Chapter 6. Mechanism: Limited Direct Execution

Dr. Jianhui Yue

Slides adapted from Dr. Youjip Won

How to efficiently virtualize the CPU with control?

- The OS needs to share the physical CPU by time sharing.
- Issue
  - **Performance**: How can we implement virtualization without adding excessive overhead to the system?
  - **Control**: How can we run processes efficiently while retaining control over the CPU?

Direct Execution

- Just run the program directly on the CPU.

OS	Program
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

Without *limits* on running programs,
the OS wouldn't be in control of anything and
thus would be "just a library"

Recap: Process Creation

1. **Load** a program code into memory, into the address space of the process.
   - Programs initially reside on disk in *executable format*.
   - OS perform the loading process lazily.
     - Loading pieces of code or data only as they are needed during program execution.

2. The program's run-time **stack** is allocated.
   - Use the stack for *local variables*, *function parameters*, and *return address*.
   - Initialize the stack with arguments → `argc` and the `argv` array of `main()` function

Recap: Process Creation (Cont.)

3. The program's **heap** is created.
   - Used for explicitly requested dynamically allocated data.
   - Program request such space by calling `malloc()` and free it by calling `free()`.

4. The OS do some other initialization tasks.
   - input/output (I/O) setup
     - Each process by default has three open file descriptors.
     - Standard input, output and error

5. **Start the program** running at the entry point, namely `main()`.
   - The OS *transfers control* of the CPU to the newly-created process.

Problem

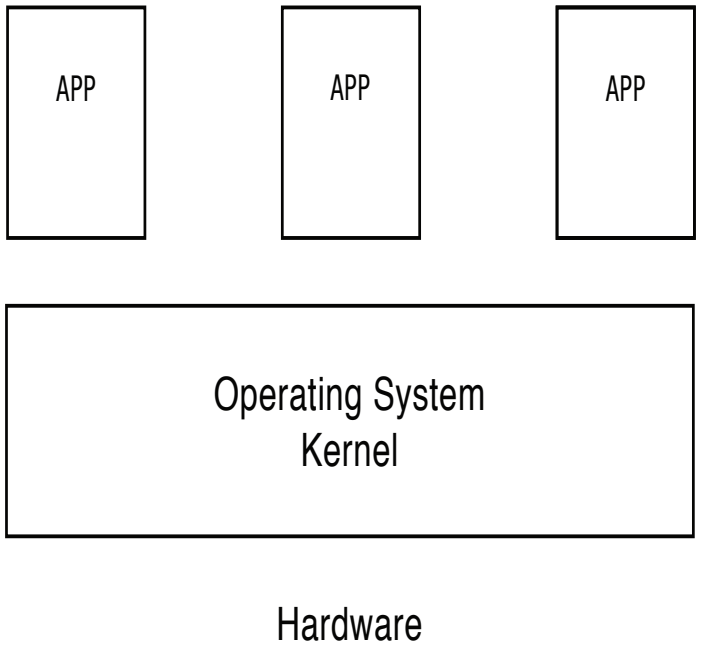
- Issue 1
  - User can do wrong thing.
  - What if?

```
int *i ;
i = 0  ;
*i = 1 ;
```

- Issue 2
  - Getting the control back from CPU is not easy.
  - What if?

```
i = -1 ;
while (i <0)
     do something;
```

OS Kernel Implements protection



Apps accesses to hardw are are limited

Kernel has full access t o all of hardware

Goal: Protection

Why does an OS have a kernel?

- Reliability
  - Prevent buggy program from causing crashes
- Security
  - Limit the scope of what untrusted programs can do
- Privacy
  - On a multiple-user OS, each user must be limited to o nly data that the user is permitted to access
- Fair resource allocation
  - OS limit the amount of resource allocated to each app

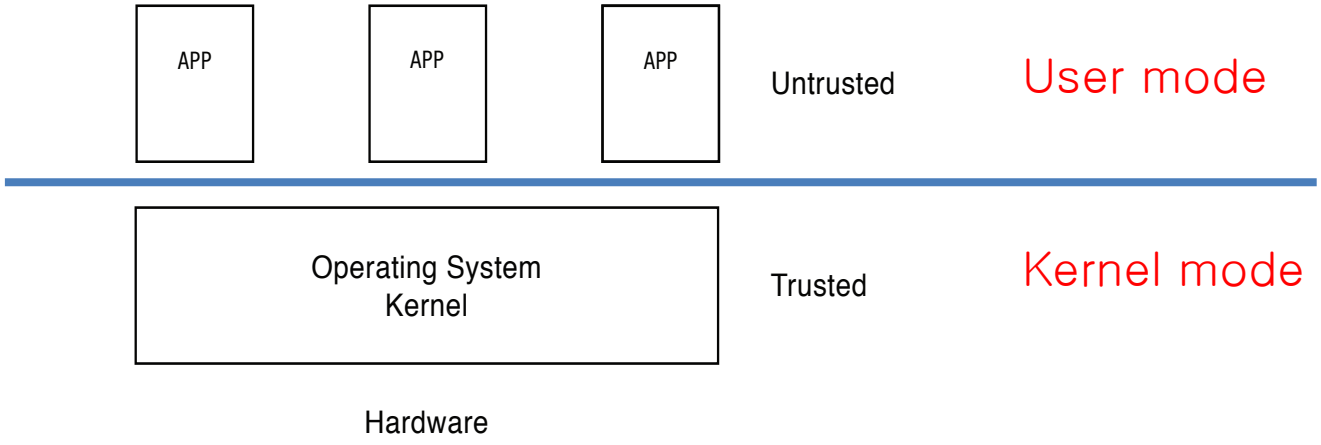
Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as …
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory

