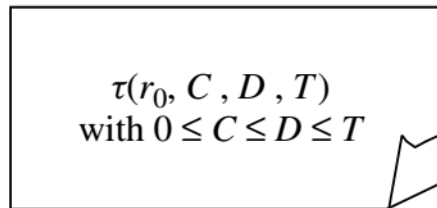


Linux real-time scheduling
including
EDF (Earliest Deadline First)
scheduling in linux 3.14

Real-time parameters for a thread



r_0 : release time of the 1st request of task

C : worst-case computation time

D : relative deadline

T : period

r_k : release time of $k+1$ th request of task

$r_k = r_0 + kT$ is represented by \uparrow

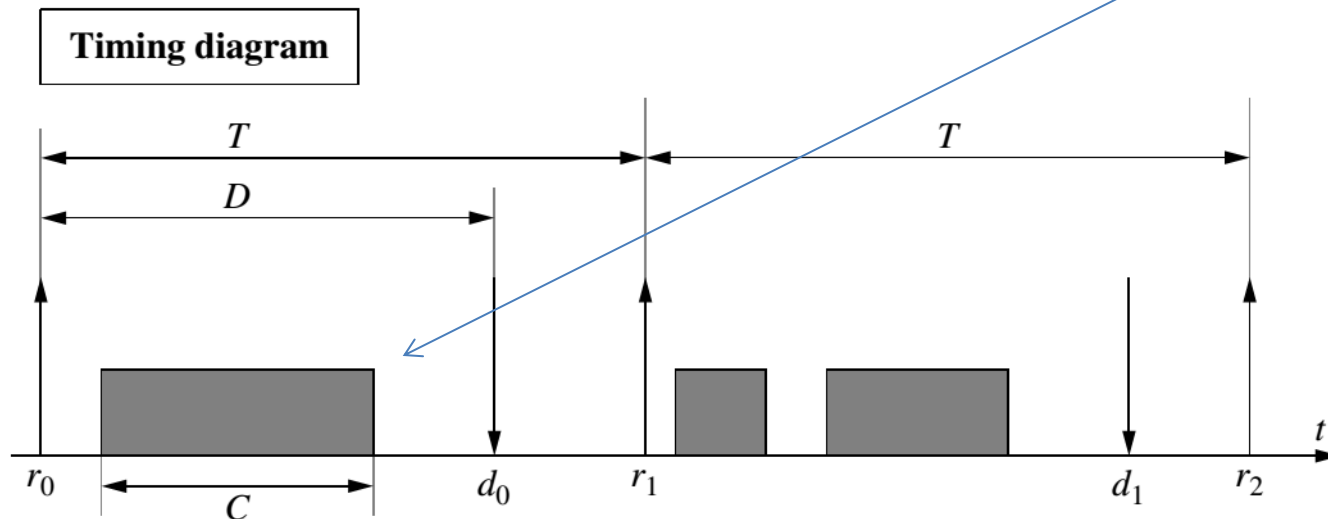
d_k : absolute deadline of $k+1$ th request of task

$d_k = r_k + D$ is represented by \downarrow

Note: for periodic task with $D = T$ (deadline equal to period)

