

CS 33

Multithreaded Programming II

Mutual Exclusion



Threads and Mutual Exclusion

Thread 1:

```
x = x+1;  
/*  
    movl x,%eax  
    addl $1,%eax  
    movl %eax,x  
*/
```

Thread 2:

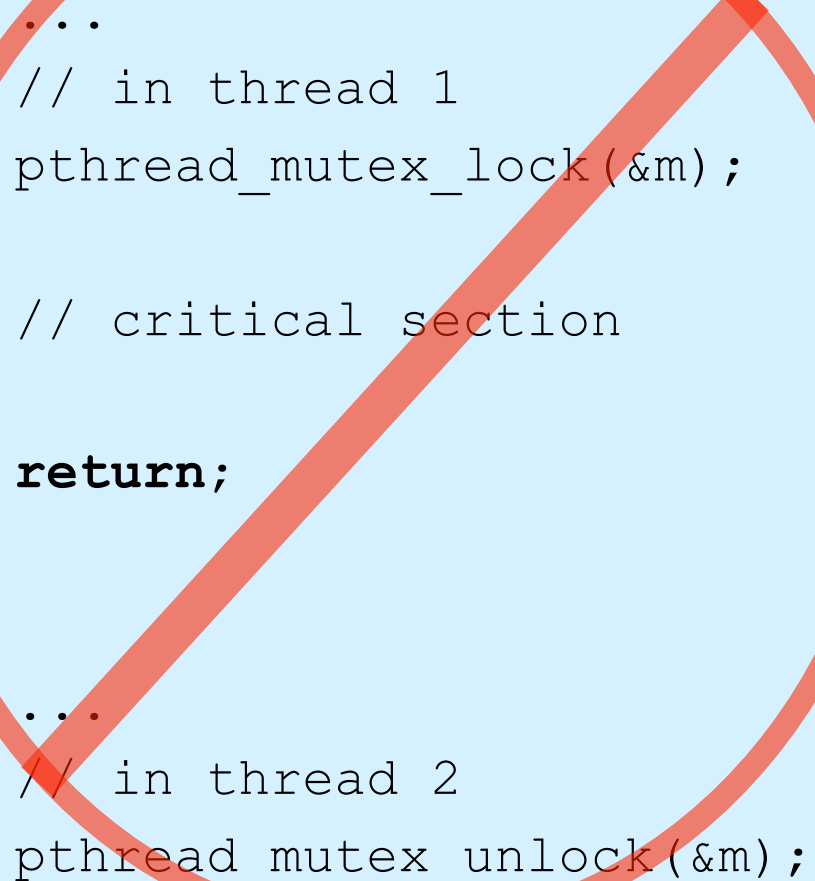
```
x = x+1;  
/*  
    movl x,%eax  
    addl $1,%eax  
    movl %eax,x  
*/
```

POSIX Threads Mutual Exclusion

```
pthread_mutex_t m =  
    PTHREAD_MUTEX_INITIALIZER;  
    // shared by both threads  
int x; // ditto  
  
pthread_mutex_lock(&m);  
  
x = x+1;  
  
pthread_mutex_unlock(&m);
```

Correct Usage

```
pthread_mutex_lock(&m);  
  
// critical section  
  
pthread_mutex_unlock(&m);
```



