Real-Time Operating Systems and Languages (2)

Real-Time and Embedded Systems (M)

Lecture 11

UNIVERSITY
of
GLASGOW

Lecture Outline

- Real-time system embedded as *part of* a larger operating system
- Open system architecture
  - Discussion of concepts
  - Advantages and disadvantages
  - Implementation using a two level scheduler
- Case study of an open system: RTLinux

Reading for this lecture: Chapters 7.9 and 12.5–12.7

Real-time Embedded Systems

- What is an embedded system?
  - "An embedded system is a special-purpose computer system built into a larger device."
  - "An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is specifically designed for a particular kind of application device. Industrial machines, automobiles, medical equipment, cameras, household appliances, airplanes, vending machines, and toys (as well as the more obvious cellular phone and PDA) are among the myriad possible hosts of an embedded system."

⇒ Special purpose, limited resources, not generally upgradeable

Real-time Embedded Systems

- Embedded systems are *closed*:
  - Run a fixed suite of applications, known *a priori*
  - Tasks are scheduled according to some well-known algorithm
  - Generally static and require predictability
  - Prove, or exhaustively demonstrate, correctness
  - Limited resources, tailored to the task at hand
  - Dedicated operating systems, scheduler support, etc.
- Embedded systems form part of the wider world:
  - Interact with the world through sensors and actuators
  - Often part of a wider system, comprising other embedded and general purpose systems
- Separation of concerns:
  - Embedded controllers engineered separately to other systems, including other embedded systems

Open Architecture for Real-Time Systems

- Advantage of traditional embedded systems: resources dedicated; predictability is guaranteed
- Disadvantage: dedicated resources are typically underused
  - Predictable, but wasteful
  - Many applications have both general-purpose and real-time components

- Desire a single system that can run general purpose and real-time applications simultaneously
  - An *open system architecture* that can support many different classes of application, removing the distinction between embedded and general purpose systems
  - Not always suitable, but can give large savings for some application types

Objectives

- Independent design choice
  - The developer of an application can use a scheduling discipline best suited to that application to control execution and resource access, independent of other applications on system
- Independent validation
  - If system validates assuming it runs alone on a processor with normalised speed *S*, it will run on a virtual share of a real processor with equivalent performance
- Independent admission and timing guarantees
  - New real-time tasks subject to admission test. If accepted, schedulability is guaranteed regardless of other applications in the system

Independently developed and validated real-time applications can share a system with other real time and non-real time applications

Implementation

- Open system architecture can only be implemented on a strictly partitioned *virtual machine*
  - Partition processor time
  - Control access to global resources

- Each application submits requirements (e.g. task characteristics, type of scheduler, etc.) to virtual machine monitor that performs acceptance test
  - The monitor partitions the physical resources into distinct virtual machines and runs schedulers for each application
  - The partitioning can be implemented using a *two-level scheduler*

