NCS Lecture 3: Embedded Systems Programming

Richard M. Murray
17 March 2008

Goals:
• Describe how event ordering works in distributed systems
• Discuss choices in NCS message delivery
• Introduce Spread, a group communications toolkit
• 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)

Communication Management: Spread

<table>
<thead>
<tr>
<th>Message type</th>
<th>Bytes/Freq</th>
<th>Recv</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicle State</td>
<td>~250 B @ 40 Hz</td>
<td>15</td>
<td>Pos, vel, acc; highest update rate</td>
</tr>
<tr>
<td>Actuator State</td>
<td>~220 B @ 30 Hz</td>
<td>3?</td>
<td>Actuators + OBD II information</td>
</tr>
<tr>
<td>Elevation Map</td>
<td>4 MB @ 10 Hz</td>
<td>0</td>
<td>Not transmitted</td>
</tr>
<tr>
<td>Cost Map</td>
<td>4 MB @ 10 Hz</td>
<td>3</td>
<td>Deltas transmitted</td>
</tr>
<tr>
<td>Trajectory</td>
<td>??? @ 5 Hz</td>
<td>2</td>
<td>Variable size (I think)</td>
</tr>
<tr>
<td>Cameras</td>
<td>640x480 @ 30 Hz</td>
<td>5</td>
<td>Firewire (~20 MB/s per camera)</td>
</tr>
</tbody>
</table>
Causality in Distributed Communications (Lamport, ’78)

Partial ordering: \( a \rightarrow b \)
- If \( a \) and \( b \) are events in the same process, then \( a \rightarrow b \)
- If \( a \) is the sending of a message by one process and \( b \) is the receipt of the same message by another process, then \( a \rightarrow b \)
- If \( a \rightarrow b \) and \( b \rightarrow c \) then \( a \rightarrow c \)
- \( a \rightarrow b \) means “\( a \) can causally effect \( b \)”

Logical Clocks
- Let \( C_i(a) \) be a clock for process \( P_i \) that assigns a number to an event
- Define \( C(b) = C_j(b) \) if \( b \) is an event in process \( P_j \)
- Clock condition: for any two events \( a, b \): if \( a \rightarrow b \) then \( C(a) < C(b) \)

Remarks
- Events are partially ordered: can compare some events but not all events
- Example: \( p_1 \rightarrow q_3 \) but \( p_3 \) and \( q_3 \) are not related
- Clocks are not unique (can choose any set of integers with appropriate relations)

Implementing a Clock

Conditions for a clock
- C1: If \( a, b \) are events in process \( P_i \) and \( a \) comes before \( b \), then \( C_i(a) < C_i(b) \)
- C2: If event \( a \) is the sending of a message by process \( P_i \) and event \( b \) is the receipt of that message by process \( P_j \), then \( C_i(a) < C_j(b) \)

Space-Time Diagram
- Add ticks for every count in each process
- Draw “tick lines” between equally numbered ticks
- C1 \( \Rightarrow \) tick line between two events
- C2 \( \Rightarrow \) every msg must cross tick line

Remarks
- Events can shift around between tick lines without changing logical clocks \( \Rightarrow \) logical time is different than physical time
Constructing Clocks

Implementation rule

• IR1: Each process $P_i$ increments $C_i$ between any two successive events
• IR2:
  • (a) If event $a$ is the sending a message $m$ by process $P_i$, then the message $m$ contains a timestamp $T_m = C_i(a)$
  • (b) Upon receiving a message $m$, process $P_j$ sets $C_j$ greater than or equal to its present value and greater than $T_m$

Remarks:

• Gives an easy algorithm for constructing a clock
• Note that $C(a) < C(b)$ does not imply $a \rightarrow b$. Still only a partial order (can only compare certain elements)

Total order

• Order events according to logical clocks
• Break ties using process number
• Allows any two events to be compared
• Total ordering is not unique (depends on choice of clocks)

Example: Resource allocation

Problem description

• Fixed processes $P_i$ sharing resource $R$
• Once a process grabs a resource, it must release it before it is use again
• Requests granted in order they were requested
• Every request is eventually granted
• Solve in distributed way; processes agree on who goes next
• Problem is non-trivial, even with central scheduling (see Lamport paper)

