Sistemas Operacionais
Segundo Trabalho
Biblioteca de Threads Não Preemptivas
Datas de Entrega: 6 e 13 de junho de 2021

Descrição do Trabalho

O objetivo principal deste trabalho é criar uma biblioteca de threads em nível de usuário para suportar multiprogramação. Mais especificamente, você irá:

- Implementar um escalonador não preemptivo
- Implementar troca de contexto entre threads
- Implementar primitivas básicas de exclusão mútua
- Medir o custo de trocas de contexto

Muito embora a biblioteca seja bastante primitiva, você lidará com vários mecanismos chaves e aplicará técnicas que são importantes para construir sistemas operacionais multiprogramados no futuro. As threads são de nível de usuário e compartilham o mesmo espaço de endereçamento.

Revisão de Plano de Projeto

Você deverá entregar um documento até o dia 6 de junho descrevendo como você pretende resolver os seguintes pontos:

1. *Thread Control Block* (TCB): Quais informações você armazenará no TCB e como elas serão inicializadas?
2. Troca de Contexto: Como você irá salvar e restaurar o contexto de uma thread?
3. Exclusão Mútua: Qual o seu plano para implementar exclusão mútua?
4. Escalonamento: Assumindo que as threads abaixo sejam colocadas na fila de tarefas prontas (*READY QUEUE*) na ordem alfabética de seus nomes e que a execução comece com a Thread 1, como a execução ocorrerá?

Thread 1

```
lock_init(&lock);
lock_acquire(&lock);
thread_yield();
lock_release(&lock);
thread_exit();
```

Thread 2

```
while(1)
{
    thread_yield();
}
```

Thread 3

```
thread_yield();
lock_acquire(&lock);
lock_release(&lock);
thread_exit();
```

Thread 4

```
lock_acquire(&lock);
lock_release(&lock);
thread_exit();
```

Iniciando

Use o código inicial disponibilizado no AVA. O código inicial contém um arquivo `Makefile` que permite que você compile o código usando o comando `make` e gere um arquivo de biblioteca (.a) que será ligado ao código dos seus testes para gerar um arquivo executável. As funções a serem implementadas estão marcadas com o comentário TODO. O diretório `examples` contém alguns exemplos para que você possa testar sua implementação.

Threads e Escalonamento

Para utilizar a biblioteca de threads, o seu programa deve inicializar a biblioteca chamando primeiro a função `thread.c:thread_init()`. Essa função deve inicializar as variáveis internas da biblioteca, como fila de threads prontas, e criar uma thread para a função principal (`main`) do seu programa.

Todas as threads rodam no mesmo espaço de endereçamento do processo principal, mas cada thread possui o seu contexto (armazenado em seu TCB) e uma pilha exclusiva. Pilhas possuem tamanho `STACK_SIZE` bytes e são alocadas dinamicamente quando uma thread é criada com a função `thread.c:thread_create()`.

Considerando que você está escrevendo um escalonador não preemptivo, as threads rodam sem ser interrompidas até que elas liberem a cpu (*yield*) ou terminem (*exit*) ao chamar `thread.c:thread_yield()` ou `thread.c:thread_exit()`. Essas funções chamam `entry.S:scheduler_entry()`, que salva os conteúdos dos registradores de propósito geral e flags no TCB da thread que estava rodando. `thread.c:scheduler()` é então chamada para escolher a próxima tarefa a ser executada usando a política round-robin. Na primeira rodada, as tarefas executam na ordem em que são especificadas pelas chamadas a `thread_create()`.

A função `thread.c:thread_join()` aguarda o término de uma thread. Ela deve verificar se a thread passada como parâmetro terminou e retornar. Caso a thread não tenha terminado, ela deve liberar a CPU para que outras threads executem.

Certifique-se que a sua biblioteca ainda funciona se todas as tarefas terminarem, mesmo que elas não chamem `pthread_exit()`.

Grafo de Chamadas para Preservação de Tarefas

O grafo de chamadas para preservação de tarefas pode parecer complicado, os diagramas abaixo ajudam a entende-lo.

Threads

```
thread_yield(thread.c)-> scheduler_entry(entry.S)-> scheduler(thread.c)

thread_exit(thread.c) -> scheduler_entry(entry.S)-> scheduler(thread.c)
```

Exclusão Mútua

`lock.c` contém três funções para uso das threads: `lock_init(l)`, `lock_acquire(l)` e `lock_release(l)`. `lock_init(l)` inicializa o lock `l`; inicialmente nenhuma thread detém o lock especificado. Caso uma estrutura de lock não inicializada seja passada para outras funções, o resultado é indeterminado e pode ser definido por você. `lock_acquire(l)` não deve bloquear a thread que a chamou quando nenhuma outra thread detém `l`.

