_SIMDERR       19      // SIMD floating point error
3223
3224 // These are arbitrarily chosen, but with care not to overlap
3225 // processor defined exceptions or interrupt vectors.
3226 #define T_SYSCALL       64      // system call
3227 #define T_DEFAULT      500      // catchall
3228
3229 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3230
3231 #define IRQ_TIMER        0
3232 #define IRQ_KBD          1
3233 #define IRQ_COM1         4
3234 #define IRQ_IDE         14
3235 #define IRQ_ERROR       19
3236 #define IRQ_SPURIOUS    31
```

```
0600 // Layout of the trap frame built on the stack by the
0601 // hardware and by trapasm.S, and passed to trap().
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // useless & ignored
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
0612
0613   // rest of trap frame
0614   ushort gs;
0615   ushort padding1;
0616   ushort fs;
0617   ushort padding2;
0618   ushort es;
0619   ushort padding3;
0620   ushort ds;
0621   ushort padding4;
0622   uint trapno;
0623
0624   // below here defined by x86 hardware
0625   uint err;
0626   uint eip;
0627   ushort cs;
0628   ushort padding5;
0629   uint eflags;
0630
0631   // below here only when crossing rings, such as from user to kernel
0632   uint esp;
0633   ushort ss;
0634   ushort padding6;
0635 };
```



```
3300 #include "mmu.h"
3301
3302   # vectors.S sends all traps here.
3303 .globl alltraps
3304 alltraps:
3305   # Build trap frame.
3306   pushl %ds
3307   pushl %es
3308   pushl %fs
3309   pushl %gs
3310   pushal
3311
3312   # Set up data segments.
3313   movw $(SEG_KDATA<<3), %ax
3314   movw %ax, %ds
3315   movw %ax, %es
3316
3317   # Call trap(tf), where tf=%esp
3318   pushl %esp
3319   call trap
3320   addl $4, %esp
3321
3322   # Return falls through to trapret...
3323 .globl trapret
3324 trapret:
3325   popal
3326   popl %gs
3327   popl %fs
3328   popl %es
3329   popl %ds
3330   addl $0x8, %esp  # trapno and errcode
3331   iret
```



```
3400 void
3401 trap(struct trapframe *tf)
3402 {
3403   if(tf->trapno == T_SYSCALL){
3404     if(myproc()->killed)
3405       exit();
3406     myproc()->tf = tf;
3407     syscall();
3408     if(myproc()->killed)
3409       exit();
3410     return;
3411   }
3412
3413   switch(tf->trapno){
3414   case T_IRQ0 + IRQ_TIMER:
3415     if(cpuid() == 0){
3416       acquire(&tickslock);
3417       ticks++;
3418       wakeup(&ticks);
3419       release(&tickslock);
3420     }
3421     lapiceoi();
3422     break;
3423   case T_IRQ0 + IRQ_IDE:
3424     ideintr();
3425     lapiceoi();
3426     break;
3427   case T_IRQ0 + IRQ_IDE+1:
3428     // Bochs generates spurious IDE1 interrupts.
3429     break;
3430   case T_IRQ0 + IRQ_KBD:
3431     kbdintr();
3432     lapiceoi();
3433     break;
3434   case T_IRQ0 + IRQ_COM1:
3435     uartintr();
3436     lapiceoi();
3437     break;
3438   case T_IRQ0 + 7:
3439   case T_IRQ0 + IRQ_SPURIOUS:
3440     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3441             cpuid(), tf->cs, tf->eip);
3442     lapiceoi();
3443     break;
3444
3445
3446
3447
3448
3449
```



```
3450   default:
3451     if(myproc() == 0 || (tf->cs&3) == 0){
3452       // In kernel, it must be our mistake.
3453       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3454               tf->trapno, cpuid(), tf->eip, rcr2());
3455       panic("trap");
3456     }
3457     // In user space, assume process misbehaved.
3458     cprintf("pid %d %s: trap %d err %d on cpu %d "
3459             "eip 0x%x addr 0x%x--kill proc\n",
3460             myproc()->pid, myproc()->name, tf->trapno,
3461             tf->err, cpuid(), tf->eip, rcr2());
3462     myproc()->killed = 1;
3463   }
3464
3465   // Force process exit if it has been killed and is in user space.
3466   // (If it is still executing in the kernel, let it keep running
3467   // until it gets to the regular system call return.)
3468   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3469     exit();
3470
3471   // Force process to give up CPU on clock tick.
3472   // If interrupts were on while locks held, would need to check nlock.
3473   if(myproc() && myproc()->state == RUNNING &&
3474      tf->trapno == T_IRQ0+IRQ_TIMER)
3475     yield();
3476
3477   // Check if the process has been killed since we yielded
3478   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3479     exit();
3480 }
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

Two Problems of LDE

Goal: efficiently **virtualize** the CPU with **control**

Solution technique: **Limited Direct Execution (LDE)**

1. Restricted Operations (system calls)
2. Switching between processes (context switching)

#2 Switching between Processes

- When a process is running on CPU, OS is **not** running

➔if OS is not running, how can it do anything at all?

