Sistemas Operativos

Projecto – Fase Final
(Escalonamento e Comunicação entre Processos)

Departamento de Ensino e Investigação de Ciências da Computação
Faculdade de Ciências
Universidade Agostinho Neto

Parte I – Escalonamento de tarefas

Objectivo

A biblioteca jcthreads é uma biblioteca de pseudo-tarefas. O jcthreads é uma extensão da biblioteca sthreads criado pelo Rick Cox (rick@cs.washington.edu).

A biblioteca sthreads é fornecida aos estudantes uma versão desta biblioteca que suporta a criação de tarefas e o seu escalonamento, usando uma política de tempo partilhado e uma lista circular (round-robin).

Pretende-se estender a biblioteca por forma a substituir a política de escalonamento fornecida por uma política de escalonamento semelhante à da versão 2.6.23 do núcleo do Linux.

Contexto

A partir da versão 2.6.23 do kernel o sistema operativo Linux passou a utilizar um algoritmo de escalonamento denominado *Completely Fair Schedule* (CFS). No CFS cada processo está ordenado pelo tempo de CPU que lhe foi atribuído (*vruntime*). Os processos com menos tempo de CPU são os primeiros a serem escalonados para o CPU. Quando um processo é criado é lhe atribuído um tempo de CPU fictício igual ao menor dos tempos de execução dos processos ainda em funcionamento. Deste modo, os processos recém-criados têm de imediato oportunidade de se executar, pois têm menos tempo de execução que todos os outros, no entanto, não têm a prioridade que teriam se lhes fosse atribuído o tempo de execução real (o tempo de execução de um processo recém criado é zero). Se tal acontecesse os processos de longa duração seriam sempre preferidos relativamente aos de curta duração. De cada vez que o escalonador se executa o *vruntime* do processo em execução é incrementado do valor em que esteve em execução, e caso o novo valor de *vruntime* seja maior que o menor valor de *vruntime* dos outros processos, o processo é retirado de execução.

A novidade deste algoritmo é que só é relevante, para o escalonamento, o tempo que um processo está em execução; o tempo que ele esteve em espera bloqueado num recurso ou que

esteve em espera na fila do escalonador são indistinguíveis. Como os processos interactivos vão estar frequentemente bloqueados em recursos o seu tempo de execução vai ser baixo pelo que, quando saírem do bloqueio, irão naturalmente ter prioridade sobre os outros.

O CFS usa apenas uma estrutura ordenada de processos por escalonar, mas de modo a evitar o problema de escalabilidade da gestão das filas da versão 2.4, o CFS usa uma árvore red-black para manter todos os processos ordenados por tempo de execução, garantindo que as operações de inserção e remoção de processos se conseguem efectuar em O(log(n)).

No CFS os processos mais prioritários são beneficiados na medida em que o tempo de execução adicionado à variável vruntime em cada execução do despacho é apenas uma fracção do tempo realmente executado. Por outro lado aos processos marcados com nice é adicionado a seu *vruntime* um valor superior ao valor de execução real (mais detalhes na secção seguinte).

O CFS não possui prioridades de "tempo real", i.e. processos que são sempre escalonados prioritariamente independentemente do seu tempo de execução. Os processos com prioridades deste tipo são geridos por outro escalonador. De facto, a partir da versão 2.6.23 o sistema operativo Linux passou a possuir um meta-escalonador que gere uma lista de escalonadores e que escolhe qual deles irá escolher o processo a executar. A política deste meta-escalonador é muito simples: os vários escalonadores são mantidos numa lista ordenada por prioridade, e os processos dos escalonadores de menor prioridade só são escolhidos se não existirem processos activos nos escalonadores de maior prioridade. Como o escalonador dos processos de "tempo real", possui uma prioridade superior ao CFS, o primeiro tem sempre prioridade sobre o segundo. Por seu lado o escalonador de "tempo real" é também muito simples escolhendo sempre os processos mais prioritários independentemente do seu tempo de execução. O meta-escalonador não é para ser implementado neste trabalho.

Implementação do escalonador

Uma das primeiras tarefas a executar é criar uma RB-tree para colocar as tarefas activas, i.e. que podem ser executadas. A chave desta RB-Tree deve ser a variável *vruntime*, ou seja o tempo de execução da thread em causa. Note que com esta solução a thread a escalonar é sempre a thread no nó mais à esquerda da árvore, pelo que do ponto de vista da eficiência, é importante manter uma variável a apontar para esse nó.

