DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING
[INQ0091623] AA 2021-22

Lab 12

Realtime Code Tunings

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

TOPICS:

Realtime is about timing precision on performing some operations.

So it reflects into:

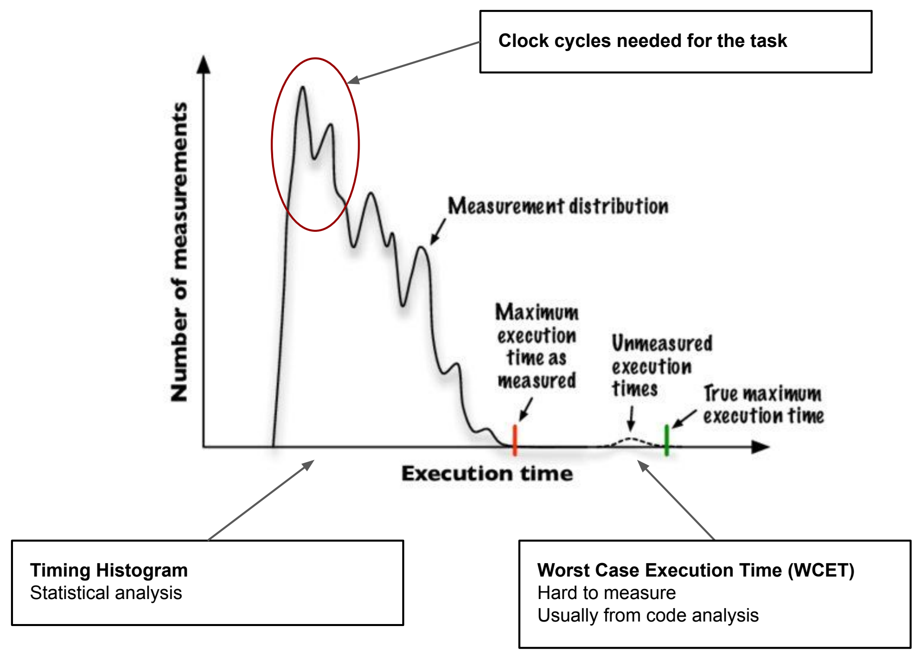
1. Measure time delays for a running and interrupt in case the time spans over a limit that would affect other tasks. This also needs a precise knowledge of what time actually is in every moment!

2. Remove as much as possible the noise that comes from other non related tasks and causes the jitter in the process operation times.

HO TO DO THAT:

- Design application with care !

- Tune the operative system



Realtime coding in GNU Linux

All scheduling system calls

```
nice(2)
       Set a new nice value for the calling thread, and return
       the new nice value.

getpriority(2)
       Return the nice value of a thread, a process group, or the
       set of threads owned by a specified user.

setpriority(2)
       Set the nice value of a thread, a process group, or the
       set of threads owned by a specified user.

sched_setscheduler(2)
       Set the scheduling policy and parameters of a specified
       thread.

sched_getscheduler(2)
       Return the scheduling policy of a specified thread.

sched_setparam(2)
       Set the scheduling parameters of a specified thread.

sched_getparam(2)
       Fetch the scheduling parameters of a specified thread.

sched_get_priority_max(2)
       Return the maximum priority available in a specified
       scheduling policy.

sched_get_priority_min(2)
       Return the minimum priority available in a specified
       scheduling policy.

sched_rr_get_interval(2)
       Fetch the quantum used for threads that are scheduled
       under the "round-robin" scheduling policy.
```

All scheduling system calls (2)

sched_yield(2)
Cause the caller to relinquish the CPU, so that some other thread be executed.

sched_setaffinity(2)
(Linux-specific) Set the CPU affinity of a specified thread.

sched_getaffinity(2)
(Linux-specific) Get the CPU affinity of a specified thread.

sched_setattr(2)
Set the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of **sched_setscheduler(2)** and **sched_setparam(2)**.

sched_getattr(2)
Fetch the scheduling policy and parameters of a specified thread. This (Linux-specific) system call provides a superset of the functionality of **sched_getscheduler(2)** and **sched_getparam(2)**.

pthread scheduling

```
#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <pthread.h>
```

