Real-time operating systems

6.1 Introduction: A **real-time operating system** (**RTOS**) is an operating system (OS) intended to serve real-time application requests. It must be able to process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.

A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is *jitter*. A *hard* real-time operating system has less jitter than a *soft* real-time operating system. The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category. An RTOS that can usually or *generally* meet a *deadline* is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS.

An RTOS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency; a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

6.2 Design philosophies

The most common designs are:

- Event-driven which switches tasks only when an event of higher priority needs servicing, called preemptive priority, or priority scheduling.
- Time-sharing designs switch tasks on a regular clocked interrupt, and on events, called round robin.

Time sharing designs switch tasks more often than strictly needed, but give smoother multitasking, giving the illusion that a process or user has sole use of a machine.

Early CPU designs needed many cycles to switch tasks, during which the CPU could do nothing else useful. For example, with a 20 MHz 68000 processor (typical of the late 1980s), task switch times are roughly 20 microseconds. (In contrast, a 100 MHz ARM CPU (from 2008) switches in less than 3

microseconds.) Because of this, early OSes tried to minimize wasting CPU time by avoiding unnecessary task switching.

Schedulin

In typical designs, a task has three states:

1. Running (executing on the CPU);
2. Ready (ready to be executed);
3. Blocked (waiting for an event, I/O for example).

Most tasks are blocked or ready most of the time because generally only one task can run at a time per CPU. The number of items in the ready queue can vary greatly, depending on the number of tasks the system needs to perform and the type of scheduler that the system uses. On simpler non-preemptive but still multitasking systems, a task has to give up its time on the CPU to other tasks, which can cause the ready queue to have a greater number of overall tasks in the ready to be executed state (resource starvation).

Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled. But the choice of data structure depends also on the maximum number of tasks that can be on the ready list.

If there are never more than a few tasks on the ready list, then a doubly linked list of ready tasks is likely optimal. If the ready list usually contains only a few tasks but occasionally contains more, then the list should be sorted by priority. That way, finding the highest priority task to run does not require iterating through the entire list. Inserting a task then requires walking the ready list until reaching either the end of the list, or a task of lower priority than that of the task being inserted.

Care must be taken not to inhibit preemption during this search. Longer critical sections should be divided into small pieces. If an interrupt occurs that makes a high priority task ready during the insertion of a low priority task, that high priority task can be inserted and run immediately before the low priority task is inserted.

The critical response time, sometimes called the flyback time, is the time it takes to queue a new ready task and restore the state of the highest priority task to running. In a well-designed RTOS, readying a new task will take 3 to 20 instructions per

ready-queue entry, and restoration of the highest-priority ready task will take 5 to 30 instructions.

In more advanced systems, real-time tasks share computing resources with many non-real-time tasks, and the ready list can be arbitrarily long. In such systems, a scheduler ready list implemented as a linked list would be inadequate.

Algorithms

Some commonly used RTOS scheduling algorithms are:

- Cooperative scheduling
- Preemptive scheduling
  - Rate-monotonic scheduling
  - Round-robin scheduling
  - Fixed priority pre-emptive scheduling, an implementation of preemptive time slicing
  - Fixed-Priority Scheduling with Deferred Preemption
  - Fixed-Priority Non-preemptive Scheduling
  - Critical section preemptive scheduling
  - Static time scheduling
- Earliest Deadline First approach
- Stochastic digraphs with multi-threaded graph traversal

Intertask communication and resource sharing

Multitasking systems must manage sharing data and hardware resources among multiple tasks. It is usually "unsafe" for two tasks to access the same specific data or hardware resource simultaneously. "Unsafe" means the results are inconsistent or unpredictable. There are three common approaches to resolve this problem:

Temporarily masking/disabling interrupts

General-purpose operating systems usually do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wishes. Some modern CPUs don't allow user mode code to disable interrupts as such control is considered a key operating system resource. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater system call efficiency and also to permit the application to have greater control of the operating environment without requiring OS intervention.

On single-processor systems, if the application runs in kernel mode and can mask interrupts, this method is the solution with the lowest overhead to prevent simultaneous access to a shared resource. While interrupts are masked and the current task does not make a blocking OS call, then the current task has *exclusive* use of the CPU since no other task or interrupt can take control, so the critical section is protected. When the task exits its critical section, it must unmask interrupts; pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the longest path through the critical section is shorter than the desired maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few instructions and contains no loops. This method is ideal for protecting hardware bit-mapped registers when the bits are controlled by different tasks.

