Interrupts, Exceptions, and System Calls

Chester Rebeiro
IIT Madras
OS & Events

• OS is event driven
  – i.e. executes only when there is an interrupt, trap, or system call
Why event driven design?

• OS cannot trust user processes
  – User processes may be buggy or malicious
  – User process crash should not affect OS

• OS needs to guarantee fairness to all user processes
  – One process cannot ‘hog’ CPU time
  – Timer interrupts
Event Types

Events

Interrupts

Hardware Interrupts

Software Interrupts

Exceptions
Events

• **Interrupts**: raised by hardware or programs to get OS attention
  – Types
    • **Hardware interrupts**: raised by external hardware devices
    • **Software Interrupts**: raised by user programs

• **Exceptions**: due to illegal operations
Event view of CPU

while(fetch next instruction)

Execute Instruction

If event
  yes -> Execute event in handler
  no  -> Current task suspended

Where?
Exception & Interrupt Vectors

- Each interrupt/exception provided a number
- Number used to index into an Interrupt descriptor table (IDT)
- IDT provides the entry point into a interrupt/exception handler
- 0 to 255 vectors possible
  - 0 to 31 used internally
  - Remaining can be defined by the OS

What to execute next?

Event occurred
## Exception and Interrupt Vectors

<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
<th>Error Code</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>Fault</td>
<td>No</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>RESERVED</td>
<td>Fault/ Trap</td>
<td>No</td>
<td>For Intel use only.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>NMI Interrupt</td>
<td>Interrupt</td>
<td>No</td>
<td>Nonmaskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>Trap</td>
<td>No</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>Trap</td>
<td>No</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>Fault</td>
<td>No</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (Undefined Opcode)</td>
<td>Fault</td>
<td>No</td>
<td>UD2 instruction or reserved opcode.</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Abort</td>
<td>Yes (zero)</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Coprocessor Segment Overrun (reserved)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point instruction.</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Fault</td>
<td>Yes</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Fault</td>
<td>Yes</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack-Segment Fault</td>
<td>Fault</td>
<td>Yes</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Fault</td>
<td>Yes</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Fault</td>
<td>Yes</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>(Intel reserved. Do not use.)</td>
<td>Fault</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>x87 FPU Floating-Point Error (Math Fault)</td>
<td>Fault</td>
<td>No</td>
<td>x87 FPU floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Fault</td>
<td>Yes (Zero)</td>
<td>Any data reference in memory.</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Abort</td>
<td>No</td>
<td>Error codes (if any) and source are model dependent.</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>Fault</td>
<td>No</td>
<td>SSE/SSE2/SSE3 floating-point instructions</td>
</tr>
<tr>
<td>20</td>
<td>#VE</td>
<td>Virtualization Exception</td>
<td>Fault</td>
<td>No</td>
<td>EPT violations</td>
</tr>
<tr>
<td>21-31</td>
<td></td>
<td>Intel reserved. Do not use.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td></td>
<td>User Defined (Non-reserved) Interrupts</td>
<td>Interrupt</td>
<td>No</td>
<td>External interrupt or INT n instruction.</td>
</tr>
</tbody>
</table>
xv6 Interrupt Vectors

• 0 to 31 reserved by Intel
• 32 to 63 used for hardware interrupts
  \[ T_{IRQ0} = 32 \] (added to all hardware IRQs to scale them)
• 64 used for system call interrupt

ref: traps.h ([31], 3152)
Events

- Interrupts
  - Hardware Interrupts
  - Software Interrupts

- Exceptions
Why Hardware Interrupts?

• Several devices connected to the CPU
  – eg. Keyboards, mouse, network card, etc.
• These devices occasionally need to be serviced by the CPU
  – eg. Inform CPU that a key has been pressed
• These events are asynchronous i.e. we cannot predict when they will happen.
• Need a way for the CPU to determine when a device needs attention
Possible Solution: Polling

- CPU periodically queries device to determine if they need attention
- Useful when device often needs to send information
  - For example in data acquisition systems
- If device does not need attention often,
  - Polling wastes CPU time
Interrupts

• Each device signals to the CPU that it wants to be serviced
• Generally CPUs have 2 pins
  – INT : Interrupt
  – NMI : Non maskable – for very critical signals
• How to support more than two interrupts?
8259 Programmable Interrupt Controller

