Operating Systems

6. Mechanism: Limited Direct Execution

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;
```

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

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

Call flow(Pintos): `write()`

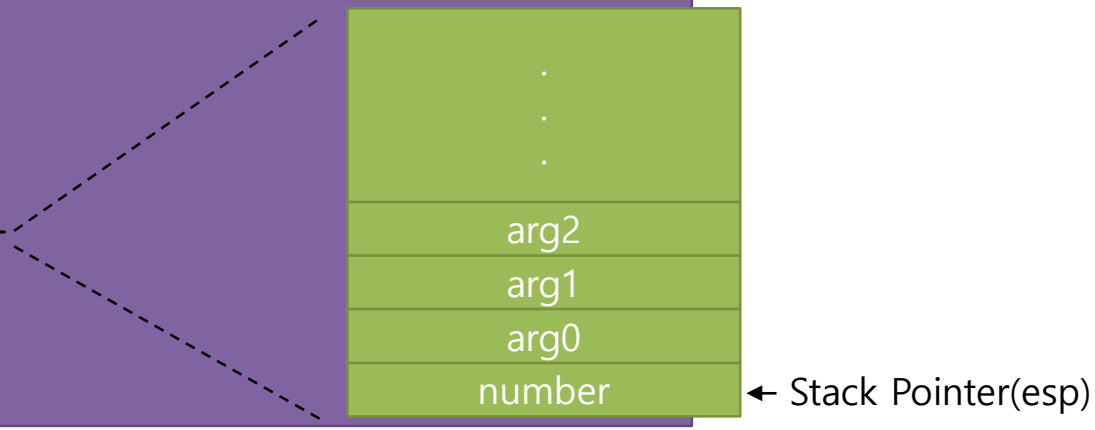
User program

```
main()
{
    write("……")
}
```

```
/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, and
   ARG2, and returns the return value as an `int'. */
#define syscall3(NUMBER, ARG0, ARG1, ARG2)                      \
        ({                                                      \
          int retval;                                           \
          asm volatile                                          \
            ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; "    \
             "pushl %[number]; int $0x30; addl $16, %%esp"      \
               : "=a" (retval)                                  \
               : [number] "i" (NUMBER),                         \
                 [arg0] "g" (ARG0),                             \
                 [arg1] "g" (ARG1),                             \
                 [arg2] "g" (ARG2)                              \
               : "memory");                                     \
          retval;                                               \
        })
```

pintos/src/lib/user/syscall.c

```
int write(…)
{
    return syscall3(SYS_WRITE, fd, buffer, size);
}
…
syscall3(…)
{
…
    push arg2
    push arg1
    push arg0
    push number
    int 0x30
}
```



Definitions of system calls (`src/lib/user/syscall.c`)

```
int
open (const char *file)
{
  return syscall1 (SYS_OPEN, file);
}

int
filesize (int fd)
{
  return syscall1 (SYS_FILESIZE, fd);
}

int
read (int fd, void *buffer, unsigned size)
{
  return syscall3 (SYS_READ, fd, buffer, size);
}

int
write (int fd, const void *buffer, unsigned size)
{
  return syscall3 (SYS_WRITE, fd, buffer, size);
}

void
seek (int fd, unsigned position)
{
  syscall2 (SYS_SEEK, fd, position);
}
```

Systemcall in pintos(systemcall-nr.h)

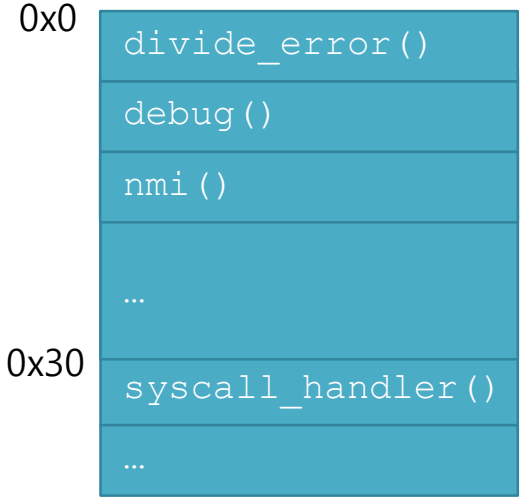
```
/* System call numbers. */
enum
  {
    /* Projects 2 and later. */
    SYS_HALT,                   /* Halt the operating system. */
    SYS_EXIT,                   /* Terminate this process. */
    SYS_EXEC,                   /* Start another process. */
    SYS_WAIT,                   /* Wait for a child process to die. */
    SYS_CREATE,                 /* Create a file. */
    SYS_REMOVE,                 /* Delete a file. */
    SYS_OPEN,                   /* Open a file. */
    SYS_FILESIZE,               /* Obtain a file's size. */
    SYS_READ,                   /* Read from a file. */
    SYS_WRITE,                  /* Write to a file. */
    SYS_SEEK,                   /* Change position in a file. */
    SYS_TELL,                   /* Report current position in a file. */
    SYS_CLOSE,                  /* Close a file. */

