Real-Time Embedded Systems

DT8025, Fall 2016

http://goo.gl/AZfc9l

Lecture 5

Masoumeh Taromirad

m.taromirad@hh.se

HÖGSKOLAN I HALMSTAD

Center for Research on Embedded Systems

School of Information Technology

Context Switching

The process of storing and restoring the state (more specifically, the execution context) of a process or thread so that execution can be resumed from the same point at a later time.

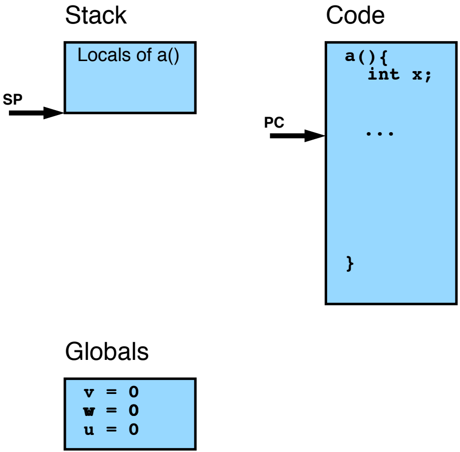
Stack Pointer

A small register that stores the address of the last program request in a stack.

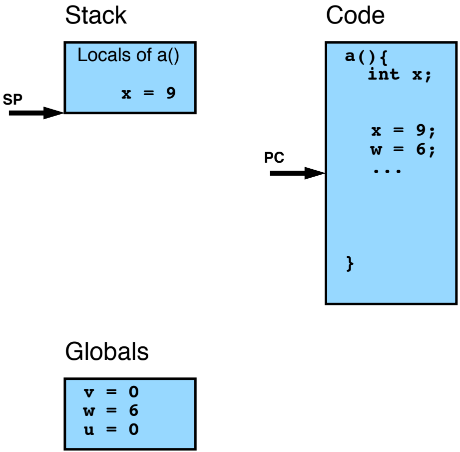
Program Counter

A processor register that indicates where a computer is in its program sequence.