- 8259 (Programmable interrupt controller) relays up to 8 interrupts to the CPU.
- Devices raise interrupts by an ‘interrupt request’ (IRQ).
- The CPU acknowledges and queries the 8259 to determine which device interrupted.
- Priorities can be assigned to each IRQ line.
- 8259s can be cascaded to support more interrupts.
Interrupts in legacy CPUs

- 15 IRQs (IRQ0 to IRQ15), so 15 possible devices
- Interrupt types
  - Edge
  - Level
- Limitations
  - Limited IRQs
  - Spurious interrupts by 8259
    - Eg. de-asserted IRQ before IRQA
Edge vs Level Interrupts

• **Level triggered Interrupt**: as long as the IRQ line is asserted you get an interrupt.
  – Level interrupt still active even after interrupt service is complete
  – Stopping interrupt would require physically deactivating the interrupt

• **Edge triggered Interrupt**: Exactly one interrupt occurs when IRQ line is asserted
  – To get a new interrupt, the IRQ line must become inactive and then become active again

• **Active high interrupts**: When asserted, IRQ line is high (logic 1)
Edge vs Level Interrupts
(the crying baby… an analogy)

• Level triggered interrupt:
  – when baby cries (interrupt) stop what you are doing and feed the baby
  – then put the baby down
  – if baby still cries (interrupt again) continue feeding

• Edge triggered interrupt
  – eg. *Baby cry monitor*, where light turns red when baby is crying. The light is turned off by a push button switch
    • if baby cries and stops immediately you see that the baby has cried (level triggered would have missed this)
    • if the baby cries and you press the push button, the light turns off, and remains off even though the button is pressed

http://venkateshabbarapu.blogspot.in/2013/03/edge-triggered-vs-level-triggered.html
Spurious Interrupts

Consider the following Sequence

1. Device asserts level triggered interrupt
2. PIC tells CPU that there is an interrupt
3. CPU acknowledges and waits for PIC to send interrupt vector
4. However, device de-asserts interrupt. What does the PIC do?

This is a spurious interrupt

To prevent this, PIC sends a fake vector number called the spurious IRQ. This is the lowest priority IRQ.
Advanced Programmable Interrupt Controller (APIC)

- External interrupts are routed from peripherals to CPUs in multi processor systems through APIC
- APIC distributes and prioritizes interrupts to processors
- Interrupts can be configured as edge or level triggered
- Comprises of two components
  - Local APIC (LAPIC)
  - I/O APIC
- APICs communicate through a special 3-wire APIC bus.
  - In more recent processors, they communicate over the system bus
LAPIC and I/OAPIC

• **LAPIC**:
  – Receives interrupts from I/O APIC and routes it to the local CPU
  – Can also receive local interrupts (such as from thermal sensor, internal timer, etc)
  – Send and receive IPIs (Inter processor interrupts)
    • IPIs used to distribute interrupts between processors or execute system wide functions like booting, load distribution, etc.

• **I/O APIC**
  – Present in chipset (north bridge)
  – Used to route external interrupts to local APIC
I/O APIC Configuration in xv6

- IO APIC: **82093AA I/O APIC**
- Function: `ioapicinit` (in `ioapic.c`)
- All interrupts configured during boot up as
  - Active high
  - Edge triggered
  - Disabled (interrupt masked)
- Device drivers selectively turn on interrupts using `ioapicenable`
  - Three devices turn on interrupts in xv6
    - UART (`uart.c`)
    - IDE (`ide.c`)
    - Keyboard (`console.c`)

ref: `ioapic.c` [73], (http://www.intel.com/design/chipsets/datashts/29056601.pdf)
1. Enable LAPIC and set the spurious IRQ (i.e. the default IRQ)

2. Configure Timer
   - Initialize timer register (10000000)
   - Set to periodic

ref: lapic.c (lapicinit) (7151)
What happens when there is an Interrupt?

1. Device asserts IRQ of I/OAPIC
   - I/O APIC transfer interrupt to LAPIC
   - LAPIC asserts CPU interrupts
     - After current instruction completes
       - CPU senses interrupt line and obtains IRQ number from LAPIC

2. Switch to kernel stack if necessary

By device and APICs

Done by CPU automatically

Done in software

 Either special 3 wire APIC bus or system bus
What more happens when there is an Interrupt?

3. Basic program state saved
   - X86 saves the SS, ESP, EFLAGS, CS, EIP, error code on stack (restored by \textit{iret} instruction). Suspends current task.