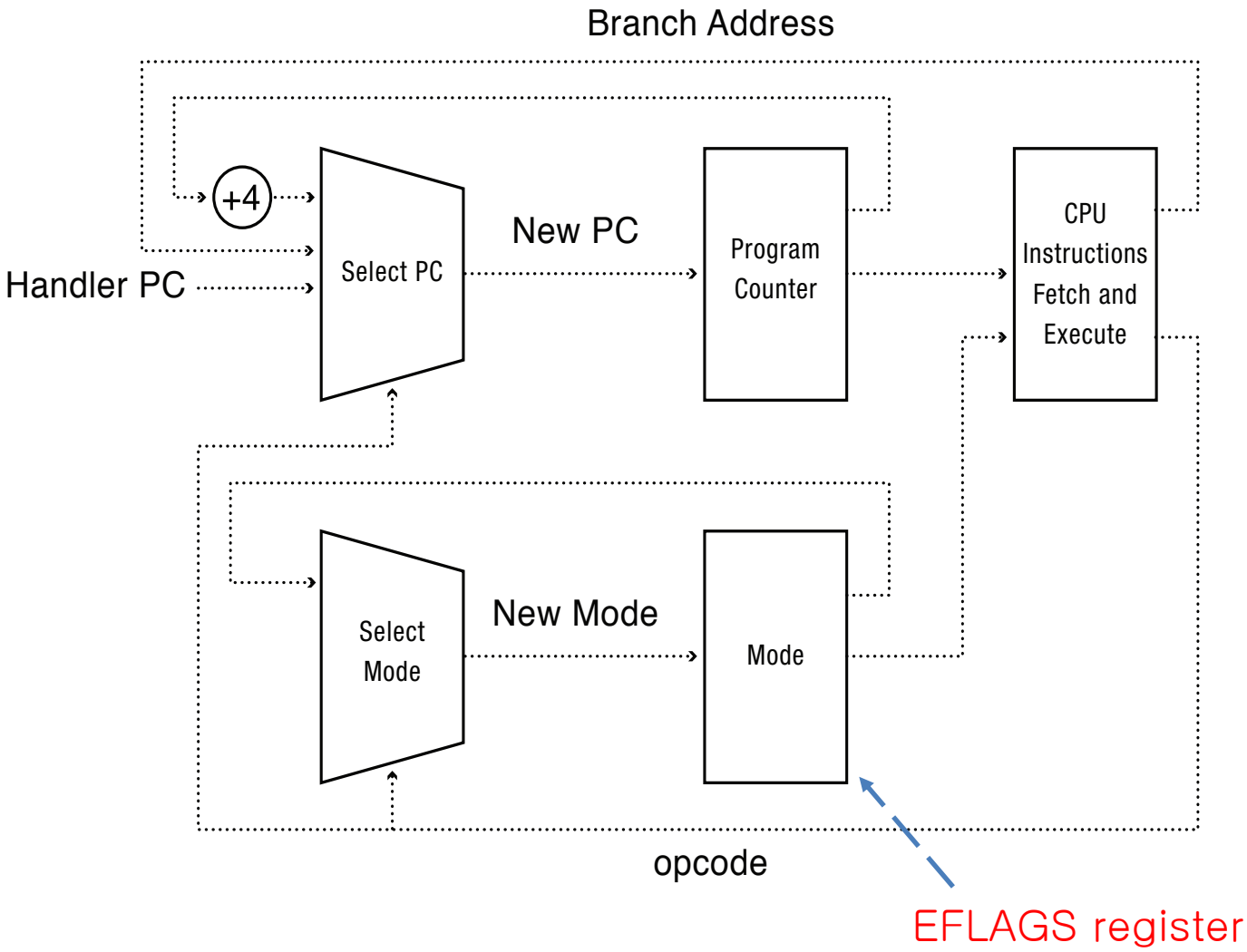
- **Solution 1:** Grant everything.
- **Solution**: Use protected control transfer
  - User mode: Applications do not have full access to hardware resources.
  - Kernel mode: The OS has access to the full resources of the machine

Hardware Support: Dual-Mode Operation

- Kernel mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device,
  - read/write any disk sector, send/read any packet
- User mode
  - Limited privileges
  - Only those granted by the operating system kernel
- On the x86, mode stored in EFLAGS register
- On the MIPS, mode in the status register

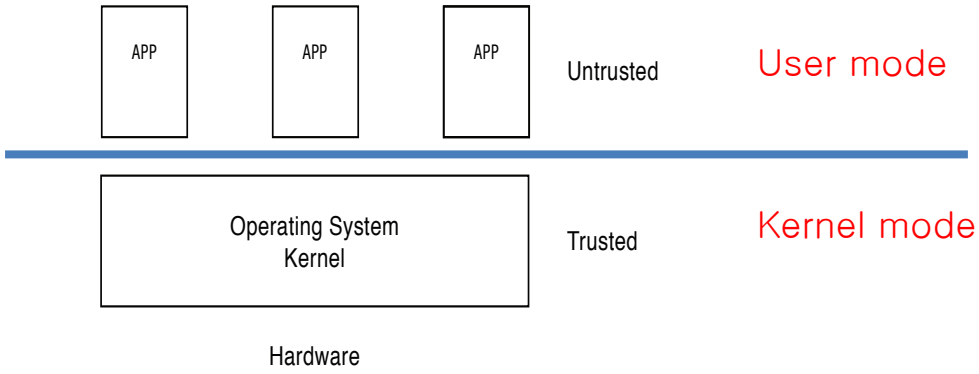


A CPU with Dual-Mode Operation



Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel
- Timer
  - To regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

