Systems@ETH Zürich

# Hardware Interrupts, Traps and Clock Interrupt

Lions' Commentary on UNIX 6th Edition

Chapters 9-11

presented by Reto Mock

# Outline

- A short history of interrupts
- Overview: Interrupts, Traps, Signals
- Interrupt handling
- The clock interrupt
- Traps
- What's different today?
- Q&A

# A short history of interrupts

- Interrupts ..
  - were developed for exception handling
  - were later applied to I/O events
  - are today also used for protection and memory management

- 1953: UNIVAC 1103 was the first computer system to use interrupts
- 1954: IBM 650 first used interrupt masking
- 1954: NBS DYSEAC introduced I/O interrupts
- 1957: Lincoln Labs TX2 with innovation of interrupt vectors
- 1961: Manchester Atlas introduced page faults

# Dijkstra's comments on interrupts

- Interrupts [..] enabled the implementation of rapid reaction to changed external circumstances without paying the price of a lot of processor time lost in unproductive waiting
- The disadvantage was that operation systems had to ensure correct execution of the various computations despite the unpredictability [..] when interrupts would take place [..]
- [..] caused endless headaches for [..] operation system designers
- The interrupt mechanism turned the computer into a nondeterministic machine [..], how could we control such a beast?

# Overview: Three mechanisms to interrupt normal instruction execution

- Hardware interrupts
  - events external to CPU
  - e.g. clock, peripherals
- Traps
  - events internal to CPU
  - e.g. illegal instruction (segmentation fault, floating point error), system call
- Software interrupts a.k.a. signals
  - e.g. SIGKIL, SIGFPT, SIGSEG
  - will be discussed by David Flatz on April 20th

# Hardware mechanism

- When an interrupt occurs
  - CPU saves the current processor status word (PS) and program count (PC)
  - PS and PC are reloaded from low area main memory
    → vector location of the certain interrupt which occurred
  - Interrupt handler procedure is executed
  - After return the original PS and PC are restored

# Interrupt handlers

| Procedure | Device | When discussed |
|---|---|---|
| clock (3725) | System Clock | In a few moments by me |
| rkintr (5451) | RK11 Disk Driver | April 20th by Max Urech |
| klxint (8070) | KL11 Interactive Terminal (Transmitter) | May 11th by Marco Calderara |
| klrint (8078) | KL11 Interactive Terminal (Receiver) | May 11th by Marco Calderara |
| pcrint (8719) | PC11 Paper Tape Reader | May 11th by Dominik Menzi |
| pcpint (8739) | PC11 Paper Tape Puncher | May 11th by Dominik Menzi |
| lpint (8976) | LP11 Line Printer | May 11th by Dominik Menzi |

# Interrupt vector locations and priorities

- Bits 7 to 5 of PS determine processor priority (0-7)

Table 4-1
Processor Status Fields

| Bits | Function | Description | Refer to Paragraph |
|---|---|---|---|
| 15–14 | current mode | select the processor operating mode:<br>00 = kernel<br>01 = supervisor<br>10 = not used<br>11 = user | 4.3.3, 4.3.4, 4.5.1 |
| 13–12 | previous mode | holds the processor mode that was in effect prior to the last trap or interrupt, for use in communication (MT/FP) instructions | 4.3.3, 4.5.2 |
| 11 | register set | selects one of two register sets for general registers 0 through 5 | 4.3.1, 4.3.2, 4.5.3 |
| 10–8 | not used | | |
| 7–5 | priority | selects one of eight processor priority levels that control scheduling of interrupt service routines | 4.5.4 |
| 4 | trace bit | controls operation of a trap function used in program debugging | 4.5.5 |
| 3–0 | condition codes | used to store information about the value of the result of the last data operation | 4.5.6, 7.2 |

# Interrupt vector locations and priorities

- Bits 7 to 5 of PS determine processor priority (0-7)
- For each interrupt the HW defines
  - The interrupt vector location
  - And the interrupt priority
- The interrupt occurs immediately if this priority is higher than the current processor priority; otherwise it is deferred

| Interrupt vector locations (partial) | | |
|---|---|---|
| **Vector loc.** | **Device** | **Prio** |
| 070/ 074 | Paper Tape Input/Output | 4 |
| 100 | Line Clock | 6 |
| 220 | RK Disk Drive | 5 |

| Trap vector locations (partial) | | |
|---|---|---|
| **V.L.** | **Trap type** | **Prio** |
| 034 | Trap Instruction | 7 (8) |
| 244 | Floating Point Error | 7 (8) |
| 220 | Segmentation V. | 7 (8) |

