Introduction to Embedded Systems

Edward A. Lee & Sanjit Seshia
UC Berkeley
EECS 124
Spring 2008

Lecture 18: Operating Systems and Microkernels

Source

This lecture draws heavily from:

Giorgio C. Buttazzo, *Hard Real-Time Computing Systems,* Springer, 2004.

Responsibilities of a Microkernel (a small, custom OS)

- Scheduling of threads or processes
  - Creating and termination of threads
  - Timing of thread activations
- Synchronization
  - Semaphores and locks
- Input and output
  - Interrupt handling

A Few More Advanced Functions of an Operating System – Not discussed here…

- Memory management
  - Separate stacks
  - Segmentation
  - Allocation and deallocation
- File system
  - Persistent storage
- Networking
  - TCP/IP stack
- Security
  - User vs. kernel space
  - Identity management

Outline of a Microkernel

- Main:
  - set up periodic timer interrupts;
  - create default thread data structures;
  - dispatch a thread (procedure call);
  - execute main thread (idle or power save, for example).
- Thread data structure:
  - copy of all machine registers
  - address at which to resume executing the thread
  - status of the thread (e.g. blocked on mutex)
  - priority, WCET, and other info to assist the scheduler

Outline of a Microkernel

- Dispatching a thread:
  - *disable interrupts;*
  - save state (registers) into current thread data structure;
  - save return address from the stack for current thread;
  - determine which thread should execute (scheduling);
  - if the same one, enable interrupts and return;
  - copy thread state into machine registers;
  - replace program counter on the stack for the new thread;
  - *enable interrupts;*
  - return.
- Timer interrupt service routine:
  - dispatch a thread.

When can a new thread be dispatched?

- *Under non-preemptive scheduling*:
  - When the current thread completes.
- *Under Preemptive scheduling:*
  - Upon a timer interrupt
  - Upon an I/O interrupt (possibly)
  - When a new thread is created, or one completes.
  - When the current thread blocks on or releases a mutex
  - When the current thread blocks on a semaphore
  - When a semaphore state is changed
  - When the current thread makes any OS call
    - file system access
    - network access
    - …

The Focus Today:
How to decide which thread to schedule?

Priorities

- Preemptive vs. non-preemptive
- Periodic vs. aperiodic tasks
- Fixed priority vs. dynamic priority
- Priority inversion anomalies
- Other scheduling anomalies

Preemptive Scheduling

Assume all threads have priorities that are either statically assigned (constant for the duration of the thread) or dynamically assigned (can vary).

Assume further that the kernel keeps track of which threads are "enabled" (able to execute, e.g. not blocked waiting for a semaphore or a mutex or for a time to expire).

Preemptive scheduling:

- At any instant, the enabled thread with the highest priority is executing.
- Whenever any thread changes priority or enabled status, the kernel can dispatch a new thread.

Rate Monotonic Scheduling

Assume *n* tasks invoked periodically with:

- periods $T_1, \ldots, T_n$
- worst-case execution times (WCET) $C_1, \ldots, C_n$
  - assumes no mutexes, semaphores, or blocking I/O
- no precedence constraints
- fixed priorities
- preemptive scheduling

If any priority assignment yields a feasible schedule, then priorities ordered by period (smallest period has the highest priority) also yields a feasible schedule.

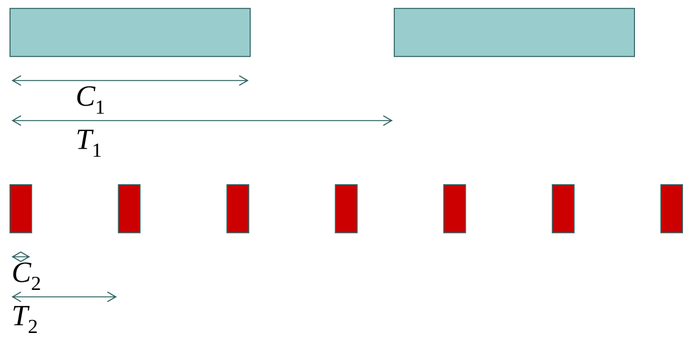
RMS is optimal in the sense of feasibility.

Liu and Leland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," J. ACM, 20(1), 1973.