4. Jump to interrupt handler
   - How does hardware find the OS interrupt handler?

5. Interrupt handler (top half)
   - Just do the important stuff like...
     - respond to interrupt
     - more storing of program state
     - schedule the bottom half
     - IRET
   - Restore flags and registers saved earlier. Restore running task.

6. Return from interrupt
   - The work horse for the interrupt

7. Interrupt handler (bottom half)

software

---

X86 saves the SS, ESP, EFLAGS, CS, EIP, error code on stack (restored by \textit{iret} instruction). Suspends current task.
Stacks

- Each process has two stacks
  - a user space stack
  - a kernel space stack
Switching Stack
(to switch or not to switch)

• When event occurs OS executes
  – If executing user process, privilege changes from low to high
  – If already in OS no privilege change

• Why switch stack?
  – OS cannot trust stack (SS and ESP) of user process
  – Therefore stack switch needed only when moving from user to kernel mode

• How to switch stack?
  – CPU should know locations of the new SS and ESP.
  – Done by task segment descriptor

Done automatically by CPU
To Switch or not to Switch

- Executing in Kernel space
  - No stack switch
  - Use the current stack

- Executing in User space
  - Switch stack to a kernel switch
How to switch stack?

Task State Segment

- Specialized segment for hardware support for multitasking
- TSS stored in memory
  - Pointer stored as part of GDT
  - Loaded by instruction: `ltr(SEG_TSS << 3)` in switchuvvm()
- Important contents of TSS used to find the new stack
  - **SS0**: the stack segment (in kernel)
  - **ESP0**: stack pointer (in kernel)

ref: (switchuvvm) ([18],1873), taskstate ([08],0850)
Saving Program State

Why?

• Current program being executed must be able to resume after interrupt service is completed
Saving Program State

When no stack switch occurs
use existing stack

- ESP before
- EFLAGS
- CS
- EIP
- Error Code
- ESP after

SS : No change
ESP : new frame pushed

Error code is only for some exceptions. Contains additional information.

When stack switch occurs
also save the previous SS and ESP

Interrupted Procedure Stack (in user space)

- ESP before
- SS
- ESP
- EFLAGS
- CS
- EIP
- Error Code
- ESP after

Procedure’s kernel stack

SS : from TSS (SS0)
ESP : from TSS (ESP0)
Finding the Interrupt/Exception Service Routine

- **IDT**: Interrupt descriptor table
  - Also called Interrupt vectors
  - Stored in memory and pointed to by IDTR
  - Conceptually similar to GDT and LDT
  - Initialized by OS at boot

\[
\text{Selected Descriptor} = \text{Base Address} + (\text{Vector} \times 8)
\]

Done automatically by CPU
Interrupt Gate Descriptor

- Points to offset in the segment which contains the interrupt handler (higher order bits).
- Points to offset in the segment which contains the interrupt handler (lower order bits).
- Points to a segment descriptor for executable code in the GDT.
- 1 Segment present
- 0 Segment absent
- Privilege level

ref: SETGATE (0921), gatedesc (0901)
Getting to the Interrupt Procedure

IDTR : pointer to IDT table in memory

Interrupt Vector

GDT or LDT

Segment Selector

IDT

Interrupt or Trap Gate

Offset

Destination Code Segment

Interrupt Procedure

Base Address

IDTR

64 bytes

(obtained from either the PIC or APIC)

Done automatically by CPU
Setting up IDT in xv6

- Array of 256 gate descriptors (idt)
- Each idt has
  - Segment Selector : SEG_KCODE
    - This is the offset in the GDT for kernel code segment
  - Offset : (interrupt) vectors (generated by Script vectors.pl)
    - Memory addresses for interrupt handler
    - 256 interrupt handlers possible
- Load IDTR by instruction lidt
  - The IDT table is the same for all processors.
  - For each processor, we need to explicity load lidt (idtinit())

ref : tvinit() (3317) and idtinit() in trap.c
Interrupt Vectors in xv6

vector0
vector1
vector2
---
---
vector i
---
vector255

vector i:
  push 0
  push i
  Jmp alltraps

Error code:
Hardware pushes error
Code for some exceptions. For others, xv6 pushes 0.

ref: vectors.s [generated by vectors.pl (run $perl vectors.pl)] ([32])
**alltraps**

