Avaliação – Sistemas Operacionais: Sincronização

Wedson Almeida Filho

Outubro 2023

Instruções

Escolha uma das questões abaixo e implemente o código como descrito na questão. Escreva também um relatório contendo uma descrição do funcionamento da implementação, assim como uma reflexão pessoal sobre os desafios, dificuldades ou momentos chaves ocorridos durante o projeto ou implementação da solução.

Tanto o código final quanto o relatório serão apresentados interativamente ao professor, o que permitirá ao aluno expor o seu conhecimento sobre o assunto e discutir aspectos diversos sobre implementações alternativas, vantagens, desvantagens, etc.

Discussão sobre as diferentes questões ou até mesmo sobre possíveis soluções entre os alunos é aceitável e mesmo incentivada, mas as soluções e relatórios a serem apresentados ao professor deverão ser escritos individualmente por cada aluno.

O professor também está disponível para esclarecer dúvidas ou discutir aspectos das questões tanto em sala de aula quanto na sala 108 do DCA, normalmente entre as 9h e 11h da manhã, de segunda-feira a quinta-feira. Idealmente, mande um email no mais tardar no dia anterior para `wedson.almeida@dca.ufrn.br` para confirmar a disponibilidade.

Introdução

Considere a seguinte interface de programação (API) para um *mutex*:

```
struct mutex {
    /* ... */
};

void mutex_init(struct mutex *m);
void mutex_lock(struct mutex *m);
void mutex_unlock(struct mutex *m);
```

Onde `mutex_init` é uma função que inicializa o *mutex* para um estado destravado; `mutex_lock` é uma função que trava (adquire) o *mutex* de forma que no máximo uma *thread* por vez consiga entrar na seção crítica definida pelo *mutex*; e finalmente `mutex_unlock` é uma função que destrava (libera) o *mutex* de forma a encerrar a seção crítica iniciada por uma chamada anterior a `mutex_lock` – se houver alguma *thread* esperando pelo *mutex*, ela terá a oportunidade de tentar novamente adquirí-lo.

A estrutura *mutex* é disponibilizada aos usuários, ou seja, ela não é um tipo *incompleto* da linguagem C. Isso possibilita que os usuários da API possam instanciá-la na pilha de chamada (e.g., como variável local) ou mesmo dentro de outras estruturas de dados.

1 Variável de condição a partir de um *mutex*

Considere um ambiente onde a API para *mutex* descrita na introdução está disponível; implemente para esse ambiente uma API que disponibilize *variáveis de condição*. A API deve ser semelhante a:

```
struct condvar {
    /* ... */
};
```

```
void condvar_init(struct condvar *c);
void condvar_wait(struct condvar *c, struct mutex *m);
void condvar_signal(struct condvar *c);
void condvar_broadcast(struct condvar *c);
```

Onde `condvar_init` inicializa a variável de condição para um estado inicial onde não há nenhuma *thread* dormindo; `condvar_wait` atomicamente destrava um *mutex* e põe a *thread* corrente para dormir esperando ser acordada pela variável de condição; `condvar_signal` acorda no máximo **uma** *thread* que esteja dormindo na variável de condição devido a uma chamada anterior a `condvar_wait`; por fim, `condvar_broadcast` é similar a `condvar_signal` com a diferença que ela acorda **todas** as *threads* que estejam dormindo na variável de condição devido a chamadas anteriores a `condvar_wait`.

2 Variável de condição a partir de um semáforo

Considere um ambiente onde a seguinte API para semáforos está disponível:

```
struct sem {
    /* ... */
};

void sem_init(struct sem *s, unsigned val);
void sem_inc(struct sem *s);
void sem_dec(struct sem *s);
```

Onde `sem_init` inicializa o semáforo com o valor `val`; `sem_inc` incrementa o valor do semáforo em uma unidade ($valor_{novo} = valor + 1$), também acordando alguma *thread* que tenha dormido esperando para decrementar o valor; e finalmente `sem_dec` decrementa o valor do semáforo em uma unidade ($valor_{novo} = valor - 1$) garantindo que o valor nunca fique negativo, para tanto ela pode ter que pôr a *thread* para dormir quando o valor do semáforo for zero até que haja um outro incremento.

Implemente, para esse ambiente, a API para variáveis de condição descrita no item 1 acima. A diferença entre as questões 1 e 2 é o ponto de partida: na primeira questão a API de *mutex* está disponível, enquanto que na segunda a API de semáforo está disponível.

3 *Mutex* de leitura e escrita a partir de um *mutex*

Considere um ambiente onde a API para *mutex* descrita na introdução está disponível; implemente para esse ambiente uma API que disponibilize um *mutex* de leitura e escrita, que é um *mutex* que pode ser travado de duas formas diferentes: para escrita (também chamado de modo exclusivo), que é idêntico a um *mutex* tradicional, ou seja, no máximo uma *thread* por vez pode entrar na seção crítica; e para leitura (também chamado de modo compartilhado), onde múltiplas *threads* podem entrar na seção crítica contanto que elas também estejam em modo de leitura, *threads* que tentam adquirir o *mutex* em modo de escrita têm que esperar até que todos os leitores destravem.