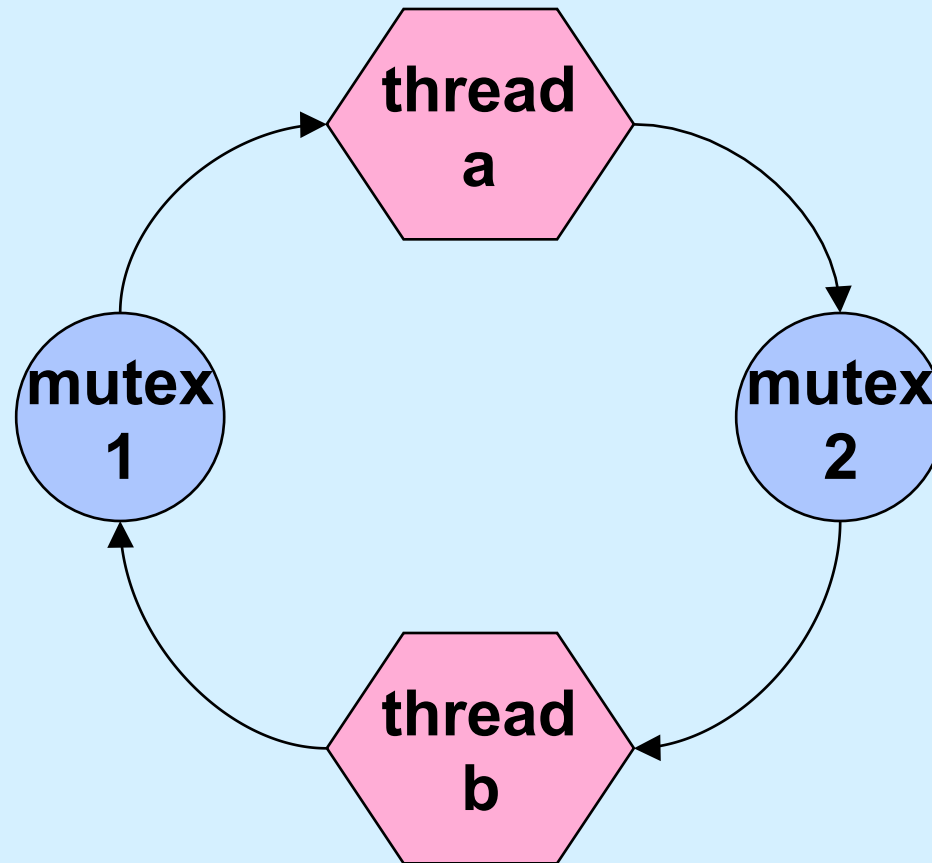
```
...  
// in thread 1  
pthread_mutex_lock(&m);  
  
// critical section  
  
return;  
  
...  
// in thread 2  
pthread_mutex_unlock(&m);
```

Taking Multiple Locks

```
proc1( ) {  
    pthread_mutex_lock(&m1);  
    /* use object 1 */  
    pthread_mutex_lock(&m2);  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m2);  
    pthread_mutex_unlock(&m1);  
}
```

```
proc2( ) {  
    pthread_mutex_lock(&m2);  
    /* use object 2 */  
    pthread_mutex_lock(&m1);  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m1);  
    pthread_mutex_unlock(&m2);  
}
```

Preventing Deadlock

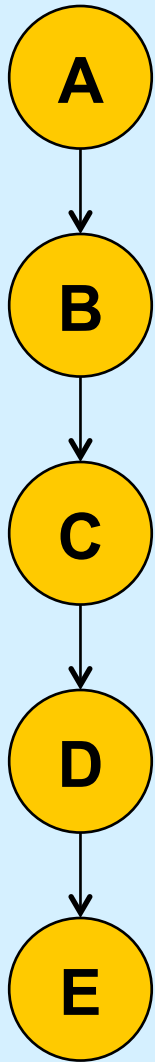


Taking Multiple Locks, Safely

```
proc1( ) {  
    pthread_mutex_lock(&m1);  
    /* use object 1 */  
    pthread_mutex_lock(&m2);  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m2);  
    pthread_mutex_unlock(&m1);  
}
```

```
proc2( ) {  
    pthread_mutex_lock(&m1);  
    /* use object 1 */  
    pthread_mutex_lock(&m2);  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m2);  
    pthread_mutex_unlock(&m1);  
}
```

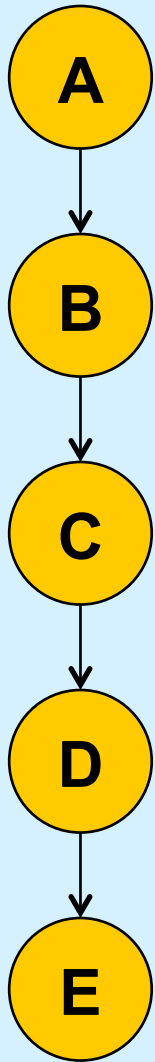

Singly Linked List: Coarse-Grained



```
typedef struct node {  
    int value;  
    struct node *next;  
} node_t;  
pthread_mutex_t global_mutex;
```

```
void add_after(node_t *after, node_t *new) {  
    pthread_mutex_lock(&global_mutex);  
    new->next = after->next;  
    after->next = new;  
    pthread_mutex_unlock(&global_mutex);  
}
```

