

Chapter 9

Multitasking

9.1 Overview

9.2 Context switching

A *task* is a unit of work on which a processor can execution actions such as dispatch, execute, suspend, etc. A *multitasking environment* provides task management that arbitrates *cpu* time allocations to several tasks in such a way that they can run "concurrently". Each task receives a *time slice* of computation then it is interrupted and another tasks takes over the *cpu* for its own time slice of computation, and so on. Consider the diagram shown in Fig. 9.1 where the time slice allotted to each task $T_i, 1 \leq i \leq 4$ is d units of time until every task completes execution.

Since the *cpu* computes a single task thread at a time, tasks are referred to as *threads*. While one thread runs the other threads are *suspended*. The kernel *scheduler* uses the task priority assigned by the user to determine which thread will run next. If all threads are given the same priority the kernel executes all tasks in a job in a *round-robin* fashion as illustrated in Fig. 9.1.

Each task has its own stack or section of the overall stack, and a special section of memory rreferred to as its *context*. The context keeps a copy of the contents of all the *cpu* registers and the threads stack when it is suspended. For illustration consider two threads A and B with contexts as shown in Fig. 9.2. The mechanism to switch contexts between threads A and B is shown in Fig. 9.3.

The context switching block shown in Fig. 9.3 will switch to the task with the

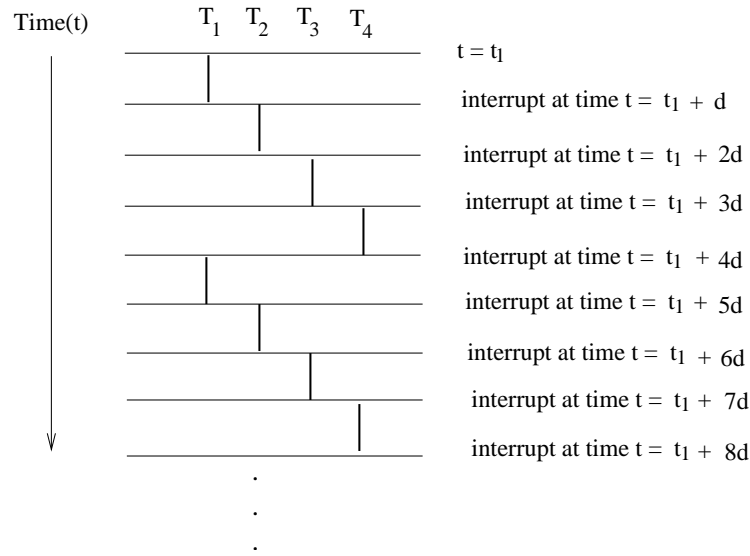


Figure 9.1: Time slicing four tasks

highest priority at the time of the interrupt.

9.3 Preemptive and non-preemptive multitasking

Tasks that can be interrupted are referred to as *preemptive*. Hardware interrupts trigger context switching; external events such as a tic from the I8253, or from any other device, let the handler (ISR) take over and switch context to another thread. Consider again two threads A and B to illustrate context management in Fig. 9.4.

Non-preemptive tasks are not interrupted. If a context switch must occur, a thread must explicitly call a kernel routine to switch to another thread. The context switch call is referred to as *yield*, and this form context switching is referred to as *cooperative* multitasking.

9.4 Critical sections

A sequence of instructions intended to access shared resources form what is known as a *critical section* in reference to the section in memory with the binary code resides. Critical sections must be protected against preemption by any other code accessing the same resources. A critical section once it is protected executes to completion. Mechanisms designed to protect critical section execution include *spin*