# Clock Interrupt: Lets see what happens

As before:

- CPU saves the current processor status word (PS) and program count (PC)
- PS and PC are reloaded from low area main memory
  → vector location of the certain interrupt which occurred

| Interrupt vector locations (partial) | | |
|---|---|---|
| **Vector loc.** | **Device** | **Prio** |
| 070/ 074 | Paper Tape Input/Output | 4 |
| **100** | **Line Clock** | **6** |
| 220 | RK Disk Drive | 5 |

| Trap vector locations (partial) | | |
|---|---|---|
| **V.L.** | **Trap type** | **Prio** |
| 034 | Trap Instruction | 7 (8) |
| 244 | Floating Point Error | 7 (8) |
| 220 | Segmentation V. | 7 (8) |

# File: unix/low.s

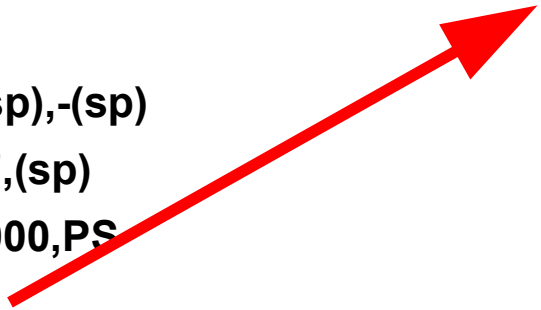
```
0502 br4   = 200
0503 br5   = 240
0504 br6   = 300
0505 br7   = 340
0506
0507 . = 0^.
0508     br      1f
0509     4
0510
0511 / trap vectors
0512     trap; br7+0.  / bus error
0513     trap; br7+1.  / illegal instruction
0514     trap; br7+2.  / bpt-trace trap
0515     trap; br7+3.  / iot trap
0516     trap; br7+4.  / power fail
0517     trap; br7+5.  / emulator trap
0518     trap; br7+6.  / system entry
0519
0520 . = 40^.
0521 .globl      start, dump
0522 1: jmp      start
0523     jmp     dump
0524
.......
0532
0533 . = 100^.
0534     kwlp; br6
0535     kwlp; br6
.......
0546 . = 240^.
0547     trap; br7+7.  / programmed interrupt
0548     trap; br7+8.  / floating point
0549     trap; br7+9.  / segmentation violation
```

# File: unix/low.s

```
0550
0551 ////////////////////////////////////////////////////////
0552 /          interface code to C
0553 ////////////////////////////////////////////////////////
0554
0555 .globl    call, trap
0556
0557 .globl    _klrint
0558 klin:     jsr     r0,call; _klrint
0559
0560 .globl    _klxint
0561 klou:     jsr     r0,call; _klxint
0562
0563 .globl    _pcrint
0564 pcin:     jsr     r0,call; _pcrint
0565
0566 .globl    _pcpint
0567 pcou:     jsr     r0,call; _pcpint
0568
0569 .globl    _clock
0570 kwlp:     jsr     r0,call; _clock
0571
0572
0573 .globl    _lpint
0574 lpou:     jsr     r0,call; _lpint
0575
0576 .globl    _rkintr
0577 rkio:     jsr     r0,call; _rkintr
0578
0579
0580
0581
0582
```

# File: unix/m40.s

```
0776 call:
0777    mov  PS,-(sp)
0778 1:
0779    mov  r1,-(sp)
0780    mfpi sp
0781    mov  4(sp),-(sp)
0782    bic  $!37,(sp)
0783    bit  $30000,PS
0784    beq  1f
0785    jsr  pc,*(r0)+
0786 2:
0787    bis  $340,PS
0788    tstb _runrun
0789    beq  2f
0790    bic  $340,PS
0791    jsr  ps,_swtch
0792    br   2b
```



```
0793 2:
0794    tst  (sp)+
0795    mtpi sp
0796    br   2f
0797 1:
0798    bis  $30000,PS
0799    jsr  pc,*(r0)+
```

# File: unix/m40.s

```
0776 call:
0777    mov  PS,-(sp)
0778 1:
0779    mov  r1,-(sp)
0780    mfpi sp
0781    mov  4(sp),-(sp)
0782    bic  $!37,(sp)
0783    bit  $30000,PS
0784    beq  1f
0785    jsr  pc,*(r0)+
0786 2:
0787    bis  $340,PS
0788    tstb _runrun
0789    beq  2f
0790    bic  $340,PS
0791    jsr  ps,_swtch
0792    br   2b
```

```
0793 2:
0794    tst  (sp)+
0795    mtpi sp
0796    br   2f
0797 1:
0798    bis  $30000,PS
0799    jsr  pc,*(r0)+
```