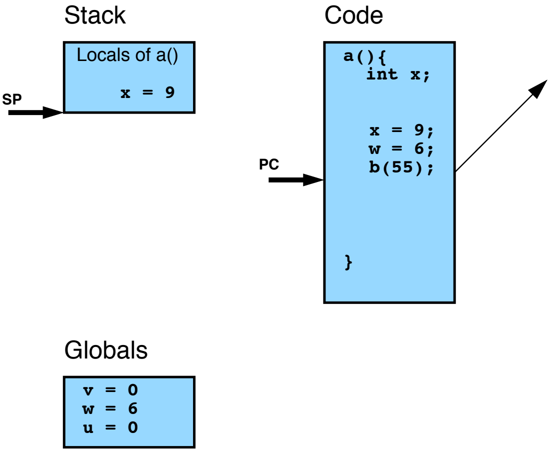
Execution of a C program



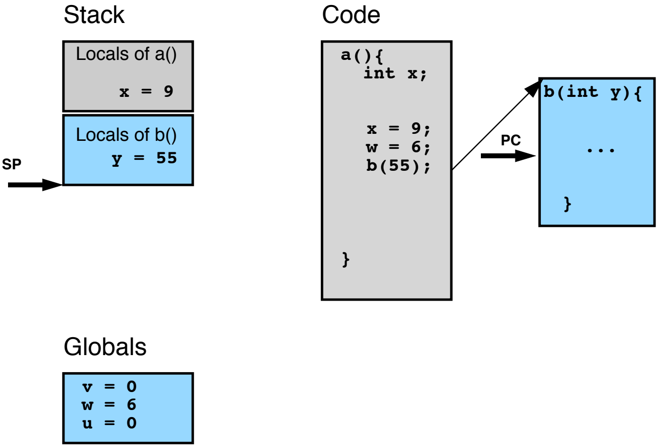
Execution of a C program



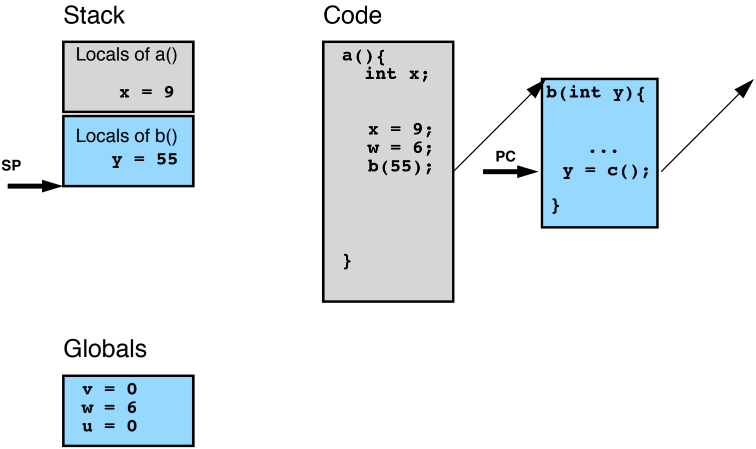
Execution of a C program



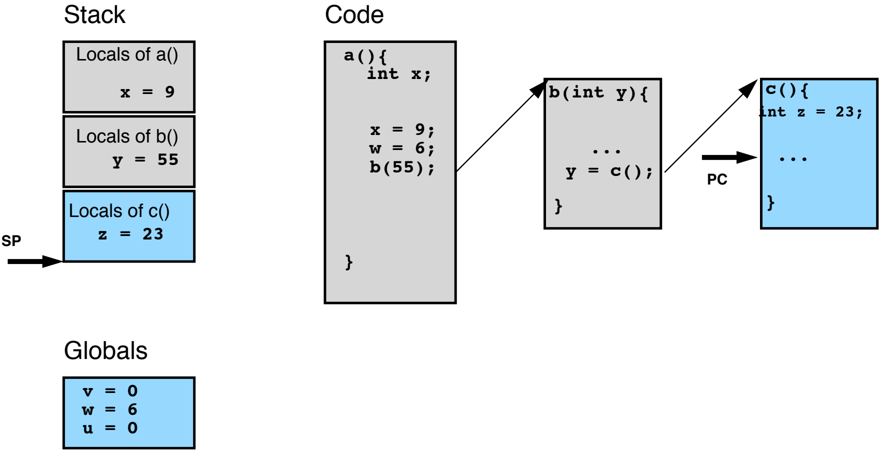
Execution of a C program



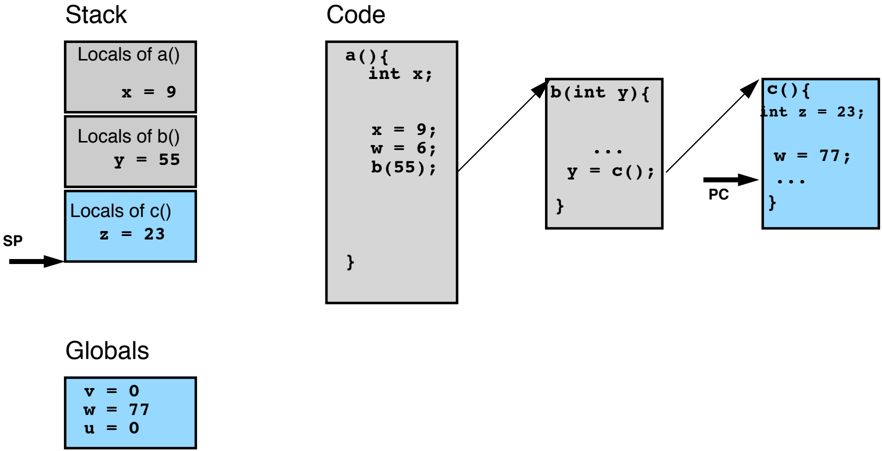