- How can OS **regain control of CPU** so it can switch from one process to another?
  - **cooperative** approach – OS **trusts** processes, and a process gives up CPU (1) frequently by making system calls, (2) **periodically** by calling `yield()` system call, or (3) does something illegal (illegal memory access) to trap into OS
    - what about infinite loop?
  - **non-cooperative** approach: OS takes control, but **how**?
    - need hardware support – **timer interrupt**
    - if a problem cannot be solved by S/W, ask H/W for help

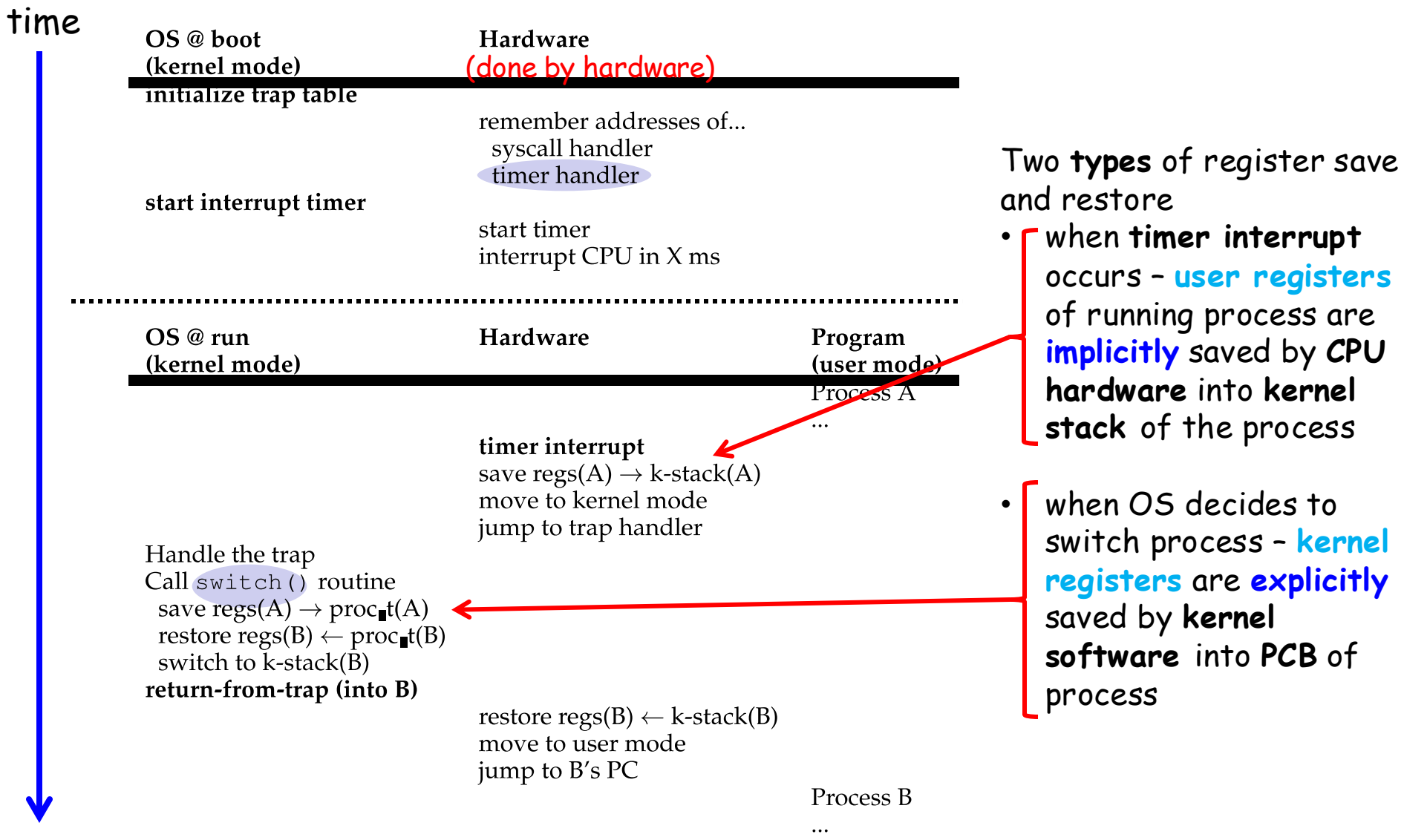
(Timer) Interrupts

- At **boot** time
  - OS informs hardware of **which code to run** when (timer) interrupts occur
  - OS starts the timer (a privileged operation)
- When **hardware** interrupt occurs, the currently running process is halted, and a **pre-configured interrupt handler** in the OS runs. At this point, the OS **regains control** of the CPU
  - **hardware** saves the **context** (machine state) of the currently running process [onto its kernel stack] such that a subsequent `return-from-trap` instruction could **restore the state** to resume its execution