Context switch active thread

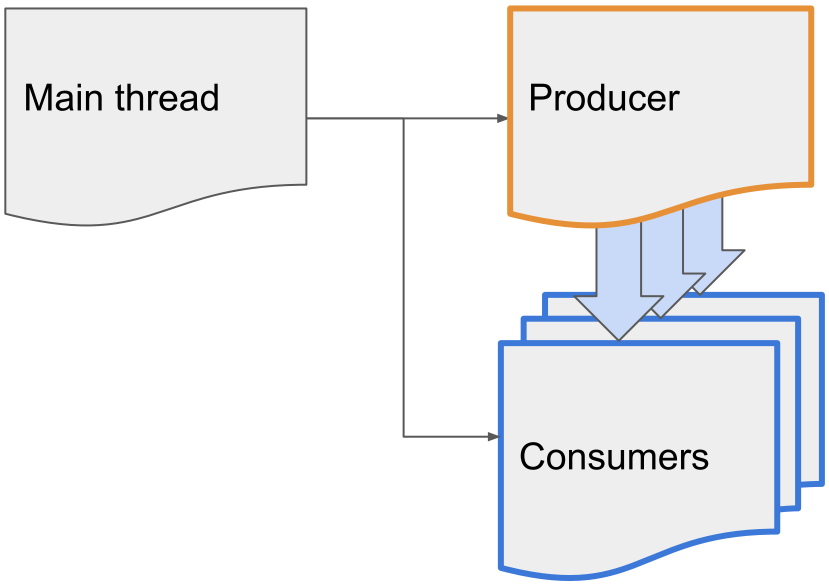
```
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);

int pthread_getaffinity_np(pthread_t thread, size_t cpusetsize, cpu_set_t *cpuset);
```

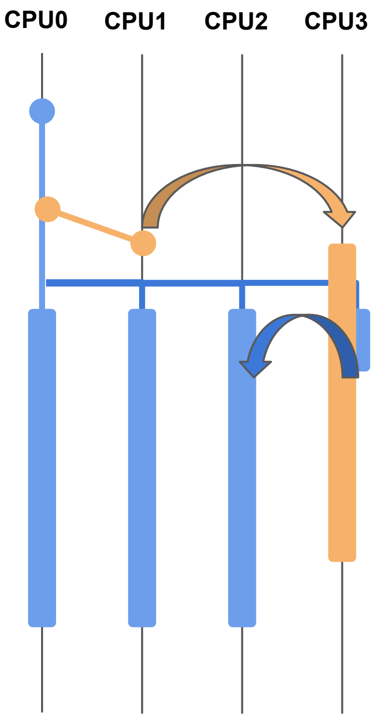
ATTRIBUTES BASED (before starting thread)

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);


int pthread_attr_setaffinity_np(pthread_attr_t *attr,
                                size_t cpusetsize, const cpu_set_t *cpuset);

int pthread_attr_getaffinity_np(const pthread_attr_t *attr,
                                size_t cpusetsize, cpu_set_t *cpuset);
```

Thread CPU affinity

Creating a thread and make it switch its context into a specific core.





Thread CPU affinity: prodcons_affinity.c

CODING EXAMPLE

CODING EXAMPLE

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sched.h> // sched_setaffinity



int main(int argc , char *args[])  {

   pthread_t threads[ MAX_THREADS];
   int i, nConsumers, producerAffinity;
   struct timespec t_start, t_end;

   sscanf(args[1], "%d", &nConsumers);
   sscanf(args[2], "%d", &producerAffinity);
   // cpu_set_t: This data set is a bitset where each bit represents a CPU.
   cpu_set_t producer_set, consumer_set;
   // CPU_ZERO: This macro initializes the CPU set set to be the empty set.
   CPU_ZERO(&producer_set);
   CPU_ZERO(&consumer_set);

   long number_of_processors =sysconf(_SC_NPROCESSORS_ONLN);
   // CPU_SET: This macro adds cpu to the CPU set set.
   CPU_SET(producerAffinity, &producer_set);

   for(i=0; i<number_of_processors; ++i)
       if( !CPU_ISSET(i, &producer_set) )
           CPU_SET(i, &consumer_set);
```

```
    pthread_mutex_init(& mutex , NULL);
    pthread_cond_init(& dataAvailable , NULL);
    pthread_cond_init(& roomAvailable , NULL);
    clock_gettime(CLOCK_REALTIME, &t_start);
    // send cpuset as param
    pthread_create(&threads[0], NULL , producer , &producer_set);
    for(i = 0; i < nConsumers; i++) {
        // send cpuset as param
        pthread_create(& threads[i+1], NULL , consumer , &consumer_set);
    }
    for(i = 0; i < nConsumers +1; i++) {
        pthread_join( threads[i], NULL);
    }
```

