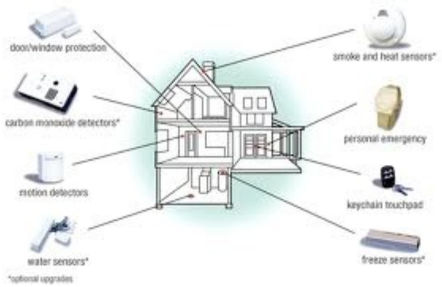
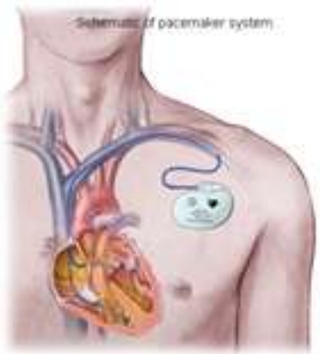
Real-time schedulers & RTOS core services

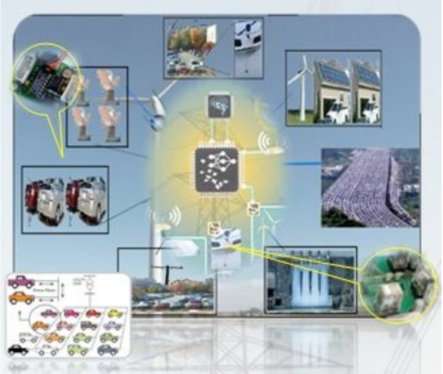
CS 202: Advanced Operating Systems

UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE



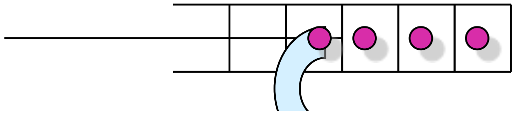


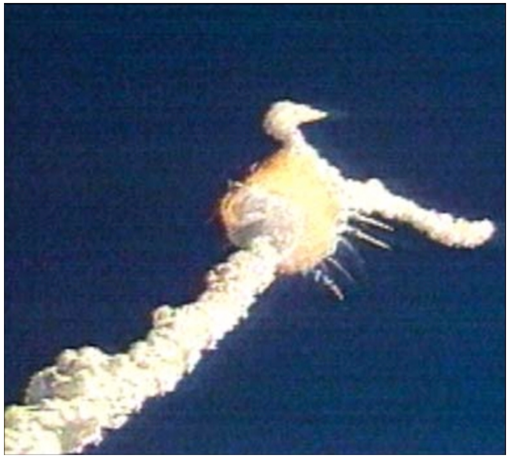
Real-time Systems

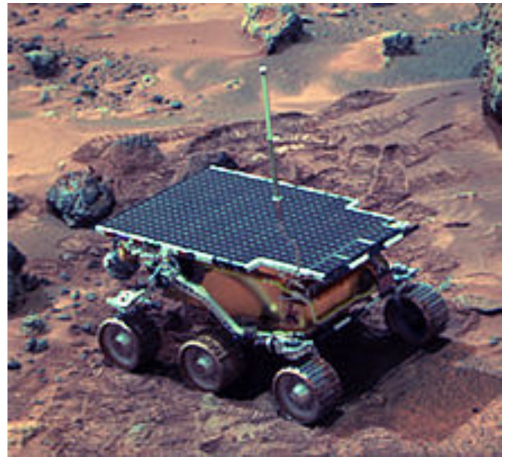


What are Real-Time Systems?

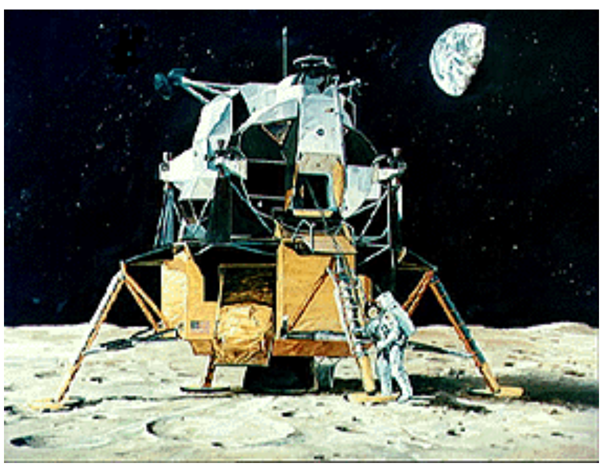
- A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences
- Real-time systems have a dual notion of correctness:
  - Logical correctness ("do the right thing")
  - Temporal correctness ("do it on time")
- A system wherein predictability is more important than performance
- Example: A robot arm picking up objects from a conveyor belt



