# Le Chat et le Communisme

# “Any Input Is a Program”: Weird Machines in ABI and Architecture Metadata


Julian Bangert
Rebecca ‘.bx’ Shapiro
Sergey Bratus

Trust Lab
Dartmouth College



“Weird machines all the way down”

# Disclaimer

- ***Turing-complete*** is just a way of describing what kind of computations an environment can be programmed to do
  (T.-c. = any kind we know, in theory)
- Wish we had a more granular scale better suited to exploit power

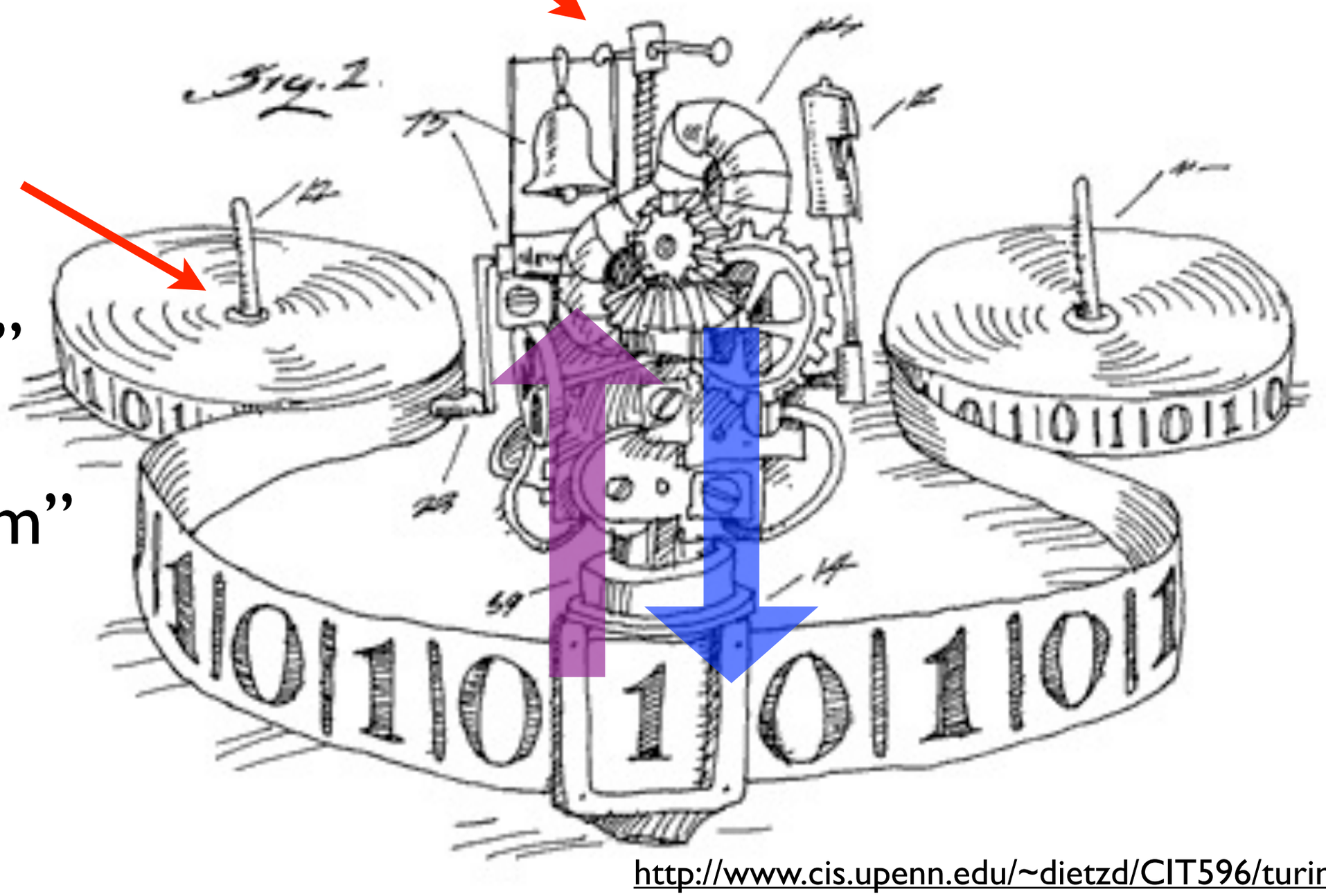
# Any Input is a Program

"Any sufficiently complex input is indistinguishable from bytecode; any code that takes complex inputs is indistinguishable from a VM/interpreter for that bytecode"

Finite state automaton

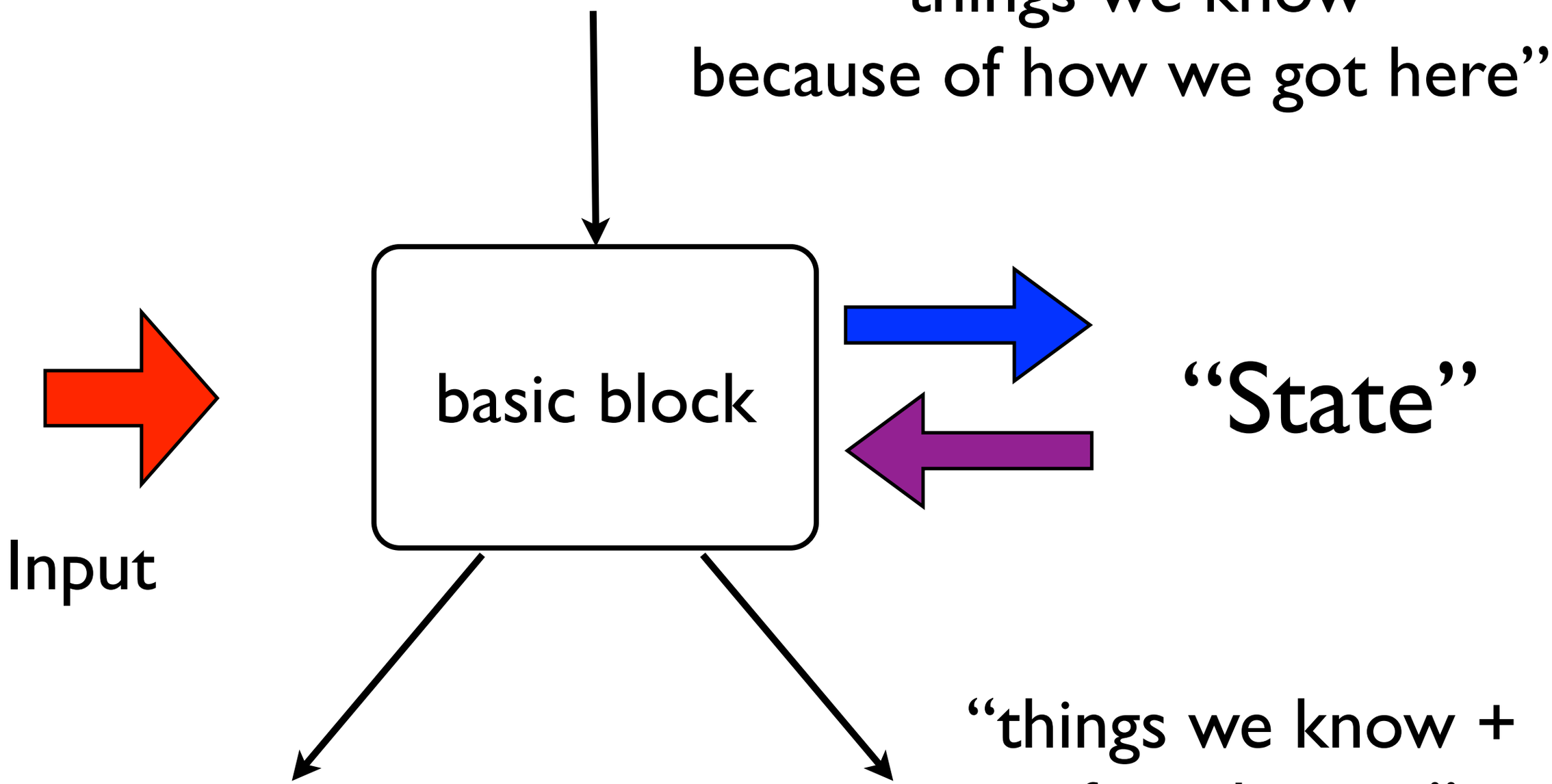
**Reads** from tape
Changes state
**Writes** to tape

Tape: both "input" and "program"

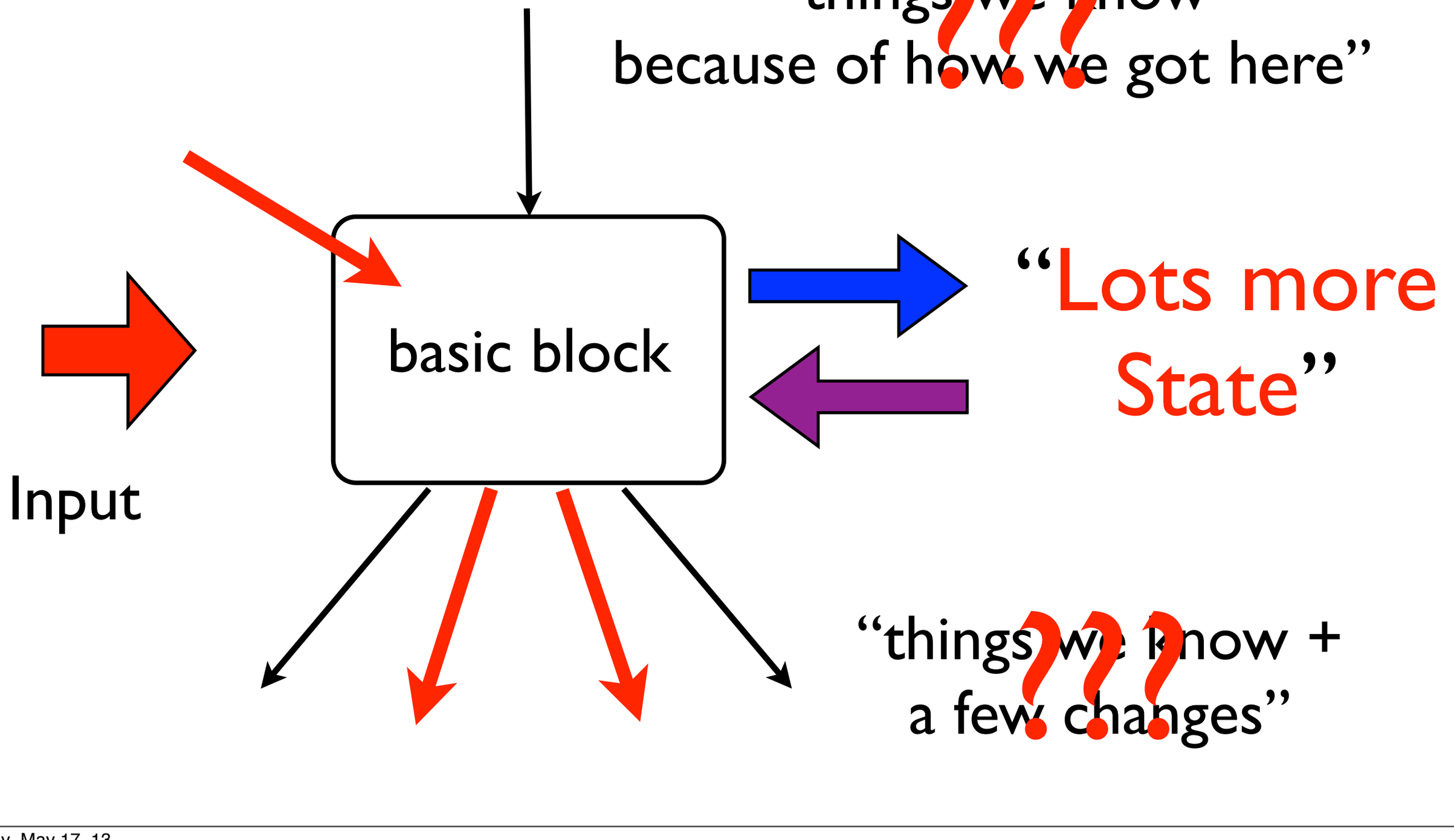



http://www.cis.upenn.edu/~dietzd/CIT596/turingMachine.gif

# Compiler's view:



# Exploit's view:



# "X Runs on Y"

- Inputs execute on parsers
- Packets execute on TCP/IP stacks
- Heaps execute on heap managers
- Binary format metadata execute on Loader/ Dynamic Linker
- PageTables + GDT + IDT execute on MMU

# "Hacking is a practical study of computational models' limits"

- "What Church and Turing did with theorems, hackers do with exploits"
- Great exploits (and effective defenses!) reveal truths about the target's **actual** computational model/properties.

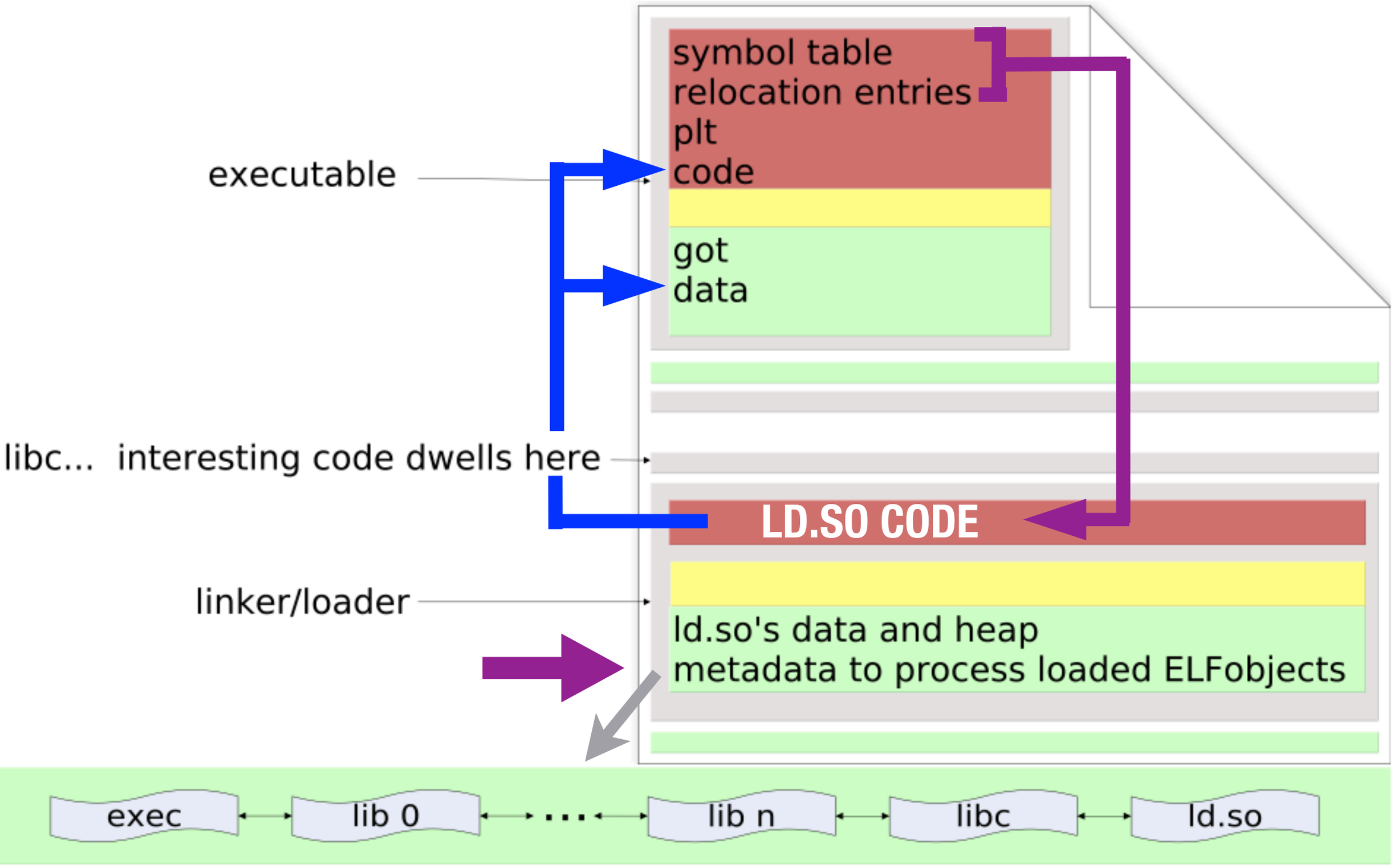
# Intro Example:
# ABI Metadata Machines

Sarah Inteman/John Kiehl

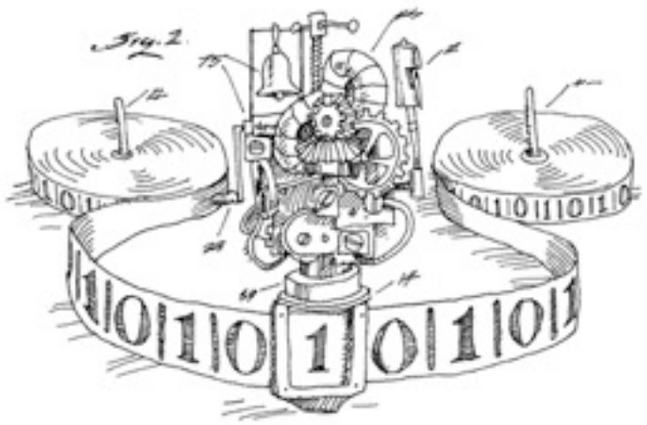
# Hacker research inspirations

- "Backdooring binary objects, **klog** [Phrack 56:9]
- "Cheating the ELF", **the grugq** [also Phrack 58:5]
- PLT redirection, **Silvio Cesare** [Phrack 56:7, ...]
- Injecting objects, **mayhem** [Phrack 61:8]
- ElfSh/**ERESI team**, http://eresi-project.org/
- LOCREATE, **skape** [Uninformed 6, 2007]
  - *Rewriting (unpacking) of binaries using REL**

