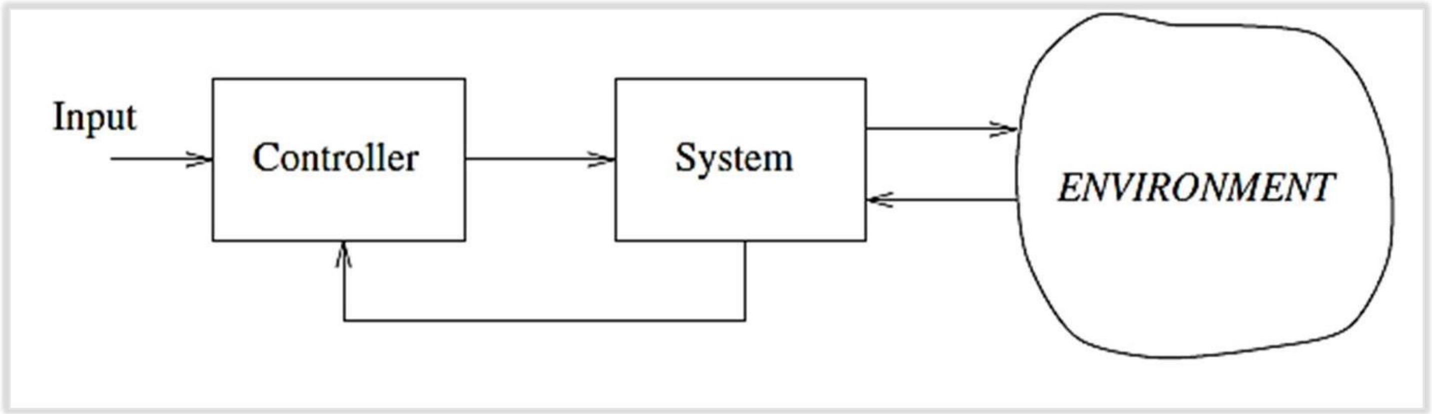
Complex Real-Time Applications, Design Issues (Chapter 11)

- We analyze some complex real-time applications requiring sensory acquisition, control, and actuation of mechanical components.
- We want to introduce specific examples showing how to characterize control applications, so that theory developed for real-time computing and scheduling algorithms can be practically used for increasing the systems' reliability.
- In fact, a precise characterization of the timing constraints for control loops and for sensory acquisition processes is important for guaranteeing a stable behavior of the controlled system, as well as a predictable performance.
- We will also show an example using DIdactic C Kernel code

Components of a Generic Control System



Control application structure:

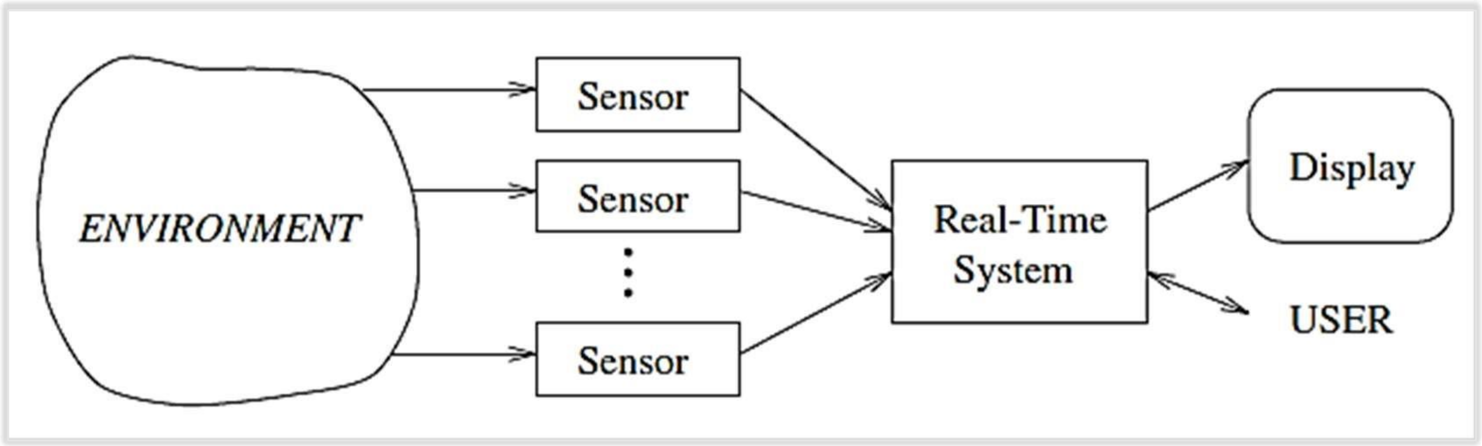
1. System to control – a plant, a car, a robot or other physical device
2. Controller – a computing system that provides the desired control function
3. Environment – all the external conditions in which the system needs to operate

Interactions Between the Controlled System and Environment

- These interactions are bidirectional and occur by means of two peripheral subsystems:
    1. An actuation subsystem – motors, pumps, engines …
    2. A sensory subsystem – microphones, cameras, transducers …
- Based on the types of interactions we can have:
    1. Monitoring systems
    2. Open-loop control systems
    3. Feedback control system

Monitoring Type Systems

Do not modify the environment but only monitor system state, process sensory data, and display the results to the user

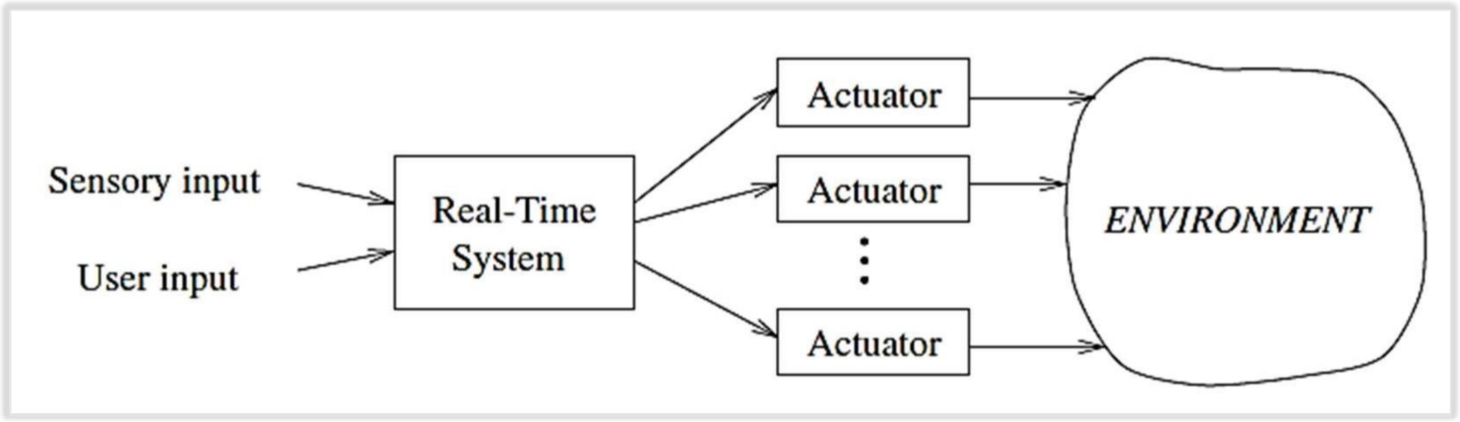


Typical applications: radar tracking, air traffic control, environmental pollution monitoring, surveillance, and alarm systems.

General system features: … periodic acquisitions of multiple sensors, sampling rates can be critical … → using a hard-real-time kernel to implement such systems is important, especially, to guarantee a predictable behavior of the system

Open-Loop Control Systems

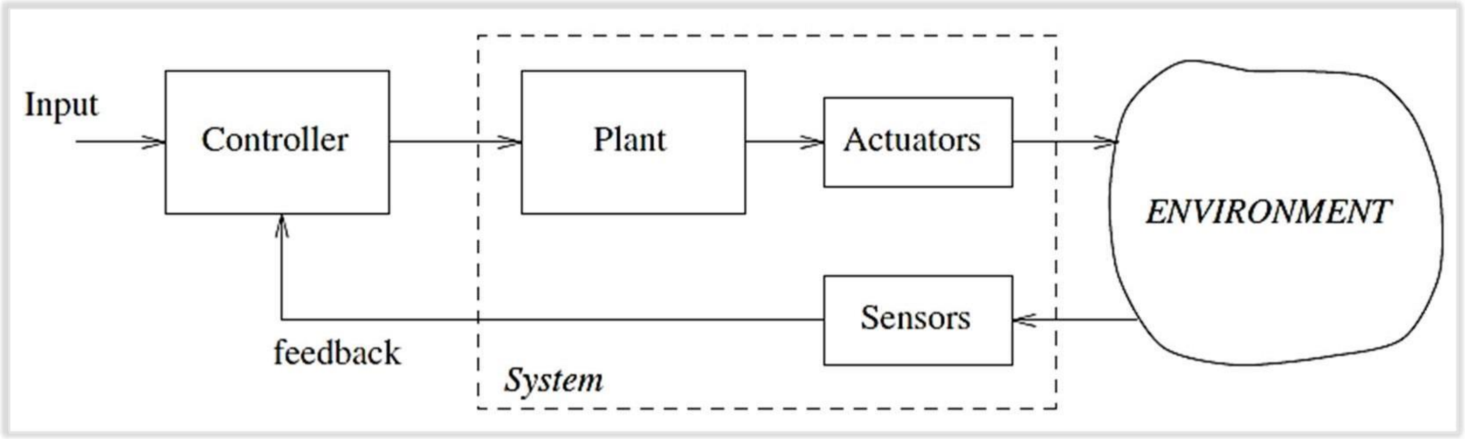
The actions performed by the actuators do not strictly depend on the current state of the environment.
Sensors are used to plan actions, but there is no feedback.



Application example: a robot workstation equipped with a vision subsystem for pick and place operation.
System features: … once the object location is identified the robot motion is fixed … real-time computing is not needed even though the pick and place operation must be completed with a deadline … → fast computing and smart programming may suffice to meet the goal

Closed-Loop Control Systems

Systems that have frequent interactions with the environment in both directions;
The actions produced by the actuators strictly depend on the current sensory information



Example applications: modern “fly-by-wire” aircrafts, the robot workstations with the camera in the feedback loop, …

General system features: … the stability of these systems depends not only on the correctness of the control algorithms but also on the timing constraints imposed on the feedback loops … the consequences of a late action can even be catastrophic …

→ the use of real-time computing is essential for guaranteeing a predictable behavior;

Additional Issues to Be Considered when Developing Critical Real-Time Applications

1. Structuring the application in a number of concurrent tasks, related to the activities to be performed
2. Assigning the proper timing constraints to tasks
3. Using a predictable operating environment able to guarantee that those timing constraints can be satisfied

Time Constraints Definitions

- A system reacts in real-time within an environment if:
  - Its response to any event in that environment is effective, according to some control strategy, while the event is occurring.
  - The results are produced within a specific deadline
- If meeting the deadline is critical for the system operation the task must be treated as a *hard task*
- If meeting the deadline is desirable but no serious consequences happen the task can be treated as *soft task*
- Activities that require regular activation should be handled as *periodic tasks*

Time Constraints Definitions

- Period task … a task whose activation is directly controlled by the kernel in a time-driven fashion, so that it is intrinsically guaranteed to be regular …
- Aperiodic task … a task that is activated by other application tasks or by external events …
  - activation requests may come from the explicit execution of specific system calls or from the arrival of an interrupt associated with the task.
- Sporadic aperiodic task … if the interrupt source is well known and interrupts are generated at a constant rate, or have a minimum interarrival time …
  - its timing constraints can be guaranteed in worst-case assumptions – that is, assuming the maximum activation rate.

Time Constraints Definitions

Phase 1: Identify all application tasks have and time constraints (including periodicity and criticality),

Phase 2: The real-time operating system supporting the application is responsible for guaranteeing that all hard tasks complete within their deadlines.

- Soft and non-real-time tasks should be handled by using a best-effort strategy (or optimal, whenever possible) to reduce (or minimize) their average response times.