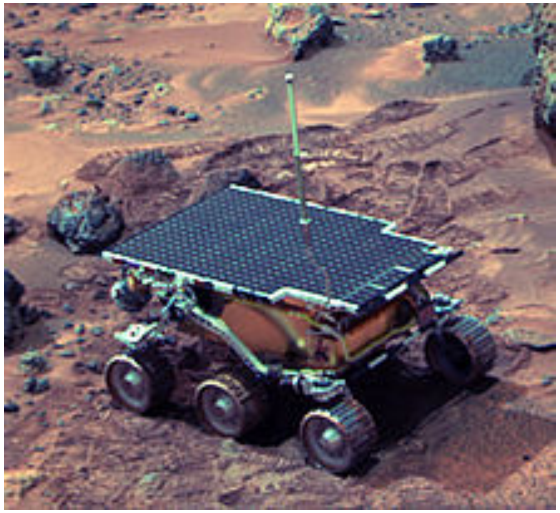




Infamous Real-Time Systems

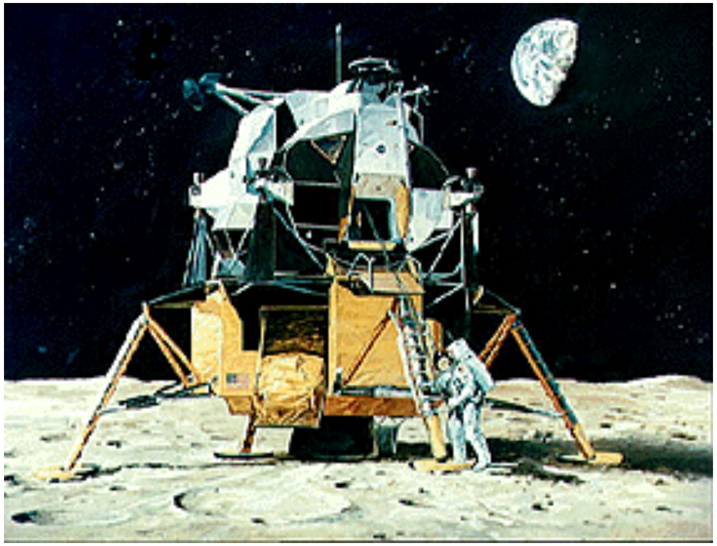


What Really Happened on Mars?



- Unmanned spacecraft landed on Mars in 1997
  - Reset reinitializes everything and terminates the execution of the current ground commanded activities
  - Caused by the classical real-time priority inversion problem (mutex-protected shared data area)
  - Tests before launch did not consider “higher data rate than we could have ever imagined” case
  - Later solved by applying a “priority inheritance protocol” first published in RTSS 1990

Apollo 11



- The first spacecraft landing on the moon
  - Software problem almost caused the landing to be nearly aborted
  - Engineers in charge decided to ignore the problem
  - Control system overloaded => buffer overflow => alarm signals
  - Low-priority jobs were not executed, but not critical at all

Patriot Missile Control System

- System used to protect Saudi Arabia during Gulf War
- Task: detect flying objects, perform prediction, trajectory matches prediction => Patriot missile launched
- 1991.2.25 --- Scud missile hit city of Dhahran, classified as false alarm (no Patriot missile launched), 28 soldiers killed!
- Software bug: real-time clock accumulating a delay of 57 microseconds per minute; the battery in question has been in operation for over100 hours => 343 milliseconds

Lessons To Be Learned

- If we cannot guarantee something not to go wrong, it will go wrong
- For real-time systems, arguments like "it works now" or "I think it will work" are worthless
- Extensive testing (which can find many bugs) **never guarantee correctness**

