Multithreaded Programming

Compiling Multi-threaded Program

- gcc –o program program.c -pthread

Singly Linked List

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

void add_after(node_t *after, node_t *new) {

   new->next = after->next;
   after->next = new;

}
```

Singly Linked List

```
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

```
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(&global_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);
}
```

Doubly Linked List: Does it 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

```
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: Deadlock

```
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

```
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->prev->mutex);
    pthread_mutex_unlock(&old->mutex);

}
```

Doubly Linked List: Does it 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->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->prev->mutex);
    pthread_mutex_unlock(&old->mutex);

}
```

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);
}
```

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 …

- Example 1

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

- Example 2

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

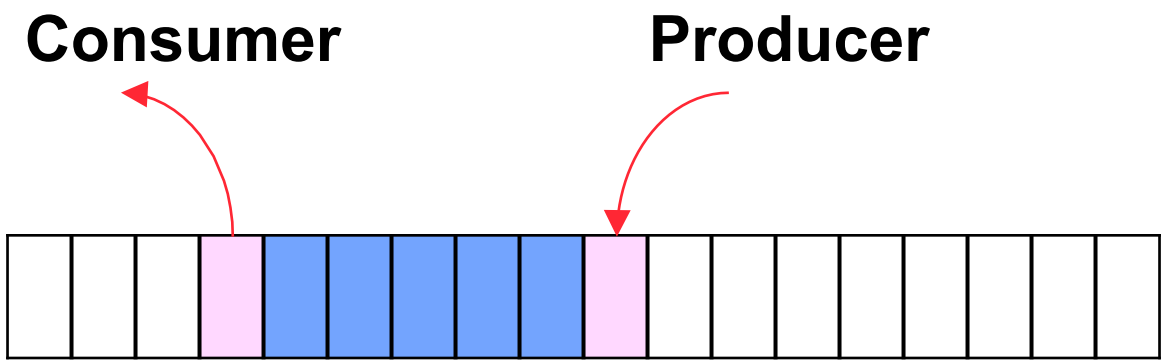
Runtime Error Checking

```
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)  //avoid deadlock by itself
   fprintf(stderr, "error caught at runtime\n");

if (pthread_mutex_unlock(&mut2) == EPERM)  //avoid unlock a mutex locked by others
 fprintf(stderr, "another error: you didn't lock it!\n");
```

Enforcing Mutual Exclusion

- Synchronize the execution of the threads
  - Mutex: a thread attempting to acquire an unavailable lock is blocked until the lock is released
  - Semaphores (Edsger Dijkstra)

Producer-Consumer Problem



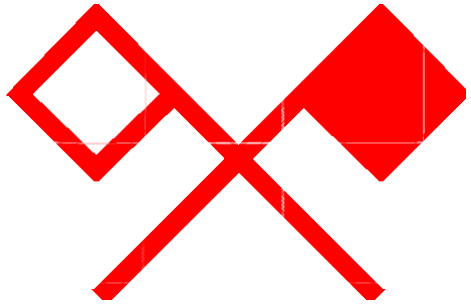
Guarded Commands

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


 ...

]
```

Semaphores

- Semaphore: non-negative global integer synchronization variable.
- Manipulated by P and V operations.
- **P(s)**
  - If *s* is nonzero, then decrement *s* by 1 and return immediately. Test and decrement operations occur atomically (indivisibly)
  - If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - After restarting, the P operation decrements *s* and returns control to the caller.
- ***V(s):***
  - Increment s by 1.
    - Increment operation occurs atomically
  - If there are any threads blocked in a P operation waiting for *s* to become non-zero, then restart exactly one
- **Semaphore invariant: *(s >= 0)***

Semaphores



- **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(charitem) {
    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;
}
```