Execution of a C program



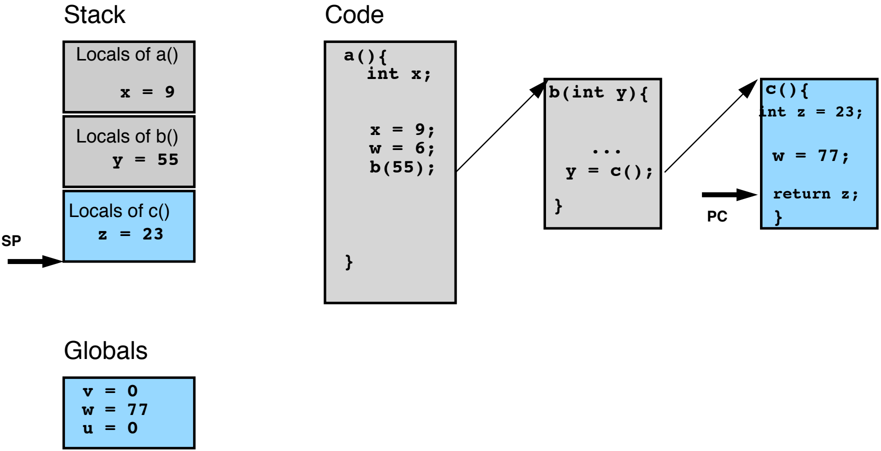
Execution of a C program



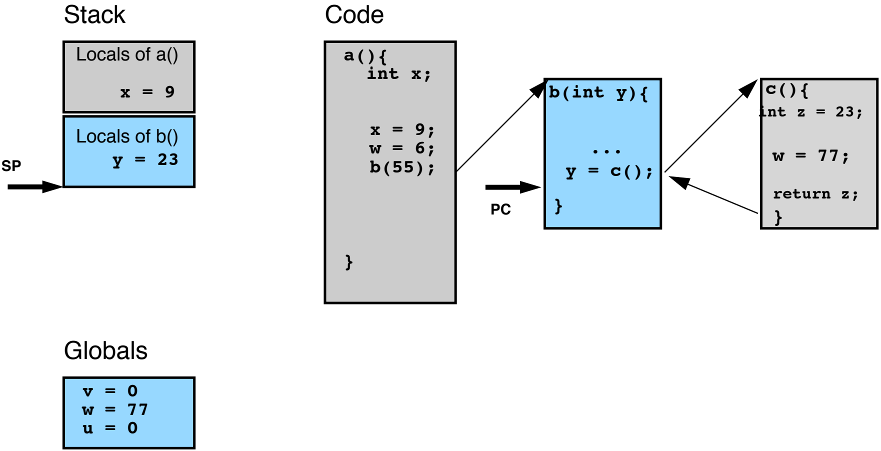
Execution of a C program



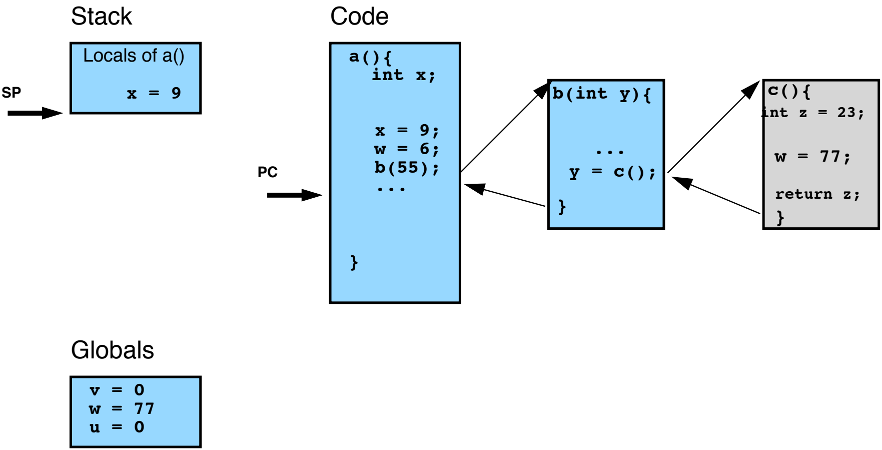
Execution of a C program



Execution of a C program



Execution of a C program



Concurrent Programs?