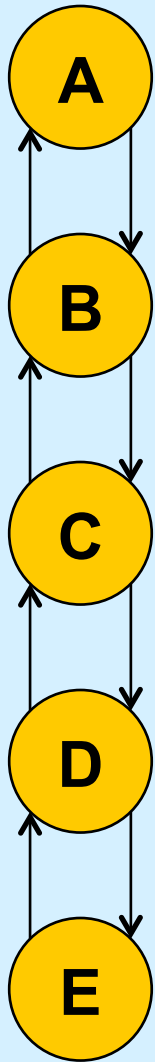
Singly Linked List: Fine-Grained



```
typedef struct node {  
    pthread_mutex_t mutex;  
    int value;  
    struct node *next;  
} node_t;
```

```
void add_after(node_t *after, node_t *new) {  
    pthread_mutex_lock(&after->mutex);  
    new->next = after->next;  
    after->next = new;  
    pthread_mutex_unlock(&after->mutex);  
}
```

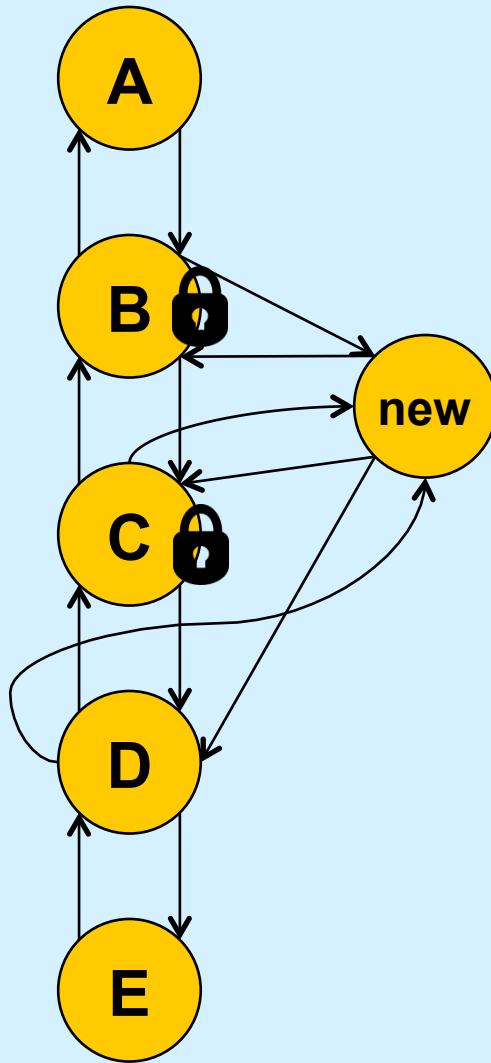
Doubly Linked List



```
void add_after(node_t *after, node_t *new) {  
    pthread_mutex_lock(&after->mutex);  
    after->next->prev = new;  
    new->next = after->next;  
    new->prev = after;  
    after->next = new;  
    pthread_mutex_unlock(&after->mutex);  
}
```

```
void delete(node_t *old) {  
    pthread_mutex_lock(&old->mutex);  
    old->prev->next = old->next;  
    old->next->prev = old->prev;  
    pthread_mutex_unlock(&old->mutex);  
}
```

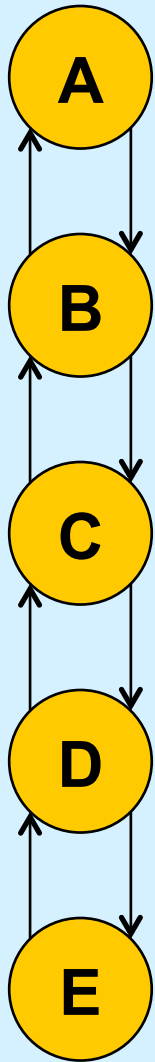
Doesn't Work ...



```
void add_after(node_t *after, node_t *new) {  
    pthread_mutex_lock(&after->mutex);  
    after->next->prev = new;  
    new->next = after->next;  
    new->prev = after;  
    after->next = new;  
    pthread_mutex_unlock(&after->mutex);  
}
```

```
void delete(node_t *old) {  
    pthread_mutex_lock(&old->mutex);  
    old->prev->next = old->next;  
    old->next->prev = old->prev;  
    pthread_mutex_unlock(&old->mutex);  
}
```