Algorithm

1. $P_i$ sends message $T_m; P_i; request$ to every other process and puts message on its queue
2. $P_j$ queues all requests and sends timestamped acknowledgement to sender
3. Process $P_i$ uses resource when
   1. $T_m; P_i; request$ is ordered before any other request in queue (according to total order)
   2. $P_j$ has been received ack from everyone with timestamp $> T_m$
3. $P_i$ removes $T_m; P_i; request$ message from queue and sends $T_m; P_i; release$ message to everyone
   • When $P_j$ receives a $T_m; P_i; release$ message, it removes message from its queue
Group Messaging Systems

Group
- Collections of processes that can send messages back and forth to everyone
- Messaging system has to keep track of people joining and leaving groups
- Goal: deliver packets reliably and causally

Ex: Alice NCS group message types
- Modules receive certain message types

Issues
- Need to track membership over time
- Need to provide different levels of reliability (at the group level)
- Need to provide different levels of ordering (or causality)
- Also need to keep track of the fact that time may be different on different computers (no global clock)

Message Ordering (“Virtual Synchrony”)

Ordering
- None - No ordering guarantee.
- FIFO by Sender - All messages sent by this sender are delivered in FIFO order.
- Causal - All messages sent by all senders are delivered in Lamport causal order.
- Total Order - All messages sent by all senders are delivered in the exact same order to all recipients

Remarks
- Imposing causality increases message overhead; need to make sure that everyone has the message
- Things get interesting with multiple groups - everyone in same collection of groups should receive all messages in same order
- HW: figure out an example where causal and total order are different
Message Reliability ("Extended Virtual Synchrony")

**Reliability**
- **Unreliable** - Message may be dropped or lost and will not be recovered.
- **Reliable** - Message will be reliably delivered to all recipients who are in group to which message was sent.
- **Safe** - The message will ONLY be delivered to a recipient if everyone currently in the group definitely has the message.

**Remarks**
- Key issue is keeping track of reliability in groups. Reliable messages should be received by everyone (eventually).
- Requires agreement algorithm across computers (who has what).
- HW: find an example where reliable messages are not safe.

---

**Spread Toolkit (Stanton ‘02)**

**Spread Functions**
- `SP_connect`: establish a connection with the spread daemon
- `SP_disconnect`: terminate connection
- `SP_join(mbox. group)`: join a group
- `SP_leave(mbox. group)`: leave a group
- `SP_multicast(…, group, message, type)`: send a message to everyone in group of given type
- `SP_receive`: receive a message

**Message types**
- Unreliable - no order, unreliable
- Reliable - no order, reliable
- FIFO - FIFO by sender, reliable
- Causal - Causal (Lamport), reliable
- Agreed - Totally ordered, reliable
- Safe - Totally ordered, safe

Note: each message has a type; these can be mixed within groups.
Features of Spread

- Number and location of servers are configurable
- Retransmits are optimized in multi-hop environments
- Guarantees are provided at servers; assumes that inter-process communications on a single computer is reliable
- Data is combined in packets when possible to increase efficiency
  - Gives a correlated channel model when data is lost
- No hardwired addresses (except for servers)

Project ideas

- How can we model a spread-based communications network from the point of view of estimation and control?
- Is it better to have one server or multiple servers? What are the latency tradeoffs?
  - Alice originally used one server per computer
  - Eventually moved to a single server (not sure why)

How Spread Works (Amir and Stanton, ‘98)

Spread daemon
- Implements group communications protocols
- Uses UDP to talk between spread hosts
- Two protocols: hop and ring

Applications
- Think client library
- Uses TCP to talk to server

Hop protocol
- UDP-based protocol between sites
  - Sites connected by slower links
  - Provide low latency communications
  - Packet loss handled on hop-by-hop basis (instead of end to end)

Ring Protocol
- Used for communications between multiple servers at the same site
  - Assumes dedicated (switched) links
  - Token ring based protocol: pass control from one server to the next
Example: Resource allocation

