ING4 SI03

Operating Systems

Lab 3 : Threads

Author(s) :
Romain BRISSE
Hyunjae LEE

Teacher(s) :
Mr. KHOURY

I attest that this work completely comes from the author(s) mind(s). If it is otherwise you will be able to find the source in the concerned section.

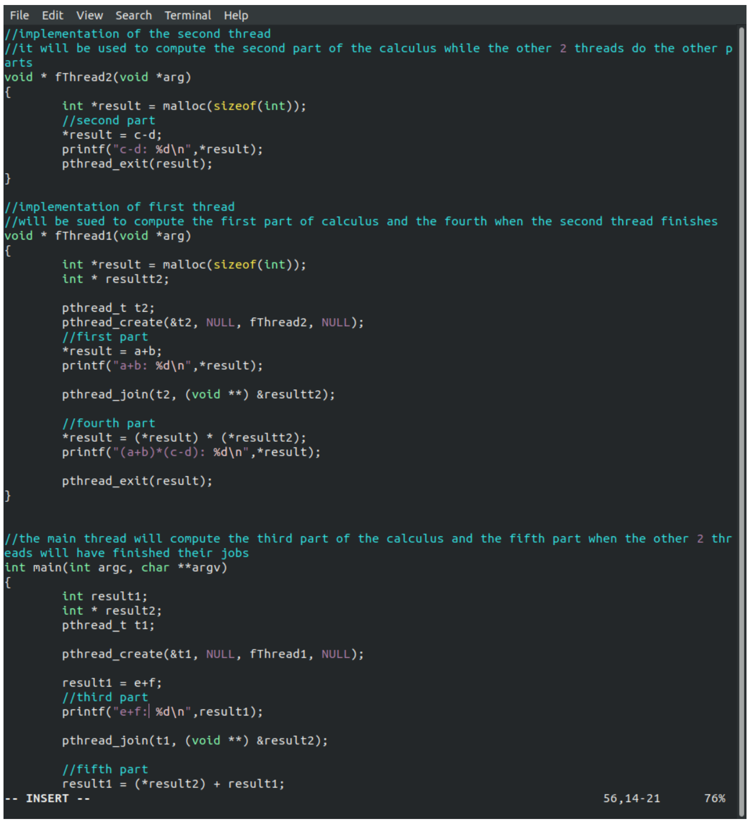
Paris, 18/10/2018

LaTeX

Table des matières

0.1 Parallel computing with threads

My code



```
//implementation of the second thread
//it will be used to compute the second part of the calculus while the other 2 threads do the other p
arts
void * fThread2(void *arg)
{
        int *result = malloc(sizeof(int));
        //second part
        *result = c-d;
        printf("c-d: %d\n",*result);
        pthread_exit(result);
}

//implementation of first thread
//will be sued to compute the first part of calculus and the fourth when the second thread finishes
void * fThread1(void *arg)
{
        int *result = malloc(sizeof(int));
        int * resultt2;

        pthread_t t2;
        pthread_create(&t2, NULL, fThread2, NULL);
        //first part
        *result = a+b;
        printf("a+b: %d\n",*result);

        pthread_join(t2, (void **) &resultt2);

        //fourth part
        *result = (*result) * (*resultt2);
        printf("(a+b)*(c-d): %d\n",*result);

        pthread_exit(result);
}


//the main thread will compute the third part of the calculus and the fifth part when the other 2 thr
eads will have finished their jobs
int main(int argc, char **argv)
{
        int result1;
        int * result2;
        pthread_t t1;

        pthread_create(&t1, NULL, fThread1, NULL);

        result1 = e+f;
        //third part
        printf("e+f: %d\n",result1);

        pthread_join(t1, (void **) &result2);

        //fifth part
        result1 = (*result2) + result1;
-- INSERT --                                                                          56,14-21      76%
```

FIGURE 1 – parallel computing using threads

Explanation

Using threads this way, I was able to achieve parallel computing.
The main threads creates and runs the third part of the calculus and a new thread that will herself execute the third part of calculus and create another thread that will finally execute the second part of the calculus.
Once this step is done, the first thread executes the fourth part of the calculus while the main thread waits for it. When it has finished, the main thread executes the last part of the calculus that gets us the final result.