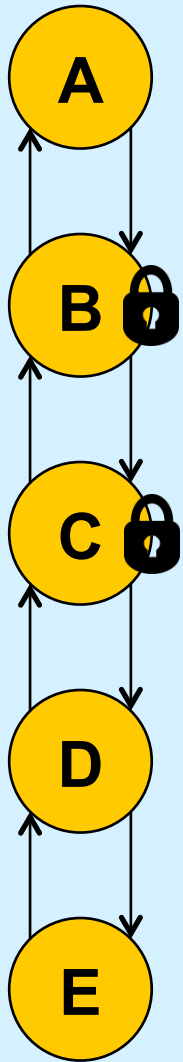
Doubly Linked List, Attempt 2



```
void add_after(node_t *after, node_t *new) {
    pthread_mutex_lock(&after->mutex);
    pthread_mutex_lock(&after->next->mutex);
    after->next->prev = new;
    new->next = after->next;
    new->prev = after;
    after->next = new;
    pthread_mutex_unlock(&new->next->mutex);
    pthread_mutex_unlock(&after->mutex);
}
```

```
void delete(node_t *old) {
    pthread_mutex_lock(&old->mutex);
    pthread_mutex_lock(&old->prev->mutex);
    pthread_mutex_lock(&old->next->mutex);
    old->prev->next = old->next;
    old->next->prev = old->prev;
    pthread_mutex_unlock(&old->next->mutex);
    pthread_mutex_unlock(&old->prev->mutex);
    pthread_mutex_unlock(&old->mutex);
}
```

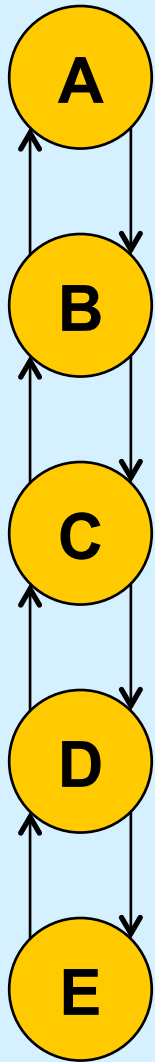
Doesn't Work ...



```
void add_after(node_t *after, node_t *new) {
    pthread_mutex_lock(&after->mutex);
    pthread_mutex_lock(&after->next->mutex);
    after->next->prev = new;
    new->next = after->next;
    new->prev = after;
    after->next = new;
    pthread_mutex_unlock(&new->next->mutex);
    pthread_mutex_unlock(&after->mutex);
}
```

```
void delete(node_t *old) {
    pthread_mutex_lock(&old->mutex);
    pthread_mutex_lock(&old->prev->mutex);
    pthread_mutex_lock(&old->next->mutex);
    old->prev->next = old->next;
    old->next->prev = old->prev;
    pthread_mutex_unlock(&old->next->mutex);
    pthread_mutex_unlock(&old->prev->mutex);
    pthread_mutex_unlock(&old->mutex);
}
```

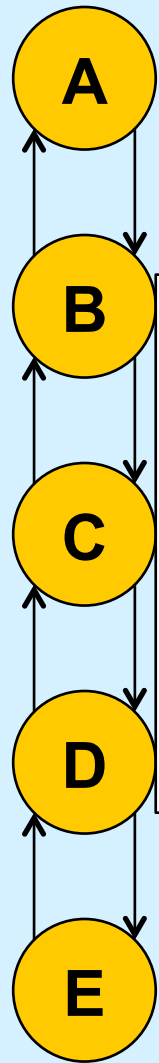
Doubly Linked List, Attempt 3



```
void add_after(node_t *after, node_t *new) {  
    pthread_mutex_lock(&after->mutex);  
    pthread_mutex_lock(&after->next->mutex);  
    after->next->prev = new;  
    new->next = after->next;  
    new->prev = after;  
    after->next = new;  
    pthread_mutex_unlock(&new->next->mutex);  
    pthread_mutex_unlock(&after->mutex);  
}
```

```
void delete(node_t *old) {  
    pthread_mutex_lock(&old->prev->mutex);  
    pthread_mutex_lock(&old->mutex);  
    pthread_mutex_lock(&old->next->mutex);  
    old->prev->next = old->next;  
    old->next->prev = old->prev;  
    pthread_mutex_unlock(&old->next->mutex);  
    pthread_mutex_unlock(&old->mutex);  
    pthread_mutex_unlock(&old->prev->mutex);  
}
```