Example of Time Constraints Derivation

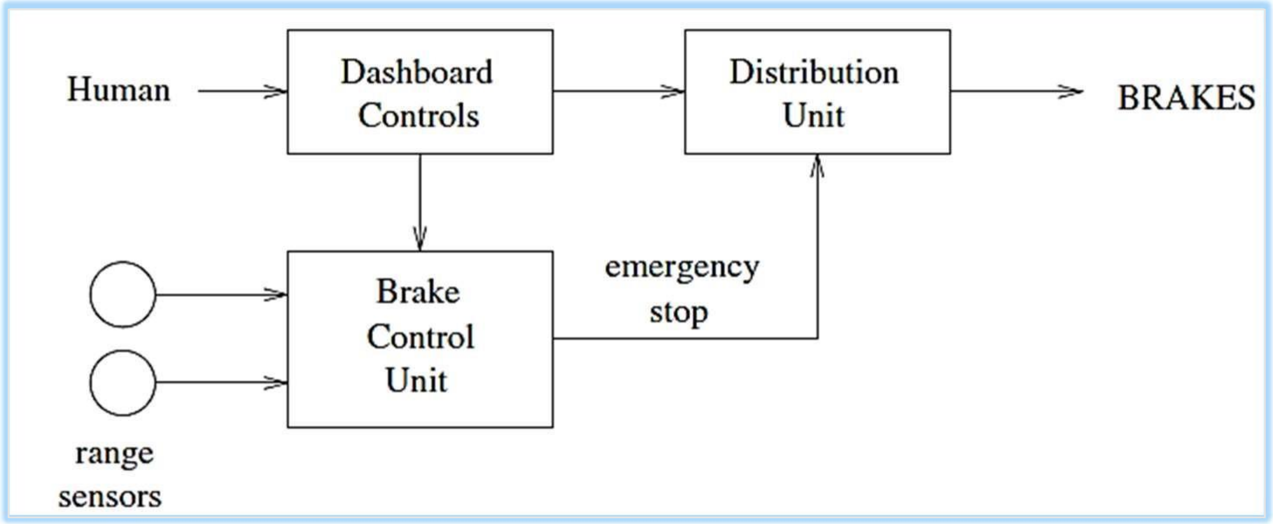
Obstacle avoidance application … a wheel-vehicle equipped with range sensors must operate in an environment running within a maximum given speed …

Examples of such applications:

- A completely autonomous system, such as a robot mobile base,
- A partially autonomous system driven by a human, such as a car or a train having an automatic braking system for stopping motion in emergency situations

Practical Obstacle Avoidance Application Considered – Automatic Braking System

Automatic braking system of a vehicle moving along a straight line



Given the criticality of the braking action, the task of emergency stop must execute periodically on the BCU.

- Let T be its period.

Compute the Worst-Case Latency for Braking

max latency $= t_f - t_0$

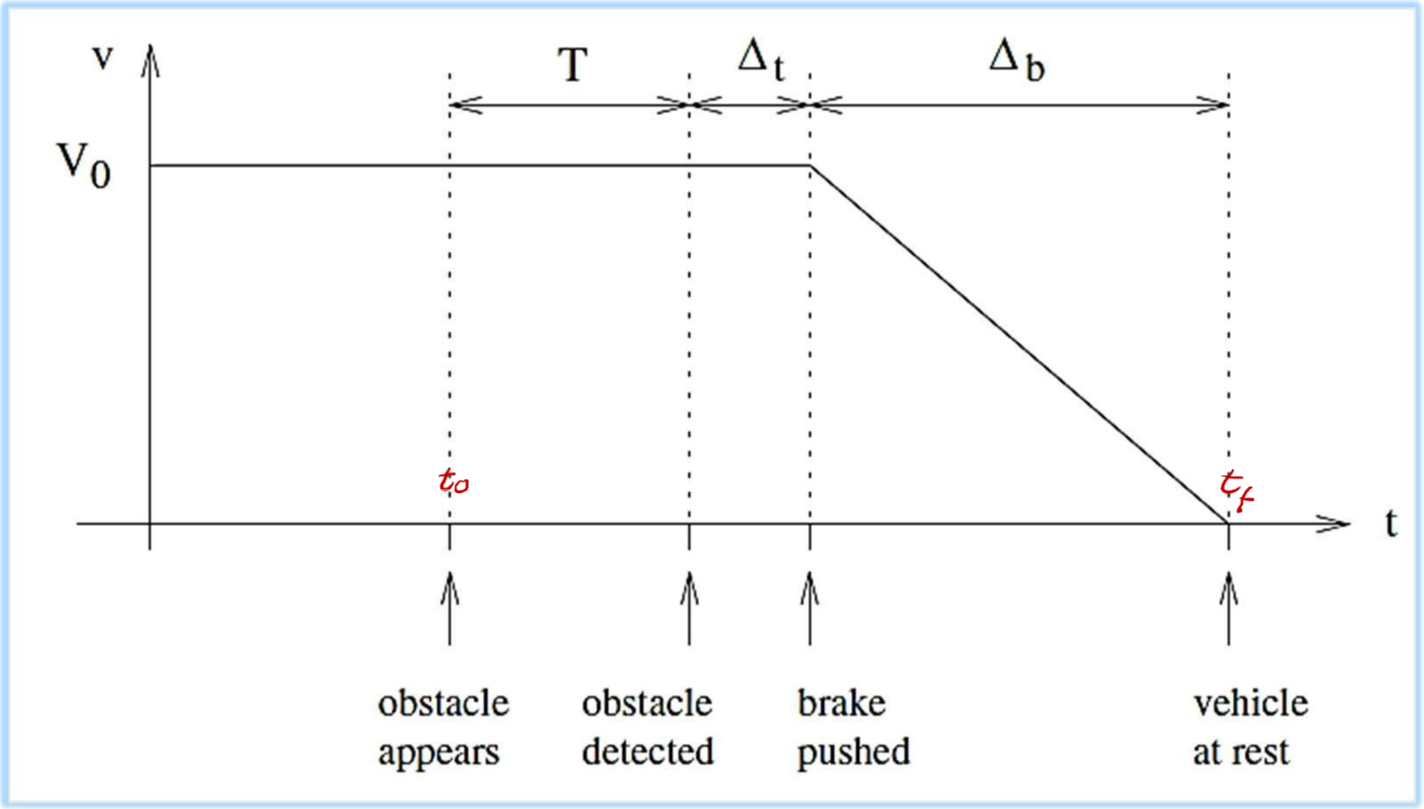
$<$ time to impact

D – defines the distance from obstacle to the vehicle

L – defines the minimum space needed for a complete stop

Then the condition to avoid impact translates to:

$D > L$



T, Time Constraint Calculation Analysis

In class work … see lecture recording and reference book …

Follow the calculations in the reference book: pp. 404-405.
[Hard Real Time Computing Systems book, Chapter 11]

Numerical Example

In class work … see lecture recording and reference book …

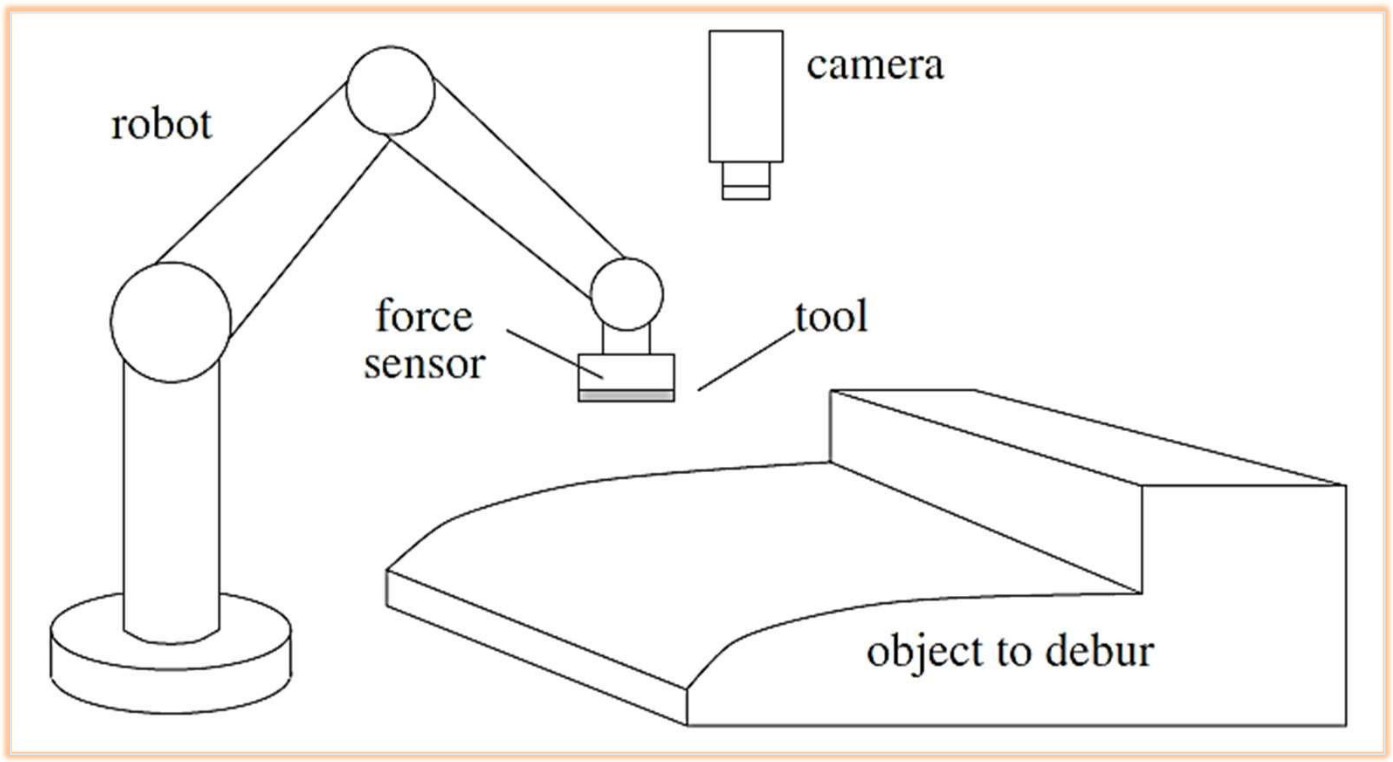
Follow the calculations in the reference book: pp. 404-405.
[Hard Real Time Computing Systems book, Chapter 11]

Example of Time Constraints Derivation

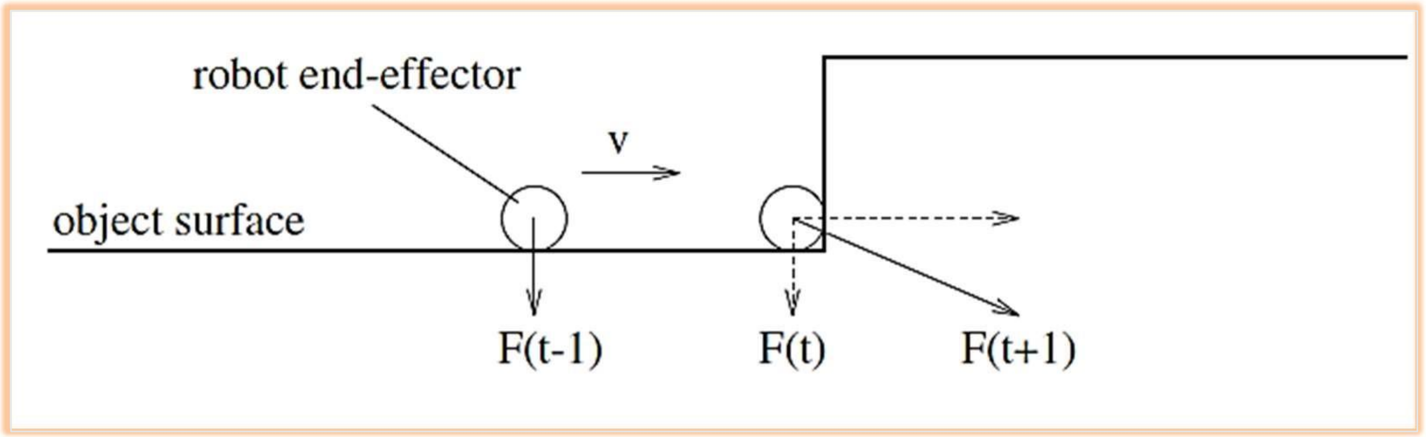
Robot deburring application: consider a robot arm that must polish an object surface with a grinding tool mounted on its wrist. This task is specified as follows:

- Slide the grinding tool on the object surface with a constant speed v, while exerting a constant normal force F that must not exceed a maximum value equal to F_{max}.