# ELF relocation machine



# ELF relocation machine



# ELF metadata machines

Relocations + symbols:
a **program** in ABI for automaton to patch
images loaded at a different virtual address:

```
typedef struct {
    Elf64_Addr r_offset;
    uint64_t   r_info; // contains type and symbol number
    int64_t    r_addend;
} Elf64_Rela;
```

```
typedef struct {
    uint32_t      st_name;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t      st_shndx;
    Elf64_Addr    st_value;
    uint64_t      st_size;
} Elf64_Sym;
```

```
Num:          Value          Size  Type    Bind    Vis      Ndx Name
7407: 0000000000376d98     8  OBJECT  GLOBAL DEFAULT  31  stdin
7408: 00000000000525c0    42  FUNC    GLOBAL DEFAULT  12  putc
```

# Symbol Relocation: Like a VM

```
typedef struct {
    Elf64_Addr r_offset;
    uint64_t   r_info; // contains type and symbol number
    int64_t    r_addend;
} Elf64_Rela;
```

| Name | Value | Field | Calculation |
|---|---|---|---|
| R_386_NONE | 0 | none | none |
| R_386_32 | 1 | *word32* | S + A |
| R_386_PC32 | 2 | *word32* | S + A - P |
| R_386_GOT32 | 3 | *word32* | G + A - P |
| R_386_PLT32 | 4 | *word32* | L + A - P |
| R_386_COPY | 5 | none | none |
| R_386_GLOB_DAT | 6 | *word32* | S |
| R_386_JMP_SLOT | 7 | *word32* | S |
| R_386_RELATIVE | 8 | *word32* | B + A |
| R_386_GOTOFF | 9 | *word32* | S + A - GOT |
| R_386_GOTPC | 10 | *word32* | GOT + A - P |

```
  R_X86_64_COPY:
memcpy(r.r_offset, s.st_value, s.st_size)
  R_X86_64_64:
*(base+r.r_offset) = s.st_value +
                     r.r_addend + base
  R_X86_64_RELATIVE:
*(base+r.r_offset) = r.r_addend+base
```

RTLD code that processes relocations looks very much like implementation of a VM's bytecode

# RTLD Conditional Branching

Special **IFUNC** symbol type for indirectly linked functions: treated as function pointer iff **st_shndx** != 0

```
#include <stdio.h>
int foo (void) __attribute__ ((ifunc ("foo_ifunc")));
static int global = 1;
static int f1 (void) { return 0; }
static int f2 (void){ return 1; }
void *foo_ifunc (void) { return global == 1 ? f1 : f2; }
int main () { printf ("%d\n", foo()); }
```

Resolution function called at runtime (if st_shndx)

```
43: 0000000000400524   11 FUNC    LOCAL  DEFAULT   13 f1
44: 000000000040052f   11 FUNC    LOCAL  DEFAULT   13 f2
57: 000000000040053a   29 FUNC    GLOBAL DEFAULT   13 foo_ifunc
62: 000000000040053a   29 IFUNC   GLOBAL DEFAULT   13 foo
```

# “If” special: STT_IFUNC

- Special symbol type for indirect functions; st_value is treated as function pointer
- IFUNC symbol only processed as function if st_shndx != 0 [will use it for cond. branches]

```
#include <stdio.h>
int foo (void) __attribute__ ((ifunc ("foo_ifunc")));
static int global = 1;
static int f1 (void) { return 0; }
static int f2 (void){ return 1; }
void *foo_ifunc (void) { return global == 1 ? f1 : f2; }
int main () { printf ("%d\n", foo()); }
```

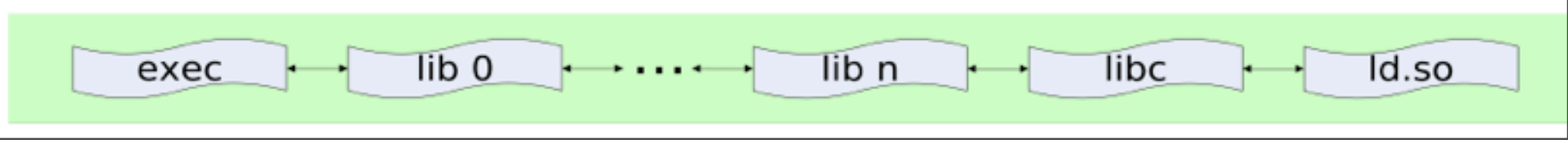
```
43: 0000000000400524    11 FUNC    LOCAL  DEFAULT    13 f1
44: 000000000040052f    11 FUNC    LOCAL  DEFAULT    13 f2
57: 000000000040053a    29 FUNC    GLOBAL DEFAULT    13 foo_ifunc
62: 000000000040053a    29 IFUNC   GLOBAL DEFAULT    13 foo
```

# Unconditional Branches [Loops]

*Relocator iterates through REL entries, marking them "done". Make it loop & unmark, by pointing REL entries at link map's structures!*

```
do
 {
   struct libname_list *lnp = l->l_libname->next;

   while (__builtin_expect (lnp != NULL, 0))
    {
      lnp->dont_free = 1;
      lnp = lnp->next;
    }

   if (l != &GL(dl_rtld_map))
     _dl_relocate_object (l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                          consider_profiling);
   ...
   l = l->l_prev;
 }
while (l);
```

TODO:
- set l->l_prev = l



# Unconditional Branches [Loops]

```
void
_dl_relocate_object (struct link_map *l, struct r_scope_elem *scope[],
                     int reloc_mode, int consider_profiling)
{
  if (l->l_relocated)
    return;
  ...
    ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
  ...
  /* Mark the object so we know this work has been done.  */
  l->l_relocated = 1;
  ...
  /* In case we can protect the data now that the relocations are
     done, do it.  */
  if (l->l_relro_size != 0)
    _dl_protect_relro (l);
  ...
}
```

TODO:
- set l->l_prev = l
- fix l->l_relocated

# Unconditional Branches [Loops]

```
void
_dl_relocate_object (struct link_map *l, struct r_scope_elem *scope[],
                     int reloc_mode, int consider_profiling)
{

  if (l->l_relocated)
    return;

  ...
    ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
  ...
  /* Mark the object so we know this work has been done.  */
  l->l_relocated = 1;

  ...
  /* In case we can protect the data now that the relocations are
     done, do it.  */
  if (l->l_relro_size != 0)
    _dl_protect_relro (l);
  ...
}
```

TODO:
- set l->l_prev = l
- fix l->l_relocated
- set l->l_relro_size = 0

# Unconditional Branches [Loops]

```
do
 {
   struct libname_list *lnp = l->l_libname->next;

   while (__builtin_expect (lnp != NULL, 0))
    {
      lnp->dont_free = 1;
      lnp = lnp->next;
    }

   if (l != &GL(dl_rtld_map))
     _dl_relocate_object (l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                          consider_profiling);

   ...
   l = l->l_prev;
 }
while (l);
```

TODO:
- set l->l_prev = l
- **fix l->l_relocated**
- set l->l_relro_size = 0

# Unconditional Branches [Loops]

Full TODO:

Note: libc/ld.so version-dependent!

Fix l->l_relocated

Set l->l_prev = l

Set l->l_relro_size = 0

Set l->l_info[DT_RELA] = &next rel to process

Fix l->l_info[DT_RELASZ]

# Unconditional Branches [Loops]

- Fix l->l_relocated
  **{offset =&(l->l_buckets), type = RELATIVE, addend=0}**
  **{offset =&(l->l_direct_opencount), type = RELATIVE, addend=0}**
  **{offset =&(l->l_libname->next), type = RELATIVE,**
  **addend=&(l->l_relocated) + 4*sizeof(int)}**

- Set l->l_prev = l
  **{offset =&(l->l_prev), type = RELATIVE, addend=&l}**

- Set l->l_relro_size = 0 (etc.)

- Set l->l_info[DT_RELA] = &next rel to process

- Fix l->l_info[DT_RELASZ]

# Unconditional Branches [Loops]

Finally, ending the loop (skipping past remaining relocation entries):

{offset =&end, type = RELATIVE, addend=0}

(end stored on stack, set to 0 )

```
for (; r < end; ++r)
  {
    ElfW(Half) ndx = version[ELFW(R_SYM) (r->r_info)] & 0x7fff;
    elf_machine_rel (map, r, &symtab[ELFW(R_SYM) (r->r_info)],
                     &map->l_versions[ndx],
                     (void *) (l_addr + r->r_offset));
  }
```

# Program in ELF metadata

## .dynsym table

(empty)

Original dynsym 0

Original dynsym 1

...

Original dynsym n

Address tape head is pointing at

Copy of tape head's value

Address of previous sym's value

IFUNC of gadget that returns 0

(A few other symbols needed for bookkeeping, like library addrs)

## .rela.dyn table

Brainfuck instruction 0

...

Brainfuck instruction n

Instructions that clean up link_map data

Instructions to force branch to next rel entry

Instructions to finish cleaning link_map data

Original .rela.dyn entry 0

...

Original .rela.dyn entry m

# What can this do?

- NB: dependent on the Glibc version (but there aren't that many)
- Traverse **ld.so**'s link_map of all dynamically loaded libraries (under ASLR)
- Resolve symbols & addresses of instructions
- Modify **GOT**
- Redirect code execution (e.g., obfuscate backdoors & other control flow) without affecting "native code" sections.

# Hiding “code” in ping

**Ping** runs **suid root**, sneak “code” into its ELF metadata:

- Given ”-t <string>”
- Usage: -t, --type=TYPE  send TYPE packets
- Code: if(strcasecmp (<string>, "echo") == 0) ...

**Goals**:

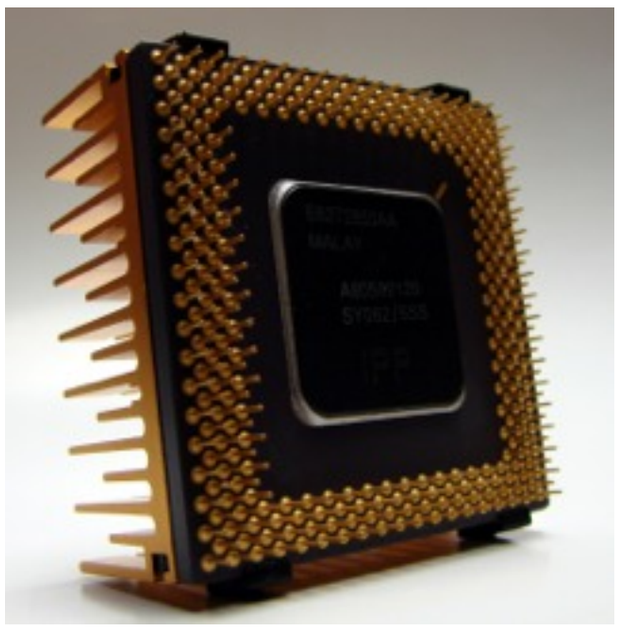
- Redirect call to strcasecmp to execl:  edit GOT entries
- Prevent call to setuid that drops root privileges: point to retq
- Work in presence of library randomization (ASLR): find base address of glibc at runtime by traversing link_map

```
Relocation section '.rela.p' at offset 0xf3a8 contains 14 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend

00000060dfe0  002d00000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
00000060e9e0  004e00000005 R_X86_64_COPY     000000000060e9e0 __progname + 0
00000060e9f0  004b00000005 R_X86_64_COPY     000000000060e9f0 stdout + 0
00000060e9f8  005100000005 R_X86_64_COPY     000000000060e9f8 __progname_full + 0
00000060ea00  005600000005 R_X86_64_COPY     000000000060ea00 stderr + 0
00000060eb40  000000000005 R_X86_64_COPY                     0000000000000000
00000060eb40  000000000001 R_X86_64_64                       0000000000000018
00000060eb40  000000000005 R_X86_64_COPY                     0000000000000000
00000060eb40  000000000001 R_X86_64_64                       0000000000000018
00000060eb40  000000000005 R_X86_64_COPY                     0000000000000000
00000060eb40  000000000005 R_X86_64_COPY                     0000000000000000
00000060eb40  000000000001 R_X86_64_64                       00000000000be6e0
00000060e028  000000000001 R_X86_64_64                       0000000000000000
00000060e218  000000000008 R_X86_64_RELATIVE                 000000000401dc2

Symbol table '.sym.p' contains 90 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 000000000060dff0     8 FUNC    LOCAL  DEFAULT  UND
```

See 29c3 talk by Rebecca "**.bx**" Shapiro,

**https://github.com/bx/elf-bf-tools**

# Page Fault Liberation Army (PFLA)*



*“Input is (still) a program!”*

*) In x86 manuals PFLA stands for “Page Faulting Linear Address”, but this sounds more fashionable

# Let's take an old and known thing...

(Young James Watt was often berated for idly watching boiling kettles)

# ...and see how far we can make it go!

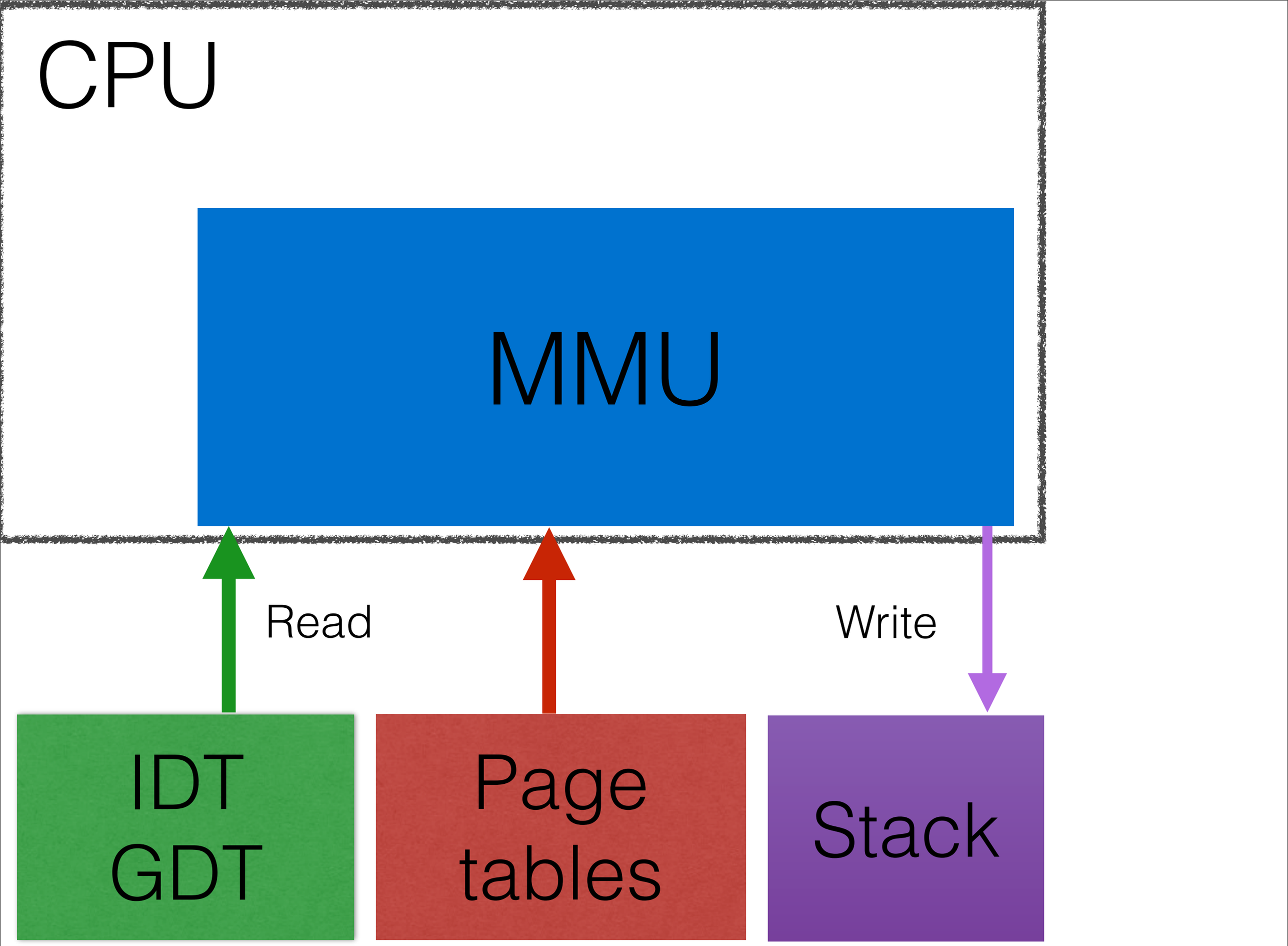
# (And where it gets stuck)

# “Page Fault Liberation”

- The x86 MMU is not just a look-up table!
- x86 MMU performs complex logic on **complex data structures**
- The MMU has **state** and **transitions** that brilliant hackers put to unorthodox uses.
- Can it be **programmed** with its input data?