0.2 Performance evaluation

0.2.1 The times function

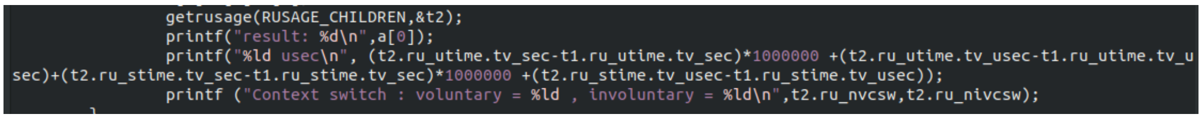
The times function is a C function used to "get process times". It means that the function stores the CPU time spent executing the active process.
However in this case I will use the function getrusage() in order to be able to include the children of the active program in my time count.

0.2.2 The getrusage function

The getrusage function is used to get the resource usage from a program, at a given instant. That is why I am able to get the time as it is also a resource (and an important one at that !) and the different context switches that happened for example.

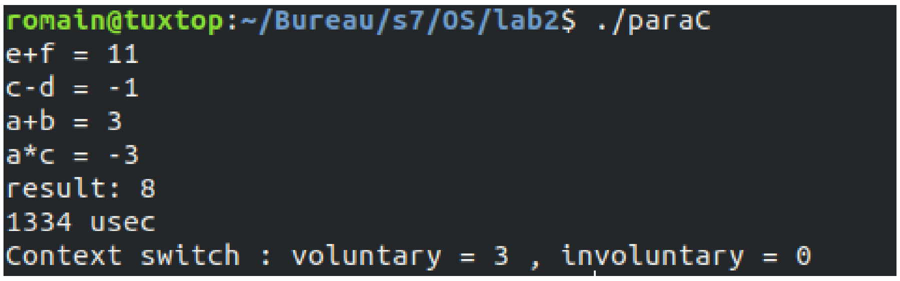
0.2.3 Performance evaluation of parallel computing

Using Processes



```
                getrusage(RUSAGE_CHILDREN,&t2);
                printf("result: %d\n",a[0]);
                printf("%ld usec\n", (t2.ru_utime.tv_sec-t1.ru_utime.tv_sec)*1000000 +(t2.ru_utime.tv_usec-t1.ru_utime.tv_u
sec)+(t2.ru_stime.tv_sec-t1.ru_stime.tv_sec)*1000000 +(t2.ru_stime.tv_usec-t1.ru_stime.tv_usec));
                printf ("Context switch : voluntary = %ld , involuntary = %ld\n",t2.ru_nvcsw,t2.ru_nivcsw);
```

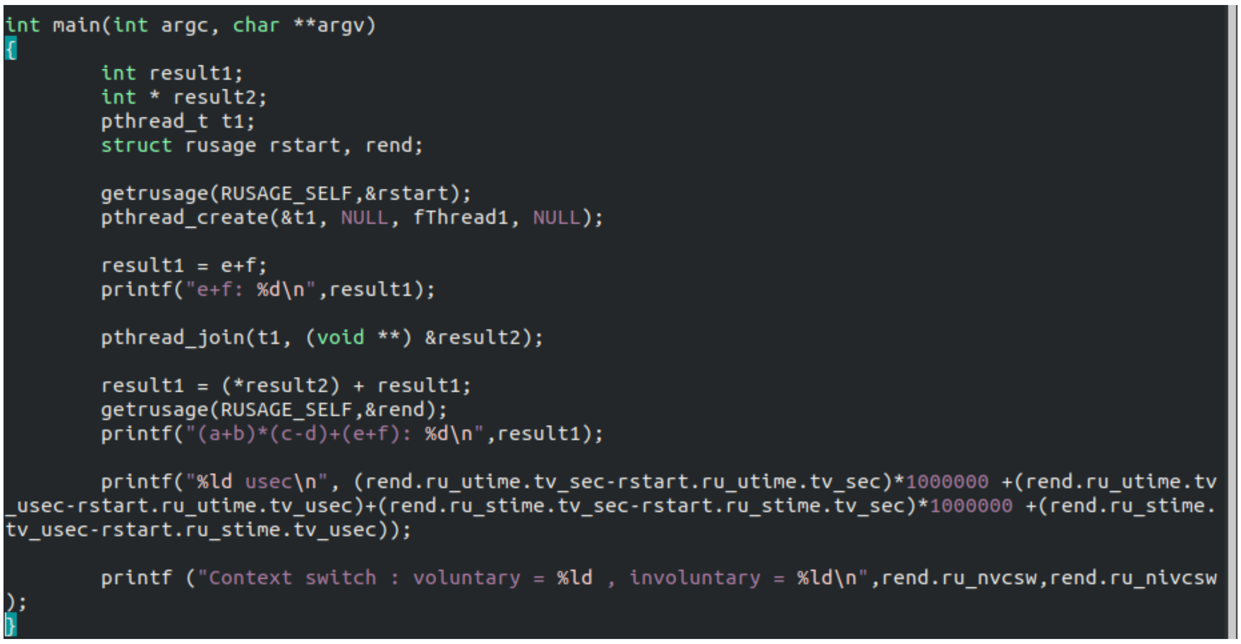
FIGURE 2 – Evaluating time in the program using threads



```
romain@tuxtop:~/Bureau/s7/OS/lab2$ ./paraC
e+f = 11
c-d = -1
a+b = 3
a*c = -3
result: 8
1334 usec
Context switch : voluntary = 3 , involuntary = 0
```