Context Switch

- OS regains control of CPU **cooperatively** via (yield) **system call** or **forcefully** via **timer interrupt**
- **Scheduler** decides which process to run **next**
- **Context switch**
  - **save** context – (execute **assembly code**) to **save** PC, registers, and kernel stack pointer of the "currently-running" process onto its **kernel stack**
  - **restore** context – (execute **assembly code**) to **restore** PC, registers, and kernel stack pointer from the **kernel stack** of the "soon-to-be-executing" process
  - execute `return-from-trap` (resume execution of this "soon-to-be-executing" process)
- By **switching stacks**, kernel enters the `switch()` code in the context of one process (the one that was interrupted) and returns in the context of another (the soon-to-be-executing process) when executing `return-from-trap`

LDE with Timer Interrupt

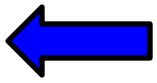


Context Switching

- Two types of register saves/restores
  - when timer interrupt occurs, **user registers** of the running process (*A*) are **implicitly saved by hardware** into the **kernel stack** of *A* → trap into kernel
  - execute Kernel's **timer interrupt handler** which decides to switch from process *A* to process *B*
    - call `switch()` so that **kernel registers** of *A* are **explicitly saved by software** (i.e., the kernel code) into memory in the **process structure (PCB)** of process *A*
    - restore **kernel registers** of *B* from *B*'s PCB
    - **change "stack pointer" to use *B*'s kernel stack (switch context)**
    - OS returns-from-trap → restore *B*'s registers (resume from *B*'s kernel stack) and start running *B* (from *B*'s PC)
  - **moves the system from running as if it just trapped into the kernel from *A* to as if it just trapped into the kernel from *B***
- Kernel enters `switch()` in the context of the interrupted process and returns in the context of another process

Where to Go When Trap?

- How does the trap know which code to run inside OS?
  - the calling process **cannot** specify an address to jump to (as you would when making a function call)
    - doing so would allow programs to jump anywhere into the kernel – a **Very Bad Idea**
- Kernel sets up a **trap table** at boot time
  - when machine boots up, it is in kernel mode, and tells hardware **what code** (**addresses of the code**) to run when certain exceptional events occur (e.g., what code should run)
    - when a **hard-disk interrupt** takes place; when a **keyboard interrupt** occurs; when a program makes a **system call**, etc.
  - kernel informs hardware of the locations (addresses) of these **trap handlers** via special **privileged** instructions



Hardware trapping mechanism

Trap Table/Vector (IDT)

- How does **trap** know which code to run inside the OS?
  - can a calling process specify an address to jump to?
    - of course **not**!
  - kernel must carefully control what code executes upon a trap (with a #)
- Setup **trap table/vector** (**interrupt descriptor table** [**IDT**]) at **boot time**
  - when machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be
  - OS tells hardware what code to run when certain exceptional events occur (e.g., hard disk interrupt, keyboard interrupt, **system call**, exception, etc.)

```
1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "memlayout.h"
1204 #include "mmu.h"
1205 #include "proc.h"
1206 #include "x86.h"
1207
1208 static void startothers(void);
1209 static void mpmain(void)  __attribute__((noreturn));
1210 extern pde_t *kpgdir;
1211 extern char end[]; // first address after kernel loaded from ELF file
1212
1213 // Bootstrap processor starts running C code here.
1214 // Allocate a real stack and switch to it, first
1215 // doing some setup required for memory allocator to work.
1216 int
1217 main(void)
1218 {
1219   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220   kvmalloc();      // kernel page table
1221   mpinit();        // detect other processors
1222   lapicinit();     // interrupt controller
1223   seginit();       // segment descriptors
1224   picinit();       // disable pic
1225   ioapicinit();    // another interrupt controller
1226   consoleinit();   // console hardware
1227   uartinit();      // serial port
1228   pinit();         // process table
1229   tvinit();        // trap vectors
1230   binit();         // buffer cache
1231   fileinit();      // file table
1232   ideinit();       // disk
1233   startothers();   // start other processors
1234   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1235   userinit();      // first user process
1236   mpmain();        // finish this processor's setup
1237 }
```



```
3350 #include "types.h"
3351 #include "defs.h"
3352 #include "param.h"
3353 #include "memlayout.h"
3354 #include "mmu.h"
3355 #include "proc.h"
3356 #include "x86.h"
3357 #include "traps.h"
3358 #include "spinlock.h"
3359
3360 // Interrupt descriptor table (shared by all CPUs).
3361 struct gatedesc idt[256];
3362 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3363 struct spinlock tickslock;
3364 uint ticks;
3365
3366 void
3367 tvinit(void)
3368 {
3369   int i;
3370
3371   for(i = 0; i < 256; i++)
3372     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3373   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3374
3375   initlock(&tickslock, "time");
3376 }
```