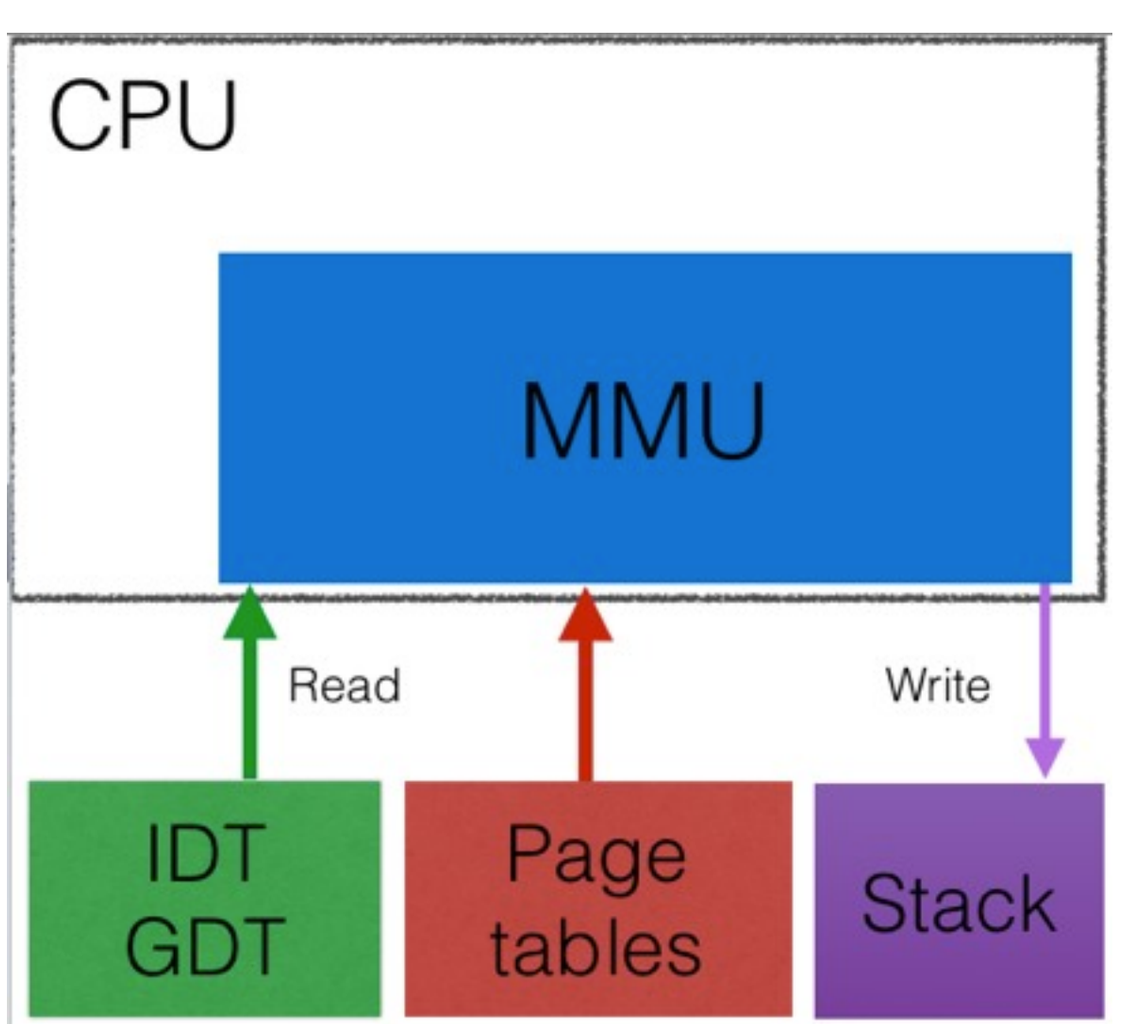
# Traps are important

- Not usually a part of a program (unless it's OS or debugger), but ***can*** be: **same** memory space
- Traps weave between hardware/microcode & software + data tables; e.g., hw reads **page tables**, writes **stack**, sets up **TLB** entries
- Set up with regs & complex tables in **RAM**
- *How much computing power is in that?*


CPU
MMU
Read
Write
IDT
GDT
Page
tables
Stack

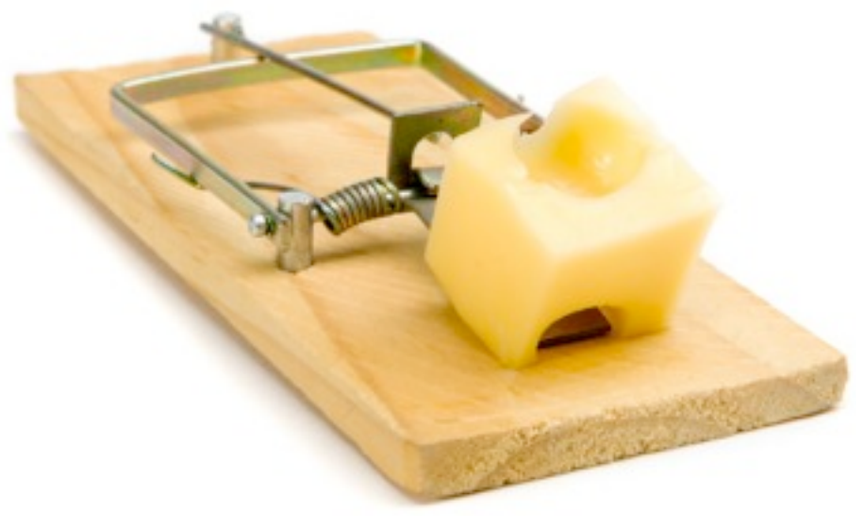
- unmapped/bad memory reference **trap**, based on **page tables** & (current) **IDT**
- hardware **writes fault info** on the stack - where it **thinks** the stack is (address in TSS)

- If we point "**stack**" into **page tables, GDT** or TSS, can we get the "tape" of a Turing machine?

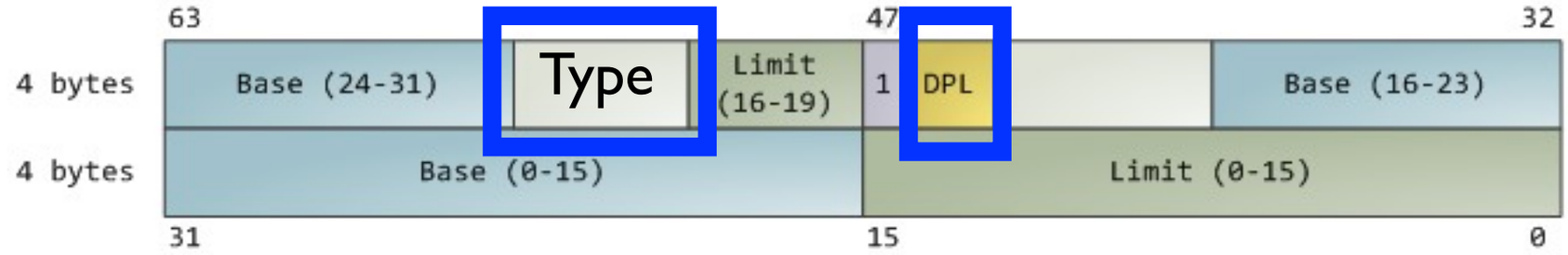


The devil's in the ~~details~~

trapping bits



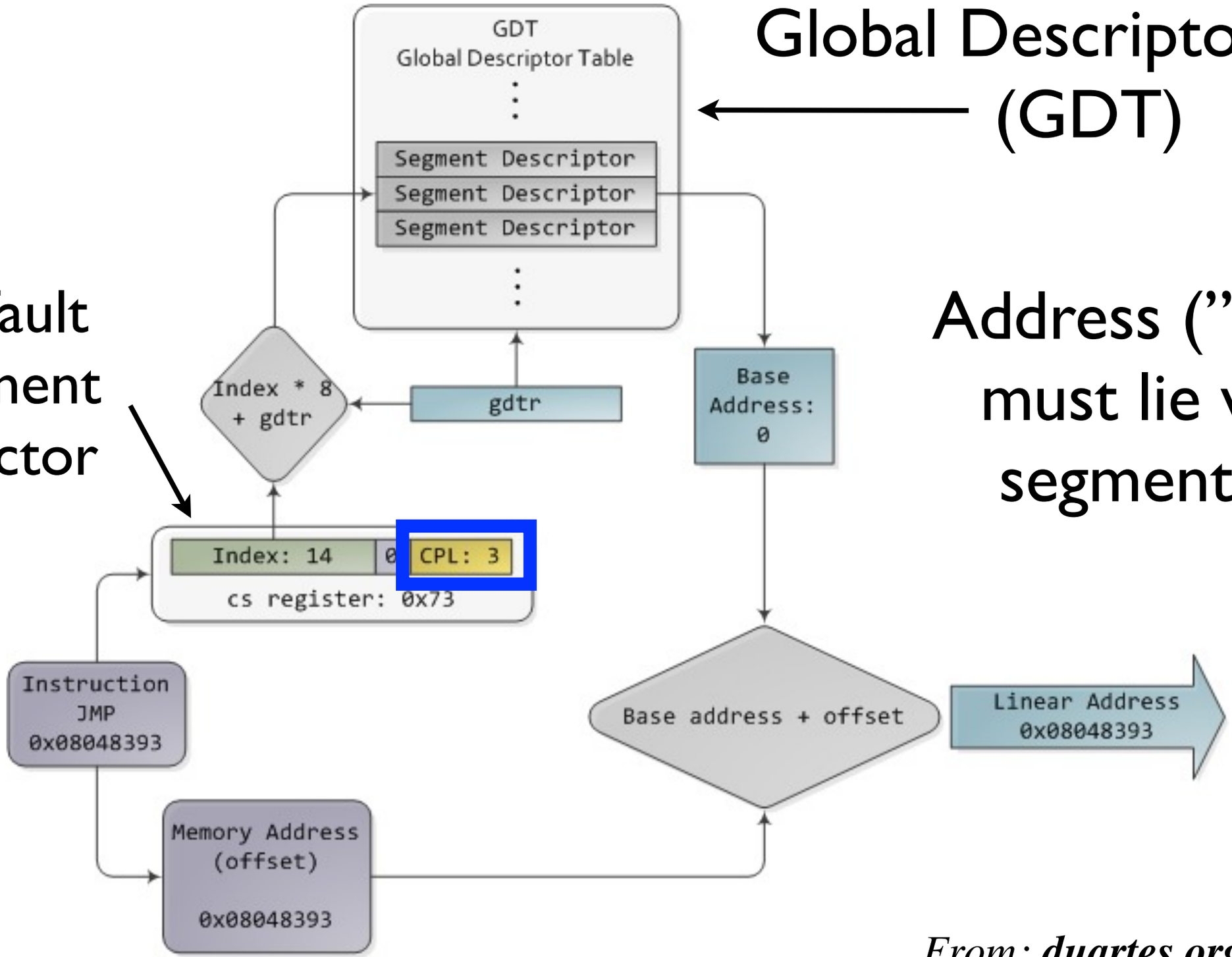
Segment descriptor:




Global Descriptor Table (GDT)

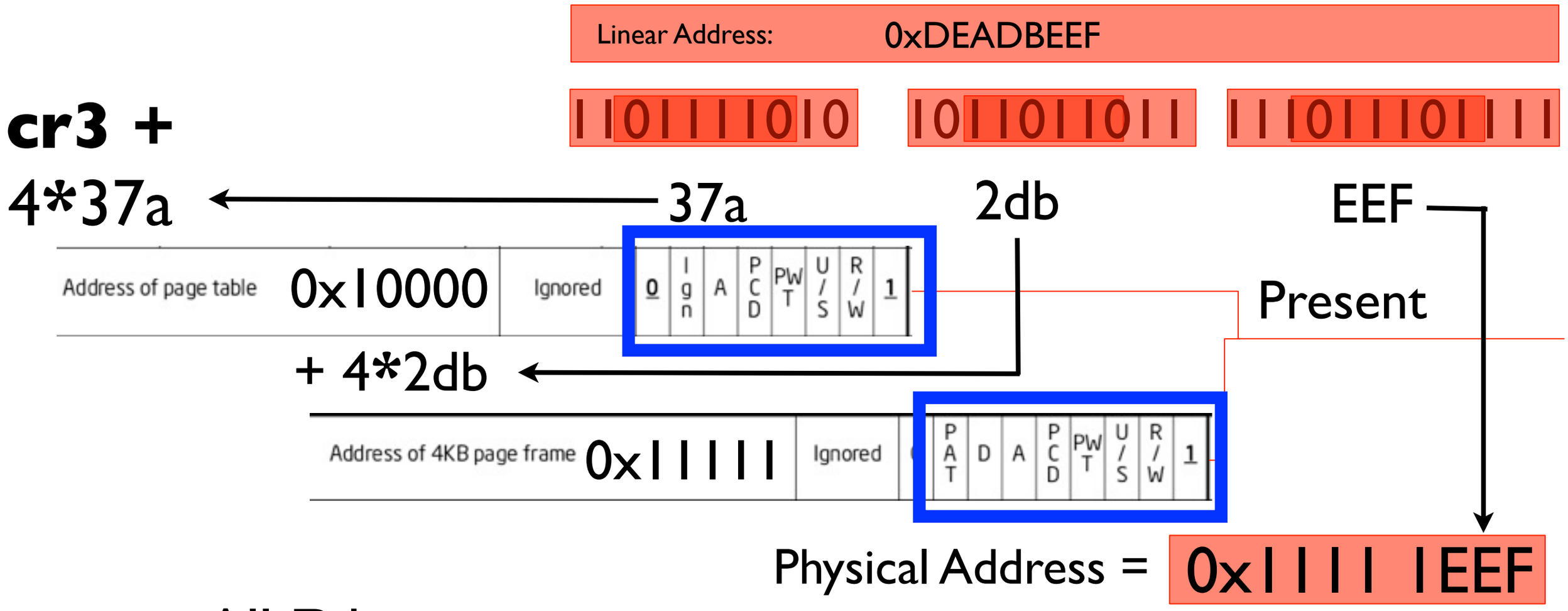
Default segment selector

Address (”offset”) must lie within segment limit




*From:* ***duartes.org/gustavo/blog/***

# Virtual Address Translation




- All **P** bits set
- Ring 3: All **U/S** bits have to be set
- Write: All **R/W** bits have to be set
- What if we violate these rules?

# ITS A TRAP

# Hidden state in MMU?

- You bet! And some of it can be "programmed"
- **Inspiration**: PaX's PAGEEXEC, emulation of the **NX** bit on all CPUs since Pentium
- Data page accesses trap when instructions are fetched from them; see http://pax.grsecurity.net/docs/pageexec.old.txt [cf. Plex86, 1999]

# Trap-hacks "Design Patterns"

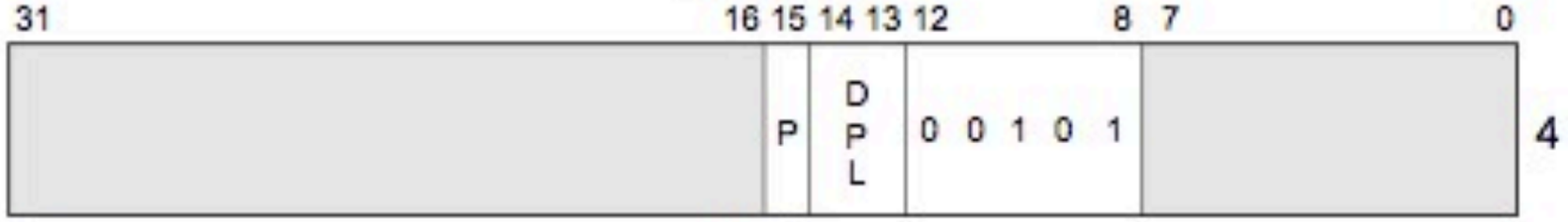
- **Overloading #PF** for security policy, labeling memory (e.g., PaX, OpenWall)
- **Combining** traps to trap on more complex events (OllyBone, "fetch from a page just written")
- Using **several** trap bits in different locations to label memory for **data flow** control (PaX UDEREF, SMAP/SMEP use)
- Storing **extra state** in TLBs (PaX PageExec)
- "Unorthodox" breakpoints, control flow, ...

# What's in a trap handler (let's roll our own)

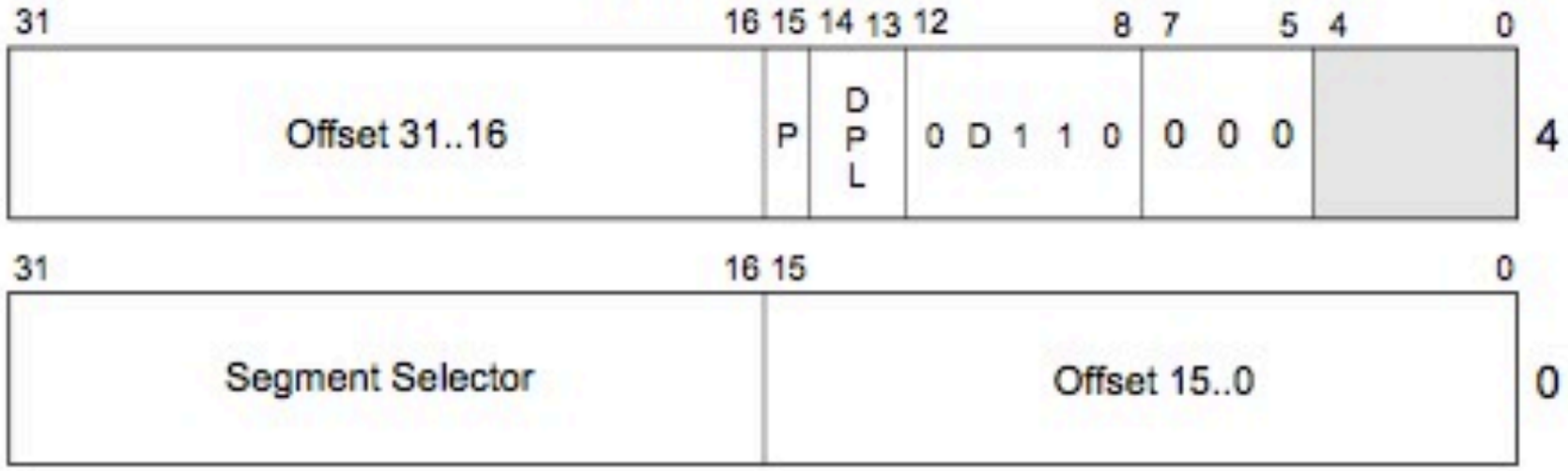
IDT entries:

...
8: #DF
...
14: #PF
...

