Synchronization Mechanisms

- Mutex Locks
  - Enforce protection and mutual exclusion
- Condition variables
  - Allow atomic checking of conditions
- Semaphores
  - Provide more general forms of synchronization

Mutex Locks

- Mutex Lock: Protects access to a shared resource
  - A thread locks a resource before accessing it
  - Another thread will have to wait for the resource to be unlocked
  - The first thread would unlock the resource after accessing it
- Mutex locks can be used to protect both shared resources and critical sections

Mutex States

- Locked: A single thread holds the mutex
  - Another thread trying to lock mutex will be blocked
  - Queue of blocked threads
- Unlocked: No thread holds the mutex
  - A thread trying to lock mutex will succeed and get control of the mutex
Mutex Operations

- **Lock**: Gain control of the mutex
  - Get lock if free
  - Block if already locked
- **Unlock**: Release the mutex
  - Unblock a waiting thread if any
  - Unblocked thread becomes new owner of mutex
- **Trylock**: Check for availability
  - Lock if available, otherwise don’t block
- All these operations are atomic

Mutex Properties

- Simple and efficient synchronization mechanism
- Typically meant to be held for short durations
- Useful for short critical sections. Examples:
  - Modifying/reading a variable value
  - Modifying pointers in a shared data linked list
- Provide exactly-one-at-a-time mutual exclusion

Mutex Example

```
counter=1;
mutex mut;
Thread A:       Thread B:
lock(mut);     lock(mut);
counter++;     counter--;
unlock(mut);   unlock(mut);
```

- Whichever thread executes lock first gets to execute completely
- The other thread waits for its turn
- Ensures consistent value of counter

POSIX Mutex: `pthread_mutex_t`

- **Initialization**: `pthread_mutex_init`
  - Can also be set to `PTHREAD_MUTEX_INITIALIZER`
  - Should be done exactly once
- **Destruction**: `pthread_mutex_destroy`
- **Locking/Unlocking Operations**:

```c
pthread_mutex_lock(pthread_mutex_t *mutex);
pthread_mutex_unlock(pthread_mutex_t *mutex);
pthread_mutex_trylock(pthread_mutex_t *mutex);
```
POSIX Mutex: Example

```c
int counter=1;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

Thread A:
pthread_mutex_lock(&mut);
counter++;
pthread_mutex_unlock(&mut);
```

Conditional Execution

- Mutex locks:
  - Control access to a shared variable or a code segment
  - Each thread eventually accesses the variable or executes the code segment
  - Typical waiting time is small
  - What if we want to execute a code segment only under certain circumstances?
    - The waiting time could be unbounded

Conditional Execution: Example

<table>
<thead>
<tr>
<th>Condition to wait for: x==y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive approach: Busy Waiting</td>
</tr>
<tr>
<td><code>while (x!=y)</code></td>
</tr>
<tr>
<td><code>/* loop */;</code></td>
</tr>
</tbody>
</table>

Conditional Execution: Example

<table>
<thead>
<tr>
<th>Condition to wait for: x==y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive approach: Busy Waiting</td>
</tr>
<tr>
<td><code>while (x!=y)</code></td>
</tr>
<tr>
<td><code>/* loop */;</code></td>
</tr>
</tbody>
</table>

Problems:
- Wastes CPU cycles
- Might prevent other threads from running, and even changing x and y
- Race conditions
Conditional Execution: Another Approach

- Use a mutex lock
  - Lock a mutex
  - Test the condition \( x = y \)
  - If true, unlock the mutex and continue
  - If false, block and unlock the mutex

Problem:
- In what order should we block and unlock the mutex?
  - Block first: Mutex remains locked, some other thread cannot access \( x \) and \( y \)
  - Unlock first: How to be notified about condition
- Need a simple way of testing and blocking

Condition Variables

- Provide atomic way of testing conditions and blocking if required
- Used in conjunction with a mutex
  - Mutex used to protect access to shared data
  - Condition variable used to signal possible satisfaction of condition, e.g.: \( x = y \)

