

MC514–Sistemas Operacionais: Teoria e Prática
1s2008

**Mutex locks simples, recursivos e
com verificação de erros**

Mutex locks

⇒ Exclusão mútua

- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Variáveis de condição

⇒ Sincronização

- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
- precisam ser utilizadas em conjunto com `mutex_locks`

Thread 0 acorda Thread 1

```
int s;                                /* Veja cond_signal.c */
```

Thread 1:

```
mutex_lock(&mutex);  
if (preciso_esperar(s))  
    cond_wait(&cond, &mutex);  
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);  
if (devo_acordar_thread_1(s))  
    cond_signal(&cond);  
mutex_unlock(&mutex);
```

Thread 0 acorda todas as threads

```
int s;          /* Veja cond_broadcast.c */
```

Thread i:

```
mutex_lock(&mutex);  
if (preciso_esperar(s))  
    cond_wait(&cond, &mutex);  
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);  
if (devo_acordar_todas_as_threads(s))  
    cond_broadcast(&cond);  
mutex_unlock(&mutex);
```

Thread 0 acorda todas as threads mas algumas delas voltam a dormir

```
int s;                                /* Veja cond_broadcast2.c */
```

Thread i:

```
mutex_lock(&mutex);  
while (preciso_esperar(s)) /* <===== */  
    cond_wait(&cond, &mutex);  
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);  
if (devo_acordar_todas_as_threads(s))  
    cond_broadcast(&cond);  
mutex_unlock(&mutex);
```

Thread 0 acorda 1 (ou +) threads

```
int s;                /* Veja cond_signal_n.c */
```

Thread i:

```
mutex_lock(&mutex);  
while (preciso_esperar(s))  
    cond_wait(&cond, &mutex);  
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);  
if (devo_acordar_pelo_menos_uma_thread(s))  
    cond_signal(&mutex);  
mutex_unlock(&mutex);
```

Importância do teste com while

- Cenário 1: Implementação não garante que apenas uma thread será acordada
- Cenário 2:
 - Thread i vai dormir pois C é verdadeira
 - Thread j acorda thread i pois torna C falsa
 - Thread k pega o lock e torna C verdadeira
 - Thread i executa de maneira inconsistente
 - Veja o código `teste_cond_wait.c`

Locks simples

Estrutura protegida por um mutex lock

```
typedef struct estrutura {  
    mutex_t lock;  
    Tipo1 campo1;  
    Tipo2 campo2;  
    Tipo3 campo3;  
} Estrutura;
```

- Como escrever as funções que fazem acesso a estes campos?

Locks simples

Funções atômicas

```
void funcao1(Estrutura *e) {  
    mutex_lock(&e->lock);  
    /* ... */  
    mutex_unlock(&e->lock);  
}
```

```
void funcao2(Estrutura *e) {  
    mutex_lock(&e->lock);  
    /* ... */  
    mutex_unlock(&e->lock);  
}
```

Locks simples

E se funcao2 invocasse funcao1?

```
void funcao2(Estrutura *e) {  
    mutex_lock(&e->lock);  
    /* ... */  
    if (condicao)  
        funcao1(e);  
    /* ... */  
    mutex_unlock(&e->lock);  
}
```

Deadlock de uma thread só

```
void f() {  
    mutex_lock(&lock);  
    mutex_lock(&lock);  
}
```

Veja o código: [deadlock.c](#)

Locks simples

E se funcao2 invocasse funcao1?

Possíveis soluções:

- Replicação de código
- Função auxiliar não atômica

```
void funcao1(Estrutura *e) {  
    mutex_lock(&e->lock);  
    aux_funcao1(e);  
    mutex_unlock(&e->lock);  
}
```

Locks recursivos

```
void f() {  
    mutex_lock(&lock);  
    /* faz alguma coisa */  
    mutex_unlock(&lock);  
}
```

```
void g() {  
    mutex_lock(&lock);  
    f();  
    /* faz outra coisa */  
    mutex_unlock(&lock);  
}
```

Locks recursivos

Implementação a partir de locks simples e variáveis de condição

```
typedef struct {  
    pthread_t thr;  
    cond_t cond;  
    mutex_t lock;  
    int c;  
} rec_mutex_t;
```

rec_mutex_lock()

```
int rec_mutex_lock(rec_mutex_t *rec_m) {  
    pthread_mutex_lock(&rec_m->lock);  
    if (rec_m->c == 0) { /* Lock livre */  
        rec_m->c = 1;  
        rec_m->thr = pthread_self();  
    } else /* Mesma thread */  
        if (pthread_equal(rec_m->thr,  
                           pthread_self()))  
            rec_m->c++;  
    else {  
        /* Thread deve esperar */  
    }
```

rec_mutex_lock()

```
else {
    /* Thread deve esperar */
    while (rec_m->c != 0)
        pthread_cond_wait(&rec_m->cond,
                           &rec_m->lock);
    rec_m->thr = pthread_self();
    rec_m->c = 1;
}
pthread_mutex_unlock(&rec_m->lock);
return 0;
}
```


rec_mutex_unlock()

```
int rec_mutex_unlock(rec_mutex_t *rec_m) {  
    pthread_mutex_lock(&rec_m->lock);  
    rec_m->c--;  
    if (rec_m->c == 0)  
        pthread_cond_signal(&rec_m->cond);  
    pthread_mutex_unlock(&rec_m->lock);  
    return 0;  
}
```

Verificação de erros

rec_mutex_unlock()

```
int rec_mutex_unlock(rec_mutex_t *rec_m) {  
    pthread_mutex_lock(&rec_m->lock);  
    if (rec_m->c == 0 ||  
        !pthread_equal(rec_m->thr,  
                        pthread_self())) {  
        pthread_mutex_unlock(&rec_m->lock_var);  
        return ERROR;  
    }  
    else  
        /* ... */  
}
```

Locks recursivos

Implementação reduzida

```
typedef struct {  
    pthread_t thr;  
    mutex_t lock;  
    int c;  
} rec_mutex_t;
```

rec_mutex_lock()

```
int rec_mutex_lock(rec_mutex_t *rec_m) {  
    if (!pthread_equal(rec_m->thr,  
                        pthread_self())) {  
        pthread_mutex_lock(&rec_m->lock);  
        rec_m->thr = pthread_self();  
        rec_m->c = 1;  
    }  
    else  
        rec_m->c++;  
    return 0;  
}
```

rec_mutex_unlock()

```
int rec_mutex_unlock(rec_mutex_t *rec_m) {  
    if (!pthread_equal(rec_m->thr, pthread_self())  
        || rec_m->c == 0)  
        return ERROR;  
    rec_m->c--;  
    if (rec_m->c == 0)  
        pthread_mutex_unlock(&rec_m->lock);  
    return 0;  
}
```

- A implementação reduzida tem comportamento equivalente à primeira?
- Veja o código `pthread_mutex_lock.c`
- Quando que as variáveis de condição são imprescindíveis?