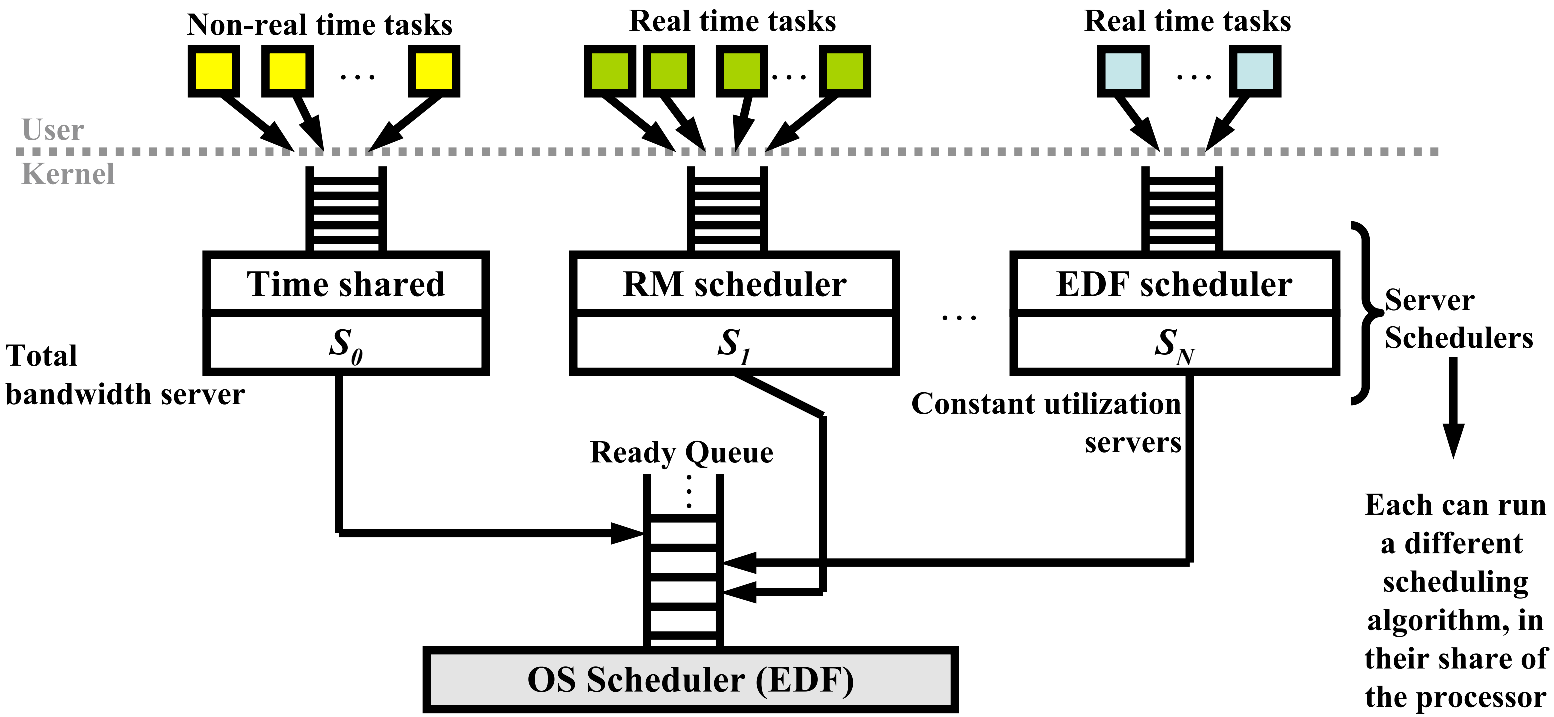
Two-Level Scheduler

- Recall:
  - A *constant utilization server* consumes a fraction $\tilde{u}_i$ of the processor
  - A *total bandwidth server* uses at least a fraction $\tilde{u}_i$ and claims idle time
  - Both run under an EDF scheduling algorithm and are defined by certain consumption and replenishment rules

- Consider a system comprising:
  - A total bandwidth server, S_0, using a fraction $\geq \tilde{u}_0$
  - A set of constant utilization servers S_i for $i = 1, 2, \ldots, n$ each using fraction $\tilde{u}_i$ of the processor
    - If $\sum_{i=0}^{n} \tilde{u}_i \leq U_{\max}$ will fairly share a processor with S_0
    - The maximum schedulable utilization, $U_{\max}$, depends on the properties of the server tasks, and their workload
  - An EDF scheduler, running these servers