deadline at next release time is represented by \updownarrow

Called a job in linux-scheduling



Rate-monotonic scheduling

- shorter period means higher priority

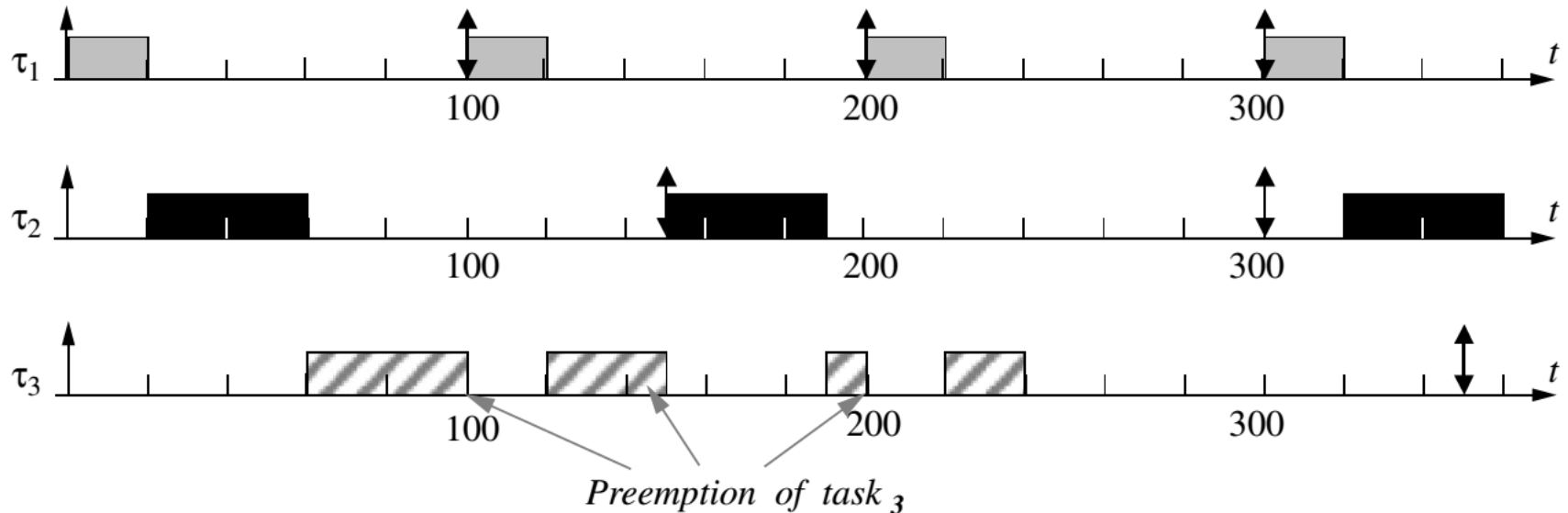


Figure 2.6 Example of a rate monotonic schedule with three periodic tasks: τ_1 (0, 20, 100, 100), τ_2 (0, 40, 150, 150) and τ_3 (0, 100, 350, 350)

Rate-monotonic scheduling criteria

- from the textbook:

We can generalize this result for an arbitrary set of n periodic tasks, and we get a sufficient schedulability condition (Buttazzo, 1997; Liu and Layland, 1973).

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n \cdot (2^{1/n} - 1) \quad (2.12)$$

This upper bound converges to $\ln(2) = 0.69$ for high values of n . A simulation study

EDF (Earliest Deadline First)

Task with Earliest Deadline First.
Dynamic algorithm.

