SPEDE Lab Manual: Chapter 13 – User Mode Tasks
<draft>

1. Privilege Levels and Segments

The "privilege level" of a piece of code running in an Intel architecture machine is determined by the least significant two bits in the Code Segment (CS) register. These bits are called the "Current Privilege Level (CPL)". A value of 00 is the *highest* (most privileged) level; a value of 11 is the *lowest* (least privileged) level. Values of 00, 01, and 10 are sometimes collectively called "System Level", while the lowest level, 11, is called "User Level". OS kernel code runs at CPL 00 (also called "Ring 0"); I/O drivers and other system routines usually run at CPL 01 or 10 ("Ring 1", "Ring2"); user tasks (processes) always run at CPL 11 ("Ring 3"), the least privileged level.

Memory is accessed as a collection of "segments", each of which is described by a "Segment Descriptor" found in the Global Descriptor Table (GDT). Every instruction that accesses memory does so using a "segment register" (either CS, DS, ES, or SS); the segment register is used as an index into the GDT to "select" a specific Segment Descriptor. Every Segment Descriptor contains, among other things, a set of "access privilege" bits (called the "Descriptor Privilege Level" or DPL). These bits specify the level of privilege the currently running code must have to access the corresponding segment. Specifically, the running code must have a CPL value which is *not numerically greater than* the DPL value of a segment in order to be allowed to access the segment.

The FLAMES startup code creates a set of Segment Descriptors in main memory – one for code (called the "Kernel Code Segment"), one for data (the "Kernel Data Segment") and one for the Stack (the Kernel Stack Segment). The Segment Descriptor DPL bits for these segments are set to 00, meaning that code must be running in Ring 0 to access them. FLAMES also arranges that a downloaded program starts running in Ring 0 by default, which allows the program to access all the default segments.

FLAMES also creates a number of additional Segment Descriptors, including "User Code", "User Data", and "User Stack" Segment Descriptors. These descriptors map the memory on the machine in the same way as the "Kernel" descriptors (which is to say, as one contiguous segment starting at zero; see the Address Translation paper), but they have access bits (DPL values) of 11. This makes them "User Segments" (although it still allows code running in Ring 0 to access them.)

The Segment Descriptors created by FLAMES are all stored in a single table, the Global Descriptor Table. FLAMES arranges that when a program starts running, the "Global Descriptor Table Register (GDTR)" points to the beginning of this table. The Segment Descriptors themselves are 8 bytes each and are stored at offsets from the beginning of the table as follows:

Segment	Offset from GDTR
Kernel Code	0x08
Kernel Data	0x10
Kernel Stack	0x18
User Code	0x30
User Data	0x38
User Stack	0x40

Figure 1. Flames Default Segment Descriptors

2. Task State Segments

In addition to "code segments", "data segments", and "stack segments", the Intel architecture defines a special kind of segment called a "Task State" segment. Just as "code" segments are used to hold code, "data" segments are used to hold data, etc., a "Task State Segment" (TSS) is used to hold the "state" of a "task". Task State Segments are structures which the OS builds in memory to describe the state (for example, the register contents) of a task. The layout of a TSS structure is shown below.

A TSS is used when a transition is made from one "task" (for example, the kernel) to another "task" (for example, a user process). The processor contains a special register called the *Task Register (TR)* which points to the Task State Segment of the "currently running task". The processor has a mechanism for automatically saving the state of the current task in a TSS and then loading the processor state from another TSS (this is described in more detail below).

The layout of a TSS is shown in the following diagram.

<insert diagram here: Figure 2. Task State Segment layout>

Every TSS must be at least 104 (0x68) bytes long (anything shorter than that generates an exception). Most of the TSS fields (GS, FS, DS, … CR3) hold data which are loaded into the processor's registers when a "task switch" to the task described by this TSS occurs, and correspondingly are used to save the processor's registers when a task switch "out of" this task occurs. For example, a TSS representing a privilege-mode task (such as the kernel) would contain CS, DS, and SS values pointing to the kernel code, data, and stack segment selectors (0x08, 0x10, and 0x18) respectively, while a TSS representing a non-privileged task would contain CS, DS, and SS values pointing to the User segment selectors.

Certain additional TSS fields (e.g. the "SSx:ESPx" and "Link" fields) hold information used as part of the task-switching process; these are described in more detail below. The bytes beyond byte 104 ("OS-specific Data Structures", "Interrupt Redirection Map", and "I/O Permission Map") are optional fields used for special purposes.

