Operating Systems (234123)

Interrupts & signals

Dan Tsafrir (29/5/2017)
Partially based on slides by Hagit Attiya
HW => OS (Section 1.1 – 1.3 in Feitelson’s “OS notes”)

INTERRUPTS
Reminder: the OS is a reactive “program”

• Usually, the OS spends most of its time
  – Waiting for events to occur (e.g., a keyboard key press)

• Whenever an event happens
  – OS handles quickly & gives CPU back to running process (if such exists)

• The goal of the OS
  – Run as little as possible, handle events as quickly as possible, and let apps run most of the time

• OS pseudo code
  – (Doesn’t look like the real thing, but does capture the essence)
    ```
    while( 1 ) {
      event_t e = wait_for_event(); /*sleep*/
      handle_event(e);
      resume_process() }
    ```

• But what are these events really?
  – Interrupts! That is it. There’s *nothing* else
Terminology note

• Our “interrupts” are Intel’s “interrupts and exceptions” such that
  – Intel’s “exceptions” ⇔ our “internal (synchronous) implicit interrupts”
  – Intel’s “interrupts” ⇔ all the rest
Who generates interrupts?

- **Two sources (two interrupt “types”)**
  - **External / asynchronous**
    - Triggered by devices external to the core’s compute circuitry
    - “Asynchronous” because they can occur at any time
    - (“between CPU operations”)
  - **Internal / synchronous**
    - Triggered by the core’s compute circuitry (CPU instructions)
      - Well, actually, it’s the software that runs on the core...
      - Which is why such interrupts are also sometimes called
        >>> “Software interrupts”
    - (“while a CPU operation occurs”)

OS (234123) - interrupts & signals 5
## External (asynchronous) interrupts

<table>
<thead>
<tr>
<th>external devices</th>
<th>when they want to say, for example,</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIC</td>
<td>packet(s) arrived! packet(s) sent! (DMA)</td>
<td>network interface controller (or network interface card)</td>
</tr>
<tr>
<td>disk controller</td>
<td>finished reading a block! finished writing a block! (DMA)</td>
<td>HDD, SDD, floppy, DVD, ... (called “block device”, as opposed to byte-addressable)</td>
</tr>
<tr>
<td>periodic clock</td>
<td>another tick has elapsed!</td>
<td></td>
</tr>
<tr>
<td>one-shot timer</td>
<td>timer expired!</td>
<td>clock</td>
</tr>
<tr>
<td>another core</td>
<td>I want you to do something!</td>
<td>IPI = inter-processor interrupt (recall that we’ve seen it used in the Linux 2.4 scheduler lecture)</td>
</tr>
<tr>
<td>keyboard</td>
<td>key was pressed!</td>
<td></td>
</tr>
<tr>
<td>mouse</td>
<td>I was moved/clicked!</td>
<td></td>
</tr>
<tr>
<td>microphone</td>
<td>I heard something!</td>
<td></td>
</tr>
<tr>
<td>many more...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Internal (synchronous) interrupts – explicit

- The software that runs on the core can generate interrupts
- Some interrupts are deliberate = explicitly invoked by SW
  - Execute the ‘int’ operation, which traps into the OS
    - System calls to request some service from OS
      - Same interrupt for all system calls (why?)
    - Debugging (software-)breakpoints
  - Ptrace-ing
    - Allows user-land to implement single-step debugging, see
      - http://en.wikipedia.org/wiki/Ptrace and
      - http://linux.die.net/man/2/ptrace
Internal (synchronous) interrupts – implicit

• Some interrupts occur implicitly, indicating an exception condition
  – The processor can’t complete the current instruction for some reason
  – So it transfers responsibility to the OS

• There are 2 types of exceptions:
  – An error condition tells the OS that the app did something illegal
    • Divide by 0, execute privileged instruction, ...
    • By default, OS kills such misbehaving processes
      – By sending a signal & setting default signal-handler that kills the process (discussed later)
  – A temporary problem
    • Memory currently unmapped (next lecture)
    • This is the OS’s fault; it can & should handle it
  – Who decides if it’s temporary?
    • OS
Interrupt handling (x86/Linux)

• Each interrupt is associated with a
  – Number (called “interrupt vector”)
    • There are 256 interrupt vectors, numbered 0...255
  – Handling routine (called “interrupt handler”)
    • Need different routine because, for example, keyboard key press is handled differently than incoming network packet
    • Can associate several devices with same interrupt vector when some additional info is made available to OS to identify source device

• Handlers pointed to by array[256] of pointer-to-functions
  – Array[j] holds pointer to the interrupt handler of vector j
  – When an interrupt fires, HW invokes the corresponding function
  – Array content is initially set by BIOS
    • (For example, to be able to use keyboard before OS is up)
  – Later, at boot time, array is reinitialized by OS with its own routines
Interrupt handling (x86/Linux)

• **The interrupt handler**
  – Save state of currently running process P
  – Execute handler
  – Return to user mode, resuming either P or another process
    • Q: When will it be another process? When the same?

