Trabalho Prático I: Biblioteca de Threads Compact Threads (cthreads)

Cristiano Salla Lunardi - 240508
Gustavo Madeira Santana - 252853

INF01142 - Sistemas Operacionais I N - 2016/2

Prof. Alexandre Carissimi

1 Introdução

Este relatório tem como objetivo descrever o que foi desenvolvido no primeiro trabalho prático da disciplina (INF01142) Sistemas Operacionais I N. A proposta desse trabalho é desenvolver uma biblioteca de threads *Compact Threads*, ou *cthreads*, para manipular threads em execução em um sistema operacional.

O desenvolvimento foi feito na linguagem C e usando uma máquina virtual em ambiente GNU/LINUX.

2 Biblioteca *Compact Threads*

A biblioteca, implementada fundamentalmente no arquivo *cthread.c*, possui todas funções requeridas para manipular as threads [1], também possuí funções auxiliares como a função*cscheduler* que faz o sorteio da thread que entrará em

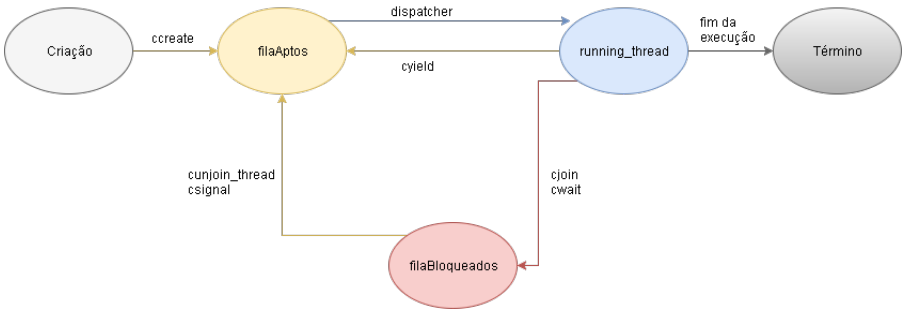

Figure 1: Diagrama de estados da biblioteca *cthreads*

execução e a função *init_cthreads*, que cuida da inicialização das estruturas de dados usadas pela biblioteca, bem como inicialização do scheduler e da thread para a função main.

O funcionamento de cada função será destacado nas seções seguintes.

2.1 Função *ccreate*

Principal função da biblioteca, *ccreate* é usada para criar threads e associa-lá a uma função. Recebe como parâmetros uma função e um argumento e retorna o *tid* da thread criada.

```
TCB_t *cthread = malloc(sizeof(TCB_t));
cthread->tid = thread_count; thread_count++;
cthread->state = PROCST_CRIACAO;
cthread->ticket = ticket_gen();

getcontext(&cthread->context);
cthread->context.uc_link = &scheduler;
cthread->context.uc_stack.ss_sp = malloc(SIGSTKSZ);
cthread->context.uc_stack.ss_size = SIGSTKSZ;
```

Figure 2: Criação e definição do contexto de uma nova thread

Para criação da thread, é realizado uma alocação na memória para o *TCB*, e alguns dados são inicializados. A thread receberá uma *tid*, valor inteiro que é incrementado a cada criação. Seu estado durante a criação é 0[2]. Para criar o contexto da thread, inicialmente é feito uso da função *getcontext* para ter o modelo pronto, depois então é definido que ao fim da execução da thread, o contexto a ser carregado deve ser o do *scheduler*, através do *uc_link*. Também é feita uma alocação de memória para a pilha que será usada pelo contexto, através da *ss_sp* e *ss_size*.

Por fim, ao ser inserido na fila de aptos, seu estado é alterado para 1[3]e a criação está completa.

2.2 Função *cyield*

Para "passar a vez" voluntariamente, onde a thread retorna para a fila de aptos após ceder voluntariamente sua vez na execução.

A thread que chamou a função *cyield* tem seu estado alterado para *PROCST_APTO* e inserido de volta na fila de aptos. O controle é então passado para o *scheduler*.

2.3 Função *cjoin*

Usada para bloquear uma thread até que outra thread termine sua execução. Tem como único parâmetro o valor do *tid* da thread que deverá finalizar sua execução. A função que chamou *cjoin* só voltará para a fila de aptos após a thread indicada terminar.

[1] ccreate, cyield, cjoin, csem_init, cwait, csignal
[2] *PROCST_CRIACAO*
[3] *PROCST_APTO*

Foi criado uma estrutura auxiliar chamada de *Join Control Block*, ou *JCB*. Neste descritor é armazenado qual thread será bloqueada, e o *tid* da thread que a libera. Existe ainda uma função auxiliar *cunjoin_thread* que cuida do desbloqueio da thread quando a indicada termina sua execução, seu comportamento é descrito em outra seção.

```
typedef struct s_JCB {
    int tid; //tid da thread bloqueante
    TCB_t *thread; // thread bloqueada
} JCB_t;
```

Figure 3: Join Control Block, descritor auxiliar para controlar threads com *join*

2.4 Função *csem_init*

Nesta função é inicializado o *semáforo*. Recebe como parâmetros um ponteiro para a struct do semáforo e a quantidade de recursos que este semáforo terá. Para simular um *mutex*, a quantidade de recursos deve ser *1*.

