OS and Scheduler Basics

Raj Rajkumar

Lecture #3

Administrivia

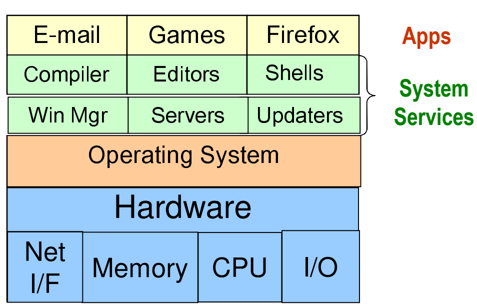
- Lab #1 Handout next week
- Recitation on Thursday (tomorrow)
  - Announce group membership
  - Receive hardware kit
  - Start prepping for Lab #1

Outline

- OS Task Abstractions
  - Processes and Threads
  - OS Scheduler
- Back to Real-Time Systems
- Rate-Monotonic Scheduling
  - Worst Arrival Phasings
  - Least Upper Scheduling Bound
- Summary

What is an Operating System?

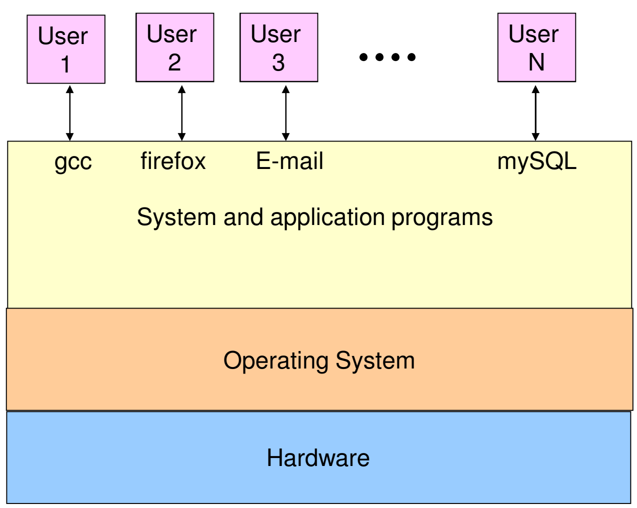
- The software layer that lies between a computer user and the computer hardware
  - Hides details of programming low-level devices
  - Separates users and processes from one another
  - Provides elegant programming interfaces for individual applications



Computer System Components

- Hardware
  - Provides basic computing resources: CPU, memory, I/O (disk, mouse, keyboard, display), network interfaces
- Operating System
  - Controls and coordinates the use of the hardware among various application programs for different users
- Application Programs
  - Define the ways in which the system resources are used to solve the computing problems of users
    - e.g. database systems, 3D games, business applications
- Users
  - People, machines, and other computers

Abstract View of System Components



Operating System Concepts

- Process Management
- Memory Management
- File Management
- I/O System Management
- Secondary Storage Management
- Networking
- User Security

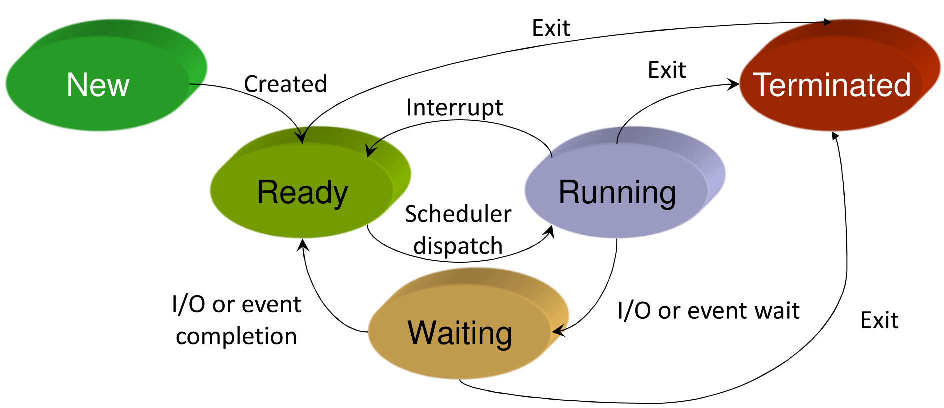
Process Management

- A ***process*** is a program in execution within its own logical address space
- A process contains
  - Address space
  - Address space contents (read-only code, global data, heap and stack)
  - PC, Stack pointer, values in register set
  - Opened file handles (open sockets, etc.)
- A process needs certain resources, including CPU time, memory, files, and I/O devices
- The OS is responsible for the following activities for process management
  - Process creation and deletion
  - Process suspension and resumption
  - Provision of facilities for:
    - process synchronization
    - process communication

