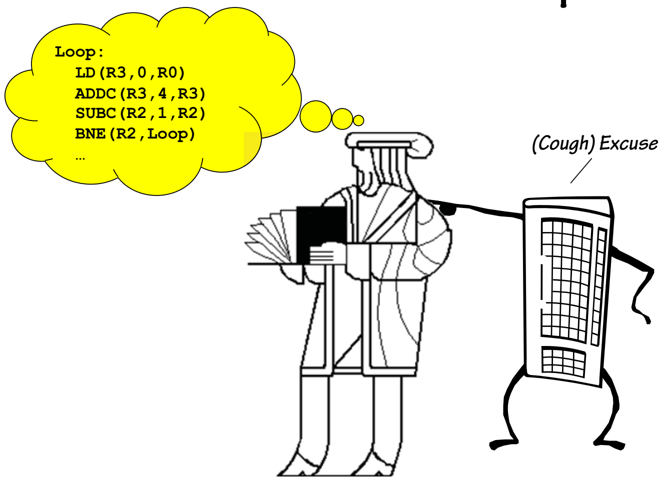
6.004 Computation Structures
Spring 2009

Devices & Interrupts



Lab #6 due tonight!

Why an OS?

What we've got:

- A Single Sequence Machine, capable of doing ONE thing at a time – one instruction, one I/O operation, one program.
- A universe of gadgets – e.g. I/O devices – that do similar things slightly differently.

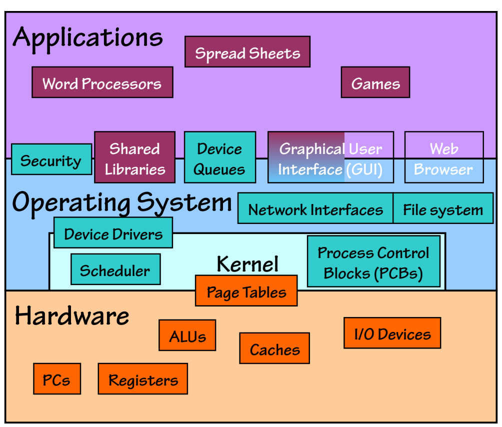
What we'd like:

- To listen to MP3s while reading email.
- To access disk, network, and screen "simultaneously".
- To write a single program that does I/O with anybody's disk.

Plausible approaches:

- An infinite supply of identical computers with uniform, high-level peripherals for every conceivable purpose… or
- An illusion: Make one real computer look like many "virtual" ones.

Operating Systems



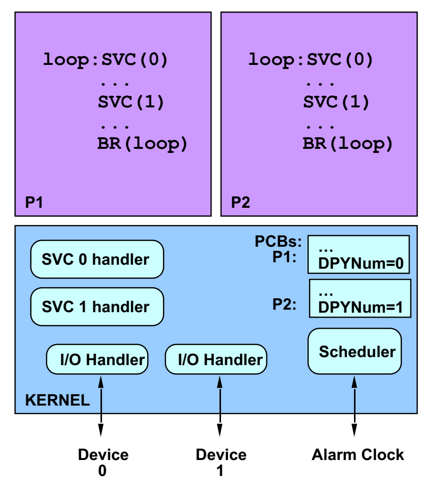
An OS is the Glue that holds a computer together.

- Mediates between competing requests
- Resolves names/bindings
- Maintains order/fairness

KERNEL - a RESIDENT portion of the O/S that handles the most common and fundamental service requests.

vir.tu.al \'v*rch-(*-)w*l, 'v*r-ch*l\ \.v*r-ch*-'wal-*t-e-\ \'v*rch-(*-)w*-le-, 'v*rch-(*-)le-\ aj [ME, possessed of certain physical virtues, fr. ML virtualis, fr. L virtus strength, virtue : being in essence or effect but not in fact - vir.tu.al.i.ty n

OS organization



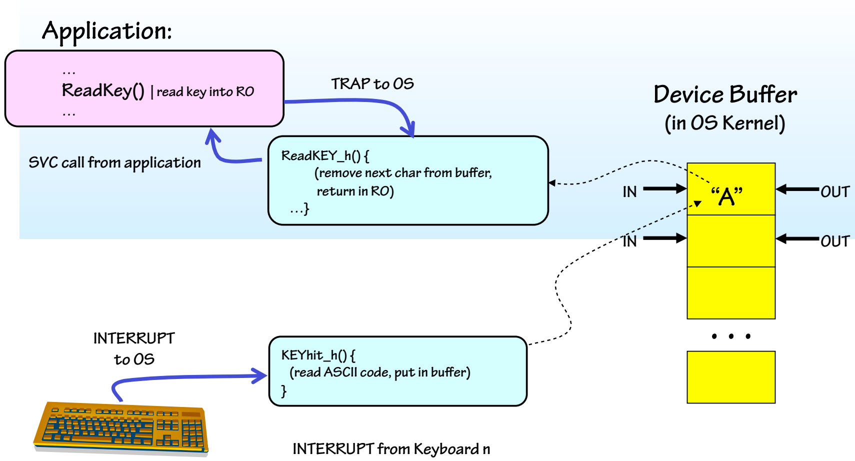
"Applications" are quasi-parallel "PROCESSES" on "VIRTUAL MACHINES", each with:

- CONTEXT (virtual address space)
- Virtual I/O devices

O.S. KERNEL has:

- Interrupt handlers
- SVC (trap) handlers
- Scheduler
- PCB structures containing the state of inactive processes

Asynchronous I/O Handling



Interrupt-based Asynchronous 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;
- KeyboardHandler invoked, runs to completion;
- state of USER program restored; program resumes.

TRANSPARENT to USER program.

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

Assume each keyboard has an associated buffer

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

```
KEYhit_h() {
  Buffer[inptr] = Keyboard.Data;
  inptr = (inptr + 1) % BUFSIZE;
}
```


ReadKey SVC: Attempt #1

A *supervisor call* (SVC) is an instruction that transfers control to the kernel so it can satisfy some user request. Kernel returns to user program when request is complete.

First draft of a ReadKey SVC handler (supporting a *Virtual* Keyboard): returns next keystroke on a user's keyboard to that user's requesting application:

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

Problem: Can't interrupt code running in the supervisor mode… so the buffer never gets filled.

ReadKey SVC: Attempt #2

A BETTER keyboard SVC handler:

```
ReadKEY_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
      /* busy wait loop */
      User.Regs[XP] = User.Regs[XP]-4;
    } else

      User.Regs[0] = ReadInputBuffer(kbdnum);
}
```

That's a funny way to write a loop

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)) {
        User.Regs[XP] = User.Regs[XP]-4;
        Scheduler( );
    } else
        User.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

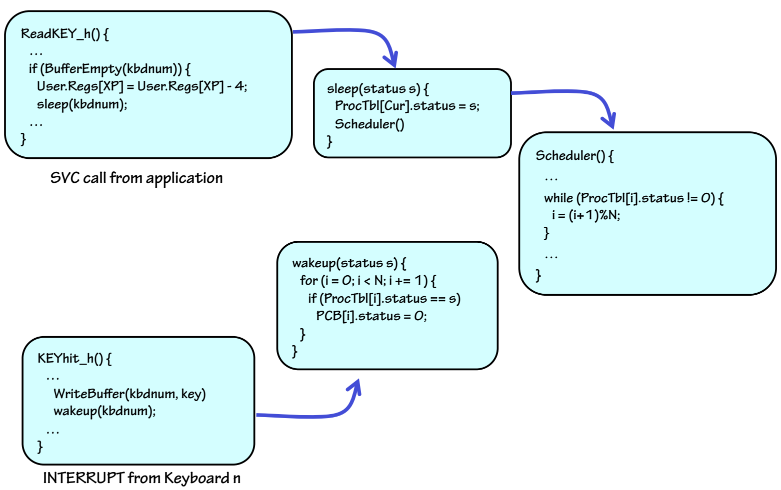
To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- 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



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"

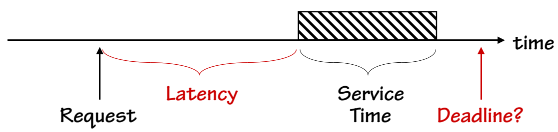
Difficult to meet "hard deadlines"
\- control applications
\- playing videos/MP3s

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?*

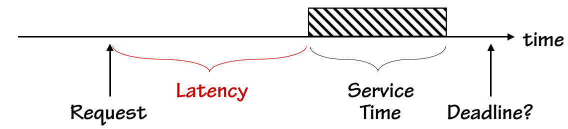


OFTEN bad things happen when service is delayed beyond some deadline - "real time" considerations:

Missed characters
System crashes
Nuclear meltdowns

"HARD" Real time constraints

Sources of Interrupt Latency



What causes interrupt latency:

We can consider this when we write our O/S

We can address this in our ISA

- State save, context switch.
- Periods of uninterruptability:
  - Long, uninterruptable instructions -- eg block moves, multi-level indirection.
  - Explicitly disabled periods (eg for atomicity, during service of other interrupts).

But, this is application dependent!

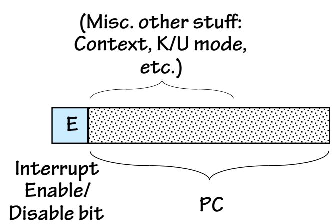
GOAL: BOUND (and minimize) interrupt latency!

- 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 (while still avoiding reentrant handlers!).