Initialize trap vector table

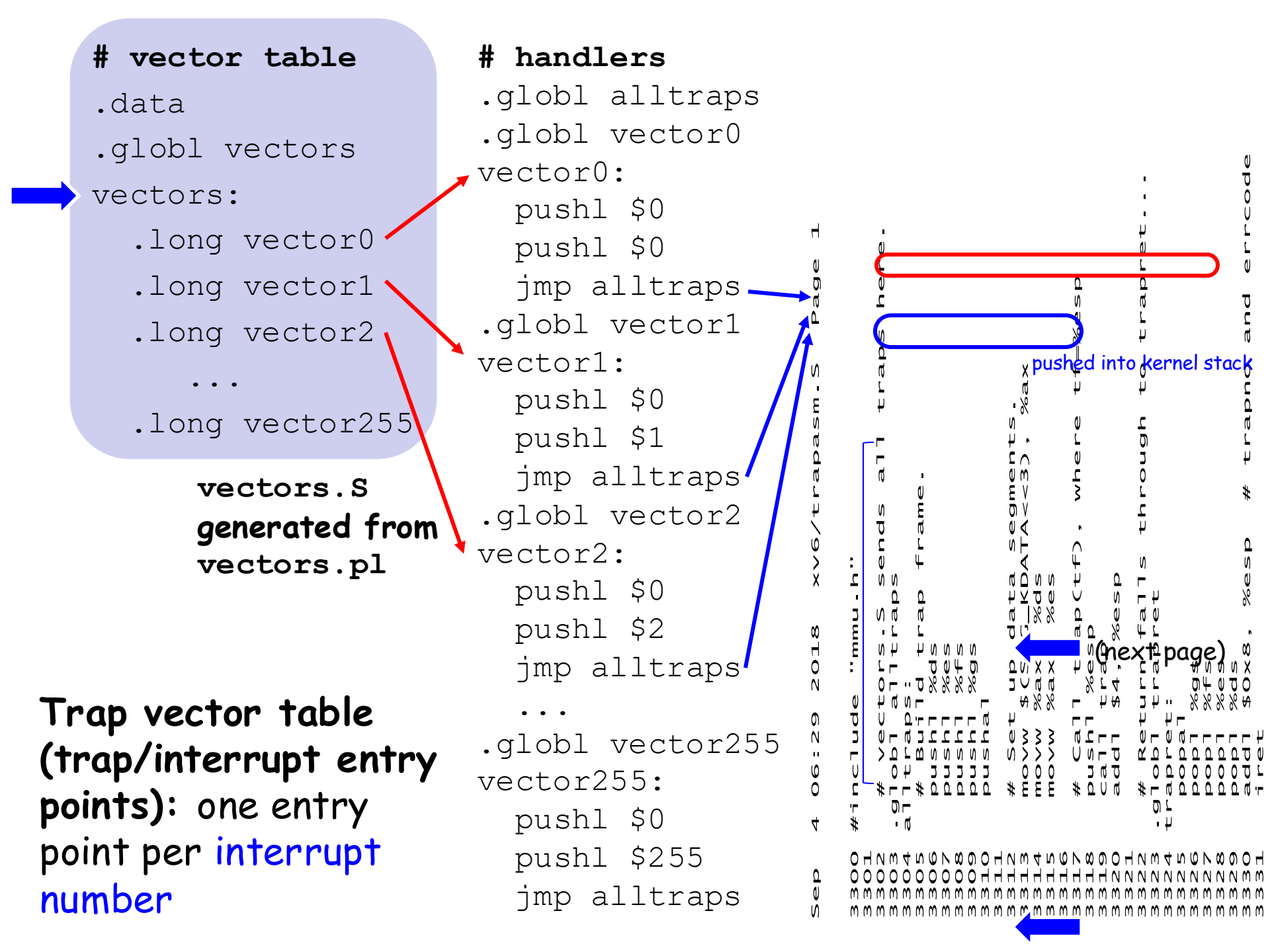
Specific for system calls

```
# vector table
.data
.globl vectors
vectors:
  .long vector0
  .long vector1
  .long vector2
    ...
  .long vector255
```

vectors.S generated from vectors.pl

```
# handlers
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
  jmp alltraps
.globl vector2
vector2:
  pushl $0
  pushl $2
  jmp alltraps
  ...
.globl vector255
vector255:
  pushl $0
  pushl $255
  jmp alltraps
```

Trap vector table (trap/interrupt entry points): one entry point per interrupt number



```
1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "memlayout.h"
1204 #include "mmu.h"
1205 #include "proc.h"
1206 #include "x86.h"
1207
1208 static void startothers(void);
1209 static void mpmain(void)  __attribute__((noreturn));
1210 extern pde_t *kpgdir;
1211 extern char end[]; // first address after kernel loaded from ELF file
1212
1213 // Bootstrap processor starts running C code here.
1214 // Allocate a real stack and switch to it, first
1215 // doing some setup required for memory allocator to work.
1216 int
1217 main(void)
1218 {
1219   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220   kvmalloc();      // kernel page table
1221   mpinit();        // detect other processors
1222   lapicinit();     // interrupt controller
1223   seginit();       // segment descriptors
1224   picinit();       // disable pic
1225   ioapicinit();    // another interrupt controller
1226   consoleinit();   // console hardware
1227   uartinit();      // serial port
1228   pinit();         // process table
1229   tvinit();        // trap vectors
1230   binit();         // buffer cache
1231   fileinit();      // file table
1232   ideinit();       // disk
1233   startothers();   // start other processors
1234   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1235   userinit();      // first user process
1236   mpmain();        // finish this processor's setup
1237 }
```