# File: unix/clock.c

```
3709 /*
3710   * clock is called straight from
3711   * the real time clock interrupt.
3712   *
.......
3724   */
3725 clock(dev, sp, r1, nps, r0, pc, ps)
3726 {
```

# The function clock(..)

- implements callouts
  - a kind of delayed function calls
- maintains user/system time
- maintains date
- wakes up processes which previously made a “sleep” system call at the appropriate time
- calls the scheduler

# Callouts (delayed function calls)

- Data structure

```
0260 struct      callo
0261 {
0262     int c_time;          /* incremental time */
0263     int c_arg;           /* argument to routine */
0264     int (*c_func)();     /* routine */
0265 } callout[NCALL];
```

Array callout:

| [0] | [1] | [2] | [3] | [4] |
|---|---|---|---|---|
| c_time | c_time | c_time | c_time | c_time |
| c_arg | c_arg | c_arg | c_arg | c_arg |
| *c_func | *c_func | *c_func | *c_func | *c_func |

- Register function to be called

```
3845 timeout(fun, arg, tim)
```

# File: unix/clock.c

```
3725 clock(dev, sp, r1, nps, r0, pc, ps)
.......
3742 /*
3743  * callouts
3744  * if done, just return
3745  * else update first non-zero time
3746  */
3747
3748 if(callout[0].c_func == 0)
3749         goto out;
3750 p2 = &callout[0];
3751 while(p2->c_time<=0 &&
                        p2->c_func!=0)
3752         p2++;
3753 p2->c_time--;
3754
```

```
3755 /*
3756  * if ps is high, just return
3757  */
3758
3759 if((ps&0340) != 0)
3760         goto out;
3761
3762 /*
3763  * callout
3764  */
3765
3766 spl5();
3767 if(callout[0].c_time <= 0) {
3768         p1 = &callout[0];
3769         while(p1->c_func != 0 &&
                                p1->c_time <= 0) {
3770                 (*p1->c_func)(p1->c_arg);
3771                 p1++;
3772         }
```

# File: unix/clock.c

```
3725 clock(dev, sp, r1, nps, r0, pc, ps)
.......
3742 /*
3743  * callouts
3744  * if done, just return
3745  * else update first non-zero time
3746  */
3747
3748 if(callout[0].c_func == 0)
3749         goto out;
3750 p2 = &callout[0];
3751 while(p2->c_time<=0 &&
                        p2->c_func!=0)
3752         p2++;
3753 p2->c_time--;
3754
```

```
3755 /*
3756  * if ps is high, just return
3757  */
3758
3759 if((ps&0340) != 0)
3760         goto out;
3761
3762 /*
3763  * callout
3764  */
3765
3766 spl5();
3767 if(callout[0].c_time <= 0) {
3768         p1 = &callout[0];
3769         while(p1->c_func != 0 &&
                                p1->c_time <= 0) {
3770                 (*p1->c_func)(p1->c_arg);
3771                 p1++;
3772         }
```

# File: unix/clock.c

```
3725 clock(dev, sp, r1, nps, r0, pc, ps)
.......
3742 /*
3743  * callouts
3744  * if done, just return
3745  * else update first non-zero time
3746  */
3747
3748 if(callout[0].c_func == 0)
3749         goto out;
3750 p2 = &callout[0];
3751 while(p2->c_time<=0 &&
                        p2->c_func!=0)
3752         p2++;
3753 p2->c_time--;
3754
```

```
3755 /*
3756  * if ps is high, just return
3757  */
3758
3759 if((ps&0340) != 0)
3760         goto out;
3761
3762 /*
3763  * callout
3764  */
3765
3766 spl5();
3767 if(callout[0].c_time <= 0) {
3768         p1 = &callout[0];
3769         while(p1->c_func != 0 &&
                                p1->c_time <= 0) {
3770                 (*p1->c_func)(p1->c_arg);
3771                 p1++;
3772         }
```

# File: unix/clock.c

```
3725 clock(dev, sp, r1, nps, r0, pc, ps)
.......
3742 /*
3743  * callouts
3744  * if done, just return
3745  * else update first non-zero time
3746  */
3747
3748 if(callout[0].c_func == 0)
3749         goto out;
3750 p2 = &callout[0];
3751 while(p2->c_time<=0 &&
                        p2->c_func!=0)
3752         p2++;
3753 p2->c_time--;
3754
```

```
3755 /*
3756  * if ps is high, just return
3757  */
3758
3759 if((ps&0340) != 0)
3760         goto out;
3761
3762 /*
3763  * callout
3764  */
3765
3766 spl5();
3767 if(callout[0].c_time <= 0) {
3768         p1 = &callout[0];
3769         while(p1->c_func != 0 &&
                                p1->c_time <= 0) {
3770                 (*p1->c_func)(p1->c_arg);
3771                 p1++;
3772         }
```