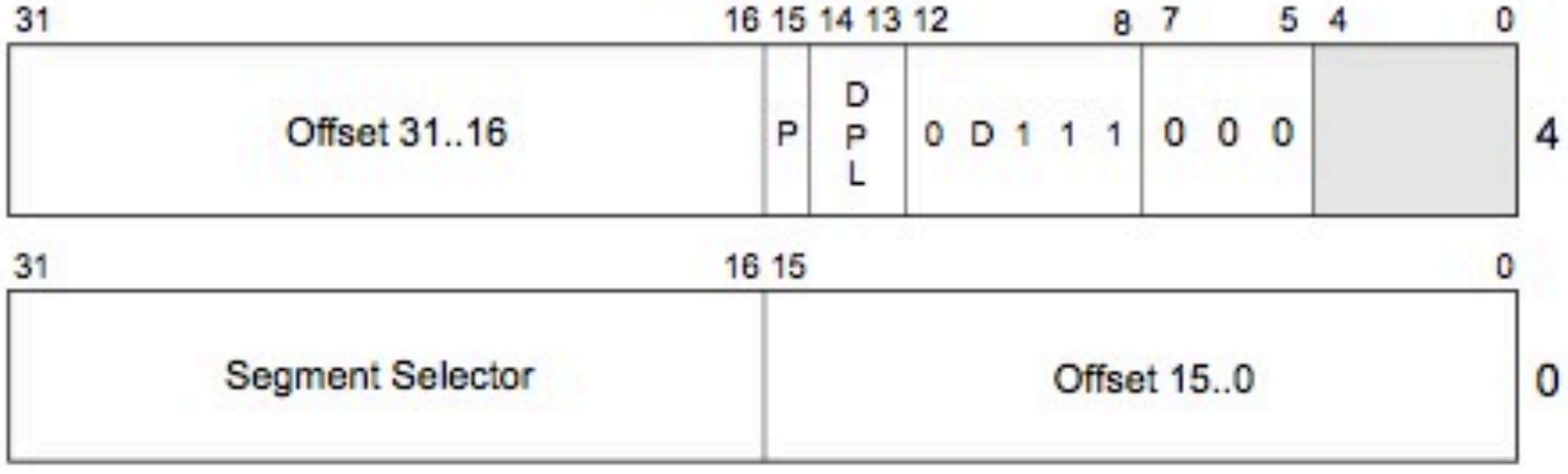
**Task Gate**



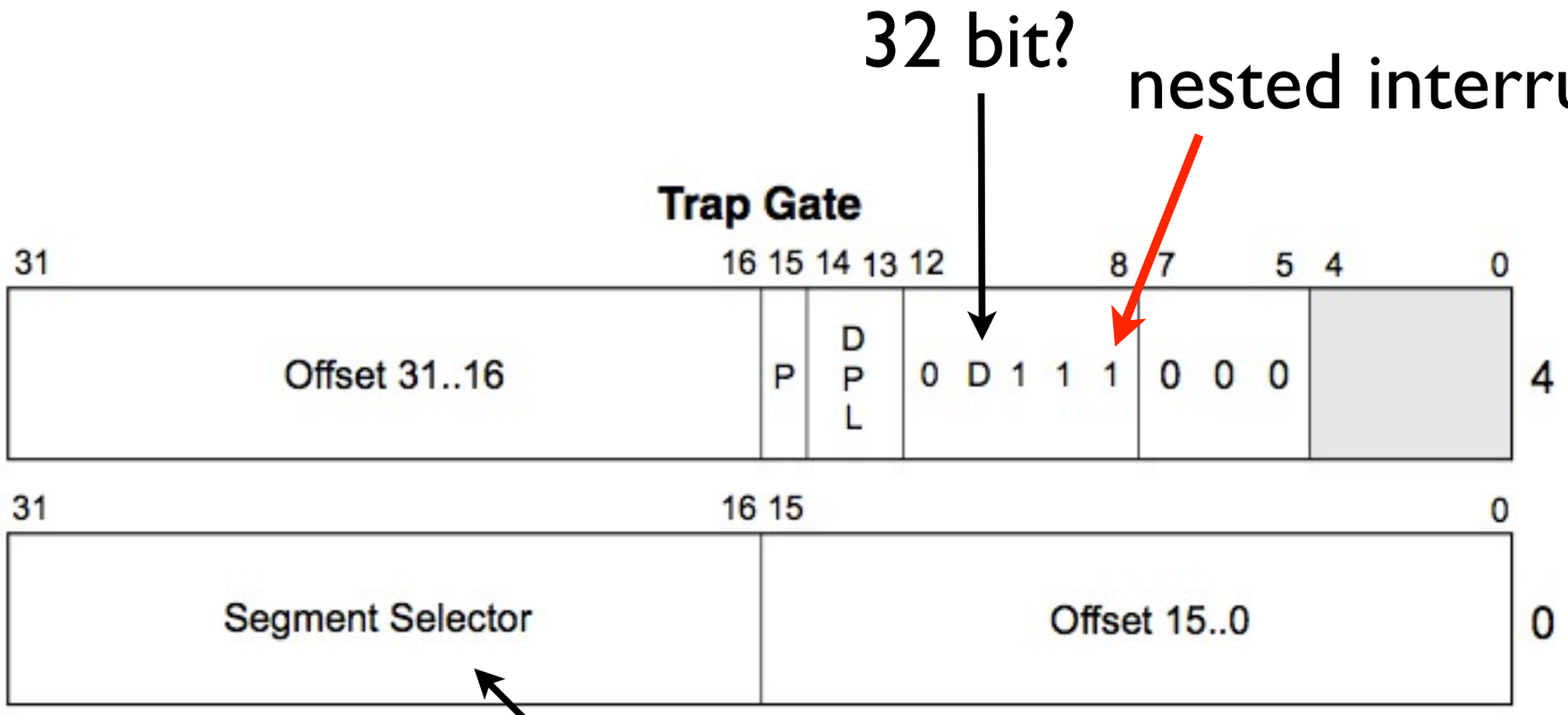

**Interrupt Gate**




**Trap Gate**



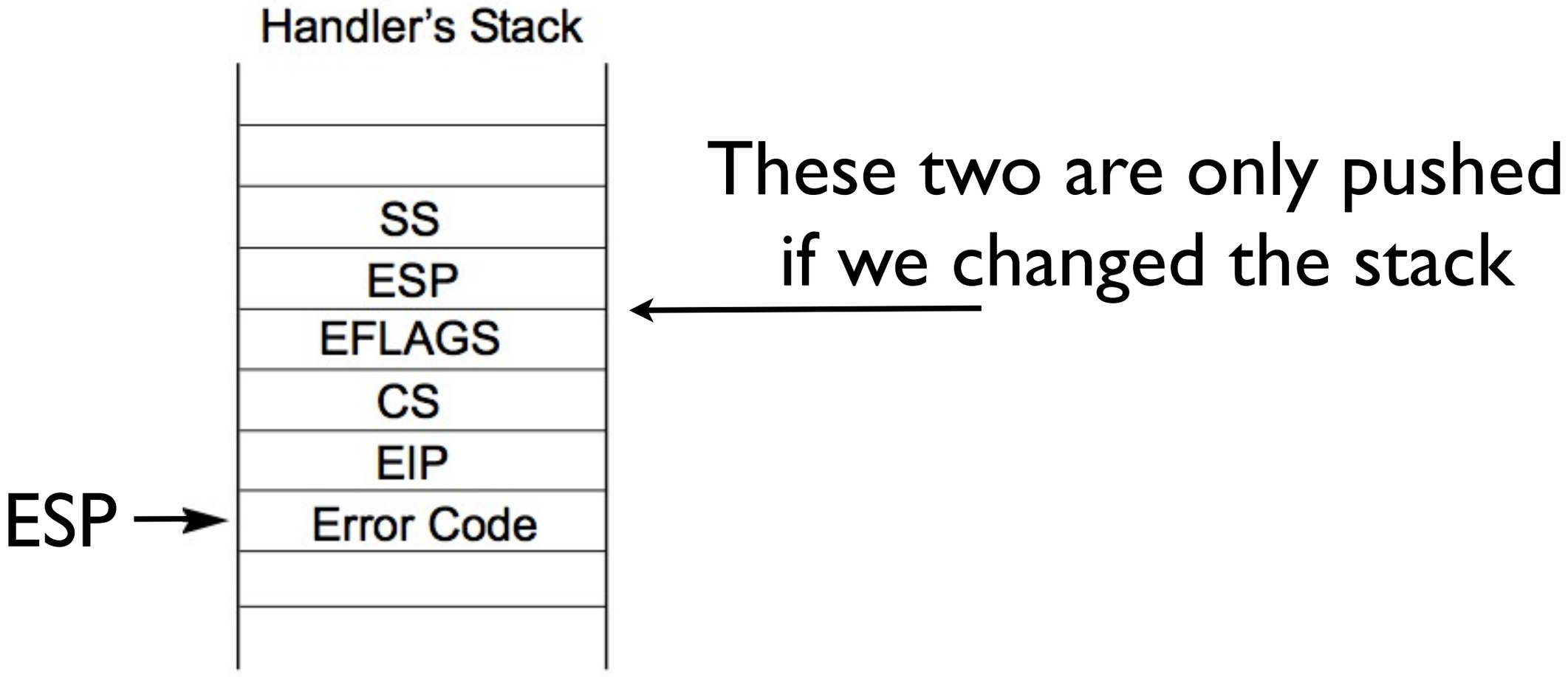
# Call through a Trap Gate




New code segment

Like a FAR call of old. If the new segment is in a lower (i.e. higher privilege) Ring, we load a new SP.

# Pushes parameters to "handler's stack"




"IRET" instruction can return from this

# What if this fails?

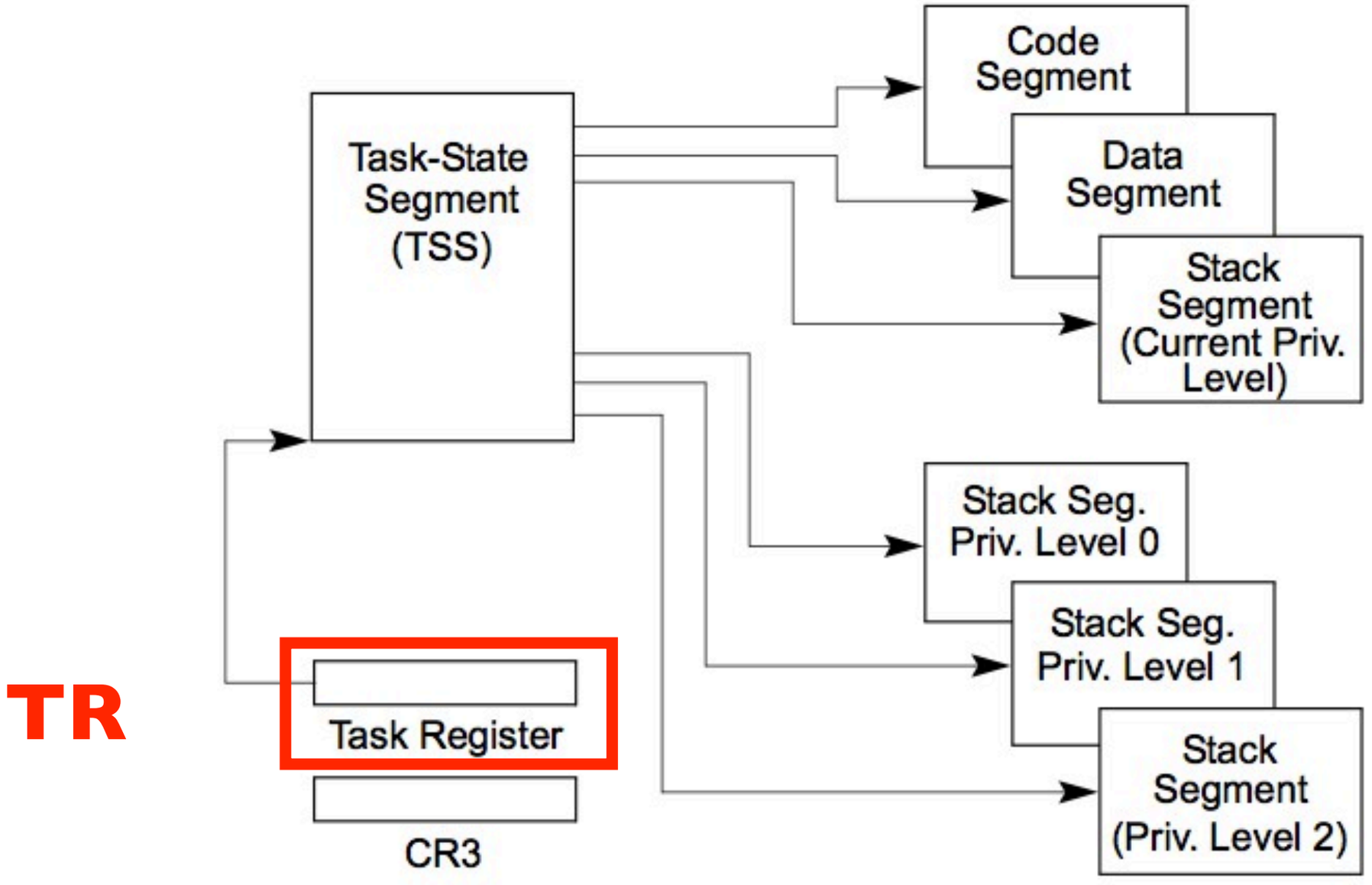
- Stack invalid?
- Code segment invalid?
- IDT entry not present?

Causes "**Double Fault**"(#8). "Triple fault" = Reboot

Usually **#DF** means OS bug, so a lot of state might be corrupted (i.e. invalid kernel stack)

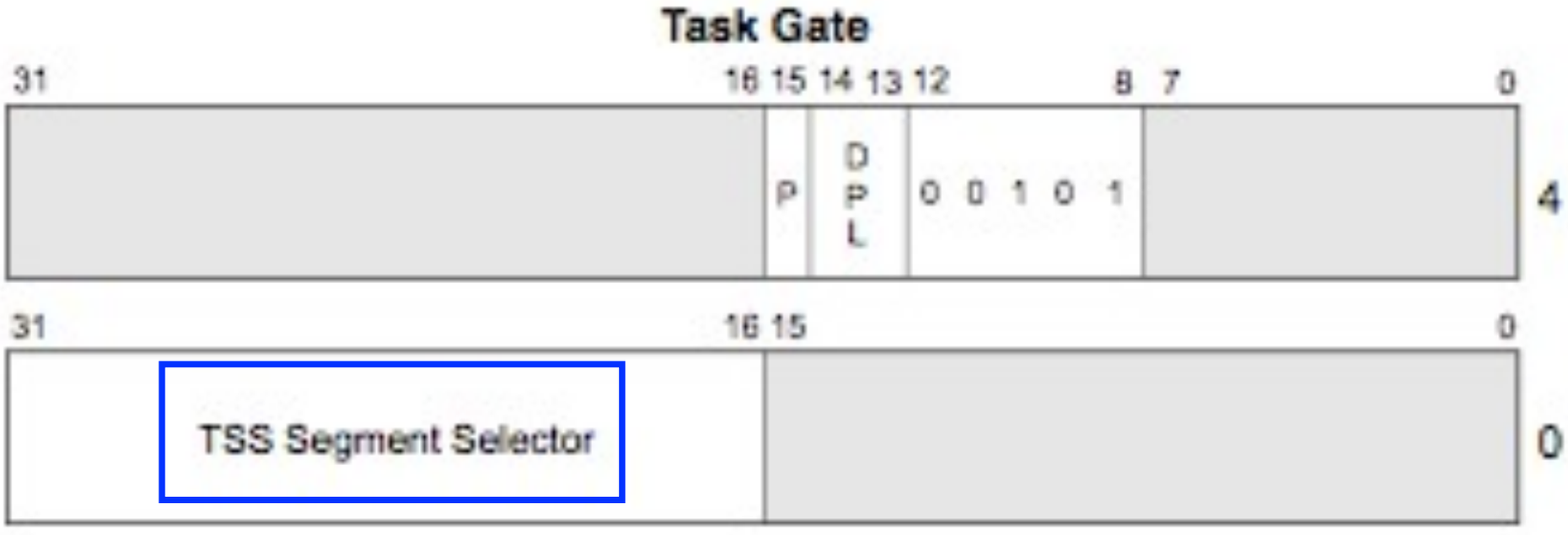
# Hardware Task Switching

**We can use it for #PF and #DF traps instead of Trap Gates**



# Task gate

- (unused) mechanism for **hardware** tasking
- Reloads (nearly) all CPU state from memory
- Task gate causes **task switch** on **trap**



# IDT-> GDT->TSS

## It still pushes the error code

IDT

**Task Gate**

| 31 … 16 | 15 | 14 13 | 12 … 8 | 7 … 0 | |
|---|---|---|---|---|---|
| | P | DPL | 0 0 1 0 1 | | 4 |

| 31 … 16 | 15 … 0 | |
|---|---|---|
| TSS Segment Selector | | 0 |

(addressed indirectly through GDT)

GDT

**TSS Descriptor**

| 31 … 24 | 23 | 22 | 21 | 20 | 19 … 16 | 15 | 14 13 | 12 | 11 … 8 | 7 … 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Base 31:24 | G | 0 | 0 | AVL | Limit 19:16 | P | DPL | 0 | Type: 1 0 B 1 | Base 23:16 | 4 |

| 31 … 16 | 15 … 0 | |
|---|---|---|
| Base Address 15:00 | Segment Limit 15:00 | 0 |

| | |
|---|---|
| AVL | Available for use by system software |
| B | Busy flag |
| BASE | Segment Base Address |
| DPL | Descriptor Privilege Level |
| G | Granularity |
| LIMIT | Segment Limit |
| P | Segment Present |
| TYPE | Segment Type |

| 31 … 15 | 0 … | |
|---|---|---|
| I/O Map Base Address | Reserved T | 100 |
| Reserved | LDT Segment Selector | 96 |
| Reserved | GS | 92 |
| Reserved | FS | 88 |
| Reserved | DS | 84 |
| Reserved | SS | 80 |
| Reserved | CS | 76 |
| Reserved | ES | 72 |
| EDI | | 68 |
| ESI | | 64 |
| EBP | | 60 |
| ESP | | 56 |
| EBX | | 52 |
| EDX | | 48 |
| ECX | | 44 |
| EAX | | 40 |
| EFLAGS | | 36 |
| EIP | | 32 |
| CR3 (PDBR) | | 28 |
| Reserved | SS2 | 24 |
| ESP2 | | 20 |
| Reserved | SS1 | 16 |
| ESP1 | | 12 |
| Reserved | SS0 | 8 |
| ESP0 | | 4 |
| Reserved | Previous Task Link | 0 |