Example of a robot deburring workstation



Force on the robot tool during deburring.



T, Q, Time Constraint Calculations, Analysis

In class work … see lecture recordings and reference book …

Follow the calculations in the reference book: pp. 406-407.
[Hard Real Time Computing Systems book, Chapter 11]

Mutilevel Feedback Control

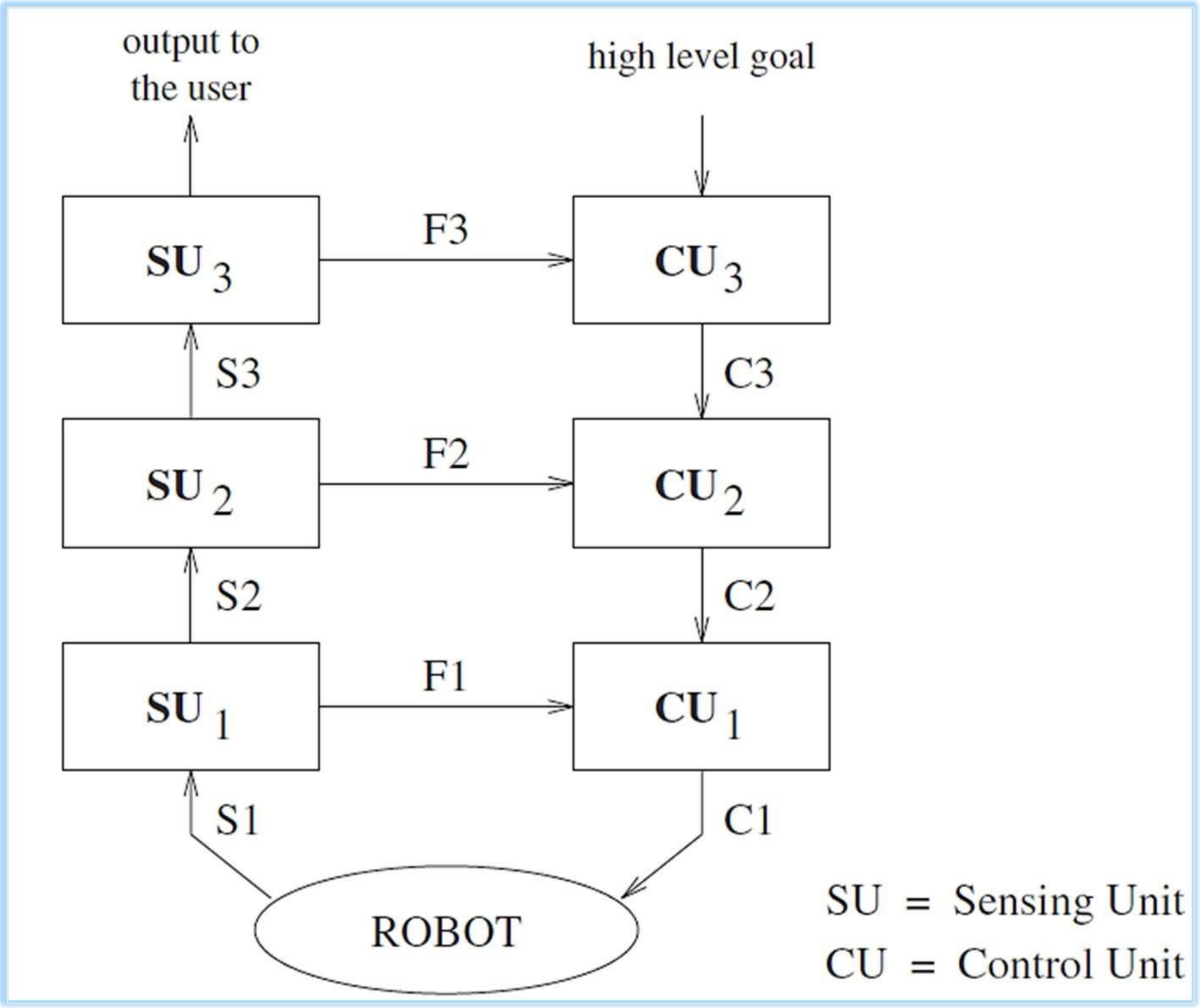
- In complex control applications characterized by nested servo loops, we could choose the frequencies of the control tasks such that to separate the dynamics of the controllers
- This approach will simplify the analysis of the stability and the design of the control law.

Complex, Hierarchical Control Architecture

Each layer of this control hierarchy effectively decomposes an input task into simpler subtasks executed at lower levels.

Control input command is given at the top, which is successively decomposed into subgoals, or subtasks, at each hierarchical level, until at the lowest level output signals drive the actuators.

Sensory data enter this hierarchy at the bottom …



Hierarchical Control Implications on Timing Constraints and Types of Tasks

The feedback enters the control hierarchy at every level.

- At the lowest level, the feedback is almost unprocessed and hence is fast-acting with very short delays,
- At higher levels feedback passes through more and more computational stages and hence it is more sophisticated and slower.

Hierarchical Control Implications on Timing Constraints and Types of Tasks

Task hierarchical implementation has two main implications:

1. Most recent data is needed at each level of control → use asynchronous communication primitives such as cyclic asynchronous buffers (CABs)
    - Blocking of tasks is avoided
    - Solves the problem of communication of tasks with different frequencies
2. When the frequencies of hierarchical nested servo loops differ largely, the design of the control laws are simplified

Hierarchical Control Implications on Timing Constraints and Types of Tasks

Numerical Example:

If at the lowest level a joint position servo is carried out with a period of 1 ms

→a force control loop closed at the middle level can be performed with a period of 10 ms

→ while a vision process running at the higher control level can be executed with a period of 100 ms

Hierarchical Design

Can be used to develop sophisticated control applications requiring sensory integration and multiple feedback loops.

  - Experimented on several robot control applications built on top of a hard-real-time kernel (DIdactic C Kernel) – See references …

- Advantage: simplify the implementation of complex tasks and provide a flexible programming interface between the tasks
- Examples presented:
  - ASSEMBLY: PEG-IN-HOLE INSERTION
  - SURFACE CLEANING
  - OBJECT TACTILE EXPLORATION
  - CATCHING MOVING OBJECTS
  - A ROBOT CONTROL EXAMPLE … example with DIdactic C Kernel code …

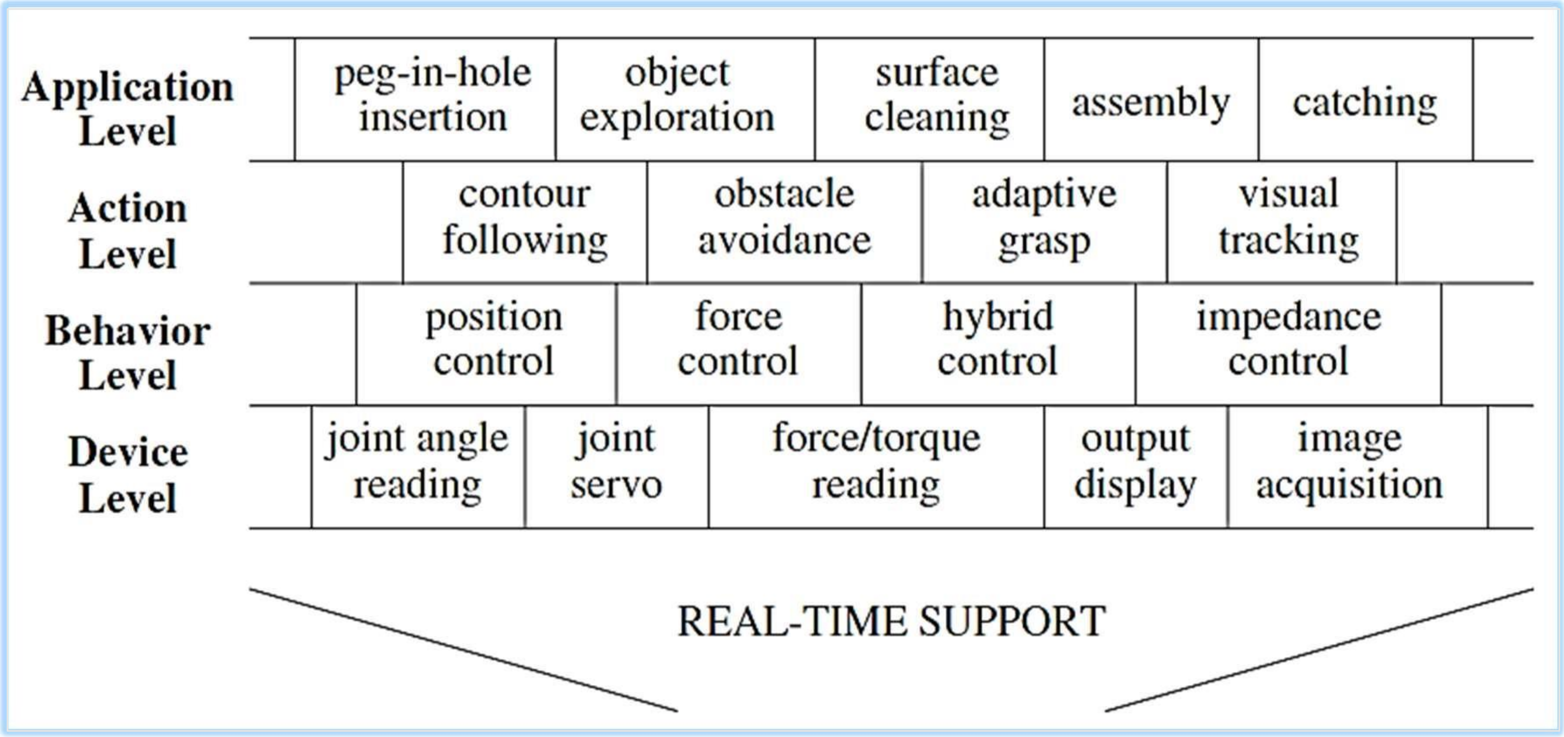
Hierarchical Software Environment for Programming Complex Robotic Applications

Application level: …

Action level: …

Behavior level: …

Device level: …



Examples of Real-Time Robotic Applications

Main feature of the examples:

- The arm trajectory must be continuously replanned based on the current sensory information
  - It cannot be pre-computed off-line to accomplish the goal

OS requirement:

- these applications require a predictable real-time support to guarantee a stable behavior of the robot and meet the specification requirements

ASSEMBLY: PEG-IN-HOLE INSERTION

Typical assembly problem: insertion of a peg into a hole, whose direction is known with some degree of uncertainty.

- Robot operation requirements are:
  - Be actively compliant during the insertion
  - Be highly responsive to force changes.
- Robot control task requirements:
  - The peg-in-hole insertion task can be performed by using a hybrid position/force control scheme
  - Both position and force servo loops must be executed periodically at a proper frequency to ensure stability.
  - Time constraints: … the position loop frequency must be about an order of magnitude higher than the force loop …

SURFACE CLEANING

Problem: cleaning of flat and delicate surface, such as a window glass, implies large arm movements

- Robot operation requirements:
  - The movement the robot end-effector (such as a brush) must be controlled within a plane parallel to the surface to be cleaned
  - The robot end-effector must be pressed against the glass with a desired constant force
- Robot control tasks:
  - The robot is usually equipped with a force sensing device and is controlled in real time to exert a constant force on the glass surface
  - End-effector orientation must be continuously adjusted to be parallel to the glass plane
  - The control operations for both these loops must proceed in parallel and be coordinated by a global planner

OBJECT TACTILE EXPLORATION

Problem: robot should perform autonomous operations working in unknown environments and perform object exploration and recognition. Robot sensors:

- Vision support; tactile and force sensors

- Robot application requirements:
  - Tactile exploration requires the robot to conform to a given geometry
  - Do not produce large changes in the force that the robot applies against the object
  - The robot needs to maintain a desired trajectory and should therefore be position-controlled
- Robot control tasks:
  - Periods of servo loops can be derived as a function of the robot speed, maximum applied forces, and rigidity coefficients, as we will show in the application implementation of this problem

CATCHING MOVING OBJECTS

Problem: catching a moving object including intelligent beings

- Robot application requirements:
  - smart sensing, visual tracking, motion prediction, trajectory planning, and fine sensory-motor coordination
  - sensing, planning, and control must be performed in real time while the target is moving
- Control task requirements:
  - Strict time constraints for the tasks described above derive from the maximum velocity and acceleration assumed for the moving object

Robot Control Example with Program Implementation: OBJECT TACTILE EXPLORATION

Application: implementation of a robot control system capable of exploring unknown objects by integrating visual and tactile information.