Process State

- As a process executes, it changes *state*
  - New: The process is being created
  - Ready: The process is waiting to be assigned to a processor
  - Running: The process is executing on the processor
  - Waiting: The process is waiting for some event (e.g. I/O, timeout) to occur
  - Terminated: The process has completed execution

Process State Diagram



Process Control Block (PCB)

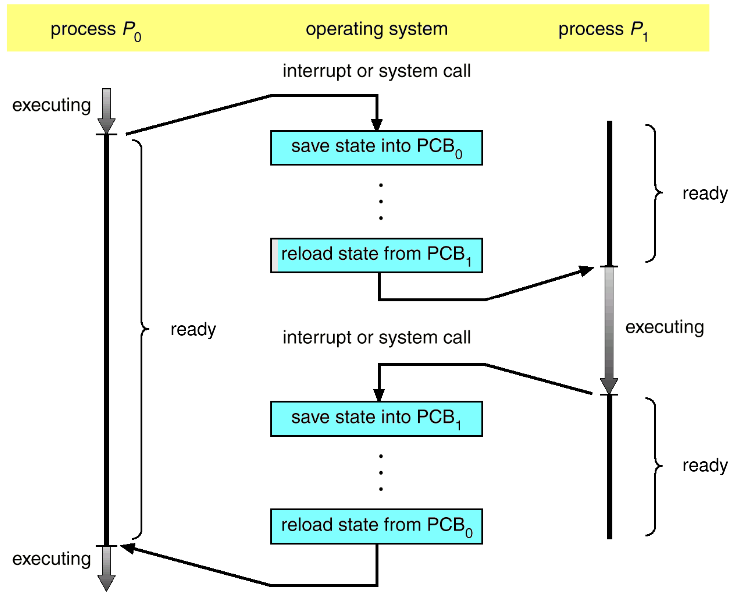
Information associated with each process stored by the OS

- Process state
- Program counter
- CPU registers
  - Content switched in/out during context switch
- CPU scheduling attributes
  - e.g. priority
- Memory-management information
  - e.g. page table, segment table
- Accounting information
  - e.g. PID, user time, constraint
- I/O status information
  - list of I/O devices allocated
  - list of open files
  - list of signals

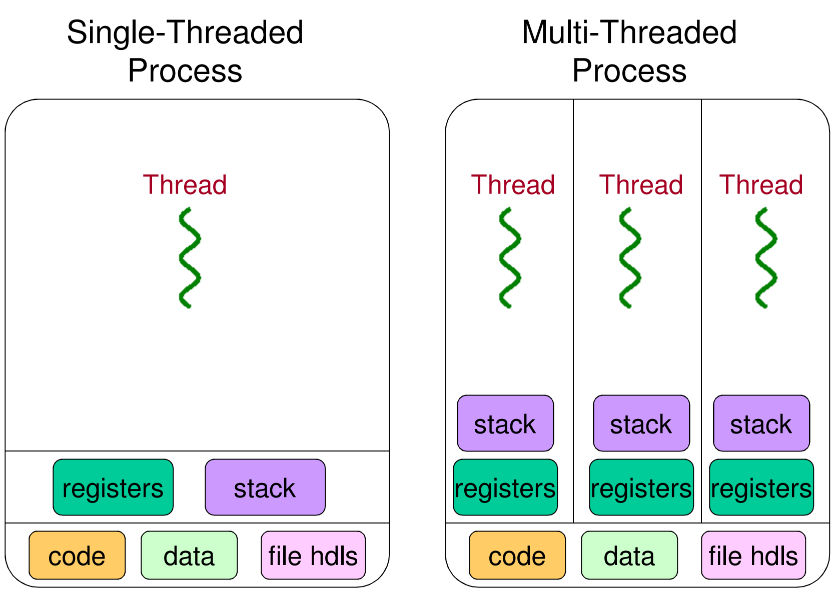
Process Control Block

Process state
Process id
Program Counter
Regiser set values
Memory limits
Scheduling attributes
List of open files
…
Pointer(s) to next PCB

