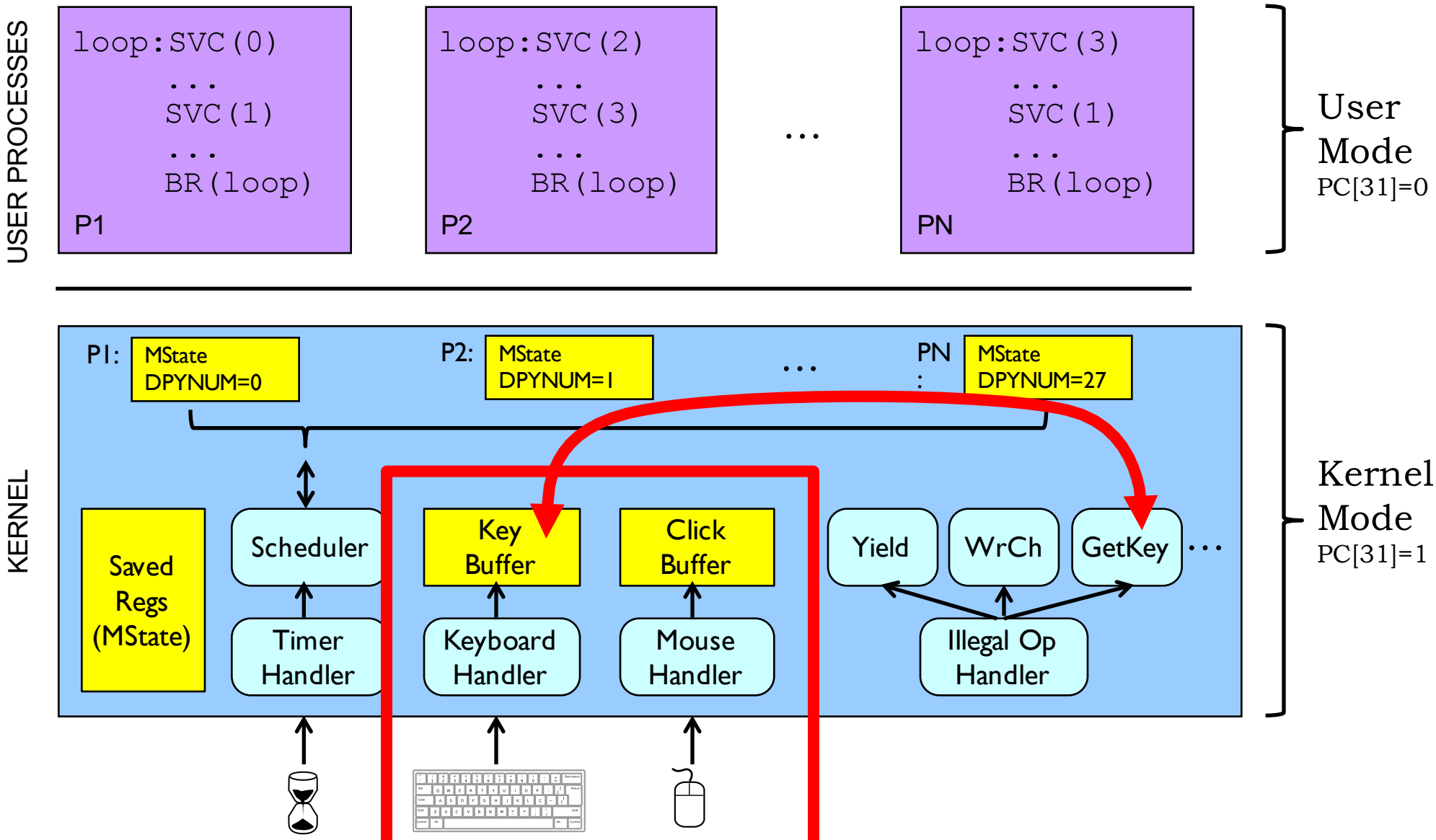
18. Devices and Interrupts

6.004x Computation Structures

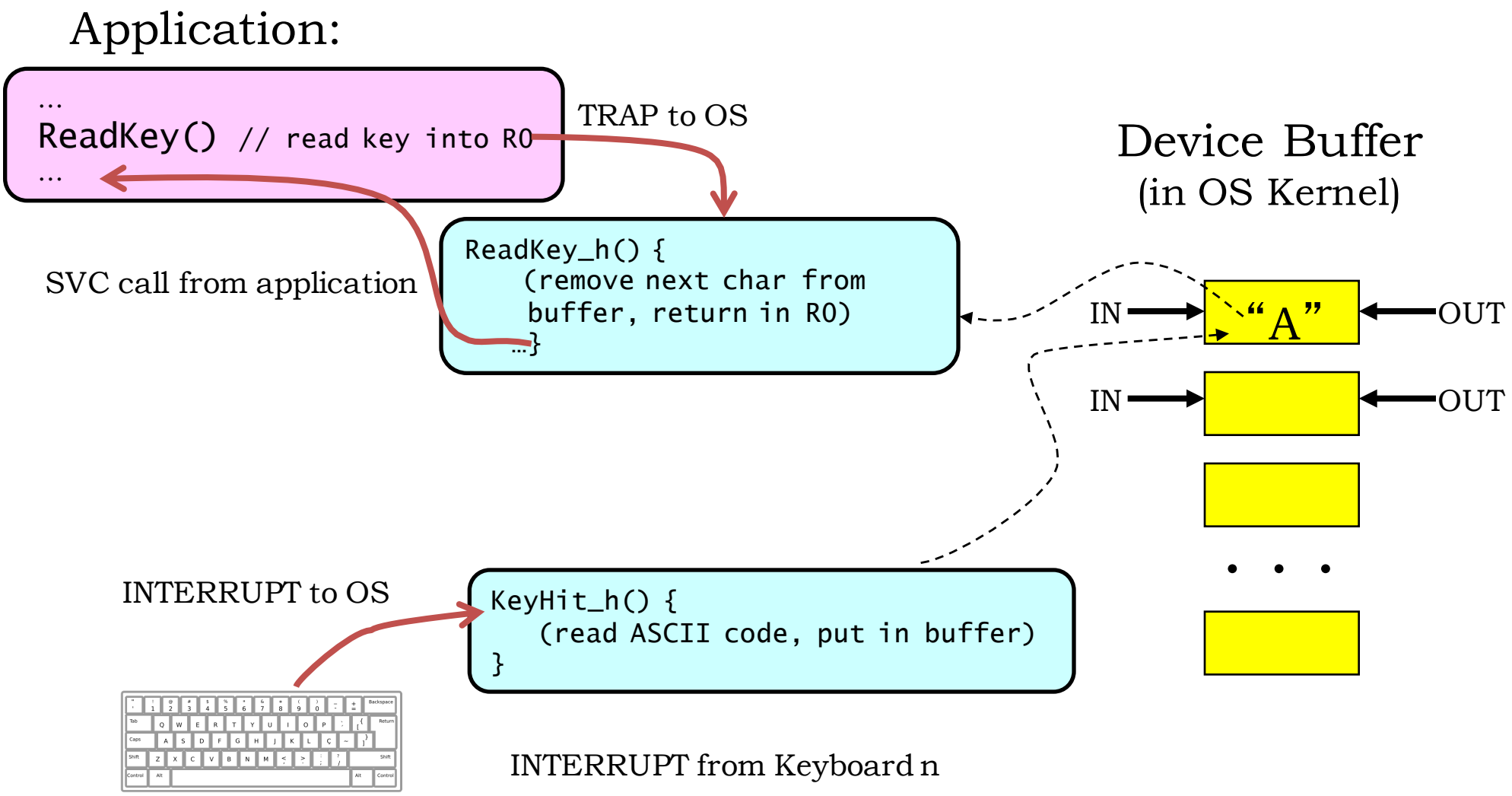
Part 3 – Computer Organization

OS Device Handlers

OS Organization: I/O Devices



Asynchronous I/O Handling



Interrupt-based Asynch I/O

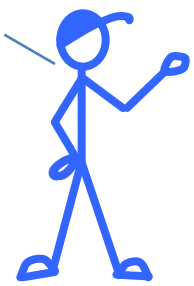
OPERATION: NO attention to Keyboard during normal operation