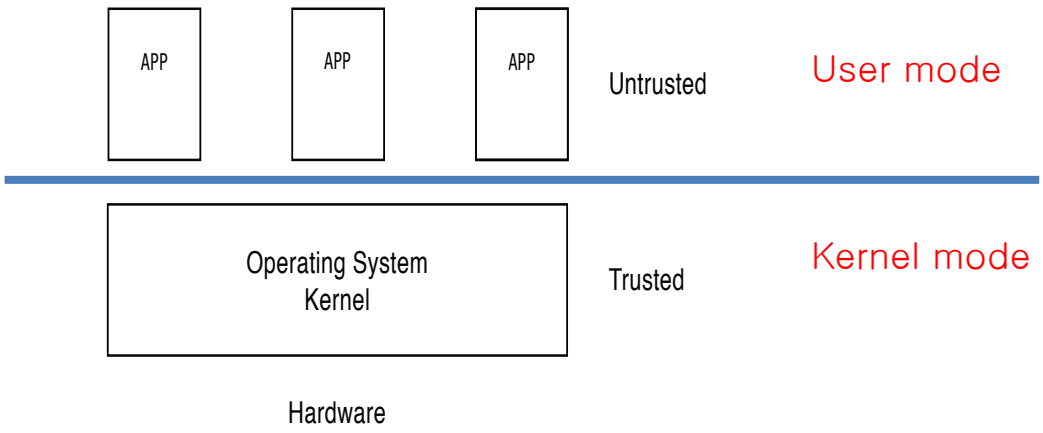


Privilege Levels

- Some processor functionality cannot be made access ible to untrusted user applications
  - e.g. HALT, change MMU settings, set clock, reset devices, m anipulate device settings, …
- Need to have a designated mediator between untrus ted/untrusting applications
  - The operating system (OS)
- Need to delineate between untrusted applications an d OS code
  - Use a “privilege mode” bit in the processor
  - 0 = Untrusted = user, 1 = Trusted = OS

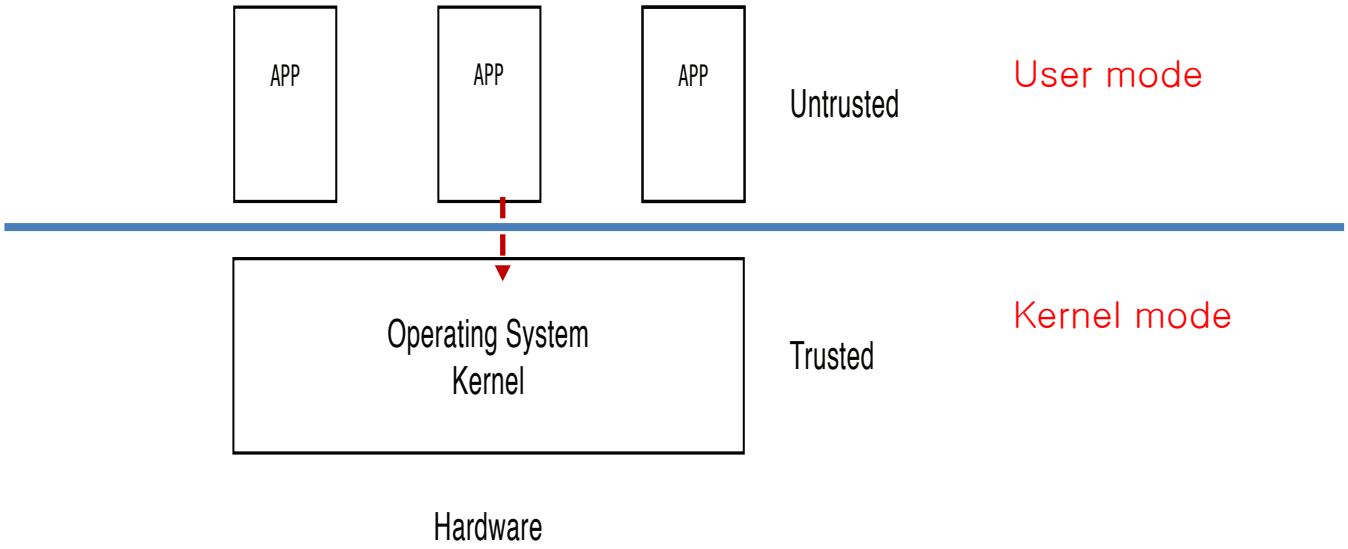
Privilege Mode

- Privilege mode bit indicates if the current program ca n perform privileged operations
  - On system startup, privilege mode is set to 1, and the processor jumps to a well-known address
  - The operating system (OS) boot code resides at this address
  - The OS sets up the devices, initializes the MMU, loads applications, an d resets the privilege bit before invoking the application
- Applications must transfer control back to OS for priv ileged operations



What should happen on executing a privileged instruction?

- Change mode bit in EFLAGs register!
- Change which memory locations a user program can access
- Send commands to I/O devices
- Read data from/write data to I/O devices
- Jump into kernel code
- …



System Call

- Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

Systemcall in XV6 (`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
```

https://github.com/mit-pdos/xv6-public/blob/master/syscall.h

Definitions of system calls in XV6 (`syscall.c`)

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
```