CPU Switch From Process 0 to Process 1



Single and Multithreaded Processes



Examples of Threads in Processes

- A web server (e.g. Apache)
  - One thread accepts a web request
  - When a request comes in, a separate thread is created to service the request
  - Many threads can support thousands of client requests
  - A fixed pool of threads can be pre-created
- A web browser (e.g. FireFox)
  - One thread displays images
  - One thread retrieves data from network
- A word processor (e.g. Word)
  - One thread displays graphics
  - One thread reads keystrokes
  - One thread performs spell checking in the background
  - One thread performs grammar checks in the background
- RPC or RMI (Java)
  - One thread receives message
  - Message service uses another thread

Threads *vs.* Processes

Threads

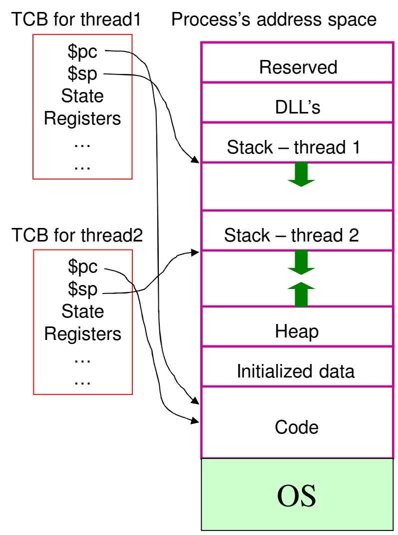
- A thread cannot live on its own, it must live within a process
- A thread has no exclusive data or heap segment
- There can be more than one thread in a process, the first thread calls `main` and has the process's stack
- Inexpensive creation
- Inexpensive context switching between threads of the same process
- If a thread dies, its stack is reclaimed by the process

Processes

- There must be at least one execution point within a process
- A process has code, data, heap and stack segments
- *(Threads within a process share code/data/heap, share I/O, but each has its own stack and registers)*
- Expensive creation
- Expensive context switching across processes
- If a process dies, its resources are reclaimed by the OS

Thread Implementation

- Process owns *address space*
- Threads within a process share the same address space
- Process Control Block (PCB) contains process-specific info
  - PID, owner, heap pointer, active threads and pointers to thread info
- Thread Control Block (TCB) contains thread-specific info
  - Stack pointer, PC, thread state, register values, pointer to process, …



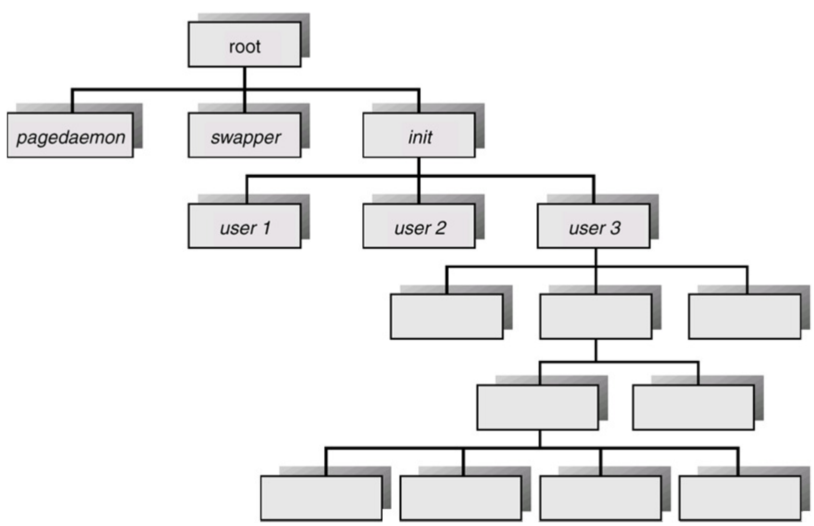
Benefits of Threads

- Responsiveness
  - When one thread is blocked, other threads in the same application process (such as your web browser) still respond
    - e.g. download images while allowing your interaction
- Resource Sharing
  - Share the same address space
  - Reduce overhead (e.g. memory)
- Economy
  - Creating a new process costs memory and resources
  - E.g. in Solaris, 30 times slower in creating process than thread
- Utilization of MP Architectures
  - Threads can be executed in parallel on shared-memory multiple processors
  - Increase concurrency and throughput

