SVCs, Devices & Interrupts

- Interrupts for Async I/O
- ReadKey SVC example
- “Real Time”
  - Latencies & Deadlines
- Weak priorities
- Strong priorities

Lab 6 due tonight
Lab 7 due next Thursday

Loop:
LD(R3, 0, R0)
ADDC(R3, 4, R3)
SUBC(R2, 1, R2)
BNE(R2, Loop)
OS Organization: Supervisor Calls

USER PROCESSES

P1:
loop:SVC(0)
... SVC(1)
... BR(loop)

P2:
loop:SVC(2)
... SVC(3)
... BR(loop)

PN:
loop:SVC(3)
... SVC(1)
... BR(loop)

KERNEL

Saved Regs (Mstate)
Scheduler
Time

MState DPYNUM=0
MState DPYNUM=1
MState DPYNUM=27

User Mode PC[31]=0

Kernel Mode PC[31]=1

Illegal Op Handler
WrCh
getKey

Saved
Regs
(Mstate)
### Adding New SVCs

```
.macro GetTOD() SVC(8)   // return time of today in R0
.macro SetTOD() SVC(9)   // set time of day to value in R0

// Sub-handler for SVCs, called from I_Il10p on SVC opcode:
// SVC instruction is in R0, its address is Reg[XP]-4
SVC_UU0:
    ANDC(r0, 0xF, r1)    // Pick out low bits,
    SHLC(r1, 2, r1)      // make a word index,
    LD(r1, SVCTbl, r1)   // and fetch the table entry.
    JMP(r1)

SVCTbl:   UU0(HaltH)    // SVC(0): User-mode HALT instruction
          UU0(WrMsgH)   // SVC(1): Write message
          UU0(WrChH)    // SVC(2): Write Character
          UU0(GetKeyH)  // SVC(3): Get Key
          UU0(HexPrtH)  // SVC(4): Hex Print
          UU0(WaitH)    // SVC(5): Wait(S), S in R3
          UU0(SignalH)  // SVC(6): Signal(S), S in R3
          UU0(YieldH)   // SVC(7): Yield()
          UU0(GetTOD)   // SVC(8): return time of day
          UU0(SetTOD)   // SVC(9): set time of day
```
New SVC Handlers

// return the current time of day in R0
GetTOD:
  LD(TimeOfDay,r0)       // load OS time of day value
  ST(r0,UserMState+4*0)  // store into user’s R0
  BR(I_Rtn)              // resume execution with updated R0 value

// set the current time of day from the value in user’s R0
SetTOD:
  LD(UserMState+4*0,r0)  // load value in (saved) user’s R0
  ST(r0,TimeOfCay)       // store to OS time of day value
  BR(I_Rtn)              // resume execution

SVCs provide controlled access to OS services and data values and offer "atomic" (uninterrupted) execution of instruction sequences.
OS Organization: I/O Devices

USER PROCESSES

P1:
```
loop:SVC(0)
... SVC(1)
... BR(loop)
```

P2:
```
loop:SVC(2)
... SVC(3)
... BR(loop)
```

PN:
```
loop:SVC(3)
... SVC(1)
... BR(loop)
```

KERNEL

Saved Registers (MState)
- Scheduler
- Timer Handler
- Keyboard Handler
- Mouse Handler
- Illegal Op Handler
- Yield
- WrCh
- GetKey

User Mode
PC[31]=0

Kernel Mode
PC[31]=1

User Processes

P1: MState DPYNUM=0
P2: MState DPYNUM=1
PN: MState DPYNUM=27

Illegal Operation Handler

GetKey

WrCh

Yield

Save Registers (MState)

Timer Handler

Keyboard Handler

Click Buffer

Scheduler

Kernel Mode

User Mode
Asynchronous I/O Handling

Application:

... ReadKey() // read key into R0 ...

SVC call from application

Device Buffer (in OS Kernel)

KeyHit_h() {
  (read ASCII code, put in buffer)
}

INTERRUPT from Keyboard n

INTERRUPT to OS

ReadKey_h() {
  (remove next char from buffer, return in R0)
}
Interrupt-based Asynch I/O

OPERATION: NO attention to Keyboard during normal operation
• on key strike: hardware asserts IRQ to request interrupt
• USER program interrupted, PC+4 of interrupted inst. saved in XP
• state of USER program saved on KERNEL stack;
• Keyboard handler invoked, runs to completion;
• state of USER program restored; program resumes.

