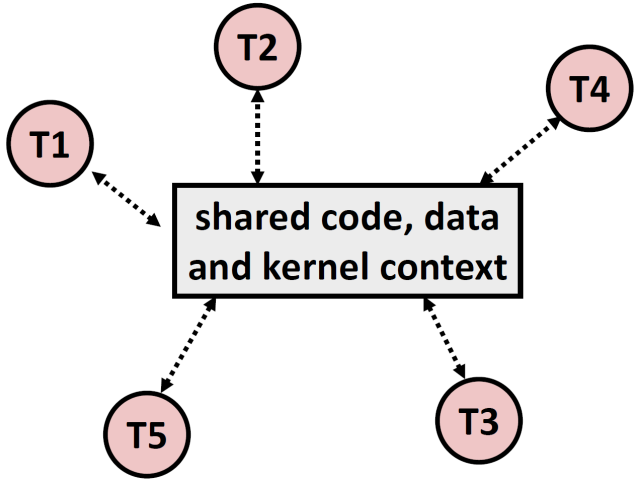
Thread and Multithreaded Programming

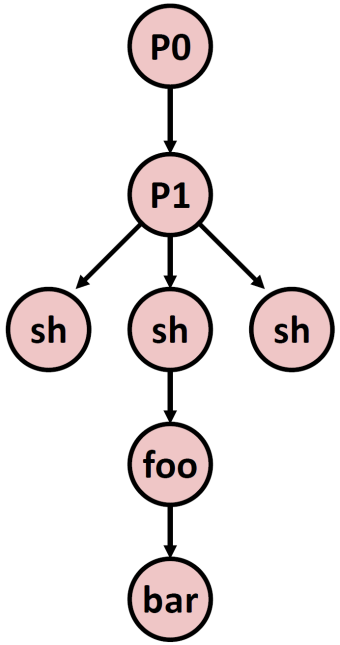
Logical View of Thread

- **Threads associated with a process form a pool of peers**
  - Unlike processes which form a tree hierarchy

Threads in a process



Process hierarchy



Example

pthread_create(&tid1, 0, tproc, (void *)1)

pthread_create(&tid2, 0, tproc, (void *)2)

printf("T0\n");

...

```
void   *tproc(void   *arg)
{
    printf("T%dl\n", (long)arg);
    return  0;
}
```

Detached Threads

- Threads associated with a process: pool of threads
- A thread can kill any of its peers or wait for any of its peers to terminate
- At any point in time, a thread is *joinable* or *detached*
- Not joinable

```
pthread_create(&thread, 0, server, 0);
pthread_detach(thread);
```

- When it terminates, it will be reaped automatically.
  - ex. Detached threads on a server: each thread serves a request, no need to wait for each thread to terminate

Thread

- Thread context:
  - Thread ID
  - Stack
  - Stack pointer
  - Program counter
  - Condition codes
  - General-purpose register values

Data Sharing

- Each thread shares the rest of the process context with other threads:
  - The entire user virtual address space
  - Read-only text(code)
  - Read/write data
  - Heap
  - Same set of open files
  - Any shared library code…..

Stack Size (2MB by default)

```
pthread_t thread;
pthread_attr_t thr_attr;
pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);
...
pthread_create(&thread, &thr_attr, startroutine, arg);
...
```

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
```

- Thread 3:

x= x-1;

```
movl x, %eax
subl $1, %eax
movl %eax, x
```

Mutexes

- A synchronization construct providing mutual exclusion
- Code locking:
  - only one thread is executing a particular piece of code at once
- Data locking
  - only one thread is accessing a particular data structure at once
- Must be initialized

POSIX Threads Mutual Exclusion

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int x;
Pthread_mutex_lock(&m);
x = x+1;
pthread_mutex_unlock(&m);
```

Mutexes

- An important restriction:
- the thread that locked a mutex should be the thread that unlocks it.
- incorrect to unlock a mutex that is not held by the caller

Mutexes

Correct Usage:

……

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

Incorrect:

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

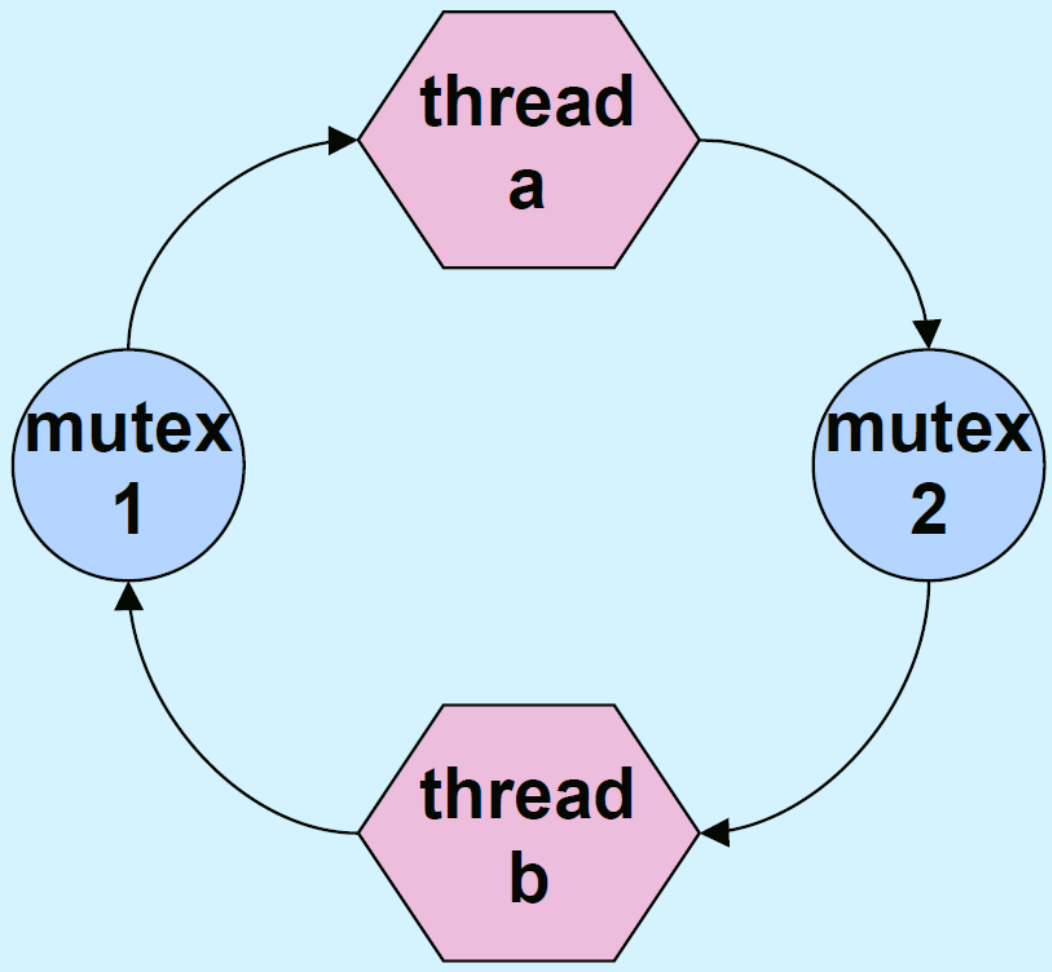
(incorrect usage)

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



- If all threads take locks in the same order, deadlock cannot happen.

Taking Multiple Locks, Safely

- Take locks in the same order

```
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

```
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(&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);
}
```

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);
}
```

A node may not totally removed from the list

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);
}
```

A node may be already unlinked while you are trying to lock it

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 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) {
        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);
}
```