Interrupt Disable/Enable

INTERRUPT DISABLE BIT (part of processor status)... in PC:

Often in separate Processor Status Word ...



E=1: DISABLED
E=0: ENABLED

e.g.
- BETA K-mode bit (disables interrupts, other functions)
- Often separate bit/ mechanism

TYPICAL OPERATION: (as with Beta K-mode bit):

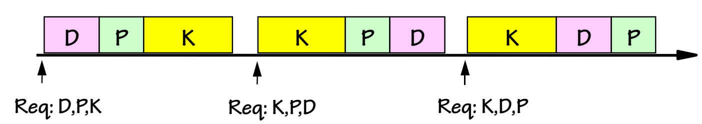
- ONLY take interrupts if E=0; else defer.
- SAVE OLD E on interrupt, install new E from interrupt vector (along with PC, etc). New E=1 (to disable interrupts during handler).
- Run handler, with interrupts disabled.
- On JMP (at return from handler), saved state restored to processor, resuming interrupted program (with E=0 again).

Scheduling of Multiple Devices

"TOY" System scenario:

Actual w/c Latency	DEVICE	Service Time
500 + 400 = 900	Keyboard	800
800 + 400 = 1200	Disk	500
800+ 500 = 1300	Printer	400

What is the WORST CASE latency seen by each device?



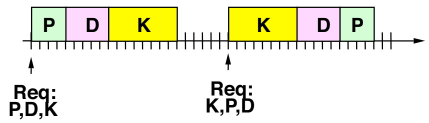
Assumptions:

- 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!

… can we improve this?

Weak (non-preemptive) Priorities

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; PLUS
- Service of ALL higher-priority devices.

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

vs 1200 –
Now delayed by only 1 service!

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...

Priority	Latency w/preemption	Actual Latency	DEVICE	Serv. Time	Max. Delay
1	D,P 900	900	Keybrd	800	
3	~0	800	Disk	500	300
2	[D] 500	1300	Printer	400	

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

SCHEME:

- Expand E 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_i of highest requesting device, say D_i
  - Take interrupt if and only if P_i > PRI, set PRI = P_i.

PC: | PRI | Program Counter |

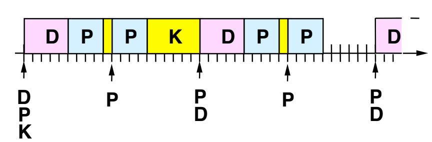
Strong priorities:
KEY: Priority in Processor state
Allows interruption of (certain) handlers
Allows preemption, but not reentrance
BENEFIT: Latency seen at high priorities UNAFFECTED by service times at low priorities.

Recurring Interrupts

Consider interrupts which recur at bounded rates:

Actual Latency	DEVICE	P	Serv. Time	Max. Delay	Max. Freq
900	Keybrd	3	800		100/s
0	Disk	5	500	300	500/s
500	Printer	4	400		1000/s

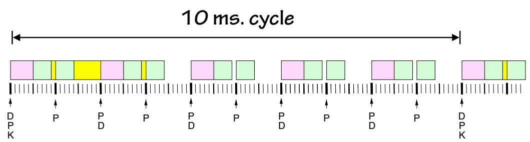
Note that interrupt LATENCIES don't tell the whole story—consider COMPLETION TIMES, eg for Keyboard in example to the right.



Keyboard service not complete until 3ms after request. Often *deadlines* used rather than max. delays.

Interrupt Load

How much CPU time is consumed by interrupt service?



Actual Latency	DEVICE	P	Serv. Time	Max. Delay	Max. Freq	% Load
900	Keybrd	3	800		100/s	800u*100/s = 8%
0	Disk	5	500	300	500/s	500u*500/s = 25%
500	Printer	4	400		1000/	400u*1000/s = 40%

Remaining fraction (27%) is left over for application; *trouble if its <0!*

Example: Ben visits 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

	Task	Period	Service time	Deadline	
16.6 %	Supply ship guidance	30ms	5ms	25ms	C,G = 10 + 10 + (5) = 25
25%	Gyroscopes	40	10	20	C = 10 + (10) = 20
10%	Cabin pressure	100	? 10	100	S,G = 5 + 10 + (10) = 25

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 delay for each type of interrupt until completion of the corresponding service routine?

Example: Ben visits ISS (cont'd)

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.

	Task	Period	Service time	Deadline	
16.6%	Supply ship guidance	30ms	5ms	25ms	[G] 10 + 5
25%	Gyroscopes	40	10	20	10
50%	Cabin pressure	100	? 50	100	100

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*10) – (4*5) = 50
2. What fraction of the time will the processor spend idle? 8.33%
3. What is the worst-case delay for each type of interrupt until *completion* of the corresponding service routine?

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