Process Creation

- A parent process creates children processes, which in turn create other processes, forming a process tree (or hierarchy)
- Resource sharing options:
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution:
  - Parent and children execute concurrently
  - Parent waits to exit until children terminate

Process Tree on a UNIX System



Process Creation (Cont.)

- Address space
  - Child's space is duplicate of parent's
  - Child process has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates a new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program

C Program Forking Separate Process

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
int pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
   }
   else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
        /* parent waits for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
   }
}
```

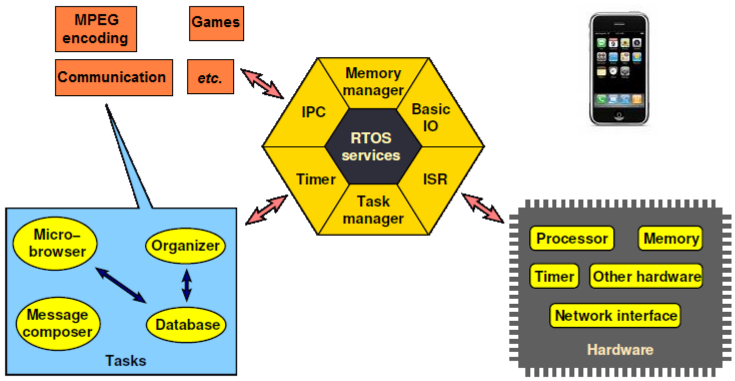
Process Termination

- Process executes last statement and asks the operating system to destroy it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

Now, back to real-time systems…

An RTOS-CentricView

- RTOS: Real-Time Operating System



Real-time System

- A **real-time system** is a system whose specification includes both logical and temporal correctness requirements.
  - **Logical Correctness:** Produces correct outputs.
    - Can by checked, for example, by Hoare logic.
  - **Temporal Correctness:** Produces outputs at the right time.
    - It is not enough to say that "brakes were applied"
    - You want to be able to say "brakes were applied at the right time"
      - In this course, we spend much time on techniques for checking temporal correctness.
      - The question of how to specify temporal requirements, though enormously important, is shortchanged in this course.

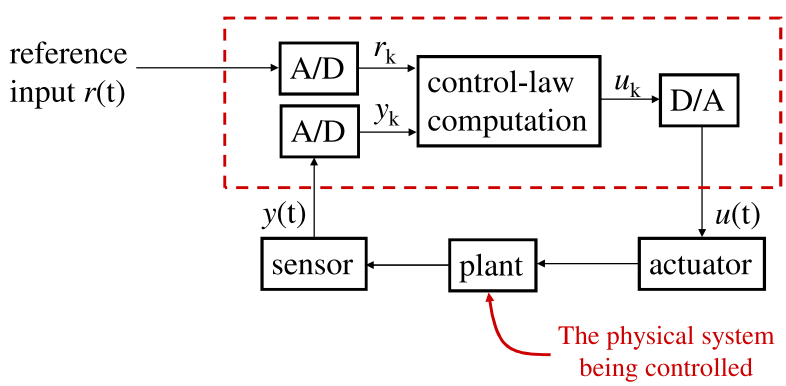
Characteristics of Real-Time Systems

- Event-driven, reactive.
- High cost of failure.
- Concurrency/multiprogramming.
- Stand-alone/continuous operation.
- Reliability/fault-tolerance requirements.
- **Predictable behavior.**

Example Real-Time Applications

Many real-time systems are **control systems**.

Example 1: A simple one-sensor, one-actuator control system.



Simple Control System (cont'd)

Pseudo-code for this system:

```
set timer to interrupt periodically with period T;
at each timer interrupt, do
    do analog-to-digital conversion to get y;
    compute control output u;
    output u and do digital-to-analog conversion;
end do
```

T is called the **sampling period**. *T* is a key design choice. *T*ypical range for *T*: milliseconds to seconds.

Multi-rate Control Systems

More complicated control systems have multiple sensors and actuators and must support control loops of different rates.

Example 2: Helicopter flight controller.

Do the following in *each* 1/180-sec. cycle:
- **validate sensor data and select data source;**
- **if failure, reconfigure the system**

Every sixth cycle do:
- **keyboard input and mode selection;**
- **data normalization and coordinate transformation;**
- **tracking reference update**
- **control laws of the outer pitch-control loop;**
- **control laws of the outer roll-control loop;**
- **control laws of the outer yaw- and collective-control loop**

Every *other* cycle do:
- **control laws of the inner pitch-control loop;**
- **control laws of the inner roll- and collective-control loop**

Compute the control laws of the inner yaw-control loop;

Output commands;

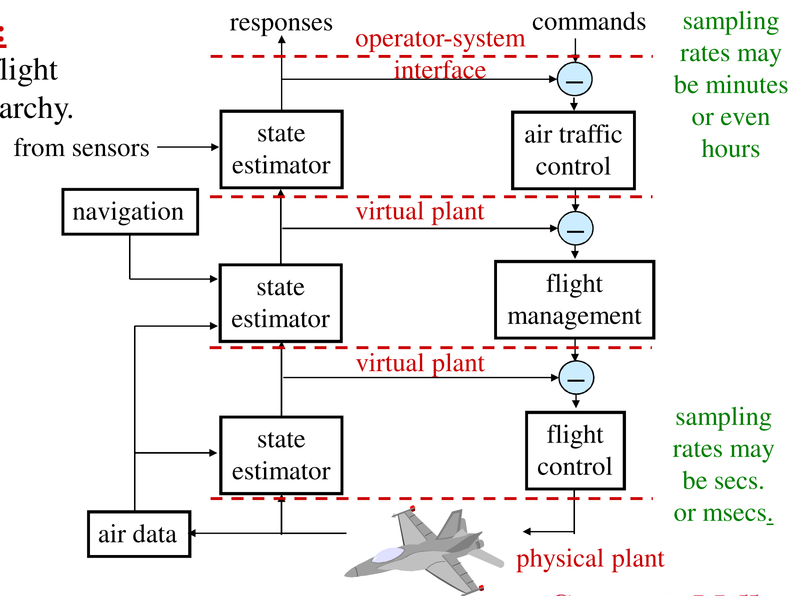
Carry out built-in test;

Wait until beginning of the next cycle

Note: Having only **harmonic** rates simplifies the system.

Hierarchical Control Systems

Example 3:
Air traffic-flight control hierarchy.

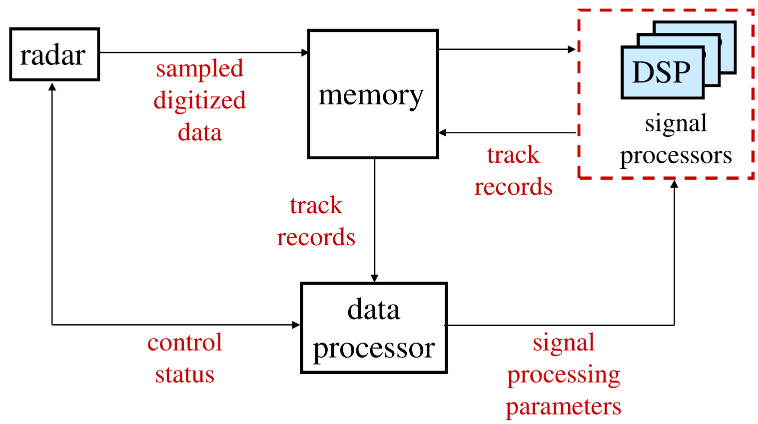


Signal-Processing Systems

Signal-processing systems transform data from one form to another.

- Examples:
  - Digital filtering.
  - Video and voice compression/decompression.
  - Radar signal processing.
- Response times range from a few milliseconds to a few seconds.

Example: Radar System



Other Real-Time Applications

- **Real-time databases.**
  - Transactions must complete by deadlines.
  - **Main dilemma:** Transaction scheduling algorithms and real-time scheduling algorithms often have conflicting goals.
  - Data may be subject to **absolute** and **relative temporal consistency** requirements.
- **Multimedia.**
  - Want to process audio and video frames at steady rates.
    - TV video rate is 30 frames/sec. HDTV is 60 frames/sec.
    - Telephone audio is 16 Kbits/sec. CD audio is 128 Kbits/sec.
  - **Other requirements:** Lip synchronization, low jitter, low end-to-end response times (if interactive).

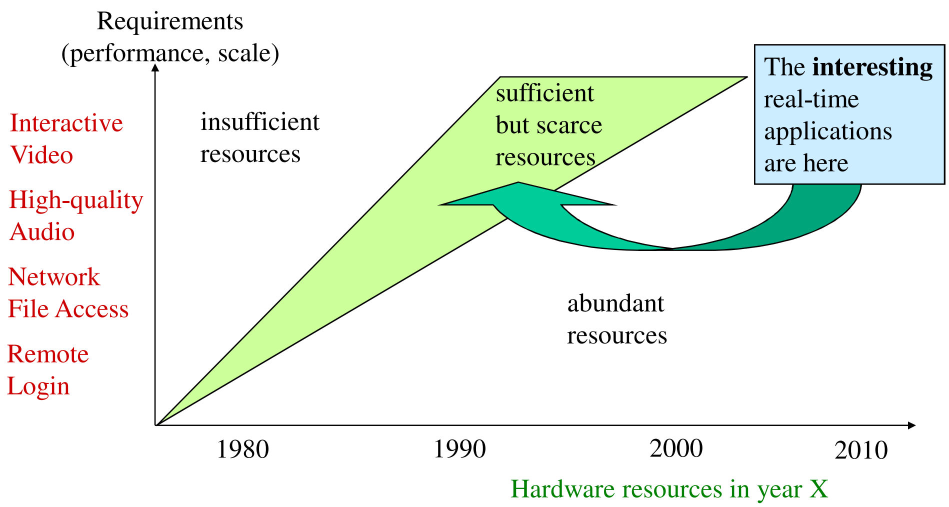