    /* Project 3 and optionally project 4. */
    SYS_MMAP,                   /* Map a file into memory. */
    SYS_MUNMAP,                 /* Remove a memory mapping. */

    /* Project 4 only. */
    SYS_CHDIR,                  /* Change the current directory. */
    SYS_MKDIR,                  /* Create a directory. */
    SYS_READDIR,                /* Reads a directory entry. */
    SYS_ISDIR,                  /* Tests if a fd represents a directory. */
    SYS_INUMBER                 /* Returns the inode number for a fd. */
  };
```

Register systemcall handler at IDT

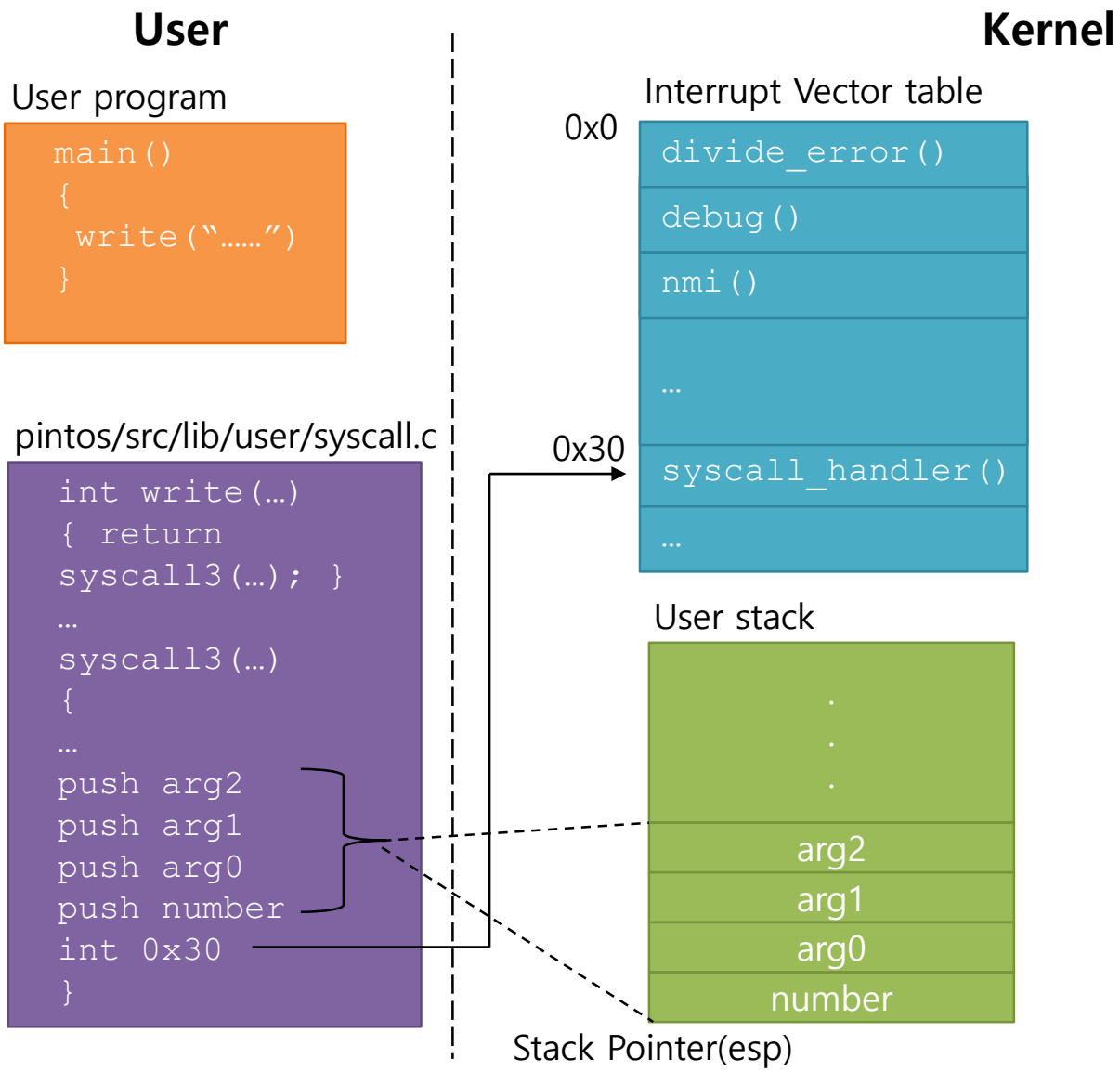
- Interrupt descriptor table
  - Table of the locations of the interrupt handlers for the associated interrupt number.
  - Register the systemcall handler at interrupt `0x30(src/userprog/syscall.c)`.

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}
```

