UNIVERSITÀ DEGLI STUDI DI PARMA
Dipartimento di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria Informatica

CORSO DI SISTEMI OPERATIVI E IN TEMPO REALE
Esercitazione *Real-Time* n.2

1 Scheduling Real Time

Il linguaggio C++ non definisce API *native* per manipolare lo *scheduling real-time* dei thread, però la classe `std::thread` mette a disposizione il metodo `native_handle()` con il quale è possibile avere accesso all'implementazione dell'oggetto *thread* ed utilizzare API proprietarie del sistema operativo (se esistono) per controllarne lo scheduling.

La maggior parte dei sistemi operativi *UNIX-like* (*GNU/Linux*, *MacOSX*, ecc.) rispondono ad uno standard denominato *POSIX (Portable Operating System Interface for uniX)* che, tra la altre cose, definisce un set di API in linguaggio C per creare, sincronizzare e controllare lo scheduling di thread: i thread C++ in ambiente *linux* sono realizzati facendo uso delle API POSIX (si noti il parametro `-pthread` del compilatore `g++`).

1.1 POSIX Threads & Realtime Extensions

Lo standard POSIX definisce tre differenti *politiche di scheduling*:

- `SCHED_OTHER`: Politica *Non Real-Time*. È lo scheduler di default, il cui comportamento dipende dal sistema operativo. Non fornisce prestazioni real-time e tipicamente gestisce i processi e thread mediante un meccanismo di *priorità dinamiche*;
- `SCHED_FIFO`: Politica *Real-Time FIFO*. I thread schedulati secondo questa politica hanno un valore numerico che ne indica la *priorità statica*. In ogni momento viene posto in esecuzione il thread *pronto* avente priorità maggiore. I thread aventi la medesima priorità vengono inseriti in una coda e posti in esecuzione con politica *FIFO*;
- `SCHED_RR`: Politica *Real-Time RoundRobin*. Funziona allo stesso modo dello scheduler FIFO, ma in questo caso lo scheduler effettua la preemption di un thread che rimane in esecuzione per un tempo superiore ad un *quanto temporale* fissato, riposizionandolo in fondo alla coda relativa al suo valore di prioprità.

Ogni thread *linux* schedulato in modalità non-RT possiede una **priorità dinamica** che viene costantemente aggiornata dal sistema operativo in modo da impedire che un processo in esecuzione possa tenere bloccati indefinitamente gli altri processi del sistema.

La **priorità statica** associata ad un thread non-RT è sempre pari a 0, mentre per un thread RT questo valore può variare nel range [1..99] (nei sistemi GNU-Linux), pertanto un thread RT (avente priorità maggiore di 0) assume immediatamente il controllo della CPU qualora siano in esecuzione solo processi non-RT.

Si può notare che un processo RT può impedire indefinitamente l'esecuzione dei processi non-RT che si trovano in esecuzione nel sistema, causandone *starvation*, per questo un processo può assegnare una politica di scheduling RT ad uno dei suoi thread soltanto se viene eseguito con i privilegi dell'utente *root*.

Lo standard POSIX definisce in modo chiaro le proprietà che deve soddisfare uno scheduler di tipo *FIFO* e *RR*, quindi è lecito supporre che la schedulazione dei thread di un programma (interamente) RT sia riproducibile su differenti architetture *POSIX-compliant*. Al contrario le proprietà dello scheduler *OTHER* non sono specificate dallo standard POSIX, che le lascia a discrezione del sistema operativo che lo implementa. Per questa ragione la schedulazione dei thread di un programma non-RT deve essere considerata nel caso generale non deterministica e non riproducibile.

1.2 Assegnare priorità RT ad un thread C++:

Lo scheduling e la priorità statica di un thread POSIX fanno parte degli *attributi* del thread e possono essere manipolati mediante API apposite: assieme agli esercizi viene fornita una piccola libreria *wrapper* basata su di esse (`librt_pthread`) che ne consente un uso agevole in linguaggio C++.

```
#include "rt/priority.h"

class priority;

priority get_priority(const std::thread & th);
priority this_thread::get_priority();

void set_priority(std::thread & th, const priority & p);
void this_thread::set_priority(const priority & p);
```

La classe `priority`, dichiarata all'interno del namespace `rt`, permette di creare oggetti che rappresentano un possibile valore di priorità (e scheduler) assegnabile ad un thread. La libreria definisce le costanti seguenti:

- `priority::not_rt`: corrisponde ad uno scheduler non real-time (SCHED_OTHER in ambiente POSIX);
- `priority::rt_min` corrisponde ad uno scheduler real-time (SCHED_FIFO in ambiente POSIX) ed al relativo valore minimo di priorità disponibile nel sistema operativo corrente (in ambiente GNU/Linux è pari a 1);
- `priority::rt_max` corrisponde ad uno scheduler real-time (SCHED_FIFO in ambiente POSIX) ed al relativo valore massimo di priorità disponibile nel sistema operativo corrente (in ambiente GNU/Linux è pari a 99)

La classe `priority` possiede operatori logici di confronto che forniscono l'ordinamento $not\_rt < rt\_min < rt\_max$, sono inoltre definiti operatori matematici (incremento, decremento, ecc.) che permettono di manipolare le priorità all'interno del range `[not_rt,rt_max]`, ed infine sono disponibili le seguenti funzioni:

- `get_priority()` e `this_thread::get_priority()`: ritornano il valore di `priority` rispettivamente del thread che viene passato come parametro o del thread chiamante;
- `set_priority()` e `this_thread::set_priority()`: settano il valore di `priority` al thread che viene passato come parametro o al thread chiamante, rispettivamente; se il processo corrente non possiede i privilegi necessari per eseguire l'operazione, viene sollevata un'eccezione di tipo `permission_error`.