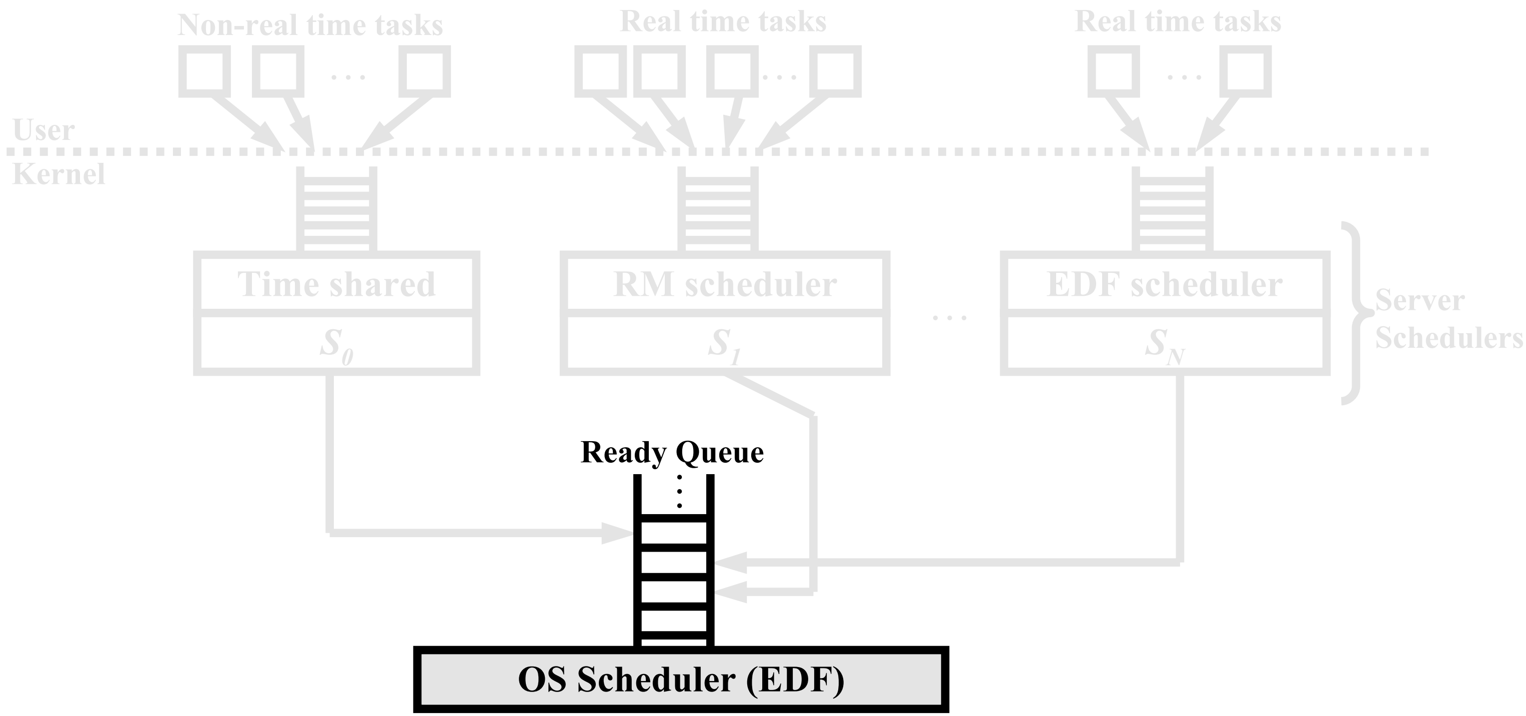
Operation of a Two-Level Scheduler

- The servers, S_i, are scheduled according to an EDF algorithm by the *OS scheduler*
- Each server runs an internal *server scheduler* to schedules jobs within the server, subdividing the time allocated to that server