Lessons To Be Learned

- If we cannot guarantee something not to go wrong, it will go wrong
- For real-time systems, arguments like “it works now” or “I think it will work” are worthless
- Extensive testing (which can find many bugs) **never guarantee correctness**

Correctness should be guaranteed at design time by formal analysis & verification with clearly stated assumptions and assertions

Priority-driven Scheduling

- Priority-driven are dominantly applied in real-time systems
  - Jobs have priorities, and whenever the scheduler selects a job to run, it selects the highest-priority ready job
- Also called work-reserving or list scheduling
  - Never leave the processor idle intentionally
  - Defined by the list of job priorities

Outline

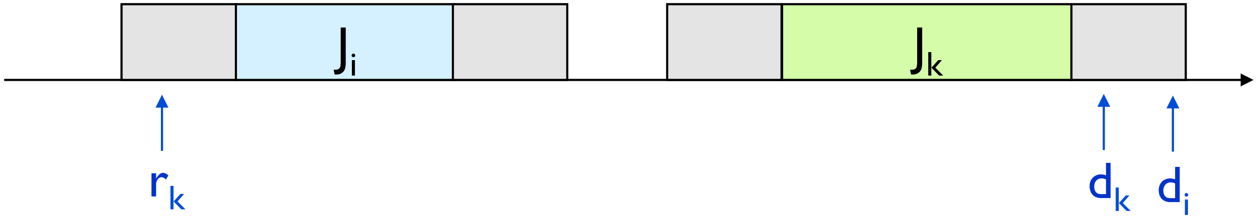
- We consider both earliest-deadline-first (EDF) and least-laxity-first (LLF) (often called least-slack-time-first) scheduling

Optimality of EDF

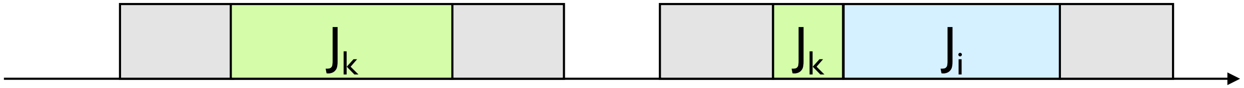
[Liu and Layland, Horn] When preemption is allowed and jobs do not self-suspend nor contend for resources, the EDF algorithm can produce a feasible schedule of a set J of jobs with arbitrary release times and deadlines on a processor if and only J has feasible schedules.

Proof

- We show that any feasible schedule of J can be systematically transformed into an EDF schedule
- Suppose parts of two jobs J_i and J_k are executed out of EDF order:

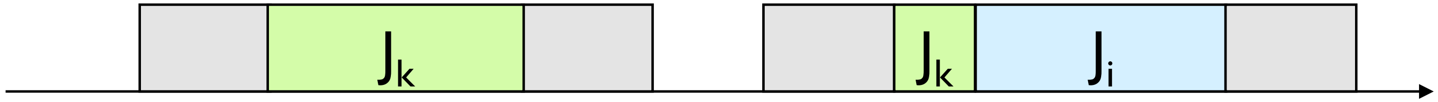


- This situation can be corrected by performing a "swap":



Proof (Continued)

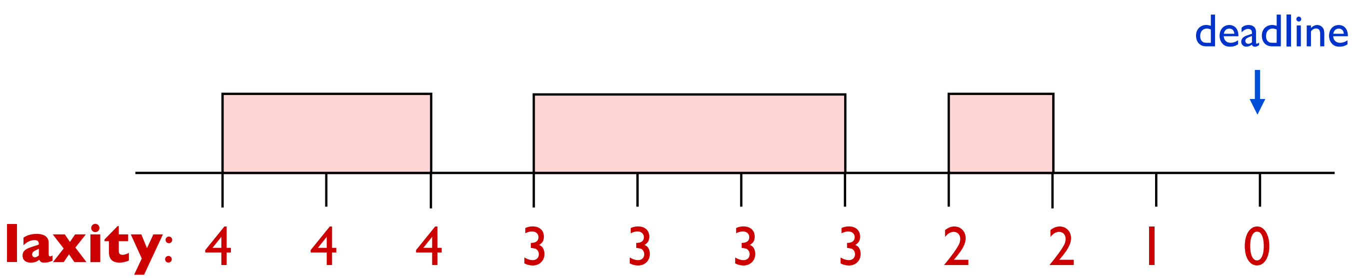
- If we inductively repeat this procedure, we can eliminate all out-of-order violations
- However, the resulting schedule may still fail to be an EDF schedule because it has idle intervals where some job is ready:



- Such idle intervals can be easily eliminated by moving some jobs forward:

LLF Scheduling

- Definition: At any time t, the slack (or laxity) of a job with deadline d is equal to (d - t) minus the time required to complete the remaining portion of the job



- LLF Scheduling: The job with the smallest laxity has highest priority at all times

↳ more overhead compared to EDF
⇒ Calculating laxity

Optimality of LLF

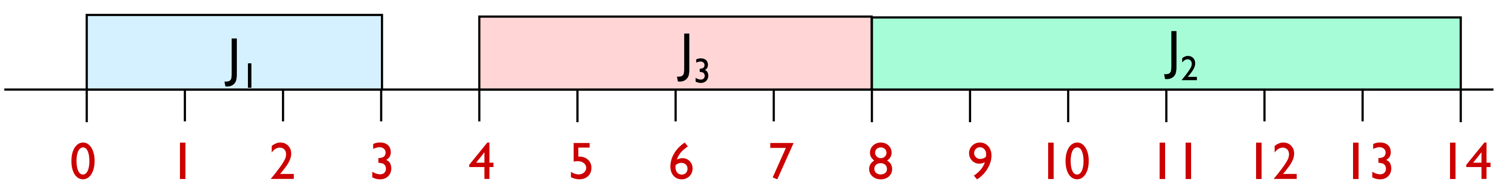
Theorem 4-3: When preemption is allowed and jobs do not self-suspend nor contend for resources, the LLF algorithm can produce a feasible schedule of a set J of jobs with arbitrary release times and deadlines on a processor if and only J has feasible schedules.

- The proof is similar to that for EDF and is left as an exercise
- Question: Which of EDF and LLF would be preferable in practice?

Preemptive v.s. Nonpreemptive EDF

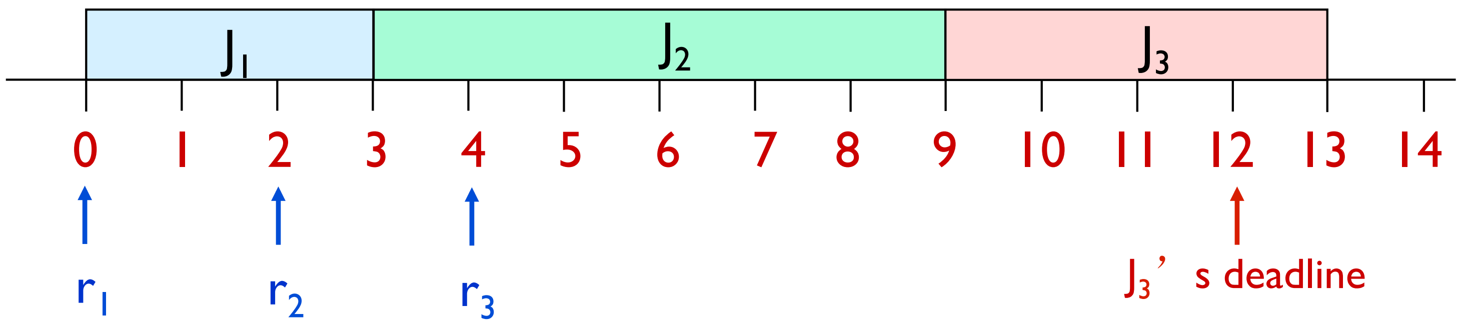
<u>Theorem</u>: Nonpreemptive EDF is not optimal.

- Proof by a counterexample: Consider a system of three jobs J_1, J_2, and J_3 such that $(r_1, e_1, d_1) = (0, 3, 10)$, $(r_2, e_2, d_2) = (2, 4, 8)$, $(r_3, e_3, d_3) = (4, 6, 14)$. Here is a feasible schedule:



Proof (Continued)

- Here shows the nonpreemptive EDF schedule, we see a deadline miss!