Um aspecto importante é o facto de a variável *vruntime* crescer indefinidamente, o que pode provocar **overflows**. Tenha este aspecto em consideração.

A cada tick de relógio é necessário adicionar ao vruntime da thread activa o valor gasto desde a última vez e verificar se é necessário que se seja *preempted*. Para evitar que uma thread fique muito pouco tempo em execução deve existir uma variável de configuração que estabelece o menor tempo possível que uma thread tem que estar activa (min_delay). De notar que este conceito é diferente do conceito de quantum. Antes de min_delay uma tarefa não pode ser preempted por outra tarefa gerida pelo CFS (pode, no entanto, ser preempted por tarefas em escalonadores de maior prioridade, que neste trabalho não são implementados). Sugere-se que o min_delay seja igual a 5 ticks.

O valor adicionado a cada tick de relógio depende da prioridade da thread. De facto, o valor do tempo a adicionar deve ser multiplicado por prioridade da thread + nice: vruntime = vruntime + ticks[1]*(prioridade+nice). Deste modo as tarefas de prioridade mais baixa têm muito mais tempo que as restantes. Sendo que o valor da prioridade varia entre 1 e 10 e o valor de nice entre 0 e 10.

[1] ticks é o número de ticks de relógio que a tarefa esteve em execução.

Concretização

A biblioteca disponibilizada suporta comutação provocada explicitamente através da invocação da `rotina sthread_yield()` e efectua a comutação das tarefas de cada vez que ocorre um signal periódico que simula a interrupção de relógio. O material fornecido para esta parte do trabalho ajuda a compreender como gerar e tratar signals. Na biblioteca disponibilizada encontra-se ainda uma implementação de monitores.

Na concretização da implementação do escalonamento proposto terá que ter em conta os seguintes pontos:

- A função de criação de tarefas (sthread_create) terá que passar a receber o valor de prioridade inicial para além dos argumentos que já recebia. O protótipo da função passará a ser:

  ```
  sthread_t sthread_create(sthread_start_func_t start_routine, void *arg, int priority);
  ```

- Quando uma tarefa é criada deve ter o valor da prioridade igual a 1 e o valor do nice igual a 0.
- Implemente a função int sthread_nice(int nice) que modifica o valor do nice para a tarefa onde é chamada e retorna o valor da prioridade que a tarefa terá na próxima execução.
- Altere as funções correspondentes à implementação dos monitores para que estes continuem a funcionar com o novo escalonamento. De notar que a forma que as tarefas têm de se bloquear é através do uso dos monitores. Necessitará por isso desse mecanismo de sincronização para bloquear e desbloquear tarefas por forma a testar correctamente o escalonador;
- Implemente uma função cujo protótipo é void `sthread_dump()`. Esta função imprime o estado das listas de tarefas activas e bloqueadas. O output terá que ser obrigatoriamente o seguinte:

Para mais detalhes, ver o Anexo A.

Parte II – Comunicação entre processos

Objectivo

Estender o sistema bancário, implementado em exercícios anteriores, com algumas funcionalidades que facilitam a sua utilização por diferentes operadores.

Registo das operações realizadas

Para efeitos de monitorização do sistema por uma entidade supervisora, pretende-se que todos os comandos realizados pelas tarefas trabalhadoras sejam registados num diário de operações persistente.

Mais precisamente, o sistema bancário deve passar a registar num ficheiro cada comando que é executado e terminado por cada tarefa trabalhadora.

Esse registo deve ser feito num ficheiro chamado **`log.txt`** criado na mesma directoria onde o sistema bancário se executa.

A cada comando terminado, a tarefa trabalhadora que o executou deve escrever uma nova linha no final do ficheiro segundo o formato **`TID: comando`**, em que TID denota o identificador da tarefa e comando denota o comando e respetivos argumentos.

A ordem dos comandos guardados no diário de operações (log) deve respeitar a ordem pela qual os comandos foram efectivamente executados. Em particular, qualquer par de comandos envolvendo a mesma conta devem surgir no diário pela ordem pela qual foram efetivamente executados.