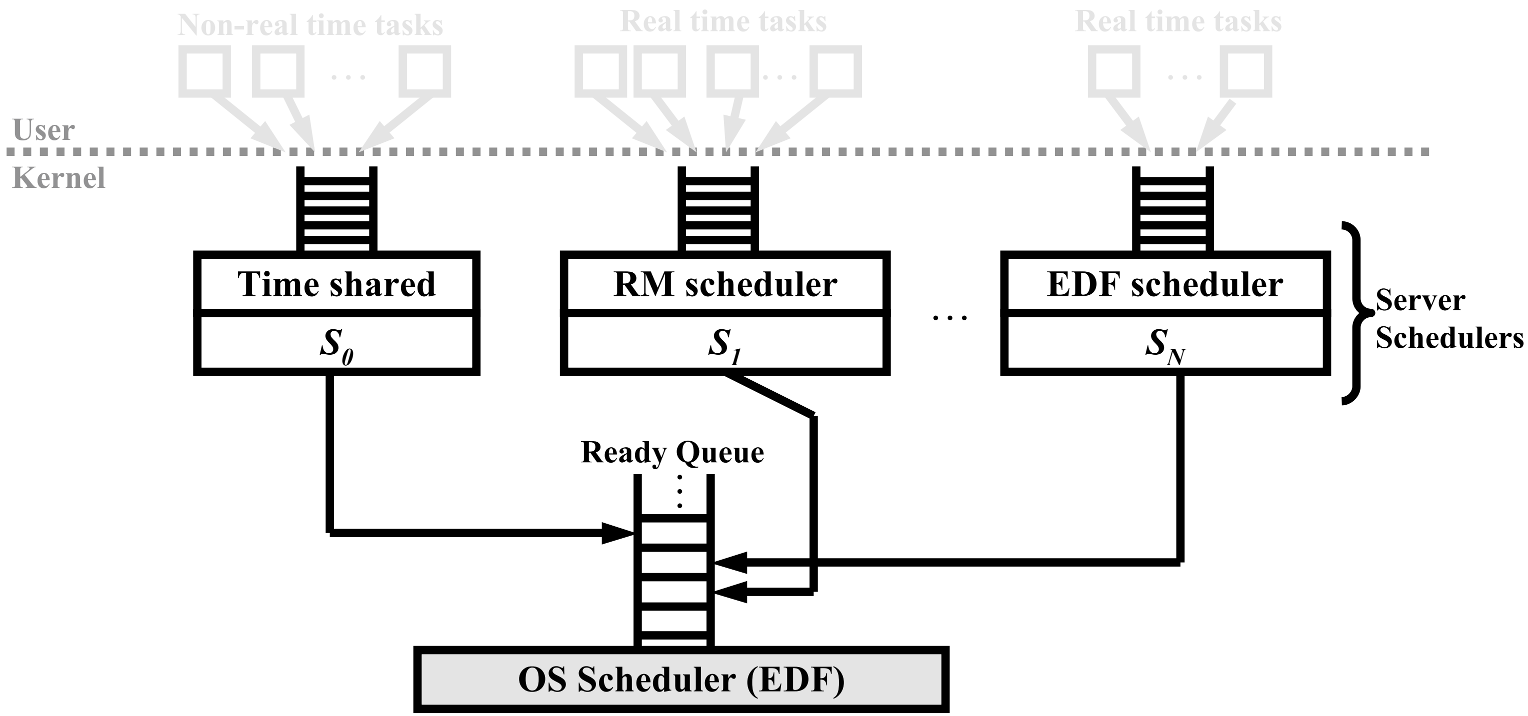
Operation of a Two-Level Scheduler

- The OS scheduler maintains an EDF ready queue, used to select which server to execute
  - Servers are eligible to run if they have work to do, and budget remaining
  - The server with the earliest deadline among the ready servers is scheduled



