Enhance protection and memory management

- User process in ring3
- More enhancement

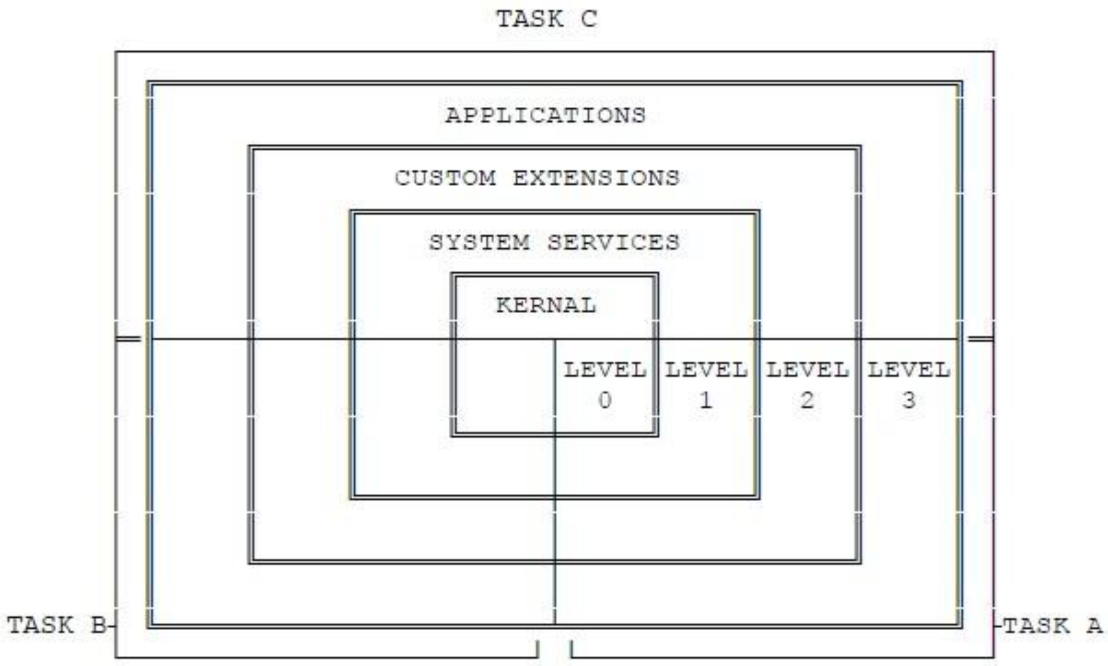
- User process in ring3

Put user processes in ring3

- When creating the first process
  - use the user code segment & user data segment
  - allocate one page for user stack
    - 0xbffff000 – 0xbfffffff
- But this will cause some problems.
  - We have to fix it.

Levels of privilege

- There are 4 types of the privilege level for a segment in 386 CPU.



A problem

- In real OS, user process has low privilege level.
  - code, data and stack are in ring3
- Interrupt handlers cannot run with user stack according to the hardware mechanism.
  - Why?
  - noncomforming stack: SS.DPL > CPL
- What if interrupt handlers are permitted to run with user stack?
  - Can you construct a counter-example to harm the system?

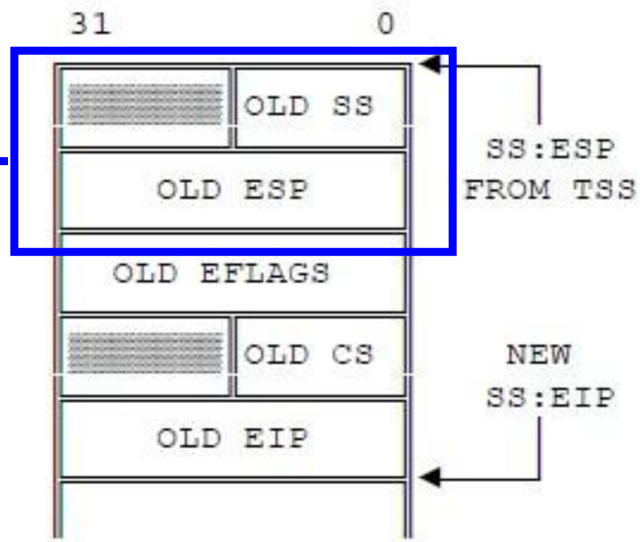
Solution

- Before handling the interrupt, perform stack switching.
  - use another stack instead of the user stack
  - stack switching is performed by hardware
  - kernel graruntees enouth space in the new stack to handle the interrupt