Are *All* Systems Real-Time Systems?

- **Question:** Is a payroll processing system a real-time system?
  - It has a time constraint: Print the pay checks (say) every two weeks.
- Perhaps it is a real-time system in a definitional sense, but it does *not* pay us to view it as such.
- We are interested in systems for which it is not *a priori* obvious how to meet timing constraints.
  - Wide variety of constraints
  - Really tight timing constraints
  - Different levels of criticality

The "Window of Scarcity"

- Resources may be categorized as:
  - **Abundant:** Virtually any system design methodology can be used to realize the timing requirements of the application.
  - **Insufficient:** The application is ahead of the technology curve; no design methodology can be used to realize the timing requirements of the application.
  - **Sufficient but scarce:** It is possible to realize the timing requirements of the application, but careful resource allocation is required.

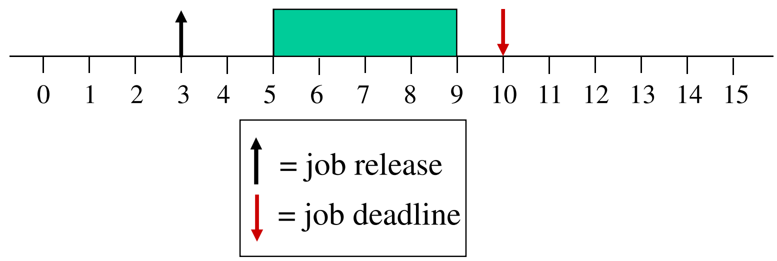
Example: Interactive/Multimedia Applications



Hard vs. *Soft* Real Time

- **Task:** A sequential piece of code.
- **Job:** Instance of a task.
- Jobs require **resources** to execute.
  - **Example resources:** CPU, network, disk, critical section.
  - We will simply call all hardware resources "processors".
- **Release time of a job:** The time instant the job becomes ready to execute.
- **Absolute Deadline of a job:** The time instant by which the job must complete execution.
- **Relative deadline of a job:** "Deadline – Release time".
- **Response time of a job:** "Completion time – Release time".

Example



- Job is released at time 3.
- Its (absolute) deadline is at time 10.
- Its relative deadline is 7.
- Its response time is 6.

Hard Real-Time Systems

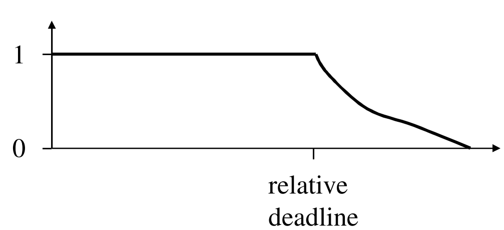
- A **hard deadline** *must* be met.
  - If *any* hard deadline is *ever* missed, then the system is **incorrect**.
  - Requires a means for **validating** that deadlines are met.
- **Hard real-time system:** A real-time system in which all deadlines are hard.
  - We mostly consider hard real-time systems in this course.
- **Examples:** Nuclear power plant control, flight control.

Soft Real-Time Systems

- A **soft deadline** may *occasionally* be missed.
  - **Question:** How to define "occasionally"?
- **Soft real-time system:** A real-time system in which some deadlines are soft.