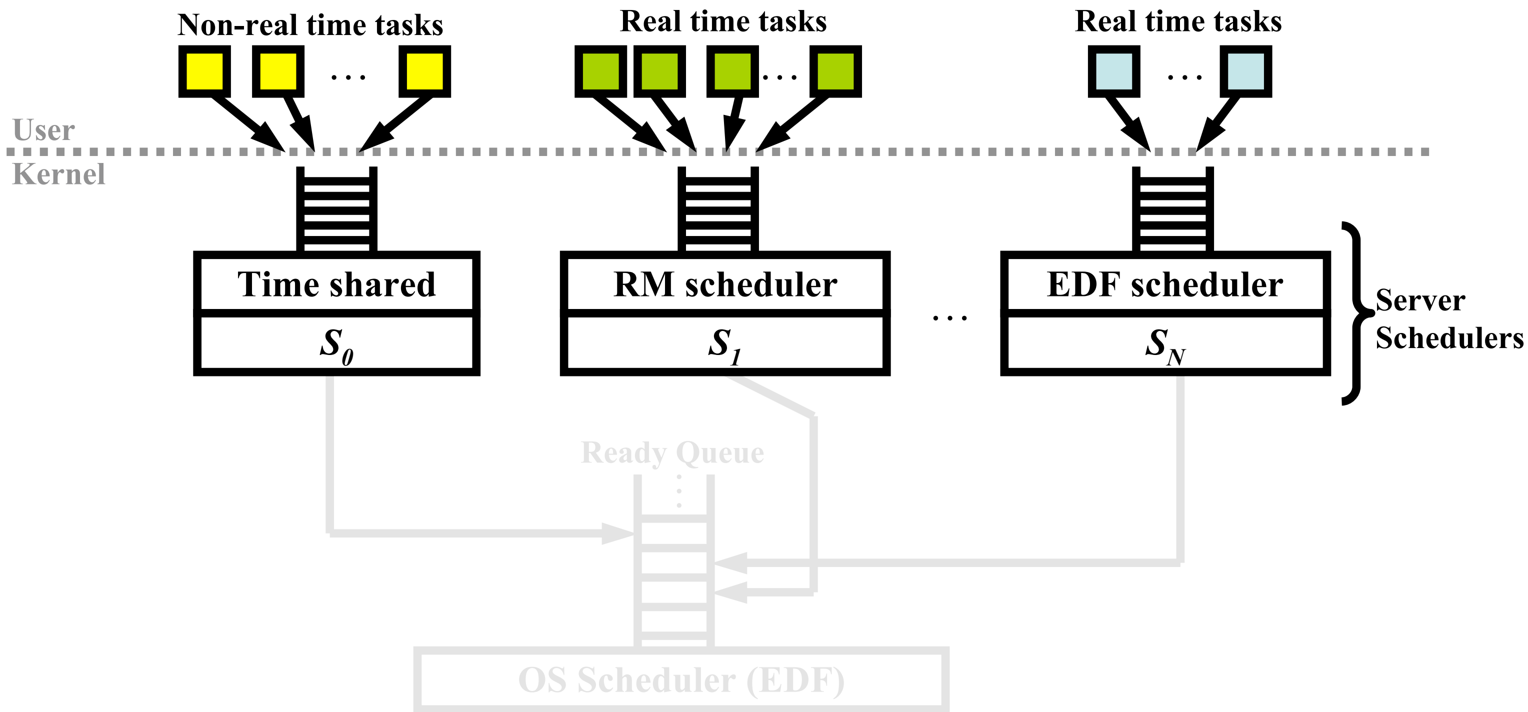
Operation of a Two-Level Scheduler

- Both levels of scheduler are implemented in the kernel, to avoid doubling the context switch overhead
  - Applications see a virtual machine: their server and its internal scheduler
  - The underlying OS scheduler is invisible to applications



Operation of a Two-Level Scheduler

- The virtual machines are strictly partitioned from each other:
  - When executed by the OS scheduler, the virtual machine server scheduler picks one of these threads to execute, according to its local policy
  - Each can have different policy, and schedule its threads according to a different algorithm