inform hardware of trap vector

```
3360 // Interrupt descriptor table (shared by all CPUs).
3361 struct gatedesc idt[256];
3362 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3363 struct spinlock tickslock;
3364 uint ticks;
3365
3366 void
3367 tvinit(void)
3368 {
3369   int i;
3370
3371   for(i = 0; i < 256; i++)
3372     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3373   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3374
3375   initlock(&tickslock, "time");
3376 }
3377
3378 void
3379 idtinit(void)
3380 {
3381   lidt(idt, sizeof(idt));
3382 }
```

initialize trap vector table

load interrupt descriptor table register:

```
0525 static inline void
0526 lidt(struct gatedesc *p, int size)
0527 {
0528   volatile ushort pd[3];
0529
0530   pd[0] = size-1;
0531   pd[1] = (uint)p;
0532   pd[2] = (uint)p >> 16;
0533
0534   asm volatile("lidt (%0)" : : "r" (pd));
0535 }
```

```
1200 // Common CPU setup code.
1201 static inline void
1202 mpmain(void)
1203 {
1204   cprintf("cpu%d: starting %d\n", cpu->id, cpu->id);
1205   idtinit();       // load idt register
1206   xchg(&(mycpu()->started), 1); // tell startothers() we're up
1207   scheduler();     // start running processes
1208 }
```

```
0850   ushort t;           // Trap on task switch
0851   ushort iomb;        // I/O map base address
0852 };
0853
0854 // Gate descriptors for interrupts and traps
0855 struct gatedesc {
0856   uint off_15_0 : 16;   // low 16 bits of offset in segment
0857   uint cs : 16;         // code segment selector
0858   uint args : 5;        // # args, 0 for interrupt/trap gates
0859   uint rsv1 : 3;        // reserved(should be zero I guess)
0860   uint type : 4;        // type(STS_{IG32,TG32})
0861   uint s : 1;           // must be 0 (system)
0862   uint dpl : 2;         // descriptor(meaning new) privilege level
0863   uint p : 1;           // Present
0864   uint off_31_16 : 16;  // high bits of offset in segment
0865 };
0866
0867 // Set up a normal interrupt/trap gate descriptor.
0868 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0869 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0870 // - sel: Code segment selector for interrupt/trap handler
0871 // - off: Offset in code segment for interrupt/trap handler
0872 // - dpl: Descriptor Privilege Level -
0873 //        the privilege level required for software to invoke
0874 //        this interrupt/trap gate explicitly using an int instruction.
0875 #define SETGATE(gate, istrap, sel, off, d)                \
0876 {                                                         \
0877   (gate).off_15_0 = (uint)(off) & 0xffff;                \
0878   (gate).cs = (sel);                                      \
0879   (gate).args = 0;                                        \
0880   (gate).rsv1 = 0;                                        \
0881   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0882   (gate).s = 0;                                           \
0883   (gate).dpl = (d);                                       \
0884   (gate).p = 1;                                           \
0885   (gate).off_31_16 = (uint)(off) >> 16;
0886 }
0887
0888 #endif
```

```
3360 // Interrupt descriptor table (shared by all CPUs).
3361 struct gatedesc idt[256];
3362 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3363 struct spinlock tickslock;
3364 uint ticks;
3365
3366 void
3367 tvinit(void)
3368 {
3369   int i;
3370
3371   for(i = 0; i < 256; i++)
3372     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3373   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3374
3375   initlock(&tickslock, "time");
3376 }
```

```
3400 void
3401 trap(struct trapframe *tf)
3402 {
3403   if(tf->trapno == T_SYSCALL){
3404     if(myproc()->killed)
3405       exit();
3406     myproc()->tf = tf;
3407     syscall();
3408     if(myproc()->killed)
3409       exit();
3410     return;
3411   }
3412
3413   switch(tf->trapno){
3414   case T_IRQ0 + IRQ_TIMER:
3415     if(cpuid() == 0){
3416       acquire(&tickslock);
3417       ticks++;
3418       wakeup(&ticks);
3419       release(&tickslock);
3420     }
3421     lapiceoi();
3422     break;
3423   case T_IRQ0 + IRQ_IDE:
3424     ideintr();
3425     lapiceoi();
3426     break;
3427   case T_IRQ0 + IRQ_IDE+1:
3428     // Bochs generates spurious IDE1 interrupts.
3429     break;
3430   case T_IRQ0 + IRQ_KBD:
3431     kbdintr();
3432     lapiceoi();
3433     break;
3434   case T_IRQ0 + IRQ_COM1:
3435     uartintr();
3436     lapiceoi();
3437     break;
3438   case T_IRQ0 + 7:
3439   case T_IRQ0 + IRQ_SPURIOUS:
3440     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3441             cpuid(), tf->cs, tf->eip);
3442     lapiceoi();
3443     break;
```



```
3450   default:
3451     if(myproc() == 0 || (tf->cs&3) == 0){
3452       // In kernel, it must be our mistake.
3453       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3454               tf->trapno, cpuid(), tf->eip, rcr2());
3455       panic("trap");
3456     }
3457     // In user space, assume process misbehaved.
3458     cprintf("pid %d %s: trap %d err %d on cpu %d "
3459             "eip 0x%x addr 0x%x--kill proc\n",
3460             myproc()->pid, myproc()->name, tf->trapno,
3461             tf->err, cpuid(), tf->eip, rcr2());
3462     myproc()->killed = 1;
3463   }
3464
3465   // Force process exit if it has been killed and is in user space.
3466   // (If it is still executing in the kernel, let it keep running
3467   // until it gets to the regular system call return.)
3468   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3469     exit();
3470
3471   // Force process to give up CPU on clock tick.
3472   // If interrupts were on while locks held, would need to check nlock.
3473   if(myproc() && myproc()->state == RUNNING &&
3474      tf->trapno == T_IRQ0+IRQ_TIMER)
3475     yield();
3476
3477   // Check if the process has been killed since we yielded
3478   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3479     exit();
3480 }
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
```



```
3200 // x86 trap and interrupt constants.
3201
3202 // Processor-defined:
3203 #define T_DIVIDE         0      // divide error
3204 #define T_DEBUG          1      // debug exception
3205 #define T_NMI            2      // non-maskable interrupt
3206 #define T_BRKPT          3      // breakpoint
3207 #define T_OFLOW          4      // overflow
3208 #define T_BOUND          5      // bounds check
3209 #define T_ILLOP          6      // illegal opcode
3210 #define T_DEVICE         7      // device not available
3211 #define T_DBLFLT         8      // double fault
3212 // #define T_COPROC      9      // reserved (not used since 486)
3213 #define T_TSS           10      // invalid task switch segment
3214 #define T_SEGNP         11      // segment not present
3215 #define T_STACK         12      // stack exception
3216 #define T_GPFLT         13      // general protection fault
3217 #define T_PGFLT         14      // page fault
3218 // #define T_RES        15      // reserved
3219 #define T_FPERR         16      // floating point error
3220 #define T_ALIGN         17      // alignment check
3221 #define T_MCHK          18      // machine check
3222 #define T_SIMDERR       19      // SIMD floating point error
3223
3224 // These are arbitrarily chosen, but with care not to overlap
3225 // processor defined exceptions or interrupt vectors.
3226 #define T_SYSCALL       64      // system call
3227 #define T_DEFAULT      500      // catchall
3228
3229 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3230
3231 #define IRQ_TIMER        0
3232 #define IRQ_KBD          1
3233 #define IRQ_COM1         4
3234 #define IRQ_IDE         14
3235 #define IRQ_ERROR       19
3236 #define IRQ_SPURIOUS    31
```



```
3700 void
3701 syscall(void)
3702 {
3703   int num;
3704   struct proc *curproc = myproc();
3705
3706   num = curproc->tf->eax;
3707   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3708     curproc->tf->eax = syscalls[num]();
3709   } else {
3710     cprintf("%d %s: unknown sys call %d\n"
3711             curproc->pid, curproc->name, num);
3712     curproc->tf->eax = -1;
3713   }
3714 }
```