Imagine we had 2 CPUs, then we could run two programs at the same time!

One way of programming this in only 1 CPU is to keep track of 2 stack pointers and 2 program counters!

BUT ...
We want to provide means for two programs to execute concurrently! As if we had 2 CPUs!

What might a program look like?

```
main(){
  create_thread(decoder_main);
  controller_main();
}
```

Notice that the function create_thread takes a *function* as an argument!

The role of create_thread is to provide one extra **Program Counter** and **Stack Pointer**.

What we need ...

Thread Data Structure

A data structure describing a thread allowing us to keep track of the threads.

```
struct thread_block{
   ...
   void    (*fun)(void *)    // function to run
   void    (*arg);           // argument to the above
   Context context;          // pc and sp
   ...
};
typedef struct thread_block *thread
```

What we need ...

Variables for tracking threads

1. a queue of `threads`,
2. and the `current` thread.

Thread creation

A way of creating a thread: creating, initialising, and updating associated data structures.

A mean for interleaving

A way of interleaving fragments of the threads; yielding execution so that another thread can take over.

The kernel as a C library

ithreads.h

```
struct thread;
typedef struct thread thread;

int create_thread(void (*func)(void *),  void *arg);
int yield(void); // Your task in lab2!

struct lock{
    int locked;
    thread waitQ;
};
typedef struct lock lock;

void lock(lock *m);    // Your task in lab2!
void unlock(lock *m); // Your task in lab2!
```

Thread Data Structure

```
struct thread{
   ...
   void    (*fun)(void *)   // function to run
   void    (*arg);          // argument to the above
   Context context;       // pc and sp
   ...                    // ...
};
```

Context
It is very much platform dependent!
It is related to the function-call stack frame.
The size is some number of times the size of a function-call stack frame.

Global Variables

Current Thread

Keep track of the current (running) thread.

```
thread *runningthread;
```

Threads Queue

A queue of all the created threads (ready to run; runnable).

```
thread_queue *threadrunqueue;
```

Creating Threads

```
int create_thread(void (*func)(void *),  void *arg);
```

1. create a `thread` and initialise it, particularly the `context`.

   ```
   thread *t;
   memset(t, 0, sizeof *t);
   t->startfn = func;
   t->startarg = arg;

   memset(&t->context.uc, 0, sizeof t->context.uc);
   ```

2. enqueue the newly created thread `t` into the threads ready queue (`threadrunqueue`).

   ```
   //this will enqueue t in the ready queue!
   threadready(t);
   ```

The ready queue, the current thread and yield

Yielding control

By keeping the runnable threads in a queue, we can define a function `yield()` to switch execution to another thread.

`yield()` must

- enqueue the current thread in the ready queue
- pick a new thread from the ready queue and make it the current thread
- perform the context switch (also called dispatch)

Context Switch

Context Switch

```
static void contextswitch(Context *from, Context *to)
{
  //check if it is a valid context!
  if(getcontext(&from->uc) == 0)
    //set the current context!
    setcontext(&to->uc);
  return 0;
}
```

Who is the first current thread?

`main`

For `main` there is a **PC** and a **SP** and execution is set off when turning power on!

But `yield` should be able to deal with it as any other thread!

We introduce a thread block without initialization to be the first current thread. The first dispatch will set the context before enqueuing it in the ready queue!

Scheduling

When there are fewer processors than tasks (the usual case), or when tasks must be performed at a particular time, a scheduler must intervene.

The core of an implementation of threads is a scheduler. that decides which thread to execute next when a processor is available to execute a thread.

Scheduler

Makes the decision about what to do next at certain points in time, such as the time when a processor becomes available.

Real-time Systems

When in addition to any ordering constraints between the tasks, there are also timing constraints which relate the execution of a task to real-time.

Real-time

The physical time in the environment of the computer executing the task.

Real-time programs can have all manner of timing constraints

- deadline
- executed no earlier than a particular time
- executed periodically with some specified period
- ...

Basics of Scheduling

Scheduler

Decides what task to execute next when faced with a choice in the execution of a concurrent program or set of programs.