Interactions Between Schedulers

- The underlying OS scheduler runs EDF to partition available time for each virtual machine scheduler
  - Needs to know when the next scheduling event – *deadline* – will occur on each virtual machine
    - The two levels of scheduler must cooperate for efficiency
  - Depends on the application running in each virtual machine
    - Non-real time applications
      - No deadlines
      - The total-bandwidth server, S_0, internally schedules the non-real time tasks according to a time sharing algorithm with time slice x
    - Predictable real-time applications
      - Contain periodic tasks with fixed release times and known resource access patterns
      - Known event times
    - Unpredictable applications must be estimate time of next event
      - Contains aperiodic or sporadic tasks, or periodic tasks with significant release time jitter; occurrence of scheduling events only known at run-time

Predictable Real Time Applications

- Can run within predictable real time applications within constant utilization server, and meet deadlines, provided:
  - The *required capacity* of the task is available
    - A job need e units of execution time at normal speed to meet timing constraints
    - Is run on a slow processor, speed $u < 1$
      - u denotes the fraction of the original processor speed
    - Multiply the execution time e of all jobs by $1/u$ to check if the system is still schedulable on the slow processor
    - The minimum fraction of speed at which the application is schedulable is its *required capacity*
  - Appropriate replenishment rules used for the server
    - Vary depending on scheduling algorithm used within virtual machine scheduler
      - Clock driven schedulers
      - Priority scheduled but not preemptable
      - Priority scheduled and preemptable, with known event times

Replenishment Rules: Clock Driven

- A clock driven scheduler is characterised by a cyclic frame of size f and the workload appears to the server as a single thread
  - Scheduled on a constant utilization server of size $\tilde{u}_i$ equal to required capacity
    - Ready for execution at the start of each cyclic frame
    - Execution time of $f \cdot \tilde{u}_i$ each cycle
    - Budget replenished each cycle, deadline set to beginning of next cycle
  - Loops using a fixed fraction of the processor time
  - Executes the application according to its pre-computed cycle

Replenishment Rules: Non-preemptable Periodic

- Priority scheduled non-preemptable tasks can also be scheduled within a constant utilization server
  - The server scheduler orders jobs in the application within its allocation according to the scheduling algorithm requested by the application
  - Since jobs are non-preemptable, the server must not be pre-empted while a job is running
    - Otherwise a job could be pre-empted by another server running on the OS scheduler
    - Limits maximum schedulable utilization of the complete system (*not just this server*):
      - Let B denote maximum execution time of all jobs
      - Let D_{min} denote the minimum relative deadline of all jobs
      - All servers are schedulable provided $\Sigma\tilde{u}_i < 1 - B/D_{min}$
      - Implications on the acceptance test for the open system, since EDF non-optimal in this case

Replenishment Rules: Preemptable Periodic (1)

- Priority scheduled preemptable tasks executed on a server that is similar to a constant utilization server, but with slightly different replenishment rules
  - Why different rules?
    - Consider two jobs J_1 and J_2 running on a slow processor with speed 0.25
      - Each job has execution time 0.25
      - Job J_1 is released at 0.5 and must complete by 1.5
      - Job J_2 is released at 0.0 and must complete by 2.0
    - Execution:
      - Job J_2 starts at time 0.0, by time 0.5 has executed for 0.125
      - Job J_1 starts at time 0.5, pre-empts J_2, and executes to completion by 1.5
      - Job J_2 resumes at 1.5, executes to completion at 2.0
      - All deadlines are met

Replenishment Rules: Preemptable Periodic (2)

- Priority scheduled preemptable tasks executed on a server that is similar to a constant utilization server, but with slightly different replenishment rules
  - Why different rules?
    - Consider the same jobs, running on ¼ of a normal speed processor
      - Job J_2 starts at time 0.0 with server budget 0.25 and deadline 1
      - Job J_2 uses the budget, and completes before time 0.5
      - Job J_1 is released at time 0.5, but the server budget is gone
      - At time 1.0 the server budget is replenished to 0.25 and its deadline set to 2.0; the server is eligible to run, and will execute job J_1 when it runs
      - Because the server deadline is 2.0, the server may not execute until after J_1 has missed it's deadline at 1.5
    - Problem: J_2 consumed more execution time than it would on the slow processor, preventing J_1 from running
      - Used 0.25 compared to 0.125 on slow processor
      - Can be considered a form of priority inversion