Doubly Linked List, Attempt 3



Quiz 1

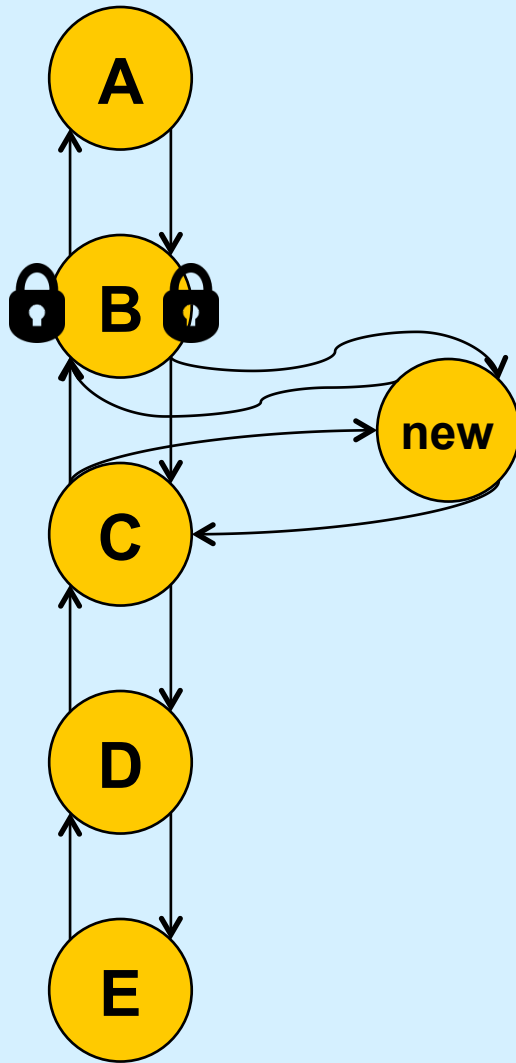
Does this work?

- a) yes
- b) no

```
void add_after(node_t *after, node_t *new) {  
    pthread_mutex_lock(&after->mutex);  
    pthread_mutex_lock(&after->next->mutex);  
    after->next->prev = new;  
    new->next = after->next;  
    new->prev = after;  
    after->next = new;  
    pthread_mutex_unlock(&new->next->mutex);  
    pthread_mutex_unlock(&after->mutex);  
}
```

```
void delete(node_t *old) {  
    pthread_mutex_lock(&old->prev->mutex);  
    pthread_mutex_lock(&old->mutex);  
    pthread_mutex_lock(&old->next->mutex);  
    old->prev->next = old->next;  
    old->next->prev = old->prev;  
    pthread_mutex_unlock(&old->next->mutex);  
    pthread_mutex_unlock(&old->mutex);  
    pthread_mutex_unlock(&old->prev->mutex);  
}
```


Doubly Linked List

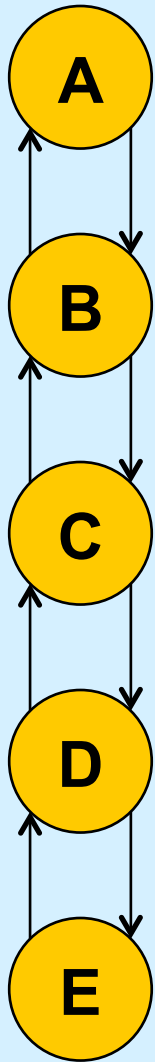


More Machinery ...

```
proc1( ) {  
    pthread_mutex_lock(&m1);  
    /* use object 1 */  
    pthread_mutex_lock(&m2);  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m2);  
    pthread_mutex_unlock(&m1);  
}
```

```
proc2( ) {  
    while (1) {  
        pthread_mutex_lock(&m2);  
  
        if (!pthread_mutex_trylock(&m1))  
            break;  
        pthread_mutex_unlock(&m2);  
    }  
  
    /* use objects 1 and 2 */  
  
    pthread_mutex_unlock(&m1);  
    pthread_mutex_unlock(&m2);  
}
```