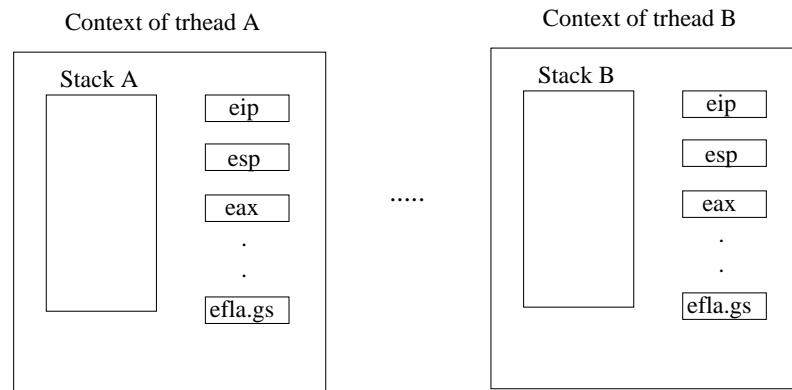


Figure 9.2: Context of threads A and B

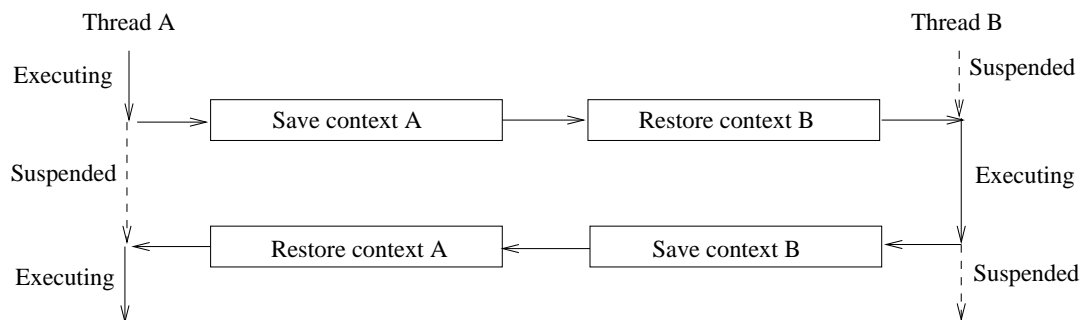


Figure 9.3: Context switch mechanism

locks and *mutual exclusion objects (mutex)*.

Spin locks. A binary flag is set before entering a critical section and cleared on exit. While the flag is set all other access to the same resource is blocked. Fig. 9.5 shows a possible implementation of a spin lock.

Note that if the flag is set other thread is using the resource and the current access request is blocked. A possible check of a flag uses the *xchg* instruction to ensure an atomic access to 1) access its current state, and 2) set it if it was cleared.

```

...
L1:  mov al, 1
      xchg byte [_flag], al
      or al, 1
      jz L1
...

```

If *_flag* is set, the resource is already taken and *xchg* does not change its state; the

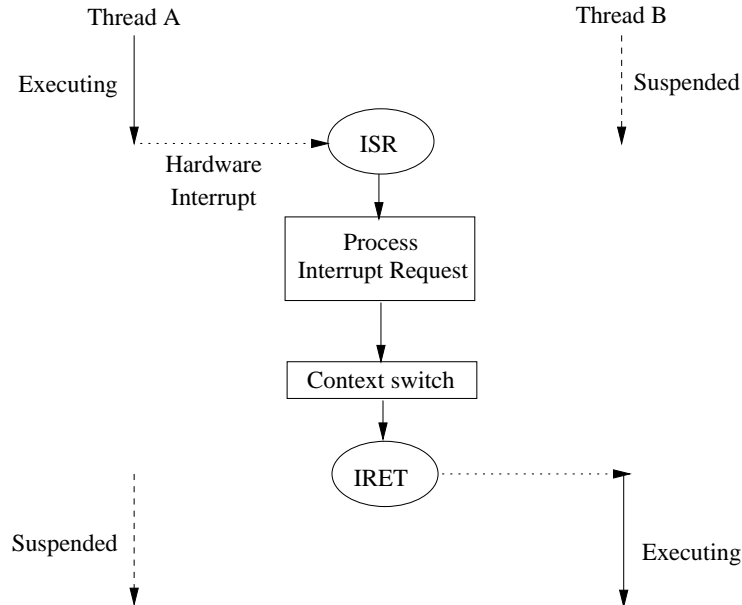


Figure 9.4: Context switch by hardware interrupts

checking continues for one more iteration. If `_flag` is cleared, the execution of the `xchg` instruction sets `_flag` to 1 but the non-zero result of the `or` instruction changes the flow out of the loop into the critical section of the code. An instruction such as `mov byte [_flag], 0` will clear the flag at the end of the critical section and release the resource.

MUTual EXclusion objects (Mutex). Using a mutex requires calling a kernel function on either side of the critical section. The first call blocks the thread until the mutex is available. If the mutex is available then the threads continues. At the end of the critical section the mutex is released via a second kernel call.

At most a single thread owns a mutex at anyone time. A *Semaphore* can be used to manage access to multiple resources; A semaphore is initialized to a count N that corresponds to the number of resources available within the a shared set. The counter is decremented or incremented as resources are allocated or realeased. When $N = 0$ threads are blocked until resources are released.