TRANSPARENT to USER program.

Keyboard Interrupt Handler (in O.S. KERNEL):

```c
struct Device {
    char Flag, Data;
} Keyboard;

KeyHit_h() {
    Buffer[inptr] = Keyboard.Data;
    inptr = ((inptr + 1) % BUFSIZE);
}
```
ReadKey SVC: Attempt #1

SVC recap: SVC, encoded as illegal instruction, causes an exception. OS notices special SVC opcode, dispatches to appropriate sub-handler based on index in low-bits of SVC inst.

First draft of a ReadKey SVC handler (supporting a virtual keyboard): returns next keystroke on a user’s keyboard in response to the SVC request:

```c
ReadKey_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum)) {
        /* busy wait loop */
    }
    UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

Problem: Can't interrupt code running in the supervisor mode... so the buffer never gets filled.
A BETTER keyboard SVC handler:

```c
ReadKey_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        /* busy wait loop */
        UserMState.Regs[XP] = UserMState.Regs[XP]-4;
    } else
        UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

This one actually works!

Problem: The process just wastes its time-slice waiting for someone to hit a key...
**ReadKey SVC: Attempt #3**

EVEN BETTER: On I/O wait, YIELD remainder of quantum:

ReadKey_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        UserMState.Regs[XP] = UserMState.Regs[XP]-4;
        Scheduler( );
    } else
    UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}

RESULT: Better CPU utilization!!

Does timesharing cause CPU use to be less efficient?

- COST: Scheduling, context-switching overhead; but
- GAIN: Productive use of idle time of one process by running another.
Sophisticated Scheduling

To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- Processes can be in **ACTIVE** or **WAITING** (“sleeping”) states;
- Scheduler cycles among **ACTIVE PROCESSES** only;
- Active process moves to **WAITING** status when it tries to read a character and buffer is empty;
- Waiting processes each contain a code (eg, in PCB) designating what they are waiting for (eg, keyboard N);
- Device interrupts (eg, on keyboard N) move any processes waiting on that device to **ACTIVE** state.

UNIX kernel utilities:

- `sleep(reason)` - Puts CurProc to sleep. “Reason” is an arbitrary binary value giving a condition for reactivation.
- `wakeup(reason)` - Makes active any process in `sleep(reason)`. 
ReadKey SVC: Attempt #4

SVC call from application

ReadKey_h() {
  ...
  if (BufferEmpty(kbdnum)) {
    sleep(kbdnum);
  } else {
    ...
  }
}

sleep(status s) {
  ProcTbl[Cur].status = s;
  Scheduler();
}

Scheduler() {
  ...
  while (ProcTbl[i].status != 0) {
    i = (i+1)%N;
  }
  ...
}

wakeup(status s) {
  for (i = 0; i < N; i += 1) {
    if (ProcTbl[i].status == s)
      ProcTbl[i].status = 0;
  }
}

INTERRUPT from Keyboard n

KeyHit_h() {
  ...
  WriteBuffer(kbdnum, key);
  wakeup(kbdnum);
  ...
}

See Example in topic videos
The Need for “Real Time”

Side-effects of CPU virtualization
+ abstraction of machine resources
  (memory, I/O, registers, etc.)
+ multiple “processes” executing concurrently
+ better CPU utilization
- Processing throughput is more variable

Our approach to dealing with the asynchronous world
- I/O - separate “event handling” from “event processing”

Difficult to meet “hard deadlines”
- control applications
- playing videos/MP3s

Real-time as an alternative to time-sliced or fixed-priority preemptive scheduling
Interrupt Latency

One way to measure the real-time performance of a system is **Interrupt Latency**:

- **HOW MUCH TIME** can elapse between an interrupt request and the **START of its handler**?

![Diagram of interrupt latency with time axis and labels for latency (L), service time (S), deadline (D), and request]

Sometimes bad things happen when service is delayed beyond its “**dead**”-line:

- Missed characters
- Automobile crashes
- Nuclear meltdowns

“HARD” Real time constraints

\[
L_{\text{MAX}} + S = D
\]
Sources of Interrupt Latency

What causes interrupt latency:

- State save, context switch.
- Periods of un-interruptability:
  - Long, uninterruptable instructions – e.g. block moves
  - Explicitly disabled periods (e.g. during service of other interrupts).

