Avaliação – Sistemas Operacionais: Sincronização

Aluna: Sthefania Fernandes Silva

Matrícula: 20210072430

Questão escolhida: mutex melhorado a partir de um futex

Considere a seguinte implementação de mutex com futex discutida em sala de aula.

```
void mutex_lock(struct mutex *m)
{
    uint32_t v;
    for (;;) {
        v = 0;
        if (atomic_compare_exchange_strong(&m->v, &v, 1)) {
            break;
        }
        futex(&m->v, FUTEX_WAIT, v);
        }
    }

void mutex_unlock(struct mutex *m)
{
    atomic_store(&m->v, 0);
    futex(&m->v, FUTEX_WAKE, 1);
}
```

Ela tem o seguinte problema de desempenho: todas as vezes que a função mutex unlock é chamada, ela faz uma chamada de sistema que requer uma transição para o núcleo (kernel) do sistema operacional para acordar alguma thread que possa estar dormindo, mesmo quando não há nenhuma chance de haver threads dormindo (i.e., quando não há contenção); essa transição é relativamente cara e pode se tornar proibitiva se for chamada frequentemente.

Implemente uma versão melhorada da API de mutex que evita esse problema de desempenho, ou seja, caso não haja contenção (i.e., existe apenas uma thread por vez travando e destravando o mutex), as implementações tanto de mutex lock quanto de mutex unlock não fazem nenhuma transição para o núcleo de sistema operacional.

Resolução

Para evitar a chamada do kernel para acordar alguma thread que possa estar dormindo, o código sofreu algumas alterações. Primeiramente, foram definidas 3 variáveis:

1. LOCK_FREE (0): esse estado indica que o mutex está desbloqueado e disponível para aquisição.
2. LOCK_TAKEN (1): esse estado indica que o mutex já está bloqueado por uma thread.
3. LOCK_WAITING (2): nesse estado o mutex está bloqueado, e uma ou mais threads estão aguardando para adquiri-lo.

Diante disso, a função mutex_unlock verifica se há threads em espera antes de acordar uma thread adormecida. Isso é feito com a função atomic_dec que decrementa o valor da variável m->v e retorna o valor anterior dela.

Então, se o valor anterior de m->v for igual a LOCK_TAKEN, significa que não há threads em espera. Logo, nenhuma thread precisa ser acordada.

Se o valor do estado for diferente de LOCK_TAKEN, significa que há pelo menos uma thread adormecida. Então fazemos a chamada do sistema para acordá-la. Então, se o mutex é definido como desbloqueado (LOCK_FREE). Em seguida, a função acorda uma thread que estava esperando para adquirir o mutex.

```
void mutex_unlock(struct mutex *m)
{
      if(atomic_dec(&m->v) != LOCK_TAKEN){
        atomic_store(&m->v, 0);
        futex(&m->v, FUTEX_WAKE, 1);
        // 1 indica que acorda uma única thread
        //nessa alteração só é possível chamar quando há alguém pra
acordar
      }
}
```

Já na função mutex_lock, dentro de um loop, faço uma comparação atômica entre &m->v e &v, se os dois forem iguais m->v recebe LOCK_TAKEN (1) e como retorno tenho true. Isso significa que o mutex estava livre e agora passou a ficar bloqueado para que a thread possa ser executada.

```
if (atomic_compare_exchange_strong(&m->v, &v, LOCK_TAKEN)) {
         break; //sai do loop
        }
```

Se isso não for verdade, há duas possibilidades: ainda não há threads esperando e pode haver uma ou mais threads esperando.

Se não havia threads esperando até agora, vou indicar que agora existe uma através do LOCK_WAITING. Isso significa que precisamos alterar o valor de LOCK_TAKEN para LOCK_WAITING usando a comparação atômica com &m->v.

A outra opção é que há threads esperando, logo m->v é igual LOCK_WAITING.

Para ambos os casos chamamos FUTEX_WAIT para adormecer a thread, garantindo que o mutex estava com LOCK_WAITING ao realizar essa operação.

```
else if (atomic_compare_exchange_strong(&m->v, LOCK_TAKEN, LOCK_WAITING)
|| m->v==LOCK_WAITING){
       futex(&m->v, FUTEX_WAIT, LOCK_WAITING);
        }
```

Abaixo o código completo:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdatomic.h>
#include <pthread.h>
#include <linux/futex.h>

#define LOCK_FREE 0
#define LOCK_TAKEN 1
#define LOCK_WAITING 2

struct mutex {
    atomic_uint v;
};

void mutex_init(struct mutex *m)
{
    m->v = LOCK_FREE;
}

void mutex_lock(struct mutex *m)
{
    uint32_t v; //variável de que determina estado do futex

    for (;;) {
       v = LOCK_FREE; //inicio setando v como LOCK_FREE, que significa
```

```
que o mutex está desbloqueado

       /*em seguida, comparo &m->v com &v se forem iguais m->v recebe
LOCK_TAKEN (1)
       e como retorno tenho true, assim o mutex estava livre mas agora
ficou bloqueado
       para a thread poder ser executada
       */
       if (atomic_compare_exchange_strong(&m->v, &v, LOCK_TAKEN)) {
             break; //sai do loop
       }
       else if (atomic_compare_exchange_strong(&m->v, LOCK_TAKEN,
LOCK_WAITING) || m->v==LOCK_WAITING){
            futex(&m->v, FUTEX_WAIT, LOCK_WAITING);
       }

    }
}

void mutex_unlock(struct mutex *m)
{
    if(atomic_dec(&m->v) != LOCK_TAKEN){
       atomic_store(&m->v, LOCK_FREE);
       futex(&m->v, FUTEX_WAKE, 1);
        // 1 indica que acorda uma única thread
       //nessa alteração só é possível chamar quando há alguém pra
acordar
    }

}
```

Dificuldades

Foi um desafio entender como funcionava, afinal é um conteúdo abstrato e eu não consegui testar com os códigos de apoio do SIGAA. Por essa razão, não tenho certeza se o código está 100% correto.

Entretanto, com base no que estudei e nas referências que consultei, assumo que consegui entender o que precisava ser feito e resolver o problema. Principalmente na função unlock que foi mais simples e intuitiva de se resolver, a função lock foi a que me causou mais dúvidas se o loop estaria funcionando da forma que imaginei.

Referências

[Futexes Are Tricky]

Futex: fast userspace mutex