Thread CPU affinity (producer)

```
static void *producer(void *arg)
{
   int item = 0, i;
   // SET Affinity
   pthread_t thread_id = pthread_self();
   cpu_set_t *cpuset = (cpu_set_t *)arg;

   int status;
   status = pthread_setaffinity_np(thread_id, sizeof(cpu_set_t), cpuset);

   // CHECK CPU LOCATION //
   status = pthread_getaffinity_np(thread_id, sizeof(cpu_set_t), cpuset);

   pthread_mutex_lock(& mutex);
   printf("PRODUCER deployed in CPU:");
   for (int j = 0; j < CPU_SETSIZE; j++)
       if (CPU_ISSET(j, cpuset)) printf(" %d", j);
   printf("\n");
   pthread_mutex_unlock(&mutex);

   // PRODUCER CODE //
   ...
 }
```

CODING EXAMPLE CODING EXAMPLE

Thread CPU affinity (consumer)

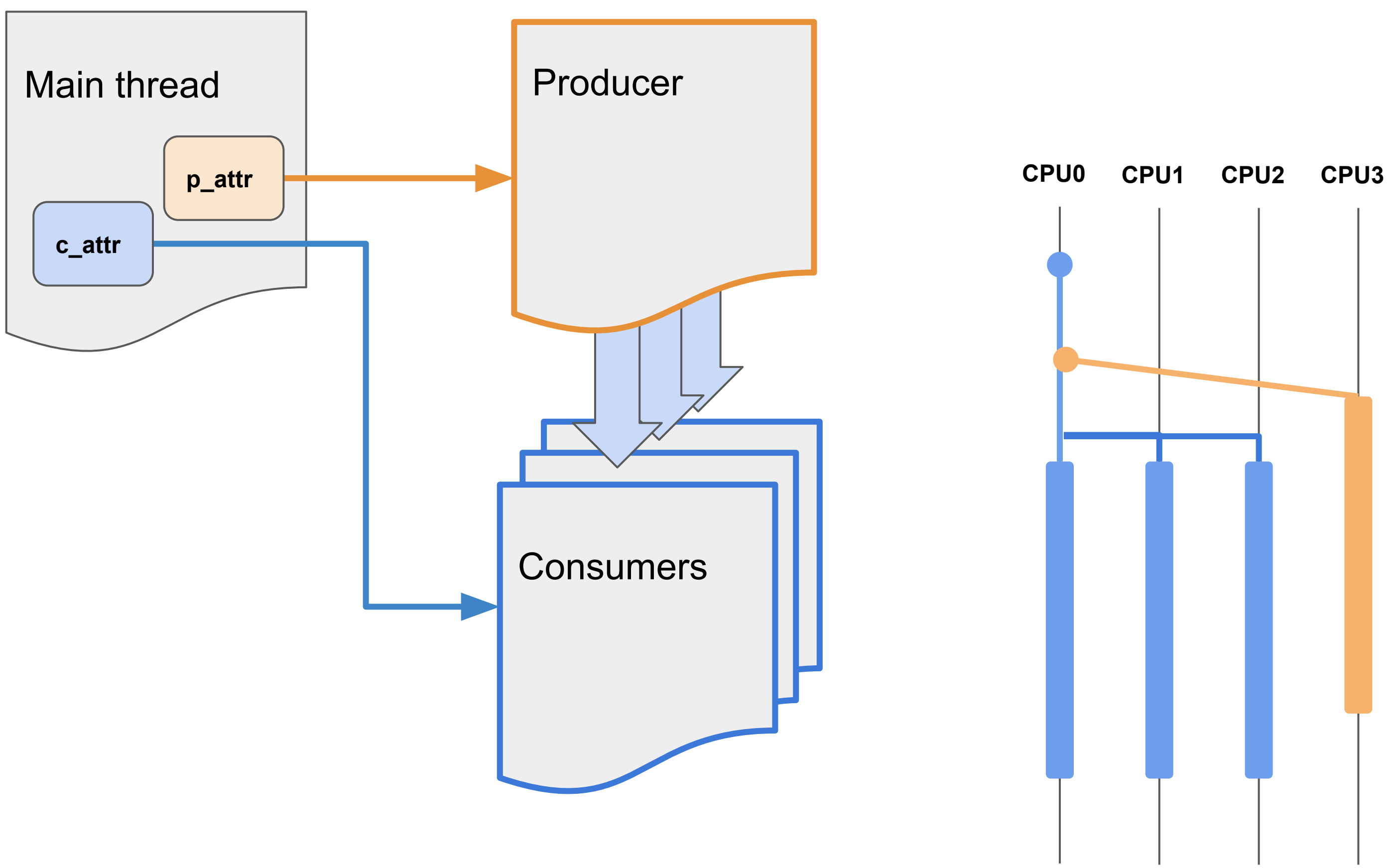
CODING EXAMPLE

CODING EXAMPLE

```
static void *consumer(void *arg)
{
   int item;
   // SET Affinity
   pthread_t thread_id = pthread_self();
   cpu_set_t *cpuset = (cpu_set_t *)arg;

   int status;
   status = pthread_setaffinity_np(thread_id, sizeof(cpu_set_t), cpuset);
   if (status) {
       perror("pthread_setaffinity_np");
       exit(EXIT_FAILURE);
   }

   // CHECK CPU LOCATION //
   status = pthread_getaffinity_np(thread_id, sizeof(cpu_set_t), cpuset);
   if (status) {
       perror("pthread_getaffinity_np");
       exit(EXIT_FAILURE);
   }
   pthread_mutex_lock(& mutex);
   printf("CONSUMER deployed in CPU:");
   for (int j = 0; j < CPU_SETSIZE; j++)
       if (CPU_ISSET(j, cpuset)) printf(" %d", j);
   printf("\n");
   pthread_mutex_unlock(&mutex);

   // CONSUMER CODE //
   for(;;)
   ...
}
```

Thread CPU affinity

Creating a thread and make it switch its context into a specific core.



Thread CPU affinity: prodcons_attributes.c

CODING EXAMPLE

```
int set_realtime_attribute(pthread_attr_t *attr, int policy, int priority, cpu_set_t *cpuset) {

   int status;
   struct sched_param param;

   // initialize default attributes
   pthread_attr_init(attr);

   // get current thread attributes parameters
   status = pthread_attr_getschedparam(attr, &param);