GOAL: BOUND (and minimize) interrupt latency!

- Optimize interrupt sequence context switch
- Make unbounded-time instructions interruptable (state in registers, etc).
- Avoid/minimize disable time
- Allow handlers to be interrupted, in certain cases.
"TOY" System scenario:

<table>
<thead>
<tr>
<th>DEVICE</th>
<th>Actual w/c Latency</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Keyboard</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td>Disk</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Printer</td>
<td>400</td>
</tr>
</tbody>
</table>

500 + 400 = 900
800 + 400 = 1200
800 + 500 = 1300

What is the WORST CASE latency seen by each device?

Assumptions:
- Infrequent interrupt requests (each happens only once/scenario)
- Simultaneous requests might be served in ANY order…. Whence
- Service of EACH device might be delayed by ALL others!
ISSUE: Processor becomes interruptable (at fetch of next instruction), several interrupt requests are pending. Which is served first?

WEAK PRIORITY ORDERING: Check in prescribed sequence, eg: DISK > PRINTER > KEYBOARD.

Latencies with WEAK PRIORITIES:
Service of each device might be delayed by:
- Service of 1 other (arbitrary) device, whose interrupt request was just honored;
- Service of ALL higher-priority devices.

<table>
<thead>
<tr>
<th>DEVICE</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>800</td>
</tr>
<tr>
<td>Disk</td>
<td>500</td>
</tr>
<tr>
<td>Printer</td>
<td>400</td>
</tr>
</tbody>
</table>

Actual w/c
Latency  DEVICE   Service Time
900       Keyboard  800
800       Disk      500
1300      Printer   400

vs 1200 – Now delayed by only 1 service!
Setting Priorities

How should priorities be assigned given hard real-time constraints? We’ll assume each device has a service deadline $D$. If not otherwise specified, assume $D$ is the time until the next request for the same device, e.g., the keyboard handler should be finished processing one character before the next arrives.

“Earliest Deadline” is a strategy for assigning priorities that is guaranteed to meet the deadlines if any priority assignment can meet the deadlines:

1. Sort the requests by their deadlines
2. Assign the highest priority to the earliest deadline, second priority to the next deadline, and so on.
3. Weak priority scheduling: choose the pending request with the highest priority, i.e., that has the earliest deadline.
The Need for Preemption

Without preemption, ANY interrupt service can delay ANY other service request... the slowest service time constrains response to fastest devices. Often, tight deadlines can’t be met using this scheme alone.

EXAMPLE: 800 uSec deadline (hence 300 uSec maximum interrupt latency) on disk service, to avoid missing next sector...

<table>
<thead>
<tr>
<th>Priority</th>
<th>Latency w/ preemption</th>
<th>Device</th>
<th>Service Time (S)</th>
<th>Deadline (D)</th>
<th>$L_{\text{MAX}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[D,P] 900</td>
<td>900us</td>
<td>Keyboard</td>
<td>800us</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>~0</td>
<td>800us</td>
<td>Disk</td>
<td>500us</td>
<td>800us</td>
</tr>
<tr>
<td>2</td>
<td>[D] 500</td>
<td>1300us</td>
<td>Printer</td>
<td>400us</td>
<td></td>
</tr>
</tbody>
</table>

CAN'T SATISFY the disk requirement in this system using weak priorities!

need PREEMPTION: Allow handlers for LOWER PRIORITY interrupts to be interrupted by HIGHER priority requests!
Strong Priority Implementation

**STRONG PRIORITY ORDERING:** Allow handlers for LOWER PRIORITY interrupts to be preempted (interrupted) by HIGHER PRIORITY requests.

**SCHEME:**
- Expand supervisor bit in PC to be a PRIORITY integer PRI (eg, 3 bits for 8 levels)
- ASSIGN a priority to each device.
- Prior to each instruction execution:
  - Find priority $P_{\text{DEV}}$ of highest requesting device, say $D_i$
  - Take interrupt if and only if $P_{\text{DEV}} > PRI$, set $PRI = P_{\text{DEV}}$.

