Labo PRT 01

Denis Bourqui

1ères mesures

empty loop

```
[deni@silet ressources]$ ./loop
1'579'238'515
[deni@silet ressources]$ ./loop2
1'908'801'614
[deni@silet ressources]$ ./loop3
1'872'463'019
```

Division

```
[deni@silet ressources]$ ./div
nombre d iteration 693'268'451
[deni@silet ressources]$ ./div2
nombre d iteration 1'976'865'497
[deni@silet ressources]$ ./div3
nombre d iteration 1'786'456'252
[deni@silet ressources]$
```

Multiplication

```
[deni@silet ressources]$ ./mult
nombre d iteration 1'271'481'512
[deni@silet ressources]$ ./mult2
nombre d iteration 1'657'767'121
[deni@silet ressources]$ ./mult3
nombre d iteration 1'681'079'503
[deni@silet ressources]$
```

On voit que les optimisation de compilations arivent effectivement a accellerer la boucle en question. La difference entre l'optimation 2 et 3 son pas significative.

Commande PS

```
 Here are the different values that the s, stat and state output specifiers
(header
       "STAT" or "S") will display to describe the state of a process:

               D    uninterruptible sleep (usually IO)
               I    Idle kernel thread
               R    running or runnable (on run queue)
               S    interruptible sleep (waiting for an event to complete)
               T    stopped by job control signal
               t    stopped by debugger during the tracing
               W    paging (not valid since the 2.6.xx kernel)
               X    dead (should never be seen)
               Z    defunct ("zombie") process, terminated but not reaped by its
parent
```

```
   For BSD formats and when the stat keyword is used, additional characters may
be
   displayed:

           <    high-priority (not nice to other users)
           N    low-priority (nice to other users)
           L    has pages locked into memory (for real-time and custom IO)
           s    is a session leader
           l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
           +    is in the foreground process group
```

Ordonnancement en temps partagé

En forcant l'execution du BASH sur un seul processeur

```
[deni@silet ressources]$ ./loop
152'276'612 cycles/s 23921


[deni@silet ressources]$ ./loop & ./loop
[2] 24100
93'822'784 cycles/s 24101
94'034'256 cycles/s 24100
[2]+  Done                    ./loop


[deni@silet ressources]$ ./loop & ./loop & ./loop
[2] 24493
[3] 24494
97'566'535 cycles/s 24494
98'324'732 cycles/s 24495
[3]+  Done                    ./loop
97'566'535 cycles/s 24493
[2]+  Done                    ./loop


[deni@silet ressources]$ ./loop & ./loop & ./loop & ./loop
[2] 24771
[3] 24772
[4] 24773
```

```
81'554'851 cycles/s 24773
82'370'344 cycles/s 24774
82'638'085 cycles/s 24772
82'003'700 cycles/s 24771
[4]+  Done                    ./loop
```

Vu que ces programmes doivent tous tourner sur le même processeur, on voit que plus de programme qu'il y a, moins de cycles par secondes sont fait.

Pour comparer: la commande `./loop & ./loop & ./loop` arrive à 200 million de cycles sur un BASH qui peux lancer des processus sur plusieurs processeur.

2 Processus BASH sur le même processeur

```
BASH 1
[deni@silet ressources]$ sleep 1 & ./loop & ./loop
[2] 35955
[3] 35956
96'240'096 cycles/s 35956
95'607'231 cycles/s 35957
[2]-  Done                    sleep 1

BASH 2
[deni@silet ressources]$ ./loop && ./loop
151'872'241 cycles/s 35958
346'553'773 cycles/s 36029
```

Les programmes du BASH 2 sont plus rapide.

Exécutez ensuite, à l’aide de la commande taskset précédemment vue, le logiciel cpu_loop sur le même CPU où get_cpu_number tourne. Qu’est-ce qu’il se passe ?

On voit que dès qu'on démarre les loops (à la seconde ...850) le processus change de processeur (de n°3 au n°2).

```
 ./get_cpu_number
on processor 3 at time 1600949831
on processor 2 at time 1600949850
on processor 0 at time 1600949850
on processor 1 at time 1600949850
on processor 0 at time 1600949852
on processor 1 at time 1600949854
```

Nice

Si on lance une de deux loops avec nice, on voit que cela a moins de cycles par secondes que l'autre car il lui donne plus souvent la priorité.

```
nice -n 5 ./loop & ./loop
[1] 71971
246857811 cycles/s 71972
80521034 cycles/s 71971
```

La somme des nombres de cycles par secondes reste environ la même comme si il n'y avait pas de `nice`. Seul la répartition diffère.

