

# **Processos e Threads**

**Mutex locks e variáveis de condição**

# Semáforos

- Exclusão mútua
- Sincronização

# Exemplo de exclusão mútua

## Leitores e escritores

```
semaforo db = 1;
```

### **Leitor:**

```
while(true)
    wait(db);
    le_db();
    signal(db);
```

### **Escritor:**

```
while(true)
    wait(db);
    escreve_db();
    signal(db);
```

# Exemplo de sincronização

## Produtor-Consumidor

```
semaforo cheio = 0, vazio = N;
```

Produtor:

```
while (true)
    wait(vazio);
    f = (f + 1) % N;
    buffer[f] = produz();
    signal(cheio);
```

Consumidor:

```
while (true)
    wait(cheio);
    i = (i + 1) % N;
    consome(buffer[i]);
    signal(vazio);
```

# Mutex locks

⇒ Exclusão mútua

- `pthread_mutex_lock`
- `pthread_mutex_unlock`

# Exemplo de exclusão mútua

## Leitores e escritores

```
mutex_lock db = 1;
```

### **Leitor:**

```
while(true)
    mutex_lock(db);
    le_db();
    mutex_unlock(db);
```

### **Escritor:**

```
while(true)
    mutex_lock(db);
    escreve_db();
    mutex_unlock(db);
```

# Variáveis de condição

⇒ Sincronização

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

## Exemplo com wait/signal

```
int s; /* Variável compartilhada */
```

### **Thread 1:**

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

### **Thread 0:**

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



## Exemplo com wait/signal

```
int s; /* Variável compartilhada */
```

### **Thread i:**

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

### **Thread 0:**

```
mutex_lock(&lock);  
if (devo_acordar_alguma_thread(s))  
    cond_signal(&cond);  
mutex_unlock(&lock);
```

## Exemplo com wait/broadcast

```
int s; /* Variável compartilhada */
```

### **Thread i:**

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

### **Thread 0:**

```
mutex_lock(&lock);  
if (devo_acordar_uma_ou_mais_threads(s))  
    cond_broadcast(&cond);  
mutex_unlock(&lock);
```

# **Poder computacional equivalente**

**É possível implementar...**

- semáforos utilizando mutex locks e variáveis de condição
- mutex locks e variáveis de condição utilizando semáforos

# Implementação de semáforos usando mutex locks e variáveis de condição

```
typedef struct {  
    int value;      /* Valor atual do semáforo */  
    int n_wait;    /* Número de threads esperando */  
    mutex_t lock;  
    cond_t cond;  
} sem_t;
```

## **sem\_init**

```
int sem_init(sem_t *sem, int pshared,
             unsigned int value) {
    sem->value = value;
    sem->n_wait = 0;
    mutex_init(&sem->lock, NULL);
    cond_init(&sem->cond, NULL);
    return 0;
}
```

## sem\_wait

```
int sem_wait(sem_t * sem) {
    mutex_lock(&sem->lock);
    if (sem->value > 0)
        sem->value--;
    else {
        sem->n_wait++;
        cond_wait(&sem->cond, &sem->lock);
    }
    mutex_unlock(&sem->lock);
    return 0;
}
```

## sem\_trywait

```
int sem_trywait(sem_t * sem) {
    int r;
    mutex_lock(&sem->lock);
    if (sem->value > 0) {
        sem->value--;
        r = 0;
    } else
        r = EAGAIN;
    mutex_unlock(&sem->lock);
    return r;
}
```

## sem\_post

```
int sem_post(sem_t * sem) {
    mutex_lock(&sem->lock);
    if (sem->n_wait) {
        sem->n_wait--;
        cond_signal(&sem->cond);
    } else
        sem->value++;
    mutex_unlock(&sem->lock);
    return 0;
}
```



## **sem\_getvalue**

```
int sem_getvalue(sem_t *sem, int *sval) {  
    mutex_lock(&sem->lock);  
    *sval = sem->value;  
    mutex_unlock(&sem->lock);  
    return 0;  
}
```

