RTAES - Escalonamento de processos

António Barros, Cláudio Maia

Outubro 2022

Objectivos

Os objectivos deste conjunto de actividades são os seguintes:

1. Implementar um programa em C que solicita ao SO a modificação do seu modo de escalonamento.
2. Executar esse programa e observar a forma como é escalonado.
3. Lançar múltiplos processos concorrentes desse programa, e observar o trabalho do escalonador.

1 Introdução

1.1 Escalonadores em Linux

Por pré-definição, o Linux organiza a execução de processos utilizando o *Completely Fair Scheduler* (CFS). Cada novo processo que chega ao sistema é automaticamente escalonado por este escalonador. O CFS atribui uma fatia de tempo considerada "justa", de cada vez que selecciona um processo para executar, e funciona muito bem para processos interactivos, sem restrições temporais exigentes.

Para os processos que têm restrições temporais mais exigentes, Linux dispõe de outros escalonadores denominados de *real-time*. Na realidade, estes escalonadores não proporcionam as garantias necessárias para sistemas de tarefas *hard real-time*; no entanto são suficientes para sistemas de tarefas *soft real-time*.

Os escalonadores *real time* são:

`SCHED_FIFO` Este escalonador selecciona o processo mais antigo na fila de mais alta prioridade para executar. Cada prioridade tem uma fila que contém os processos da mesma prioridade ordenados por ordem cronológica de chegada.

`SCHED_RR` Este escalonador atribui uma fatia de tempo a cada processo seleccionado para executar e vai rodando os processos da mesma prioridade.

`SCHED_DEADLINE` Este escalonador selecciona para execução o processo que tem o prazo (*deadline*) mais próximo.

Estes escalonadores implementam filas por níveis de prioridades. Isto significa que seleccionam sempre o processo à cabeça da fila de maior prioridade que não estiver vazia. A norma POSIX determina que existam, no mínimo 32 níveis de prioridades, mas a implementação do Linux disponibiliza desde a prioridade 1 (mais baixa) até à prioridade 99 (mais alta).

Para a implementação das estratégias de escalonamento *Rate Monotonic* (RM) e *Deadline Monotonic* (DM) é utilizado o escalonador `SCHED_FIFO`:

- em **RM** as prioridades são atribuídas na razão inversa do período (i.e. menor período, maior prioridade);
- em **DM** as prioridades são atribuídas na razão inversa do prazo relativo (i.e. menor prazo relativo, maior prioridade).

Para a implementação da estratégia de escalonamento *Earliest Deadline First* (EDF) é utilizado o escalonador `SCHED_DEADLINE`.

1.2 Afinidade a processadores

O Linux disponibiliza a funcionalidade de definir o conjunto de processadores (ou núcleos de processamento) em que cada processo pode executar. Esta funcionalidade está implementada na forma de extensões às funcionalidades de escalonamento, ou seja, não faz parte da norma POSIX. Consequentemente, um programa que utilize estas extensões não garante que possa ser portado para outros sistemas não-Linux.

O conjunto de processadores em que um processo pode executar é representado através de uma máscara de bits, em que cada posição representa um processador. O valor do bit numa posição determina se o processo pode ou não ser executado no processador associado: 0 significa que não pode ser executado nesse processador, 1 significa que pode ser executado nesse processador. Por pré-definição, um processo pode ser escalonado em qualquer processador (escalonamento global).

A alteração da afinidade do processo passa por construir uma máscara em que se activam (i.e. coloca-se 1) as posições referentes aos processadores. A API do Linux disponibiliza um conjunto de macros que devem ser utilizadas para garantir a correcta operação de uma máscara de afinidade: `CPU_ZERO`, `CPU_SET` e `CPU_CLEAR`.

2 Preparação

2.1 Fontes de documentação técnica básica

A documentação básica encontra-se nas páginas do manual do Linux. Os tópicos principais desta aula (chamadas ao sistema associadas ao escalonador) encontram-se descritas na página do manual `sched(7)`. Este é um ponto de partida para a descoberta.

```
$ man sched
```

2.2 Obter detalhes da arquitectura da plataforma

Deve conhecer os detalhes do processador que vai utilizar, sobretudo o número de núcleos de processamento (i.e. *cores*). Pode obter esta informação executando o comando `lscpu`:

```
$ lscpu
```

Pode obter informação mais detalhada sobre cada um dos núcleos de processamento consultando o ficheiro `/proc/cpuinfo`:

```
$ cat /proc/cpuinfo
```

O Raspberry Pi 4 tem quatro núcleos de processamento, identificados com os números 0 a 3.

