RTAES - Threads e prioridades (SCHED_FIFO)

António Barros, Cláudio Maia

Dezembro 2022

Objectivos

Os objectivos deste conjunto de actividades são os seguintes:

1. Implementação de um programa multi-thread, com a definição de uma tarefa periódica.
2. Atribuição de prioridades específicas a cada thread.
3. Observação da execução concorrente das threads de um processo, sob a política de escalonamento SCHED_FIFO.

1 Introdução

Por definição, todos os programa implementado na linguagem de programação C começa a sua execução na função principal `int main()`. Essa *linha* (*thread*) de execução é a primeira e, muitas vezes de um programa. Mas, a norma POSIX permite a existência de linhas de execução paralelas e autónomas num processo. Cada linha segue o seu percurso, de forma autónoma das outras linhas – com o seu *program counter* e *stack* específicos – até, eventualmente, terminar.

Uma thread termina quando a função que define retorna (i.e. "chega ao fim"), ou quando é chamada explicitamente a função `pthread_exit()`. Pode consultar a página do manual

```
$ man 3 pthread_exit
```

Deve ter em atenção que um processo termina em uma de duas situações:

1. a função principal chega ao fim e retorna normalmente, ou
2. a última thread viva do processo termina.

No caso 1., o retorno da função principal causa a terminação de todas as threads que possam estar a executar. A melhor forma de evitar esta situação (se for desejável), é terminar a thread principal com a chamada da função `pthread_exit()`, em vez de permitir um retorno normal.

```
int main()
{
  /* (...) */
  pthread_exit(NULL);
  return 0;
}
```

1.1 Criar uma thread

Com excepção da primeira thread, cada thread adicional tem que ser criada explicitamente com a função `pthread_create()`. A explicação detalhada desta função pode ser encontrada na página do manual

```
$ man 3 pthread_create
```

1.2 Definir a prioridade de uma thread

A prioridade de uma thread é definida através da função `pthread_setschedparam()`. Deve consultar a página do manual para mais detalhes sobre esta função

```
$ man 3 pthread_setschedparam
```

1.3 Definir a afinidade da thread a núcleos de processamento específicos

A afinidade de uma thread é definida através da função `pthread_setaffinity_np()`. Deve consultar a página do manual para mais detalhes sobre esta função

```
$ man 3 pthread_setaffinity_np
```

1.4 Suspender uma thread até ao próximo instante de activação

Para se implementar uma thread periódica, é necessário estabelecer um mecanismo para activar a thread no instante de activação seguinte. A função `clock_nanosleep()` define um temporizador que irá disparar no instante indicado na sua chamada. Para obter detalhes sobre esta função, consulte a página de manual

```
$ man 2 clock_nanosleep
```

2 Implementar um programa multi-thread

No programa seguinte, a função `void tarefa_A(void *arg)` define o modelo de uma thread que simula uma tarefa periódica.

Na função principal, são criadas as tarefas, cada uma com o seu conjunto de atributos específicos.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sched.h>

#define NUM_THREADS 3
#define TARGET_CORE 2

struct taskargs {
    struct timespec period;
    int priority;
    int number_of_jobs;
    int dummy_cycles;
};

void * periodic_task(void *arg) {
    struct taskargs * targs;
    struct timespec next, period;
    int i, dummy_cycles;
    int job, number_jobs;
    int priority;
    struct sched_param params;
    cpu_set_t cores_mask;

    /* Read parameters from arg. */
    targs = (struct taskargs *) arg;
    period = targs->period;
    priority = targs->priority;
    number_jobs = targs->number_of_jobs;
    dummy_cycles = targs->dummy_cycles;
```