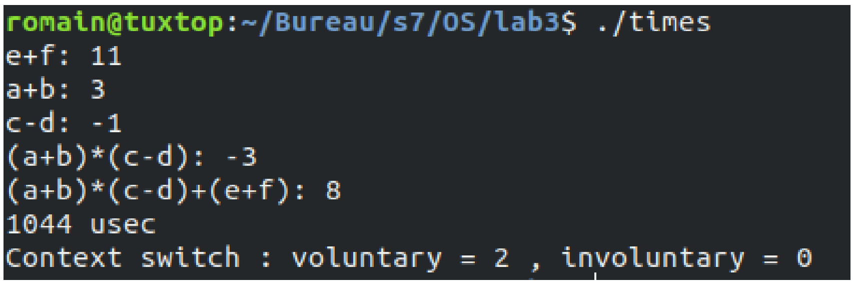
FIGURE 3 – the obtained result

Using Threads



```
int main(int argc, char **argv)
{
        int result1;
        int * result2;
        pthread_t t1;
        struct rusage rstart, rend;

        getrusage(RUSAGE_SELF,&rstart);
        pthread_create(&t1, NULL, fThread1, NULL);

        result1 = e+f;
        printf("e+f: %d\n",result1);

        pthread_join(t1, (void **) &result2);

        result1 = (*result2) + result1;
        getrusage(RUSAGE_SELF,&rend);
        printf("(a+b)*(c-d)+(e+f): %d\n",result1);

        printf("%ld usec\n", (rend.ru_utime.tv_sec-rstart.ru_utime.tv_sec)*1000000 +(rend.ru_utime.tv
_usec-rstart.ru_utime.tv_usec)+(rend.ru_stime.tv_sec-rstart.ru_stime.tv_sec)*1000000 +(rend.ru_stime.
tv_usec-rstart.ru_stime.tv_usec));

        printf ("Context switch : voluntary = %ld , involuntary = %ld\n",rend.ru_nvcsw,rend.ru_nivcsw
);
}
```

FIGURE 4 – Evaluating time in the program using threads



```
romain@tuxtop:~/Bureau/s7/OS/lab3$ ./times
e+f: 11
a+b: 3
c-d: -1
(a+b)*(c-d): -3
(a+b)*(c-d)+(e+f): 8
1044 usec
Context switch : voluntary = 2 , involuntary = 0
```

FIGURE 5 – the obtained result

Observations

As the previous screenshots can attest. Using threads takes less CPU time to compute the same calculus. Even if it is by only 300 microseconds, the reason for this difference is probably that switching between processes forces the CPU to replace the entire file descriptor grid whereas switching between threads does not.
In these screenshots the difference between the number of voluntary context switches is only due to a slightly different way of programming. (Using the main as a thread) There are also no involuntary context switches, simply because the tasks given to the different processes or threads are far inferior to the quantum time of the scheduler and so to make a forced context switch happen.