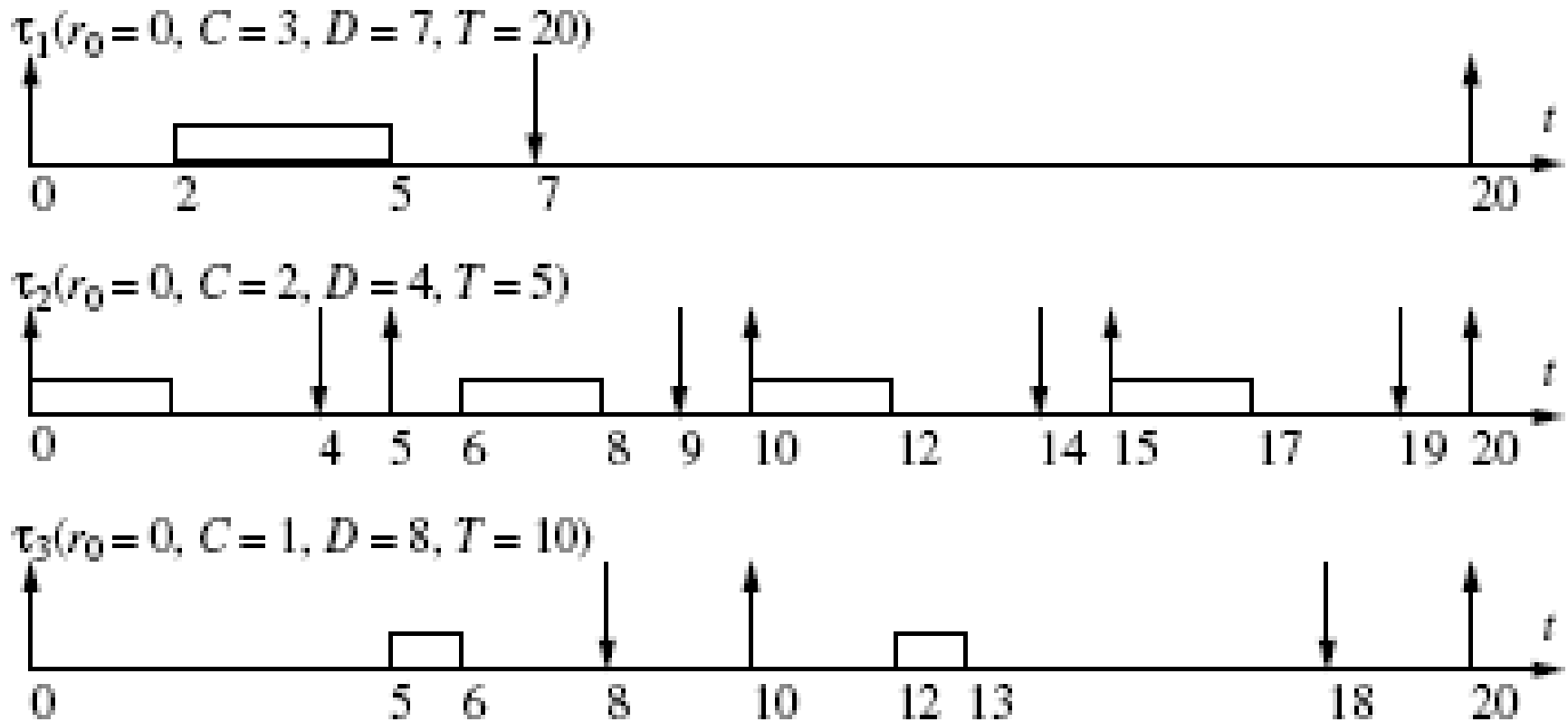


Figure 2.8 EDF schedule

EDF scheduling criteria

- from the textbook:

It is important to notice that a necessary and sufficient schedulability condition exists for periodic tasks with deadlines equal to periods. A set of periodic tasks with deadlines equal to periods is schedulable with the EDF algorithm if and only if the processor utilization factor is less than or equal to 1:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \quad (2.14)$$