• **Context type**
  – If asynchronous interrupts, kernel is said to be in “interrupt context”
    • It doesn’t directly serve the process that has just been stopped
  – If synchronous, kernel is in “process context”
    • It needs to provide some service to the process that invoked it (whether explicitly or implicitly)
Interrupt handlers are not schedulable entities

• **When an interrupt fires**
  – Its handler runs immediately (scheduler not involved!), that is
  – Unless the interrupt is currently “masked”
  – In which case handler will run when the interrupt is unmasked

• **Interrupt masking**
  – An interrupt handlers constitutes a different (OS) thread of execution
  – So need to worry about synchronization (races, mutex, deadlocks, …)
  – Thus, HW supports mask[256] bitmap, and it blocks those with set bits

• **What happens to masked interrupts?**
  – If interrupt vector K should fire but is masked
  – x86 HW remembers at most 2 instances of K (the rest get lost, if exist)

• **Since handlers run immediately when their interrupts fire**
  – The OS scheduler doesn’t control them even if what’s currently running is “more important”, so…
“Top half” & “bottom half”

• Handling might take a long time
  – But we want to allow OS to decide if it has more important stuff to do
  – Also, x86 HW waits for OS to say “EOI” (end of interrupt) before it allows other, equal or lower priority interrupts to fire

• Solution: split the handler into two
  – Fast/Hard/First-Level Interrupt handler (FLIH)
    • What we must do when receiving an interrupt, and nothing else
    • Not interruptible => as small and as quick as possible
  – Slow/Soft/Second-Level interrupt handler (SLIH)
    • All the rest (lower priority)
    • Interruptible; will be executed later by some kernel thread

• In Linux
  – FLIHs = “top-halves”
  – SLIHs = “bottom-halves”
Example

• When receiving a packet from the network

  – Possible top half
    (not schedulable, occurs immediately)
    • Put a pointer to the received packet in a queue for later processing
    • Give the NIC a new free memory buffer, such that it will have room to save subsequent incoming packets

  – Possible bottom half
    (will happen later when the scheduler decides)
    • Hand packet to network stack
    • Copy the content of the packet to the memory space of the corresponding user application
    • Notify the user app that a new packet has arrived
Resume an interrupted process - where?

- If asynchronous interrupt (external)
  - Resume in the next process instruction,
  - Right after the last instruction that was executed

- If system call (synchronous=internal, explicit)
  - Likewise, resume in the next instruction
  - Right after the last instruction that triggered the trap

- If exception (synchronous=internal, implicit)
  - Resume in the same instruction that the HW failed to execute
    - Called “restartable instruction”
    - This makes sense for temporal problems (unmapped memory)
      - Thus, OS doesn’t send signal to app
  - It makes less sense for error conditions (divide by 0 etc.)
    - Thus, signal is sent to app (default signal handler kills app)
    - If app changed the default & doesn’t fix things => endless loop!
Interrupt handling in multicores – where?

- **Internal**=synchronous interrupts (system call, divide by 0, ...)
  - Each core handles on its own

- **External**=async interrupts (generated by external devices)
  - Can do static partition of interrupts between cores
    - E.g., all interrupts go to core 0
    - Or, all interrupts in range x–y go to core j
  - Can do dynamic partition of interrupts between cores
    - E.g., round robin
    - Or, send interrupt to least loaded core
    - ...

- **Homework**
  - In a Linux shell do
    - watch –n1 cat /proc/interrupts
Polling

• **With I/O-intensive devices (like 10-100Gb/s NICs)**
  – OS might be overwhelmed by very many interrupts, telling it what it already knows: that there are lots of incoming packets to handle
  – This is called an “interrupt storm”

• **Handling that many interrupts consumes lots of cycles**
  – Cycles that could be used more productively for actually receiving & processing the packets

• **Polling is an alternative to interrupt-based processing**
  – OS turns off interrupts and instead polls the NIC every once in a while
  – Effective when OS is mostly busy handling the packets

• **Cons**
  – Polling too frequently might waste CPU cycles if not enough packets
  – Polling too infrequently increases latency & may result in packet loss