Replenishment Rules: Preemptable Periodic (3)

- Priority scheduled preemptable tasks executed on a server that is similar to a constant utilization server, but with slightly different replenishment rules
  - Modified replenishment rules solve this:
    - Let t be latest of the current deadline, or current time
    - Let t' be the release time of the next job in the task
    - Budget $= \min(e_i, (t' - t) \cdot \tilde{u}_i)$ rather than e_i
    - Deadline $= \min(t + e_i/\tilde{u}_i, t')$ rather than $t + e_i/\tilde{u}_i$
  - Prevent jobs consuming more time than they would on a slower processor, if they would be limited by pre-emption, by explicitly taking into account pre-emption time
    - (Modified replenishment rules also needed for non-preemptable periodic tasks)

Unpredictable Real Time Applications

- If jobs unpredictable, cannot derive accurate replenishment rules
  - Run the server under the constant utilization rules, giving $q\tilde{u}_i$ units of budget every q time units
  - The scheduling quantum, q, is a key parameter
- The server can over-budget the application up to the size of the scheduling quantum
  - Can result in priority inversion, as before…

- If a bound, t', on time to the next event can be given, the server can be scheduled with
  - Budget $= (t' + q - t).s_i$
  - Deadline $= t' + q$
- Bounds the size of the scheduling quantum, and hence the duration of any priority inversion

Scheduling Overhead

- Might think that the two-level scheduler is very inefficient
  - Not so if all applications are predictable:
    - Same number of context switches
    - More work to determine what to run, but insignificant compared to context switch overhead
  - In unpredictable applications running, overhead depends on the scheduling quantum
    - Small quantum gives better real-time performance, but higher overhead
    - 30% overhead not unusual, but still often better than dedicated hardware

- If unpredictable jobs are rare, the two level scheduler works well and allows real-time and non-real-time jobs to share a processor

Admission Control

- All jobs start execution in non-real time mode, so they don't disrupt already running real-time jobs
- A job may switch to real-time scheduling on its own server, subject to an acceptance test
- Jobs must provide:
  - Required capacity $\tilde{u}_i$ and scheduling algorithm
  - Maximum execution time B_i of all non-preemptable sections
  - Existence of aperiodic/sporadic tasks, if any
  - Shortest relative deadline D_{min}