- **Examples:** Telephone switches, multimedia applications.

Defining "Occasionally"

- **One Approach:** Use probabilistic requirements.
  - For example, 99% of deadlines will be met.
- **Another Approach:** Define a "usefulness" function for each job:



- **Note:** Validation is *much* trickier here.

Reference Model

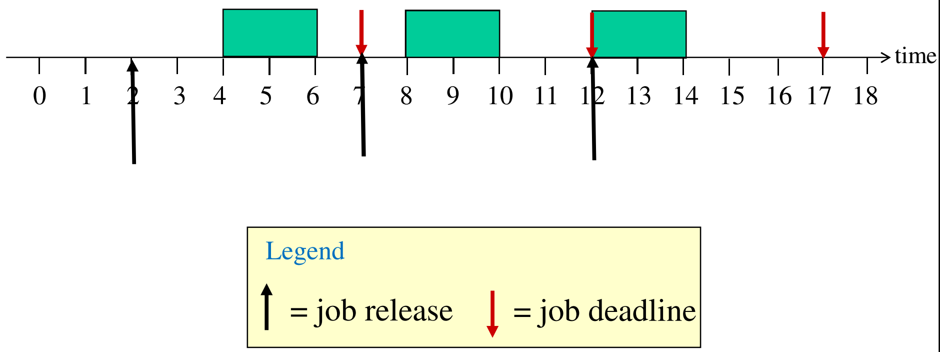
- Each job J_i is characterized by its **release time** r_i, **absolute deadline** d_i, **relative deadline** D_i, and **computation time** C_i.
  - Sometimes a range of release times is specified: $[r_i^-, r_i^+]$. This range is called **release-time jitter**.
- Likewise, sometimes instead of c_i, execution time is specified to range over $[c_i^-, c_i^+]$.
  - **Note:** It can be difficult to get a precise estimate of c_i (more on this later).

Periodic, Sporadic, Aperiodic Tasks

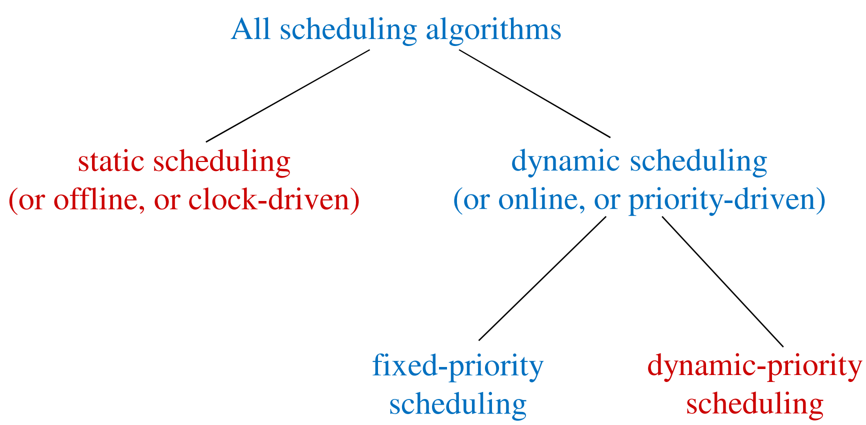
- **Periodic task:**
  - We associate a **period T_i** (as in $1/f$) with each task τ_i.
  - T_i is the interval between job releases of a task τ_i.
- **Sporadic and Aperiodic tasks:** Released at arbitrary times.
  - **Sporadic:** Has a hard deadline.
  - **Aperiodic:** Has no deadline or a soft deadline.

Examples

A periodic task τ_i with $r_i = 2$, $T_i = 5$, $C_i = 2$, $D_i = 5$ may execute like this:



Classification of Scheduling Algorithms



Summary of Lecture So Far

- Real-time Systems
  - characteristics and mis-conceptions
  - the “window of scarcity”
- Example real-time systems
  - simple control systems
  - multi-rate control systems
  - hierarchical control systems
  - signal processing systems
- Terminology
- Scheduling algorithms

Real Time Systems and You

- Embedded real time systems enable us to:
  - manage the vast power generation and distribution networks,
  - control industrial processes for chemicals, fuel, medicine, and manufactured products,
  - control automobiles, ships, trains and airplanes,
  - conduct video conferencing over the Internet and interactive electronic commerce, and
  - send vehicles high into space and deep into the sea to explore new frontiers and to seek new knowledge.

Real-Time Systems

- Timing requirements
  - meeting deadlines
- Periodic and aperiodic tasks
- Shared resources
- Interrupts

What's Important in Real-Time

Metrics for real-time systems differ from that for time-sharing systems.

	Time-Sharing Systems	Real-Time Systems
Capacity	High throughput	Schedulability