3. Task Segment Descriptors

In the same way in which code, data, and stack segments have "descriptors" in the GDT, Task State Segments have "descriptors" which specify various kinds of information about them. These Task State Segment Descriptors reside in the Global Descriptor Table, just like code/data/stack segment descriptors.

A TSS Descriptor contains the ***base address*** of the corresponding TSS, the ***limit (size)*** of the TSS, and various bits/fields giving access permission information and other related data. In the same way that code, data, and stack segment descriptors control what happens when a memory segment is accessed, TSS Descriptors control what happens when a Task State Segment is accessed (which happens when a "task switch" is invoked; in particular, the *Task Register* is loaded with the base address of the TSS for the "current task").

The layout of a Task State Descriptor is shown in the following diagram. Note that Flames does not create any default TSS Descriptors (nor any Task State Segments for that matter); an OS wishing to use hardware task switching must create its own TSS Descriptors in the GDT and corresponding Task State Segments in memory.

<insert diagram here: Figure 3. Task State Segment Descriptor layout>

4. Task Gates

In a simple OS not using hardware task switching, when an interrupt occurs while a process is running it causes a transfer through an "interrupt gate" in the Interrupt Descriptor Table to an Interrupt Service Routine whose CS:EIP value (i.e. the new "program counter") is provided by the gate. However, the interrupt gate does not provide for any switch in privilege level, nor does it provide for a new "stack"; any stack operations happen by default on the current process's stack (unless/until the interrupt service code changes stacks) and all operations happen at the privilege level of the process which was interrupted. [1]

In order to support hardware task switching (including changes between rings) each "interrupt gate" must be replaced with a *task gate*. A Task Gate is a structure which contains a reference to a TSS Descriptor (which in turn points to a Task State Segment, as described above). The layout of a Task Gate is shown in the following diagram. P is the "Segment

[1] In some architectures there is a mechanism for explicitly forcing a change of privilege level – for example, by setting a bit in the Flags register – and such a change in privilege level automatically changes to a different stack; this is not true in the Intel architecture.

Present" bit, which should be '1'. DPL is the "Descriptor Privilege Level" – the privilege level required to use the task gate. DPL should be set to '11' for Kernel Service interrupts (software INT instructions) so that user-mode tasks can invoke kernel services.

ByteValue	Offset
0 0 0 0 0 0 0 0	7
0 0 0 0 0 0 0 0	6
P DPL S 0 1 0 1	5
0 0 0 0 0 0 0 0	4
MSB of TSS Selector (offset in GDT)	3
LSB of TSS Selector (offset in GDT)	2
0 0 0 0 0 0 0 0	1
0 0 0 0 0 0 0 0	0

Figure 4. Task Gate Format

When an interrupt or exception occurs and the corresponding IDT vector contains a Task Gate, the processor saves the current context (that is, all the processor registers) in the TSS pointed to by the Task Register (the "current task's TSS"), then loads a new processor context (set of register values) from the TSS pointed to by the Task Gate.

An interrupt which transfers through a Task Gate also automatically stores the 16-bit selector (that is, the index into the GDT) for the *interrupted* task's TSS Descriptor in the "Link" field of the new TSS. Since all interrupt task gates should reference the kernel's Task State Segment (that is, switch into the kernel), this allows the kernel to save the TSS Descriptor selector for the interrupted task in the task (process's) PCB so that when the task (process) is eventually dispatched again the kernel knows where to find the task's TSS Descriptor.

5. User-mode Dispatching via Task Switching

Switching between privileged (*ring 0* or *system*) mode and non-privileged (*ring 3* or *user*) mode requires modifying the kernel's dispatch mechanism to use *task switching.* To do this, the OS must create a Task State Segment and corresponding TSS Descriptor for each user-mode process, plus one for the kernel (the kernel is considered a "task"; see below). Then, instead of pushing the state of the user-mode process on the user's stack manually (in the form of a "trap frame"), the kernel causes the CPU to execute a *hardware task switch* to load the state of the new task (process). The mechanism for doing this is based on how the Intel IRET instruction works.

When an IRET instruction is executed at the end of the Dispatcher, the processor pops CS:EIP and EFlags off the current stack. It then checks the value of the NT [Nested Task] bit in the EFlags register popped from the stack. If the EFlags NT bit is '0', the processor continues as before (i.e. it transfers to the code pointed to by the popped CS:EIP values). However, if the EFlags NT bit is '1', the processor interprets this as a "return from a nested task back to a previous task" (actually it's an "exit from the kernel to the process being dispatched", but the processor views things from a "task" point of view).