Redirecção do output das simulações

Pretende-se também que o standard output (**`stdout`**) das simulações (processos filho lançados pelo sistema bancário) passe a ser redirecionado para ficheiros individuais na directoria onde o sistema bancário foi executado.

Cada processo filho executado deve ter o seu standard output redirecionado para um ficheiro chamado **`banco-sim-PID.txt`**, criado na mesma directoria em que executa o sistema bancário, em que PID se refere ao pid do processo filho. No caso desse ficheiro já existir antes, o ficheiro anterior deve ser eliminado.

Remote banking

Para aumentar a capacidade de utilização da máquina onde executa o sistema bancário, pretende-se permitir que diferentes utilizadores na mesma máquina introduzam comandos no sistema através de diferentes terminais.

Para tal, o sistema deixa de ler comandos directamente a partir do standard input (**`stdin`**). Em alternativa, o sistema bancário passa a ter um **`pipe com nome`** por onde recebe comandos. O pipe deve ter o nome **banco-pipe** e estar localizado na directoria onde o sistema foi executado, ou na directoria **`/tmp/banco-pipe`** caso existam problemas com privilégios de acesso para escrita.

Devem ser implementados os seguintes pontos:

1. Deve ser criado um programa **`banco-terminal`** que, executado num processo autónomo, lê ordens do teclado e envia os comandos já interpretados para o **`named pipe`** alvo.

   O envio é feito sob forma de uma sequência de bytes correspondente ao conteúdo (binário) de uma estrutura do tipo **`comando_t`**. Deve assumir-se que tanto os processos terminais como o processo a executar o Sistema bancário representam a estrutura **`comando_t`** de forma idêntica em memória.

   A interpretação dos comandos deixa de ser feita no sistema e passa para o progama **`banco-terminal`**. O **`pathname`** do **`named pipe`** é indicado como argumento na linha de comando do banco-terminal. Em cada momento poderá haver múltiplos processos a correr este programa e a enviar concorrentemente comandos para o mesmo sistema.
2. O programa banco-terminal deve aceitar o comando **`sair-terminal`** que permite sair do programa banco-terminal sem terminar o sistema. Nota: o comando sair mantem o mesmo comportamento, terminando o programa.
3. O programa banco-terminal deve mostrar no ecrã o output dos comandos introduzidos pelo utilizador e executados no sistema bancário. O formato é o mesmo dos trabalhos anteriores (1 linha de texto com o resultado da operação). Deve mostrar também, numa nova linha, o **tempo total de execução** do comando no sistema, medido na aplicacão banco-terminal.

   O sistema bancário não deve mostrar o output das operações, esta é agora tarefa dos terminais.

   Nota: Caso não consigam implementar esta funcionalidade na totalidade, os resultados podem ser mostrados no sistema bancário, contando parcialmente para a avaliação.
4. Enquanto o *output* não for impresso, o terminal não avança para ler o próximo comando do teclado.

Experiências

Para verificar a implementação dos requisitos siga o seguinte procedimento:

1. Abrir 4 terminais.
2. No terminal 1 lançar o sistema bancário.
3. No terminal 2 verificar que foi criado o ficheiro banco-pipe.
4. No terminal 3 lançar o **`banco-terminal`** `banco-pipe`, ou **`banco-terminal`** `/tmp/banco-pipe`,e escrever **`simular 4`**.
5. No terminal 2 verificar que foi criado um ficheiro `banco-sim-PID.txt` em que PID e o identificador do processo (filho) que foi criado pelo Sistema bancário na execução do comando simular. Verificar que o conteúdo desse ficheiro e o output do comando simular.
6. Verificar se o ficheiro **`log.txt`** e criado, e se nele é escrito o conteúdo correspondente aos comandos executados pelo sistema bancário.
7. Em seguida no terminal 4 lançar o **`banco-terminal`** `banco-pipe` e escrever **`simular 5`**.
8. Voltar a verificar o ponto 5.
9. Finalmente, verificar que por cada comando simular que introduzir no terminal 3 ou 4 será criado o correspondente ficheiro banco-sim-PID.txt.

10. Escrever **`sair-terminal`** num dos terminais que executa banco-terminal. Verificar que esse processo `banco-terminal` termina, mas os demais continuam em execução.