## **sem\_destroy**

```
int sem_destroy(sem_t *sem) {  
    if (sem->n_wait)  
        return EBUSY;  
    mutex_destroy(&sem->lock);  
    cond_destroy(&sem->cond);  
    return 0;  
}
```

# Implementação de mutex locks utilizando semáforos Sem verificação de erros

```
typedef struct {  
    sem_t sem;  
} mutex_t;
```

## mutex\_init e mutex\_destroy

```
int mutex_init(mutex_t *lock, mutex_attr* attr) {  
    return sem_init(&lock->sem, 0, 1);  
}
```

```
int mutex_destroy(mutex_t *lock) {  
    return sem_destroy(&lock->sem);  
}
```

## mutex\_lock e mutex\_unlock

```
int mutex_lock(mutex_t *lock) {  
    return sem_wait(&lock->sem);  
}
```

```
int mutex_unlock(mutex_t *lock) {  
    return sem_post(&lock->sem);  
}
```

# Implementação de variáveis de condição usando locks e semáforos

```
typedef struct {  
    mutex_t lock;  
    sem_t sem;  
    int n_wait;  
} cond_t;
```

## cond\_init

```
int cond_init(cond_t *cond) {  
    mutex_init(&cond->lock, NULL);  
    sem_init(&cond->sem, 0, 0);  
    n_wait = 0;  
    return 0;  
}
```

## cond\_signal

```
int cond_signal(cond_t *cond) {
    mutex_lock(&cond->lock);
    if (cond->n_wait > 0) {
        cond->n_wait--;
        sem_post(&cond->sem);
    }
    mutex_unlock(&cond->lock);
    return 0;
}
```



## cond\_broadcast

```
int cond_broadcast(cond_t *cond) {
    mutex_lock(&cond->lock);
    while (cond->n_wait > 0) {
        cond->n_wait--;
        sem_post(&cond->sem);
    }
    mutex_unlock(&cond->lock);
    return 0;
}
```

## cond\_wait

```
int cond_wait(cond_t *cond,  
              mutex_t *mutex_externo) {  
    mutex_lock(&cond->lock);  
    cond->n_wait++;  
    mutex_unlock(&cond->lock);  
    mutex_unlock(mutex_externo);  
    sem_wait(&cond->sem);  
    mutex_lock(mutex_externo);  
    return 0;  
}
```

## Se são equivalentes, como optar?

- Mutex locks e variáveis de condição
  - Separação clara entre sincronização e exclusão mútua
  - Mais fácil de expressar condições complexas para bloqueio
- Semáforos
  - Representação mais compacta para contadores

# Produtor-Consumidor

## Primeira tentativa

```
cond_t pos_vazia, pos_ocupada; mutex_t lock;  
int i = 0, f = 0, c = 0;
```

### **Produtor:**

```
    mutex_lock(&lock);  
    if (c == N)  
        cond_wait(&pos_vazia, &lock);  
    f = (f+1)%N;  
    buffer[f] = produz();  
    c++;  
    if (c == 1)  
        cond_signal(&pos_ocupada);  
    mutex_unlock(&lock);
```

# Produtor-Consumidor

## Primeira tentativa

### Consumidor:

```
mutex_lock(&lock);
if (c == 0)
    cond_wait(&pos_ocupada, &lock);
i = (i+1)%N;
consume(buffer[i]);
if (c == N-1)
    cond_signal(&pos_vazia);
c--;
mutex_unlock(&lock);
```

# Produtor-Consumidor

## Paralelismo entre as threads

```
semaforo cheio = 0, vazio = N;
```

Produtor:

```
while (true)
    wait(vazio);
    f = (f + 1) % N;
    buffer[f] = produz();
    signal(cheio);
```

Consumidor:

```
while (true)
    wait(cheio);
    i = (i + 1) % N;
    consome(buffer[i]);
    signal(vazio);
```

