Review : Segmentation, Interrupt Descriptor Table, Kernel stacks

If there is only one CPU, only one process could be running on the CPU at any time. The kernel time-slices the CPU execution time, so that multiple processes appear to run concurrently. As we saw last time, while a process is running the kernel is *not* controlling the CPU at that time. The kernel regains control of the CPU if one of the following occurs, for example:

- The process makes a system call
- The process causes an exception
- An external interrupt occurs

The Interrupt Descriptor table specifies what code to execute, and at what privilege level, on any of these events, based on the corresponding interrupt vector number.

If there are multiple CPUs, each CPU independently executes in this fashion. For example, one CPU could be executing a process, while another CPU could be executing in the kernel mode. Notice that by definition, one process (or one thread of a process, if a process could have multiple threads) may only be executing on at most one CPU at any time.

On a switch from process execution to kernel execution (from unprivileged to privileged mode), the CPU also switches the stacks. The stack for the kernel mode is obtained using the Task State Segment (TSS). If the kernel uses a *process model*, each process will have a separate kernel stack. Hence, the kernel must load the kernel stack of the process into TSS, before transferring execution control to that process.

If the kernel uses an *interrupt model*, each CPU has only one stack, which remains constant throughout the execution of the system. In this case, every switch from the process to the kernel starts on a *clean* stack, i.e., on an entry it will only contain the values pushed by this switch (e.g., saved SS, saved ESP, saved EFLAGS, saved CS, saved EIP). A kernel may want to interrupt its own execution, while it was running on behalf of a process. For example, this can happen if the process made a system call to read contents from a disk file. Reading the file contents may take a long time, and so the kernel may want to *context switch* to another process while the disk is responding. In this case, the kernel must save the contents of the current stack (somewhere in its data structures where it maintains per-process information), before loading the new process. In future, if the kernel wants to switch back to the old process (e.g., if the disk device has supplied the requested contents of the file), the kernel must initialize the stack using the saved contents, before transferring control to the old process.

In the process model, the kernel is always executing in the context of the stacks of one of the processes. If the kernel wants to interrupt its execution, it simply saves the current stack pointer, and loads the stack pointer of the new process. On a return to the old process, the old stack pointer is restored.

The kernel needs to store information per process. For example, it needs to store the list of all processes, their state (e.g., are they ready to run, are they waiting for an event to complete, are they already running, etc.), values that define their address space (e.g., segment selectors), and values that define their kernel stacks (stack pointers in process model, or contents of the stack in the interrupt model).

This information is typically stored as a list of ``Process Control Blocks'' (PCBs). Each PCB contains process state:

- pid
- state : ready, running, blocked, etc.
- address space (e.g., segment selectors)
- kernel stack
- saved registers
- other information

This list is allocated in the kernel address space (inaccessible to the other processes). The "handlers" consult these data structures to decide what action to take on any interrupt/exception.

For the most part, the kernel starts execution in one of the handlers. For example, the kernel implements a system call handler that internally calls the functions that implement the functionality of that call. One of the system calls is `yield()`, which tells the kernel that the process is interested in relinquishing the CPU to another process (if any exists). This may happen if a process is waiting for an event, and knows that it is going to take a long time. In this case, instead of occupying the CPU, a process may make the "good gesture" of calling `yield()`.

If a process does not call `yield()`, the kernel has other mechanisms to take control away from the process. This is called preemption of a process. Preemption is necessary to ensure that the process cannot keep executing forever, and the OS can acquire control back. A typical way to do this is to use a hardware timer device. The OS configures the timer device to generate an interrupt periodically. It then sets up the interrupt handler for the timer interrupt, before tranferring control to the process. On a timer interrupt the kernel's handler is invoked, and it is free to let the process continue execution (by simply returning from the interrupt), or to *context-switch* to another process (which involves saving the current state of the execution of the current process and loading the state of execution of the new process, before returning from the interrupt).

Typically, the kernel will have 4-5 descriptor entries in the GDT. One of those entries will define the kernel address space (using `kbase` and `klimit`), and another one will define the user (or process) address space (using `ubase` and `ulimit`). On a context switch, the kernel will overwrite the descriptor entry containing `ubase` and `ulimit` with the base and limit of the new process, before transferring control to it.

FAQ

- **GDT can have up to 2^13 entries. Why can't we use one entry per process? Why do we need to overwrite GDT entries on every context switch?**
  Because when one process (say P1) is running, the address space of another process (say P2) should not be accessible through the GDT. So it makes sense to overwrite P2's GDT descriptor (its base and limit) with P1's GDT descriptor before context switching to P1.
- **Why does the GDT have 2^13 entries? Isn't this very wasteful, if we are going to use only 4-5 entries?**
  Actually the hardware allows you to specify the size of the GDT (which can be up to 2^13 entries). The GDTR register actually points to a memory area of six bytes that specifies the GDT base and the GDT size. (see the semantics of the LGDT instruction in Intel Reference Manual volume 2 (page 427) for full details.
- **Can a process customize interrupt/exception handlers?**
  The UNIX abstractions do not allow this. But one could potentially design an OS where this is allowed. Customization in UNIX is done through *signal handlers*, but the mapping of exceptions to signals is fixed inside UNIX. Also signal handlers execute in unprivileged mode.
- **Why have two abstractions: signal handlers (for process) and trap handlers (for kernel)?**
  Traps are a hardware abstraction. Signals are an operating system abstraction. It is possible for an OS to not have signals (this is an OS design decision), and yet support traps from the hardware. Similarly, not all signals are caused due to hardware traps (e.g., signals can be generated using the kill system call).
- **If multiple devices (e.g., USB mouse and USB keyboard) are connected to the same hardware port (with the same interrupt vector), and one of them raises and interrupt, how does the CPU find out whom to serve?**
  Interrupts are only a way of requesting immediate attention by *some* device registered on that vector. If multiple devices are registered, the interrupt handler will typically ask each of them if any of them needs its attention (using I/O ports or memory mapped I/O).

Memory Management

Let's now continue our discussion on memory management.

Discuss malloc(), free() implementations in kernel, and at process-level.

Discuss palloc(), free() implementation in kernel.

Fragmentation

- Fragmentation within kernel data structures : use kmalloc and kfree
- Fragmentation within process data structures : use process's malloc and free
- Fragmentation of process address space :
  - Last time : use multiple segments (complicates programming model)
  - Next time : Paging