   // set to not inherit parameter from parent thread
   status = pthread_attr_setinheritsched(attr, PTHREAD_EXPLICIT_SCHED);

   // set the real-time scheduler as SHED_FIFO
   status = pthread_attr_setschedpolicy(attr, policy);

   // set the real-time priority parameter to 50 and apply it to scheduler attributes
   param.sched_priority = priority;
   status = pthread_attr_setschedparam(attr, &param);

   //
   // CPU AFFINITY
   //
   if(cpuset != NULL) {
       status = pthread_attr_setaffinity_np(attr, sizeof(cpu_set_t), cpuset);
       if(status) {
           perror("pthread_attr_setaffinity_np");
           return status;
       }
   }

   return status;
}
```

CODING EXAMPLE

Thread CPU affinity: prodcons_attributes.c

CODING EXAMPLE

```
int main(int argc , char *args[])
{
   pthread_t threads[ MAX_THREADS];
   int i, status, nConsumers, producerAffinity;

   // cpu_set_t: This data set is a bitset where each bit represents a CPU.
   cpu_set_t producer_set, consumer_set;

   // CPU_ZERO: This macro initializes the CPU set set to be the empty set.
   CPU_ZERO(&producer_set);
   CPU_ZERO(&consumer_set);

   long number_of_processors = sysconf(_SC_NPROCESSORS_ONLN);

   // CPU_SET: This macro adds cpu to the CPU set set.
   CPU_SET(producerAffinity, &producer_set);
   for(i=0; i<number_of_processors; ++i)
       if( !CPU_ISSET(i, &producer_set) )
           CPU_SET(i, &consumer_set);


   pthread_attr_t producer_attr, consumer_attr;
   set_realtime_attribute(&producer_attr, SCHED_FIFO, 10, &producer_set);
   set_realtime_attribute(&consumer_attr, SCHED_OTHER, 0, &consumer_set);


   // start producer with real=time scheduler, and send cpuset as param
   pthread_create(&threads[0], &producer_attr, producer , NULL);
   // start consumers, and send cpuset as param
   for(i = 0; i < nConsumers; i++) {
       pthread_create(& threads[i+1], &consumer_attr, consumer , NULL);
   }
   for(i = 0; i < nConsumers + 1; i++) {
       pthread_join( threads[i], NULL);
   }
```