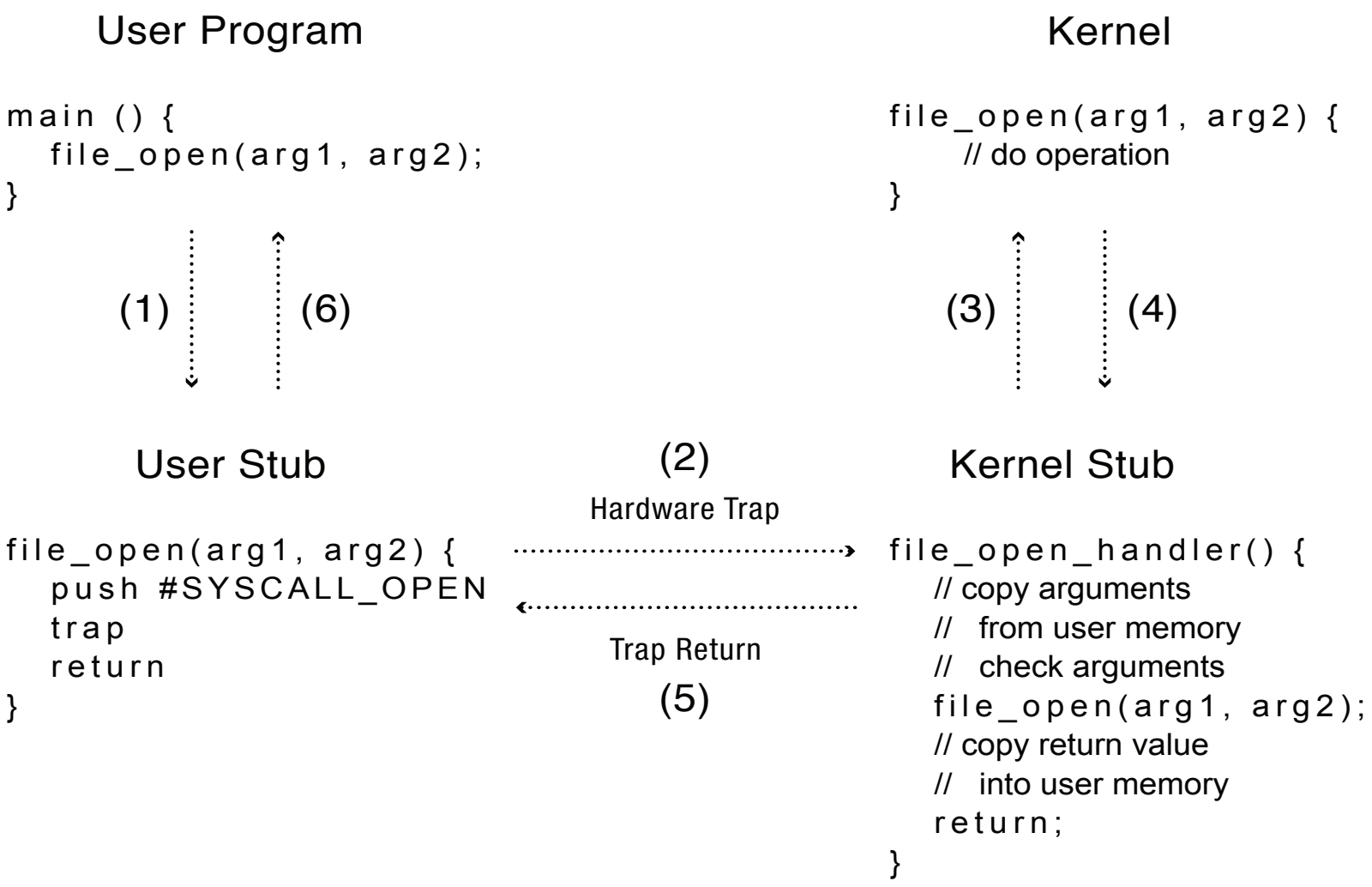
```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[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,
};
```

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

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

https://github.com/mit-pdos/xv6-public/blob/master/syscall.c

Stub Pair for System Call



Kernel System Call Handler

Kernel Stub

```
file_open_handler() {
    // copy arguments
    //   from user memory
    //   check arguments
    file_open(arg1, arg2);
    // copy return value
    //   into user memory
    return;
}
```

- Locate arguments
  - In registers or on user stack
  - *Translate* user addresses into kernel addresses
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back into user memory
  - *Translate* kernel addresses into user addresses

User-level system call stub

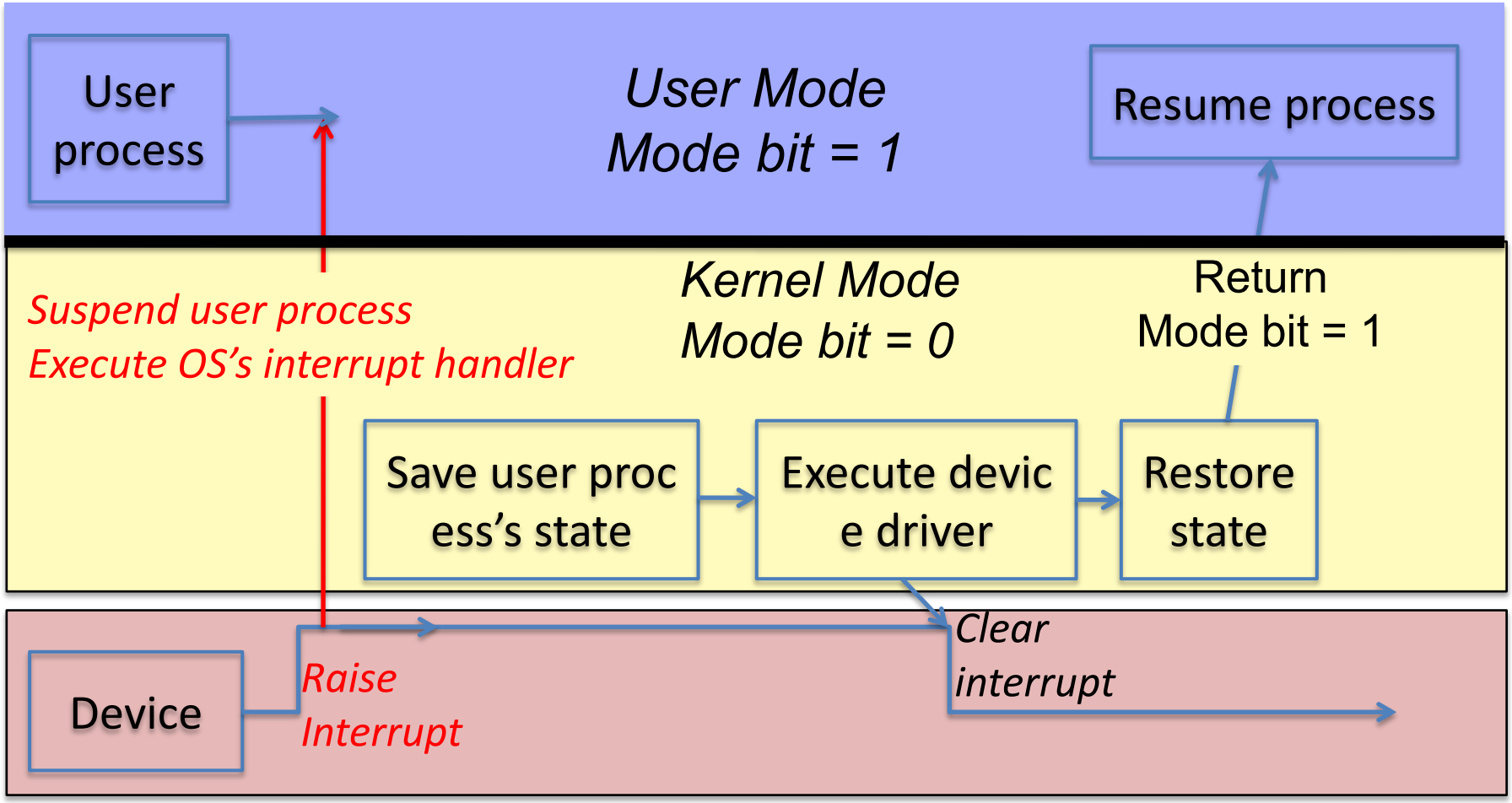
```
// We assume that the caller put the filename onto the stack,
// using the standard calling convention for the x86.

open:
// Put the code for the system call we want into %eax.
 movl    #SysCallOpen, %eax

// Trap into the kernel.
int #TrapCode

// Return to the caller; the kernel puts the return value in %eax.
 ret
```