# Produtor-Consumidor

```
cond_t pos_vazia, pos_ocupada; mutex_t lock_v, lock_o;  
int i = 0, f = 0, nv = N, no = 0;
```

## Produtor:

```
    mutex_lock(&lock_v);  
    if (nv == 0) cond_wait(&pos_vazia, &lock_v);  
    nv--;  
    mutex_unlock(&lock_v);  
    f = (f+1)%N;  
    buffer[f] = produz();  
    mutex_lock(&lock_o);  
    no++;  
    cond_signal(&pos_ocupada);  
    mutex_unlock(&lock_o);
```

# Produtor-Consumidor

## Consumidor:

```
mutex_lock(&lock_o);  
if (no == 0) cond_wait(&pos_ocupada, &lock_o);  
no--;  
mutex_unlock(&lock_o);  
i = (i+1)%N;  
consume(buffer[i]);  
mutex_lock(&lock_v);  
nv++;  
cond_signal(&pos_vazia);  
mutex_unlock(&lock_v);
```



# Produtores-Consumidores

## Locks e variáveis de condição

```
cond_t pos_vazia, pos_ocupada;
```

```
mutex_t lock_v, lock_o;
```

```
int nv = N, no = 0;
```

```
mutex_t lock_i, lock_f;
```

```
int i = 0, f = 0;
```

# Produtores-Consumidores

## Produtor:

```
item = produz();  
mutex_lock(&lock_v);  
while (nv == 0) cond_wait(&pos_vazia, &lock_v);  
nv--;  
mutex_unlock(&lock_v);  
mutex_lock(&lock_f);  
f = (f+1)%N;  
buffer[f] = item;  
mutex_unlock(&lock_f);  
mutex_lock(&lock_o);  
no++;  
cond_signal(&pos_ocupada);  
mutex_unlock(&lock_o);
```

# Produtores-Consumidores

## Consumidor:

```
mutex_lock(&lock_o);
while (no == 0) cond_wait(&pos_ocupada, &lock_o);
no--;
mutex_unlock(&lock_o);
mutex_lock(&lock_i);
i = (i+1)%N;
item = buffer[i];
mutex_unlock(&lock_i);
mutex_lock(&lock_v);
nv++;
cond_signal(&pos_vazia);
mutex_unlock(&lock_v);
consome(item);
```

# Produtores-Consumidores

## Solução híbrida

```
sem_t pos_vazia = N, pos_ocupada;
```

```
mutex_t lock_i, lock_f;
```

```
int i = 0, f = 0;
```

# Produtores-Consumidores

## Produtor:

```
item = produz();  
wait(&pos_vazia);  
mutex_lock(&lock_f);  
f = (f+1)%N;  
buffer[f] = item;  
mutex_unlock(&lock_f);  
signal(&pos_ocupada);
```

# Produtores-Consumidores

## Consumidor:

```
wait(&pos_ocupada);  
mutex_lock(&lock_i);  
i = (i+1)%N;  
item = buffer[i];  
mutex_unlock(&lock_i);  
signal(&pos_vazia);  
consome(item);
```

# Filósofos (Tanenbaum)

## Como reescrever este código com locks e variáveis de condição?

```
semaforo lock;  
semaforo filosofo[N] = {0, 0, 0, ..., 0}  
int estado[N] = {T, T, T, ..., T}
```

### Filósofo i:

```
while (true)  
    pensa();  
    pega_garfos();  
    come();  
    solta_garfos();
```

```
testa_garfos(int i)
    if (estado[i] == H && estado[fil_esq] != E &&
        estado[fil_dir] != E)
        estado[i] = E;
        signal(filosofo[i]);
```

```
pega_garfos()
    wait(lock);
    estado[i] = H;
    testa_garfos(i);
    signal(lock);
    wait(filosofo[i]);
```

```
solta_garfos()
    wait(lock);
    estado[i] = T;
    testa_garfos(fil_esq);
    testa_garfos(fil_dir);
    signal(lock);
```