Responsiveness	Fast average response	Ensured worst-case response
Overload	Fairness	Stability

- schedulability is the ability of tasks to meet all hard deadlines
- latency is the worst-case system response time to events
- stability in overload means the system meets critical deadlines even if all deadlines cannot be met

Scheduling Policies

- CPU scheduling policy: a rule to select task to run next
  - cyclic executive
  - Rate-monotonic/deadline-monotonic
  - earliest deadline first
  - least laxity first
- Assume preemptive, priority scheduling of tasks
  - analyze effects of non-preemption later

Rate Monotonic Scheduling (RMS)

- Priorities of periodic tasks are based on their rates: the highest rate gets the highest priority.
- Theoretical basis
  - optimal fixed scheduling policy (when deadlines are at end of period)
  - analytic formulas to check schedulability
- Must distinguish between scheduling and analysis
  - Rate-monotonic scheduling forms the basis for rate-monotonic analysis
  - however, we consider later how to analyze systems in which rate-monotonic scheduling is *not* used
  - any scheduling approach may be used, but all real-time systems should be analyzed for timing

Rate Monotonic Analysis (RMA)

- Rate-monotonic analysis is a set of mathematical techniques for analyzing sets of real-time tasks.
- Basic theory applies only to independent, periodic tasks, but has been extended to address
  - priority inversion
  - task interactions
  - aperiodic tasks
- Focus is on RM**A**, not RM**S**

Why Are Deadlines Missed?

- For a given task, consider
  - **preemption**: time waiting for higher priority tasks
  - **execution**: time to do its own work
  - **blocking**: time delayed by lower priority tasks
- The task is schedulable if the sum of its preemption, execution, and blocking is less than its deadline.
- **Focus**: identify the biggest hits among the three and reduce, as needed, to achieve schedulability

Summary

- Real-time goals are:
  - Predictable response,
  - guaranteed deadlines, and
  - stability in overload.
- Any scheduling approach may be used, but all real-time systems should be analyzed for timing.
- Rate-monotonic analysis (RMA)
  - based on rate-monotonic scheduling theory
  - analytic formulas to determine schedulability
  - framework for reasoning about system timing behavior
  - separation of timing and functional concerns
- Provides an engineering basis for designing real-time systems