Esempio

Il codice seguente mostra un esempio di utilizzo della classe `priority`:

```
void f(int num)
{
  rt::priority my_prio(rt::this_thread::get_priority());

  std::this_thread::sleep_for(std::chrono::seconds(num));

  std::cout << "Thread " << num << " is executing at priority "
    << my_prio << "." << std::endl;
}

int main()
{
  std::thread t1(f, 1), t2(f, 2);

  try
  {
    rt::priority p1(rt::priority::rt_min + 50);
    rt::priority p2(p1 + 10);

    rt::set_priority(t1, p1);
    rt::set_priority(t2, p2);
  }
  catch (rt::permission_error & e)
  {
    std::cout << "Failed to set priority: " << e.what() << std::endl;
  }

  t1.join(); t2.join();
  return 0;
}
```

Esercizio 1

Il programma C++ contenuto nel file `hello.cpp` *tenta* di stampare la stringa `Hello World!` a video, delegando ad una serie di thread figli la scrittura di ogni singola lettera. Dal momento che ogni figlio prima di stampare la lettera che gli compete si mette in attesa sul metodo `wait()` dell'oggetto `barrier`, la sequenza delle lettere stampate dipende dall'ordine con cui i thread si risvegliano nell'istante in cui l'ultimo di essi invoca `wait()` sulla barriera.

Assegnare opportunamente una politica di scheduling real-time ai thread figli per fare in modo che il comportamento del programma diventi univoco (su una macchina *uniprocessore*) e la stringa stampata a video diventi corretta, senza introdurre ulteriori sincronizzazioni esplicite tra i thread.

1.3 Impostare l'affinità di un thread C++:

La libreria (`librt_pthread`) fornisce l'header file `affinity.h` che consente di modificare l'*affinità* di un thread, cioè la possibilià di limitare l'esecuzione di un thread ad un sottoinsieme dei *core* disponibili nell'hardware corrente.

Purtroppo non esistono API *POSIX* per controllare questa funzionalità, però in ambiente GNU/Linux esistono alcune estensioni allo standard (qualificate dal suffisso `_np`, cioè *Non Portable*) che permettono di farlo, pertanto l'implementazione corrente della libreria può funzionare soltanto in tale ambiente.

Il datatype `affinity`, dichiarato all'interno del namespace `rt` (come ri-definizione della classe `std::bitset<32>`), permette di creare oggetti utili per modificare l'affinità dei singoli thread, mediante le funzioni seguenti:

- `get_affinity()` e `this_thread::get_affinity()`: ritornano il valore di `affinity` rispettivamente del thread che viene passato come parametro o del thread chiamante;
- `set_affinity()` e `this_thread::set_affinity()`: settano il valore di `affinity` al thread che viene passato come parametro o al thread chiamante, rispettivamente.

Esercizio 1bis

Se il programma `hello.cpp` dell'esercizio precedente (in versione real-time) viene eseguito su di una macchina *multicore*, la sola manipolazione delle priorità dei thread non sarà sufficiente per ottenere il risultato corretto, ma occorre anche garantire che tutti i thread che scrivono la stringa di output vengano eseguiti sul medesimo core del processore.

Modificare il programma real-time ottenuto dal file `hello.cpp` affinchè i thread figli possano essere eseguiti solo sul medesimo core, e verificare che il funzionamento diventi corretto anche su un hardware *multiprocessore* e/o *multicore*.

1.4 Eseguire task real-time periodici

Durante la costruzione di un'applicazione real-time è estremamente comune incontrare la necessità di definire operazioni che dovranno essere ripetute con una periodicità ben definita e la cui precisione influenza l'efficienza e la correttezza dell'applicazione complessiva. Una applicazione di questo tipo può essere schematizzata come segue (pseudo codice):

```
void periodic_task()
{
  while (true)
  {
    core_task();
    wait_until_time();
  }
}
```

Il progetto accurato della parte di temporizzazione richiede di realizzare la parte di codice `wait_until_time` in modo che l'esecuzione di `core_task` avvenga nel rispetto della perioditicà richiesta, riducendone la variazione e la deriva nei limiti dell'hardware e del sistema operativo.

Per fare questo è opportuno progettare la temporizzazione in modo che il risultato sia indipendente dalla variabilità del tempo di esecuzione di `core_task`, utilizzando funzionalità del C++11 che consentono di sospendere l'esecuzione del thread corrente fino ad un istante temporale espresso in modo *assoluto* rispetto ad un *clock* di sistema:

```
template<class Clock, class D>
void this_thread::sleep_until(const chrono::time_point<Clock,D>& time);
```

Quando questo metodo viene invocato, il thread viene posto in attesa fino all'istante temporale `time`, specificato rispetto ad uno dei clock di sistema che sono stati già introdotti nell'esercitazione n.1. In questo specifico contesto, in cui è di primaria importanza garantire la massima stabilità possibile alla temporizzazione, sarà opportuno utilizzare il clock monotono `steady_clock` che, per definizione, garantisce tick di ampiezza costante tra loro.

Esercizio 2

Il programma C++ contenuto nel file `timing.cpp` realizza un'esecuzione periodica della funzione `do_some_stuff()` ma la temporizzazione risente della variazione del tempo di esecuzione della funzione, perciò si richiede di modificare il codice per ottenere una corretta temporizzazione basata su pause *assolute*, facendo uso della primitiva `sleep_until`.

Esercizio 2bis

Modificare il programma così ottenuto affichè l'esecuzione della funzione `do_some_stuff()` sia eseguita 8 volte e temporizzata secondo la successione di Fibonacci (1s 1s 2s 3s 5s 8s 13s 21s), senza risentire della variazione del tempo di esecuzione della funzione stessa.