3 Modificar o modo de escalonamento

Em Linux, um novo processo é colocado automaticamente na fila do escalonador de processos "normais": o CFS. No entanto é possível alterar o modo de escalonamento utilizando a chamada ao sistema `sched_setscheduler()`, passando como argumentos *(i)* o PID do processo, *(ii)* a política de escalonamento a seguir e *(iii)* a nova prioridade do processo.

Analise e depois compile o seguinte programa no Raspberry Pi, gerando o executável `exercicio1`.

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

int main()
{
    struct sched_param param;

    param.sched_priority = sched_get_priority_max(SCHED_FIFO);

```

```
    sched_setscheduler(getpid(), SCHED_FIFO, &param);

    while (1) {
        printf("[%d] I am here!\n", getpid());
        sleep(5);
    }
}
```

Altere o programa de forma a que a prioridade possa ser indicada na linha de comando. Por exemplo, se se pretender que o processo tenha a prioridade 15, o programa deverá ser chamado da seguinte forma:

```
$ ./exercicio1 15
```

Não se esqueça de verificar que a prioridade indicada é válida: *(i)* é um valor numérico e *(ii)* se encontra entre os valores mínimo e máximo para o escalonador `SCHED_FIFO`.

4 Definir a afinidade aos núcleos de execução

O Linux tem uma extensão ao POSIX que permite definir o conjunto de processadores em que um processo pode executar. Para tal, utiliza-se a chamada ao sistema `sched_setaffinity()` que requer como argumentos *(i)* o PID do processo, *(ii)* o tamanho da máscara de bits que representa o conjunto dos processadores e *(iii)* o endereço onde a máscara de bits se encontra na memória.

Analise e depois compile o seguinte programa no Raspberry Pi, gerando o executável `exercicio2`.

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

int main()
{
    cpu_set_t mask;

    /* Set affinity to the 2nd core -> {0, 1, 2, 3} */
    CPU_ZERO(&mask);
    CPU_SET(1, &mask);

    if(sched_setaffinity(getpid(), sizeof(cpu_set_t), &mask) == -1){
        perror("set_affinity: ");
    }

    while (1) {
        printf("[%d] I am here!\n", getpid());
        sleep(5);
    }
}
```

Analise o seguinte programa que vem como exemplo na página de manual `sched_setaffinity(2)`.

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE);} while (0)

int main(int argc, char *argv[])
{
```

```
12      cpu_set_t set;
13      int parentCPU, childCPU;
14      int nloops;
15
16      if (argc != 4) {
17          fprintf(stderr, "Usage: %s parent-cpu child-cpu num-loops\n",
18                          argv[0]);
19          exit(EXIT_FAILURE);
20      }
21
22      parentCPU = atoi(argv[1]);
23      childCPU = atoi(argv[2]);
24      nloops = atoi(argv[3]);
25
26      CPU_ZERO(&set);
27
28      switch (fork()) {
29      case -1:            /* Error */
30          errExit("fork");
31
32      case 0:             /* Child */
33          CPU_SET(childCPU, &set);
34
35          if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
36              errExit("sched_setaffinity");
37
38          for (int j = 0; j < nloops; j++)
39              getppid();  /* Just burning some time... */
40
41          exit(EXIT_SUCCESS);
42
43      default:            /* Parent */
44          CPU_SET(parentCPU, &set);
45
46          if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
47              errExit("sched_setaffinity");
48
49          for (int j = 0; j < nloops; j++)
50              getppid();  /* Also just burning some time... */
51
52          wait(NULL);     /* Wait for child to terminate */
53          exit(EXIT_SUCCESS);
54      }
55  }
```

Assim que tiver compreendido o que este programa faz, compile-o no Raspberry Pi, gerando o executável `exercicio3`. Execute o programa duas vezes:

- na primeira vez coloque os dois processos a executar no mesmo núcleo de processamento,
- na segunda vez coloque os dois processos a executar em núcleos de processamento distintos.

Em ambas as execuções, utilize o utilitário `time` para medir os tempos de execução (por exemplo):

```
$ time -p ./exercicio3 1 1 100000000
   (...)
$ time -p ./exercicio3 2 3 100000000
   (...)
```

Observe os resultados e determine uma possível razão para as diferenças.