Robot requirements:

- Exert desired forces on the object surface and follow its contour by means of visual feedback

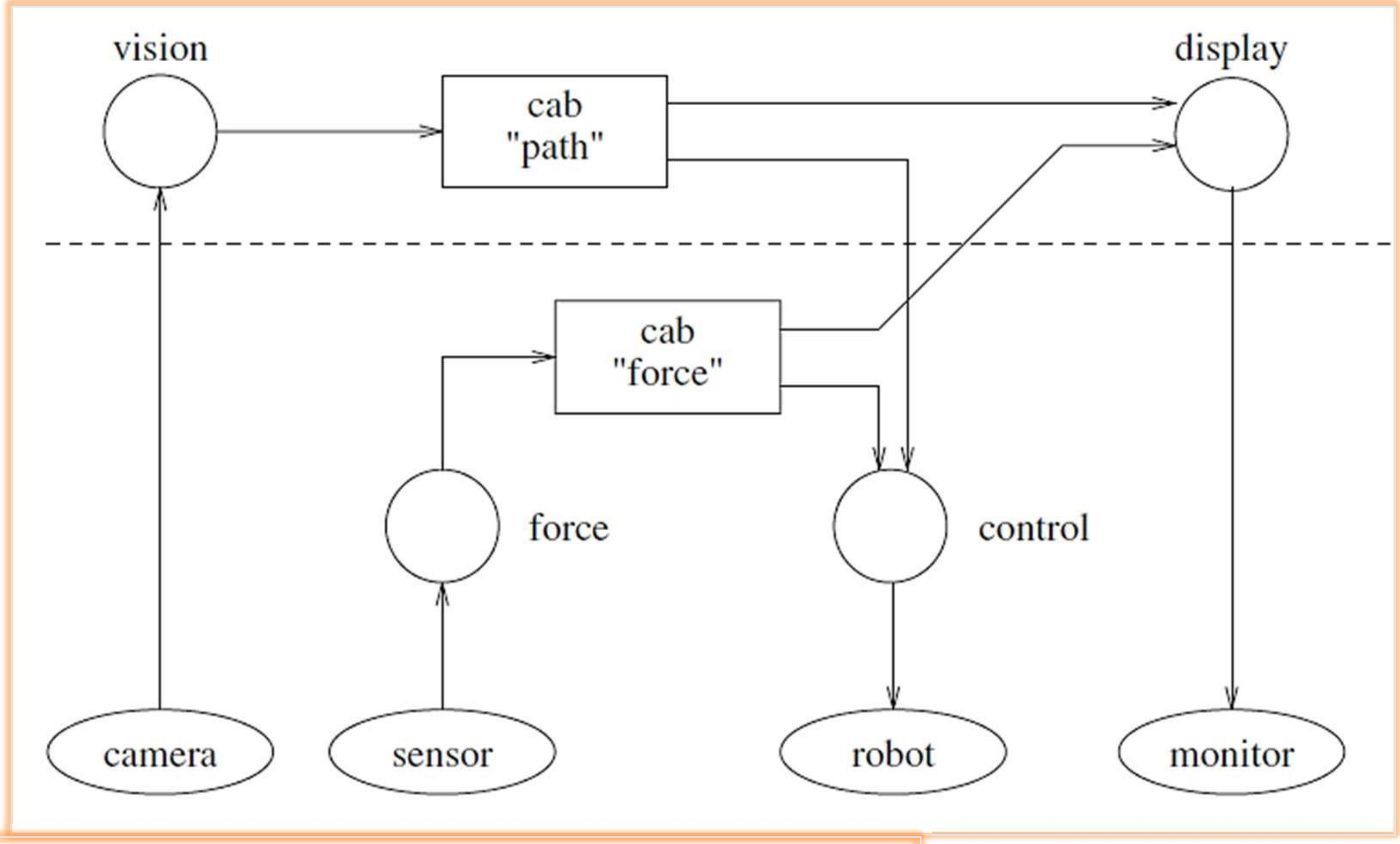
System implementation:

- Robot arm system equipped with a wrist force/torque sensor and a CCD camera
- The software control architecture is organized as two servo loops
  - The inner loop is dedicated to image acquisition, force reading, and robot control,
  - The outer loop performs scene analysis and surface reconstruction

Application: Software Control Architecture

Organized as two servo loops

1. inner loop is dedicated to image acquisition, force reading, and robot control
2. outer loop performs scene analysis and surface reconstruction



processes are indicated by circles; CABs are indicated by rectangles.

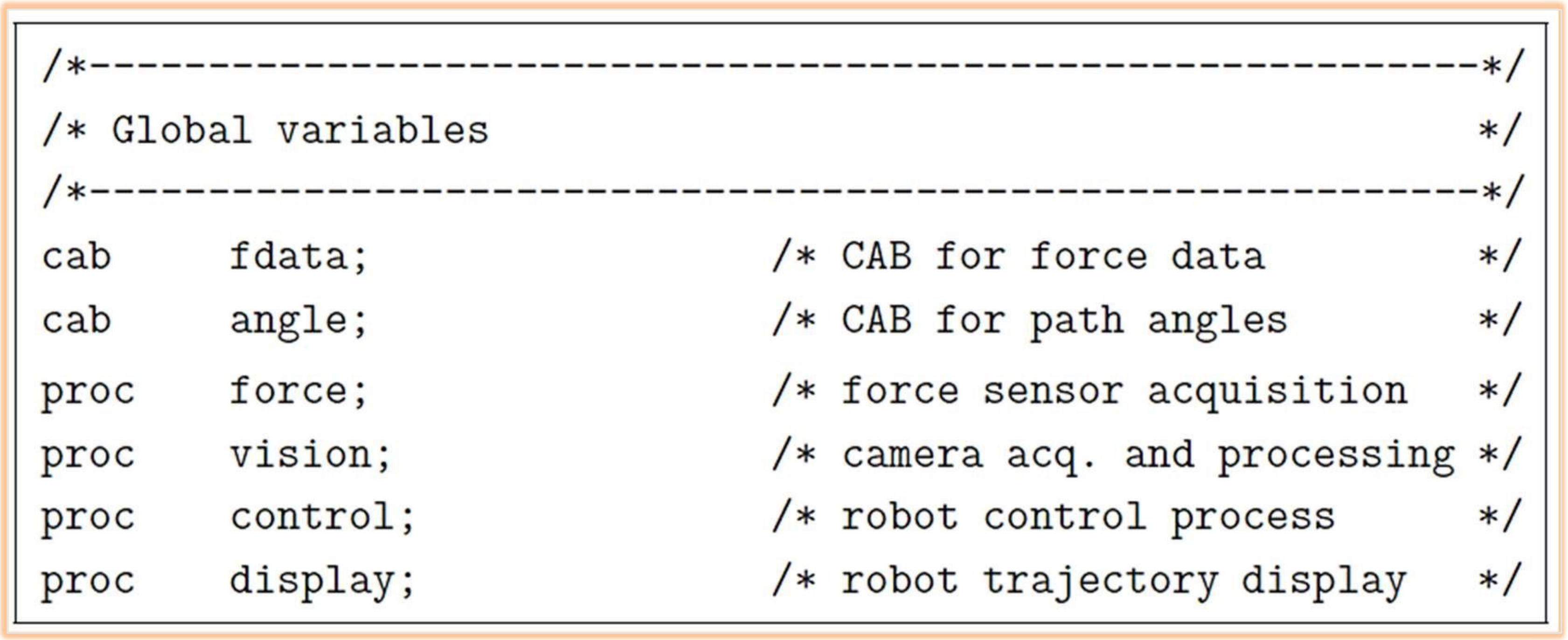
Application: Software Processes

1. Sensory acquisition process – hard real-time periodic task
   - Reads the force/torque sensor and puts data in CAB named force
   - Task period T = 20 ms
2. Visual process – hard real-time periodic task
   - Reads the image memory filled by the camera frame grabber
   - Computes the next exploring direction based on user defined strategy
   - Task period T = 80 ms
3. Robot control process – hard real-time periodic task –> T = 28 ms
   - Computes the cartesian set points for a robot arm controller
   - A hybrid position/force control scheme is used to move the robot end-effector along a direction tangential to the object surface and to apply forces normal to the surface
4. Representation process – soft periodic task → T = 60 ms
   - Reconstructs the object surface based on the current force/torque data and on the exploring direction

Source Code Implementation: global constants

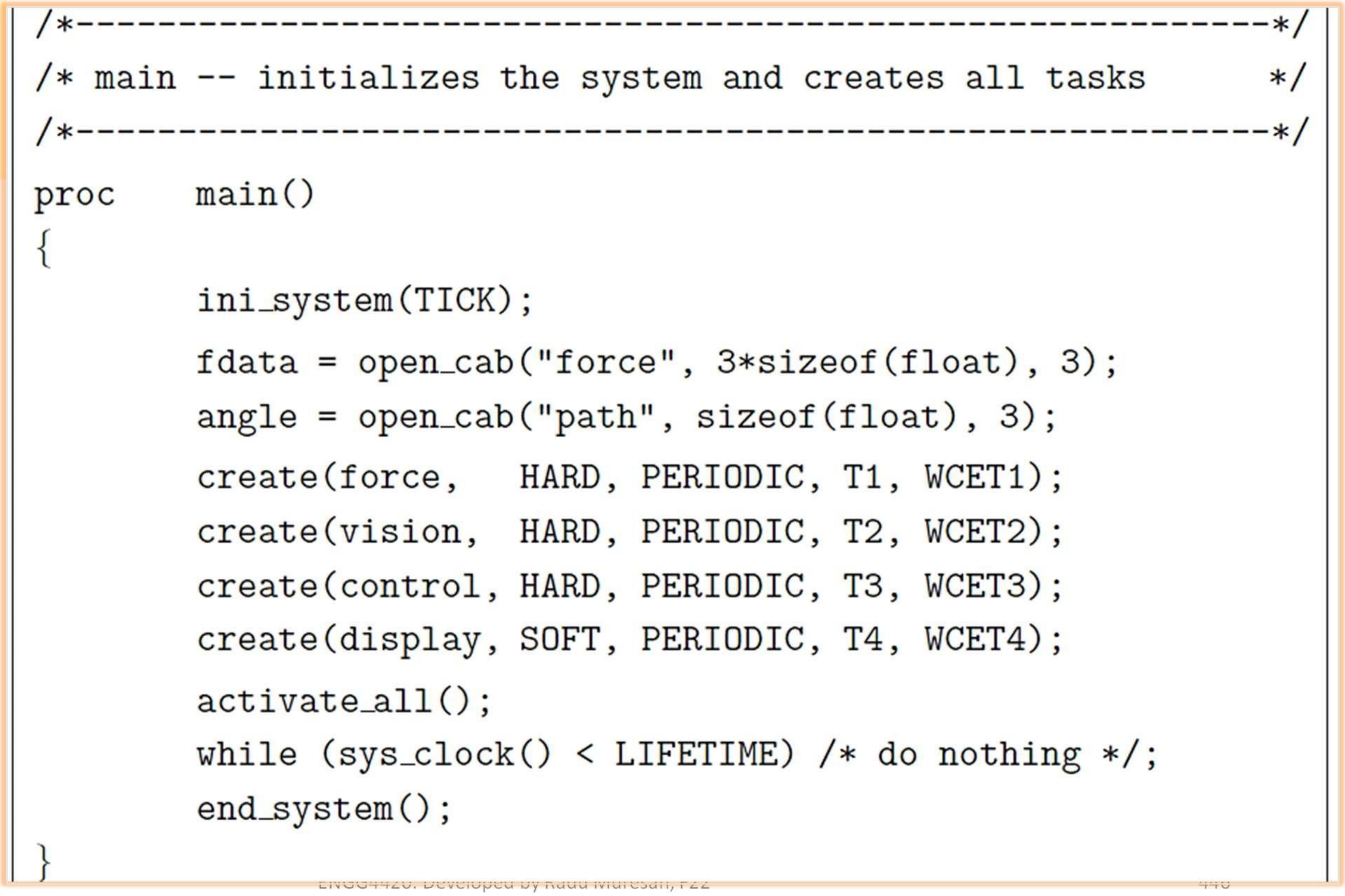
```
/*-----------------------------------------------------------*/
/* Global constants                                          */
/*-----------------------------------------------------------*/
#include "dick.h"                /* DICK header file          */
#define TICK        1.0          /* system tick (1 ms)        */
#define T1          20.0         /* period for force   (20 ms) */
#define T2          80.0         /* period for vision  (80 ms) */
#define T3          28.0         /* period for control (28 ms) */
#define T4          60.0         /* period for display (60 ms) */
#define WCET1       0.300        /* exec-time for force   (ms) */
#define WCET2       4.780        /* exec-time for vision  (ms) */
#define WCET3       1.183        /* exec-time for control (ms) */
#define WCET4       2.230        /* exec-time for display (ms) */
```