- 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):

Assume each keyboard has an associated buffer



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

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

SVCs for Input/Output

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:

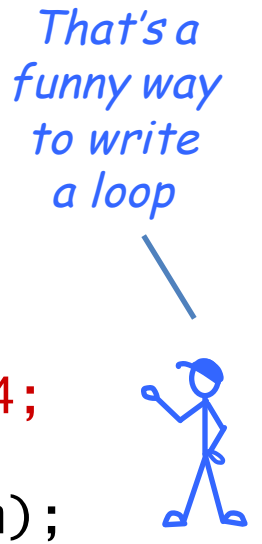
```
ReadKey_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum)) {
        /* busy wait loop */
    }
    UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

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

ReadKey SVC: Attempt #2

A BETTER keyboard SVC handler:

```
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

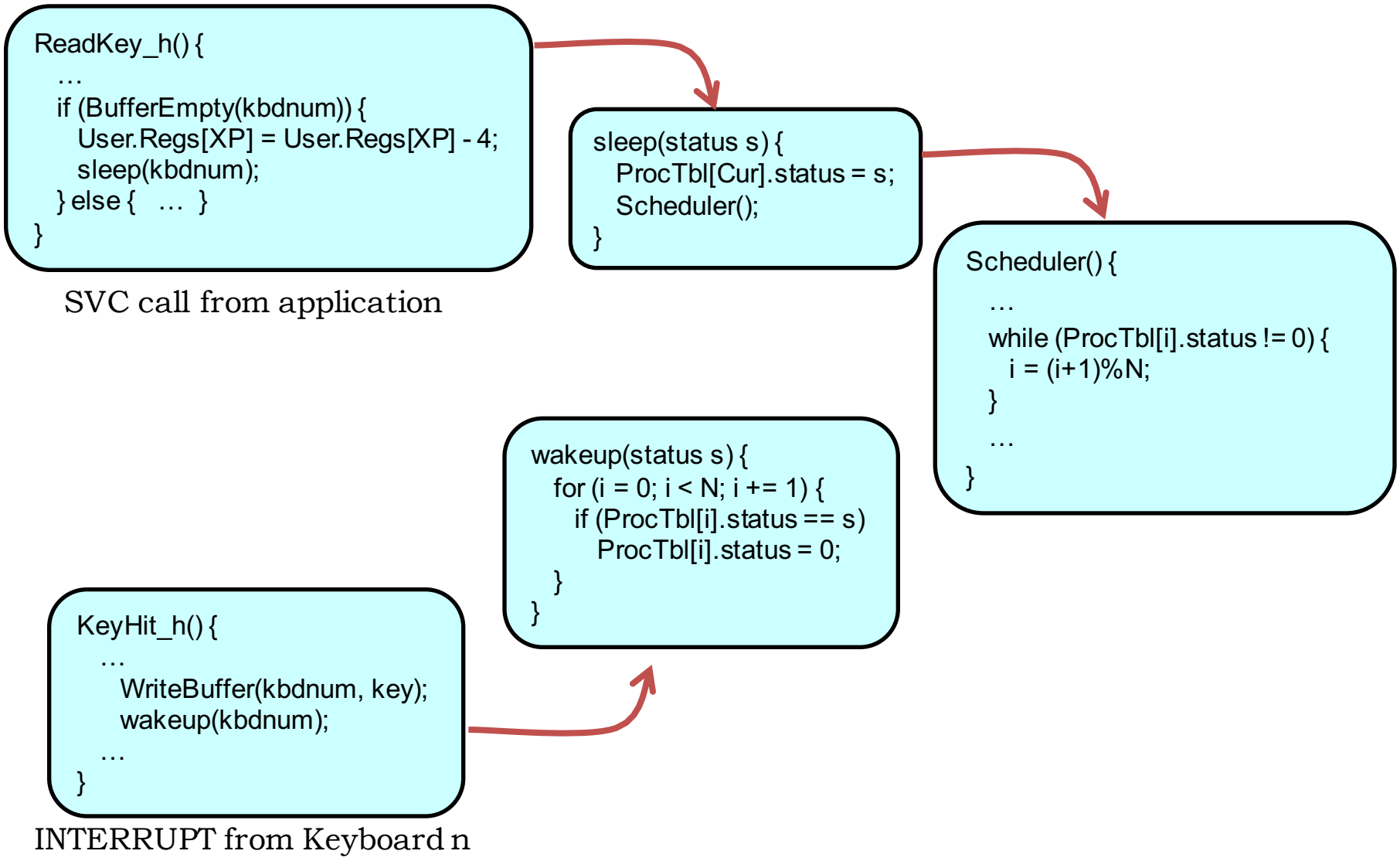
To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- 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



