IE411: Operating Systems

Synchronization patterns, Equivalence

Signaling pattern

Thread A	Thread B
`1. statement a1`	`1. statement b1`

- Want to guarantee
  - a1 happens before b1
- Idea
  - use a semaphore to indicate whether or not A has finished a1

Signaling pattern

Thread A	Thread B
1. statement a1	1. statement b1

- Want to guarantee
  - a1 happens before b1
- Idea
  - use a semaphore to indicate whether or not A has finished a1
- What should we set the semaphore value to?

Signaling pattern

Thread A

```
1. statement a1
2. V(aDone)
```

Thread B

```
1. P(aDone)
2. statement b1
```

- Semaphore aDone (initially 0)
- B has to wait until A executes a1

Signaling pattern

Thread A

```
1. statement a1
2. V(aDone)
```

Thread B

```
1. P(aDone)
2. statement b1
```

- Semaphore aDone (initially 0)
- B has to wait until A executes a1
  - so let B issue P(aDone) before b1

Signaling pattern

Thread A

```
1. statement a1
2. V(aDone)
```

Thread B

```
1. P(aDone)
2. statement b1
```

- Semaphore aDone (initially 0)
- B has to wait until A executes a1
  - so let B issue P(aDone) before b1
  - and A issue V(aDone) after a1 to signal B that it can execute b1

Signaling pattern

Thread A

```
1. statement a1
2. Statement a2
```

Thread B

```
1. statement b1
```

- Want to guarantee
  - a1 happens before b1
  - b1 happens before a2

Signaling pattern

- What is wrong with this solution?

```
1. statement a1
2. V(Done)
3. P(Done)
4. statement a2
```

```
1. P(Done)
2. statement b1
3. V(Done)
```

Signaling pattern

- What is wrong with this solution?

```
1. statement a1
2. V(Done)
3. P(Done)
4. statement a2
```

```
1. P(Done)
2. statement b1
3. V(Done)
```

- Can't guarantee b1 happens before a2
  - unless B does its P() operation before A reaches its P() operation

Signaling pattern

- Idea: let A wait for B using a different semaphore

Thread A

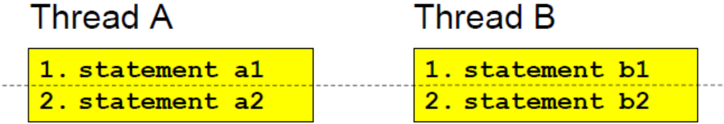
```
1. statement a1
2. V(aDone)
3. P(bDone)
4. statement a2
```

Thread B

```
1. P(aDone)
2. statement b1
3. V(bDone)
```

- finally correct?

Rendezvous



- Want to guarantee
  - a1 happens before b2
  - b1 happens before a2
  - we don't care about the order of a1 and b1

Rendezvous

- Idea: use two semaphores, aArrived, bArrived (both initialized to 0)

```
1. statement a1
2. V(aArrived)
3. P(bArrived)
4. statement a2
```

```
1. statement b1
2. V(bArrived)
3. P(aArrived)
4. statement b2
```

- Does this solution work?

Rendezvous (variant 1)

- Suppose we have B perform its operations in the opposite order

```
1. statement a1
2. V(aArrived)
3. P(bArrived)
4. Statement a2
```

```
1. statement b1
2. P(aArrived)
3. V(bArrived)
4. Statement b2
```

- Does this solution work?

Rendezvous (variant 2)

- Yet another variant

```
1. statement a1
2. P(bArrived)
3. V(aArrived)
4. statement a2
```

```
1. statement b1
2. P(aArrived)
3. V(bArrived)
4. statement b2
```

- Does this solution work?

Rendezvous (variant 2)

- Yet another variant

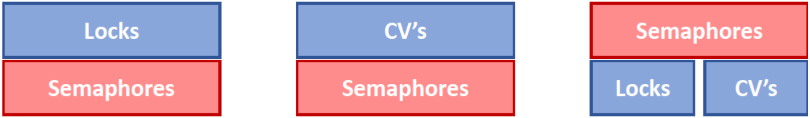
```
1. statement a1
2. P(bArrived)
3. V(aArrived)
4. statement a2
```

```
1. statement b1
2. P(aArrived)
3. V(bArrived)
4. statement b2
```

- Does this solution work?
- Deadlocks are possible
  - Assuming A arrives first, it will block at its P
  - When B arrives, it will also block, since A wasn't able to signal aArrived
  - So, neither thread can proceed, and never will

Equivalence

- Claim: Semaphores are equally powerful as lock+CVs
- This means we can build each out of the other



Lock implementation using semaphores

- Finish this implementation

```
typedef struct {

} lock_t;

void init(lock_t *lock) {

}
void acquire(lock_t *lock) {

}
void release(lock_t *lock) {

}
```

Lock implementation using semaphores

- Semaphore is initialized to ____

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

void init(lock_t *lock) {
    sem_init(&lock->sem, ??);
}
void acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}
void release(lock_t *lock) {
    sem_post(&lock->sem);
}
```

Lock implementation using semaphores

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

void init(lock_t *lock) {
    sem_init(&lock->sem, 1);
}
void acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}
void release(lock_t *lock) {
    sem_post(&lock->sem);
}
```

CV implementation using semaphores (attempt 1)

- Finish this implementation using semaphores and locks

```
typedef struct {
    sem_t sem;              // initially 0
    lock_t lock;
} cond_t;

void cond_wait(cond_t *c) {
    // assumes that lock is held
    ??
}

void cond_signal(cond_t *c) {
    ??
}
```

CV implementation using semaphores (attempt 1)

- You might have tried ...

```
typedef struct {
    sem_t sem;            // initially 0
    lock_t lock;
} cond_t;

void cond_wait(cond_t *c) { // assumes lock is held
    release(&c->lock);      // release lock and go to sleep
    sem_wait(&c->sem);
    acquire(&c->lock);      // grab lock before returning
}

void cond_signal(cond_t *c) {
    sem_post(&c->sem);      // wake up a sleeping waiter
}
```

- This solution is incorrect (why?)

CV implementation using semaphores (attempt 1)

- You might have tried . . .

```
typedef struct {
   sem_t sem;           // initially 0
   lock_t lock;
} cond_t;

void cond_wait(cond_t *c) { // assumes lock is held
   release(&c->lock);       // release lock and go to sleep
   sem_wait(&c->sem);
   acquire(&c->lock);       // grab lock before returning
}

void cond_signal(cond_t *c) {
   sem_post(&c->sem);       // wake up a sleeping waiter
}
```

- This solution is incorrect (why?)
  - cond_signal wakes up threads in the far future!

CV implementation using semaphores (attempt 2)

- Finish this implementation using semaphores and locks

```
typedef struct {
    sem_t sem;              // initially 0
    lock_t lock;
    lock_t priv_lock;       // initially 1
    int num_waiters;        // initially 0
} cond_t;

void cond_wait(cond_t *c) {
    // assumes that lock is held
    ??
}

void cond_signal(cond_t *c) {
    ??
}
```

CV implementation using semaphores (attempt 2)

- On the whiteboard