Funções do Unix/Linux

Para esta parte sugerimos que sejam utilizadas as seguintes funções da API do Unix/Linux: **`unlink`**, **`mkfifo`**, **`open`**, **`close`**, **`dup/dup2`**, **`read`**, **`write`** e **`difftime`**.

Entrega e Avaliação

- A data limite para a entrega do é 31/07/2022 até às 17h. A submissão é feita através do GitHub e confirmada via email.

    - O projecto deve ser hospedado em um repositório git privado que deve ser partilhado com o utilizador `joaojdacosta` no GitHub ou `joaojdacosta` no Bitbucket. Plataformas como GitHub ou BitBucket oferecem contas gratuitas para estudantes. Caso o grupo não tenha acesso a um repositório privado gratuito, entre em contato com o docente. Todos os artefatos, scripts, código, medições e o relatório devem estar presentes no repositório, juntamente com um ficheiro README.md de nível superior que explica: i) a estrutura do repositório e ii) como executar uma pequena demonstração do sistema.

    - Os estudantes devem submeter também um ficheiro no formato zip extraído do GitHub ou BitBucket (ultima actualização antes do fim do prazo) com o código-fonte e makefile atraves do email joaojdacosta@gmail.com. No assunto do email escreva **FCN_SO2021-22: Entrega do projecto**. E no conteúdo do email o anexo e os nomes dos integrantes do grupo.

- A **avaliação é individual**, cada elemento do grupo deve perceber o trabalho apresentado, dominar os conceitos utilizados para implementação e poder estender o código, se necessário.

Cooperação entre Grupos

Os estudantes são livres de discutir com outros colegas soluções alternativas para o exercício. No entanto, em caso algum os estudantes podem copiar ou deixar copiar o código do exercício. Caso duas soluções sejam cópias, ambos grupos reprovarão à disciplina.

Anexo A – Pacote simplethreads

1. Material fornecido

O material dado consiste num pacote que permite crair tarefas que correm em modo utilizador. Esse pacote é o **`sthreads.zip`** que se encontra no repositório da disciplina. Alguns ficheiros que destacamos neste pacote são:

Directório/Ficheiro	**Conteúdo**
`sthread_lib/`	Biblioteca de tarefas
`sthread_lib/sthread_user.c`	Onde vão ser implementadas as funções necessárias para a biblioteca de tarefas.
`sthread_lib /sthread_ctx.{c,h}`	Módulo para criar novas pilhas de execução e para comutar entre elas
`sthread_lib /sthread_switch_i386.h`	Funções assembly para comutar entre pilhas e para salvaguardar registos
`sthread_lib/sthread_time_slice.{c,h}`	Suporte para gerar signals e para controlá-los
`include/`	Contém sthread.h e config.h que são as interfaces públicas, a incluir em programas que usem ou comuniquem com os programas ou bibliotecas referidas.
`test-sthreads/`	Contém vários testes para a biblioteca sthreads

Tabela 1: Ficheiros destacados na biblioteca sthreads

As rotinas no ficheiro **`sthread_ctx.h`** realizam toda a manipulação da pilha, alterações ao PC (program counter), guardam registos e outras manipulações de baixo nível.

O objectivo da Parte I do trabalho é a administração dos contextos de execução das tarefas, pelo que apenas é necessário implementar as funções que se encontram no sthread_user. Durante a implementação modificar apenas o ficheiro **`sthread_user.c`** e, eventualmente, os ficheiros **`sthread.{h,c}`**, enquanto que **`sthread_ctx_t`** não deve ser modificado directamente; para tal devem usar em vez disso as rotinas declaradas em **`sthread_ctx.h`**. Considerando o sistema em camadas (Fig. 2) apenas tem que implementar o "rectângulo a cinzento".

Aplicação (por exemplo, test-create)	
`sthread.c`	
`sthread_pthread`	`sthread_user`
`Pthreads Lib`	`sthread_ctx.c`

`sthread.h`

`Sthread_ctx.h`

Tabela 2: Sistema em camada

Na camada superior encontra-se a aplicação que utilizará o pacote **`sthreads`** (através da API definida em **`sthread.h`**). **`sthread.c`** vai então chamar as rotinas implementadas neste trabalho em **`sthread_user.c`** ou as rotinas em **`sthread_pthread.c`** que fazem uso das pthreads (alterando a variável PTHREADS nas Makefile). O **`sthread_user.c`** por sua vez é construído em cima das rotinas do sthread_ctx (como descrito em **`sthread_ctx.h`**).