- Question: Should we conclude from this result that preemptive EDF is always better than non-preemptive EDF in practice?
  – Note: The EDF optimality proof assumes there is no penalty due to preemption
  – Are there other practical issues we have ignored?

↳ High context switch overhead

Real-time operating systems

Core Services

- Most RTOSs offer APIs for at least:
  - Timers and clocks (e.g., to allow threads to "sleep")
  - Scheduling primitives (priority, preemption control)
  - Synchronization (e.g., semaphores)
  - Interrupt control (e.g., to register ISRs, etc.)
  - Access to I/O devices (e.g., map I/O ports into a thread's address space)
  - Reserving memory (e.g., to prevent memory from being paged out by the OS) ↳ Predictability

Time Services

- Every system has at least one hardware device that contains a **counter**
  - Counts down from some initial value and generates an interrupt when 0 is reached
- A counter can (usually) be programmed to act as either:
  - **Periodic Timer:** Generates an interrupt every *p* time units
  - **One-Shot Timer:** Must be re-initialized after each interrupt
- The counter can serve as a clock and/or a timer
  - **Clock:** A counter and ISR that together keep (system) time
  - **Timer:** A counter and ISR that triggers an event e at time *t* (relative or absolute)

Clock Resolution

- The **resolution** of a clock is the granularity of time provided by the clock
  - The hardware resolution may be on the order of nanoseconds
  - However, the resolution seen by the application is usually in the order of a few microseconds to milliseconds
- Clock resolution is fundamentally limited by:
  - CPU frequency (interrupts are only processed between instructions)
  - Max. instruction length (on modern Intel systems, some instructions can execute for hundreds of cycles)
  - Interrupt latency (time-service interrupts may be delayed)
    ↳ Split interrupt handling

Software Clocks

- "Current time" has many definitions
- Most RTOSs keep track of at least the following clocks:
  - Absolute (wall-clock) time (including daylight saving time, leap seconds, etc.)
  - Monotonic time (without discontinuities, e.g., microseconds since system startup)
  - Per-thread execution time (CPU allocation)

Software Timers

- Most OSs (including all real-time POSIX-compliant systems) allow a thread to have its own software timers
- Threads can create, set, cancel, and destroy timers
- When a timer is created by a thread, the kernel creates a timer data structure, which includes:
  - The clock associated with the timer
  - The thread that created the timer
  - The expiration of the timer in absolute or relative time, once set
  - The timer type: one-shot or periodic
  - An event handler: a routine to execute when the timer expires

Software Timers

- **Synchronous Timers** suspend the calling thread until the timer has expired
  - For example, timer_sleep(relative_time) or timer_sleep_until(absolute_time)
- **Asynchronous Timers** count down while the calling thread executes
  - When the timer expires, the associated handler is called
  - Threads may have multiple asynchronous timers active
  - Asynchronous timers can be used as a timing monitor to log missed deadlines (see Figure 12-4 in Liu)

Scheduler Activation

- **Event-driven** → Sleeping beauty design
  - The scheduler is invoked on job release, job completion, and when a job blocks/resumes
  - Job releases are triggered by interrupts / timers
- **Quantum-driven**
  - The scheduler is invoked by a periodic timer
  - Periodic job releases may be triggered by the scheduler
- Hybrid approaches are common
  - In current Linux versions, the scheduler is invoked in response to events, but also by a periodic scheduler tick (usually once every millisecond)

Scheduling Mechanisms

- Virtually all RTOSs support Priority-driven Scheduling
  - Many RTOSs provide 256 priority levels
    - We have already seen that 256 levels approximates an ideal system
  - General-purpose OSs usually provide 10-20 levels

    20 levels not enough for real time systems
    - NT has only 16 real-time priority levels
    - Exception: Linux, which provides 100 levels.

      More granular control
  - The priority is usually set (once) when the thread is created and stored in the TCB
  - The kernel (usually) maintains a ready queue for each priority level
  - Dispatching the highest priority ready thread requires finding the highest priority nonempty queue
    - How do we do this?
    - What is the complexity of this?