Interrupt Illustrated



Key Concept of System Call

- **Trap** instruction

```
int n
```

  - Raise the privilege level to kernel mode
  - Save registers at the kernel stack. (`eip, cs, eflags, esp, ss`)
  - Locates the destination in the kernel
  - Jumps to the destination.
  - Interrupt number for system call : `0x64` @ xv6, `0x30` @Pintos
    - e.g. `int 0x64`
- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode

What code to run in trap?

- Sources of trap
  - Completion of disk IO
  - Keyboard interrupt
  - System call
- Trap handler: the code to run for each interrupt number (trap number)
- Trap table
  - The address of the trap handlers.
  - Hardware Informs the location of the trap table to OS when booting.
  - The instruction to inform the location of the trap table is also privileged instruction.
  - You cannot execute this instruction in user mode.

Invention of System Call

- ATLAS (1962-1971)
  - World's first supercomputer with virtual memory(paging), extracode (system call),...

Computer Systems
G. Bell, D. Siewiorek, and S. H. Fuller, Editors

The Manchester Mark I and Atlas: A Historical Perspective

S. H. Lavington
University of Manchester

In 30 years of computer design at Manchester University two systems stand out: the Mark I (developed over the period 1946–49) and the Atlas (1956–62). This paper places each computer in its historical context and then describes the architecture and system software in present-day terminology. Several design concepts such as address-generation and store management have evolved in the progression from Mark I to Atlas. The wider impact of Manchester innovations in these and other areas is discussed, and the contemporary performance of the Mark I and Atlas is evaluated.

Key Words and Phrases: architecture, index registers, paging, virtual storage, extracodes, compilers, operating systems, Ferranti, Manchester Mark I, Atlas, ICL
CR Categories: 1.2, 4.22, 4.32, 6.21, 6.30

1. Introduction and Overview

In the period 1946–76 five computer systems have been designed and implemented at Manchester University. A general account of the prototypes and their industrial derivatives has been given elsewhere [6], along with a comprehensive list of some 60 references to their hardware and software. The main purpose here is to highlight two of the more significant of these five designs. The latest computer in the Manchester series, MU5, is described fully in a companion article [4].

As far as active University research is concerned, Manchester's involvement with digital computers dates from December 1946 when F. C. Williams and Tom Kilburn joined the University from their wartime posts at the Telecommunications Research Establishment. The computer projects, first in the Department of Electrical Engineering and then since 1964 in the Department of Computer Science, have followed the pattern summarized in Table I. This table relates primarily to hardware development; associated system software activity has naturally spanned similar periods, beginning in a small way in 1949 but gathering momentum with the release of the first compiler (1952).

The five prototype computers in the table are the Mark I, the Meg, an experimental transistor computer, the Muse (later Atlas), and the MU5. The names of the five industrially produced derivatives are respectively the Ferranti Mark I, Ferranti Mercury, Metropolitan-Vickers MV950, Ferranti Atlas, and the ICL 2980. The ICL 2980 is not in fact a direct derivative but its architecture owes much to, and has a great deal in common with, MU5. As may be inferred from the table, the cooperation between industry and university has been a fruitful and continuous process since the autumn of 1948. The only one of the five Manchester projects to receive direct government funding was the MU5, which in addition had significant help from ICL in the form of production facilities and engineering support.

The Mark I and Atlas have been chosen for closer study not only because they contain significant innovations, but because they convey an evolutionary progression with respect to the following design themes: i) Instruction format, ii) operand address-generation, iii) store management, and iv) sympathy with high-level language usage. The evolution is continued in MU5 [4]. Whilst all three machines were conceived as general-purpose computers, the internal architecture has tended to favor high-speed scientific applications.

Of the two Manchester computers omitted from detailed analysis in this paper, the Meg (precursor of the Ferranti Mercury) has been passed over because it was essentially an updating of the Mark I concept. By changing the technology and providing parallel access to the main store, the Meg became faster, more compact and easier to maintain. Apart from the incorporation of hardware floating point arithmetic, the instruction format and repertoire were similar to that of the Mark I. The market area of the Ferranti Mercury was much the same as that of the IBM 704, though the 704 was faster and considerably more expensive. The other Manchester computer to be omitted, the experimental point-contact transistor machine, was designed as a small and economic system using a drum as the main store. To help avoid the consequent latency problems a pseudo two-address (or 1 + 1) instruction format was used, in which the address of the next instruction was contained within each instruction. The transistor computer was in this respect untypical of the

Author's Address: Department of Computer Science, University of Manchester, Manchester, M13 9PL, United Kingdom.