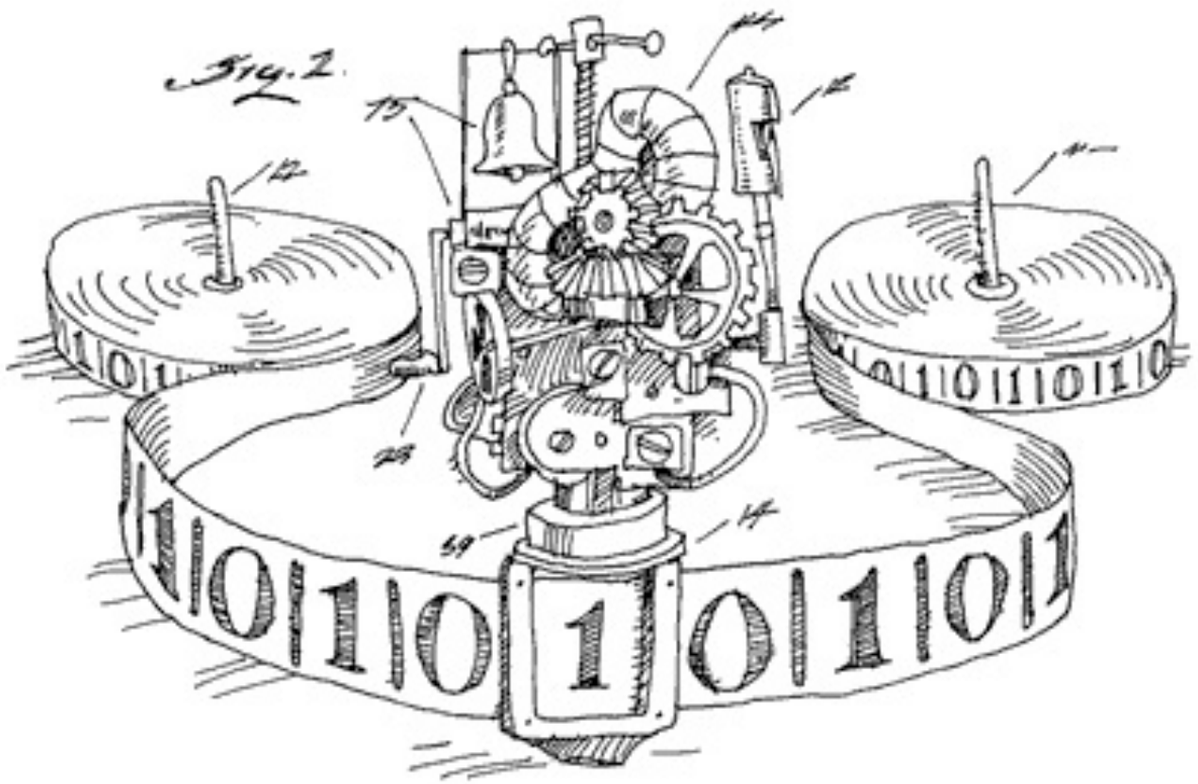
# Interrupt to Task Gate

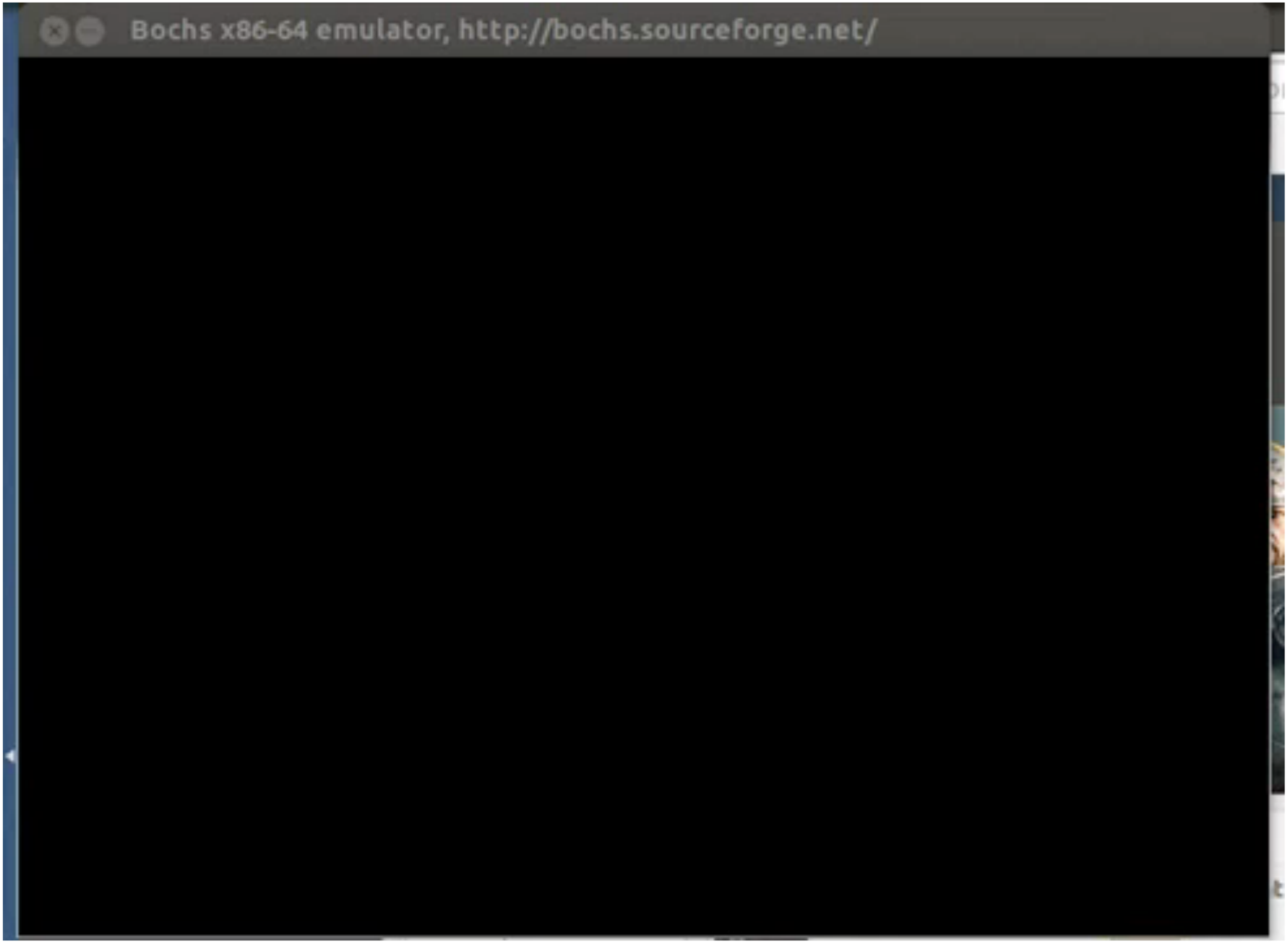
1. **Save state** to location pointed to by **TR**
2. Find Task (**GDT**), validate + check Busy=0
3. **Load** new state (EIP, CR3, stack...)
4. Push error code

Doublefault

Begin executing new EIP

# Look Ma, it's a Turing machine!




Bochs x86-64 emulator, http://bochs.sourceforge.net/

# A one-instruction machine

Instruction Format:
Label = (X <-Y,A,B)

```
Label:
    X=Y
    If X<4:
        Goto B
    Else
        X-=4
        Goto A
```

- "Decrement-Branch-If-Negative"
- Turing complete (!)
- "Computer Architecture: A Minimalist Perspective" by Gilreath and Laplathe (~$200)
- Or Wikipedia :)

# Implementation sketch:

- If EIP of a handler is pointed at invalid memory, we get another **page fault** immediately; keep EIP invalid in all tasks
- Var Decrement: use TSS' SP, pushing the stack decrements SP by 4.
- Branch: <4 or not? Implemented by **double fault** when SP cannot be decremented

# Dramatis Personae I

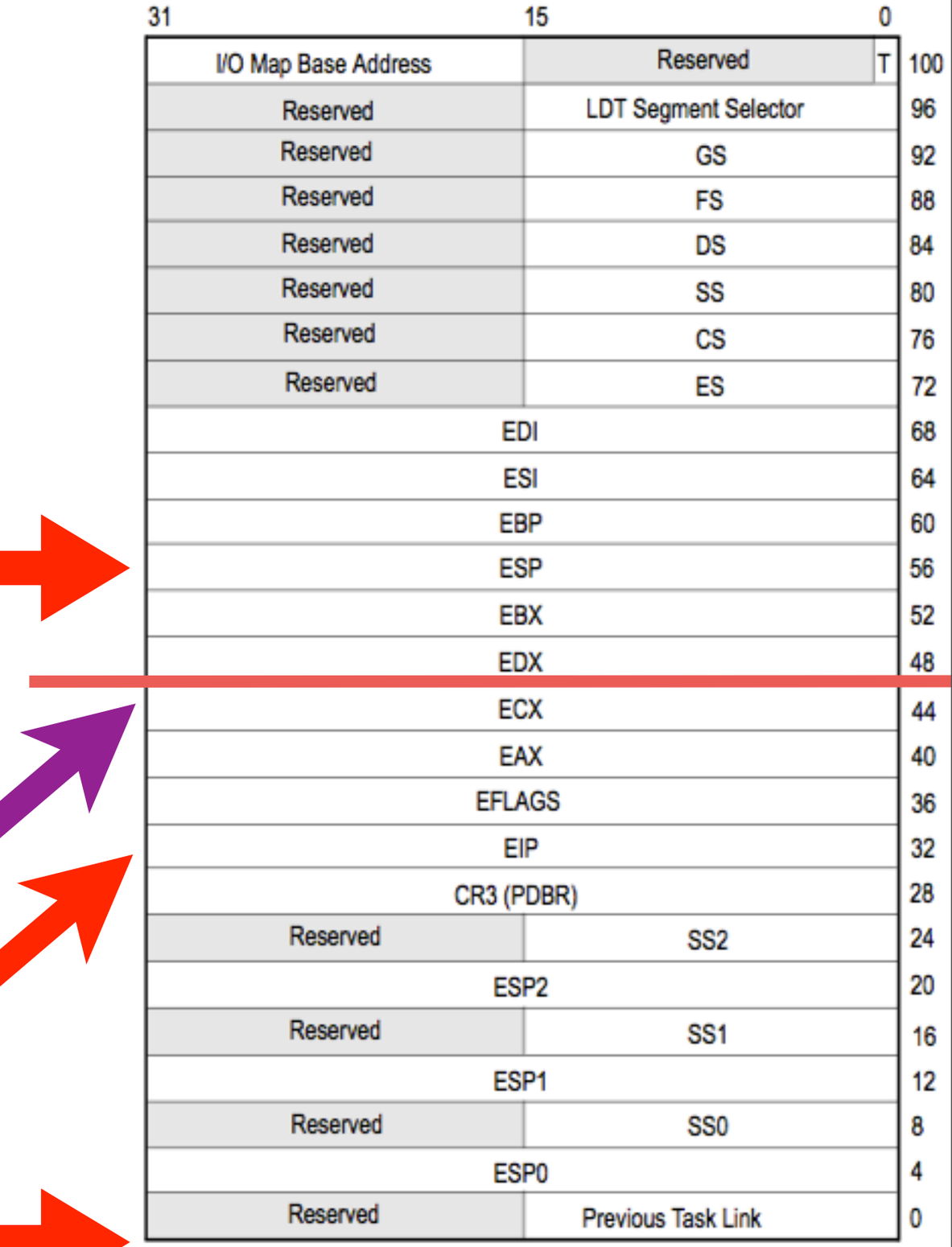
- One GDT to rule them all
- One TSS Descriptor *per instruction*, aligned with the end of a page
- IDT is mapped differently, *per instruction*
- A target (branch-not-taken) in Int 14, **#PF**
- B target (branch taken) in Int 8, **#DF**

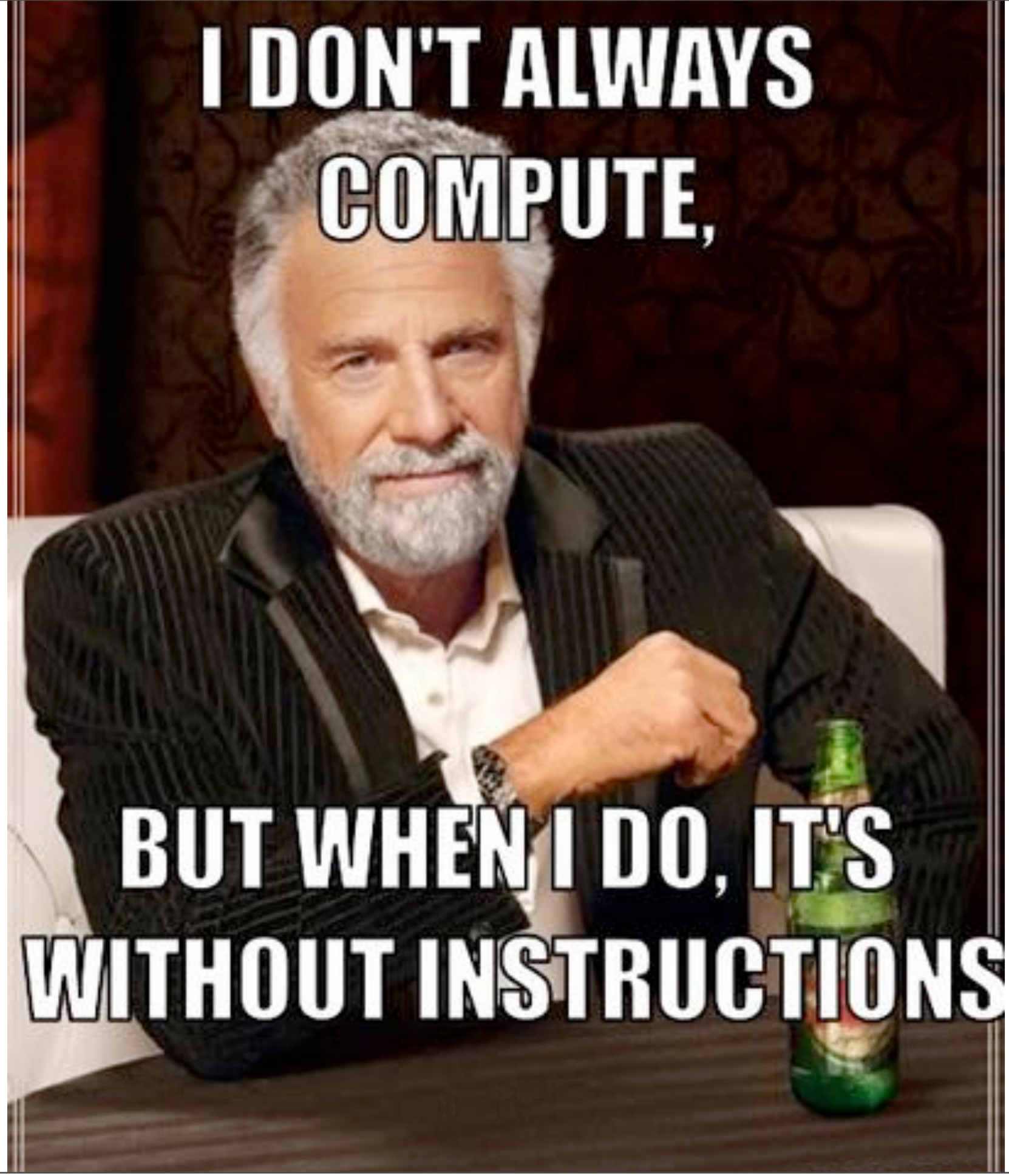
# Dramatis Personae II

- Higher half of TSS (variables)
  - Map A.Y, B.Y (the value we want to load for next instruction) at their TSS addresses
    - map X (the value we want to write) at the addr of the current task
- So we have the move and decrement

- We split these TSS across a page boundary
- Variables are stack pointer entries in a TSS
- Upper Page: ESP and segments
- Lower Page: EAX, ECX, EIP, CR3 (page tables)

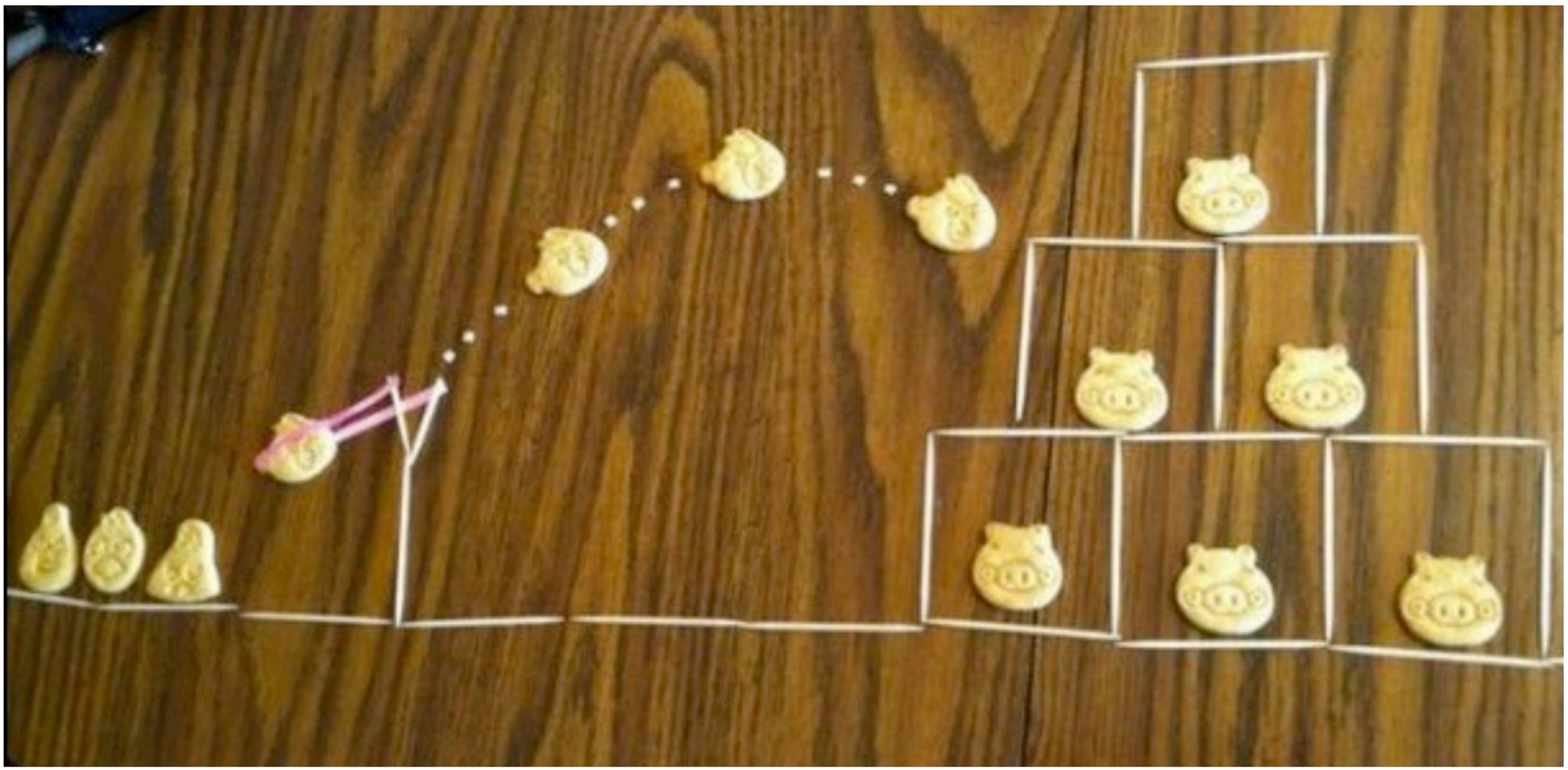
Labels: A, B, C, ...