New member of current state

- The old stack pointer is stored in the new stack.
  - still can go back to user stack when returning from an interrupt

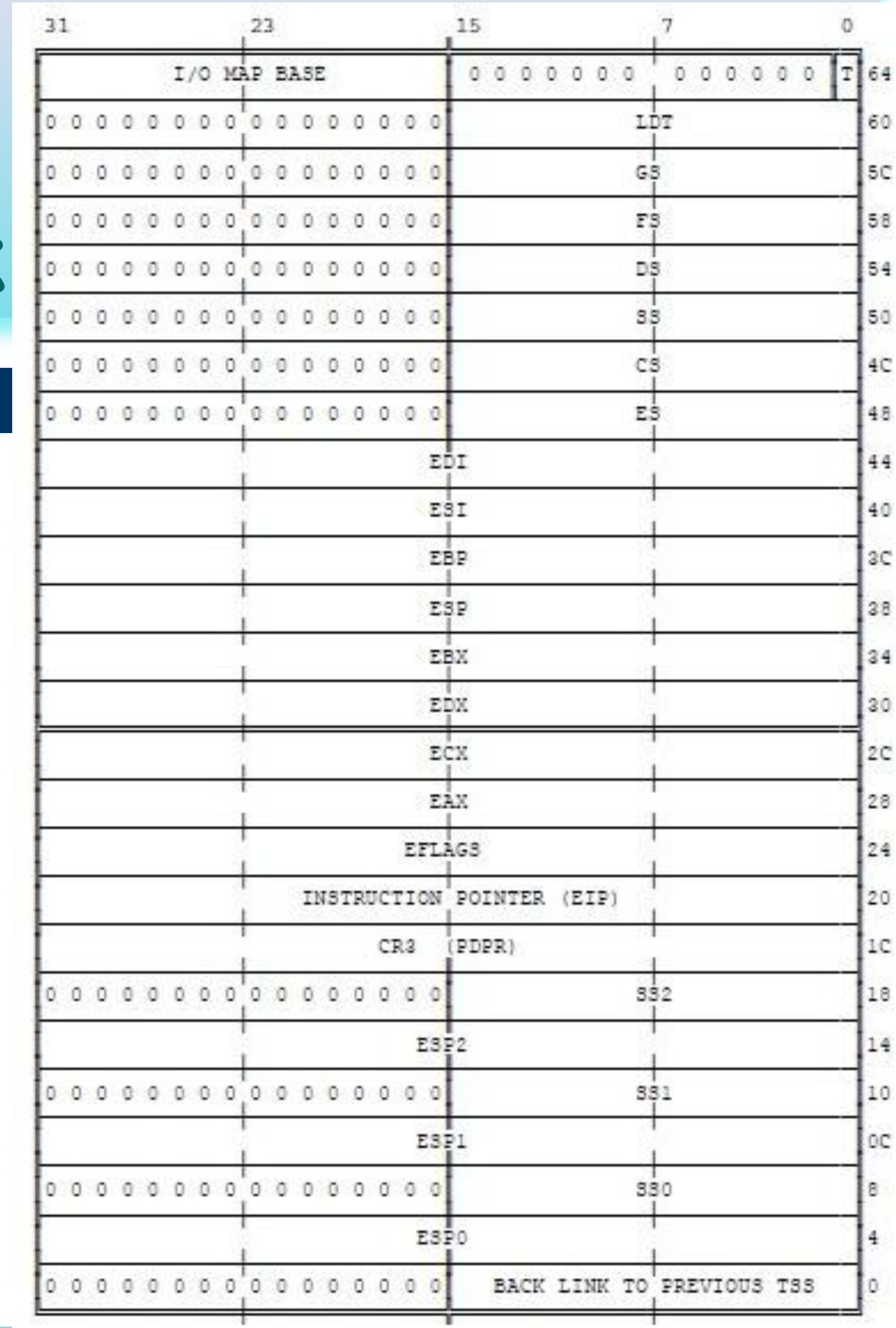