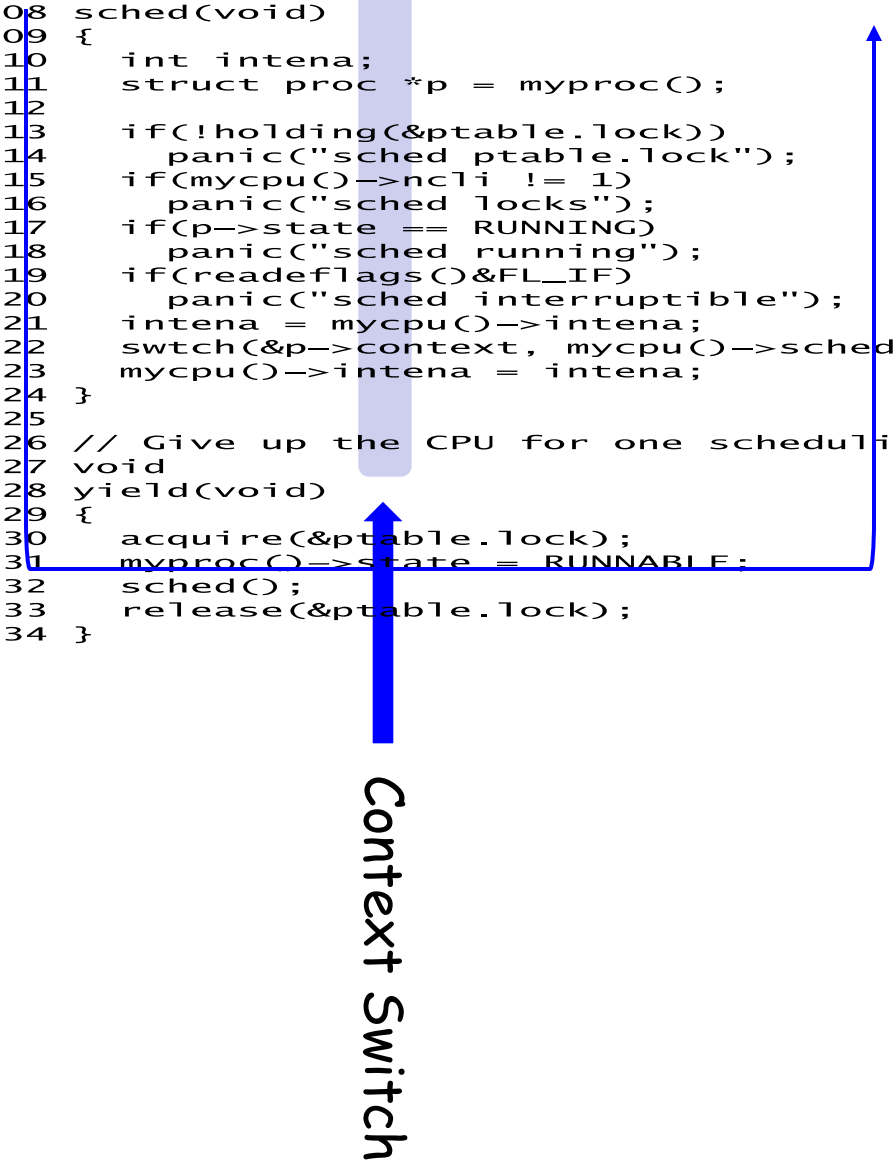
(next page)