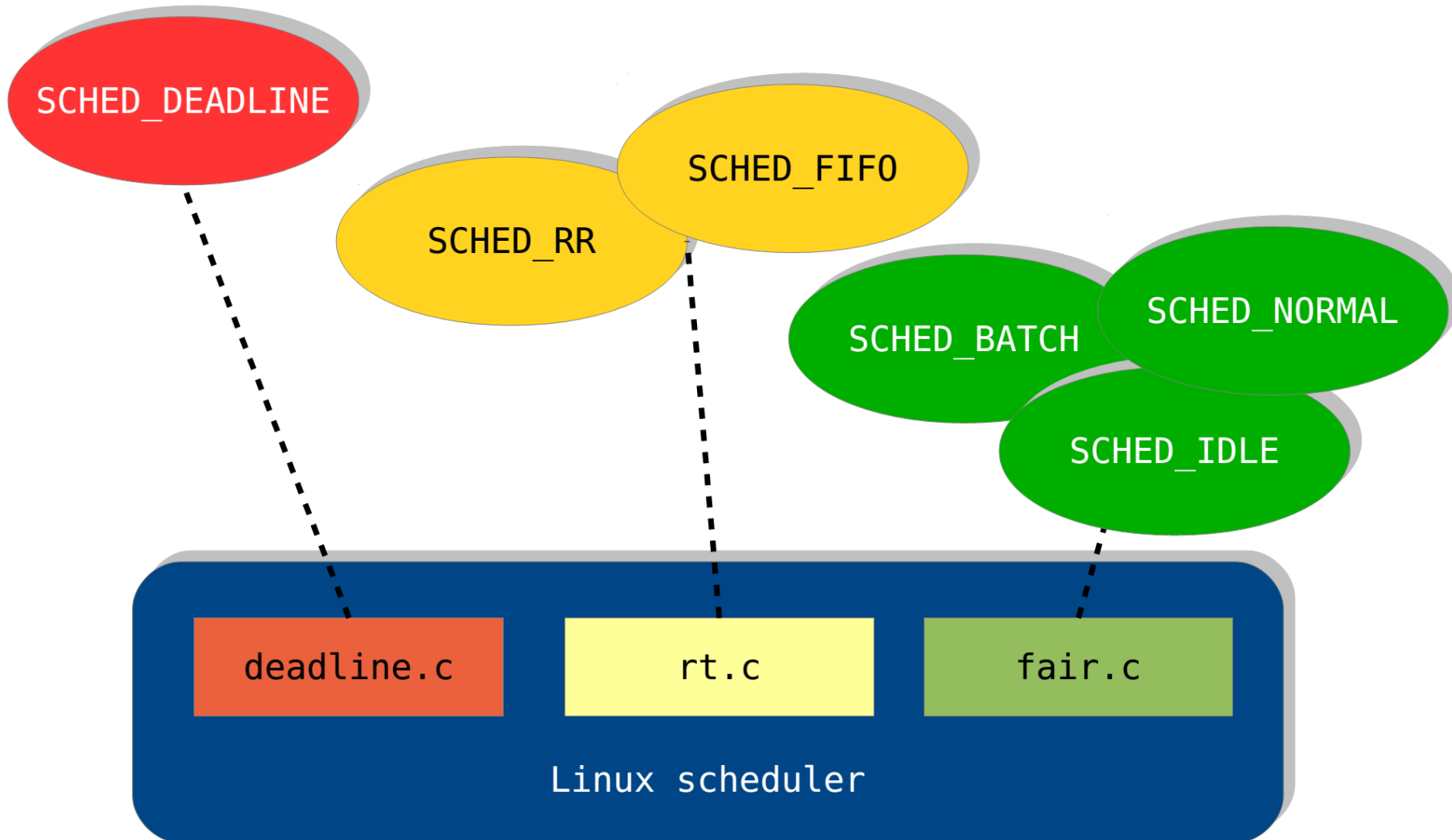
A hybrid task set is schedulable with the EDF algorithm if (sufficient condition):

$$\sum_{i=1}^n \frac{C_i}{D_i} \leq 1 \quad (2.15)$$



A mixture of periodic and aperiodic tasks

A new scheduling policy/class



POSIX: thread attributes

```
#include <pthread.h>
```

```
int pthread_attr_init(pthread_attr_t *attr);  
int pthread_attr_destroy(pthread_attr_t *attr);
```

The **pthread_attr_init()** function initializes the thread attributes object pointed to by *attr* with default attribute values.

After this call, individual attributes of the object can be set using various related functions, and then the object can be used in one or more [pthread_create\(3\)](#) calls that create threads.

Struct pthread_attr_t contains:

int	__detachstate	
int	__schedpolicy	
struct	__sched_param	__schedparam
int	__inheritsched	
int	__scope	
size_t	__guardsize	
int	__stackaddr_set	
void *	__stackaddr	
size_t	__stacksize	

A guard area consists of virtual memory pages that are protected to prevent read and write access. If a thread overflows its stack into the guard area, then, on most hard architectures, it receives a **SIGSEGV** signal, thus notifying it of the overflow.

POSIX: Inherit scheduling attributes or not?

```
#include <pthread.h>
```

```
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);  
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inheritsched);
```

The following values may be specified in *inheritsched*:

PTHREAD_INHERIT_SCHED

Threads that are created using *attr* inherit scheduling attributes from the creating thread; the scheduling attributes in *attr* are ignored.

PTHREAD_EXPLICIT_SCHED

Threads that are created using *attr* take their scheduling attributes from the values specified by the attributes object.

The default setting of the inherit-scheduler attribute in a newly initialized thread attributes object is **PTHREAD_INHERIT_SCHED**.