point to the user stack



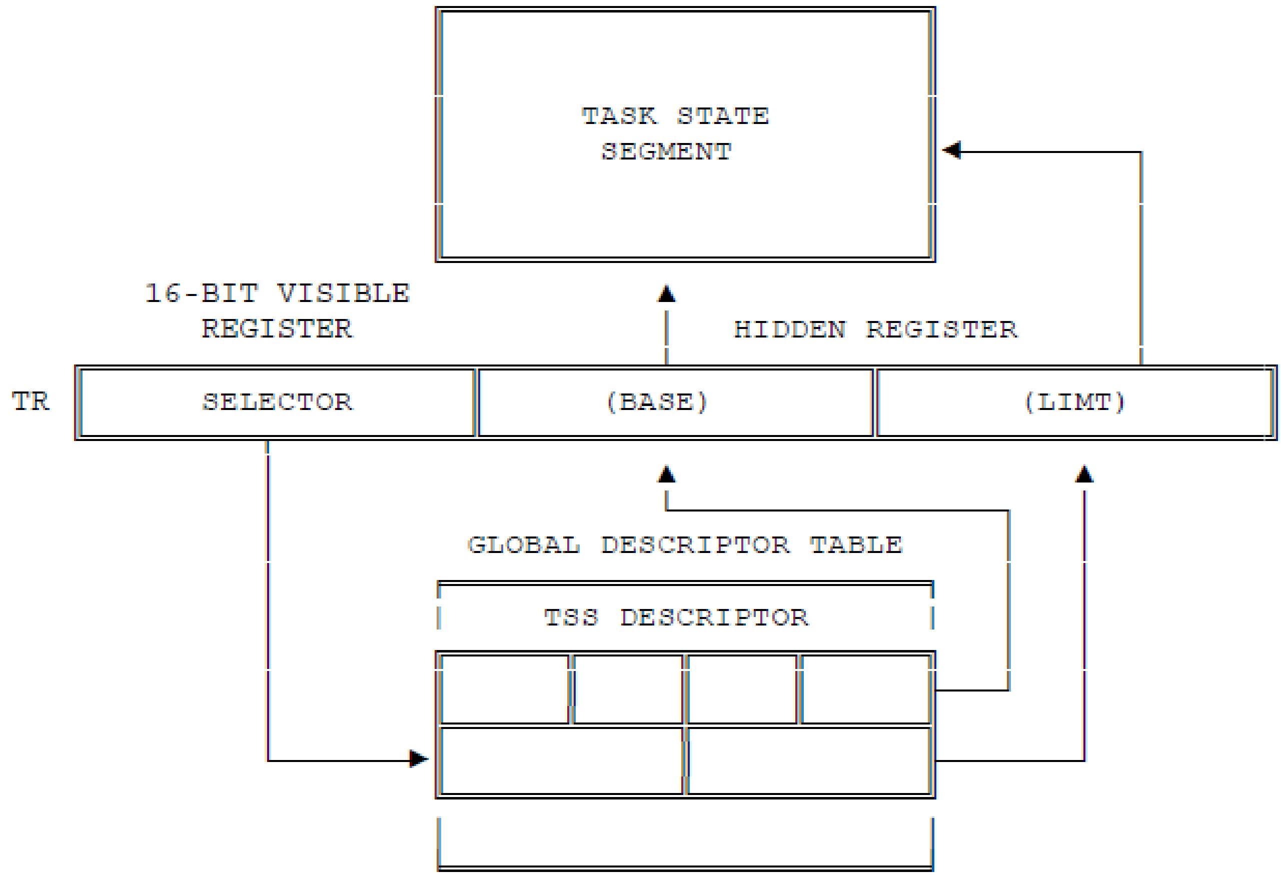
Finding new stack

- TSS
  - task state segment
- 3 stacks for 3 PL
  - ring0, ring1, ring2
  - Why no stack for ring3?