Para inicializar o semáforo é criado uma fila, onde será armazenado quais threads estão disputando recursos. Para usar ou liberar um recurso, são usadas as funções *cwait* e *signal*, respectivamente. Seus comportamentos são explicados a seguir

2.5 Função *cwait*

A função *cwait* tem como finalidade fazer a requisição de recursos do *semáforo*. Caso o semáforo tenha recursos disponíveis, seu indicador de recursos é decrementado e a thread segue sua execução. Caso não haja recursos disponíveis, a thread é bloqueada e entra para a fila do semáforo.

2.6 Função *csignal*

A função *csignal* é usada para indicar que a thread está liberando recursos para o semáforo. Uma vez que ela é chamada, o número de recursos do semáforo é incrementado, consequentemente a primeira thread na fila do semáforo é colocada na fila de aptos, podendo então fazer uso do recurso liberado pela thread que chamou a função *csignal*.

3 Funções Auxiliares

3.1 Função *init_cthreads*

Essa função tem a finalidade de inicializar as estruturas de dados usadas pela biblioteca, bem como inicialização do contexto do *scheduler* e da thread para a função *main*. Nele são inicilizadas filas para as threads que estão em estado

Apto e *Bloqueado*, e também do ponteiro para a thread que está em execução, que ao inicializar é a *main*.

3.2 Função *cscheduler*

O *scheduler* implementado é do tipo não preemptivo, sem priodidades, e segue uma política de loteria. Toda thread ao ser criada recebe um bilhete [4], número aleatório entre 0 e 255. O *scheduler* então sorteia um número e escolhe a thread que tem um bilhete mais próximo do sorteado. Em caso de duas threads com mesmo bilhete, a escolhida será a que tiver menor *tid*.

```
  // inicialização do scheduler
  getcontext(&scheduler);
  scheduler.uc_link = &main_thread.context; //scheduler volta para main
  scheduler.uc_stack.ss_sp = ss_scheduler;
  scheduler.uc_stack.ss_size = SIGSTKSZ;
  makecontext(&scheduler, (void (*)(void))cscheduler, 0);
```

Figure 4: Inicialização do scheduler, e criação de seu contexto.

3.3 Função *find_thread*

Função auxiliar para saber se uma dada thread existe em uma dada fila. Recebe como parâmetros um *tid* e uma *fila* (fila de aptos ou bloqueados por exemplo). Retorna *0* se a thread foi encontrada ou *-1* se a thread não foi encontrada.

3.4 Função *remove_thread*

Função auxiliar para remover uma dada thread em uma dada fila. Recebe como parâmetros um *tid* e uma *fila* (fila de aptos ou bloqueados por exemplo). Retorna *0* se a thread foi encontrada e removida ou *-1* se não foi possível remover a thread especificada.

3.5 Função *cunjoin_thread*

Esta função foi criada para auxiliar a função *cjoin*. Toda vez que o *scheduler* entra em execução, a função *cunjoin_thread* é chamada para verificar se a thread que acabou de terminar sua execução estava segurando alguma outra thread [5]. Em caso positivo, a função remove esta interdepedência de threads e coloca a thread bloqueada na fila de *aptos*.

[4] o número é gerado pela função ticket_gen
[5] através do uso da função cjoin

3.6 Função *ticket_gen*

Encarregada de gerar um número aleatório entre *0* e *255*, para ser usada como bilhete da thread, assim como número sorteado no processo de escolha de uma thread pelo *scheduler*. Faz isso da função *Random2* fornecida pelo arquivo *support.o*.

3.7 Função *cidentify*

Ao chamar esta função, é mostrado os componentes do grupo.

3.8 Funções *debugOn* e *debugOff*

Função auxiliar para fazer o debug da biblioteca. Ao chamar *debugOn*, diversos *printf* são ativados nas funções para ficar evidente ao usuário o que está sendo feito. Para desativar basta chamar a função *debugOff*. Ambas não recebem nenhum parâmetro.

4 Testes

Todos testes podem ser executados com o modo debug ligado, basta adicionar debug ao nome do teste. Por exemplo, para o teste1 seria executado ./teste1debug em favor de ./teste1. Para o teste2 se executaria ./teste2debug, e assim por diante.

4.1 Teste do *ccreate* com *cyield*

- Arquivo: teste1.c — ./teste1 — ./teste1debug
- Funções usadas: ccreate, cyield
- Threads criadas: 104

4.2 Teste do *cjoin*

- Arquivo: teste2.c — ./teste2 — ./teste2debug
- Funções usadas: ccreate, cjoin, cyield
- Threads criadas: 3

4.3 Teste do *semáforo*

- Arquivo: teste3.c — ./teste3 — ./teste3debug
- Funções usadas: ccreate, csem_init, cwait, csignal, cyield
- Threads criadas: 5

5 Problemas

5.1 Segmentation fault...

Um problema foi identificado ao misturar a função cjoin com as diretivas do semáforo. Dependendo da ordem que ocorre a execução, por exemplo, cwait seguindo de um cjoin, o teste é encerrado com erro de segmentação.

Um dos comandos usados para debug foi:

```
valgrind --trace-children=yes --track-fds=yes --log-fd=2
--error-limit=no --leak-check=full --show-possibly-lost=yes
--track-origins=yes --show-reachable=yes <executavel>
```

Diversos erros de segmentação foram corrigodos durante a produção do trabalho. Alocações estáticas foram trocadas por alocações dinâmicas onde era conveniente. Também foi implementado um melhor controle do que não estava mais em uso, consequentemente liberando a memória e evitando então erros de segmentação. O único erro que restou foi o que envolve o cjoin em uso com o operador de semáforo.