UCR

Dispatching Threads

- An efficient way of doing this in software:
  - Data structures:
    - Assume a K-bit queue-status word and Ω priority queues
    - Let each bit of the word represent the status of Ω/K priority queues
    - If a bit is set, one of the Ω/K associated queues has a ready thread
  - Finding a ready thread:
    - If the queue-status word > 0 then a thread is ready
      - Perform a binary search on the word (comparing its value to powers of 2) to find the highest-order bit set
      - Find the highest priority non-empty queue of the Ω/K queues associated with the highest-order bit set, and dequeue the job at the head of the queue
      - Clear the associated bit in the queue-status word if all Ω/K queues are empty
  - Complexity: at most $\Omega/K + \log_2 K$
    - At most 13 comparisons for a 32-bit queue-status word and 256 priorities

Thread Dispatching in Practice

- Many architectures have instructions for finding the highest non-zero bit in hardware
  - On x86, the Bit Scan Forward (BSF) instruction yields the index of the highest non-zero bit in a 32-bit word
- Using one bit per queue, the scheduler can find the highest-priority non-empty queue with at most eight BSF instructions (assuming 256 priority levels)
  - Only four instructions on 64-bit systems

(256/64 = 4)

Preemption Locks

- Almost all RTOSs allow a thread to disable preemption by disabling interrupts
  - Preemption locks are usually used to protect critical sections in both the kernel and the application threads
- However, some also support disabling preemption by setting a mode control flag to disable the scheduler
  - With this feature, hardware interrupts (such as the clock) are not disabled when context switching is disabled

Resource Access Control

- The most common resource access control method used in real-time systems is a user-level implementation of the nonpreemptable critical section (NPCS) protocol:
  - Each thread disables preemption (using some form a preemption lock) just before accessing a resource
  - The thread enables preemption (releases the preemption lock) as soon as it is done with the resource
- How does this guarantee that the resource is always available when the thread accesses it?

RTOS Design Goals

- Most RTOSs are used in embedded systems with limited resources (CPU, memory)

  ⇒ Microkernel

  - **Design time:** the OS should be modular and extensible
  - **Run time:** often a fixed set of tasks, only little changes in workload
  - OS services designed to limit worst-case latencies
- In contrast, best-effort OS (e.g., Windows, Linux)
  - "One size fits all"
  - Must handle wide variety of workloads
  - Optimized for average-case performance

Capabilities of Commercial RTOSs

- Commercially available RTOSs:
  - LynxOS, QNX, pSOSystem, Nucleus RTOS, VRTX, and VxWorks
- Each of these systems shares the following attributes:
  - Compliant or partially compliant to the Real-Time POSIX API Standard:
    - Preemptive, fixed-priority scheduling
    - Standard synchronization primitives (mutex and message passing)
    - Each also has its own API
  - Modular and scalable:
    - The kernel is small so that it can fit in ROM in embedded systems
    - I/O, file, and networking modules can be added

Shared Attributes (Continued)

- Fast and efficient:
  - Most are microkernel systems
  - Low overhead
  - Small context switch time, interrupt latency, and semaphore get/release latency: usually one to a few microseconds
  - Nonpreemptable portions of kernel functions are highly optimized, short, and as deterministic as possible
  - Many have system calls that require no trap: applications run in kernel mode
- Flexible scheduling:
  - All offer at least 32 priority levels: min required by real-time POSIX
  - Most offer 128 or 256 priority levels
  - FIFO or RR scheduling for equal-priority threads

Shared Attributes (Continued)

split interrupt in two parts
⇒ Small part ⇒ "acknowledging" interrupt executed first
⇒ complex part ⇒ Scheduled for later

- Support split interrupt handling
- Relatively High Clock and Timer Resolution:
  - Nominal timer resolution in the nanosecond range
  - Actual timer resolution in the microsecond range
- No Paging or Swapping: ⇒ Maintaining predictability
  - May provide logical-to-physical memory translation but no paging
  - May not offer memory protection: often the kernel and all tasks execute in kernel mode, sharing one common address space
  - Level of memory protection may be settable (ranging from "none" to "private virtual memory")
- Optional Networking Support:
  - Can be configured to support TCP/IP with an optional module

"Partial Paging"
⇓
"Cache colouring"
⇓
Partitioning cache lines among processes

UCR