- We only use the stack for ring0.



Finding TSS


TASK STATE
SEGMENT
16-BIT VISIBLE
REGISTER
HIDDEN REGISTER
TR
SELECTOR
(BASE)
(LIMT)
GLOBAL DESCRIPTOR TABLE
TSS DESCRIPTOR

What should you do?

- set tf->ss & tf->esp to the user stack when creating the first user process
- set tss.esp0 to the new process' kernel stack during context switching
- That's all!
- Why does each user process own its kernel stack?
  - What if we set only one kernel stack for all user processes?

Everything else is done by hardware

- stack switching
  - finding TSS
  - loading kernel stack
  - store old stack pointer in the kernel stack
  - store current state
- returning from interrupt to user space
  - When executing iret, switch back to the user stack according to tf->ss & tf->eps.

- Other enhancement

Catch "Segmentation fault"

- When a user process tries to perform illegal operation, kill it, then print "Segmentation fault" to tty, instead of triggering system panic.

- First, enhance exit() and waitpid().
  - void exit(int exit_code);
  - int waitpid(pid_t pid);
    - return the exit code of the process "pid"

Catch “Segmentation fault” (cont.)

- One implementation is as following
  - Distinguish between illegal operations from user processes and bugs from kernel.
    - How?
  - After identifying illegal operations, issue an exit() system with a special exit code.
  - shell waits for child process to exit, it checks chlid process' exit code.
    - If it is the special exit code, print “Segmentation fault” on the screen.

Set lower privilege level for user pages

- restricting addressable domain
  - clear U/S bit in PDE & PTE for kernel space
  - set U/S bit in PDE & PTE for user space
- type checking
  - clear R/W bit in PDE & PTE for pages for user code
  - set R/W bit in PDE & PTE for pages for others
- For combining two-level protection, see i386 manual for details.

Copy-on-write

- Copy-on-write may dramaticly decrease the cost of fork().
- When handling fork(), maps child process' code and data to the father's pages, instead of cloning the whole address space.
  - User stack should still be cloned.
- Label these pages as read-only in the two parties' PTE.

Copy-on-write (cont.)

- Once any of the two parties writes to a labeled page, a page fault exception is triggered.
  - Distinguish this accessing from an illegal operation.
    - How to distinguish?
  - Clone this page for the child process, and set the two pages writable.
  - Pages for code are still read-only.

Copy-on-write (cont.)

- When a process exits, the shared pages should not be reclaimed right away.
  - Another process still needs to access these pages.
  - A page is reclaimed when no other process will access it.
- How to maintain the process-page relationship?

malloc and free

- heap management
  - allocate and free memory dynamically
- Where is heap?
  - next to the static data
  - grow up (to the direction of high address)
- mallco() and free() are not system calls.
  - OS will not manage user heaps directly.

malloc and free (cont.)

- void* sbrk(int inc);
  - system call to change the size of data segment
  - increase the size of data segment by “inc” bytes
  - when inc < 0, the size of data segment is decreased
  - return the top of data segment
- malloc(nr_byte)
  - first check whether there is a continuous space of size “nr_byte” in heap
    - if yes, return the start address of this memory space
    - if not, call sbrk() for more heap space, then perform allocation

malloc and free (cont.)

- call sbrk(0) when malloc() is called for the first time
  - obtain the top of static data / the start of heap
- When handling fork(), pages for user heap should be cloned, too.
  - can apply copy-on-write
- When handling exit(), pages for user heap should be reclaimed, too.
  - avoid memory leak