

Universidade Federal de Minas Gerais
DCC605: Sistemas Operacionais
Trabalho Prático

[Cronograma e execução](#)

[Biblioteca de threads de usuário](#)

[Parte 1: Criando e executando threads](#)

[Parte 2: Terminando e esperando threads](#)

[Parte 3: Preempção](#)

[Parte 4: Evitando condições de corrida](#)

[Parte 5: Adicionando suporte à espera](#)

[Implementação, entrega e avaliação](#)

[Implementação da biblioteca](#)

[Relatório](#)

[Pontos extras](#)

Cronograma e execução

Liberado: 4 de abril

Entrega: 4 de maio

Execução: em grupo de até quatro alunos

Valor: 17 pontos

Biblioteca de threads de usuário

Neste trabalho iremos implementar uma biblioteca de threads em espaço do usuário para o Linux. Para tanto, iremos utilizar a interface de controle de contextos [ucontext] e sinais assíncronos.

Parte 1: Criando e executando threads

Na primeira parte você irá apenas criar threads, sem suporte à preempção. Isto é, uma thread irá ocupar a CPU até que ela dê licença para outra thread. Sua biblioteca the threads deverá implementar uma função:

```
void dccthread_init(void (*func)(int), int param);
```

Onde [func] é um ponteiro para uma função que não retorna valor e que recebe um inteiro como parâmetro; e [param] é o parâmetro que será passado para [func]. A função [dccthread_init] nunca retorna. Ela deve inicializar as estruturas internas da biblioteca de threads e depois criar uma thread para executar [func] passando o parâmetro [param].

Nesta parte 1, sua função [dccthread_init] deve (i) iniciar uma thread [gerente] que irá fazer o escalonamento das threads criadas, (ii) iniciar uma thread [principal] que irá executar a função [func], e (iii) executar a thread [principal].

Cada thread (como as threads [gerente] e [principal]), é representada por uma estrutura [ucontext_t] declarada em [ucontext.h] e gerenciada pelas funções [getcontext], [setcontext], [makecontext] e [swapcontext]. Estude a documentação e os exemplos de uso destas funções em suas páginas no manual. Em particular, note que você precisa chamar [getcontext] e inicializar a pilha da nova thread antes de chamar [makecontext].

Nesta parte um você também irá implementar a função da biblioteca que cria novas threads:

[illegible]

Onde [name] é o nome da thread, [func] é a função que será executada e [param] é o parâmetro que será passado para a função [func]. Os parâmetros [func] e [param] funcionam de forma idêntica aos parâmetros de mesmo nome na função [dccthread_init].

A função [dccthread_create] deve inicializar uma nova thread e adicioná-la à lista de threads prontas para executar, juntamente às outras threads criadas e à thread [principal].

Por último, você deverá criar uma função para permitir que uma thread dê licença da CPU:

```
void dccthread_yield(void);
```

A função [dccthread_yield] deve retirar a thread atual da CPU e chamar a thread [gerente] para que esta escolha a próxima thread que deve ser executada.

Cada thread deve ter seu estado salvo em uma estrutura [dccthread_t], que você deverá definir. Note que [dccthread_t] precisa conter pelo menos um campo do tipo [ucontext_t] para armazenar o estado da thread. Para trocar threads use [swapcontext]; esta função salva o estado da thread atual e carrega o estado da nova thread, de forma similar à troca de contexto do xv6 visto em sala.

Depois de implementadas estas funções você deverá ser capaz de executar o seguinte teste:

```
#include <stdlib.h>
#include <stdio.h>
#include "dccthread.h"

void f1(int dummy)
{
    dccthread_t *tid = dccthread_self();
    char *name = dccthread_name(tid);
    for(int i = 0; i < 3; i++) {
        printf("thread %s on iteration %d\n", name, i);
        dccthread_yield();
    }
}

void test1(int dummy) {
    dccthread_t *tid1 = dccthread_create("t1", f1, 0);
    dccthread_t *tid2 = dccthread_create("t2", f1, 0);
    dccthread_t *tid3 = dccthread_create("t3", f1, 0);
    for(int i = 0; i < 10; i++) {
        printf("test1 yielding\n");
        dccthread_yield();
    }
    exit(EXIT_SUCCESS);
}

int main(int argc, char **argv)
{
    dccthread_init(test1, 0);
}
```

Parte 2: Terminando e esperando threads

Nesta parte iremos implementar funções que darão suporte à terminação e espera por threads. Você deverá implementar as funções:

```
void dccthread_exit(void);  
void dccthread_wait(dccthread_t *tid);
```

Onde a função [dccthread_exit] termina a thread atual e a função [pthread_wait] bloqueia a thread atual até que a thread [tid] termine de executar. Para facilitar, você pode assumir que cada thread do programa só pode ser “esperada” por uma thread. O programa deve terminar quando a última thread chamar [dccthread_exit].

Note que seu programa deve funcionar para as duas sequências possíveis de evento:

1. Quando uma função chama [dccthread_exit] e nenhuma função a espera; quando uma função chama [dccthread_wait] passando o identificador [tid] de uma thread que já terminou.
2. Quando uma função chama [dccthread_exit] e já existe uma função esperando; quando uma função chama [dccthread_wait] passando o identificador [tid] de uma thread que ainda está executando.