- Job is accepted if $\Sigma\tilde{u}_i < 1 - \max(B_i/D_{min})$

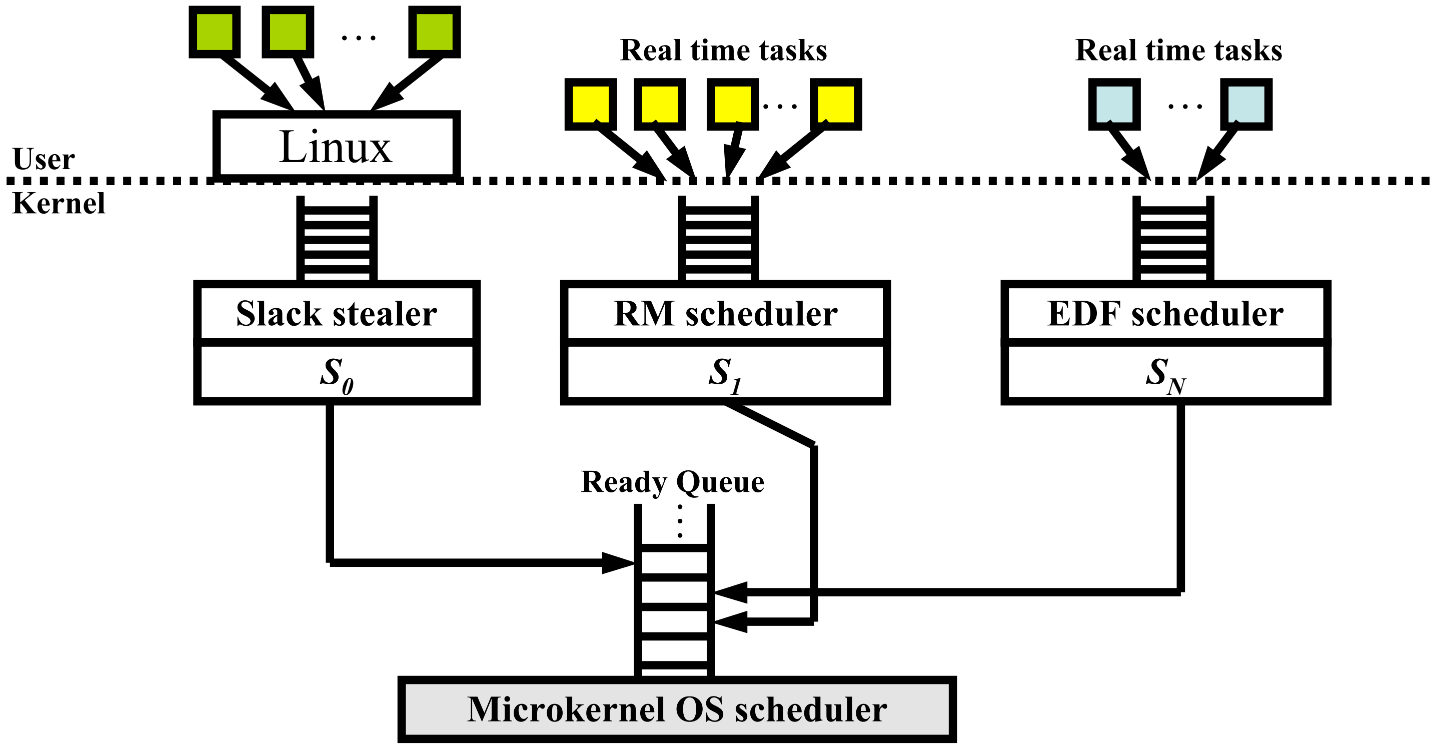
Constraints and Implementations

- The open system architecture provides a conceptually clean way to share resources between tasks with different requirements
- Disadvantage:
  - Several applications share a hardware resource, so failure of the hardware or OS scheduler can take down an entire set of applications
  - Trade-off cost saving for potential reduction in reliability

- Full concept not widely implemented:
  - Similar, but less powerful, systems are widely used commercially:
    - Older versions of Symbian mobile phones running a real-time microkernel to handle the voice processing, with SymbianOS running as a background task to support the UI and user applications
    - RTLinux

Case Study: RTLinux

- A simple example of a two-level scheduler
  - The OS scheduler is a microkernel real-time operating system
  - Real-time tasks run directly on the microkernel
    - RM and EDF schedulers provided
  - Linux runs as the idle task



Case Study: RTLinux

- A modified Linux kernel runs above the microkernel
  - All hardware access is arbitrated by the microkernel
  - Interrupts emulated in software on the microkernel
  - Linux can *always* be pre-empted if a real-time task needs to run
- Communication between real-time and non-real-time tasks done by FIFO buffers, locked into memory
  - Appear as normal devices (**`/dev/rft1`**) under Linux
  - Non-blocking and atomic access from the real-time kernel

- Conceptually, RTLinux maps closely onto the open system architecture
  - Differs in the details

Summary

- Concepts of real-time on embedded systems
- The idea of an open system architecture, to support a range of application types on a single system
- Strategies for implementing the open system architecture, using a two-level scheduler
- Overview of RTLinux, as a simple system using a two-level scheduler