Goals:
- Describe multi-threaded execution in the context of control systems
- Introduce *Pthreads*, a standard library for multi-threaded programming

Reading:
- “POSIX Threads Programming”, Lawrence Livermore National Laboratory. 2006 (online tutorial)
Traditional Control Systems Implementation (Sparrow)

Simplest case: interrupt driven loop
- Use HW/SW interrupts to run control routine at an accurate and fixed rate
- “Servo loop” overrides normal program operation
- Need to be careful about interaction of variables in servo loop with main pgm

Variations:
- Time-triggered protocols - scheduling of events to allow multiple “servos”

Sample program
- Discrete time implementation
  \[ z_{k+1} = A_c z_k + B_c e \]
  \[ y_k = C_c z_k + D_c e \]
- Uses quasi-sparrow implementation

```
load_controller(file)
servo_setup(loop, rate, flags);
servo_enable();

loop()
{
    y = read_measurement();
    r = read_reference();
    xnew = Ac * x + Bc * (r - y);
    u = Cc * x + Dc * (r - y);
    write_control(u);

    x = xnew;
}
```
Multi-Threaded Programming

**Basic Idea**
- Separate code into independent segments (“threads”)
- Switch between threads, allowing each to run “simultaneously”
- Threads share memory and devices; allows rapid sharing of information

**Advantages**
- Avoid manual coding to eliminate pauses due to hardware response
- Multiple control loops become separate threads; OS insures execution
- Allows messages (or signals) to be received in middle of long computation

**Threads vs Processes**
- Processes have separate memory space and device handles
- Requires interprocess communication to share data

**Issues**
- Race conditions
- Dead locks (“deadly embrace”)
- Asynchronous operations
Definition

“A race condition is a flaw in a system or process where the output exhibits unexpected critical dependence on the relative timing of events.” (wikipedia)

Application to threads

• Execution of threads is controlled by the operating system (OS)
• It is possible for threads to be pre-empted and another thread to run ⇒ can’t assume anything about order
• While easy to understand, race conditions can be hard to locate and debug

Example

• Thread 1: compute sqrt of number
• Thread 2: update number on condition

Code:

```c
thread1() {
    if (x < 0)
        y = 0;
    else
        y = sqrt(x);
}
thread2() {
    if (event) x = x - 1;
} 
```
Mutual Exclusion (mutex)

Solution: exclude overlapping access
- Semaphores introduced by Djikstra in 1960s to handle this problem
- Key idea: protect “critical sections” of code by setting a “mutex”
- Mutex_lock: wait for mutex to be unblocked (if it isn’t already), then set
  - While a mutex_lock is being blocked, the OS can execute code
- Mutex_unlock: unset the mutex

Atomicity
- Operating systems need to insure that mutexes are “atomic” operations - no instructions executed while checking and setting the flag
- If this doesn’t happen, you can get a race condition in setting the flag (which is what we are trying to avoid…)

```c
thread1() {
    mutex_lock(xmtx);
    if (x < 0)
        y = 0;
    else
        y = sqrt(x);
    mutex_release(xmtx);
}

thread2() {
    mutex_lock(xmtx);
    if (event) x = x - 1;
    mutex_release(xmtx);
}  
```
Issue #2: Deadlocks

Possible execution

- Thread 1 executes up to line 3
  - Locks xmtx mutex
- Switch to thread 2
- Thread 2 executes through line 12
  - Locks ymtx mutex
- Compute() blocks on xmtx mutex
- Switch to thread 1
- Thread 1 blocks on ymtx mutex
- (no further execution)

Remarks

- Easy to fix, but sometimes hard to spot (especially when using subroutines)

Solution: lots of debugging

- Formal tools exist, but generally can’t operate at programming code level

```c
1 thread1() {
2   mutex_lock(xmtx);
3   if (x < 0) x = 0;
4   mutex_lock(ymtx);
5   y = sqrt(-x);
6   mutex_lock(ymtx)
7   mutex_release(xmtx);
8 }
9
10 thread2() {
11   mutex_lock(ymtx);
12   if (y == 0) compute();
13   mutex_unlock(ymtx);
14 }
15
16 compute() {
17   mutex_lock(xmtx);
18   if (event) x = x - 1;
19   mutex_release(xmtx);
20 }
```
Thread Usage

When to use threads
- Main usage is when the program has to wait on a process or resource
- Eliminate threads if they aren’t needed (eg, tight interlocking with no waits)

Avoiding deadlocks
- Never put a mutex around a call that might itself block (I/O call, mutex, etc)
- If you have to use nested mutex’s, make sure they are in the same order whenever they are invoked

Performance improvements
- Try to keep critical sections as small as possible (avoids excessive waiting)
- Combine accesses to same variables in nearby sections
- Use buffers to minimize lock times

Conditional variables
- Allows a thread to sleep until a certain condition is met
- Used in conjunction with a mutex

```
1 thread1() {
2   mutex_lock(xmtx);
3   while (!condition)
4     cond_wait(&cond, &mutex);
5   do_something();
6   mutex_unlock(xmtx);
7 }
8
9 thread2() {
10  mutex_lock(xmtx);
11  // make condition TRUE
12  if (cond)
13     cond_signal(&cond);
14  mutex_unlock(xmtx);
15 }
```

Releases mutex on entry and relocks on return
Issue #3: Asynchronous Execution

Execution is non-deterministic
- Operating system determines when to execute individual threads
- Different operating systems will give different sequences of operations
- Avoid tuning scheduling rules to let OS optimize (eg, multi-processor core)

Use mutexes and conditions if needed
- Can insure partial synchronization by using mutexes and conditions, but
- Avoid overly constraining threads; can get worse performance than just doing things sequentially