Doubly Linked List



```
void delete(node_t *old) {
    while(1) {
        pthread_mutex_lock(&old->mutex);
        if (pthread_mutex_trylock(&old->prev->mutex) != 0) {
            pthread_mutex_unlock(&old->mutex);
            continue;
        }
        break;
    }
    pthread_mutex_lock(&old->next->mutex);
    old->prev->next = old->next;
    old->next->prev = old->prev;
    pthread_mutex_unlock(&old->next->mutex);
    pthread_mutex_unlock(&old->mutex);
    pthread_mutex_unlock(&old->prev->mutex);
}
```

Dining Philosophers Problem



Practical Issues with Mutexes

- **Used a lot in multithreaded programs**
 - **speed is really important**
 - » **shouldn't slow things down much in the success case**
 - **checking for errors slows things down (a lot)**
 - » **thus errors aren't checked by default**

Set Up

```
int pthread_mutex_init(pthread_mutex_t *mutexp,  
    pthread_mutexattr_t *attrp)
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutexp)
```

```
int pthread_mutexattr_init(pthread_mutexattr_t *attrp)
```

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attrp)
```

Stupid Mistakes ...

```
pthread_mutex_lock(&m1);  
pthread_mutex_lock(&m1);  
    // really meant to lock m2 ...
```

```
pthread_mutex_lock(&m1);  
    ...  
pthread_mutex_unlock(&m2);  
    // really meant to unlock m1 ...
```

Runtime Error Checking

```
pthread_mutexattr_t err_chk_attr;  
pthread_mutexattr_init(&err_chk_attr);  
pthread_mutexattr_settype(&err_chk_attr,  
    PTHREAD_MUTEX_ERRORCHECK);
```

```
pthread_mutex_t mut1;  
pthread_mutex_init(&mut1, &err_chk_attr);
```

```
pthread_mutex_lock(&mut1);
```

```
if (pthread_mutex_lock(&mut1) == EDEADLK)  
    fprintf(stderr, "error caught at runtime\n");
```

```
if (pthread_mutex_unlock(&mut2) == EPERM)  
    fprintf(stderr, "another error: you didn't lock it!\n");
```


Example

```
void InsertList(val_t v) {  
    list_t *new = (list_t *)malloc(sizeof(list_t));  
    new->val = v;  
    pthread_mutex_lock(&m);  
    new->next = head;  
    head = new;  
    pthread_mutex_unlock(&m);  
}
```

InsertList(val);

thread 1

InsertList(val);

thread 2

Example

```
void InsertList(val_t v) {  
    list_t *new = (list_t *)malloc(sizeof(list_t));  
    new->val = v;  
    pthread_mutex_lock(&m);  
    new->next = head;  
    head = new;  
    pthread_mutex_unlock(&m);  
}
```

InsertList(val);

thread 1

InsertList(val);

thread 2

InsertList(val);
InsertList(mustBeNext);

thread 3

Example

```
void InsertList(val_t v) {  
    list_t *new = (list_t *)malloc(sizeof(list_t));  
    new->val = v;  
    pthread_mutex_lock(&m);  
    new->next = head;  
    head = new;  
    pthread_mutex_unlock(&m);  
}
```