A API deve ser semelhante a:

```
struct rwmutex {
    /* ... */
};

void rwmutex_init(struct rwmutex *m);
void rwmutex_read_lock(struct rwmutex *m);
void rwmutex_read_unlock(struct rwmutex *m);
void rwmutex_write_lock(struct rwmutex *m);
void rwmutex_write_unlock(struct rwmutex *m);
```

Onde `rwmutex_init` inicializa o *mutex* para um estado destravado. A função `rwmutex_read_lock` trava o *mutex* em modo de leitura de forma que outras *threads* possam também travá-lo imediatamente para leitura mas não possam travá-lo para escrita imediatamente, precisariam dormir. A função `rwmutex_read_unlock` destrava um *mutex* previamente travado com `rwmutex_read_lock`. A função `rwmutex_write_lock` trava

o *mutex* em modo de escrita de forma que nenhuma outra *thread* possa travar o mesmo *mutex* em qualquer modo, todas precisariam dormir. Por fim, `rwmutex_write_unlock` destrava um *mutex* previamente travado com `rwmutex_write_lock`.

4 *Mutex* a partir de um evento chaveado

Considere um ambiente onde a seguinte API para *eventos chaveados* está disponível.

```
struct evento_chaveado {
    /* ... */
};

void ec_init(struct evento_chaveado *e);
void ec_sleep(struct evento_chaveado *e, size_t chave);
void ec_wake(struct evento_chaveado *e, size_t chave);
```

Onde `ec_init` inicializa um evento chaveado de forma que não haja nenhuma *thread* acordando nem sendo acordada. `ec_sleep` põe a *thread* corrente para dormir até que outra *thread* a acorde usando o mesmo evento chaveado e chave. `ec_wake`, por sua vez, acorda uma *thread* que tenha dormido usando a chave especificada; entretanto, se não houver nenhuma *thread* dormindo nessa chave, a função `ec_wake` também faz com que a *thread* corrente durma até que outra *thread* chame `ec_sleep` com a mesma chave. Por isso motivo, eventos chaveados também são conhecidos como *rendezvous* porque requerem um encontro entre duas *threads*.

Implemente a API para *mutex* descrita na introdução usando API de eventos chaveados descrita acima.

5 *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.

6 *Mutex* a partir de uma `wait_queue_head_t` do Linux

Considere a seguinte API para filas de espera do núcleo do Linux:

```
void init_waitqueue_head(struct wait_queue_head *wq_head);
void init_wait_entry(struct wait_queue_entry *wq_entry, int flags);
void prepare_to_wait(struct wait_queue_head *wq_head,
                     struct wait_queue_entry *wq_entry, int state);
void schedule(void);
void finish_wait(struct wait_queue_head *wq_head,
                 struct wait_queue_entry *wq_entry);
int wake_up(struct wait_queue_head *wq_head);
```

Detalhes sobre a API podem ser encontrados neste link para a declaração da API. Em todo caso, a função `init_waitqueue_head` inicializa uma fila de espera para o estado vazio, e `init_wait_entry` inicializa um elemento que pode ser adicionado a uma fila de espera. A função `prepare_to_wait` adiciona um elemento (previamente inicializado) a uma fila de espera de forma que a *thread* poderá ser acordada no futuro quando chegar a sua vez na fila. A função `finish_wait` retira o elemento da fila, se ele ainda estiver na fila; pode ser usada, por exemplo, por uma *thread* que tenha desistido de dormir ou tenha acordado por algum outro motivo (por exemplo, porque recebeu um sinal). A função `schedule` chama o escalonador para escolher a próxima *thread* para rodar; se a *thread* que chamou estiver em uma fila, ela só será escolhida para rodar novamente depois de ter sido acordada. Por fim, `wake_up` acorda a *thread* associada ao primeiro elemento da fila de espera.

Implemente a API de *mutex* (descrita na introdução) para o núcleo do Linux usando a API de filas de espera descrita acima.

7 *Mutex* a partir de um `KEVENT` do Windows

Considere a seguinte API para eventos do núcleo do Windows:

```
void KeInitializeEvent(KEVENT *Event, EVENT_TYPE Type, BOOLEAN State);
NTSTATUS KeWaitForSingleObject(PVOID Object, KWAIT_REASON WaitReason,
                KPROCESSOR_MODE WaitMode, BOOLEAN Alertable,
                PLARGE_INTEGER Timeout);
LONG KeSetEvent(KEVENT *Event, KPRIORITY Increment, BOOLEAN Wait);
```

Onde `KeInitializeEvent` inicializa um evento com o tipo e estado inicial especificados, mais detalhes estão disponíveis neste link para a documentação. Uma vez que um evento tenha sido inicializado, a função `KeWaitForSingleObject` pode ser usada para esperar que o evento seja sinalizado, detalhes sobre os argumentos estão disponíveis neste link para a documentação. Por fim, a função `KeSetEvent` sinaliza o evento, assim acordado uma (ou mais, dependendo dos argumentos) *threads* que estejam esperando numa chamada anterior a `KeWaitForSingleObject`, mais detalhes estão disponíveis neste link para a documentação.

Note que a principal diferença entre eventos e eventos chaveados é que no caso do uso de eventos, a *thread* que chama `KeSetEvent` nunca dorme. Se não há nenhuma *thread* dormindo, o evento é posto em um estado sinalizado e a próxima *thread* que tentar esperar vai imediatamente completar a espera. Ou seja, o evento contém um estado que pode ser sinalizado ou não.

Implemente a API de *mutex* (descrita na introdução) para o núcleo do Windows usando a API de eventos descrita acima.