Condition Variables: Operations

- Uses a mutex lock
- wait:
  - Atomically unlocks mutex and blocks
  - Thread owns mutex when it returns from wait
- signal:
  - Notifies a waiting thread about a condition
- signalAll/broadcast:
  - Notifies all waiting threads about a condition
- These operations are atomic

Condition Variables: Example

```c
cond_var cond;
mutex mut;
lock(mut);
while (x!=y)
  wait(cond, mut);
do_something();
unlock(mut);
```
**Condition Variables: Usage**

- **wait operation:**
  - A thread must hold mutex when calling wait
  - Blocking and releasing of mutex is an atomic operation done inside wait
- **signal operation:**
  - A thread receiving signal may still wait to grab the mutex if not free
  - It must check condition even after being signaled on condition variable

---

**POSIX Condition Variable: pthread_cond_t**

- **Initialization:** `pthread_cond_init`
  - Can also be set to `PTHREAD_COND_INITIALIZER`
  - Should be done exactly once
- **Destruction:** `pthread_cond_destroy`
- **Wait/signal Operations:**
  ```
  pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  pthread_cond_timedwait(..., struct timespec *time);
  pthread_cond_signal(pthread_cond_t *cond);
  pthread_cond_broadcast(pthread_cond_t *cond);
  ```

---

**POSIX Condition Variable: Example**

```c
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
pthread_mutex_t mut;

pthread_mutex_lock(&mut);
while (x!=y)
  pthread_cond_wait(&cond, &mut);
do_something();
pthread_mutex_unlock(&mut);

pthread_mutex_lock(&mut);
  x=y;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mut);
```

---

**Semaphores**

- **Generalization** of mutex locks
  - Mutex allows access to exactly one thread
  - Semaphore allows controlled access to multiple threads
- **Semaphore has a non-negative counter**
  - The value of the counter determines the access control and synchronization
- **Can be thought of as a C struct**
  ```
  struct {
    int counter;
    thread_list list;
  }
  ```
Semaphore Operations

- **Wait**
  - Also called P (proberen), down, lock
  - Either decrements counter or blocks if counter is 0

- **Signal**
  - Also called V (verhogen), up, unlock
  - Either increments counter or wakes up a blocked thread

```c
if (S->counter > 0)
    S->counter--;
else
    add to S->list and block;
```

```c
if (S->list != NULL)
    remove thread from S->list
else
    S->counter++;
```

Semaphore Properties

- wait and signal are atomic operations
- OS implements these as critical sections
- Can be used for mutual exclusion or signaling
- Behavior depends on initial value of counter

Controlling access to a Critical Section

```c
wait(S)  
Critical section  
signal(S)
```

Each thread must call wait and signal in correct order
How many threads can run in critical section if:
- Initial value of S->counter is 1
- Initial value of S->counter is 10
- Initial value of S->counter is 0

Synchronizing Execution Order

Process A:
- Statement 1;
- signal(S);

Process B:
- wait(S);
- Statement 2;

- In which order would the threads run if:
  - Initial value of S->counter is 0
  - Initial value of S->counter is 1
- Can we do this kind of synchronization with mutex locks?
POSIX Semaphore: sem_t

- Initialization:
  ```c
  sem_init(sem_t *sem, int pshared, unsigned value);
  ```
  - `sem`: Semaphore variable
  - `pshared`:
    - 0 => sharing within process
    - Non-zero => shared across processes
  - `value`: initial value (must be non-negative)

- Destruction: `sem_destroy(sem_t *sem);`

POSIX Semaphore: Operations

- Wait/signal:
  ```c
  sem_wait(sem_t *sem);
  sem_post(sem_t *sem);
  sem_trywait(sem_t *sem);
  ```

POSIX Semaphore: Example

```c
sem_t sem;
/* Initialize process-local semaphore with initial value=1 */
sem_init(&sem, 0, 1);
sem_wait(&sem);
/* Critical section */
sem_post(&sem);
```