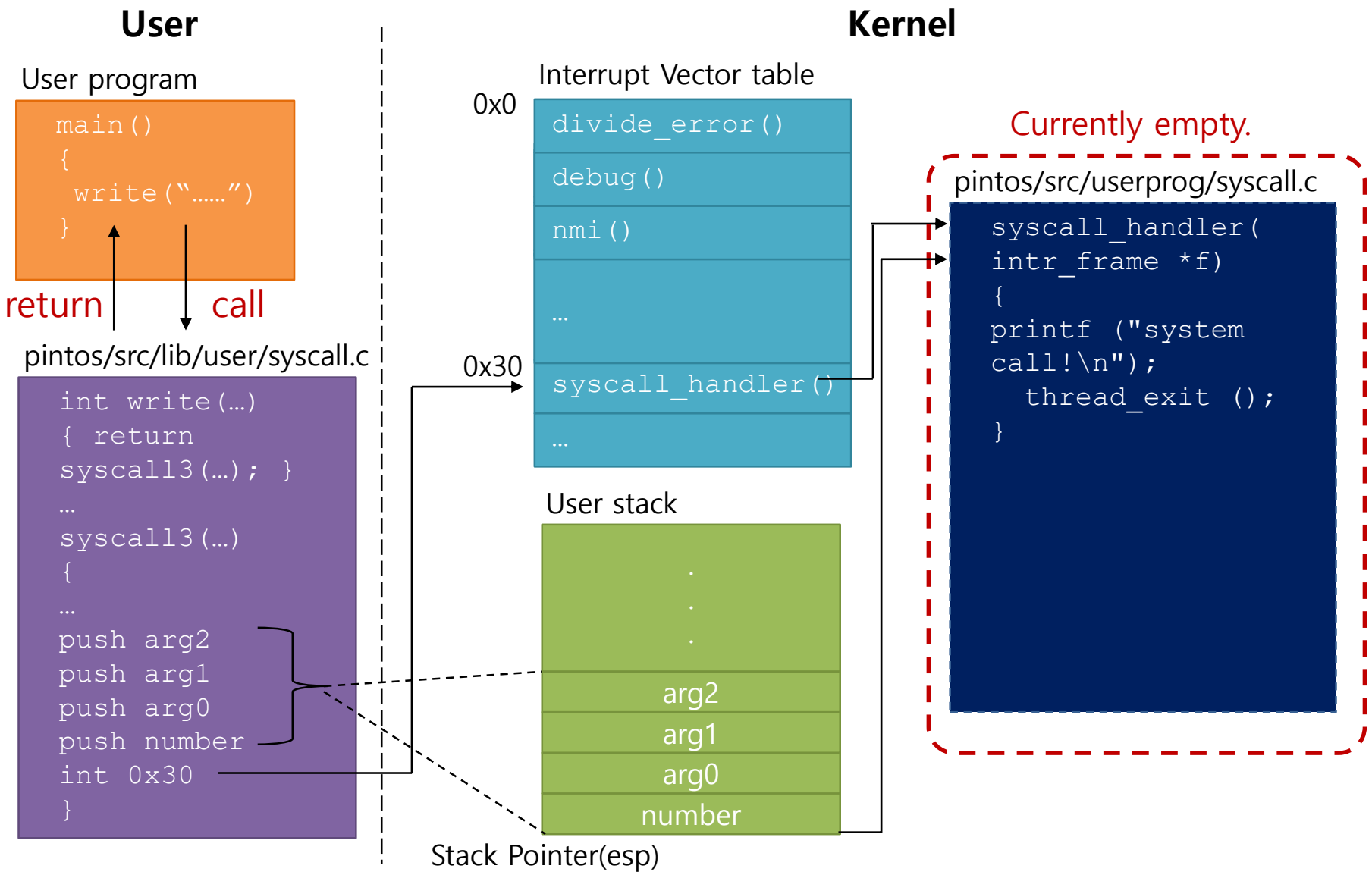
Interrupt Descriptor table



Call flow(Pintos): `write()`



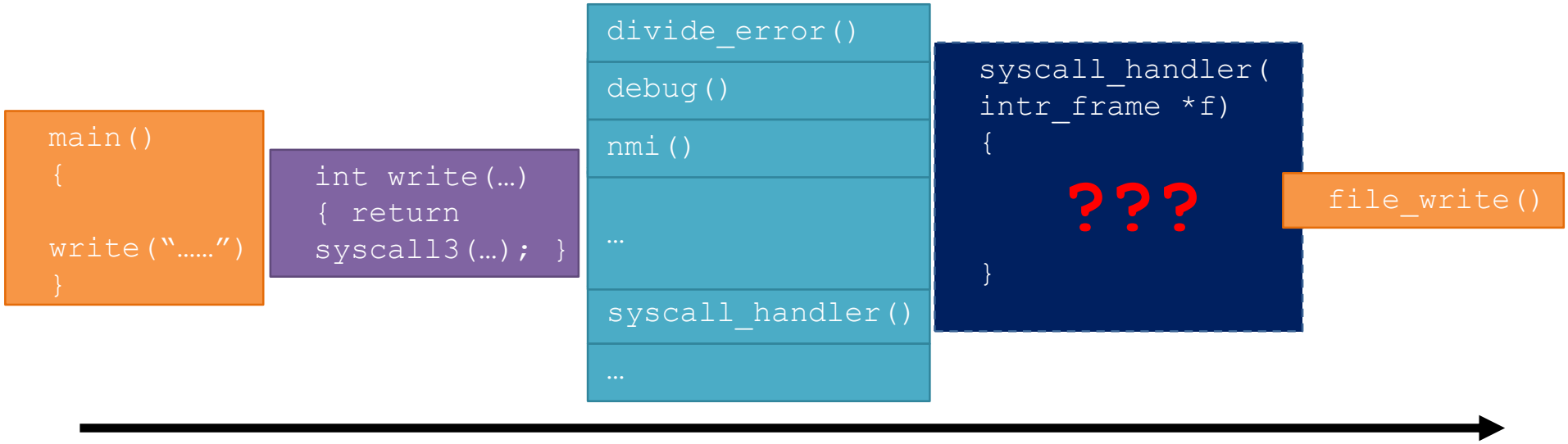
Call process of System call (Pintos)



Implementation of write()(src/filesys.c)

How do you bind the write() system call to `file_write()`?

```
off_t file_write (struct file *file, const void *buffer, off_t size) {
    off_t bytes_written = inode_write_at (file->inode, buffer, size, file->pos);
    file->pos += bytes_written;
    return bytes_written;
    }
```



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.

Computer invented by Britain.

3. Atlas

3.1 Objectives of the Project

By 1956 it was clear that Britain was falling behind the United States in the production of high-performance computers. The MUSE ("micro-second") project, started by Kilburn at Manchester in the autumn of 1956, was a conscious effort to remedy the situation. From January 1959 Ferranti Ltd. officially became

Britain declares war on Germany in 1939

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

Limited Direction Execution (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main trap (via `exit()`)
Free memory of process Remove from process list		

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

lines that are to be used outside it.

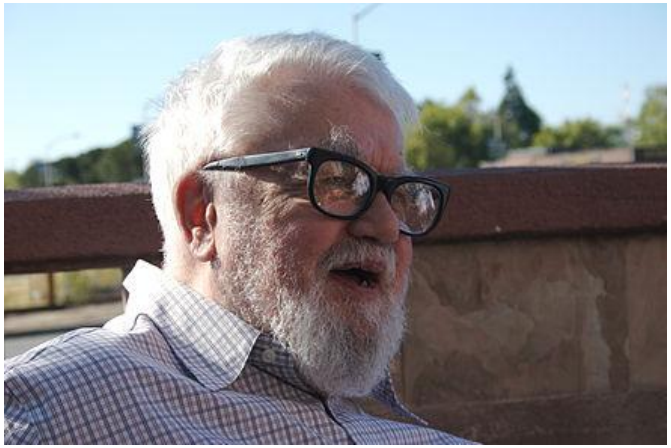
3. The channel 17 clock routine.

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

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	

Limited Direct Execution Protocol (Timer interrupt)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	

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 …

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.