POSIX: Setting priority and policy

```
#include <pthread.h>
```

```
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);  
int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);
```

Scheduling parameters are maintained in the following structure:

```
struct sched_param  
{ int sched_priority; /* Scheduling priority */ };
```

```
#include <pthread.h>
```

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);  
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
```

The supported values for *policy* are **SCHED_FIFO**, **SCHED_RR**, and **SCHED_OTHER**

POSIX: And a program

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>

pthread_t A_id ;

void* A(void *vptr)
{ ....}

int main(void)
{
    struct sched_param params;
    params.sched_priority = 19;
    pthread_attr_t attr;
    pthread_attr_init( &attr );
    pthread_attr_setinheritsched( &attr, PTHREAD_EXPLICIT_SCHED );
    pthread_attr_setschedpolicy( &attr, SCHED_FIFO );
    pthread_attr_setschedparam( &attr, &params );

    pthread_create( &A_id, &attr, &A, NULL );
    pthread_join( A_id, NULL);

    return 0;
}
```

An history of LKML posts

- started within the ACTORS EU project (2008)
- originally developed and maintained by Dario Faggioli et al.
- then I switched in (v4 April 6th, 2012)
- 9 versions posted
- Merged by Linus on Sun, 2 Feb 2014 17:12:22
- **WE ARE IN MAINLINE!!! YAY!!!**
- Current stable is **Linux 3.14.2**

EDF in linux 3.14

- SCHED_DEADLINE
 - EDF (Earliest Deadline First scheduling)
 - Dynamic on-line algorithm
 - At each job-arrival to the job-queue the thread amongst those running or in the job-queue having the nearest absolute deadline is chosen for execution in the CPU.
 - » If the arriving job has a nearer absolute deadline than a running job then the running job is immediately sent back to the job-queue and the newly arrived job is executed in the CPU instead(preemption)
 - CBS (Constant Bandwidth Server)
 - The CBS-algorithm evicts a thread from the CPU if it exceeds its sched_runtime in its period.
 - Thus temporal isolation between threads is secured.
 - » A "mad" thread cannot cause other threads missing their deadlines

Thread attributes in sched.h

```
struct sched_attr {
    u32 size;
    u32 sched_policy;
    u64 sched_flags;

    /* SCHED_NORMAL, SCHED_BATCH */
    s32 sched_nice;

    /* SCHED_FIFO, SCHED_RR */
    u32 sched_priority;

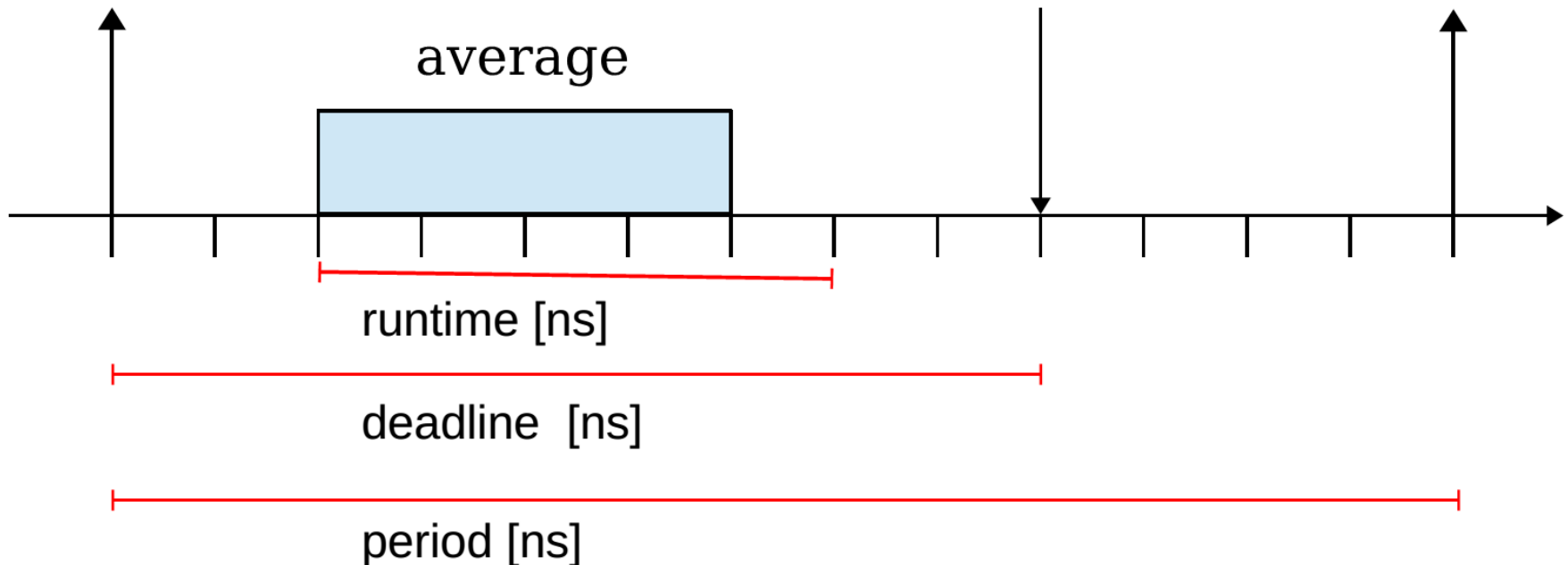
    /* SCHED_DEADLINE */
    u64 sched_runtime;
    u64 sched_deadline;
    u64 sched_period;
};

int sched_setattr(pid_t pid, const struct sched_attr *attr, unsigned int flags);

int sched_getattr(pid_t pid, const struct sched_attr *attr, unsigned int size,
unsigned int flags);
```