Reasoning about concurrent code
- It is still possible to prove things about multi-threaded execution
- Example: Lyapunov like functions
  - Let V be a positive function whose minimum corresponds to desired state
  - Show that each portion of code does not increase V’s value
  - Show that some portions of code decrease value of V
  - Conclude that V will approach minimum value
- Formal methods: temporal logic, unity
POSIX Threads (Pthreads)

Thread creation
• pthread_create call a function as a new thread of execution
• pthread_exit terminate the current thread
• pthread_join wait for a specific thread to exit

Mutexes
• pthread_mutex_init initialize a mutex
• pthread_mutex_lock lock a mutex (blocks until mutex is available)
• pthread_mutex_unlock unlock a mutex (and unblock first blocked threads)
• pthread_mutex_destroy free up resources associated with a mutex

Conditional variables
• pthread_cond_init initialize a condition
• pthread_cond_wait wait until condition is satisfied (paired with a mutex)
• pthread_cond_signal signal that a condition is now satisfied
• pthread_cond_destroy free up resources associate with a mutex

Read/write locks
• Variation on mutexes that allow multiple unblocking reads
Example: Threaded Control Loop

```
1 pthread_create(..., sensor, ...);
2 pthread_create(..., actuator, ...);
3 pthread_create(..., control, ...);
4 display();
5
6 sensor() {
7   // initialization
8   while (1) {
9     pthread_mutex_lock(smtx);
10    y = read_measurement();
11    r = read_reference();
12     pthread_mutex_unlock(smtx);
13     usleep(S_WAIT_USEC);
14   }
15 }
16
17 actuation() ...
18
19 control() {
20   // initialization
21   while (1) {
22     pthread_mutex_lock(smtx);
23     err = r - y;
24     pthread_mutex_unlock(smtx);
25     xnew = Ac * x + Bc * err;
26     pthread_mutex_lock(amtx);
27     u = Cc * x + Dc * err;
28     pthread_mutex_unlock(amtx);
29     sleep(C_WAIT_USEC)
30   }
31 }
32 }
33
```

Notes

- Process inputs/outputs asynchronously
- HW: is this OK? Optimal?
Thread Scheduling

Thread scheduling policies

- **FIFO** - threads are called in first in, first out order within each priority level
  - Thread continues to run until a higher priority thread is runnable
  - Threads at same priority must block in order for other threads to run

- **Round-robin** - each thread is called in sequence, within priority level
  - Thread runs for fixed period of time before it is pre-empted

- **Other** - implementation specific
  - Operating system defines how threads are scheduled
  - This is the default (and undefined!)

Homework

- Write a simple multi-threaded program using pthreads that reads numbers from an input stream (terminal), averages all numbers that have been read, and prints out the average once a second [use three threads]

Project Ideas

- Expand sparrow to allow multi-threaded servo and channel (I/O) execution

- Analyze how to best use mutex’s for minimizing control latency when reading inputs via the network and writing outputs via (slow) serial ports

- Convert a controller from continuous to discrete for a multi-threaded control system using FIFO or round-robin scheduling
Thread Usage in Alice

<table>
<thead>
<tr>
<th>Module</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>adrive (actuation)</td>
<td>19</td>
</tr>
<tr>
<td>trajFollower</td>
<td>10</td>
</tr>
<tr>
<td>astate (state estimator)</td>
<td>10</td>
</tr>
<tr>
<td>plannerModule</td>
<td>4</td>
</tr>
<tr>
<td>fusionMapper</td>
<td>16</td>
</tr>
</tbody>
</table>

* doesn’t count heartbeat and logging threads

<table>
<thead>
<tr>
<th>Module</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>ladarFeeder (5)</td>
<td>8</td>
</tr>
<tr>
<td>stereoFeeder (2)</td>
<td>7</td>
</tr>
<tr>
<td>road (road follower)</td>
<td>5</td>
</tr>
<tr>
<td>superCon</td>
<td>3</td>
</tr>
<tr>
<td>DBS</td>
<td>3</td>
</tr>
</tbody>
</table>
Example: State Client

- Asynchronously reads actuator state from adrive via thread
- Passes data back to calling module (eg, trajFollower)

```cpp
void CStateClient::getActuatorStateThread() {
    int actuatorstatesocket =
        m_skynet.listen(SNactuatorstate, ALLMODULES);

    while(m_bRunThreads) {
        if(m_skynet.get_msg(actuatorstatesocket,
            &m_rcvdActuatorstate, sizeof(m_rcvdActuatorstate), 0,
            &pActuatorstateMutex) != sizeof(m_rcvdActuatorstate))
            skynet_error();
        DGCSetConditionTrue(condNewActuatorState);
    }
}

void CStateClient::UpdateActuatorState() {
    DGClockMutex(&m_actuatorstateMutex);
    memcpy(&m_actuatorState, &m_rcvdActuatorstate, sizeof(…));
    DGCunlockMutex(&m_actuatorstateMutex);
}

void CStateClient::WaitForNewActuatorState() {
    DGCWaitForConditionTrue(condNewActuatorState);
    UpdateActuatorState();
    condNewActuatorState.bCond = false;
}
```
Summary: Multi-Threaded Control Systems

**Advantages**
- Allows more efficient processor usage when waiting for hardware
- Convenient method for handling independent, parallelizable tasks

**Cautions**
- Race conditions
- Deadlocking
- Asynchronous execution
- Debugging

**Open Issues for Control Theory**
- How should we implement feedback controllers in a multi-threaded, asynchronous environment (and does it matter)?
- How do we verify that multi-threaded programs satisfy the control design specification and designer’s intent (or is this automatic)?
- How do we implement multi-rate controllers (of the sort that Giotto is designed to handle) in a threaded process?