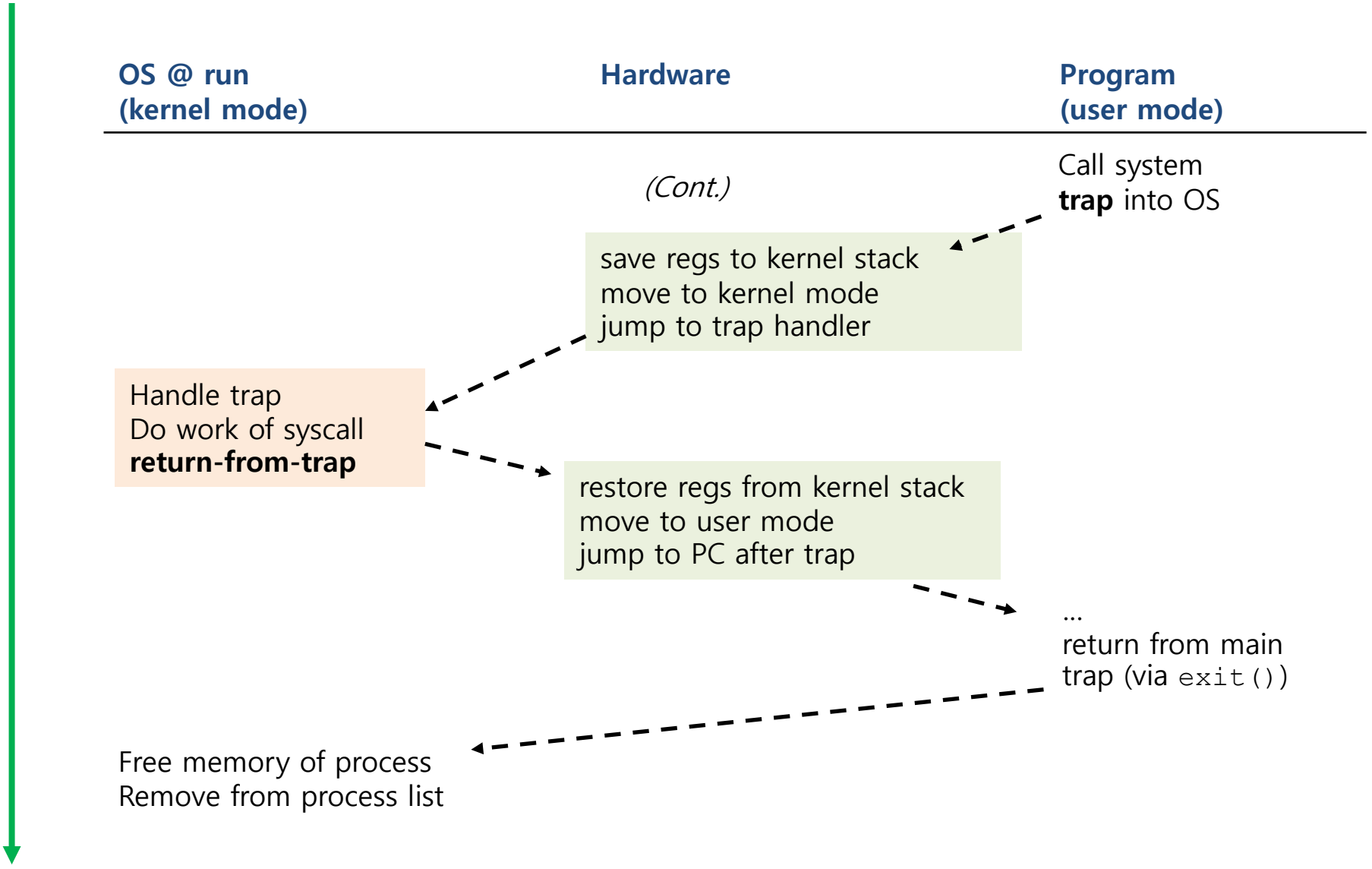
Limited Direction Execution

OS @ boot (kernel mode)	Hardware
initialize trap table	remember address of … syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from -trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() … Call system **trap** into OS

Timeline

Limited Direction Execution (Cont.)

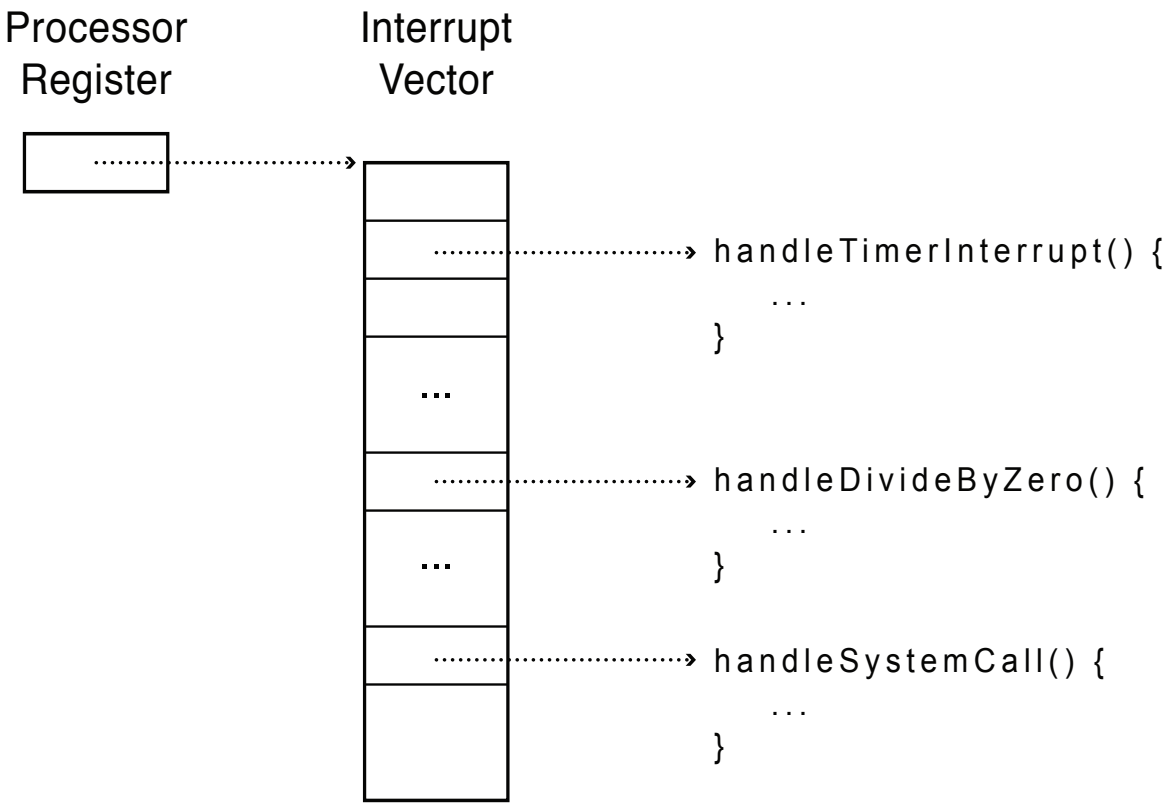


How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel (security)
- Atomic transfer of control
  - The followings are changed at same time:
    - Program counter, Stack pointer, Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred
  - Save the process state to memory upon interrupts, enter kerne l mode, execute the handler
  - Restore the process state from the memory after the handler i s done.

