Real-Time Programming

Lecture 3

Farhang Nemati

Spring 2016

Repetition

- Hard Real-Time, Soft Real-Time Systems
- Real-Time Task
- Properties of Real-Time Systems
  - Complexity, Reliability, Concurrency, Interactive with physical world, Schedulabilty, Fault Tolerant, Predictability
- POSIX Standard
  - `pthread_create(), pthread_join(), pthread_cancel(), pthread_attr_t, pthread_getschedparam(), pthread_setschedparam(), sleep(), nanosleep().`
- Timing facilities that a RTOS has to provide
  - Access to Clock, i.e., system time
  - Clock Rate
  - Possibility for delaying the execution of a task for an interval of time

POSIX Clocks and Timing Facilities

- Clock: A software entity to keep time, in seconds and nanoseconds
  - Is updated by system-clock ticks
  - It can be system-wide or local for a process (program)
  - CLOCK_REALTIME is a system-wide real-time clock. It represents seconds and nanoseconds since Epoch (1 January 1970 00:00:00).
- #include <time.h>
- Get clock resolution

```
clock_getres(clockid_t, struct timespec *)
```

Example:

```
struct timespec ts;
clock_getres(CLOCK_REALTIME, &ts);
// seconds in ts.tv_sec and nanoseconds in ts.tv_nsec
```

POSIX Clocks and Timing Facilities

- Get the current clock time

```
clock_gettime(clockid_t, struct timespec *)
```

Example:

```
struct timespec ts;
clock_gettime(CLOCK_REALTIME, &ts);
// seconds in ts. tv_sec and nanoseconds in ts.tv_nsec
```

POSIX Clocks and Timing Facilities

- Set the clock time

```
int clock_settime(clockid_t, const struct timespec *)
```

Example:

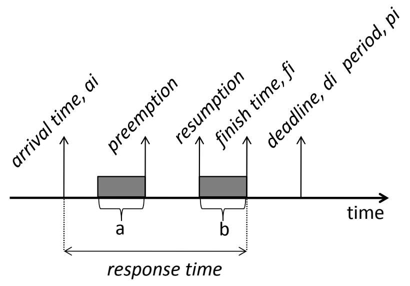
```
struct timespec ts;
// put seconds in ts.tv_sec and nanoseconds in ts.tv_nsec
ts.tv_sec = 100;
ts.tv_nsec = 950;

clock_settime(CLOCK_REALTIME, &ts);

//Setting CLOCK_REALTIME requires appropriate privileges
```

Task Terminology

- Task parameters



execution time = a + b

Worst Case Execution Time (e_i) = max(execution time)

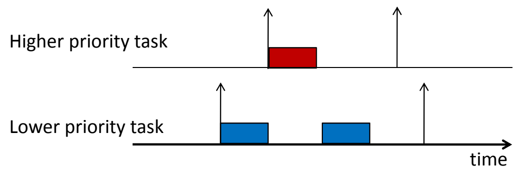
Task Model

- Task model includes a set of tasks with given assumptions.

$$T = \{\tau_1, \tau_2, \ldots \tau_n\},$$
$$\tau_i(e_i, \rho_i, d_i)$$

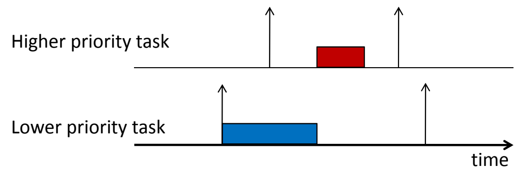
Preemptive, Non-preemptive Tasks

- Preemptive: A preemptive task can be interrupted by higher priority tasks at any time during its execution



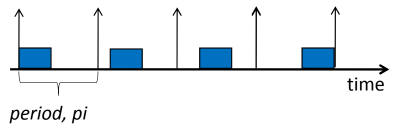
Preemptive, Non-preemptive Tasks

- Non-preemptive: a non-preemptive task cannot be interrupted by other tasks until end of its execution



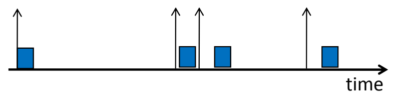
Periodic, Aperiodic, Sporadic Tasks

- Periodic: Execution of the task is repeated at constant time intervals. This constant interval is the period of the task



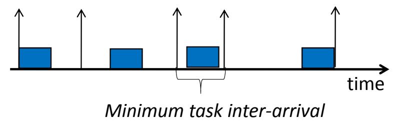
Periodic, Aperiodic, Sporadic Tasks

- Aperiodic: The task may execute at any time, e.g., in reaction to an external event.

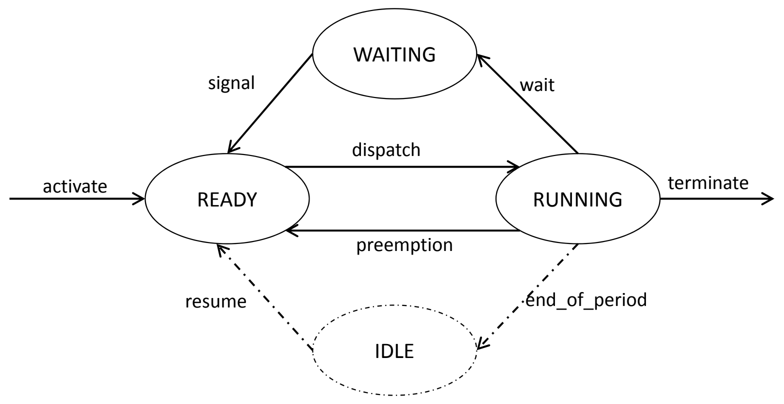


Periodic, Aperiodic, Sporadic Tasks

- Sporadic: An aperiodic task, however the time between two consecutive executions of the task shouldn't be less than a time interval, i.e., minimum time interval between two instances of the task.