Depois de implementadas estas funções você deverá ser capaz de executar o seguinte teste:

```
#include <stdlib.h>  
#include <stdio.h>  
  
#include "dccthread.h"  
  
void f1(void *data)  
{  
    dccthread_t *tid = dccthread_self();  
    char *name = dccthread_name(tid);  
    for(int i = 0; i < 3; i++) {  
        printf("thread %s on iteration %d\n", name, i);  
        dccthread_yield();  
    }  
    dccthread_exit();  
}
```

```

void test2(int dummy) {
    for(int i = 0; i < 5; i++) {
        dccthread_t *t1 = dccthread_create("t1", f1, 0);
        dccthread_t *t2 = dccthread_create("t2", f1, 0);
        dccthread_wait(t1);
        dccthread_wait(t2);
    }
    dccthread_exit();
}

int main(int argc, char **argv)
{
    dccthread_init(test2, 0);
}

```

Parte 3: Preempção

Nesta parte você irá tornar sua biblioteca preemptiva, retirando threads da CPU “a força” caso elas não chamem [dccthread_yield]. Para realizar isso, você precisará de um temporizador que irá disparar regularmente. Verifique a documentação das funções [timer_create], [timer_delete] e [timer_settime]. O objetivo é usar estas funções para gerar um sinal a cada 10 ms. Você deve usar um temporizador que conte o tempo de CPU utilizado pelo processo [CLOCK_PROCESS_CPUTIME_ID]; seu tratador de sinais irá então chamar [dccthread_yield] para forçar a thread atual a dar licença da CPU.

Depois de implementado o temporizador, você deve ser capaz de executar o teste abaixo alternando entre t1 e t2 (sem preempção as threads não dão licença da CPU):

```

#include <stdlib.h>
#include <stdio.h>
#include "dccthread.h"

void f1(int dummy) {
    for(unsigned long long i = 0; i < 1000000000ULL; i++);
    dccthread_exit();
}

void test2(int dummy) {
    for(int i = 0; i < 5; i++) {
        dccthread_t *t1 = dccthread_create("t1", f1, 0);
        dccthread_t *t2 = dccthread_create("t2", f1, 0);
        dccthread_wait(t1);
        dccthread_wait(t2);
    }
    dccthread_exit();
}

int main(int argc, char **argv)
{
    dccthread_init(test2, 0);
}

```

Parte 4: Evitando condições de corrida

O que acontece se a thread atual chamar [dccthread_yield] e o temporizador disparar em seguida? Este tipo de situação pode levar à corrupção das estruturas de dados de sua biblioteca de threads. Uma solução para evitar que as estruturas de dados de sua biblioteca sejam corrompidas, você deve desabilitar o temporizador imediatamente após uma thread chamar [dccthread_yield], [dccthread_wait], [dccthread_exit] ou [dccthread_create]. Além disso, você também deve desabilitar interrupções enquanto a thread [gerente] estiver executando.

Para desabilitar e habilitar o temporizador basta bloquear e desbloquear, respectivamente, o sinal enviado pelo temporizador. Você pode fazer isso usando a função [sigprocmask]. Em geral, você vai querer chamar [sigprocmask(SIG_BLOCK, ...)] assim que uma thread chamar uma função da biblioteca, como [dccthread_yield], e chamar [sigprocmask(SIG_UNBLOCK, ...)] antes de retornar da função da biblioteca.

Note que a máscara de sinais bloqueados é distinta para cada [ucontext_t]. Logo, você deve controlar quais e quando os sinais são bloqueados por cada thread.

Para testar se seu código evita corrupção de dados durante preempção, substitua a função [f1] no exemplo anterior por:

```
void f2(int dummy) {
    for(unsigned long long i = 0; i < 1000000000ULL; i++) {
        dccthread_yield();
    }
    dccthread_exit();
}
```

A não ser que você implemente o bloqueio de sinais como descrito acima, seu código provavelmente sofrerá uma falha de segmentação.

Parte 5: Adicionando suporte à espera

Como nossa biblioteca de threads usa mapeamento muitos-para-um, uma thread não pode chamar [sleep] sem bloquear todas as threads do programa. Para solucionar esta limitação vamos criar uma função [dccthread_sleep] que retira a thread da CPU (como [dccthread_yield]) e da fila de prontos por uma quantidade de tempo pré-determinada.

Sua biblioteca the threads deverá implementar uma função:

```
void dccthread_sleep(struct timespec ts);
```

Onde [ts] é uma variável do tipo [struct timespec] que contém o tempo pelo qual a thread deve ser mantida fora da fila de prontos.

Para implementar esta funcionalidade sua biblioteca deverá fazer uso de temporizadores adicionais. Para implementação de [dccthread_sleep] você pode usar o temporizador [CLOCK_REALTIME] da família de funções [timer_create], [timer_delete] e [timer_settime].

Implementação, entrega e avaliação

Implementação da biblioteca

O cabeçalho [dccthread.h] com as declarações das funções da biblioteca estão disponíveis no Moodle; não é necessário modificar este arquivo. Seu grupo deve entregar um arquivo [dccthread.c] que implementa as funções atendendo às funcionalidades descritas acima.

Relatório

Cada grupo deve preencher e entregar o arquivo [report.txt] incluso no pacote com as bibliotecas.

Pontos extras

Serão avaliados como atividades extras valendo pontos:

1. Desenvolvimento de baterias de testes além das descritas nesta especificação;
2. Mapeamento muitos para muitos.