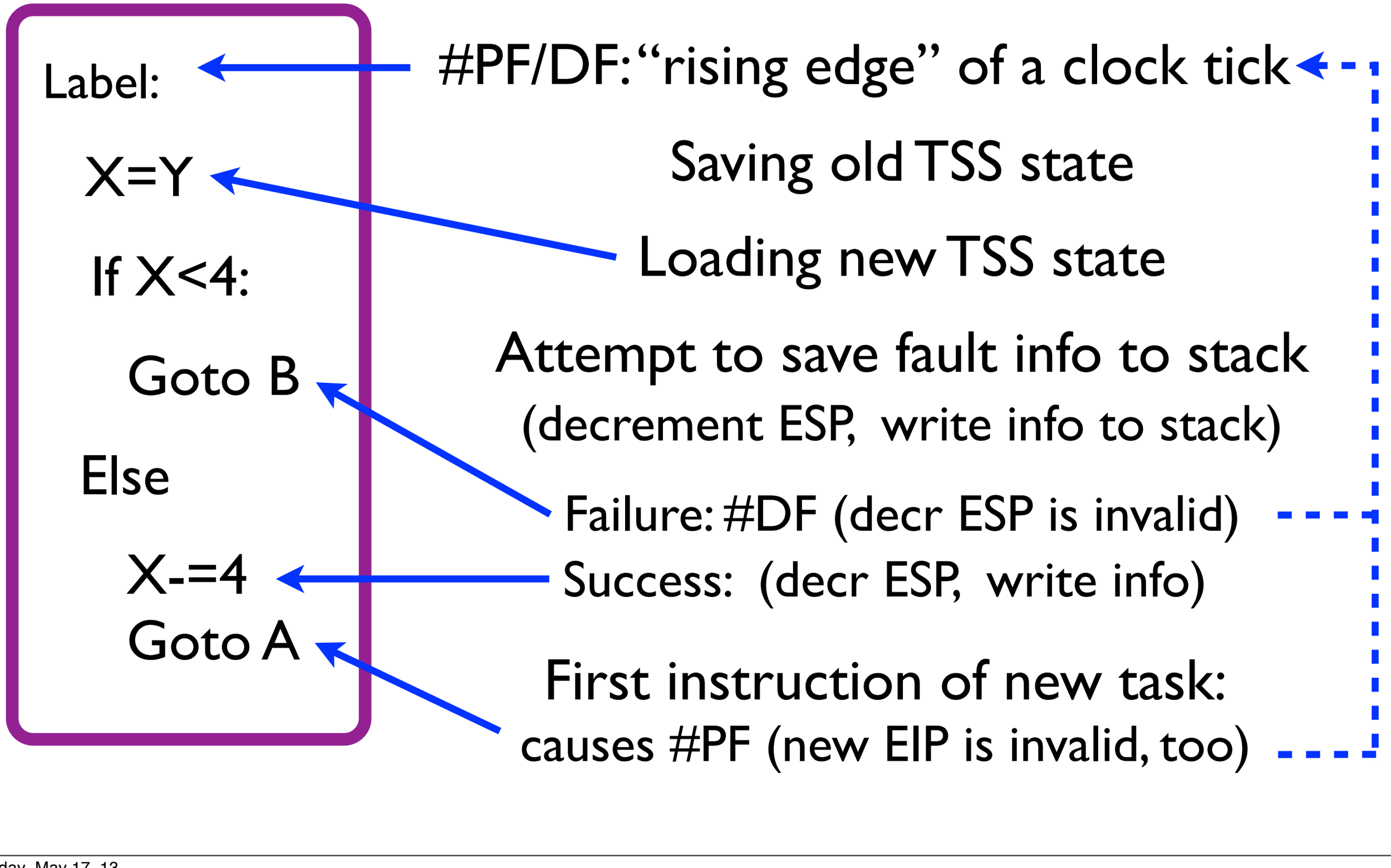

I DON'T ALWAYS
COMPUTE,
BUT WHEN I DO, IT'S
WITHOUT INSTRUCTIONS

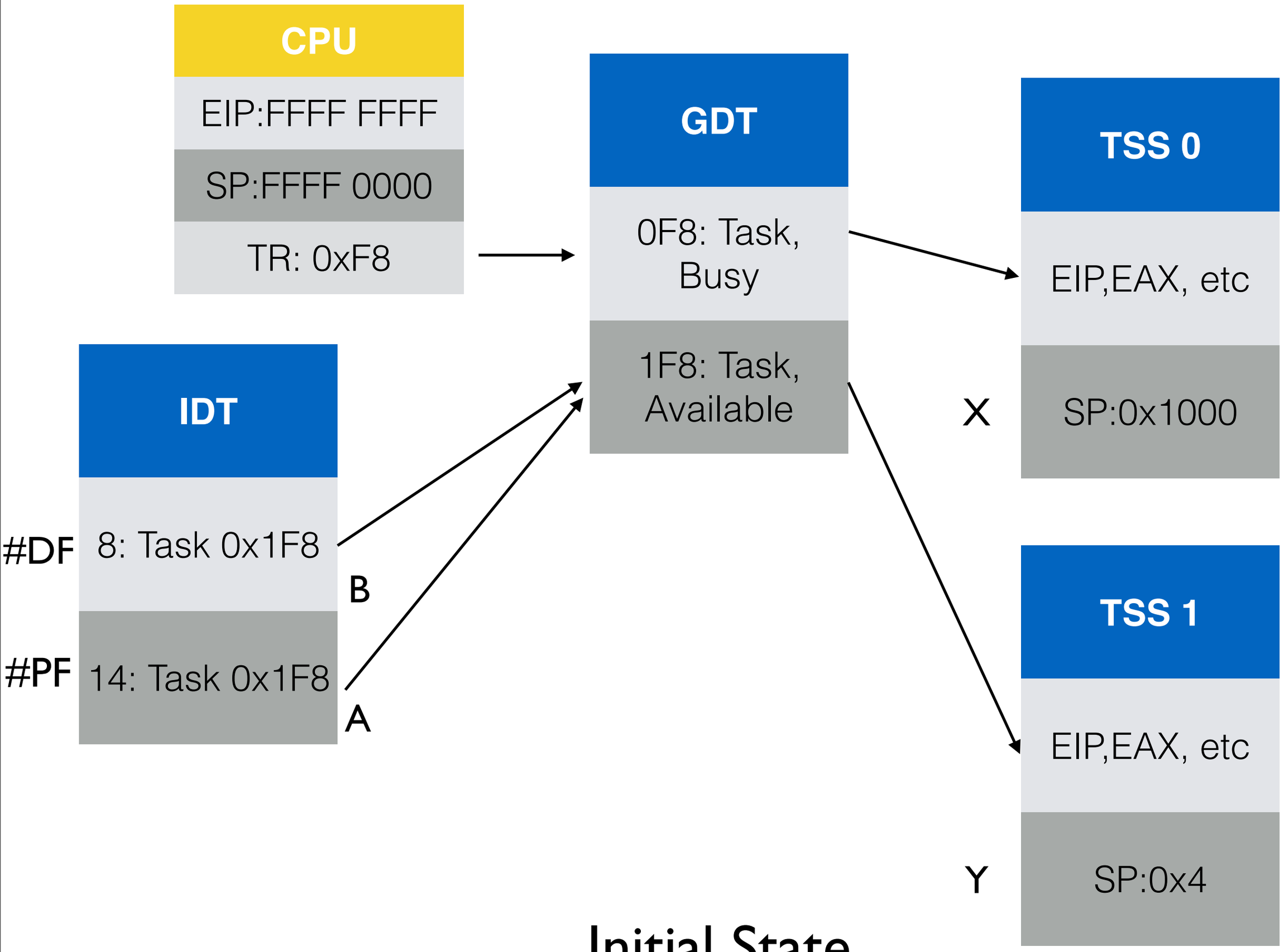
# Let's step through an instruction

(Some details glossed over;
think of it as a fairy tale, not a lie)