Problem description
- Fixed processes $P_i$ sharing resource $R$
- Once a process grabs a resource, it must release it before it is used again
- Requests granted in order they were requested
- Every request is eventually granted
- Solve in distributed way; processes agree on who goes next
- Problem is non-trivial, even with central scheduling (see Lamport paper)

Solution using Spread
- Assume totally ordered, reliable messages ("agreed" message type)
- All processes and resource in single spread group

Algorithm
1. $P_i$ sends multicast message to group requesting resource
2. $P_j$ queues all requests and sends ack
3. Process $P_i$ uses resource when
   - $P_i$ request is at top of queue
   - Ack has been received from everyone
4. $P_i$ sends release message when done
   - $P_j$ dequeues release when message received

2. Note: spread provides single order

Process Management: Pthreads

Example: vehicle actuation (adrive)
- Interface to 7 different actuation systems (steer, gas, brake, etc)
- Each actuator involves different rates, delays, comms blocking
- Asynchronous receipt of commands and state + xmit actuator state
- Hard to synchronize individual actuators with main control loop

Approach: multi-threaded programming
- Break program up into independent execution threads
- Operating system can switch threads on blocking => always executing
- Challenges: asynchronous operation, simultaneous access to variables
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;
}

C(s) \rightarrow \text{Tustin} \rightarrow C(z)

Multi-Threaded Programming

Single threaded execution
- (wait)
- (wait)
- (wait)
- read meas
- ctrl
- write ctrl
- computation

Multi-threaded execution
- suspended execution

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
Issue #1: Race Conditions

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:
```
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 Dijkstra 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…)

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

thread2() {
  mutex_lock(xmtx);
  if (event) x = x - 1;
  mutex_unlock(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

```
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_unlock(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_unlock(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, xmtx);
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 }
```
**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 (e.g., 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

```c
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 control() {
18   // initialization
19   while (1) {
20      pthread_mutex_lock(smtx);
21      err = r - y;
22      pthread_mutex_unlock(smtx);
23      xnew = Ac * x + Bc * err;
24      pthread_mutex_lock(amtx);
25      u = Cc * x + Dc * err;
26      pthread_mutex_unlock(amtx);
27      usleep(C_WAIT_USEC)
28   }  
29 }
30
```

**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

### Example: State Client

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

```c
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;
}
```

- Thread to read msgs
- Infinite loop
- Read msg (blocks until available)
- Unblock anyone waiting
- Copy state into buffer
- Use mutex to insure completeness
- Block until new state msg arrives
Verifying Multi-Threaded Programs

SPIN (Holzmann)
- Model system using PROMELA (Process Meta Language)
  - Asynchronous processes
  - Buffered and unbuffered message channels
  - Synchronizing statements
  - Structured data
- Simulation: Perform random or iterative simulations of the modeled system's execution
- Verification: Generate a C program that performs a fast exhaustive verification of the system state space
- Check for deadlocks, livelocks, unspecified receptions, and unexecutable code, correctness of system invariants, non-progress execution cycles
- Also support the verification of linear time temporal constraints

TLA/TLC (Lamport et al)
- Temporal Logic of Actions (TLA): Leslie Lamport, 1980's
- Behavior (a sequence of states) is described by an initial predicate and an action
  \[ \text{Spec} \equiv \text{Init} \land \Box \text{Action} \]
- Specify a system by specifying a set of possible behaviors
- Theorem: A temporal formula satisfied by every behavior
  \[ \text{Theorem} \equiv \text{Spec} \Rightarrow \Box \text{Properties} \]

TLA+
- Can be used to write a precise, formal description of almost any sort of discrete system
- Especially well suited to describing asynchronous systems
- Tools: Syntactic Analyzer, TLC model checker

Summary: Embedded Systems Programming

Advantages
- Increased modularity
- Simplified programming*

Cautions
- Asynchronous execution
- Race conditions
- Deadlocking
- Debugging

Open Issues for Control Theory
- How do we best implement controllers in this setting?
- How do we verify that programs satisfy the specifications and design intent
- How do we implement multi-rate controllers using threaded process and distributed computing?