Interrupt Vector

- Table set up by OS kernel; pointers to code to r un on different events



Problem 2: Switching Between Processes

- How can the OS regain control of the CPU so that it can switch between *processes*?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making **system calls** such as `yield`.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access
  - Ex) Early versions of the Macintosh OS, The old Xerox Alto system

A process gets stuck in an infinite loop.
→ Reboot the machine

A Non-Cooperative Approach: OS Takes Control

- **A timer interrupt**
  - During the boot sequence, the OS start the timer.
  - The timer raise an interrupt every few milliseconds.
  - When the interrupt is raised :
    - The currently running process is suspended.
    - Save enough of the state of the process.
    - A pre-configured interrupt handler in the OS runs.

A timer interrupt gives OS the ability to run again on a CPU.

Timer interrupt

- Programming the timer interval
  - Programmed to interrupt CPU periodically
  - Single CPU: PIT (Programmable Interval Timer)
  - Multiple CPU: APIC(Advanced Programmable Interrupt Controller)
    - LAPIC: Local APIC per processor
    - IO APIC on system bus

```
// timer
while (1)
    i = INTERVAL ;
    while (--i > 0) ;
    raise interrupt ;
```

Timer interrupt (1963)

A TIME-SHARING DEBUGGING SYSTEM FOR A SMALL COMPUTER

J. McCarthy
Computation Center, Stanford University, Stanford, California
S. Boilen
Bolt Beranek and Newman Inc., Cambridge, Mass.
E. Fredkin
Information International Inc., Maynard, Mass.
J. C. R. Licklider
Advanced Research Projects Agency, Department of Defense

Channels 6, 7, 11, 14 and 15 are allocated to typewriters.

2. The channel 16 dispatcher.

The computer is in restricted mode during the operation of the time-sharing system. As we stated earlier, this means that *io* instructions and instructions that halt the machine lead to sequence breaks on channel 16. The user programs his input-output just as if there were no time-sharing system. Therefore, when a

tines that are to be used outside it.

3. The channel 17 clock routine.

Every 20 milliseconds or so a sequence break signal is given on channel 17. Since channel 17 is the lowest priority channel this break can take effect only when no typewriter, paper tape or channel 16 dispatcher break is in progress. Moreover, except when the channel 17 program turns off the sequence break system it can be interrupted by typewriter or paper tape sequence breaks.

The basic task of the channel 17 clock routine is to decide whether to remove the current user from core and if so to decide which user program to swap in as he goes out. A user may be removed from core for any of several reasons.

J. McCarthy (1927 – 2011)

- Pioneers of AI along with Alan Turing, Marvin Minsky, Allen Newell.
- The first to use the term "Artificial Intelligence."
- Inventor of 'Lisp' which is a mother of 'scheme', a function programming language.

Saving and Restoring Context

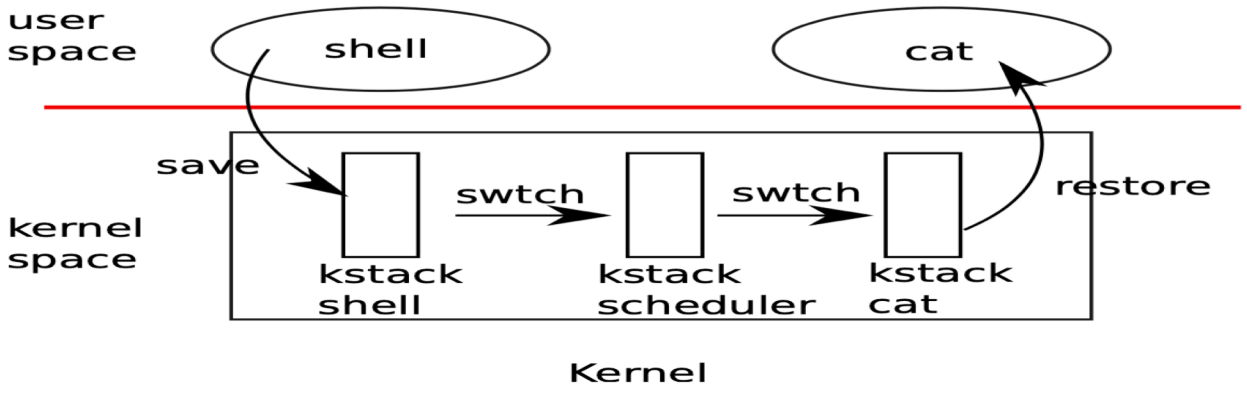
- Scheduler makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS executes context switch.

Context Switch

- A low-level piece of assembly code
  - **Save a few register values** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
  - **Restore a few** for the soon-to-be-executing process from its kernel stack
  - **Switch to the kernel stack** for the soon-to-be-executing process

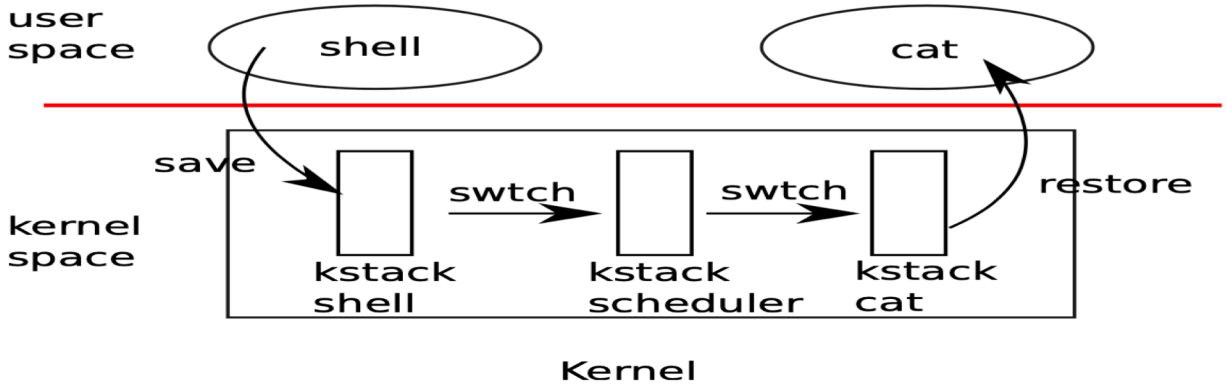
Mode Switch(User->Kernel)