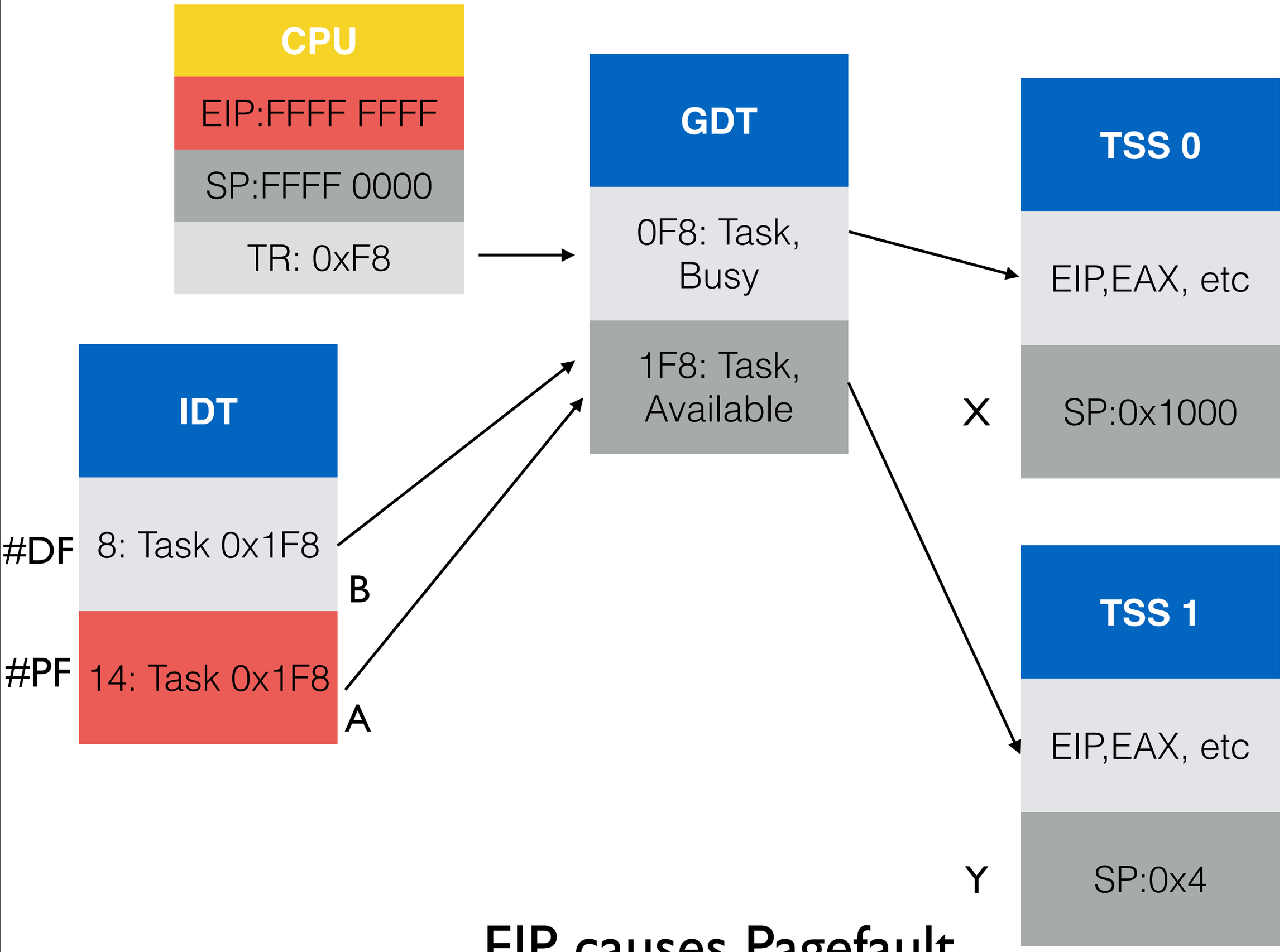


# Instruction by the numbers (or, "PFLA fetch-decode-execute" loop)

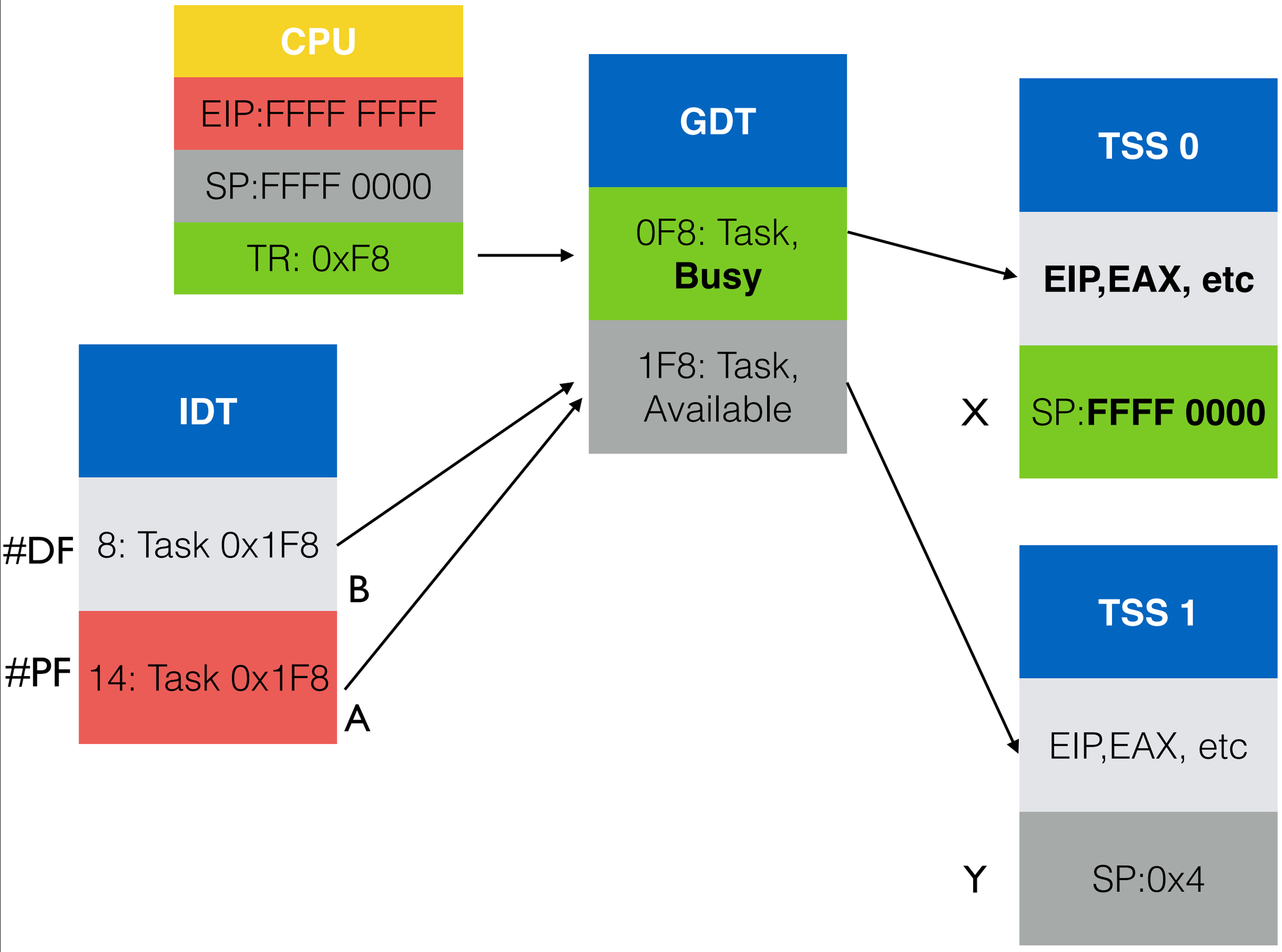




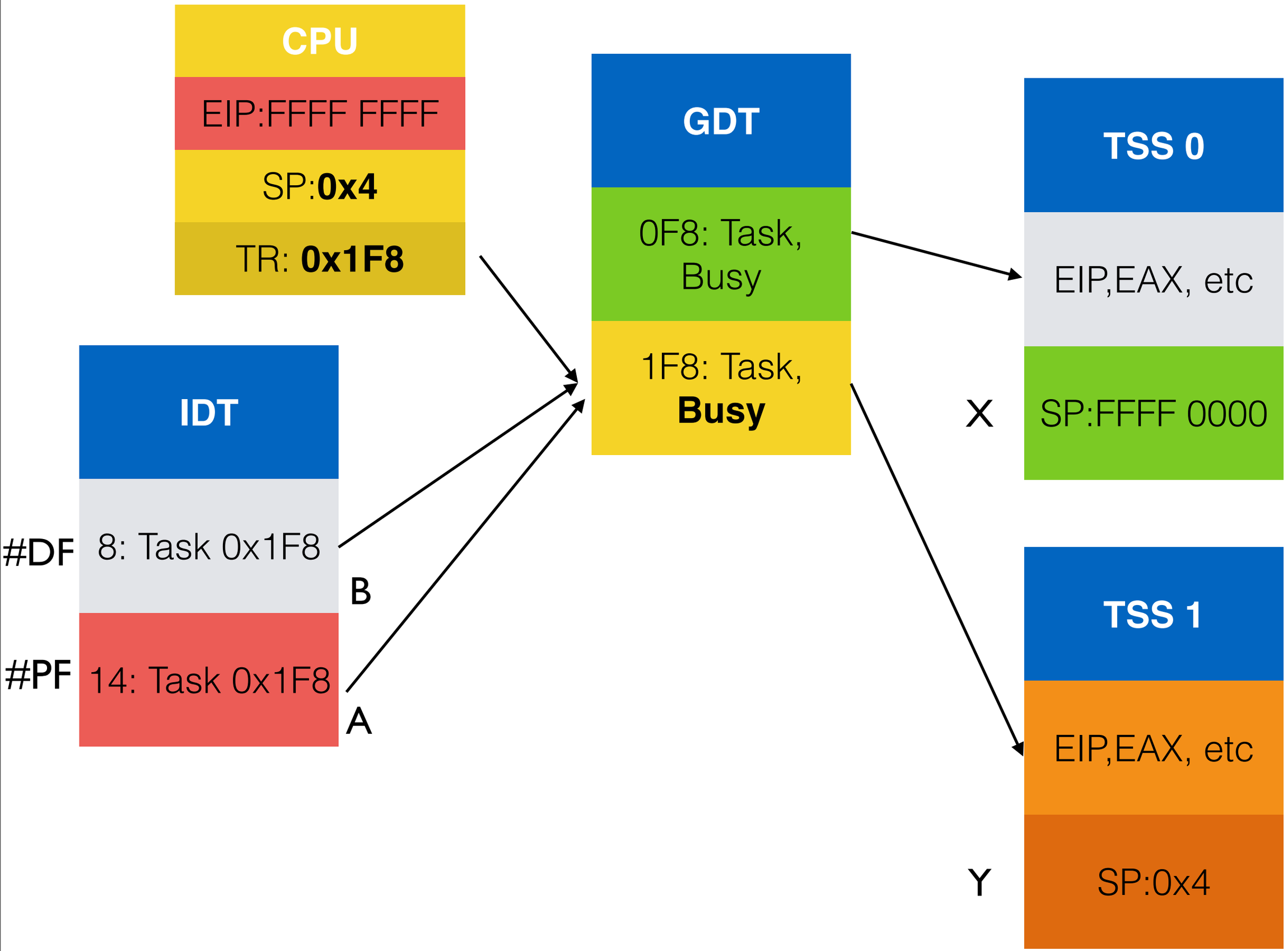

# Initial State



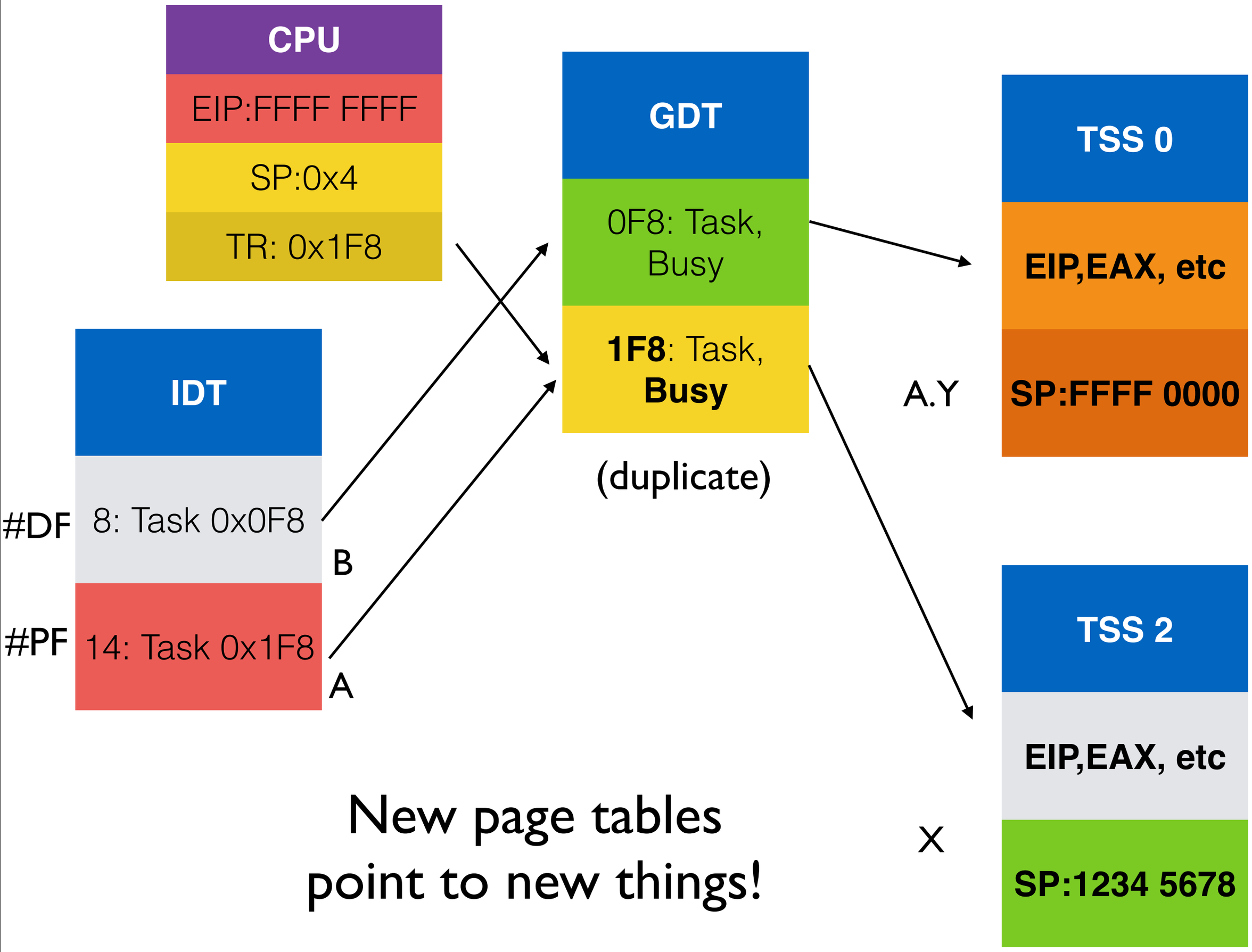

# EIP causes Pagefault




# CPU state is saved to current task



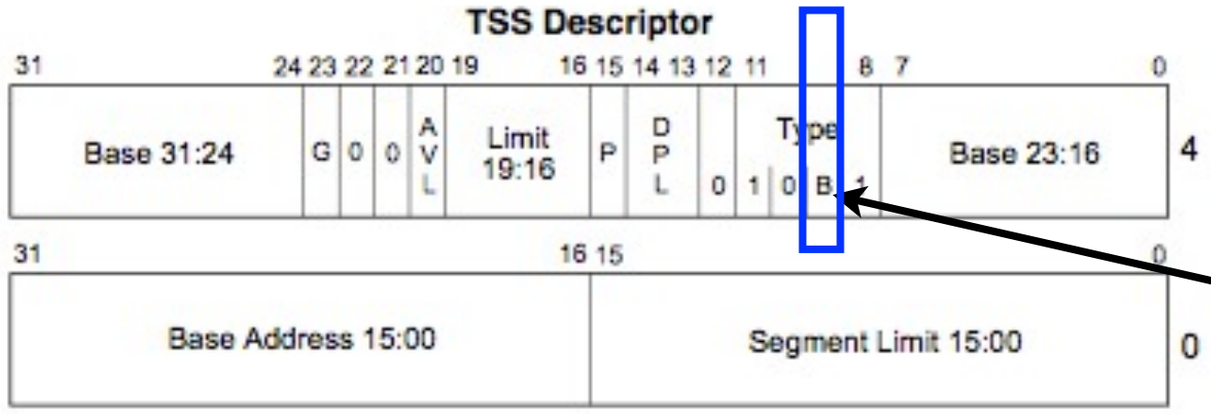

# CPU loads interrupt task




New page tables
point to new things!

# “Implementation Problem”

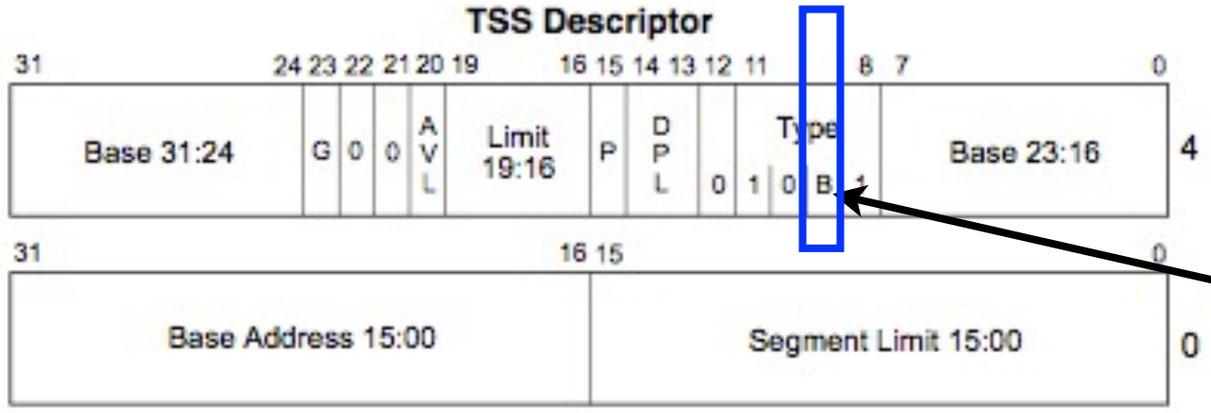
# That one pesky busy bit...




| | |
|---|---|
| AVL | Available for use by system software |
| B | Busy flag |
| BASE | Segment Base Address |
| DPL | Descriptor Privilege Level |
| G | Granularity |
| LIMIT | Segment Limit |
| P | Segment Present |
| TYPE | Segment Type |

CPU won't load task if this is set

# That one pesky busy bit...



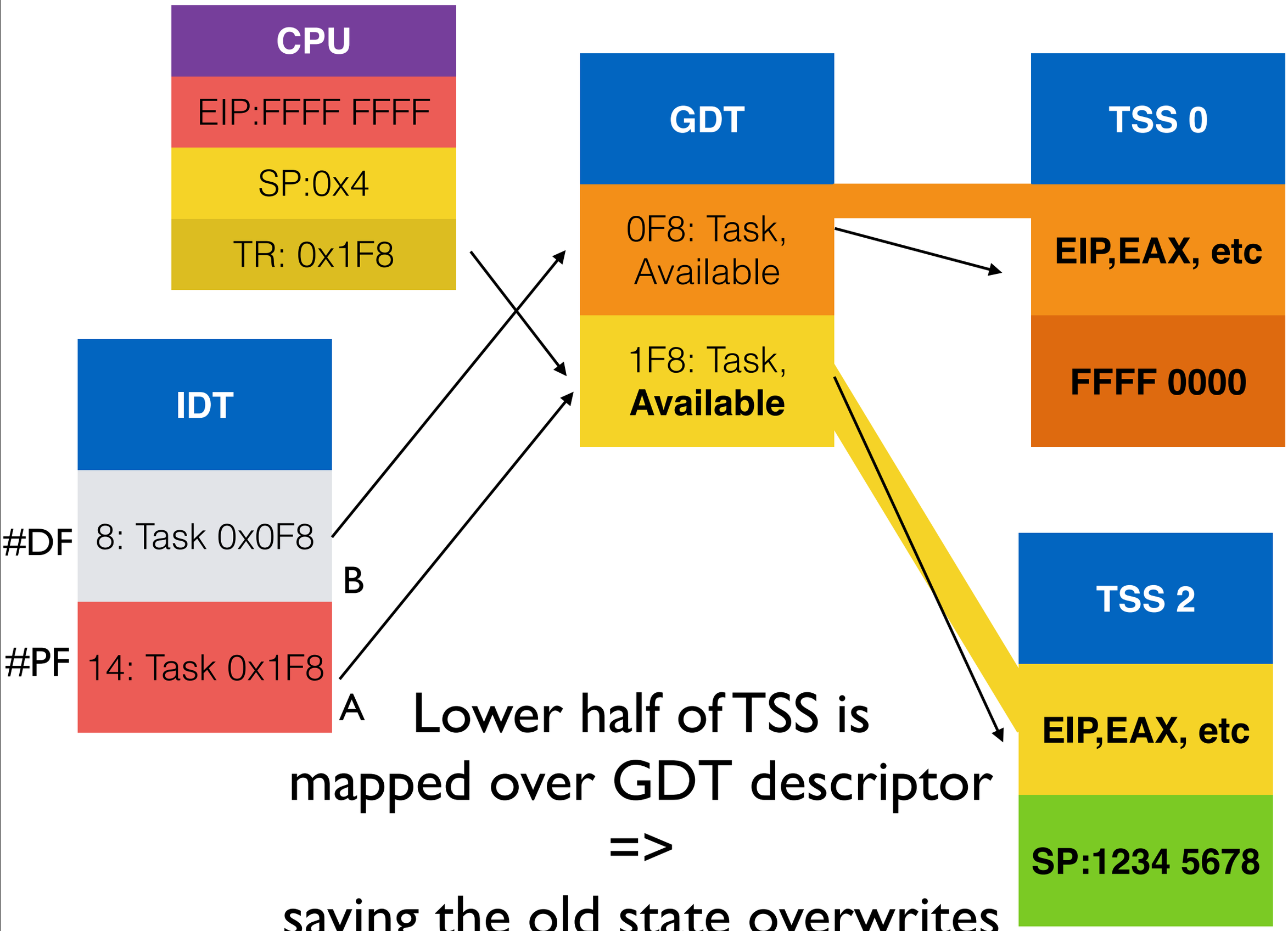

| | |
|---|---|
| AVL | Available for use by system software |
| B | Busy flag |
| BASE | Segment Base Address |
| DPL | Descriptor Privilege Level |
| G | Granularity |
| LIMIT | Segment Limit |
| P | Segment Present |
| TYPE | Segment Type |

CPU won't load task if this is set

We need to overwrite it. Luckily, the CPU always saves all the state (even if not dirty).

So: map the lower half of TSS over GDT, so that saved EAX,ECX from TSS overwrite descriptor; same content, only busy bit cleared.

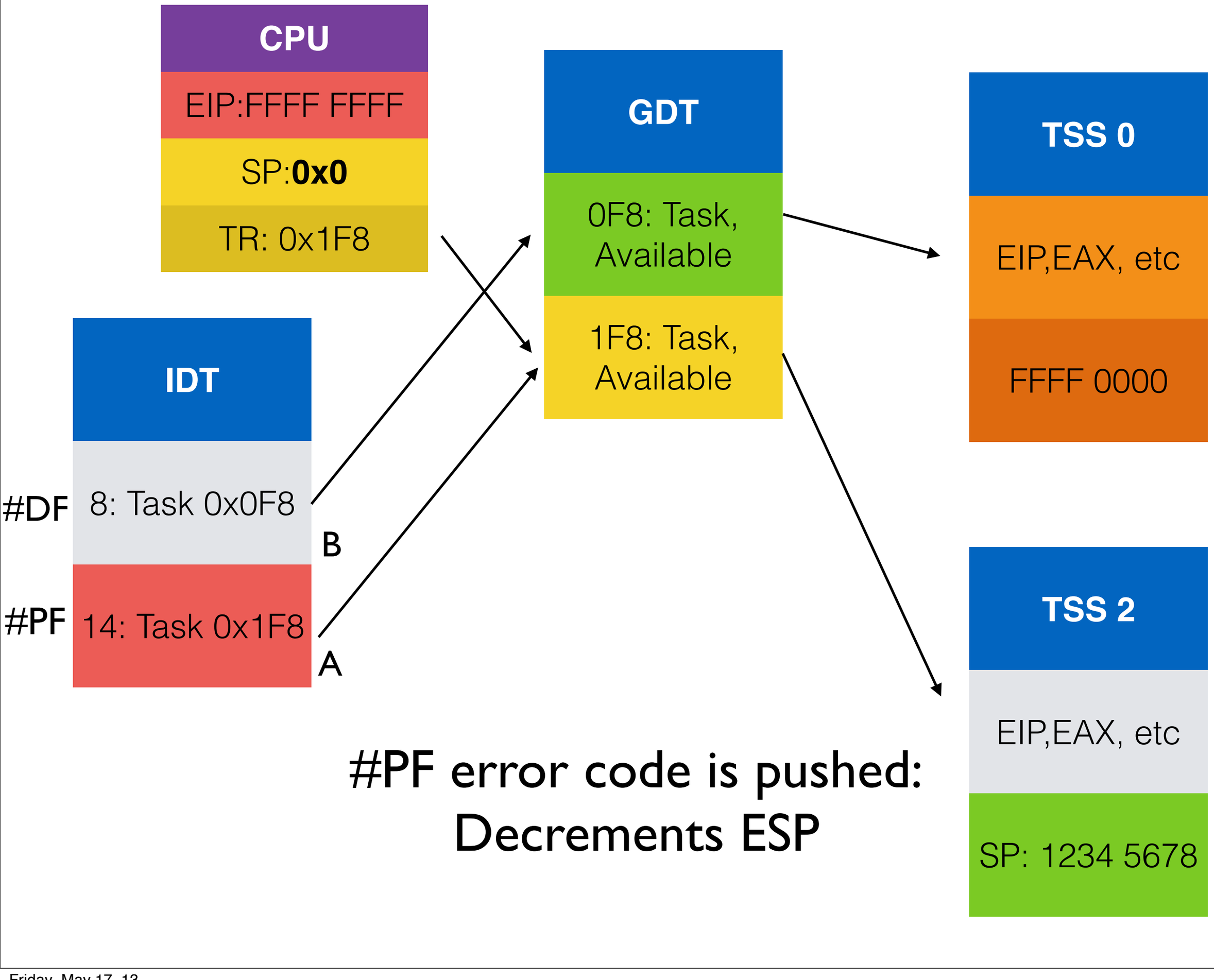
# Dealing with that bit needs a nuclear option...