• **Compromise**
  – Interrupt coalescing (next slide)
Interrupt coalescing

• By HW
  – Devices are mindful not to create “interrupt storm”, so
  – By default they typically coalesce several interrupts into one
  – For example, NICs typically let a few microseconds to elapse before emitting another interrupt
  – (Sometimes the coalescing interval is configurable by SW)

• By SW
  – Example: Linux’s NAPI (“new API” :-/)
  – Linux kernel uses the interrupt-driven mode by default
  – Switches to polling mode when the flow of incoming packets exceeds a certain threshold
  – See http://en.wikipedia.org/wiki/New_API for more details
DMA & interrupts

– Recall: DMA = direct memory access
– When an I/O device accesses memory directly, without CPU involvement
– OS requests I/O devices to transfer data
– Devices do it on their own, with DMA operations (read or write)
– Conceptually (= when disregarding coalescing), when a DMA operation is finished, devices fire an interrupt
  • Not only when receiving (reading) data
  • But also when sending (writing) data
  • Because need to notify OS when the send buffer becomes free
Example – blocking read from disk

• process
  => read()
  => OS
  => process is suspended (give CPU to others)
  => find blocks from which data should be read
  => initiate DMA operation
  => DMA done
  => device fires interrupt to notify DMA is done
  => OS top-half adds relevant bottom half to relevant list
  => later, bottom half scheduled to run & copies data to user
  => makes process runnable
  => process scheduled and can now read the data
OS => process

POSIX SIGNALS
What are signals & signal handlers

• Signal = notification “sent” to a process
  – To asynchronously notify it that some event occurred

• Upon receiving a signal
  – The process stops whatever it is doing & handles it

• Default signal handling action
  – Either die or ignore (depends on the type of the signal)

• Process may configure how it handles any signal type
  – Every signal can have a different handler
  – Except for two (discussed shortly)

• Signals have names and numbers (standardized by POSIX)
  – Do ‘man 7 signal’ in shell/Google for a listing/explanation of all signals
  – **HOMEWORK**: take a few minutes to quickly survey all signals
Silly example

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sigfpe_handler(int signum) {
    fprintf(stderr,"I divided by zero (sig=%d)!\n", signum); // prints SIGFPE in this case
    exit(EXIT_FAILURE); // what happens if not exiting?
}

int main() {
    signal(SIGFPE, sigfpe_handler);
    int x = 1/0;
    return 0;
}
Another silly example

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sigint_handler(int signum) {
    printf("I'm ignoring your ctrl-c!\n");
}

int main() {
    // when pressing ctrl-c in shell
    // => SIGINT is delivered to foreground process
    signal(SIGINT, sigint_handler);
    for(;;) { /* endless look*/ }
    return 0;
}
```
Another silly example

<0>dan@csa:~$ ./a.out

# here I clicked ctrl-c (=> deliver SIGINT)
I'm ignoring your ctrl-c!
# here I clicked ctrl-c again (=> deliver SIGINT)
I'm ignoring your ctrl-c!
# here I clicked ctrl-z => deliver SIGSTP, can’t ignore

[1]+ Stopped ./a.out

<148>dan@csa:~$ ps

    PID TTY          TIME CMD
10148 pts/19   00:00:00 bash
21709 pts/19   00:00:12 a.out
21710 pts/19   00:00:00 ps

<0>dan@csa:~$ kill -9 21709   # 9=SIGKILL, can’t ignore

[1]+ Killed ./a.out

<0>dan@csa:~$
Another silly example

<0>dan@csa:~$ ./a.out
# here I clicked ctrl-c (=> deliver SIGINT)
I'm ignoring your ctrl-c!
# here I clicked ctrl-c again (=) deliver SIGINT)
I'm ignoring your ctrl-c!
# here I clicked ctrl-z => deliver SIGSTP, can’t ignore
[1]+ Stopped                 ./a.out
<148>dan@csa:~$

Note that when I click ctrl-c, the OS gets an interrupt from the keyboard, which it then “translates” into a signal delivered to the relevant process

<0>dan@csa:~$ ps
  PID  TTY      TIME CMD
 10148 pts/19   00:00:00 bash
 21709 pts/19   00:00:12 a.out
 21710 pts/19   00:00:00 ps
<0>dan@csa:~$ kill -9 21709  # 9=SIGKILL, can’t ignore
[1]+ Killed                  ./a.out
<0>dan@csa:~$
Notice

• ‘kill’ utility receives as argument any valid signal number
  – Not just 9 (=SIGKILL)

• There are two signals that a process can’t ignore
  – SIGKILL (terminated the receiving process)
  – SIGSTOP (suspends the receiving process)

• Is the affect of SIGSTOP reversible?
  – Yes, when you send to the process SIGCONT

• Actually, SIGCONT can’t be ignored either
  – But process can set a handler for it, which will be invoked immediately
    when the process gets hit by the SIGCONT and is resumed as a result

• What can you do with SIGSTOP/SIGCONT?
Example: ask running daemon how much work it did