CODING EXAMPLE

Program execution analysis

sudo ./prodcons_attributes 3 3 &

ps -A | grep prodcons

top -H -p 1185846

MiB Swap: 12288.0 total, 7971.2 free, 4316.8 used. 2521.3 avail Mem

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1185846	root	-11	0	36896	1040	932	S	0.3	0.0	0:00.11	prodcons_attrib
1185849	root	20	0	36896	1040	932	S	0.3	0.0	0:00.08	prodcons_attrib
1185845	root	20	0	36896	1040	932	S	0.0	0.0	0:00.00	prodcons_attrib
1185847	root	20	0	36896	1040	932	S	0.0	0.0	0:00.07	prodcons_attrib
1185848	root	20	0	36896	1040	932	R	0.0	0.0	0:00.06	prodcons_attrib

Linux EDF implementation

Realtime schedulers in Linux (REPLY)

The POSIX realtime scheduler provides the **FIFO (first-in-first-out)** and **RR (round-robin)** scheduling policies. It schedules each task according to its **fixed priority** (i.e. the task with the highest priority will be served first).

The difference between the FIFO and RR schedulers can be seen when two tasks share the same priority:

FIFO -> the task that arrived first will receive the processor, running until it goes to sleep.
RR -> the tasks with the same priority will share the processor in a round-robin fashion.

Once an RR task starts to run, it will run for a maximum quantum of time. If the task does not block before the end of that time slice, the scheduler will put the task at the end of the round-robin queue of the tasks with the same priority and select the next task to run.

In the realtime scheduler, the user needs to provide the scheduling policy and the fixed priority. For example:

chrt -f 10 video_processing_too

With this command, the video_processing_tool task will be scheduled by the realtime scheduler, with a priority of 10, under the FIFO policy (as requested by the -f flag).

period and deadline

Each realtime task is composed of N recurrent activations; a task activation is known as a job.

The activation pattern can be described as:

periodic: A task is said to be periodic when a job takes place after a fixed offset of time from its previous activation. For instance, a periodic task with period of 2ms will be activated every 2ms. Tasks can also be ***sporadic***. A sporadic task is activated after, at least, a minimum inter-arrival time from its previous activation. For instance, a sporadic task with a 2ms period will be activated after at least 2ms from the previous activation.

aperiodic: when there is no activation pattern that can be established.

Tasks can also have:

- ***implicit deadline***: when the deadline is equal to the activation period,
- ***constrained deadline***: when the deadline can be less than (or equal to) the period,
- ***arbitrary deadline***: when the deadline is unrelated to the period.

Early Deadline First

Using these patterns, realtime researchers have developed ways to compare scheduling algorithms by their ability to schedule a given task set.

It turns out that, for *uniprocessor systems*, the ***Early Deadline First*** (EDF) scheduler was found to be optimal (a scheduling algorithm is optimal when it fails to schedule a task set only when no other scheduler can schedule it).

The deadline scheduler is optimal for periodic and sporadic tasks with **deadlines less than or equal to their periods** on uniprocessor systems. Actually, for either periodic or sporadic tasks with implicit deadlines, the EDF scheduler can schedule any task set as long as the task set does not use more than 100% of the CPU time.

The SCHED_DEADLINE

How linux implements the Early Deadline First (EDF) scheduling policy?

SCHED_DEADLINE has been proposed by Dario Faggioli, Fabio Checconi, Michael Trimarchi, Claudio Scordino ***"An EDF scheduling class for the Linux kernel"*** at *11th Real-Time Linux Workshop (RTLWS)* ***2009***

In the deadline scheduler the user has three parameters to set:

1. the ***period*** is the activation pattern of the realtime task. In a practical example, if a video-processing task must process 60 frames per second, a new frame will arrive every 16 milliseconds, so the period is 16 milliseconds.

