INTERRUPTS, EXCEPTIONS & TRAPS

Operating System
Hebrew University
Spring 2011

MEMORY HIERARCHY

• Main Memory - located on chips inside the computer (outside CPU).
• The program instructions and the processes' data are kept in main memory.
• External Memory - disk. Information stored on a disk is not deleted when the computer turned off.
• The main memory has less storage capacity than the hard disk. The hard disk can write and read information to and from the main memory. The access speed of main memory is much faster than a hard disk.
• Programs are stored on the disk until they are loaded into memory, and then use the disk as both the source and destination of the information for their processing.
**Definitions: Kernel vs. Process**

- The **kernel** is the core of the operating system and it has complete control over everything that happens in the system. The kernel is trusted software, but all other programs are considered untrusted software.
- A **process** is an executing instance of a program. An active process is a process that is currently advancing in the CPU (while other processes are waiting in memory for their turns to use the CPU).
- The CPU can be in kernel mode or in user mode.
- When the CPU is in kernel mode, it is assumed to be executing trusted software, and thus it can execute any instructions and reference any memory addresses.

**Definitions: CPU User & Kernel Mode**

- **User mode** is a non-privileged mode in which processes are forbidden to access those portions of memory that have been allocated to the kernel or to other programs.
- When a user mode process wants to use a service that is provided by the kernel (e.g., a system call), the system must switch temporarily into kernel mode.
- Code running in kernel mode has access to key system resources.
- The entire kernel, which is not a process but a controller of processes, executes only in kernel mode.
- When the kernel has satisfied the request by a process, it returns the processor to user mode.

**Definitions: System Calls & I/O**

- A **system call** is a request to the kernel in an operating system by an active process for a service performed by the kernel.
- **Input/output (I/O)** is any program, operation or device that transfers data to or from a peripheral device (such as disk drives, keyboards, mice and printers).

**Hardware Interrupts: Motivation**

- Much of the functionality embedded inside a computer is implemented by hardware devices other than the processor.
- Since each device operates at its own pace, a method is needed for synchronizing the operation of the processor with these devices.
- One solution is for the processor to sit in a tight loop, asking each device about its current state.
- When data is available in one of the devices, the processor can then read and process the incoming bytes.

**Hardware Interrupts: Solution**

- Instead of polling hardware devices to wait for their response, each device is responsible for notifying the processor about its current state.
- When a hardware device needs the processor's attention, it simply sends an electrical signal (hardware interrupt) through a dedicated pin in the interrupt controller chip (located on the computer's motherboard).

This method works but it has two main disadvantages:

1. **Wasteful** in terms of processing power - the processor is constantly busy reading the status of the attached devices instead of executing some useful code.
2. When the rate of data transfer is extremely high, the **processor might lose data** bytes arriving from the hardware devices.
Definitions: **Interrupts**

- Processes can be interrupted by interrupts, exceptions or traps.
- An interrupt is a signal to the CPU indicating that an event has occurred, and it results in changes in the sequence of instructions that is executed by the CPU. Interrupts are events which aren’t part of the running program’s regular, pre-planned code.
- In the case of hardware interrupts, the signal originates from a hardware device such as a keyboard (i.e., when a user presses a key), mouse or system clock (used to coordinate the computer’s activities).

**Interrupt Causes**

- Caused by an external event which typically needs routine attention.
- For example:
  - Disk drive has data that was requested 20 ms ago.
  - User pressed a key on the keyboard.
  - User sneezed, causing mouse to move.
  - Timer (used by the OS as an alarm clock) expired. E.g., when several programs running simultaneously.

**Dealing with Interrupts**

- Interruption handling is like a dealing with a function call, with the hardware calling a function (“handler”) to deal with it. Hence, we need to save the state as it was when the interruption happened, handle the interruption, and then return to the state as it was.
- Combination of hardware & software is necessary to deal with interrupts.

**The Interrupt Controller**

- The interrupt controller serves as an intermediate between the hardware devices and the processor.
- The interrupt controller has several input lines that take requests from the different devices.
- Its responsibility is to alert the processor when one of the hardware devices needs its immediate attention.
- The controller passes the request to the processor, telling it which device issued the request (which interrupt number triggered the request).

**The basic mechanism**

Similar to a function call:
1. Getting the interrupt
2. Transfer control
3. Saving current state
4. The request is serviced
5. Previous state is restored
6. Return control

**Getting the Interrupt**

- External event interrupts the main program execution.
- An electronic signal is provided to the processor - indicating the need to handle an interrupt request.
- This signal is only recognized at the end of the instruction cycle loop (after the current instruction has been processed, but before the next instruction is "fetched" from memory).
**Transfer control**

Control is transferred to a different "program" - the kernel - switching to kernel mode.