```c
int g_count=0;

void do_work() { for(int i=0; i<10000000; i++); }

void sigusr_handler(int signum) {
    printf("Work done so far: %d\n", g_count);
}

int main() {
    signal(SIGUSR1,sigusr_handler);
    for(;;) { do_work(); g_count++; }
    return 0;
}
```
Example: ask running daemon how much work it did

```bash
<0>dan@cse:~$ ./a.out &
[1] 23998
# note: shell’s kill also accepts strings as signals
<0>dan@cse:~$ kill -USR1 23998
Work done so far: 626
<0>dan@cse:~$ kill -USR1 23998
Work done so far: 862
<0>dan@cse:~$ kill -USR1 23998
Work done so far: 1050
<0>dan@cse:~$ kill -9 23998
[1]+ Killed ./a.out
```
Some signals

- **SIGSEGV, SIGBUSS, SIGILL, SIGFPE**
  - ILL = illegal instruction (trying to invoke privileged instruction)
  - SEGV = segmentation violation (illegal memory reference)
  - BUS = bus error (same as SEGV, more or less)
  - FPE = floating point exception (despite name, actually *all* arithmetic errors, nor just floating point; example: divide by zero)
  - Notice that these are driven by the associated (HW) interrupts
    - The OS gets the associated interrupt
    - The OS interrupt handler sees to it that the misbehaving process gets the associated signal
    - The default signal handler for these signals: core dump + die

- **SIGCHLD**
  - Parent gets it whenever fork()ed child terminates or is SIGSTOP-ed

- **SIGALRM**
  - Get a signal after some specified time
  - Set by system calls: alarm() & setitimer()
Some signals

• **SIGTRAP**
  – When debugging / single-stepping a process
  – E.g., can be delivered upon each instruction, or in a breakpoint

• **SIGUSR1, SIGUSR2**
  – User decides the meaning (e.g., see our daemon example)

• **SIGXCPU**
  – Delivered when a process used up more CPU than its soft-limit allows
  – Soft/hard limits are set by the system call: `setrlimit()`
  – Soft-limits warn the process it’s about to exceed the hard-limit
  – Exceeding the hard-limit => SIGKILL will be delivered

• **SIGPIPE**
  – Write to pipe with no readers
Some signals

• **SIGIO**
  – Can configure file descriptors such that a signal will be delivered whenever some I/O is ready
  – Typically makes sense when also configuring the file descriptors to be “non blocking”
    • E.g., when read()ing from a non-blocking file descriptor, the system call immediately returns to user if there’s currently nothing to read

• **And a few more**
  – man 7 signal
Signal vs. interrupts

• **When do they occur**
  – Both are asynchronous, but signals are invoked only when returning from kernel to user, and interrupts can really fire anytime

• **Who (i) triggers them and (ii) defines their semantics**
  – Interrupts: hardware – devices (async) or cores (sync)
  – Signals: software (OS) – HW is unaware

• **Who handles them, who blocks them**
  – Interrupts: OS
  – Signals: processes
Signal system calls

- **int kill(pid_t *pid, int *sig)**
  - (Not the shell utility, the actual system call)
  - Allow a process to send a signal to another process (or to itself)
  - man 2 kill – [http://linux.die.net/man/2/kill](http://linux.die.net/man/2/kill)

- **int sigprocmask(int *how, const sigset_t *set, sigset_t *oldset)**
  - Since signals happen asynchronously they might lead to race conditions
  - This system call allows programmers to block/unblock signals
  - Similarly to how OS disables/enables interrupt
  - how = SIG_BLOCK (+=), SIG_UNBLOCK (-=), SIG_SETMASK (=)
  - man 2 sigprocmask – [http://linux.die.net/man/2/sigprocmask](http://linux.die.net/man/2/sigprocmask)

- **int sigaction(int *signum,**
  const struct sigaction *act, struct sigaction *oldact)**
  - Finer control over how signals operate
  - man 2 sigaction – [http://linux.die.net/man/2/sigaction](http://linux.die.net/man/2/sigaction) (read it!)
• `ssize_t read(int fd, void *buf, size_t count);`
  – What happens if getting signal while read()ing?
  – The read system call returns -1, and it sets the global variable ‘errno’
    to hold EINTR