```
InsertList(val);
```

thread 1

```
InsertList(val);
```

thread 2

```
pthread_mutex_lock(&m);  
InsertList(val);  
InsertList(mustBeNext);  
pthread_mutex_unlock(&m);
```

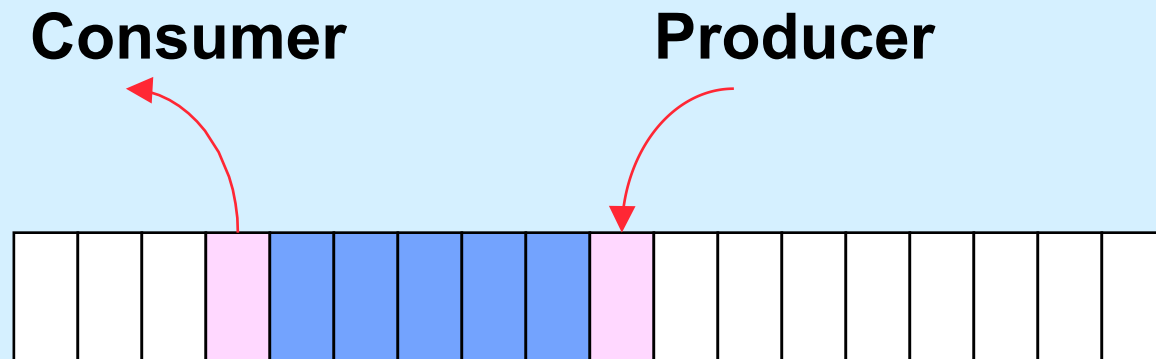
thread 3

Creating Recursive Mutexes

```
pthread_mutexattr_t recursive_attr;  
pthread_mutexattr_init(&recursive_attr);  
pthread_mutexattr_settype(&recursive_attr,  
    PTHREAD_MUTEX_RECURSIVE);
```

```
pthread_mutex_t mut;  
pthread_mutex_init(&mut, &recursive_attr);  
pthread_mutex_lock(&mut);  
if (pthread_mutex_lock(&mut) == EDEADLK)  
    // won't happen -- mutex is recursive  
    ;  
pthread_mutex_unlock(&mut); // decrements lock count  
pthread_mutex_unlock(&mut); // really unlocks it
```

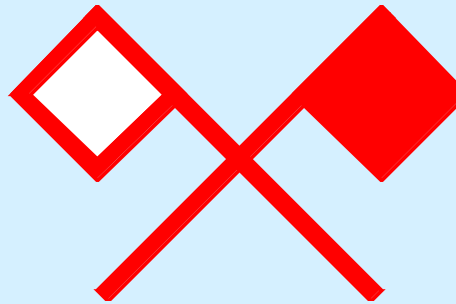
Producer-Consumer Problem



Guarded Commands

```
when (guard) [  
  /*  
    once the guard is true, execute this  
    code atomically  
  */  
  
  ...  
  
]
```

Semaphores



- **P(S) operation:**

```
when (S > 0) [  
    S = S - 1;  
]
```

- **V(S) operation:**

```
[S = S + 1;]
```

Quiz 2

```
semaphore S = 1;  
int count = 0;
```

```
void proc( ) {  
    P(S);  
    count++;  
    ...  
    count--;  
    V(S);  
}
```

The function proc is called concurrently by n threads. What's the maximum value that count will take on?

- a) 1
- b) 2
- c) n
- d) indeterminate

- **P(S) operation:**
 when (S > 0) [
 S = S - 1;
]
- **V(S) operation:**
 [S = S + 1;]

Producer/Consumer with Semaphores

```
Semaphore empty = BSIZE;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;
```

```
void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    if (++nextin >= BSIZE)
        nextin = 0;
    V(occupied);
}
```

```
char Consume( ) {
    char item;
    P(occupied);
    item = buf[nextout];
    if (++nextout >= BSIZE)
        nextout = 0;
    V(empty);
    return item;
}
```

POSIX Semaphores

```
#include <semaphore.h>
```

```
int sem_init(sem_t *semaphore, int pshared, int init);
```

```
int sem_destroy(sem_t *semaphore);
```

```
int sem_wait(sem_t *semaphore);
```

```
    /* P operation */
```

```
int sem_trywait(sem_t *semaphore);
```

```
    /* conditional P operation */
```

```
int sem_post(sem_t *semaphore);
```

```
    /* V operation */
```

Producer-Consumer with POSIX Semaphores

```
sem_init(&empty, 0, BSIZE);  
sem_init(&occupied, 0, 0);  
int nextin = 0;  
int nextout = 0;
```

```
void produce(char item) {  
  
    sem_wait(&empty);  
    buf[nextin] = item;  
    if (++nextin >= BSIZE)  
        nextin = 0;  
    sem_post(&occupied);  
}
```

```
char consume( ) {  
    char item;  
    sem_wait(&occupied);  
    item = buf[nextout];  
    if (++nextout >= BSIZE)  
        nextout = 0;  
    sem_post(&empty);  
    return item;  
}
```