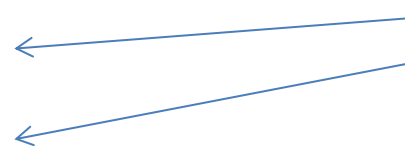
Rule of thumb for real-time parameters

simple rule of thumb, see literature for optimal settings...



A SCHED_DEADLINE thread is set up like this

```
#include <sched.h>
...
struct sched_attr attr;
attr.size = sizeof(struct attr);
attr.sched_policy = SCHED_DEADLINE;
attr.sched_runtime = 300000000;
attr.sched_period = 1000000000;
attr.sched_deadline = attr.sched_period;
...
if (sched_setattr(gettid(), &attr, 0))
    perror("sched_setattr()");
...
```



The diagram consists of two blue arrows pointing from the right towards the code. The top arrow points to the value 300000000 in the line `attr.sched_runtime = 300000000;`. The bottom arrow points to the value 1000000000 in the line `attr.sched_period = 1000000000;`. Both arrows originate from a single point on the right and point towards their respective values.

Nano-seconds

The class SCHED_DEADLINE

- Sched_deadline threads will always preempt threads from other thread classes
- Only root threads can belong to the sched-deadline class
- A thread cannot fork
- A thread can be periodic
 - Next release exactly one period later
- A thread can be sporadic
 - Next release at least one period later
- A thread can be aperiodic
 - Only deadline is defined for the thread
- New threads are only allowed to enter if they do not destroy the EDF-schedule for the threads, which already are in the EDF-schedule.
 - If not allowed to enter: the sched_setattr function returns EBUSY
- All thread parameters sched_runtime, sched_deadline and sched_period must be bigger than 1024 nano-seconds and less than 2^{63} nano-seconds (a pretty big number)
- The sched_deadline class can per default maximum occupy 95% of the time
 - A different value can be set in the file `/proc/sys/kernel/sched_rt_runtime_us`

More CPU's or more CPU cores

- Migration
 - Each cpu/core has it's own job queue job queue (a periodic execution of a thread is called a job)
 - The sched_deadline scheduler can migrate a thread to another CPU/core if that improves the situation.
 - Then n CPU's/cores run jobs from the n threads with the n earliest deadlines.

Still missing for SCHED_DEADLINE

- Priority inheritance
 - For mitigating "priority inversion"
 - A thread having a lock(mutex) on a resource, could "inherit" a earlier deadline from a thread also wanting to lock the ressource. In this way the resouce will be released earlier.
- Thread group bandwidth handling and scheduling(a wish from the car industry?)
- Non-root thread access to the sched-deadline class