2. The ***run time*** is the amount of CPU time that the application needs to produce the output. In the most conservative case, the runtime must be the worst-case execution time (WCET), which is the maximum amount of time the task needs to process one period's worth of work. For example, a video processing tool may take, in the worst case, five milliseconds to process the image. Hence its run time is five milliseconds.

3. The ***deadline*** is the maximum time in which the result must be delivered by the task, relative to the period. For example, if the task needs to deliver the processed frame within ten milliseconds, the deadline will be ten milliseconds.

The SCHED_DEADLINE in chrt

It is possible to set deadline scheduling parameters using the chrt command. For example, the above-mentioned tool could be started with the following command:

```
chrt  -d  --sched-runtime  5000000   \
          --sched-period   16666666  \
          --sched-deadline 10000000  \
          0 video_processing_tool
```

Where:

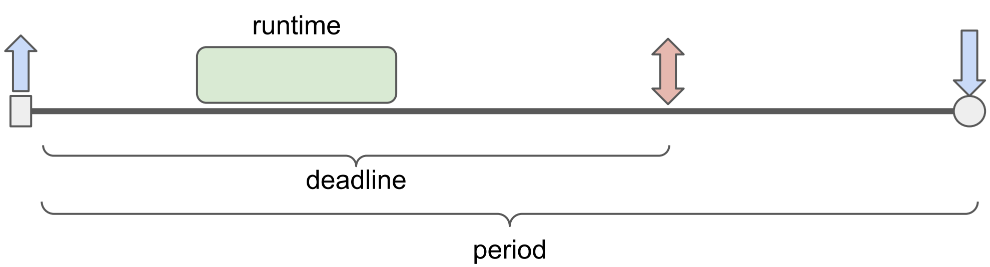
--sched-runtime 5000000 is the run time specified in nanoseconds (5ms)

--sched-deadline 10000000 is the relative deadline specified in nanoseconds (10ms).

--sched-period 16666666 is the period specified in nanoseconds

0 is a placeholder for the (unused) priority, required by the chrt command

In this way, the task will have a guarantee of 5ms of CPU time every 16.6ms, and all of that CPU time will be available for the task before the 10ms deadline passes.



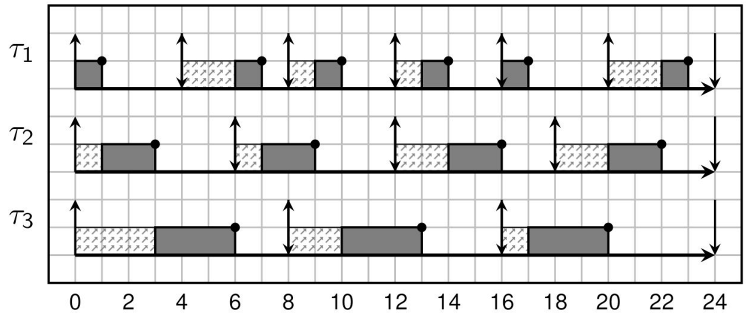
EXAMPLE

Consider, for instance, a system with three periodic tasks with deadlines equal to their periods:

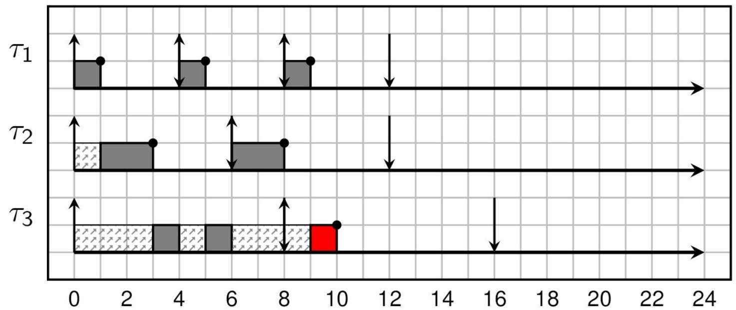
Task	Runtime (WCET)	Period = Deadline
T_1	1	4
T_2	2	6
T_3	3	8

The CPU time utilization (U) of this task set is less than 100%: **U =** 1/4 + 2/6 + 3/8 = **23/24**

For such a task set, the EDF scheduler would present the following behavior:



It is not possible to use a fixed-priority scheduler to schedule this task set while meeting every deadline; regardless of the assignment of priorities, one task will not run in time to get its work done. The resulting behavior will look like this:



PROS and CONS of EDF

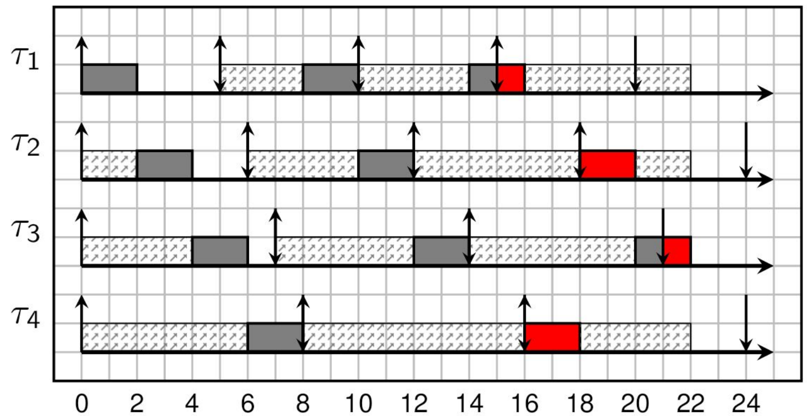
PROS:

- The main advantage of deadline scheduling is that, once you know each task parameters, you do not need to analyze all of the other tasks to know that your tasks will all meet their deadlines.
- Deadline scheduling often results in fewer context switches and, on uniprocessor systems, deadline scheduling is able to schedule more tasks than fixed priority-scheduling while meeting every task deadline.

CONS:

- It is not possible to ensure a minimum response time for any given task. In the fixed-priority scheduler, the highest-priority task always has the minimum response time.
- The EDF scheduling algorithm is implemented in O(log(n)) where fixed-priority can be implemented with O(1) complexity. However, the fixed-priority requires an “offline computation” of the best set of priorities by the user, which can be as complex as O(N).
- The deadline scheduler is not optimal and does not guarantee U>1 for more than 1 CPU
- The deadline scheduler does not support CPU affinity yet !

If, for some reason, the system becomes overloaded, for instance due to the addition of a new task or a wrong WCET estimation, it is possible to face a domino effect: once one task misses its deadline by running for more than its declared run time, all other tasks may miss their deadlines as shown by the regions in red below:



with fixed-priority scheduling, only the tasks with lower priority than the task which missed the deadline would have been affected.

Multi processors (Dhall’s Effect)

In multi-core systems, global, clustered, and arbitrary deadline schedulers are not optimal.

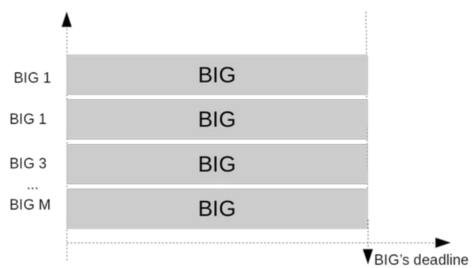
For example, in a system with M processors, it is possible to schedule M tasks with a run time equal to the period. For instance, a system with four processors can schedule four "BIG" tasks with both run time and period equal to 1000ms. In this case, the system will reach the maximum utilization of:

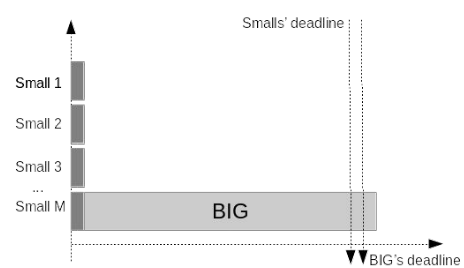
4 * 1000/1000 = 4

But.. what if a task set is composed of four small tasks with the minimum runtime, let's say 1ms, at every 999 milliseconds period, and just one task BIG task, with runtime and period of one second. The load of this system is:

4 * (1/999) + 1000/1000 = 1.004

As 1.004 is smaller than four, intuitively, one might say that the system is schedulable, But that is not true for global EDF scheduling. That is because, **if all tasks are released at the same time, the M small tasks will be scheduled in the M available processors**. Then, the big task will be able to start only after the small tasks have run, hence finishing its computation after its deadline.





EDF is able to guarantee deadline only up to U = 1. No matter how many processor you have.