Creates a trapframe
Stack frame used for interrupt

Setup kernel data and code segments

Invokes trap (3350 [33])

ref: trapasm.S [32] (alltraps), trap.c [33] (trap())
trapframe

By hardware
Pushed by hardware or software

by software

argument for trap
(pointer to this trapframe)

ref : struct trapframe in x86.h (0602 [06])
trapframe struct

```
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint eosp; // 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  );
```
Interrupt Handlers

• Typical Interrupt Handler
  – Save additional CPU context (written in assembly)
    (done by alltraps in xv6)
  – Process interrupt (communicate with I/O devices)
  – Invoke kernel scheduler
  – Restore CPU context and return (written in assembly)
Interrupt Latency

Interrupt latency can be significant.
Importance of Interrupt Latency

• **Real time systems**
  – OS should ‘guarantee’ interrupt latency is less than a specified value

• **Minimum Interrupt Latency**
  – Mostly due to the interrupt controller

• **Maximum Interrupt Latency**
  – Due to the OS
  – Occurs when interrupt handler cannot be serviced immediately
    • Eg. when OS executing atomic operations, interrupt handler would need to wait till completion of atomic operations.
Atomic Operations

Global variable:
int x;

for(i = 0; i < 1000; ++i)
x++

x = x * 5

Value of x depends on whether an interrupt occurred or not!

Solution: make the part of code atomic (i.e. disable interrupts while executing this code)
Nested Interrupts

- Typically interrupts disabled until handler executes
  - This reduces system responsiveness
- To improve responsiveness, enable Interrupts within handlers
  - This often causes nested interrupts
  - Makes system more responsive but difficult to develop and validate
- **Interrupt handler approach**: design interrupt handlers to be small so that nested interrupts are less likely
Small Interrupt Handlers

• Do as little as possible in the interrupt handler
  – Often just queue a work item or set a flag
• Defer non-critical actions till later
Top and Bottom Half Technique (Linux)

- **Top half**: do minimum work and return from interrupt handler
  - Saving registers
  - Unmasking other interrupts
  - Restore registers and return to previous context

- **Bottom half**: deferred processing
  - eg. Workqueue
  - Can be interrupted
Interrupt Handlers in xv6

- vectors.S
- alltraps
  (alltraps.S)
- trap
  (trap.c)

Interrupt s specific handler
Example (Keyboard Interrupt in xv6)

- Keyboard connected to second interrupt line in 8259 master
- Mapped to vector 33 in xv6 (T_IRQ0 + IRQ_KBD).
- In function trap, invoke keyboard interrupt (kbdintr), which is redirected to consoleintr
Keyboard Interrupt Handler

- **consoleintr (console.c)**
- **get pressed character (kbdgetc (kbd.c))**
  - talks to keyboard through specific predefined io ports
- **Service special characters**
- **Push into circular buffer**
System Calls and Exceptions
Events

- Interrupts
  - Hardware Interrupts
  - Software Interrupts
- Exceptions
Hardware vs Software Interrupt

- A device (like the PIC) asserts a pin in the CPU

- An instruction which when executed causes an interrupt
Software Interrupt

Software interrupt used for implementing system calls

– In Linux INT 128, is used for system calls
– In xv6, INT 64 is used for system calls
Example (write system call)

```
printf("%s", str);
```
System call processing in kernel

Almost similar to hardware interrupts

- INT 64
  - vectors.S
  - alltraps (alltraps.S)
  - trap (trap.c)
  - syscall (syscall.c)