```
32
33      /* Set affinity of thread to cores -> {0, 1, 2, 3} */
34      CPU_ZERO(&cores_mask);
35      CPU_SET(TARGET_CORE, &cores_mask);
36      pthread_setaffinity_np(pthread_self(), sizeof(cores_mask), &
        cores_mask);
37
38      /* Set thread priority under SHED_FIFO scheduler. */
39      params.sched_priority = priority;
40      pthread_setschedparam(pthread_self(), SCHED_FIFO, &params);
41
42      /* Sleep for 100 microseconds, to allow other threads to modify
        their priorities. */
43      usleep(100);
44
45      /* Read current time. */
46      clock_gettime(CLOCK_MONOTONIC, &next);
47
48      for (job = 1; job <= number_jobs; job++) {
49          /* Do some work, burn some clock cycles. */
50          for (i = 0; i < dummy_cycles; i++) {
51              getpid();
52          }
53
54          /* Set timer for next activation. */
55          next.tv_sec += period.tv_sec;   /* Seconds */
56          next.tv_nsec += period.tv_nsec; /* Nanoseconds*/
57          if(next.tv_nsec > 1000000000L) {
58            /* Nanoseconds can not exceed one second. */
59              next.tv_nsec -= 1000000000L;
60              next.tv_sec++;
61          }
62          clock_nanosleep(CLOCK_MONOTONIC,
63                          TIMER_ABSTIME, &next, 0);
64      }
65  }
66
67
68  int main()
69  {
70      struct taskargs targs[NUM_THREADS];
71      pthread_t threads[NUM_THREADS];
72      int i;
73
74      /* HIGH priority task. */
75      targs[0].period.tv_sec = 0;
76      targs[0].period.tv_nsec = 50000000; /* 50 ms */
77      targs[0].priority = 3;
78      targs[0].number_of_jobs = 20;
79      targs[0].dummy_cycles = 10000; /* ---> 5 ms */
80
81      /* LOW priority task. */
82      targs[1].period.tv_sec = 0;
83      targs[1].period.tv_nsec = 500000000; /* 500 ms */
84      targs[1].priority = 1;
85      targs[1].number_of_jobs = 2;
86      targs[1].dummy_cycles = 500000; /* ---> 260 ms */
87
```

```
88      /* INTERMEDIATE priority task. */
89      targs[2].period.tv_sec = 0;
90      targs[2].period.tv_nsec = 100000000; /* 100 ms */
91      targs[2].priority = 2;
92      targs[2].number_of_jobs = 10;
93      targs[2].dummy_cycles = 10000; /* --->  5 ms */
94
95      /* Create three concurrent threads. */
96      for(i = 0; i < NUM_THREADS; i++) {
97          pthread_create(&threads[i], NULL, periodic_task, &targs[i]);
98      }
99
100     /* Wait for the 3 threads to complete their executions. */
101     for (i = 0; i < NUM_THREADS; i++) {
102         pthread_join(threads[i], NULL);
103     }
104
105     return 0;
106 }
```

Neste programa, a tarefa de mais baixa prioridade tem o maior período ($500ms$) e cada trabalho executa um ciclo com 500 mil iterações, o que resulta aproximadamente em $260ms$, ou seja, uma utilização ligeiramente superior a 50%. Note, no entanto, que esta associação número de iterações/tempo do trabalho pode variar de sistema para sistema; só é possível determinar com maior precisão através dos traços de execução em cada sistema específico.

Compile este programa, indicando para incluir a biblioteca `pthread`

```
1 $ gcc periodic.c -lpthread -o periodic
```

Execute o programa com o `trace-cmd` a registar os eventos de escalonamento, e observe o registo com o `kernelshark`.

```
1 $ sudo trace-cmd record -e sched_switch
```

Verifique se o traço de execução das threads está de acordo com o previsto, dadas as prioridades atribuídas. Tente também estabelecer uma relação entre o número de iterações do ciclo realizadas por um trabalho e o seu tempo de execução[1].

Observe se existem alguns momentos em que as threads são interrompidas (qualquer que seja a sua prioridade) para permitir ao kernel executar alguma operação (processo `kworker`).

Verifique que nos momentos em que nenhuma tarefa tem trabalho para executar, o kernel escalona o processo `swapper` que coloca o processador em inactividade (*idle*).

Pode variar os atributos das threads, para observar as possíveis alterações do traço resultante.

[1] Sugere-se que faça esta medição na tarefa de maior prioridade.