Threads chamam `lock_acquire(l)` para adquirir o lock `l`. Se nenhuma outra thread detém `l`, ela marca o lock como ocupado e retorna. Caso contrário, ela usa `lock.c:block(...)` para indicar ao escalonador que a thread agora está esperando pelo lock e deve ser colocada na **fila de tarefas bloqueadas associadas ao lock**. Se houver uma ou mais threads na fila de tarefas bloqueadas quando `lock_release(l)` é chamada pela thread que detém o lock, o lock permanece ocupado e a thread no início da fila de tarefas bloqueadas é colocada no final da **fila de tarefas prontas**, usando uma operação normal de inserção em fila. Se não houver threads esperando, ela marca o lock como livre e retorna. Os resultados de uma thread que tenta adquirir um lock que ela já possui ou liberar um lock que ela não detém é indeterminado e pode ser definido por você.

O código inicial contém uma implementação não adequada (usando espera ocupada) de exclusão mútua em `lock.c` para que você possa escrever e testar separadamente a parte relativa a threads do seu trabalho. Quando você estiver pronto para testar a sua própria implementação, mude a constante `SPIN` em `lock.c` para `FALSE`.

OBS: As notas de aula descrevem como sincronizar acesso aos campos associados a um lock. Como não há preempção neste kernel, se `lock_acquire(l)` e `block()` não chamarem `thread_yield()` no meio da manipulação de estruturas de dados, você não precisa se preocupar com condições de corrida enquanto essas funções realizam suas tarefas. Em particular, não é necessário proteger as estruturas de dados do escalonador com um lock test-and-set.

Medições de Tempo de `thread_yield`

A função `util.c:get_timer()` retorna o número de ciclos do processador desde o boot. Utilizando `util.c:print_int()` e `util.c:print_str()`, use a primeira linha da tela para mostrar o número de ciclos usados por `thread_yield` criando um programa de teste com duas threads. Não considere os tempos de `thread_exit`, `if`, `print_str` ou `print_int`.

Ponto Extra

Para um ponto extra no trabalho, implemente um escalonador justo. Isso significa que o escalonador sempre escolhe a próxima tarefa a executar aquela que teve o menor tempo de CPU até o momento da escolha. Não inclua o tempo para troca de contexto quando estiver contabilizando os tempos de execução de cada tarefa. Prove que o seu escalonador é justo escrevendo tarefas que mostram que o tempo de CPU é dividido igualmente entre as tarefas.

Crie uma maneira de ligar e desligar as funcionalidades do ponto extra e explique como isso é feito no arquivo README. Deixe as funcionalidades do ponto extra desligadas por padrão.

Entrega do Trabalho

O trabalho pode ser feito em grupo de no máximo três alunos e deve ser entregue até o dia 13 de junho de 2021. Porém, você terá de entregar um documento no dia **6 de junho de 2021** com as diretrizes iniciais indicando como pretende implementar o trabalho (veja a Seção Revisão de Plano de Projeto). Além do código fonte devidamente comentado, o grupo deve entregar um breve relatório descrevendo o trabalho. Neste relatório, o grupo deve incluir uma breve introdução, decisões de implementação, funcionalidades não implementadas, problemas enfrentados na implementação, análise de desempenho com gráficos, etc. O relatório deve ser entregue em um arquivo PDF. Tanto o relatório quanto os arquivos fontes da implementação devem ser colocados em um arquivo ZIP e enviados via AVA. Arquivos em outros formatos, como RAR, não serão considerados e o trabalho não será avaliado. Inclua em seu arquivo ZIP apenas os arquivos fontes e Makefile, ou seja, não inclua arquivos compilados (.o).

Avaliação

Além da correção do programa, o professor e/ou assistente de ensino poderão agendar entrevistas com o grupo. Durante a entrevista, o grupo deverá explicar o funcionamento do programa e responder a perguntas relativas ao projeto.

Algumas Dicas

- Uma das primeiras coisas que você deve fazer é implementar uma estrutura de dados de fila.
- Salvar e restaurar os registradores de propósito geral (`rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi`) é relativamente fácil, mas o registrador de flags (`rflags`) é mais complicado. A única maneira de se salver `rflags` é com a instrução `pushfq`, que coloca o conteúdo de `rflags` na pilha. De modo semelhante, a única maneira de restaurar `rflags` é usando `popfq`, que retira o conteúdo do topo da pilha e copia para `rflags`. Você deve ter cuidado para não mudar nenhum registrador antes dele ser salvo.
- Comece com o conjunto mais simples de tarefas:
    - Primeiro tente com uma tarefa (usando o programa de teste no diretório `examples/basic`)
    - Se o seu programa funciona para essa tarefa, tente uma outra.
    - Se o seu programa funciona para todas as tarefas individualmente, tente combinações de várias tarefas.