Feasibility for RMS

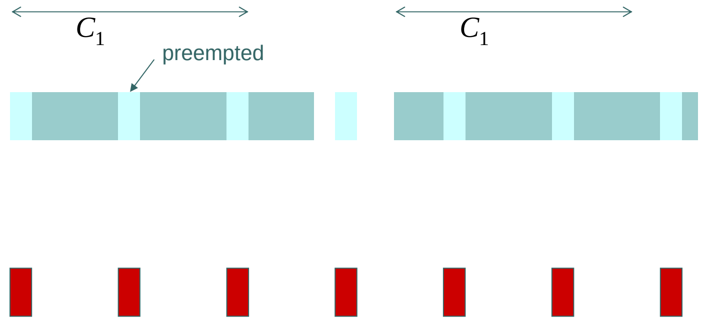
Feasibility is defined for RMS to mean that every task executes to completion once within its designated period.

Showing Optimality of RMS: Consider two tasks with different periods



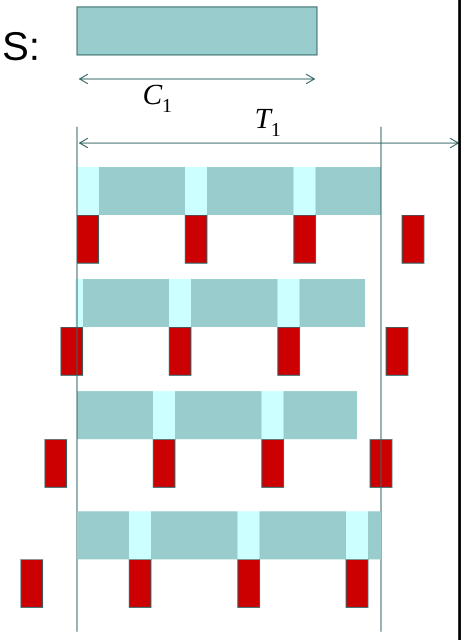
Non-preemptive schedule is not feasible.

Showing Optimality of RMS: Consider two tasks with different periods



Preemptive schedule with the red task having higher priority is feasible. Note that preemption of the blue task extends its completion time.

Showing Optimality of RMS: Alignment of tasks



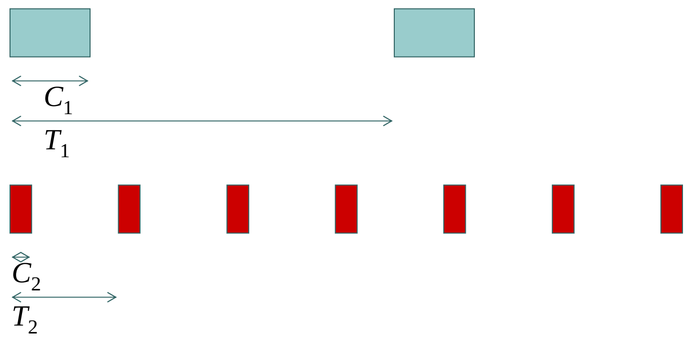
Completion time of the lower priority task is worst when its starting phase matches that of higher priority tasks.

Thus, when checking schedule feasibility, it is sufficient to consider only the worst case: All higher priority tasks start their cycles at the same time.

Showing Optimality of RMS: (for two tasks)

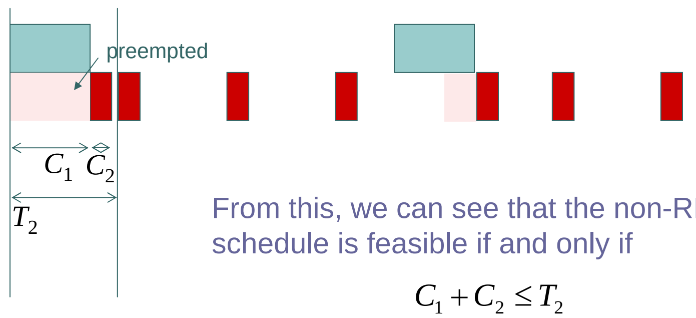
It is sufficient to show that if a non-RMS schedule is feasible, then the RMS schedule is feasible.