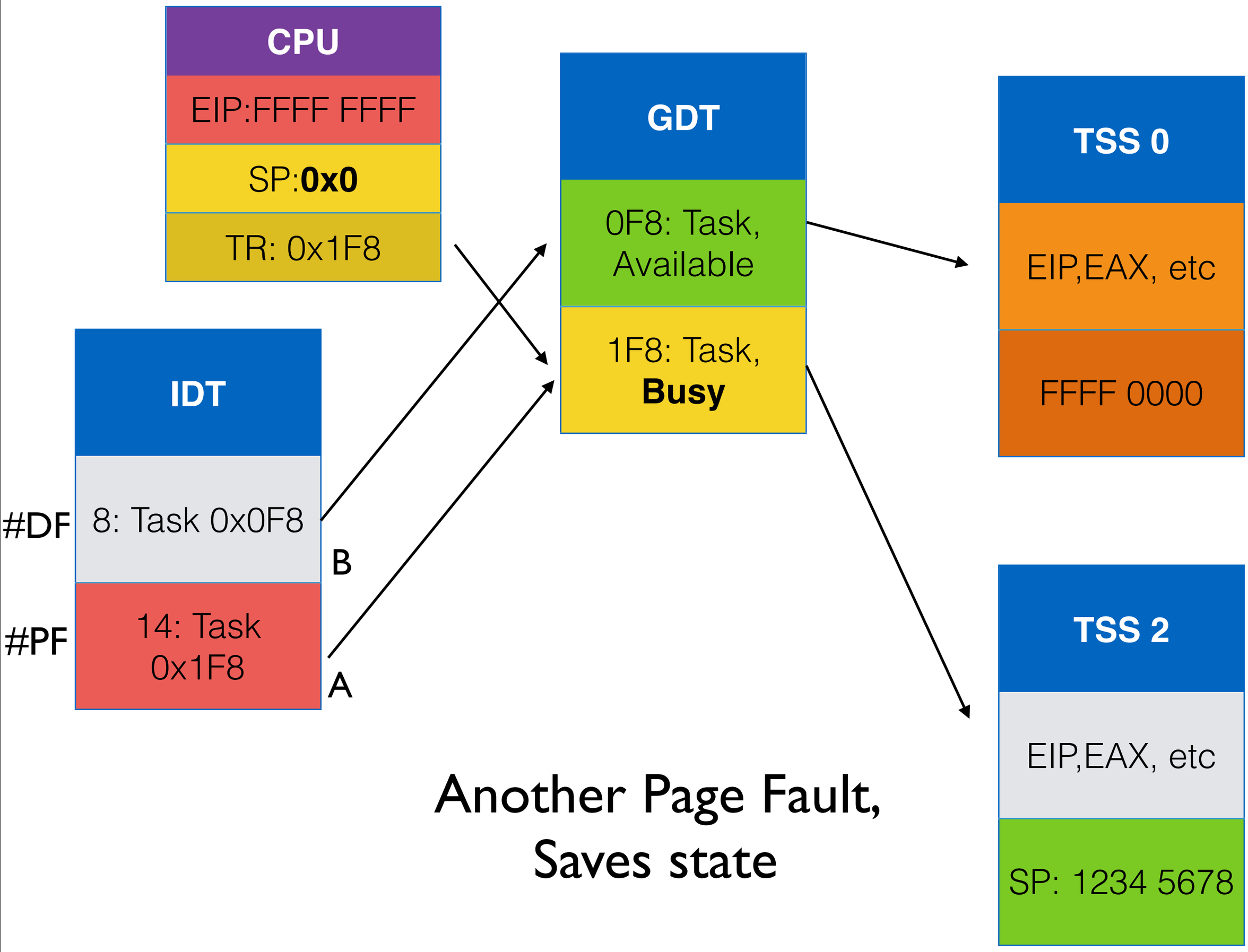

Lower half of TSS is mapped over GDT descriptor

=>

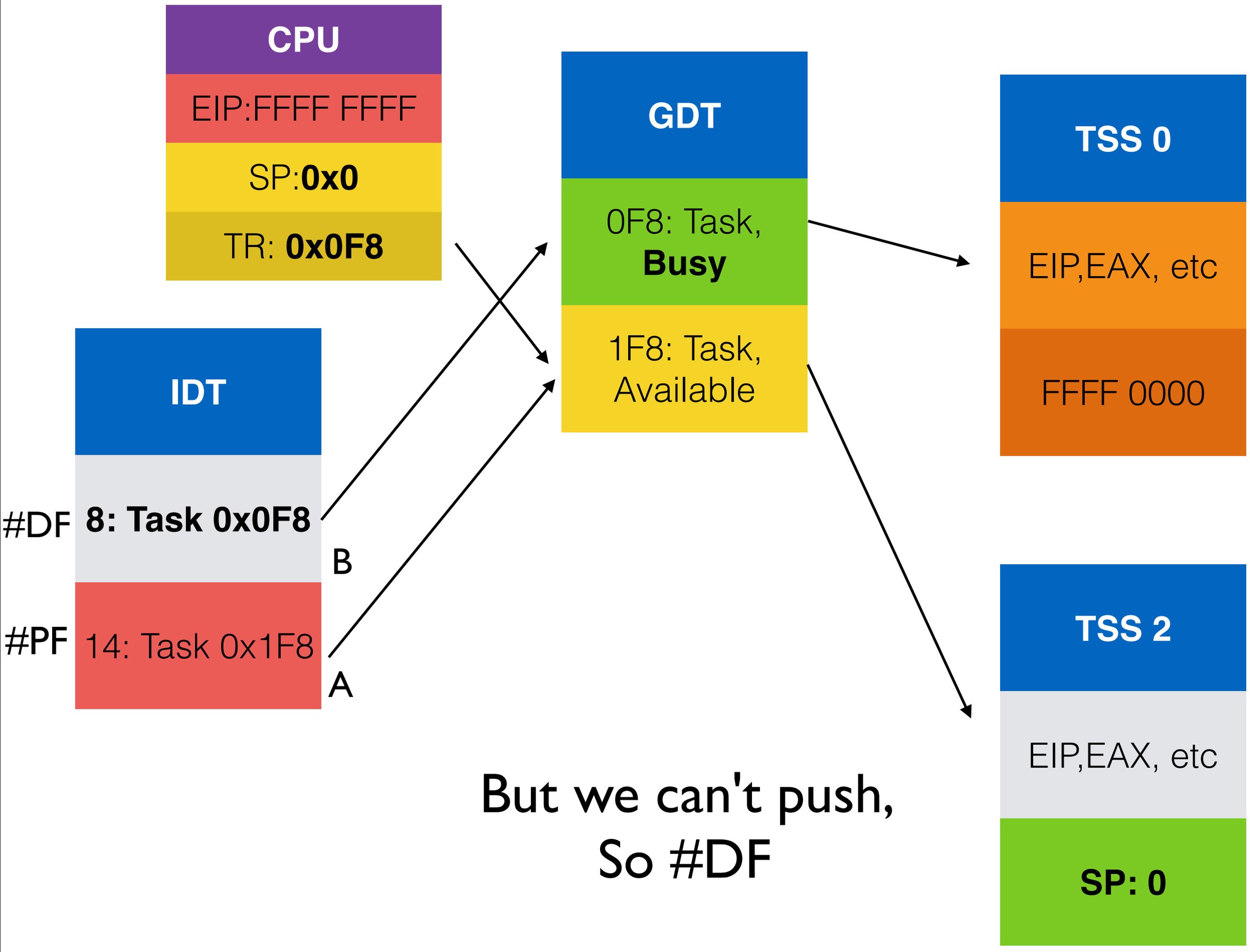
saving the old state overwrites the GDT entry busy bit!

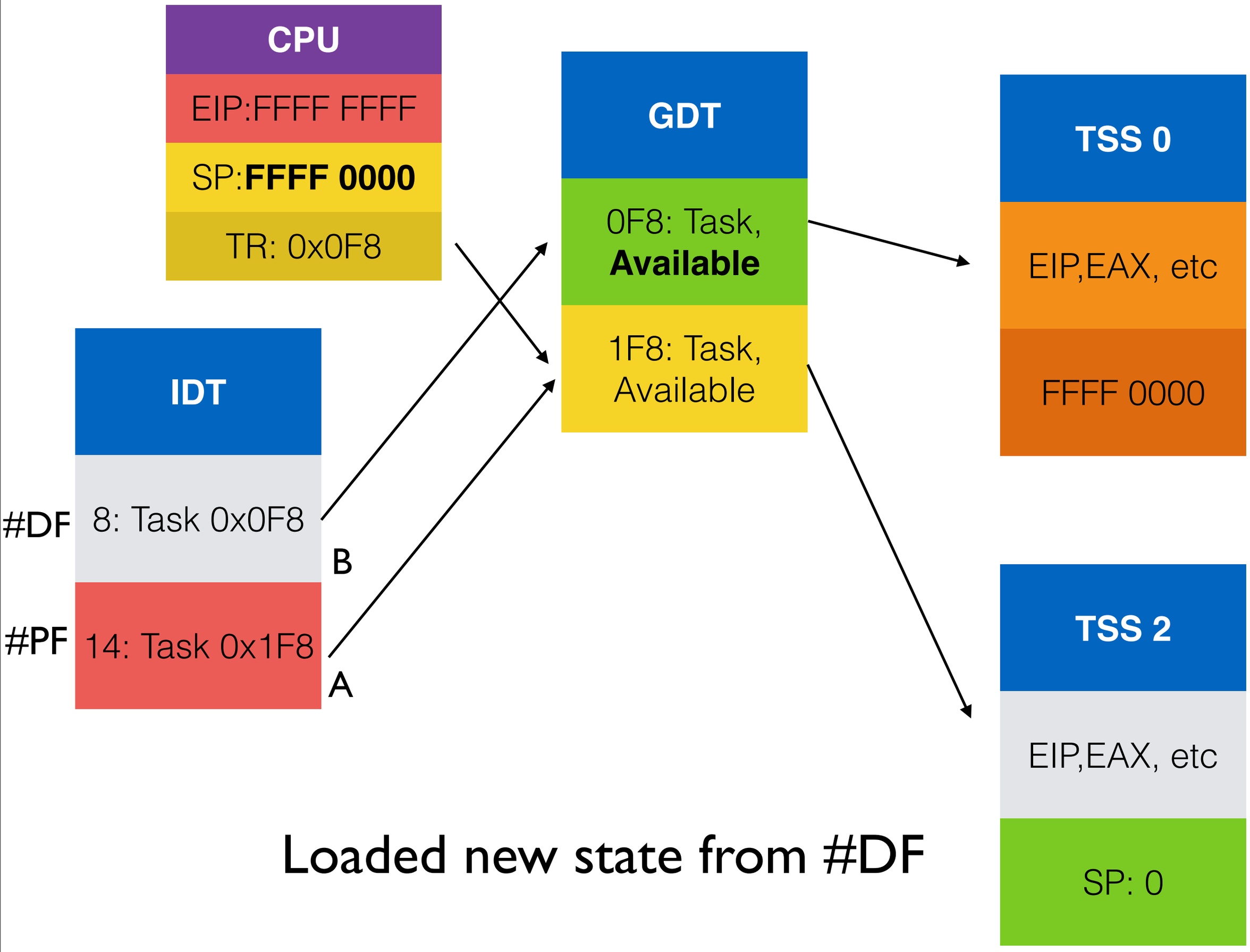



# #PF error code is pushed: Decrements ESP



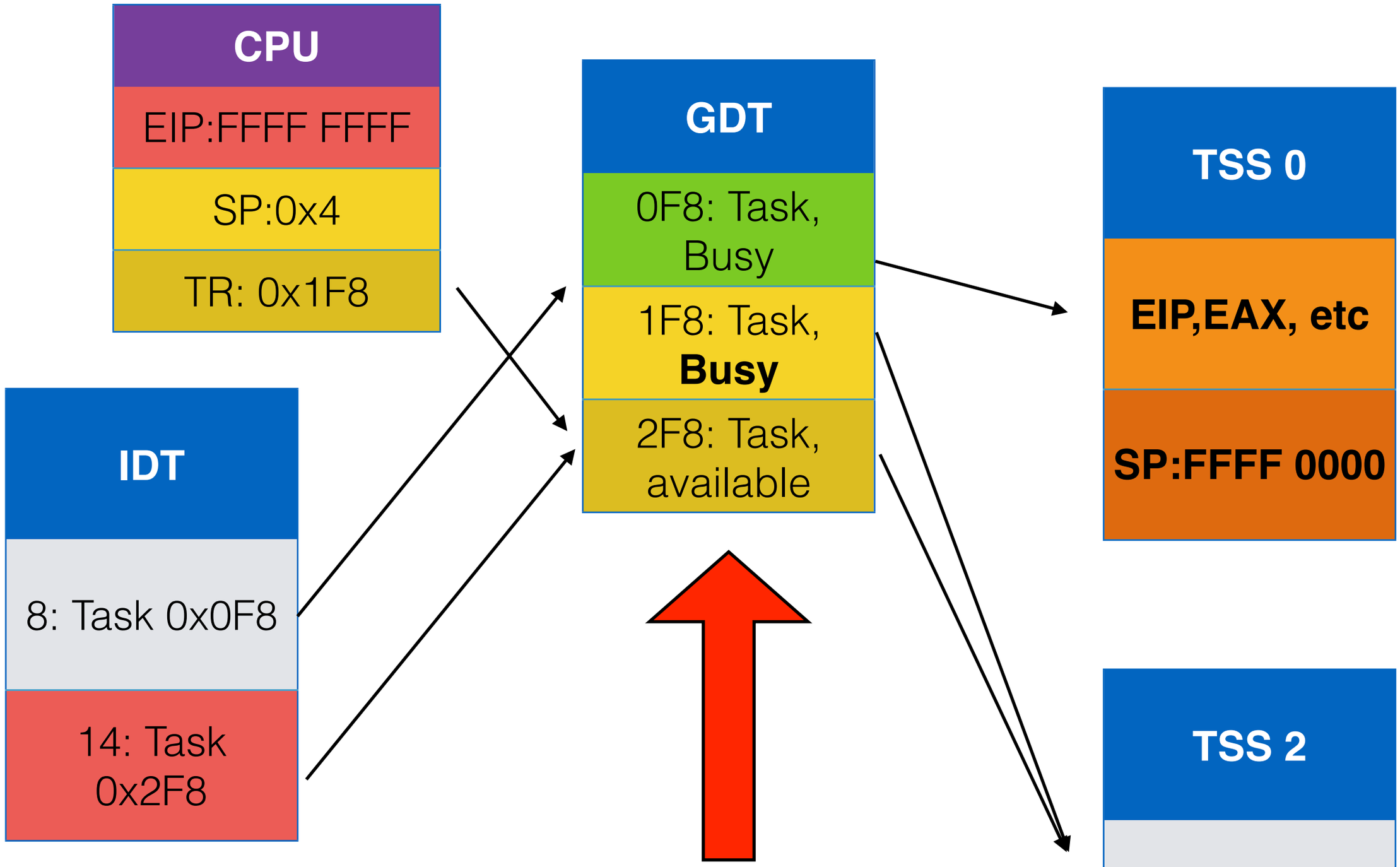

# Another Page Fault, Saves state


CPU
EIP:FFFF FFFF
SP:0x0
TR: 0x0F8
GDT
0F8: Task, Busy
1F8: Task, Available
TSS 0
EIP,EAX, etc
FFFF 0000
IDT
#DF
8: Task 0x0F8
B
#PF
14: Task 0x1F8
A
TSS 2
EIP,EAX, etc
SP: 0
But we can't push,
So #DF




Loaded new state from #DF

# And now to face the uglier truth...




IDT trick must take care of
task switch logic checking TR contents
=> must duplicate GDT descriptors

# Meanwhile, on the FSB

## (Slightly redacted)

| | |
|---|---|
| Write 0x8 | 0xFFFF 0000 |
| Read 0x1008 | 0x4 |
| Write 0x2008 | 0x0 |
| Read 0x8 | 0xFFFF 0000 |

## And they all compute happily ever after (for all we know)

# What restrictions do we have?

- Needs kernel access to set up :)
- No two double faults in a row
- Can only use our one awkward instruction
- Can only work with SP of TSS aligned across page (very limited coverage of phys. mem)


In Soviet Russia,
Red Pill takes you

# White Hat Takeaway

- Check how your tools handle old/unused CPU features
- Don’t trust the spec

# Black Hat Takeaway

- A really nice, big Red Pill
- With more work, you can probably make it work differently in Analysis tools
- Or just shoot down the host

# Strawhat Takeaway

- It's a weird machine! (And we like them)
- We are working on 64 bit, better tools
  - Compiler, debugger
- See how it works on different hardware?

@bxsays

https://github.com/bx/elf-bf-tools

@JulianBangert

https://github.com/jbangert/trapcc

"There is never enough time.
Thank you for yours!"

*--Dan Geer*