Multitasking Systems

From OSDev Wiki

Multitasking Systems are operating systems (or even system extensions) which divide available processor time between several tasks automatically, creating the illusion that the tasks are running simultaneously.

Contents

Types of Multitasking Systems

There are many ways multitasking can be achieved.

Cooperative Multitasking

This concept runs an application until it exits or yields control back to the OS. Examples for cooperative multitasking systems are pre-X MacOS, or Windows 3.x.

In some single language cooperative multitasking systems, such as Oberon and ruby, the compiler/interpreter automatically ensures that the code will periodically yield control; it allows such program to run in multi-threading on non-preemptive OS such as DOS. As yielding isn't necessary anymore, I'm not sure if we can still say it's cooperative multitasking.

Preemptive Multitasking

In a preemptive multitasking system, the OS can take away control from (preempt) an application after a time slice is used up or a signal occurred. An example would be e.g. Linux, *BSD, post-3.x Windows, BeOS, or AmigaOS.

You can further subdivide these systems into those who can preempt applications, and those who can preempt *the kernel itself*. Linux (pre-2.6 kernel) is an example of the former, while e.g. AmigaOS is an example for the latter. This is a major concern for multimedia applications or any "soft" [Real-Time Systems] because a non-preemptive kernel introduces latencies that can ruin such "near real-time" performance.

How does it work

You have programs running. Each program has some binary code to be executed by the processor and an execution context made of e.g. registers state, stack content, etc.

Since you have a single CPU, you have a single program executed at a given moment and its execution context is the state of the cpu's registers while you may have plenty of programs sleeping, waiting for their turn with their context saved in OS's datastructures, right ?

Now, the OS has set up a timer interrupt, which causes the OS call a specific interrupt service routine at regular interval. When the timeslot for the current program runs out, the routine will save the current CPU context into a datastructure, select a new program to be run for the next timeslot, and load the CPU registers with the values that were saved in that process's datastructure.

If you still want to figure out, imagine a machine with an accumulator and a stack pointer. You can save the machine state, switch and restore another machine state with

```
PUSH                          ;; put the accumulator's content on stack
LOAD [current_pid]            ;; load the current PID in the accumulator
STORE_SP [context_table+ACC]  ;; save the stack of the suspended task

;; get somehow the PID for the next task into ACC, without using the stack

LOAD_SP [context_table+ACC]   ;; load the stack of the chosen task
STORE [current_pid]           ;; store it's PID
POP                           ;; get back the accumulator's content
IRET                          ;; end of interrupt
```

Now imagine two programs on that machine, e.g.

```
IMM 0  ;; into accumulator
here:
INC    ;; accumulator++
JMP here
```

and

```
IMM 0
there:
DEC  ;; accumulator --
JMP there
```

Sketch on a papersheet the memory of the machine, with the pid_table, the two stacks , the current_pid variable, and then go. You should be able to emulate the behavior of that machine if --say-- you have a interrupt every 10+some_number_you_get_by_rolling_a_dice machine instructions.

And you'll see it can continue working on program 1 after it has been interrupted for 10+ instructions by program 2.

Of course "10+dice" is not enough to get decent performances. On modern systems, your timeslice is usually of a few milliseconds, which makes millions of instructions! Moreover, you don't always switch when the timer arise (because you might want the system clock more accurate than the switching rate) and you can switch on other events (e.g. because a program needs to wait for something or because something important like the end of a disk request happened).

See Also

Articles

- Context Switching
- Monotasking Systems
- Scheduling Algorithms

Threads

- Designing a kernel to be preemptible
- Making a fully preemptible kernel

External Links

- Computer Multitasking on Wikipedia.

Retrieved from "http://wiki.osdev.org/index.php?title=Multitasking_Systems&oldid=9294"
Category: Task Models

- This page was last modified on 29 November 2009, at 19:32.
- This page has been accessed 47,912 times.