Example: Match Handler with OS

Example: Match Handler to OS


```
ReadCh_h() {  // Version R1
  if (BufferEmpty(0))
    UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
  else
    UserMState.Regs[0] = ReadInputBuffer(0);
}
```

Always reads from the same buffer


```
ReadCh_h() {  // Version R2
  int kbdnum = ProcTbl[Cur].DPYNum;
  while (BufferEmpty(kbdnum));
  UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

Oops! Infinite loop?


```
ReadCh_h() {  // Version R3
  int kbdnum = ProcTbl[Cur].DPYNum;
  if (BufferEmpty(kbdnum)) {
    UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
    Scheduler();
  } else
    UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

Model A: A timeshared Beta system whose OS kernel is uninterruptable

Model B: A timeshared Beta system which enables device interrupts during handing of SVC traps

Model C: A single-process (not timeshared) system which runs dedicated application code

R1

```
ReadCh_h() {  // Version R1
  if (BufferEmpty(0))
    UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
  else
    UserMState.Regs[0] = ReadInputBuffer(0);
}
```

R2

```
ReadCh_h() {  // Version R2
  int kbdnum = ProcTbl[Cur].DPYNum;
  while (BufferEmpty(kbdnum));
  UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

R3

```
ReadCh_h() {  // Version R3
  int kbdnum = ProcTbl[Cur].DPYNum;
  if (BufferEmpty(kbdnum)) {
    UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
    Scheduler();
  } else
    UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

	A	B	C
R1	X	X	X
R2	X	X	X
R3	X	X	

Model A: A timeshared Beta system whose OS kernel is uninterruptable

Model B: A timeshared Beta system which enables device interrupts during handing of SVC traps

Model C: A single-process (not timeshared) system which runs dedicated application code

Which handler & OS?

"I get compile-time errors; Scheduler and ProcTbl are undefined!"

R1

```
ReadCh_h() {  // Version R1
  if (BufferEmpty(0))
    UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
  else
    UserMState.Regs[0] = ReadInputBuffer(0);
}
```

R2

```
ReadCh_h() {  // Version R2
  int kbdnum = ProcTbl[Cur].DPYNum;
  while (BufferEmpty(kbdnum));
  UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

R3

```
ReadCh_h() {  // Version R3
  int kbdnum = ProcTbl[Cur].DPYNum;
  if (BufferEmpty(kbdnum)) {
    UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
    Scheduler();
  } else
    UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

	A	B	C
R1		X	X
R2	X	X	X
R3	X	X	X

Model A: A timeshared Beta system whose OS kernel is uninterruptable

Model B: A timeshared Beta system which enables device interrupts during handing of SVC traps

Model C: A single-process (not timeshared) system which runs dedicated application code

Which handler & OS?

"Hey, now the system always reads everybody's input from keyboard 0. In addition, it seems to waste a lot more CPU cycles than it used to."