Source Code Implementation: global variables



```
/*-----------------------------------------------------------*/
/* Global variables                                          */
/*-----------------------------------------------------------*/
cab     fdata;                 /* CAB for force data           */
cab     angle;                 /* CAB for path angles          */
proc    force;                 /* force sensor acquisition     */
proc    vision;                /* camera acq. and processing   */
proc    control;               /* robot control process        */
proc    display;               /* robot trajectory display     */
```

Source Code Implementation
main process



```
/*----------------------------------------------------------*/
/* main -- initializes the system and creates all tasks     */
/*----------------------------------------------------------*/

proc    main()
{
        ini_system(TICK);
        fdata = open_cab("force", 3*sizeof(float), 3);
        angle = open_cab("path", sizeof(float), 3);
        create(force,   HARD, PERIODIC, T1, WCET1);
        create(vision,  HARD, PERIODIC, T2, WCET2);
        create(control, HARD, PERIODIC, T3, WCET3);
        create(display, SOFT, PERIODIC, T4, WCET4);
        activate_all();
        while (sys_clock() < LIFETIME) /* do nothing */;
        end_system();
}
```

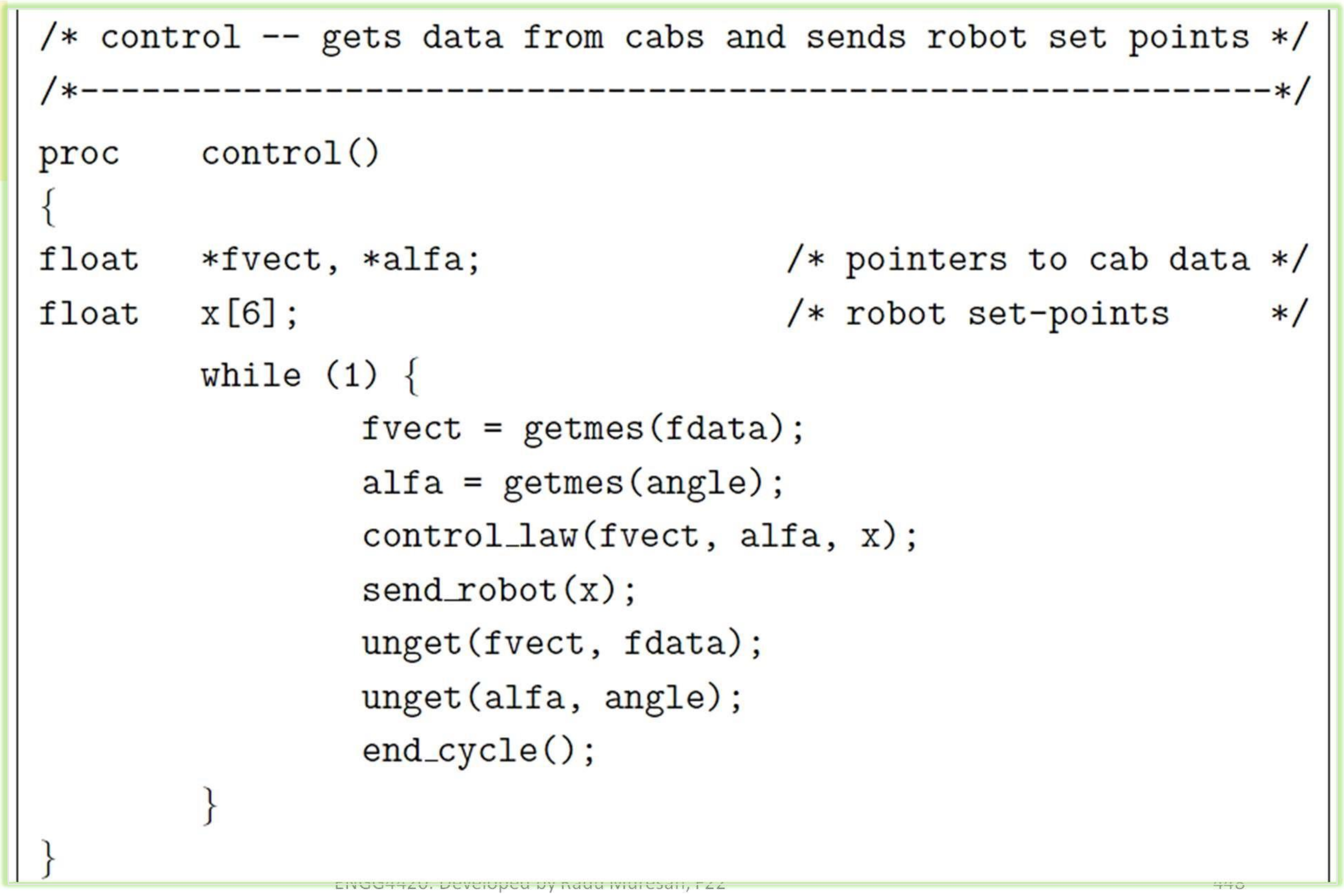
Source Code Implementation: force task

```
/*-----------------------------------------------------------*/
/* force -- reads the force sensor and puts data in a cab    */
/*-----------------------------------------------------------*/

proc    force()
{
float   *fvect;                        /* pointer to cab data */

        while (1) {
                fvect = reserve(fdata);
                read_force_sensor(fvect);
                putmes(fvect, fdata);
                end_cycle();
        }
}
```

Source Code Implementation
control task

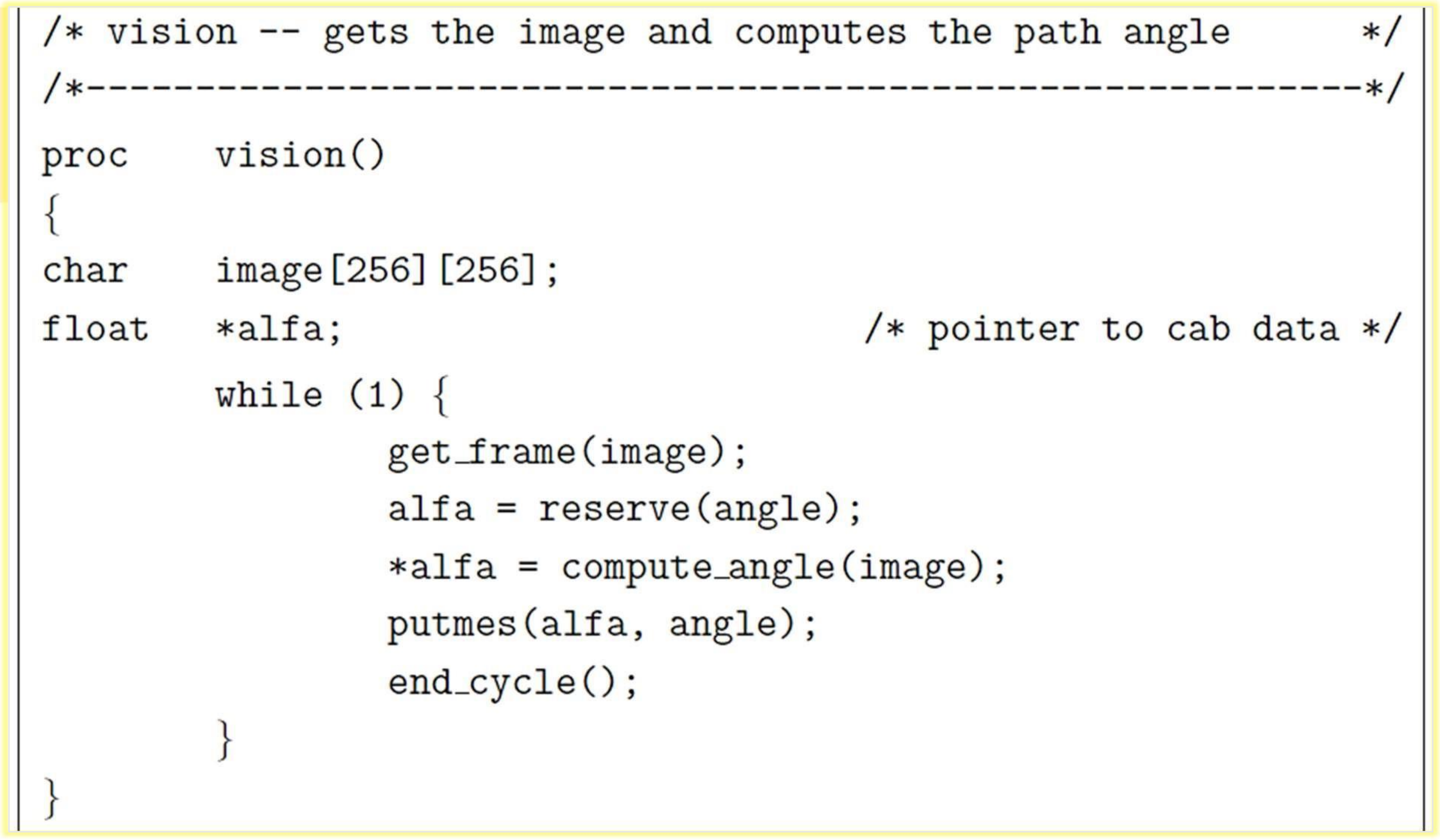


```
/* control -- gets data from cabs and sends robot set points */
/*----------------------------------------------------------*/

proc    control()
{
float   *fvect, *alfa;                  /* pointers to cab data */
float   x[6];                           /* robot set-points     */

        while (1) {
                fvect = getmes(fdata);
                alfa = getmes(angle);
                control_law(fvect, alfa, x);
                send_robot(x);
                unget(fvect, fdata);
                unget(alfa, angle);
                end_cycle();
        }
}
```