Multiprocessor Scheduler

Decides not only which task to execute next, but also on which processor to execute it. The choice of processor is called processor assignment.

Basics of Scheduling

Scheduling Decision

- assignment: which processor should execute the task.
- ordering: in what order each processor should execute its tasks.
- timing: the time at which each task executes.

Each of these three decisions may be made at

- design time, before the program begins executing, or at
- run time, during the execution of the program.

Basics of Scheduling

Different types of schedulers

Fully-static Scheduler

- Makes all three decisions at design time.
- The result is a precise specification for each processor of what to do when.

Static Order Scheduler

- performs the task assignment and ordering at design time.
- defers timing until run time: the decision of when in physical time to execute a task.
- The decision may be affected, for example, by whether a mutual exclusion lock can be acquired.
- also called off-line scheduler.

Basics of Scheduling

Different types of schedulers

Static Assignment Scheduler

- performs the assignment at design time
- and ordering and timing at run time.
- a run-time scheduler decides during execution what task to execute next.
- also called on-line scheduler

Basics of Scheduling

Different types of schedulers

Fully-dynamic Scheduler

- performs all decisions at run time.
- When a processor becomes available, the scheduler makes a decision at that point about what task to execute next on that processor.
- also called on-line scheduler

... more combinations!

- Assignment of a task may be done once for a task, at run time just prior to the first execution of the task.

Basics of Scheduling

Preemptive vs. non-preemptive

Preemptive Scheduler

Makes scheduling decision during the execution of a task, assigning a new task to the same processor.
It may decide to stop the execution of a task and begin execution of another one.
The interruption of the first task is called preemption.

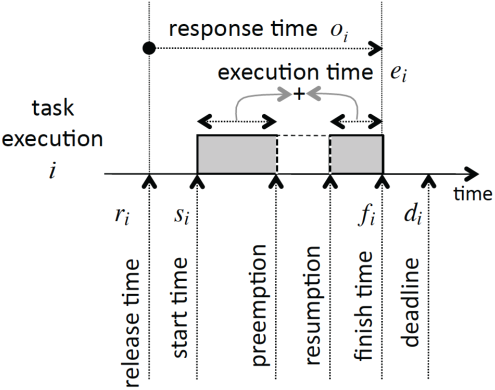
Non-preemptive Scheduler

Always lets tasks run to completion before assigning another task to execute on the same processor.

In preemptive scheduling, a task may be preempted if it attempts to acquire a mutual exclusion lock and the lock is not available.

Basics of Scheduling

Basic definitions

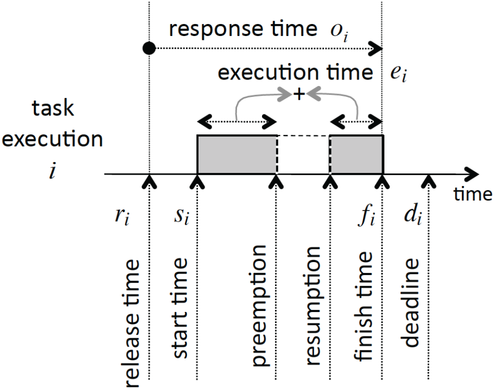


release time, r_i

the earliest time at which a task is enabled.

Basics of Scheduling

Basic definitions

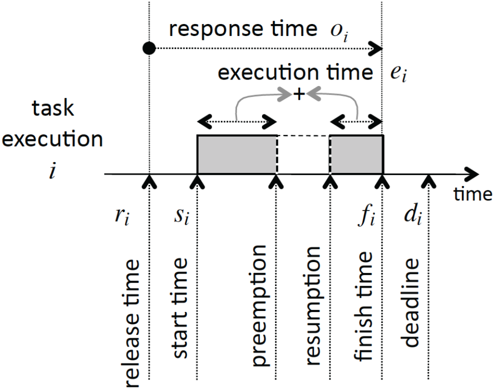


start time, s_i

the time at which the execution actually starts. $s_i \geq r_i$.