R1

```
ReadCh_h() {  // Version R1
  if (BufferEmpty(0))
    UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
  else
    UserMState.Regs[0] = ReadInputBuffer(0);
}
```

R2

```
ReadCh_h() {  // Version R2
  int kbdnum = ProcTbl[Cur].DPYNum;
  while (BufferEmpty(kbdnum));
  UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

R3

```
ReadCh_h() {  // Version R3
  int kbdnum = ProcTbl[Cur].DPYNum;
  if (BufferEmpty(kbdnum)) {
    UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
    Scheduler();
  } else
    UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}
```

	A	B	C
R1	X	X	X
R2	X	X	X
R3	X		X

Model A: A timeshared Beta system whose OS kernel is uninterruptable

Model B: A timeshared Beta system which enables device interrupts during handing of SVC traps

Model C: A single-process (not timeshared) system which runs dedicated application code

Which handler & OS?

"Neat, the new system seems to work fine.
It even wastes less CPU time than it used to!"

Real Time

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"

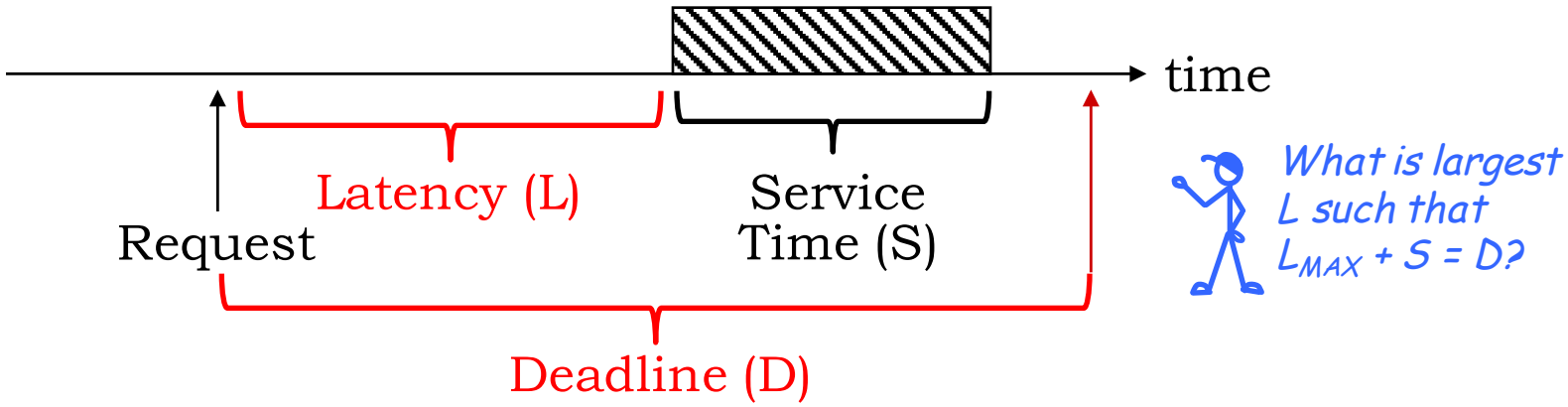
- control applications, e.g., ESC on cars
- playing videos/MP3s

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:*

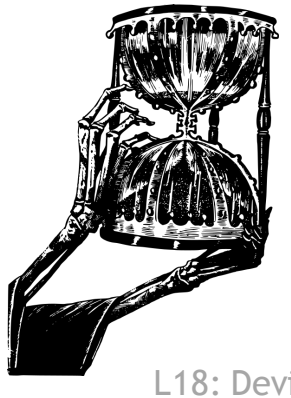
- *HOW MUCH TIME can elapse between an interrupt request and the START of its handler?*



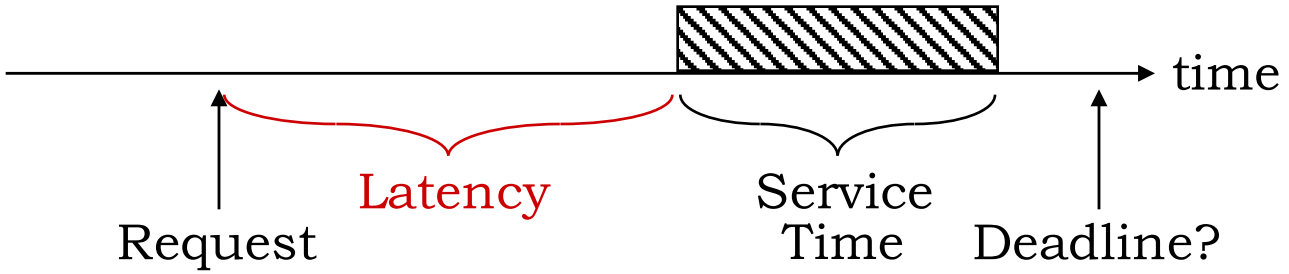
Sometimes bad things happen when service is delayed beyond its "dead"-line:

Missed characters
Automobile crashes
Nuclear meltdowns

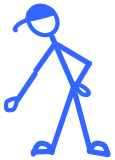
} "HARD" Real time constraints



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.

Weak Priorities

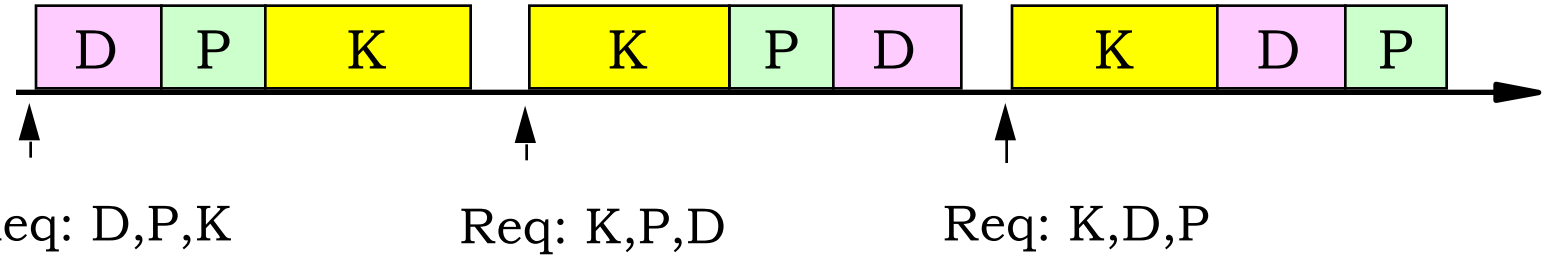
Scheduling of Multiple Devices

"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

Can we do better?

What is the WORST CASE latency seen by each device?

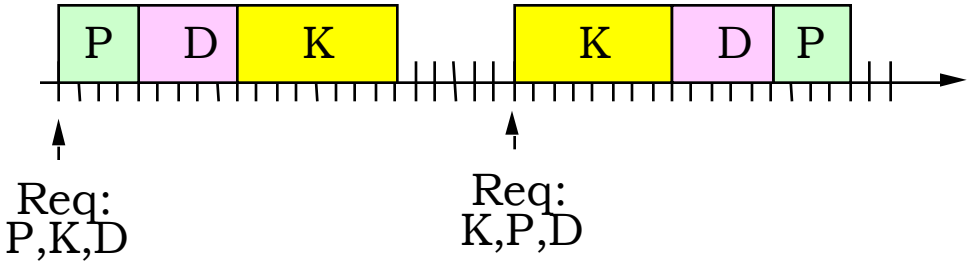


Assumptions:

- 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!

Weak (Non-preemptive) Priorities

ISSUE: Processor becomes interruptable on returning to user mode, several interrupt requests are pending. Which is served first?



WEAK PRIORITY ORDERING: Check in prescribed sequence, e.g.: 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.

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.

Strong Priorities

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	Latency using weak priority	Device	Service Time (S)	Deadline (D)	L_{MAX}
1	[D,P] 900	900us	Keyboard	800us		
3	~0	800us	Disk	500us	800us	300us
2	[D] 500	1300us	Printer	400us		

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_{DEV} of highest requesting device, say D_i
  - Take interrupt if and only if $P_{DEV} > PRI$, set $PRI = P_{DEV}$.

Strong priorities:

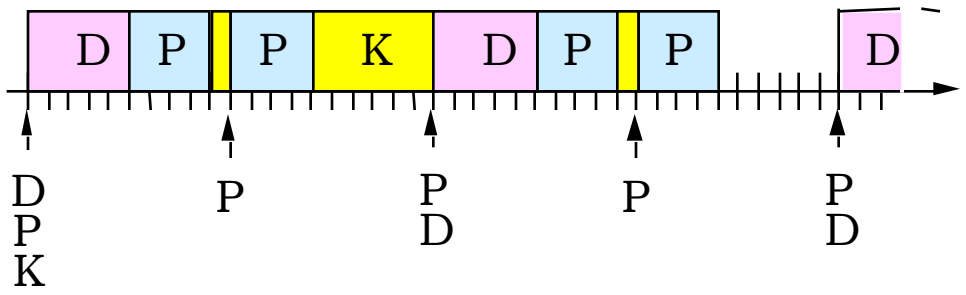
KEY: Priority in Processor state
Allows interruption of (certain) handlers
Allows preemption, but not reentrance