Codage

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <inttypes.h>
#include <errno.h>
#define EXECUTION_TIME 10 /* In seconds */

/* Barrier variable */
pthread_barrier_t barr;
time_t start_time;
void *f_thread(void *arg)
{
    pthread_barrier_wait(&barr);
    unsigned long nb_iterations = 0;
    while (time(NULL) < (start_time + EXECUTION_TIME)) {
        ++nb_iterations;
    }
    *((unsigned long*) arg) = nb_iterations;
}

int main(int argc, char **argv)
{
    int nb_threads;
    int min_prio;
    int max_prio;
    int i;
    int *prio_value;
    unsigned long *nb_iterations;
    unsigned long total_iterations = 0;
    pthread_t *threads;

    /* Parse input */
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s NB_THREADS\n", argv[0]);
        return EXIT_FAILURE;
    }
    nb_threads = strtoimax(argv[1], (char **)NULL, 10);
    if (nb_threads <= 0)
    {
        fprintf(stderr, "NB_THREADS must be > 0 (actual: %d)\n", nb_threads);
        return EXIT_FAILURE;
    }
    /* memory allocation */
```

```
    prio_value = (int *)calloc(nb_threads, sizeof(int));
    nb_iterations = (long *)calloc(nb_threads, sizeof(unsigned long));
    threads = (long *)calloc(nb_threads, sizeof(pthread_t));

    /* Get minimal and maximal priority values */
    min_prio = sched_get_priority_min(SCHED_FIFO);
    max_prio = sched_get_priority_max(SCHED_FIFO);
    max_prio -= min_prio;
    fprintf(stdout, "minimal prio: %d, maximal prio: %d\n", min_prio, max_prio);

    /* Initialize barrier */
    if (pthread_barrier_init(&barr, NULL, nb_threads))
    {
        fprintf(stderr, "Could not initialize barrier!\n");
        return EXIT_FAILURE;
    }

    start_time = time(NULL);

    /* Set priorities and create threads */
    for(i = 0; i < nb_threads; ++i){

        prio_value[i] =  ((max_prio + min_prio) / nb_threads) * i + min_prio + 1;
        fprintf(stdout, "new thread init with prio %d\n", prio_value[i]);


        /* Set thread attributes necessary to use priorities */
        struct sched_param sched_param;
        pthread_attr_t thread_attr;
        int policy;



        if(pthread_attr_init(&thread_attr)){
            fprintf(stderr, "Error on pthread_attr_init\n");
        }

        if(pthread_attr_getschedparam(&thread_attr, &sched_param)){
            fprintf(stderr, "Error on pthread_attr_getschedparam\n");
        }

        if(pthread_attr_setschedpolicy(&thread_attr, SCHED_FIFO)){
            fprintf(stderr, "Error on pthread_attr_setschedpolicy\n");
        }

        if(pthread_attr_setinheritsched(&thread_attr, PTHREAD_EXPLICIT_SCHED)){
            fprintf(stderr, "Error on pthread_attr_setinheritsched\n");
        }

        sched_param.sched_priority = prio_value[i];

        if(pthread_attr_setschedparam(&thread_attr, &sched_param)){
            fprintf(stderr, "Error on pthread_attr_setschedparam with prio value
%d\n", sched_param.sched_priority );
        }

        if(pthread_create(threads+i, &thread_attr, f_thread, (void *)
(nb_iterations+i))){
```

```
            fprintf(stderr, "Error on pthread_create\n");
        }

        if(pthread_getschedparam(threads[i], &policy , &sched_param)){
            fprintf(stderr, "Error on pthread_getschedparam\n");
        }
        policy = SCHED_FIFO;
        if(pthread_setschedparam(threads[i], policy, &sched_param)){
            fprintf(stderr, "Error on pthread_setschedparam\n");
        }
        switch(pthread_setschedprio(threads[i], prio_value[i])){
            case EINVAL:
                fprintf(stderr, "prio is not valid for the scheduling policy of
the specified thread.\n");
                break;
            case EPERM:
                fprintf(stderr, "The caller does not have appropriate privileges
to set the specified priority.\n");
                break;
            case 0:
                fprintf(stdout, "created thread %d with prio %d, policy: %d\n",
i, prio_value[i], policy);
        }

    }

    fprintf(stdout, "all thread started\n");
    /* Wait for the threads to complete and set the results. beginning back of
the threads array. (from high prio to low prio thread) */
    for(i = nb_threads -1 ; i >= 0; --i){
        fprintf(stdout, "waiting on thread number %d\n", i);
        pthread_join(threads[i], NULL);
    }

    fprintf(stdout, "All threads ended in %d seconds\n",time(NULL) -  start_time
);
    /* summing iterations */
    for(i = 0; i < nb_threads; ++i){
        total_iterations += nb_iterations[i];
    }

    for (i = 0; i < nb_threads; ++i)
    {
        fprintf(stdout, "[%02d] %ld (%2.0f%%)\n",
                prio_value[i], nb_iterations[i],
                100.0 * nb_iterations[i] / total_iterations);
    }
    getchar();
    return EXIT_SUCCESS;
}
```

Difference entre Nice et Priorités

Nice travaille seulement dans le userspace alors que les priorités agissent dans le kernel space.

Les priorités peuvent prendre des nombres plus extrême que nice.