Basics of Scheduling

Basic definitions

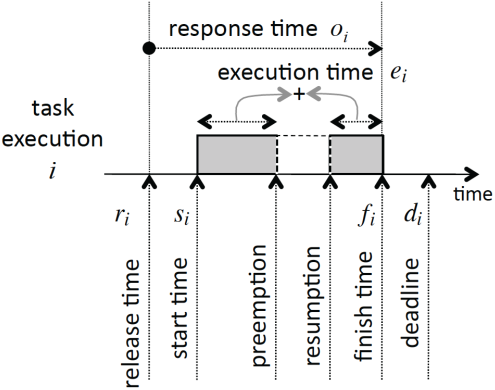


finish time, f_i

the time at which the task completes execution. $f_i \geq s_i$.

Basics of Scheduling

Basic definitions

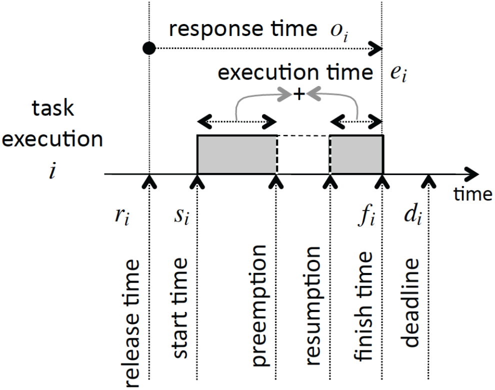


response time, o_i

the time that elapses between when the task is first enabled and when it completes execution. $o_i = f_i - r_i$.

Basics of Scheduling

Basic definitions

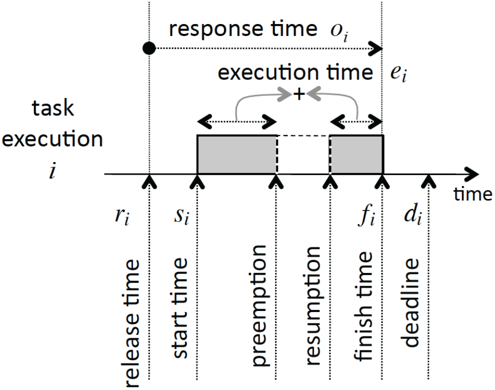


execution time, e_i

the total time that the task is actually executing.

Basics of Scheduling

Basic definitions



deadline, d_i

the time by which a task must be completed.

Comparing Schedulers

The choice of scheduling strategy is governed by considerations that depend on the goals of the application.

Feasible Schedule

A schedule that accomplishes the goal that all task executions meet their deadlines: $f_i \leq d_i$.

Comparison Criteria

- **utilization**: the percentage of time that the processor spends executing tasks (vs. being idle).
- maximum **lateness** (L) for a set of tasks T:

$$L = max_{i \in T}(f_i - d_i)$$

- **total completion time** (M) for a finite set of tasks T:

$$M = max_{i \in T}(f_i) - min_{i \in T}(r_i)$$

Scheduling Strategies

Rate Monotonic Scheduling

Earliest Deadline First

EDF with Precedences

...

Static priorities – method

Rate monotonic (RM)

Under the given assumptions, there exists a static priority assignment rule that is really simple

The shorter the period, the higher the priority

d For RM, the actual priority values do not matter, only their relative order.

Because of our inverse priority scale, we can simply implement RM by letting $P_i = D_i \ (=T_i)$

RM example

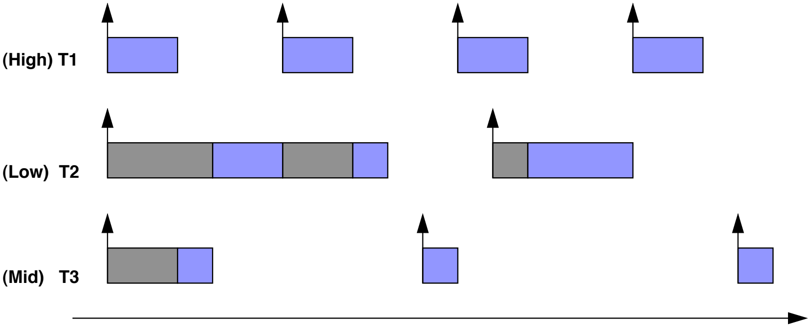
Given a set of periodic tasks with periods

T1	=	25ms
T2	=	60ms
T3	=	45ms

Valid priority assignments

P1	=	10	P1	=	1	P1	=	25
P2	=	19	P2	=	3	P2	=	60
P3	=	12	P3	=	2	P2	=	45

RM example



Period = Deadline. Arrows mark start of period.
Blue: running. Gray: waiting.