When an IRET detects an NT bit of '1' in EFlags, instead of transferring to the process code pointed to by the popped CS:EIP values it performs a *hardware task switch*. This is accomplished as follows. First, the processor saves its state into the *current task's* TSS (pointed to by the Task Register). Next, it fetches the value from the "Link" field of the current TSS and loads it into the Task Register. This value must point to a TSS Descriptor which references the new task's TSS (that is, the TSS of the task being dispatched). Finally, the processor loads a new state (register values) from the fields in the new TSS. The CS:EIP values in the new TSS determine where the new process starts executing; the CPL bits in the Code Segment pointed to by the CS register from the new TSS determine the privilege level of the new process.[2]

Note that this scenario requires that the kernel be thought of as a "task", with its own "Task State Segment". When the Dispatcher executes the IRET which performs the hardware task switch, it switches tasks by saving the current state into the *current* (kernel) TSS and loading the *new* (process) state from *the TSS pointed to by the old (kernel) TSS Link field*. This means that the Dispatcher must arrange to load the kernel TSS Link field with the user process TSS selector prior to each user process dispatch.

Note also that the hardware "validates" all fields involved in a task switch before actually using them, and generates an exception if it finds an invalid field. This means the structure of each TSS, TSS Descriptor, and Task Gate must be validly filled. In particular, note that the "LDT Selector" field in a TSS must contain a selector value (offset into the GDT) which points to a valid "Local Descriptor Table" Descriptor. This is true even if the code is not using the Local Descriptor Table. If the LDT entry in the GDT is not valid (or the LDT Selector in the TSS is not valid) then an "Invalid TSS" exception is generated.

Dispatching a process in the manner described in this section amounts to tricking the hardware into thinking it is "returning from the 'kernel task' to an 'interrupted task'". The hardware has a mechanism for keeping track of which tasks have been started but not finished (that is, which are "busy"). Each TSS Descriptor in the GDT contains a "Busy" bit (B). If this bit is not in the proper state when a task switch occurs then a General Protection (GP) exception is raised.

Specifically, it must be the case that when the kernel dispatches a task using an IRET, *the "busy" bit in the TSS for the new task must already be set to '1' when the task switch occurs* (otherwise, it doesn't look to the hardware like this is really a "return to an interrupted task"). The kernel must insure that a task's Busy bit is set to '1' when it first dispatches the task. After the first dispatch nothing further need be done; the task's Busy bit will remain set until the task is

[2] This means that the CS, DS, ES, and SS values loaded from the new TSS should point to *User* Segment Selectors if the process being dispatched is to run in user mode.

terminated (this happens automatically because the hardware sets the Busy bit to '1' whenever a task switch causes a task to be entered – but it won't be set for the *first* dispatch unless the kernel code makes it so).

Similarly, it must be the case that *when an interrupt causes a task switch from a process back to the kernel task, the kernel task must NOT have its Busy bit currently set to '1'* (otherwise, it looks to the hardware like it is trying to enter the kernel "task" when that task is already busy). However, this is handled automatically: when the kernel transfers to a process via an IRET task switch the hardware interprets this as having "completed the 'kernel task'" and it clears the Busy bit in the *kernel's* TSS Descriptor – so it will be zero when an interrupt transfers back to the kernel.

6. Kernel Re-entry Under Task Switching

When an interrupt transfers through a Task Gate, it saves the processor state in the current (interrupted) task's TSS (pointed to by the current value in the Task Register) and loads a new processor state from the kernel's TSS (pointed to by the TSS selected by Task State Descriptor pointed to by the Task Gate). It also stores the index (selector) of the interrupted task's TSS Descriptor into the Link field of the new (kernel) TSS, and points the Task Register to the new (kernel) TSS. In addition, it makes one modification to the new state loaded from the kernel TSS: it sets the "NT" (Nested Task) bit in the EFlags register to '1'. This flag indicates that the kernel was entered due to leaving a (user) task (that is, the kernel is now running – in a sense – as a "nested task" of the user task).

This means that the basic kernel entry mechanism is the same as before: each entry point must push a "reason code" and then transfer to a common routine which saves the state of the running process (task). Now, however, *most of the process state has already been saved in the process's TSS by the hardware task switch*; all that remains is to save a pointer to the TSS in the process's PCB. Of course, since the layout of the TSS structure is different from that of a "trap frame" there will have to be some adjustments made to code which accesses the process state. (An alternative would be to continue to copy the process state into a "trap frame" even though it's somewhat redundant, and continue to access process state via the trap frame.)