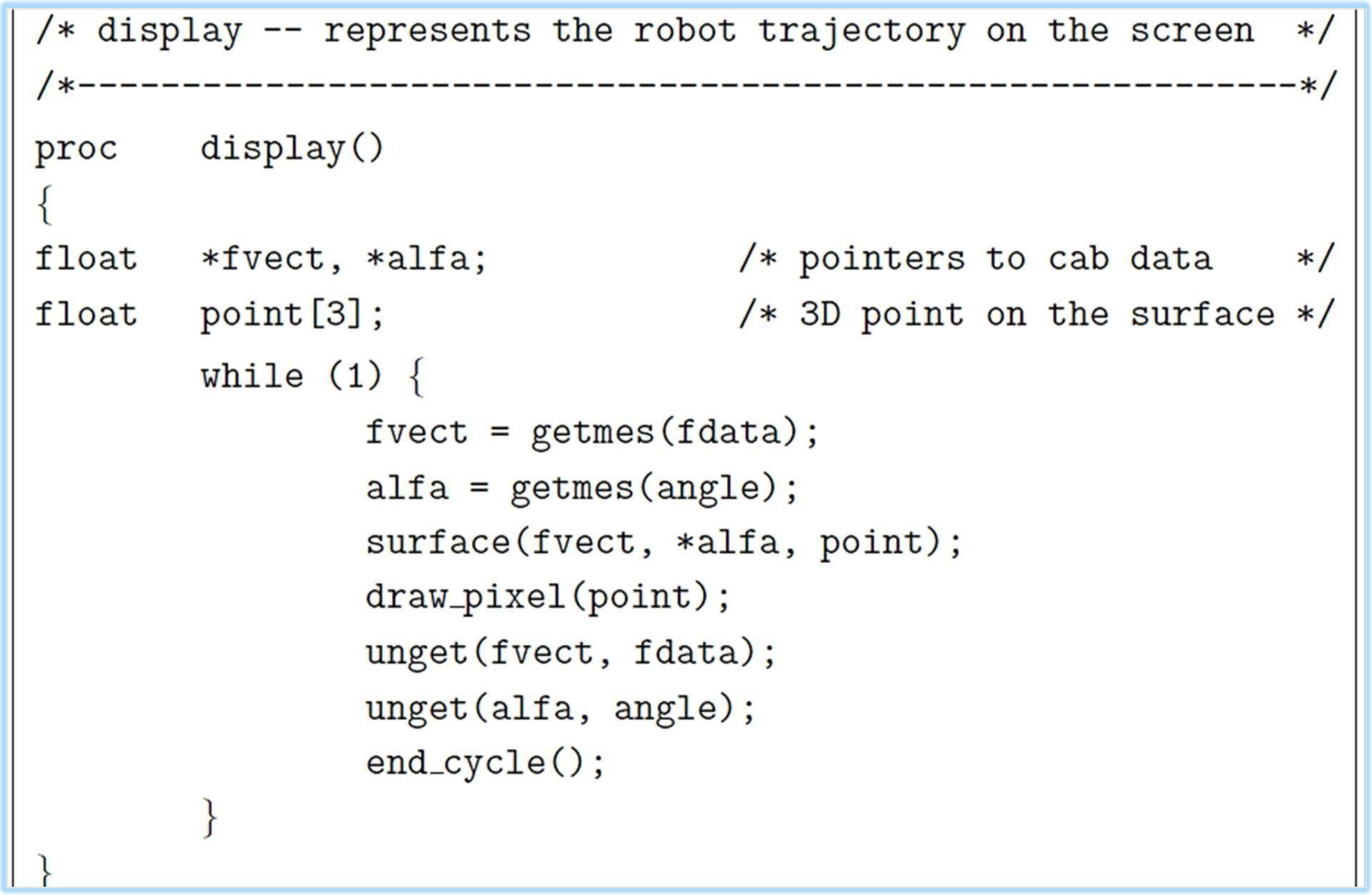
Source Code Implementation vision task



```
/* vision -- gets the image and computes the path angle      */
/*-----------------------------------------------------------*/

proc    vision()
{
char    image[256][256];
float   *alfa;                          /* pointer to cab data */
        while (1) {
                get_frame(image);
                alfa = reserve(angle);
                *alfa = compute_angle(image);
                putmes(alfa, angle);
                end_cycle();
        }
}
```

Source Code Implementation
display task



```
/* display -- represents the robot trajectory on the screen  */
/*-----------------------------------------------------------*/
proc    display()
{
float   *fvect, *alfa;                /* pointers to cab data    */
float   point[3];                     /* 3D point on the surface */
        while (1) {
                fvect = getmes(fdata);
                alfa = getmes(angle);
                surface(fvect, *alfa, point);
                draw_pixel(point);
                unget(fvect, fdata);
                unget(alfa, angle);
                end_cycle();
        }
}
```

Hard-Real Time Scheduling Algorithms

- Reference: Hard Real Time Computing Systems by Giorgio Buttazzo
- Topics covered are:
  1. Basic Concepts (Chapter 2)
  2. Aperiodic Task Scheduling (Chapter 3)
  3. Periodic Task Scheduling (Chapter 4)
  4. Fixed-Priority Servers (Chapter 5)
  5. Dynamic Priority Servers (tentative)
  6. Resource Access Protocols (tentative)

Basic Concepts (Chapter 2)

Sections covered:

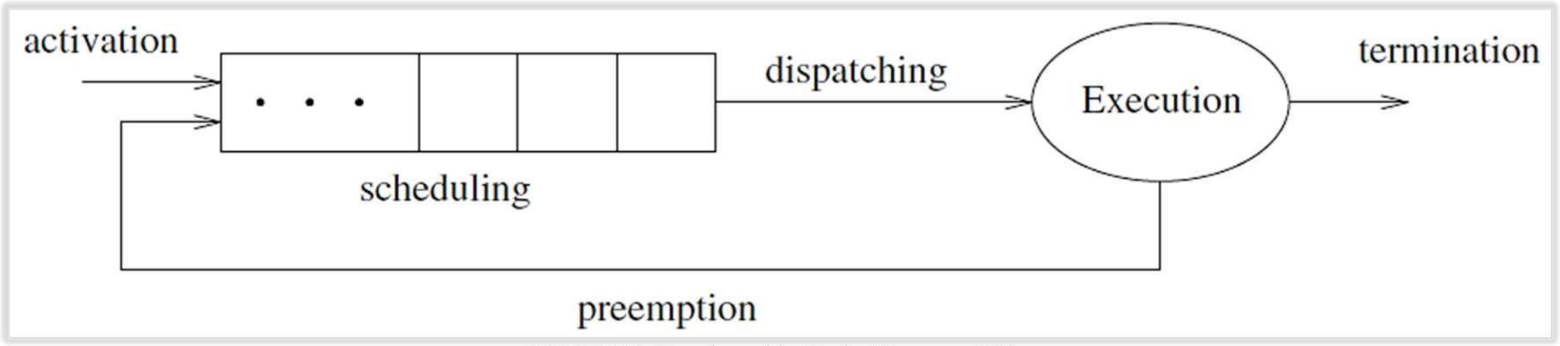
- Task sets, task set schedule
- Types of task constraints
  - Precendence constraints
  - Resource constraints
- Definition of scheduling problem
- Scheduling anomalies (reading section)

Scheduling Concepts

- The term process is used as synonym of **task** and **thread**
- A process is a computation that is executed by the CPU in a **sequential fashion**
- Scheduling policy is a predefined criterion to execute **concurrent tasks on a single processor**
- A scheduling algorithm contains the set of rules that determine **what tasks** will be executed at any time, when a single processor must execute as set of concurrent tasks
- **The specific operation** of allocating the CPU to a task selected by the scheduling algorithm is called dispatching
- A task that can potentially execute on the processor, independently on its actual availability, is called an active task. A task waiting for the processor is called a ready task, whereas the task in execution is called a running task.
- All the ready tasks waiting for the processor are kept in a queue, called ready queue.

Preemption

- Preemption relates to an operating system that allows **dynamic task activation**
  - The operation of **suspending the running task** for a higher priority task to run and inserting the low priority task into the ready queue is called preemption
- In dynamic realtime systems, preemption is important for three reasons:
  1. **Tasks performing exception handling** may need to preempt existing tasks
  2. When tasks have **different levels of criticality**
  3. Preemptive scheduling typically allows **higher efficiency of processor utilization**
- As a negative effect of preemption is that pre-emption **destroys program locality** and **introduces runtime overhead** that inflates task execution times



What is a Task Schedule?

- Given a set of tasks, J = {J1, …, Jn}, a schedule is an assignment of tasks to the processor, so that each task is executed until completion
- FORMAL DEFINITION: a schedule can be defined as a function σ as follows:

$\sigma : \mathbb{R}^+ \to \mathbb{N}$ such that $\forall t \in \mathbb{R}^+$, $\exists t_1, t_2$ such that $t \in [t_1, t_2)$ and $\forall t' \in [t_1, t_2)$ we have $\sigma(t) = \sigma(t')$.

In other words, σ(t) is an integer step function and σ(t) = k, with k > 0, means that task J_k is executing at time t, while σ(t) = 0 means that the CPU is idle.

Example of Schedule with 3 Tasks

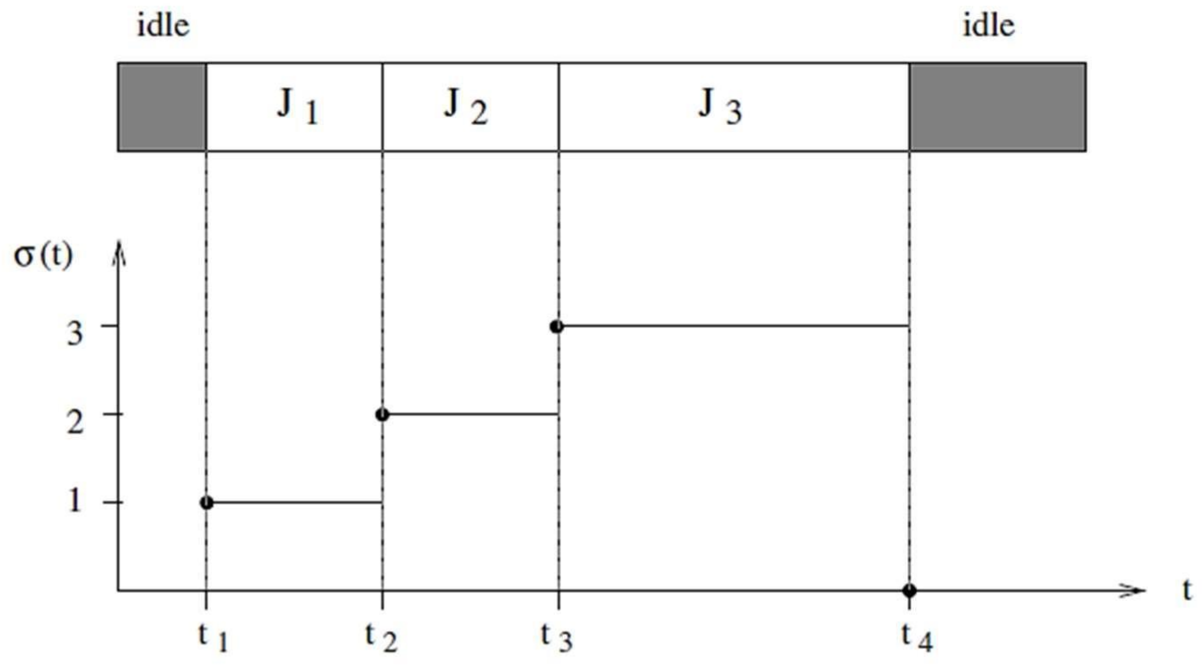
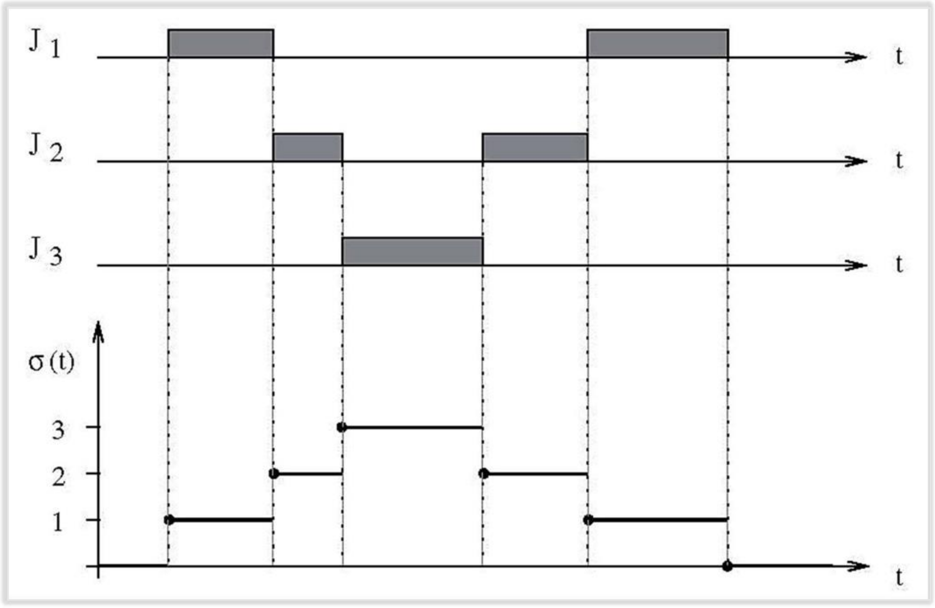


Figure 2.2 Schedule obtained by executing three tasks J_1, J_2, and J_3.