# Waking up sleeping processes

- Data structure

```
0212 int lbolt;     /* time of day in 60th not in time */
0213 int time[2]; /* time in sec from 1970 */
0214 int tout[2]; /* time of day of next sleep */
```

- In function clock(..)

```
3797 if(++lbolt >= HZ) {
3798        if((ps&0340) != 0)
3799                return;
3800 lbolt =- HZ;
3801 if(++time[1] == 0)
3802        ++time[0];
3803 spl1();
3804 if(time[1]==tout[1] && time[0]==tout[0])
3805        wakeup(tout);
```

# Waking up sleeping processes

- Data structure

```
0212 int lbolt;     /* time of day in 60th not in time */
0213 int time[2]; /* time in sec from 1970 */
0214 int tout[2]; /* time of day of next sleep */
```

- In function clock(..)

```
3797 if(++lbolt >= HZ) {
3798        if((ps&0340) != 0)
3799                return;
3800 lbolt =- HZ;
3801 if(++time[1] == 0)
3802        ++time[0];
3803 spl1();
3804 if(time[1]==tout[1] && time[0]==tout[0])
3805        wakeup(tout);
```

# Waking up sleeping processes

- Data structure

```
0212 int lbolt;     /* time of day in 60th not in time */
0213 int time[2]; /* time in sec from 1970 */
0214 int tout[2]; /* time of day of next sleep */
```

- In function clock(..)

```
3797 if(++lbolt >= HZ) {
3798        if((ps&0340) != 0)
3799                return;
3800 lbolt =- HZ;
3801 if(++time[1] == 0)
3802        ++time[0];
3803 spl1();
3804 if(time[1]==tout[1] && time[0]==tout[0])
3805        wakeup(tout);
```

# Scheduling

```
0160 #define PUSER 100
3707 #define SCHMAG 10

0218 char runin;  /* scheduling flag */
0219 char runout; /* scheduling flag */
0220 char runrun; /* scheduling flag */

0358 struct proc
0359 {
0360   char p_stat;
0361   char p_flag;
0362   char p_pri; /* priority,
.....                    negative is high */
0365   char p_time; /* resident time for scheduling */
0366   char p_cpu; /* cpu usage for scheduling */
0367   char p_nice; /* nice for scheduling */
```

# File: unix/clock.c

```
3810 for(pp = &proc[0]; pp < &proc[NPROC]; pp++
3811 if (pp->p_stat) {
3812        if(pp->p_time != 127)
3813              pp->p_time++;
3814        if((pp->p_cpu & 0377) > SCHMAG)
3815              pp->p_cpu =- SCHMAG; else
3816              pp->p_cpu = 0;
3817        if(pp->p_pri > PUSER)
3818              setpri(pp);
3819 }
3820 if(runin!=0) {
3821        runin = 0;
3822        wakeup(&runin);
3823 }
```

sched()

# Scheduling

- In UNIX the lowest process priority wins!

```
0160 #define PUSER 100

0358 struct proc
0359 {
0360   char p_stat;
0361   char p_flag;
0362   char p_pri; /* priority,
.....                    negative is high */
0365   char p_time; /* resident time for scheduling */
0366   char p_cpu; /* cpu usage for scheduling */
0367   char p_nice; /* nice for scheduling */
```

- setpri() recalculates the process priority
  - p->pri = min(127, p->p_time/16 + PUSER + p->p_nice)
- More on scheduling and program swapping on May 11th by Urs Fässler

# Traps..

- are caused by events internal to the CPU
  - e.g. illegal instruction (segmentation fault, etc.), system call
- are handled by the same HW and SW mechanism as interrupts
- result in the function trap(..) being called
  - where the parameter dev will indicate the cause of the trap

```
2693 trap(dev, sp, r1, nps, r0, pc, ps)
2694 {
.......
2702 switch(dev) {
```

- In most cases a signal will be sent to the process
- System calls are an exception to this
  - Discussed in the next hour by Kaveh Razavi

# What's different today?

- Not that much actually
  - principal HW and SW mechanism is still the same
- More peripherals → more interrupts
- Large number of interrupts (e.g. high network traffic) can slowdown or even stall the system
  - scheduling of interrupt handling as important as process scheduling
  - Interrupts can be temporarily disabled (masked)

# Questions?