7. Ring Changes

As noted earlier, Task State Segments contain extra fields labeled SSx and ESPx, where x is a number between 0 and 2. These fields are used to handle the need for *stack switches during ring changes.* If an interrupt transfers through a Task Gate and the task switch results in a *ring change* (for example, when a user-mode process running in ring 3 gets interrupted and transfers to the kernel running in ring 0), the processor automatically does two extra steps, described as follows.

First, it *loads SS:ESP from the corresponding ring-change entry in the new TSS* (for example, if this is a transfer to ring 0 then it loads SS:ESP from the TSS "SS0" and "ESP0"

entries). This causes the processor to switch to the kernel's stack (assuming the TSS was properly initialized to contain the kernel SS:ESP in the SS0:ESP0 entries).

Second, it *pushes the current (i.e. ring 3) SS:ESP values onto the new (e.g. ring 0) stack*. In this way the processor can switch automatically between different stacks when switching between "user" and "system" mode while at the same time letting the new task (the kernel) know where the old task (the user process) was operating its stack.

Note that this means that in order for the kernel to start a process (task) in "user" mode, it must first be sure to arrange that there is a valid kernel Task State Segment *set up with the correct values for the kernel mode SS:ESP values*. That is, the TSS for the *user* process must contain the values for the *kernel's* Stack Segment and Stack Pointer register in the SS0:ESP0 entries so that when the process switches to kernel mode the proper stack will be used.

Note also that this means that when the kernel is entered via a task switch from a user-mode process, *there will be additional values pushed on the kernel's stack* – specifically, the "ring-3 SS:ESP" values; kernel code must allow for the presence of these new values on the stack.

8. Summary: Starting a User-Mode Task

The following are the primary steps for starting a user-mode task:

- Create a TSS for each task somewhere in memory. Point the TSS CS/DS/ES/SS registers to the (Flames-created) User Segment Descriptors in the GDT (this is what switches to "ring 3" when the task starts executing). ES should be set to point to the user-mode Data Segment, the same as DS. These TSS areas can be allocated and freed as the user tasks come into existence and eventually terminate.

- Create a TSS Descriptor, pointing to the task's TSS, in the Global Descriptor Table. Each task's TSS Descriptor must reside in the GDT, somewhere above the Flames-created default segment descriptors. The TSS Descriptor must define the base address, limit (size) and Descriptor Privilege Level (the level of privilege required to access the TSS descriptor). The DPL for Task Descriptors should be 00, meaning only the kernel (privileged code) can access them (otherwise, user processes could invoke task switches). Like each TSS, each TSS Descriptor is only needed as long as the user task exists.

- Create a TSS for the kernel somewhere in memory. This TSS defines the state of the processor when the kernel is running – both when it starts and when it is re-entered due to a task switch out of a process. The kernel TSS must set the CS/DS/ES/SS registers to the Kernel Segments created by Flames. (ES should be set to point to the same segment as DS).

- Load Task Gates into the IDT entries for all interrupts which might happen while the user task is executing. These Task Gates must point to a TSS Descriptor in the GDT; that TSS Descriptor in turn must point to the TSS for the kernel. The Task Gates for software

interrupts (INT instructions) should contain DPL values of '11' so that user-mode processes can invoke them (Task Gates cannot be invoked by code running at a lesser privilege level than that specified in the Task Gate). No privilege check is performed on Task Gates for hardware interrupts or exceptions. All Task Gates must have their 'P' bit set to 1 ("Present") and have valid base and limit fields.

- Use the LTR (Load Task Register) instruction to point the Task Register at the TSS for the kernel. The argument to the LTR instruction is the offset in the GDT to the TSS Descriptor which points to the kernel TSS (not the address of the kernel TSS itself). Note that *loading the TR register does <u>not</u> cause a task switch;* it tells the processor what the "current task" looks like. Performing a task switch (e.g. by executing an IRET to a new task) automatically saves the current TR value in the new TSS before actually transferring to the new task; this is how the processor knows what task to "restore" when a switch back out of the new task occurs (such as when an interrupt through a Task Gate occurs).

- Start a user task by pushing a CS:EIP and EFlags value onto the stack with the "NT" bit in EFlags *set to '1'*. Executing an IRET instruction will cause the processor to suspend the current task (the kernel) and switch to the new task, loading the processor state from the new task's TSS.