Task States in RTOS



Task States in RTOS

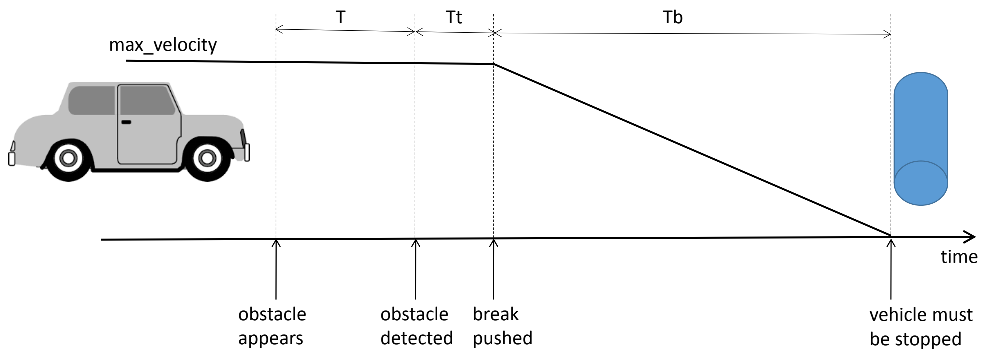
- Ready: When a task is ready to execute but it cannot execute because the processor is occupied by another higher priority task, it enters the Ready state. There could be multiple ready tasks, thus the RTOS puts these tasks in a *ready queue.*
- Running: When a task is assigned to the processor (starts executing), it enters Running state.
- Waiting: When a task during its execution has to wait for something that is not available at the moment, e.g., waits for some resource, data, a condition to be fulfilled, etc. In this case it enters Waiting state. When a task waits, for example on a resource, the RTOS puts the task into a queue associated with that resource.

Extracting Timing Restrictions

- Application requirements
- Physical World
  - Physical rules

Extracting Timing Restrictions

- Obstacle Avoidance System



Extracting Timing Restrictions

- The detection delay (period of sensor acquisition) is T time units; it should detect the obstacle and issue the break command no later than T time units. Sampling period is less than T.
- Example:
  - The maximum velocity is 30 m/s (108 km/h)
  - The transition time, Tt, is 300 ms: the car will move for $0.3 \times 30 = 9$ meters during this time. This is the time it takes after sending the brake command until the brake is actuated.
  - After pushing the brake the car moves for 90 meters until it fully stops; the braking space = 90 meters

Extracting Timing Restrictions

- If the camera can first see the obstacle when it is 100 meters away from it, calculate the maximum time period for detection task.
- Solution:
  - The car moves for 9 meter during transmission
  - The maximum distance after which the brake command has to be sent equals to: $100 - (9 + 90) = 1$ meter. Thus the period of detection task should be less than $1/30 = 0.033$ s: 33 ms

Task Synchronization and Communication

- Shared Variable Based
  - The objects in shared memory to which multiple tasks can have access
- Message Passing Based
  - Explicit exchange of data among tasks; directly or through some intermediate entity.

Shared Variable Based Synchronization and Communication

- Atomic (Indivisible) Operation
- Critical Sections
- Mutual Exclusion
- Condition Synchronization
- Busy Waiting
- Semaphores
- Mutexes
- Conditions
- Monitors
- Barriers

Shared Variable Based Synchronization and Communication

- Producer – Consumer

```
//Producer                    //Consumer
while(1)                      while(1)
{                             {
   x = calculate();              y = globalVar;
   globalVar = x;                use(y);
}                             }
```

Atomic (Indivisible) Operation

- An operation that from the rest of system's view is an Indivisible, uninterruptible, and instantaneously performed operation

```
x = x + 1;


MOVE X,D0
ADD #1,D0
MOVE D0,X
```

Shared Variable Based Synchronization and Communication

```
//Task1                       //Task2
while(1)                      while(1)
{                             {
   x = x + 1;                    x = x - 1;
   y = y + 1;                    z = z + 1;
}                             }
```

Critical Sections, Mutual Exclusion

- Critical Sections
  - A piece of code (sequence of statements) that no more than one task can execute at the same time.
- Mutual Exclusion
  - The synchronization needed to protect a critical section from being executed by more than one task.

Condition Synchronization

- A task stops proceeding until a condition is satisfied. Another task may fulfill the condition and let (signal) the waiting task know when the condition is satisfied.

```
//Task1                          //Task2
...                              ...
wait until flag = 1              falg = 1;
flag = 0;                        signal Task1;
...                              ...
```

Busy Waiting (Spinning)

- Is used for Condition Synchronization
- A task continuously (in a loop) checks a condition and proceeds when the condition is satisfied.

Busy Waiting

- Producer-Consumer

```
//Producer                       //Consumer
while(1)                         while(1)
{                                {
   x = calculate();                 while (flag != 0)
                                    {
   while (flag != 1)                   //Just wait ...
   {                                }
      //Just wait ...               flag = 1;
   }
   flag = 0;                        y = globalVar;
                                    use(y);
   globalVar = x;                }
}
```

Busy Waiting

- Livelock: The tasks may stuck in a loop for condition checking and never exit the loop
- Inefficient: wasting processor with not useful work
- Developing and testing is difficult
- ...