Chapter 31: Semaphores

Overview

- **Main Idea**: Semaphores provide a flexible synchronization primitive combining state and atomic operations.
- **Key Terms**: `wait()` (P), `post()` (V), binary semaphore, counting semaphore.

---

1. Semaphore Basics

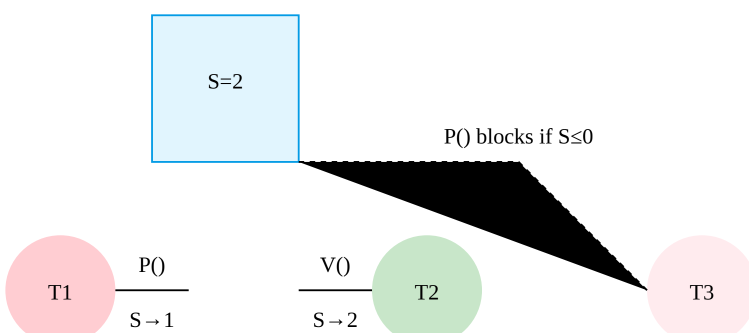
Core Operations

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);  // Initialize to 1 (binary semaphore)

sem_wait(&s);
sem_post(&s);  // V(): Increment and wake a waiter
```

Comparison with Locks/CVs

Feature	Semaphore	Lock	Condition Variable
State	Integer value	Binary (held/not)	No internal state
Wakeup	Automatic	N/A	Explicit signal
Usage	Standalone	+ CV for waiting	Requires lock



---

2. Binary Semaphores (Locks)

```
sem_t mutex;
sem_init(&mutex, 0, 1);  // Initial value=1 (available)

sem_wait(&mutex);  // Lock
// Critical section
sem_post(&mutex);  // Unlock
```

Behavior:

- Initial value=1: Available lock
- Value=0: Lock held, waiters blocked

---

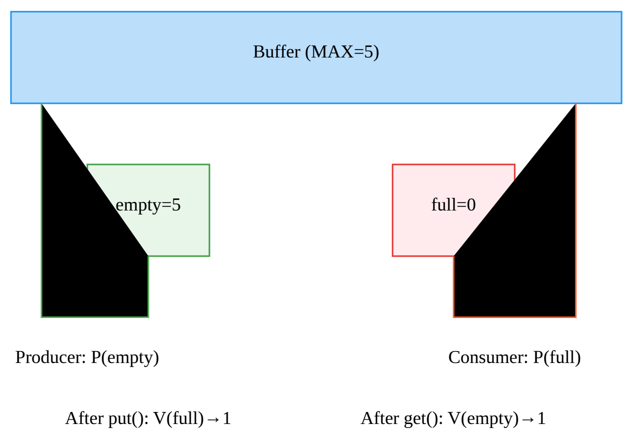
3. Counting Semaphores

Producer-Consumer (Bounded Buffer)

```
sem_t empty, full;
sem_init(&empty, 0, MAX);  // MAX empty slots
sem_init(&full, 0, 0);     // 0 full slots

void *producer(void *arg) {
    sem_wait(&empty);  // Wait for empty slot
    // Add item to buffer
    sem_post(&full);   // Signal new item
}

void *consumer(void *arg) {
    sem_wait(&full);   // Wait for item
    // Remove item from buffer
    sem_post(&empty);  // Signal free slot
}
```



---

4. Semaphore Implementation

DIY Semaphore with Locks/CVs

```c
typedef struct {
    int value;
    pthread_mutex_t lock;
    pthread_cond_t cond;
} zem_t;

void zem_wait(zem_t *s) {
    pthread_mutex_lock(&s->lock);
    while (s->value <= 0)
        pthread_cond_wait(&s->cond, &s->lock);
    s->value--;
    pthread_mutex_unlock(&s->lock);
}

void zem_post(zem_t *s) {
    pthread_mutex_lock(&s->lock);
    s->value++;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->lock);
}
```

Key Insight: Semaphores can be built from lower-level primitives.

---

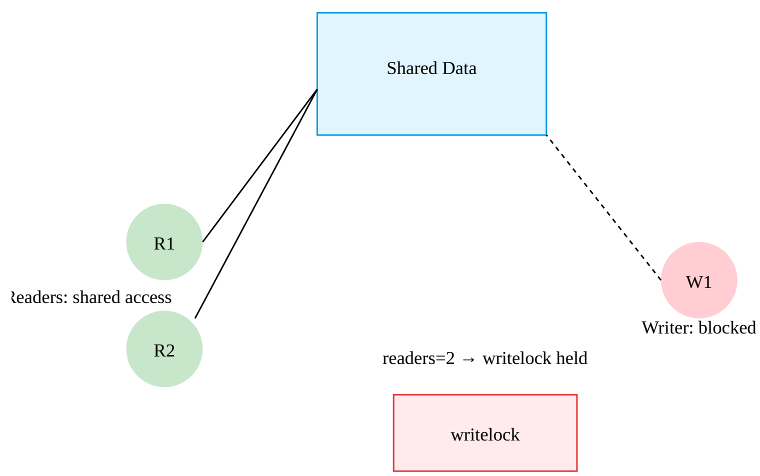
5. Classic Problems

5.1 Reader-Writer Locks

```c
sem_t writelock, lock;
int readers = 0;

void reader_enter() {
    sem_wait(&lock);
    readers++;
    if (readers == 1)
        sem_wait(&writelock);  // First reader blocks writers
    sem_post(&lock);
}

void writer_enter() {
    sem_wait(&writelock);      // Exclusive access
}
```



5.2 Dining Philosophers

```c
sem_t forks[5];
sem_t room;  // Allow only 4 philosophers

void philosopher(int i) {
    sem_wait(&room);
    sem_wait(&forks[i]);
    sem_wait(&forks[(i+1)%5]);
```

```
    // Eat
    sem_post(&forks[i]);
    sem_post(&forks[(i+1)%5]);
    sem_post(&room);
}
```

Solution: Prevents deadlock by limiting concurrent access.

---

6. Semaphore Pitfalls

1. **Incorrect Initialization**:

   ```
   sem_init(&s, 0, 0);  // Deadlock if first operation is wait()
   ```

2. **Missing Post**:

   ```
   sem_wait(&s);
   // Critical section
   // Forgot sem_post!
   ```

3. **Order Violation**:

   ```
   Thread1: sem_post(&s);  // Should wait for Thread2's wait()
   Thread2: sem_wait(&s);
   ```

---

7. Summary Table

Use Case	Semaphore Type	Initial Value
Mutual Exclusion	Binary	1
Resource Pool	Counting	N (resources)
Signaling	Binary	0

Homework Insights

1. **Semaphore Chains**: Implement thread ordering with semaphore sequences.
2. **Barrier Problem**: Use semaphores to synchronize N threads at a point.
3. **Performance**: Compare semaphore vs. CV implementations.