- From user mode to kernel mode
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (aka protected procedure call)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

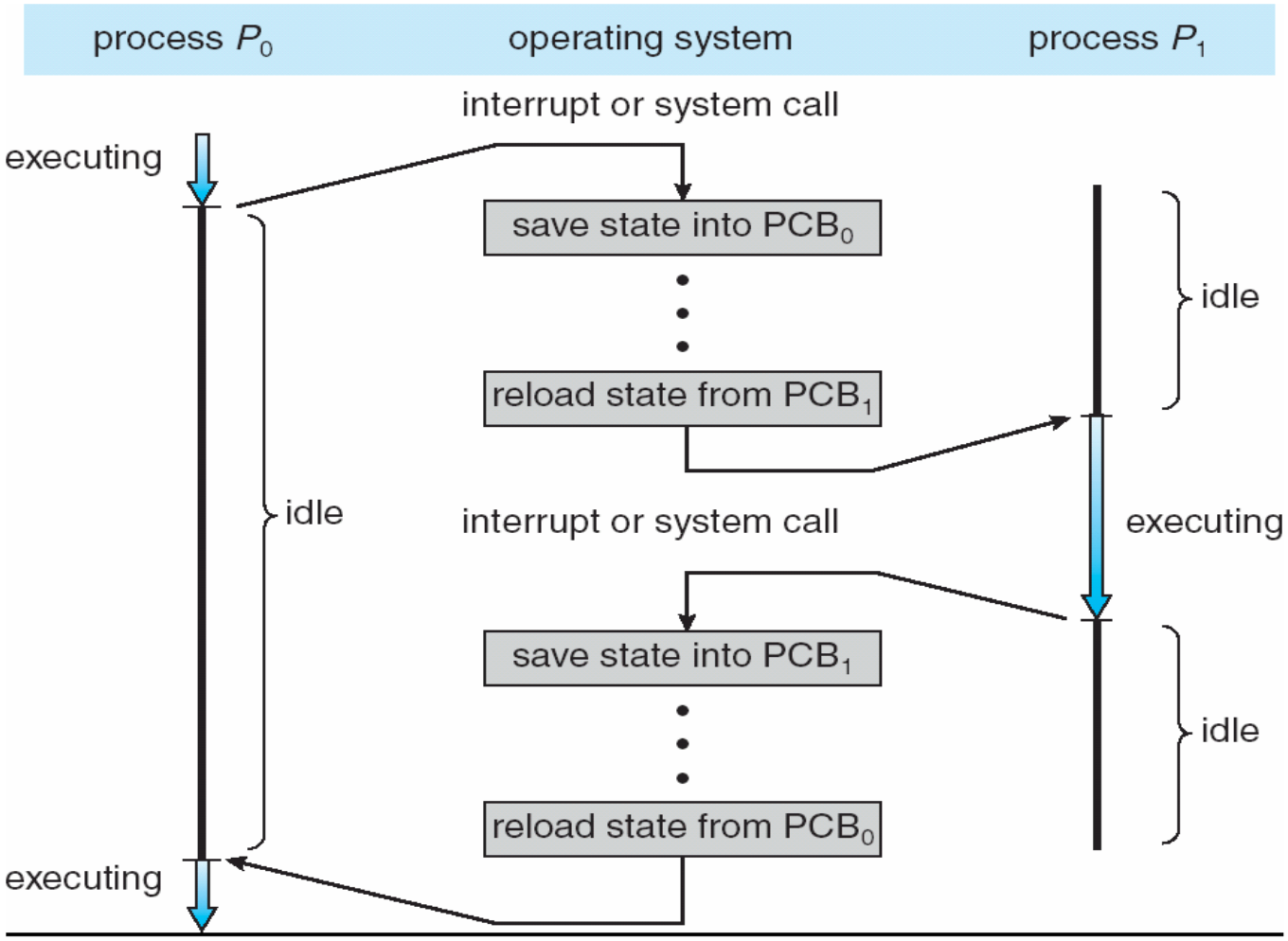


Mode Switch (Kernel ->User)

- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
    - Context switch(Save/restore context)
  - User-level upcall (UNIX signal)
    - Asynchronous notification to user program



CPU Switch From Process to Process



Limited Direct Execution Protocol (Timer interrupt)

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember address of ... syscall handler timer handler	
start interrupt timer	start timer interrupt CPU in X ms	

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Timeline

Limited Direct Execution Protocol (Timer interrupt)

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Handle the trap
Call switch() routine
  save regs(A) to proc-struct(A)
  restore regs(B) from proc-struct(B)
  switch to k-stack(B)
return-from-trap (into B)		

	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B …

Timeline

The xv6 Context Switch Code

```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7          # Save old registers
8          movl 4(%esp), %eax          # put old ptr into eax
9          popl 0(%eax)                # save the old IP
10         movl %esp, 4(%eax)          # and stack
11         movl %ebx, 8(%eax)          # and other registers
12         movl %ecx, 12(%eax)
13         movl %edx, 16(%eax)
14         movl %esi, 20(%eax)
15         movl %edi, 24(%eax)
16         movl %ebp, 28(%eax)
17
18         # Load new registers
19         movl 4(%esp), %eax          # put new ptr into eax
20         movl 28(%eax), %ebp         # restore other registers
21         movl 24(%eax), %edi
22         movl 20(%eax), %esi
23         movl 16(%eax), %edx
24         movl 12(%eax), %ecx
25         movl 8(%eax), %ebx
26         movl 4(%eax), %esp          # stack is switched here
27         pushl 0(%eax)               # return addr put in place
28         ret                         # finally return into new ctxt
```

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.