9.5 IA-32 support for task management

A task can be dispatched for execution in one of the following ways:

- An explicit *call* instruction,

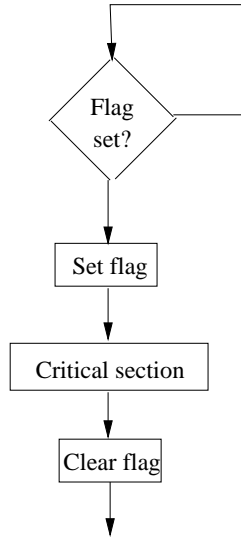


Figure 9.5: Spin lock implementation

- The execution of a jump instruction to the task,
- An implicit call to an interrupt-handler task,
- An implicit call to an exception-handler task, and
- Using an *iret* instruction (as shown in Fig. 9.4).

The context of the dispatched task is then loaded into the processor to initiate or continue execution. To support context switching in IA-32 processors, the following data structures are involved:

- Task-State Segments (TSS),
- Task-gate descriptors,
- TSS descriptors,
- Task registers
- An NT flag in the *eflags* register.

Task-State Segments (TSS). As a context switching occurs, the context of the currently executing task is saved in its own *Task-State Segment (TSS)* and execution is suspended. The context needed to restore a task to its execution state is stored in its

TSS. The TSS contents are organized as shown in Fig. 9.6. The processor updates dynamic fields when a task is suspended during a context switch. The dynamic fields that reflect the current state of a suspended task are the following:

- General-purpose registers fields
- Segment selector fields,
- The *eflag* register field,
- The *it eip* register field,
- Previous task link field,
- Local Descriptor Table select field, and
- A CR3 control register field used to point to a page directory.

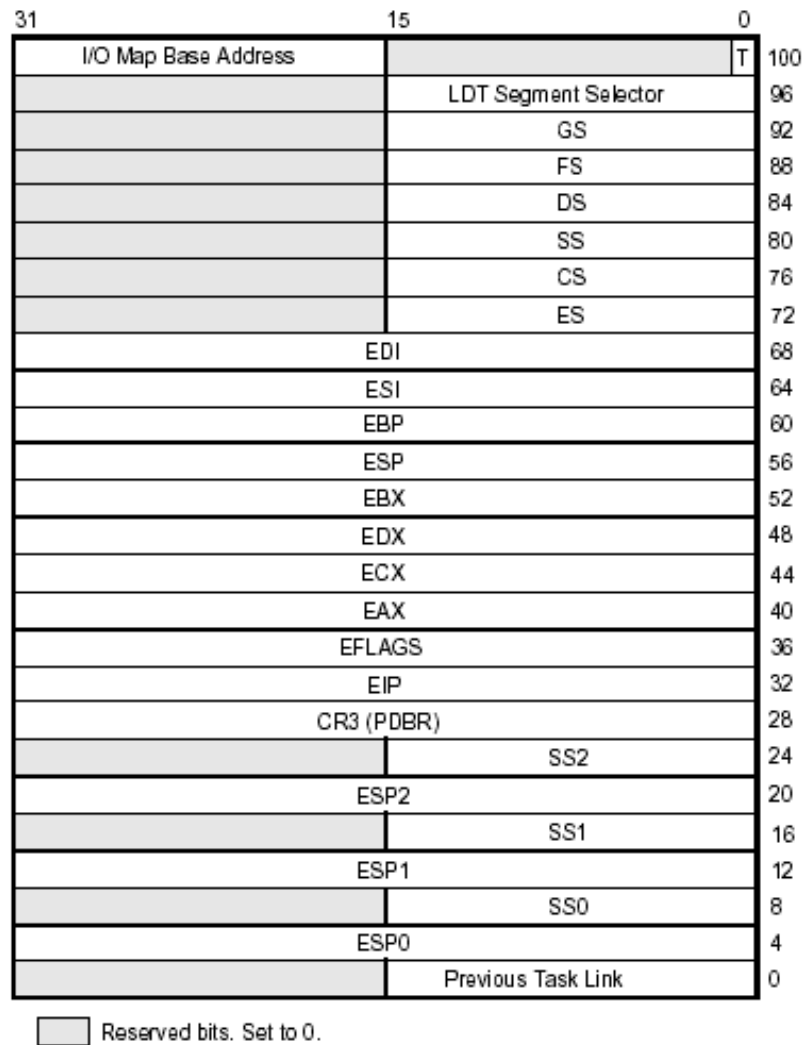


Figure 9.6: Task State Segment