- At times t_1, t_2, t_3, and t_4, the processor performs a *context switch*.
- Each interval $[t_i, t_{i+1})$ in which $\sigma(t)$ is constant is called *time slice*. Interval $[x, y)$ identifies all values of t such that $x \leq t < y$.

Preemptive Schedule

- A preemptive schedule is a schedule in which the running task can be **arbitrarily suspended** at any time, to assign the CPU to another task according to a **predefined scheduling policy**
- A schedule is feasible if all tasks can be completed according to a set of specified constraints
- A set of tasks is schedulable if there exists at least one algorithm that can produce a feasible schedule



Types of Task Constraints

There are three classes of typical constraints that can be specified on real-time tasks:

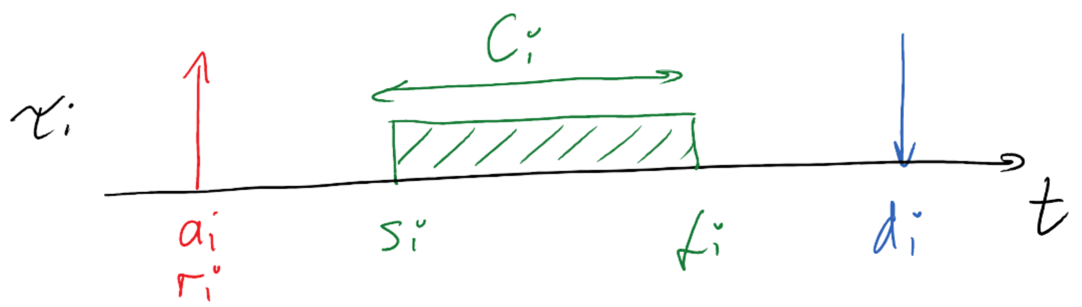
1. Timing constraints
2. Precedence constraints
3. Mutual exclusion constraints on shared resources

Timing Constraints

Timing constraints are defined on real-time systems tasks in order to characterize computational activities and achieve desired behavior of tasks

- Deadline (a typical time constraint) -- represents the time **before** which a process should complete its execution without causing any damage to the system
  - *Relative deadline* – deadline is specified with respect to **task arrival time**
  - *Absolute deadline* – deadline is specified with respect to **time zero**
- Depending on the consequences of a missed deadline real-time tasks can be:
  - Hard: A real-time task is said to be hard if *missing its deadline* may cause **catastrophic consequences** on the system under control
  - Firm: A real-time task is said to be firm if *missing its deadline* **does not cause any damage** to the system, but the output has no value
  - Soft: A real-time task is said to be soft if *missing its deadline* has **still some utility for the system**, although causing **a performance degradation**

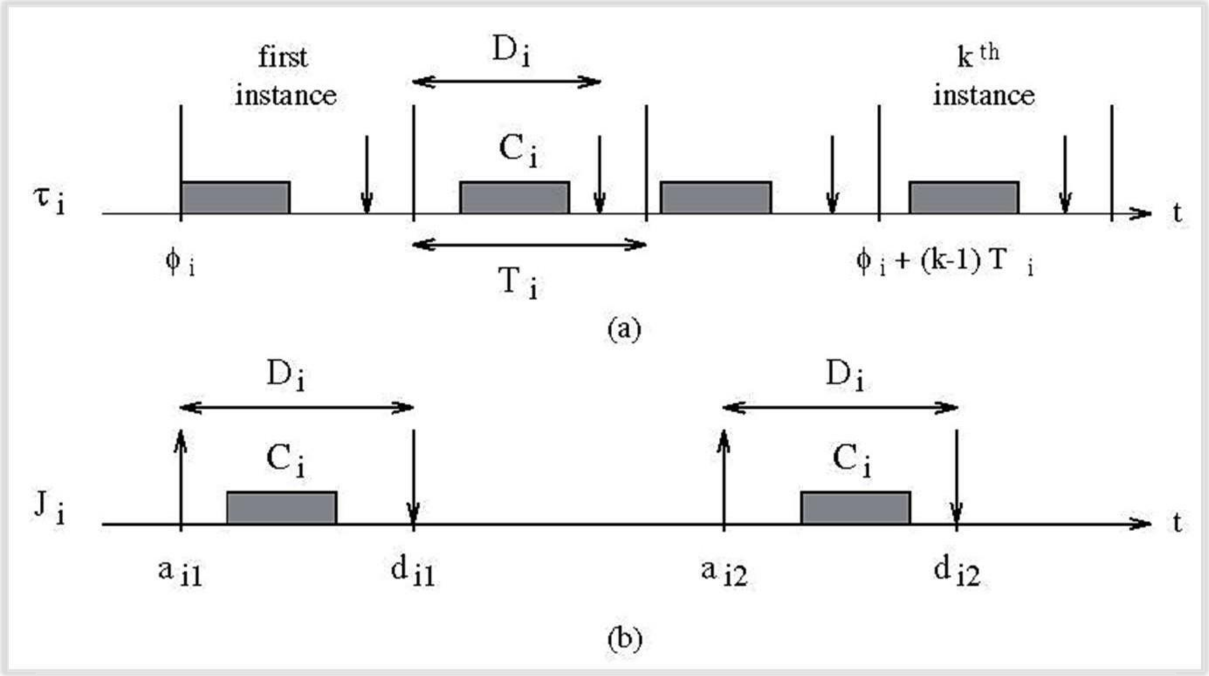
Typical Parameters of a Real-Time Task τ_i



- Arrival time, or request time or release time: *ai, ri* …
- Computation time: *Ci* …
- Absolute deadline: *di* …
- Relative deadline: *Di = di − ri*
- Start time: *si* …
- Finishing time: *fi* …
- Response time: *Ri = fi − ri*
- Criticality: *parameter* …
- Value: *vi* …
- Lateness: *Li = fi −di*
- Tardiness or exceeding time: *Ei = max(0, Li)*
- Laxity: *Xi = di − ai − Ci*

Tasks' Timing Characteristic Based on Their Regularity of Activation

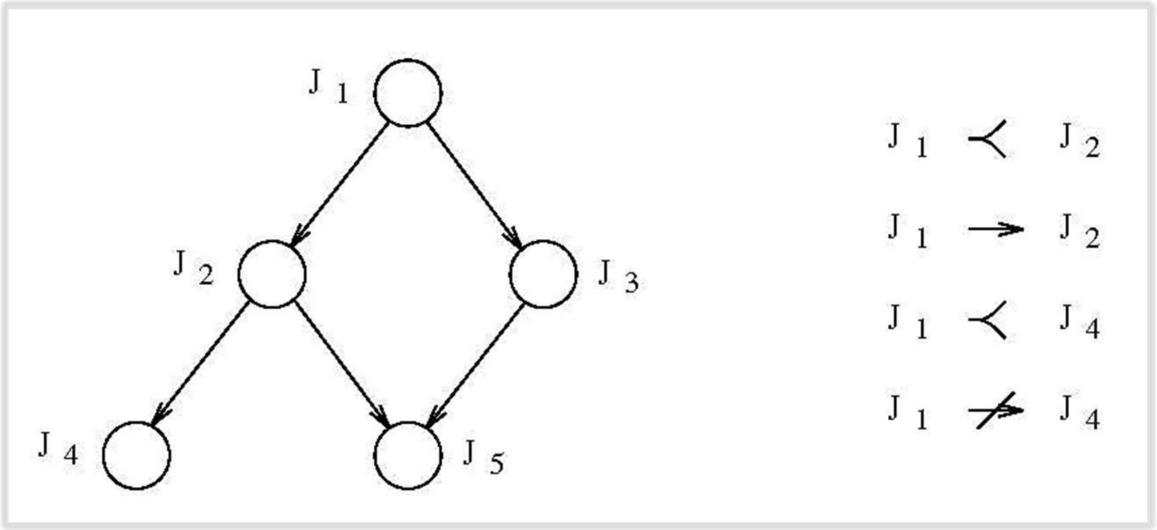
- Periodic tasks (τi) – an infinite sequence of identical activities called instances or jobs
  - Phase φi; computation time Ci; period Ti; relative deadline Di
- Aperiodic tasks (Ji) – an infinite sequence of irregularly arriving identical jobs
- Sporadic tasks – an aperiodic tasks where the consecutive jobs are separated by a minimum inter-arrival time



For a task τi, the activation time of the first periodic instance ($\tau_{i,1}$) is called phase φi

Precedence Constraints

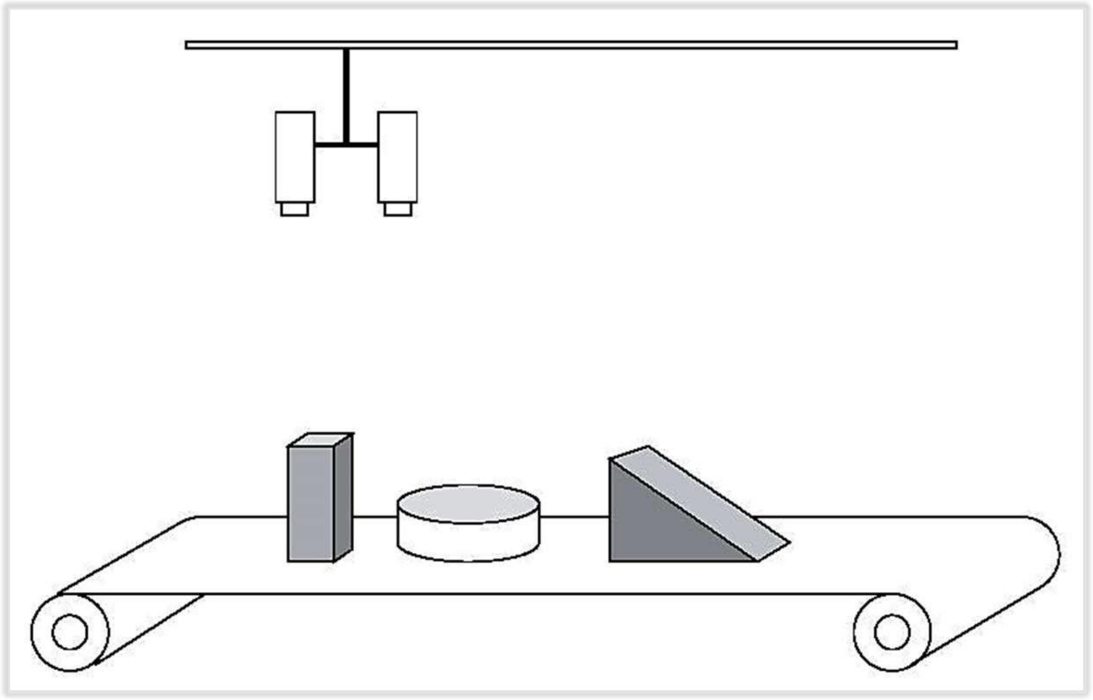
- Precedence constraints are described by a directed acyclic graph G
  - Nodes: tasks
  - Arrows: precedence relations
- Notations:
  - predecessor ($\prec$);
  - immediate predecessor ($\rightarrow$)
- In a graph G structure
  - Beginning tasks …
  - Ending tasks …