As aplicações (camada superior) não devem usar mais rotina nenhuma da biblioteca excepto as definidas em **`sthread.h`**. As aplicações não podem usar rotinas definidas noutros ficheiros nem

podem “saber” como estão implementadas as tarefas. As aplicações apenas pedem para criar tarefas e podem requerer yield ou exit. Também não devem manter listas de tarefas em execução. Isso é a função do módulo marcado a cinzento na Tabela 2.

De igual forma, o rectângulo cinzento – **`sthread_user.c`** – não deve saber como **`sthread_ctx`** está implementado. Deve usar as rotinas definidas em **`sthread_ctx.h`**.

De seguida apresentam-se algumas notas sobre as várias partes da implementação da biblioteca de tarefas-utilizador.

2. Notas sobre as Funções da Biblioteca sthreads

Quanto à **gestão de tarefas** e **escalonamento**:

- **`sthread_create()`** cria uma nova tarefa que se irá executar quando for seleccionada pelo despacho. Qualquer tarefa só deve deixar de correr quando ela própria executar o **`sthread_yield()`** ou quando se bloqueia;
- Use as rotinas fornecidas em **`sthread_ctx.h`**. Não necessita escrever nenhum código assembly, nem manipular registos, nem entender detalhadamente como está organizada a pilha;
- A rotina que é passada para **`sthread_create()`** corresponde ao programa principal da tarefa pelo que se esta terminar, tem que se assegurar que os recursos são libertados (ou seja, **`sthread_exit()`** tem que ser chamado quer explicitamente pela rotina que é passada para **`sthread_create()`** quer implicitamente após a rotina terminar);
- Deve libertar todos os recursos quando a tarefa termina. Não deve no entanto libertar a pilha de uma tarefa que se encontre ainda em execução (nota: para libertar a pilha use **`sthread_free_ctx()`**);
- A tarefa inicial deve ser tratada como qualquer outra tarefa. Por isso, caso se queira pará-la, deve ser criada um estrutura **`sthread_t`** para manter a informação do seu contexto de execução.
- Tenha cuidado com o uso de variáveis locais após chamar **`sthread_switch()`** já que os seus valores podem ser diferentes de anteriormente (é uma pilha de execução diferente);
- A função de inicialização da biblioteca sthreads, **`sthread_user_init`**, inicia o escalonador de tempo partilhado invocando **`sthread_time_slices_init`**, lançando assim um signal periódico, cujo função de tratamento inclui o algoritmo de despacho.

Quanto aos **mutexes** e **monitores**:

- Compreenda como bloquear uma tarefa, fazendo-a esperar numa fila. Como obtém a tarefa que se bloqueou? Como altera o contexto dela para passar para outra tarefa? Como volta a corrê-la?
- Quando se desbloqueia uma tarefa, não ocorre imediatamente uma mudança de tarefa. A tarefa fica executável e será executada quando seleccionada pelo despacho;
- Para colocar sincronização no sistema de tarefas existem na biblioteca duas primitivas de sincronização: **`atomic_test_and_set`** e **`atomic_clear`** que permitem realizar a exclusão mútua a baixo nível.

3. Outras notas

- Se desactivar os signals não se esqueça de os activar em todos o caminhos possíveis do seu programa. Provavelmente vai querer desactivar os signals durante todo o tempo que estiver dentro de um yield e activá-las ao completar o **`sthread_switch`**. De notar que

sthread_switch pode retornar para dois locais diferentes: para a linha a seguir a uma chamada a um **sthread_switch** ou para a sua função de começo da tarefa quando comuta para uma nova tarefa pela primeira vez. Active os signals nos dois locais;

- Não deve executar código da aplicação com os signals desligados;
- Verifique que o teste para time-slices funciona (**test-time-slices.c**) e verifique que todos os outros testes funcionam ainda.
- Uns bons valores para o período das interrupções são 10 milisegundos.
- Para comparar programas com e sem preempção comente a chamada à rotina **sthread_time_slices_init()**.