if vector = 64

Executes the System calls

Back to user process

0
# System Calls in xv6

<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork()</td>
<td>Create process</td>
</tr>
<tr>
<td>exit()</td>
<td>Terminate current process</td>
</tr>
<tr>
<td>wait()</td>
<td>Wait for a child process to exit</td>
</tr>
<tr>
<td>kill(pid)</td>
<td>Terminate process pid</td>
</tr>
<tr>
<td>getpid()</td>
<td>Return current process’s id</td>
</tr>
<tr>
<td>sleep(n)</td>
<td>Sleep for n seconds</td>
</tr>
<tr>
<td>exec(filename, *argv)</td>
<td>Load a file and execute it</td>
</tr>
<tr>
<td>sbrk(n)</td>
<td>Grow process’s memory by n bytes</td>
</tr>
<tr>
<td>open(filename, flags)</td>
<td>Open a file; flags indicate read/write</td>
</tr>
<tr>
<td>read(fd, buf, n)</td>
<td>Read n bytes from an open file into buf</td>
</tr>
<tr>
<td>write(fd, buf, n)</td>
<td>Write n bytes to an open file</td>
</tr>
<tr>
<td>close(fd)</td>
<td>Release open file fd</td>
</tr>
<tr>
<td>dup(fd)</td>
<td>Duplicate fd</td>
</tr>
<tr>
<td>pipe(p)</td>
<td>Create a pipe and return fd’s in p</td>
</tr>
<tr>
<td>chdir(dirname)</td>
<td>Change the current directory</td>
</tr>
<tr>
<td>mkdir(dirname)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>mknod(name, major, minor)</td>
<td>Create a device file</td>
</tr>
<tr>
<td>fstat(fd)</td>
<td>Return info about an open file</td>
</tr>
<tr>
<td>link(f1, f2)</td>
<td>Create another name (f2) for the file f1</td>
</tr>
<tr>
<td>unlink(filename)</td>
<td>Remove a file</td>
</tr>
</tbody>
</table>
System Call Number

System call number used to distinguish between system calls

Based on the system call number, function syscall invokes the corresponding syscall handler

mov x, %eax
INT 64

ref: syscall.h, syscall() in syscall.c
Prototype of a typical System Call

```c
int system_call( resource_descriptor, parameters)
```

- return is generally ‘int’ (or equivalent) sometimes ‘void’
- int used to denote completion status of system call sometimes also has additional information like number of bytes written to file
- What OS resource is the target here?
  - For example a file, device, etc.
  - If not specified, generally means the current process
- System call specific parameters passed.
  - How are they passed?
Passing Parameters in System Calls

• Passing parameters to system calls **not similar** to passing parameters in function calls
  – Recall stack changes from user mode stack to kernel stack.

• Typical Methods
  – Pass **by Registers** (eg. Linux)
  – Pass **via user mode stack** (eg. xv6)
    • Complex
  – Pass via a **designated memory region**
    • Address passed through registers
Pass By Registers (Linux)

- System calls with fewer than 6 parameters passed in registers
  - %eax (sys call number), %ebx, %ecx, %esi, %edi, %ebp
- If 6 or more arguments
  - Pass pointer to block structure containing argument list
- Max size of argument is the register size (e.g., 32 bit)
  - Larger pointers passed through pointers
Pass via User Mode Stack (xv6)

User process
push param1
push param2
push param3
mov sysnum, %eax
int 64

User stack
param1
param2
param3

trapframe
SS
ESP
EFLAGS
CS
EIP
Error Code
Trap Number
ds
es...
eax
ecx...
esi
edi
ESP

ESP pushed by hardware contains user mode stack pointer

Points to trapframe

proc entry for process

ref : sys_open (sysfile.c), argint, fetchint (syscall.c)
Returns from System Calls

User process
push param1
push param2
push param3
mov sysnum, %eax
int 64
....

Return value
register EAX

trapframe
SS
ESP
EFLAGS
CS
EIP
Error Code
Trap Number
ds
es
...
eax
ecx
...
esi
edi
ESP
(empty)
in system call
move result to eax in trap frame

Automatically restored by hardware while returning to user process
Events

- Interrupts
  - Hardware Interrupts
  - Software Interrupts
- Exceptions
Exception Sources

- **Program-Error Exceptions**
  - Eg. divide by zero

- **Software Generated Exceptions**
  - Example INTO, INT 3, BOUND
  - INT 3 is a break point exception
  - INTO overflow instruction
  - BOUND, Bound range exceeded

- **Machine-Check Exceptions**
  - Exception occurring due to a hardware error (eg. System bus error, parity errors in memory, cache memory errors)

Microsoft Windows: Machine check exception
Exception Types

- Exceptions in the user space vs kernel space
Faults

Exception that generally can be corrected. Once corrected, the program can continue execution.

Examples:
Divide by zero error
Invalid Opcode
Device not available
Segment not present
Page not present
Traps

Traps are reported immediately after the execution of the trapping instruction.

Examples:
Breakpoint
Overflow
Debug instructions
Aborts

Severe unrecoverable errors

Examples

Double fault: occurs when an exception is unhandled or when an exception occurs while the CPU is trying to call an exception handler.

Machine Check: internal errors in hardware detected. Such as bad memory, bus errors, cache errors, etc.