As Dhall's effect shows, the global deadline scheduler acceptance task is unable to schedule the task set even though there is CPU time available. Hence, once accepted, the tasks will be able to use all the assigned run time before their deadlines. If the user wants to guarantee that all tasks will meet their deadlines, the user has to either use a partitioned approach or to use a necessary and sufficient acceptance test, defined by:

$$\Sigma(WCETi / Pi) \le M - (M - 1) \times Umax$$

Or, expressed in words: the sum of the run time/period of each task should be less than or equal to the number of processors, minus the largest utilization multiplied by the number of processors minus one. It turns out that, the bigger Umax is, the less load the system is able to handle.

deadline scheduler insulation: cgroup

SCHED_DEADLINE does not support cpu_affinity !

SOLUTION: partitioning the cpu in the system with cgroup

For example, consider a system with eight CPUs. One big task has a utilization close to 90% of one CPU, while a set of many other tasks have a lower utilization. In this environment, one recommended setup would be to isolate CPU0 to run the high-utilization task while allowing the other tasks to run in the remaining CPUs. To configure this environment, the user must follow the following steps:

Enter in the cpuset directory and create two cpusets:

```
# cd /sys/fs/cgroup/cpuset/
# mkdir cluster
# mkdir partition
```

Enter the directory for the cluster cpuset, set the CPUs available to 1-7, the memory node the set should run in (in this case the system is not NUMA, so it is always node zero).

```
# cd cluster/
# echo 1-7 > cpuset.cpus
# echo 0 > cpuset.mems
```

Move all tasks to this CPU set

```
# ps -eLo lwp | while read thread; do echo $thread > cgroup.procs ; done
```

Then it is possible to start deadline tasks in this cpuset.

deadline scheduler insulation: cgroup

Configure the partition cpuset:

```
# cd ../partition/
# echo 0 > cpuset.mems
# echo 0 > cpuset.cpus
# echo isolated > cpuset.cpus.partition
```

Finally move the shell to the partition cpuset.

```
# echo $$ > cgroup.procs
```

The final step is to run the deadline workload.

```
# cd $HOME/crtp/crtp/src/lab12
# ./sched_deadline
```

sched_deadline.h

CODING EXAMPLE

CODING EXAMPLE

```c
#include <linux/kernel.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <time.h>
#include <linux/types.h>
#include <sched.h>
#include <linux/sched.h>
#include <sys/types.h>

#define SCHED_DEADLINE  6

/*   NR sched setattr number */
#ifndef   NR sched_setattr
#ifdef   x86 64
#define __NR_sched_setattr      314
#endif

#ifdef   i386
#define __NR_sched_setattr      351
#endif

#ifdef   arm
#define __NR_sched_setattr      380
#endif

#ifdef   aarch64
#define __NR_sched_setattr      274
#endif
#endif

/*   NR sched getattr number */
#ifndef   NR sched_getattr
#ifdef   x86 64
#define __NR_sched_getattr      315
#endif

#ifdef   i386
#define __NR_sched_getattr      352
#endif

#ifdef   arm
#define __NR_sched_getattr      381
#endif

#ifdef   aarch64
#define __NR_sched_getattr      275
#endif
#endif
```

```c
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)
{
   return syscall(__NR_sched_setattr, pid, attr, flags);
}

int sched_getattr(pid_t pid,
             struct sched_attr *attr,
             unsigned int size,
             unsigned int flags)
{
   return syscall(__NR_sched_getattr, pid, attr, size, flags);
}
```

sched_deadline.c

CODING EXAMPLE

CODING EXAMPLE

```c
#include <stdlib.h>
#include <string.h>
#include "sched_deadline.h"

int main (int argc, char **argv)
{
   int ret;
   int flags = 0;
   struct sched_attr attr;

   memset(&attr, 0, sizeof(attr));
   attr.size = sizeof(attr);

   /* This creates a 200ms / 1s reservation */
   attr.sched_policy   = SCHED_DEADLINE;
   attr.sched_runtime  = 200000000;
   attr.sched_deadline = attr.sched_period = 1000000000;

   ret = sched_setattr(0, &attr, flags);
   if (ret < 0) {
       perror("sched_setattr failed to set the priorities");
       exit(-1);
   }

   for(;;) {
       // doing computation //
       usleep(2);
       sched_yield();
   }

   exit(0);
}
```

thank you for your attention and participation following this course

<andrea.rigonigarola@unipd.it>