Dynamic priorities – method

Earliest Deadline First – EDF

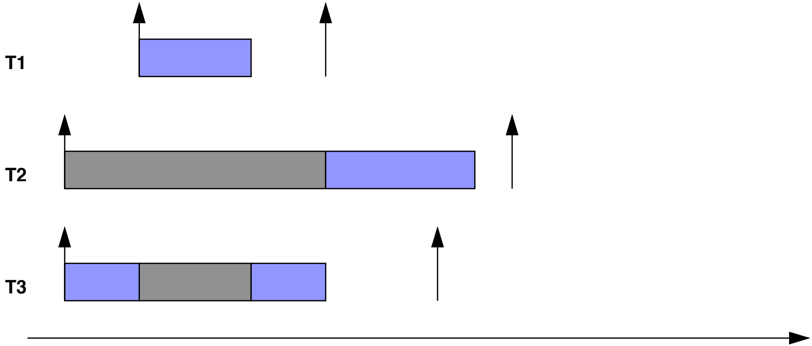
Dynamic priority assignment rule:

> The shorter the time remaining until deadline, the higher the priority

To use absolute deadlines: priorities = remaining clock cycles (before missing the deadline)

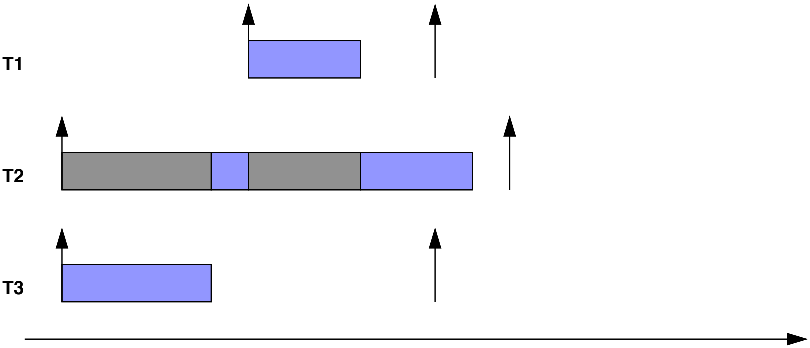
Under EDF, each activation n of periodic task i will receive a new priority: $P_{i(n)} = \mathrm{baseline}_{i(n)} + D_i$

EDF example



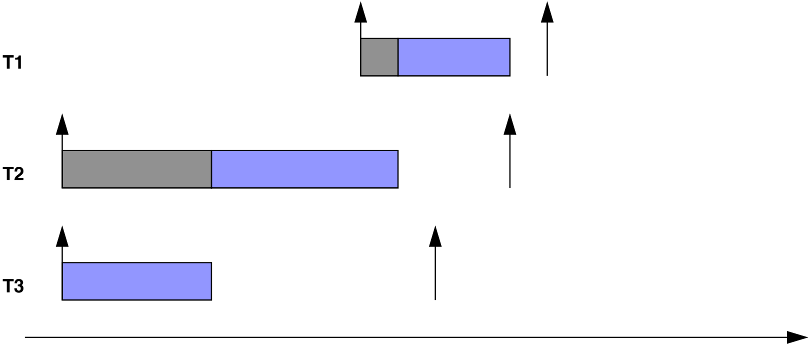
T1 arrives later, but its deadline is earlier than both T2's and T3's absolute deadlines!

EDF example



Deadline of T1 $<$ Deadline of T2

EDF example



(absolute) Deadline of T1 > (absolute) Deadline of T2