PC: \[\begin{array}{c|c}
\text{PRI} & \text{Program Counter} \\
\end{array}\]

**Strong priorities:**
**KEY:** Priority in Processor state
- Allows interruption of (certain) handlers
- Allows preemption, but not reentrance

**BENEFIT:** Latency seen at high priorities UNAFFECTED by service times at low priorities.
Consider interrupts which recur at bounded rates:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Latency using strong priority</th>
<th>Device</th>
<th>Service Time (S)</th>
<th>Deadline (D)</th>
<th>$L_{\text{MAX}}$</th>
<th>Max Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>900us</td>
<td>Keyboard</td>
<td>800us</td>
<td></td>
<td></td>
<td>100/s</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Disk</td>
<td>500us</td>
<td>800us</td>
<td>300us</td>
<td>500/s</td>
</tr>
<tr>
<td>2</td>
<td>500us</td>
<td>Printer</td>
<td>400us</td>
<td></td>
<td></td>
<td>1000/s</td>
</tr>
</tbody>
</table>

Note that interrupt LATENCIES don't tell the whole story—consider COMPLETION TIMES, e.g., for Keyboard in the example above.

Keyboard service not complete until 3 ms after request!
## Interrupt Load

How much CPU time is consumed by interrupt service?

10 ms. cycle

![Diagram showing CPU cycle with interrupts and deadlines]

<table>
<thead>
<tr>
<th>P</th>
<th>Latency</th>
<th>Device</th>
<th>Service Time (S)</th>
<th>Deadline (D)</th>
<th>(L_{\text{MAX}})</th>
<th>Max Freq.</th>
<th>% Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>900us</td>
<td>Keyboard</td>
<td>800us</td>
<td></td>
<td>100/s</td>
<td>800us*100/s = 8%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Disk</td>
<td>500us</td>
<td>800us</td>
<td>300us</td>
<td>500us*500/s = 25%</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>500us</td>
<td>Printer</td>
<td>400us</td>
<td></td>
<td>1000/s</td>
<td>400us*1000/s = 40%</td>
<td></td>
</tr>
</tbody>
</table>

- User-mode share of CPU = \(1 - \sum(S_{\text{DEV}} \times \text{max_freq}_{\text{DEV}})\) = 0.27
- Also check to see if enough CPU time to meet all deadlines
Example: Mr. Blue Visits the ISS

International Space Station’s on-board computer performs 3 tasks:
• guiding incoming supply ships to a safe docking
• monitoring gyros to keep solar panels properly oriented
• controlling air pressure in the crew cabin

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Service time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply ship guidance</td>
<td>30ms</td>
<td>5ms</td>
<td>25ms</td>
</tr>
<tr>
<td>Gyrosopes</td>
<td>40</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Cabin pressure</td>
<td>100</td>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>

Assuming a weak priority system:
1. What is the maximum service time for “cabin pressure” that still allows all constraints to be met? ≤ 10 mS
2. Give a weak priority ordering that meets the constraints G > SSG > CP
3. What fraction of the time will the processor spend idle? 48.33%
4. What is the worst-case completion time for each task?
Our Russian collaborators don’t like the sound of a “weak” priority interrupt system and lobby heavily to use a “strong” priority interrupt system instead.

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Service time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply ship guidance</td>
<td>30ms</td>
<td>5ms</td>
<td>25ms</td>
</tr>
<tr>
<td>Gyroscopes</td>
<td>40</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Cabin pressure</td>
<td>100</td>
<td>? 50</td>
<td>100</td>
</tr>
</tbody>
</table>

Assuming a strong priority system, $G > SSG > CP$:

1. What is the maximum service time for “cabin pressure” that still allows all constraints to be met? $100 - (3\times10) - (4\times5) = 50$

2. What fraction of the time will the processor spend idle? $8.33\%$

3. What is the worst-case completion time for each task?
Summary

Device interface – two parts:
- Device side: handle interrupts from device (transparent to apps)
- Application side: handle interrupts (SVCs) from application

Scheduler interaction:
- “Sleeping” (*inactive) processes waiting for device I/O
- Handler coding issues, looping thru User mode

Real Time constraints, scheduling, guarantees”
- Complex, hard scheduling problems – a black art!
- Weak (non-preemptive) vs Strong (preemptive) priorities help…
- Common real-world interrupt systems:
  - Fixed number (eg, 8 or 16) of strong priority levels
  - Each strong priority level can support many devices, arranged in a weak priority chain