Binary semaphores

When the shared resource must be reserved without blocking all other tasks (such as waiting for Flash memory to be written), it is better to use mechanisms also available on general-purpose operating systems, such as semaphores and OS-supervised interprocess messaging. Such mechanisms involve system calls, and usually invoke the OS's dispatcher code on exit, so they typically take hundreds of CPU instructions to execute, while masking interrupts may take as few as one instruction on some processors.

A binary semaphore is either **locked** or unlocked. When it is locked, tasks must wait for the semaphore to unlock. A binary semaphore is therefore equivalent to a mutex. Typically a task will set a timeout on its wait for a semaphore. There are several well-known problems with semaphore based designs such as priority inversion and deadlocks.

In priority inversion a high priority task waits because a low priority task has a semaphore, but the lower priority task is not given CPU time to finish its work. A typical solution is to have the task that owns a semaphore run at (inherit) the priority of the highest waiting task. But this simple approach fails when there are multiple levels of waiting: task *A* waits for a binary semaphore locked by task *B*, which waits for a binary semaphore locked by task *C*. Handling multiple levels of inheritance without introducing instability in cycles is complex and problematic.

In a deadlock, two or more tasks lock semaphores without timeouts and then wait forever for the other task's semaphore, creating a cyclic dependency. The simplest deadlock scenario occurs when two tasks alternately lock two semaphores, but in

the opposite order. Deadlock is prevented by careful design or by having floored semaphores, which pass control of a semaphore to the higher priority task on defined conditions.

Message passing

The other approach to resource sharing is for tasks to send messages in an organized message passing scheme. In this paradigm, the resource is managed directly by only one task. When another task wants to interrogate or manipulate the resource, it sends a message to the managing task. Although their real-time behavior is less crisp than semaphore systems, simple message-based systems avoid most protocol deadlock hazards, and are generally better-behaved than semaphore systems. However, problems like those of semaphores are possible. Priority inversion can occur when a task is working on a low-priority message and ignores a higher-priority message (or a message originating indirectly from a high priority task) in its incoming message queue. Protocol deadlocks can occur when two or more tasks wait for each other to send response messages.

Interrupt handlers and the scheduler

Since an interrupt handler blocks the highest priority task from running, and since real time operating systems are designed to keep thread latency to a minimum, interrupt handlers are typically kept as short as possible. The interrupt handler defers all interaction with the hardware if possible; typically all that is necessary is to acknowledge or disable the interrupt (so that it won't occur again when the interrupt handler returns) and notify a task that work needs to be done. This can be done by unblocking a driver task through releasing a semaphore, setting a flag or sending a message. A scheduler often provides the ability to unblock a task from interrupt handler context.

An OS maintains catalogues of objects it manages such as threads, mutexes, memory, and so on. Updates to this catalogue must be strictly controlled. For this reason it can be problematic when an interrupt handler calls an OS function while the application is in the act of also doing so. The OS function called from an interrupt handler could find the object database to be in an inconsistent state because of the application's update. There are two major approaches to deal with this problem: the unified architecture and the segmented architecture. RTOSs implementing the unified architecture solve the problem by simply disabling interrupts while the internal catalogue is updated. The downside of this is that interrupt latency increases, potentially losing interrupts. The segmented

architecture does not make direct OS calls but delegates the OS related work to a separate handler. This handler runs at a higher priority than any thread but lower than the interrupt handlers. The advantage of this architecture is that it adds very few cycles to interrupt latency. As a result, OSes which implement the segmented architecture are more predictable and can deal with higher interrupt rates compared to the unified architecture.

Memory allocation[edit]

Memory allocation is more critical in a real-time operating system than in other operating systems.

First, for stability there cannot be memory leaks (memory that is allocated, then unused but never freed). The device should work indefinitely, without ever a need for a reboot. For this reason, dynamic memory allocation is frowned upon. Whenever possible, allocation of all required memory is specified statically at compile time.

Another reason to avoid dynamic memory allocation is memory fragmentation. With frequent allocation and releasing of small chunks of memory, a situation may occur when the memory is divided into several sections, in which case the RTOS can not allocate a large continuous block of memory, although there is enough free memory. Secondly, speed of allocation is important. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable free memory block, which is unacceptable in an RTOS since memory allocation has to occur within a certain amount of time.

Because mechanical disks have much longer and more unpredictable response times, swapping to disk files is not used for the same reasons as RAM allocation discussed above.

The simple fixed-size-blocks algorithm works quite well for simple embedded systems because of its low overhead.

Examples

A common example of an RTOS is an HDTV receiver and display. It needs to read a digital signal, decode it and display it as the data comes in. Any delay would be noticeable as jerky or pixelated video and/or garbled audio.

Some of the best known, most widely deployed, real-time operating systems are

- LynxOS
- OSE
- QNX
- RTLinux
- VxWorks
- Windows CE
- Free RTOS