**Saving Current State**

- Before an interrupt can be serviced, the processor must save its current status.
- Servicing an interrupt is like performing a subroutine call.
- One of the most critical pieces of information that must be saved is the value of the Program Counter (i.e. the location of the next instruction to be performed after servicing of the interrupt is complete).
- Processing an interrupt request involves performing a series of instructions for that request. This tends to modify the contents of registers, so the registers also need to be saved.

**The Request is Serviced**

- CPU checks which device sent the interrupt request.
- The processor determines where to find the necessary instructions needed to service that specific request (typically handled using a "interrupt vector" which contains interrupt device numbers and the addresses of service subroutines for each interrupt number).
  - The interrupt vector is stored at a predefined memory location.

**Previous State is Restored**

As a final step in each service routine, all register values, including the Program Counter, must be restored to their original values as they were just before the interrupt occurred.

**Return control**

- Control is returned to the interrupted program.
- The next instruction is pointed by the program counter.
- Back to user mode!

**Example**

- `add r1, r2, r3`
- `sub r4, r5, r6`
- `xor r7, r8, r9`
- As execution reaches code above, *achoo* (user sneezes) → moving mouse → triggering an interrupt.
- Based on time of sneeze (in the middle of sub), hardware completes add and sub, but squashes xor (for now).
- The handler starts:
  - The screen pointer (the little arrow) is moved.
- The handler finishes.
- Execution resumes with xor.
**APC**

- The interrupt is effectively invisible to the interrupted program.
- The interrupt is asynchronous, as the program can't control it.
- Can also be characterized as an "asynchronous procedure call"

**Exceptions**

- Exceptions - similar to an interrupt, but not caused by an external source, but as a part of the regular run of the program.
- They are generated whenever something occurs, such that the processor can't handle an instruction - either critical (division by 0, segmentation fault - memory which hasn’t been allocated) - which usually results in program termination - or temporary.

**Exceptions**

- When an exception occurs, the registers point to the address of the instruction, which generated the exception.
- This gives the exception handler a chance to fix the condition which caused the exception to occur, before restarting the faulting instruction.
- The program is restarted at the address of the fault.
- When there is no reliable address to return to, the program is aborted.

**Page fault example**

- A program requests data that is not currently in real memory.
- An exception triggers the operating system to fetch the data from the disk and load it into main memory.
- The program gets its data without even knowing that an exception has occurred.
- The program continues with the same instruction.

**Trap**

- A trap is similar to an exception, in that it occurs in the usual run of the program, but unlike it, it is not product of some error.
- The execution of an instruction that is intended for user programs and transfers control to the operating system. Such a request from the kernel is called a system call.
- Trap causes switching to OS code and to kernel mode. Once in kernel mode, a trap handler is executed to service the request.
- Restarted at the address following the address causing the trap.

**System Call**

- A mechanism used by an application program to request service from the operating system.
- Provide the interface between a process and the operating system itself.
- Popular system calls are open, read, write, close, wait, exec, fork, exit, and kill.
- Defines the programming environment.
An Example:

open ("/tmp/foo");

User:
“open” lib function:
• store the system call number and the parameters in a predefined kernel memory location;
• trap(); (int #80 asm inst.)
• retrieve the response from a predefined kernel memory location;
• return the response to the calling application;

Kernel:
• Trap handler: transfer to gate:
  • Gate routine:
    switch(sys_call_num) {
    case OPEN: …
    }
  • store response in a predefined memory location;
  Return to user;

User:
• Trap handler: transfer to gate:
  • Gate routine:
    switch(sys_call_num) {
    case OPEN: …
    }
  • store response in a predefined memory location;
  • Return to user;

Check return values!

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
...
int status;
status=open("/tmp/foo");
if( status < 0 ) {
    perror( "Error opening file" );
    //equivalent to:
    //printf("Error opening file: %s\n",strerror(errno));
}
```

Some Possible Values:

<table>
<thead>
<tr>
<th>Error name</th>
<th>Error code (number)</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOENT</td>
<td>2</td>
<td>No such file or directory</td>
</tr>
<tr>
<td>EINTR</td>
<td>4</td>
<td>Interrupted system call</td>
</tr>
<tr>
<td>EIO</td>
<td>5</td>
<td>IO error</td>
</tr>
<tr>
<td>EACCES</td>
<td>13</td>
<td>Permission denied</td>
</tr>
<tr>
<td>EBUSY</td>
<td>16</td>
<td>Device or resource busy</td>
</tr>
</tbody>
</table>