Priority inversion scenario

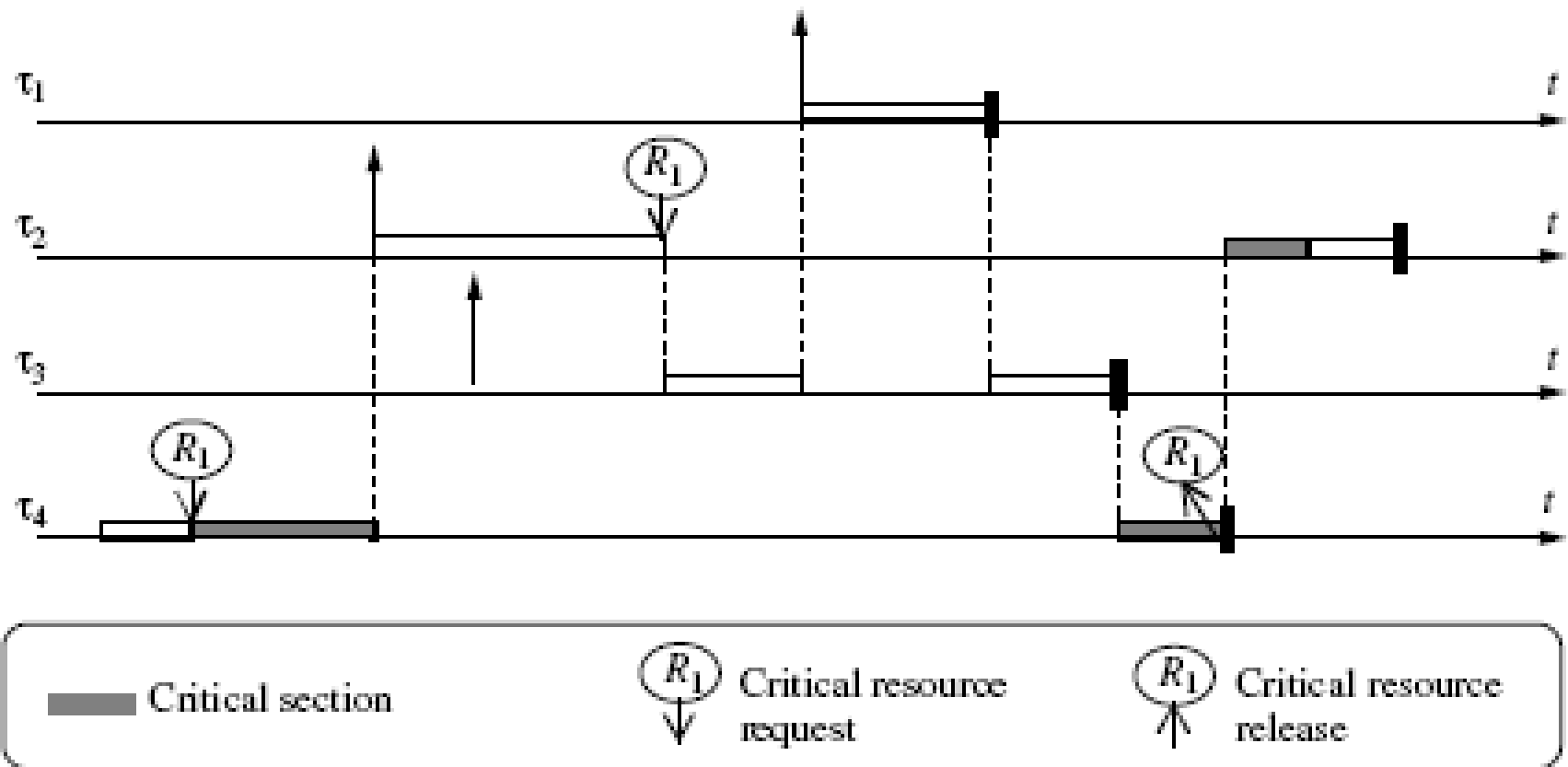


Figure 3.8 Example of priority inversion phenomenon

Linux offers the two anti-priority-inversion protocols

priority ceiling and priority inheritance

```
#include <pthread.h>
```

```
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict attr, int *restrict protocol);  
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of *protocol* may be one of: PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT which are defined in the [<pthread.h>](#) header.

When a thread is blocking higher priority threads because of owning one or more mutexes with the PTHREAD_PRIO_INHERIT *protocol* attribute, it shall execute at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

When a thread owns one or more mutexes initialized with the PTHREAD_PRIO_PROTECT protocol, it shall execute at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes or not.

```
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *restrict attr, int *restrict prioceiling);  
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);
```