system call

interrupts

timer

```
2800 // Enter scheduler.  Must hold only ptable.lock
2801 // and have changed proc->state. Saves and restores
2802 // intena because intena is a property of this
2803 // kernel thread, not this CPU. It should
2804 // be proc->intena and proc->ncli, but that would
2805 // break in the few places where a lock is held but
2806 // there's no process.
2807 void
2808 sched(void)
2809 {
2810   int intena;
2811   struct proc *p = myproc();
2812
2813   if(!holding(&ptable.lock))
2814     panic("sched ptable.lock");
2815   if(mycpu()->ncli != 1)
2816     panic("sched locks");
2817   if(p->state == RUNNING)
2818     panic("sched running");
2819   if(readeflags()&FL_IF)
2820     panic("sched interruptible");
2821   intena = mycpu()->intena;
2822   swtch(&p->context, mycpu()->scheduler);
2823   mycpu()->intena = intena;
2824 }
2825
2826 // Give up the CPU for one scheduling round.
2827 void
2828 yield(void)
2829 {
2830   acquire(&ptable.lock);
2831   myproc()->state = RUNNABLE;
2832   sched();
2833   release(&ptable.lock);
2834 }
```



```
2316 // Saved registers for kernel context switches.
2317 // Don't need to save all the segment registers (%cs, etc),
2318 // because they are constant across kernel contexts.
2319 // Don't need to save %eax, %ecx, %edx, because the
2320 // x86 convention is that the caller has saved them.
2321 // Contexts are stored at the bottom of the stack they
2322 // describe; the stack pointer is the address of the context.
2323 // The layout of the context matches the layout of the stack in swtch.S
2324 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2325 // but it is on the stack and allocproc() manipulates it.
2326 struct context {
2327   uint edi;
2328   uint esi;
2329   uint ebx;
2330   uint ebp;
2331   uint eip;
2332 };
2333
2334 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNING, ZOMBIE };
2335
2336 // Per-process state
2337 struct proc {
2338   uint sz;                     // Size of process memory (bytes)
2339   pde_t* pgdir;                // Page table
2340   char *kstack;                // Bottom of kernel stack for this process
2341   enum procstate state;        // Process state
2342   int pid;                     // Process ID
2343   struct proc *parent;         // Parent process
2344   struct trapframe *tf;        // Trap frame for current syscall
2345   struct context *context;     // swtch() here to run process
2346   void *chan;                  // If non-zero, sleeping on chan
2347   int killed;                  // If non-zero, have been killed
2348   struct file *ofile[NOFILE];  // Open files
2349   struct inode *cwd;           // Current directory
```

PCB

next page

When xv6 handles a system call, the current process's `proc->tf` is set to the interrupt trap frame, which contains the register set that will get restored to the processor when the system call returns. This way, system calls can interact with the register set and give return values back to user mode.

```
0600 // Layout of the trap frame built on the stack by the
0601 // hardware and by trapasm.S, and passed to trap().
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // useless & ignored
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
0612
0613   // rest of trap frame
0614   ushort gs;
0615   ushort padding1;
0616   ushort fs;
0617   ushort padding2;
0618   ushort es;
0619   ushort padding3;
0620   ushort ds;
0621   ushort padding4;
0622   uint trapno;
0623
0624   // below here defined by x86 hardware
0625   uint err;
0626   uint eip;
0627   ushort cs;
0628   ushort padding5;
0629   uint eflags;
0630
0631   // below here only when crossing rings, such as from user to kernel
0632   uint esp;
0633   ushort ss;
0634   ushort padding6;
0635 };
```