Consider two tasks as follows:



Showing Optimality of RMS: (for two tasks)

The non-RMS, fixed priority schedule looks like this:



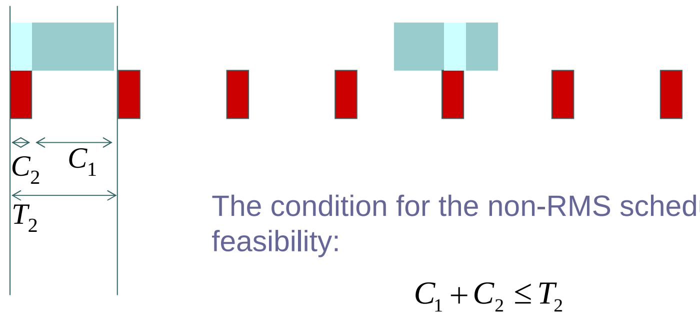
From this, we can see that the non-RMS schedule is feasible if and only if

$$C_1 + C_2 \leq T_2$$

We can then show that this condition implies that the RMS schedule is feasible.

Showing Optimality of RMS: (for two tasks)

The RMS schedule looks like this:



The condition for the non-RMS schedule feasibility:

$$C_1 + C_2 \leq T_2$$

is clearly sufficient (though not necessary) for feasibility of the RMS schedule. QED.

Comments

- This proof can be extended to an arbitrary number of tasks (though it gets much more tedious).
- This proof gives optimality only w.r.t. feasibility. It says nothing about other optimality criteria.
- Practical implementation:
  - Timer interrupt at greatest common divisor of the periods.
  - Multiple timers

Deadline Driven Scheduling: 1. Jackson's Algorithms: EDD (1955)

Given *n* independent one-time tasks with deadlines $d_1, \ldots, d_n$, schedule them to minimize the maximum *lateness*, defined as

$$L_{\max} = \max_{0 \le i < n} \{ f_i - d_i \}$$

where f_i is the finishing time of task i. Note that this is negative iff all deadlines are met.

Earliest Due Date (EDD) algorithm: Execute them in order of non-decreasing deadlines.

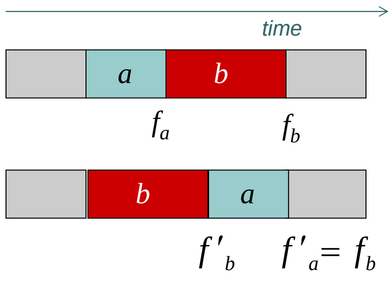
Note that this does not require preemption.

EDD is Optimal in the Sense of Minimizing Maximum Lateness

To prove, use an interchange argument. Given a schedule S that is not EDD, there must be tasks a and b where a immediately precedes b in the schedule but $d_a > d_b$. We can prove that this schedule can be improved by interchanging a and b. Thus, no non-EDD schedule is optimal, so the EDD schedule must be optimal.

Consider a non-EDD Schedule *S*

There must be tasks a and b where a immediately precedes b in the schedule but $d_a > d_b$



$$L_{\max} \le \max\{f_a - d_a, f_b - d_b\} = f_b - d_b$$

$$L'_{\max} \le \max\{f'_a - d_a, f'_b - d_b\}$$

Case 1: $f'_a - d_a > f'_b - d_b$.
Then: $L'_{max} \le f'_a - d_a = f_b - d_a \le L_{max}$
(because: $d_a > d_b$).

Case 2: $f'_a - d_a \le f'_b - d_b$.
Then: $L'_{max} \le f'_b - d_b \le L_{max}$
(because: $f'_b < f_b$).

Theorem: $L'_{max} \le L_{max}$.
Hence, S' is no worse than S.

Deadline Driven Scheduling:
1. Horn's algorithm: EDF (1974)

Extend EDD by allowing tasks to "arrive" (become ready) at any time.

Earliest deadline first (EDD): Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all arrived tasks is optimal w.r.t. minimizing the maximum lateness.

Proof uses a similar interchange argument.

Using EDF for Periodic Tasks

- The EDF algorithm can be applied to periodic tasks as well as aperiodic tasks.
  - Simplest use: Deadline is the end of the period.
  - Alternative use: Separately specify deadline (relative to the period start time) and period.

Comparison of EDF and RMS

- Favoring RMS
  - Scheduling decisions are simpler (fixed priorities vs. the dynamic priorities required by EDF. EDF scheduler must maintain a list of ready tasks that is sorted by priority.)
- Favoring EDF
  - Since EDF is optimal w.r.t. maximum lateness, it is also optimal w.r.t. feasibility. RMS is only optimal w.r.t. feasibility. For infeasible schedules, RMS completely blocks lower priority tasks, resulting in unbounded maximum lateness.
  - EDF can achieve full utilization where RMS fails to do that (see the problem set).
  - EDF results in fewer preemptions in practice, and hence less overhead for context switching.
  - Deadlines can be different from the period.