BENEFIT: Latency seen at high priorities UNAFFECTED by service times at low priorities.

Recurring Interrupts

Consider interrupts which recur at bounded rates:

Priority	Latency using strong priority	Device	Service Time (S)	Deadline (D)	L_{MAX}	Max Freq.
1	900us	Keyboard	800us			100/s
3	0	Disk	500us	800us	300us	500/s
2	500us	Printer	400us			1000/s

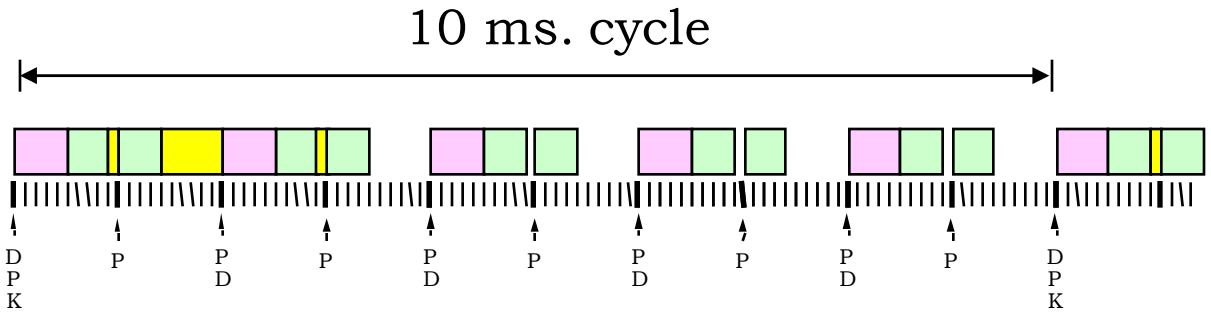


Note that interrupt LATENCIES don't tell the whole story—consider COMPLETION TIMES, e.g., for Keyboard in the example above.

Keyboard service not complete until 3 ms after request!

Interrupt Load

How much CPU time is consumed by interrupt service?

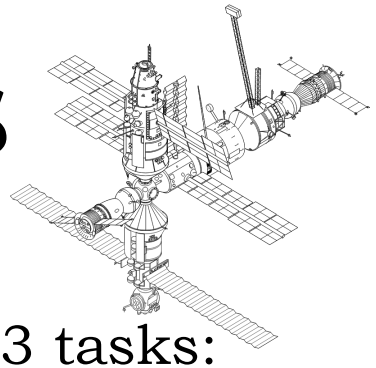


P	Latency	Device	Service Time (S)	Deadline (D)	L_{MAX}	Max Freq.	% Load
1	900us	Keyboard	800us			100/s	800us*100/s = 8%
3	0	Disk	500us	800us	300us	500/s	500us*500/s = 25%
2	500us	Printer	400us			1000/s	400us*1000/s = 40%

- User-mode share of CPU = $1 - \sum(S_{DEV} * max\_freq_{DEV}) = 0.27$
- Also check to see if enough CPU time to meet all deadlines

Example: Priorities in Action!

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

	Task	*Period*	*Service time*	*Deadline*	
16.67%	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 completion time for each task?

Example: Mr. Blue 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*	
16.67%	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 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