bottom


```
3300 #include "mmu.h"
3301
3302   # vectors.S sends all traps here.
3303 .globl alltraps
3304 alltraps:
3305   # Build trap frame.
3306   pushl %ds
3307   pushl %es
3308   pushl %fs
3309   pushl %gs
3310   pushal
3311
3312   # Set up data segments.
3313   movw $(SEG_KDATA<<3), %ax
3314   movw %ax, %ds
3315   movw %ax, %es
3316
3317   # Call trap(tf), where tf=%esp
3318   pushl %esp
3319   call trap
3320   addl $4, %esp
3321
3322   # Return falls through to trapret...
3323 .globl trapret
3324 trapret:
3325   popal
3326   popl %gs
3327   popl %fs
3328   popl %es
3329   popl %ds
3330   addl $0x8, %esp  # trapno and errcode
3331   iret
```

```
3050 # Context switch
3051 #
3052 #   void swtch(struct context **old, struct context *new);
3053 #
3054 # Save the current registers on the stack, creating
3055 # a struct context, and save its address in *old.
3056 # Switch stacks to new and pop previously-saved registers.
3057
3058 .globl swtch
3059 swtch:
3060   movl 4(%esp), %eax
3061   movl 8(%esp), %edx
3062
3063   # Save old callee-saved registers
3064   pushl %ebp
3065   pushl %ebx
3066   pushl %esi
3067   pushl %edi
3068
3069   # Switch stacks
3070   movl %esp, (%eax)
3071   movl
3072
3073   # Lo                    ed registers
3074   popl
3075   popl
3076   popl
3077   popl
3078   ret
```

Context Switching

https://samwho.dev/blog/2013/06/01/context-switching-on-x86/

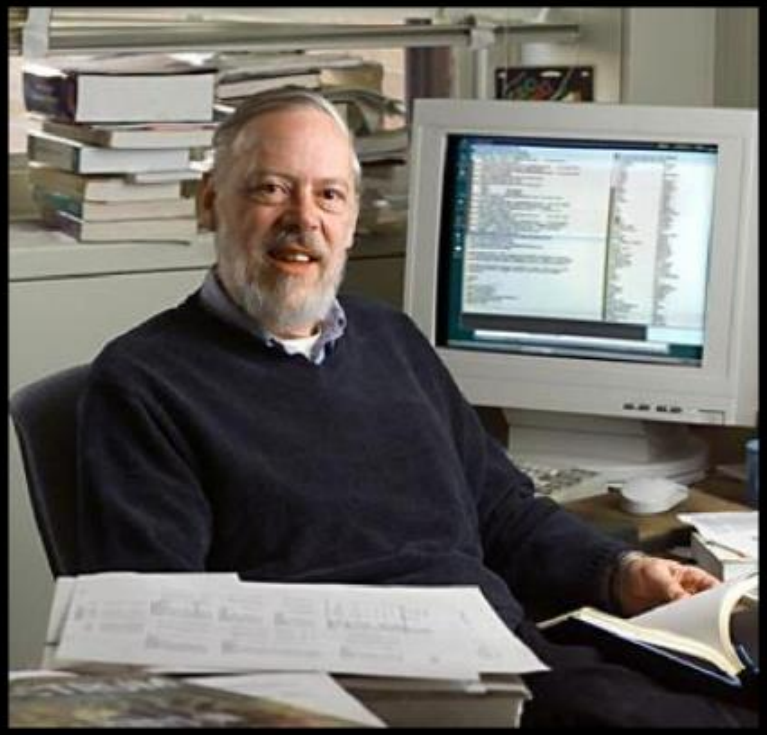
Concurrency

- What happens when a timer interrupt occurs during a system call?
- What happens when an interrupt occurs during the handling of an earlier interrupt?
- Two possible solutions
  - disabling interrupts during interrupt processing
  - locking to protect concurrent access to internal kernel data structures

Unix and Kernel Stack

- In the original implementation of UNIX, kernel memory was at a premium; main memory was roughly one million times **more expensive** per byte than it is today
- The initial system could run with only 50KB of main memory
- Instead of allocating an entire interrupt stack per process, Unix allocated just enough memory in the process control block to store the user-level registers saved on a mode switch
- In this way, Unix could suspend a user-level process with the minimal amount of memory
- Unix still needed a few kernel stacks: one to run the interrupt handler and one for every system call waiting for an I/O event to complete, but that is much less than one for every process
- Now that memory is much cheaper, most systems keep things simple and allocate **a kernel stack per process** or thread

https://www.bell-labs.com/var/articles/invention-unix/



Your attention, please:

Both of these men died in the same month of the same year. Steve was largely considered a hero, while Dennis was largely ignored by the world. Only a handful of programmers who know the real value of Dennis Ritchie´s work even know of his death.

Without Steve Jobs there is no iPhone, iPad, iPod or Macintosh. Without Dennis there is no C. Without C, there is no Unix, Windows, or Linux. Without C there is no C++ nor Objective C. There is no MacOS X, no iOS, no Photoshop, no FLStudio, no Firefox, no Safari, no Google Chrome, no Playstation, no XBox. In fact, 90% of the applications in the world are written in C or C++ or Objective C. **If you think Dennis deserves our respect, please pass this along.**