Example to Show How Precedence Graphs are Applied

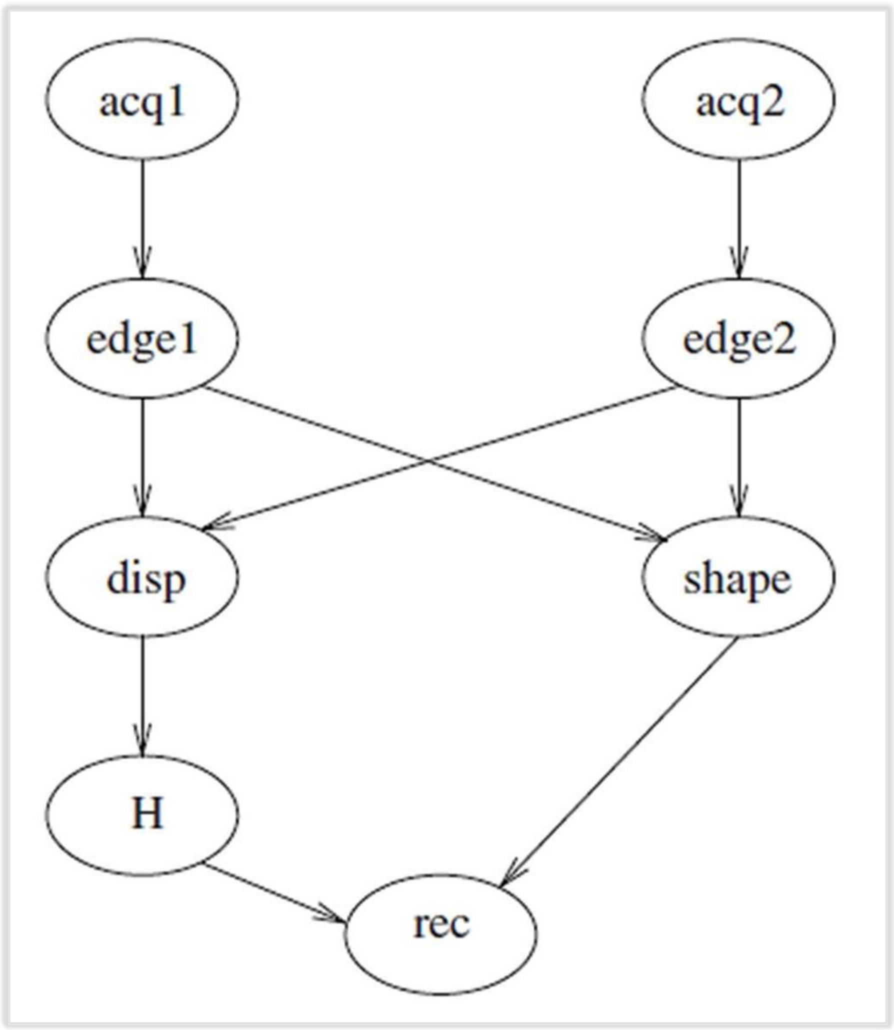
Application: several objects are moving on a conveyor belt. The objects must be recognized and classified using a stereo vision system that contains two cameras

- Object recognition process -- by integrating the two-dimensional features of the top view of the objects with the height information extracted by the pixel disparity on the two images



Tasks and Precedence Task Graph for Example

- acq1, 2: image acquisition tasks
- edge1, 2: Low-level image processing tasks (digital filtering)
- shape: extracts 2-dimensional features
- disp: computes pixel disparities from 2 images
- H: task that computes object height from disp task results
- rec: task that performs the final object recognition

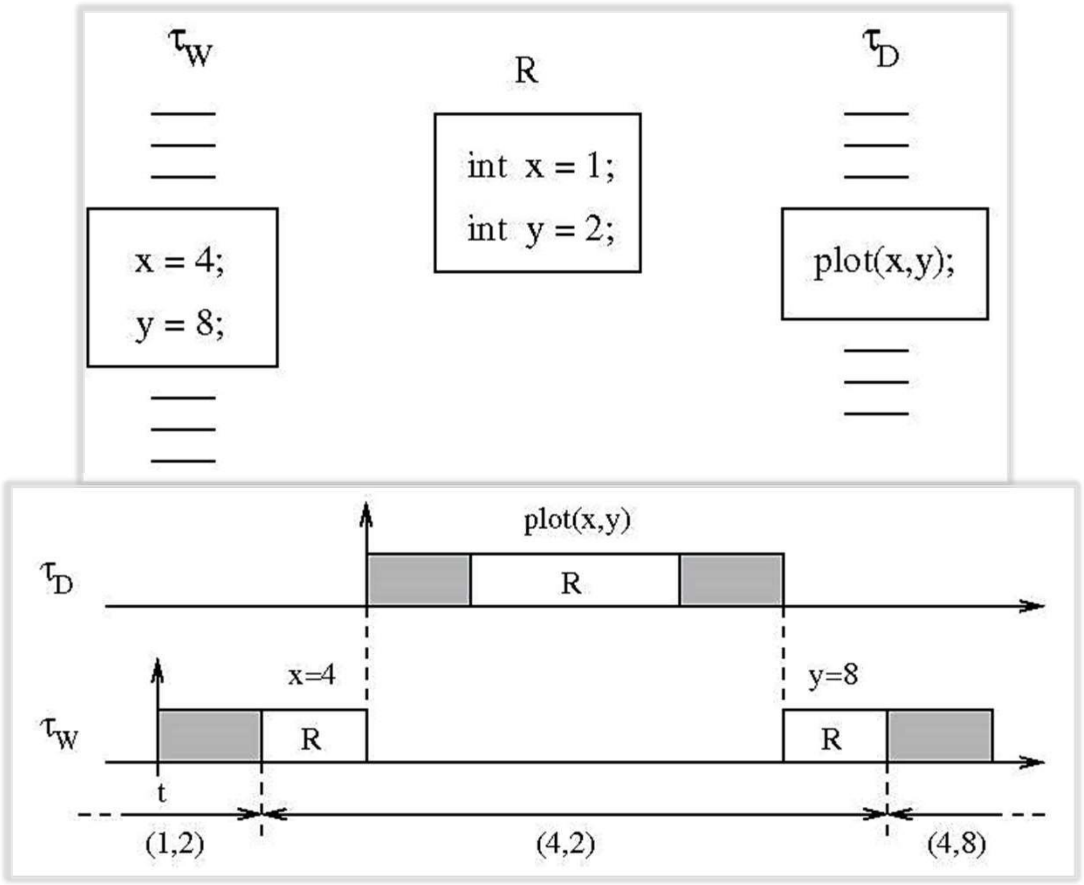


Resource Constraints

- From a process point of view, a resource is any software structure that can be used by the process to advance execution
- Examples of resources:
  - Data structure; a set of variables; a main memory area; a file; a piece of program; a set of registers; a peripheral device
- The resource can be private or shared resource
- Data consistency is important for shared resources;
  - Done through mutual exclusion; in this case a resource R is called a mutually exclusive resource

Data Consistency in Shared Resources

- R must be a mutually exclusive resource
- Critical section: code executed under mutual exclusion constraints
- The importance of mutual exclusion is shown in this example:
  - Two tasks cooperate to track a moving object
  - Prio(τ_W) < Prio(τ_D)

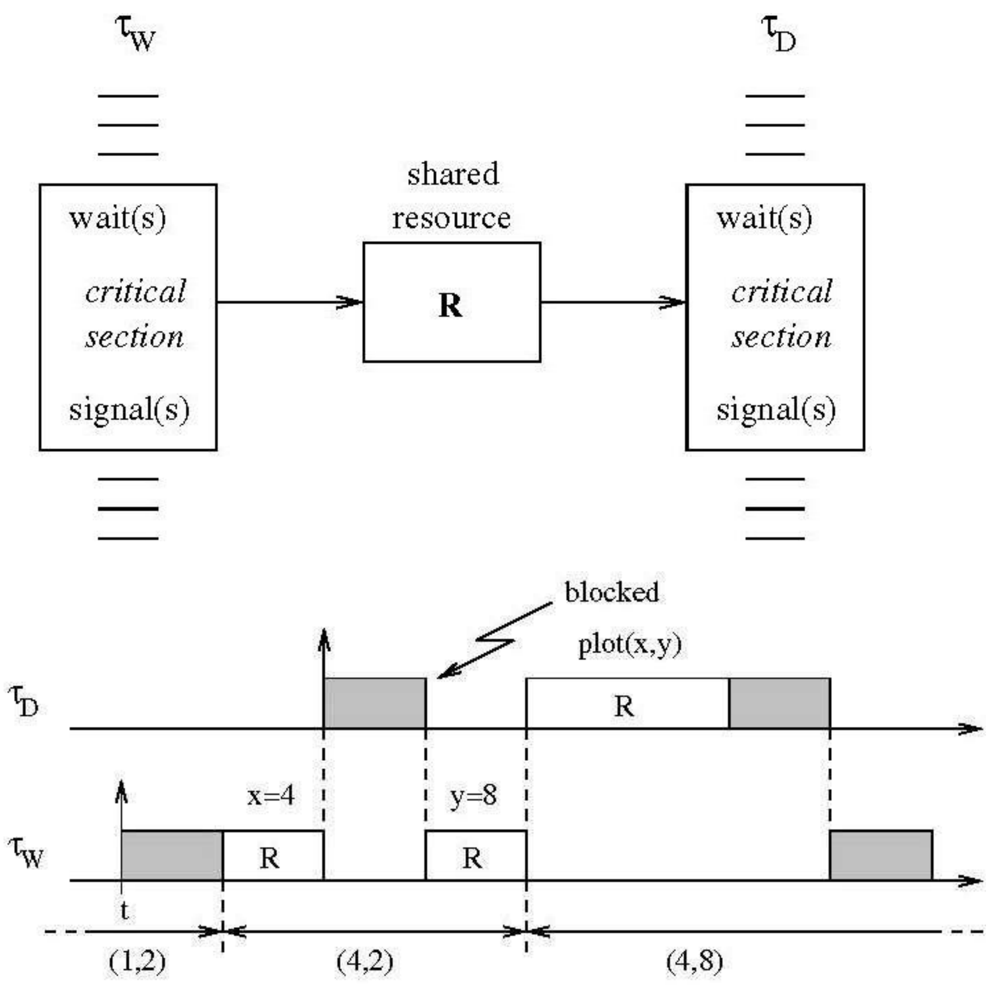


Use of Semaphore for Mutual Exclusion

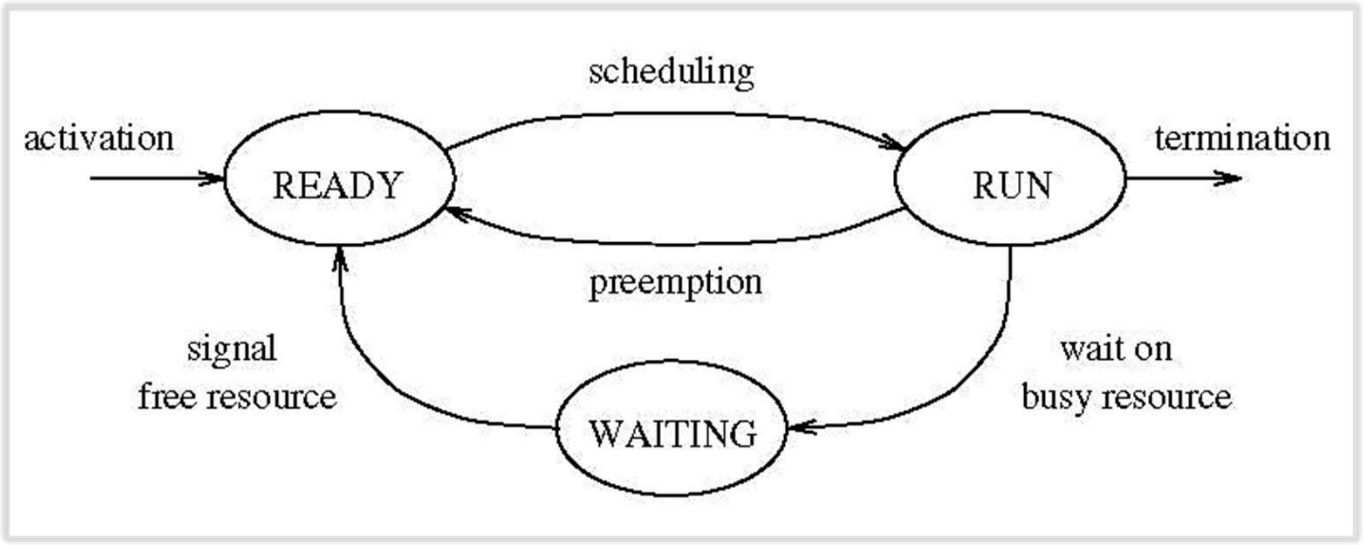
Solution to resource sharing: binary semaphore

Functions performed on a semaphore s to access a critical section are:

wait(s); signal(s)



State Transition Diagram, Due to Resource Constraints



- A task waiting for an exclusive resource is said to be blocked on that resource
- All tasks blocked on the same resource are kept in a queue associated with the semaphore protecting the resource