CSE 5372
Specification and Design of Real-Time Systems

Lecture 5: Real Time Messages

Outline

- Quiz (10 min)
- Groups Presentations (45)
- Lecture: RT Messages (45)
- Lab exercise - Messages and Pipes (60)
- Exam 1: Discussion
The message-driven approach is used both for synchronization and communication.

- It uses single construct *message*

- Theoretical issues:
  - synchronization model
  - process naming method
  - message structure
Process Synchronization Models

- **Asynchronous** - the sender proceeds immediately regardless whether the message was received or not (no-wait)

- **Synchronous** - the sender proceeds only if the message has been received

- **Remote Invocation** - the sender proceeds only when a reply has been returned from receiver

Asynchronous Message Passing

- Task T1 sends the *message* and continues
- Task T2 waits for the *message*

```
T1 (sender)
...
send(message)
...
...
```

```
T2 (receiver)
...
...
wait(message)
...
```
Synchronous Message Passing

- Synchronous message passing is implemented using asynchronous primitives (the sender waits for confirmation/acknowledgement)
  - send(M)/wait(A) => synchr send(M)
  - wait(M)/send(A) => synchr_wait(M)

```
T1 (sender)
  ...
  send(message)
  wait(acknowledgment)
  ...

T2 (receiver)
  ...
  ...
  wait(message)
  send(acknowledgment)
  ...
```

Remote Invocation

- Implemented with double “hand-shaking”
- T2, after receiving the message, constructs and sends reply (synchronously)
- T1 waits for the reply before proceeding

```
T1 (sender)
  ...
  send(message)
  Synchr_wait(reply)
  ...

T2 (receiver)
  ...
  ...
  wait(message)
  [construct reply]
  Synchr_send(reply)
  ...
```
Issues and Message Structure

- Asynchronous model is most flexible as it can implement the other two
  - Potential problems:
    - large buffer size required for the messages not read
    - “out of date” syndrome
  - Asynchronous communication can be implemented by a synchronous model using intermediary buffer
  - Message Structure:
    - Any data object defined by language should be allowed to be sent as the message
    - Message passing may be difficult
      - when sender and receiver have data objects represented in different formats
      - when using pointers to the data
    - Unit of messages are bytes

Process Naming

- Direction:
  - Direct naming scheme - sender explicitly names the receiver:
    - `send <msg> to <process_name>`
  - Indirect naming scheme - sender names intermediate entity (channel, mailbox, pipe, queue):
    - `send <msg> to <mailbox>`

- Symmetry:
  - Symmetric naming scheme - sender and receiver name each other:
    - `send <msg> to <process/mailbox>`
    - `wait <msg> from <process/mailbox>`
  - Asymmetric naming scheme - receiver waits for message regardless what process sends it:
    - `wait <msg>`
To facilitate inter-task communication, kernels provide a message queue object and message queue management service.

A message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronize.

It decouples tasks’ operation with respect to sending and receiving messages.
Messages can be sent by content or by pointer

Normally message is copied twice (memory usage)

**Message Queue storage**

- **System Pools**
  - Message of all queues have common memory area
  - Usually saves on memory use
  - Message queue with large message can use most of the pool and leave little memory for others

- **Private Buffers**
  - Requires enough memory for full capacity of every message queue that will be created
  - Uses up more memory
  - Better reliability than memory pool
Message Queue Operations

- Creating and deleting
  - Global objects
  - The queue to be used by a group of tasks or ISRs are decided in the design
  - Length of the queue, max size of the message, and wait order is assigned by the developer
  - Messages in the queue are lost when queue is deleted

Message Queue Operations: Sending

- Kernel typically fills a message queue in FIFO or priority order
- With RTOS options, urgent messages will go straight to the head of the queue
- Many implementation allows ISRs to send message to queue. ISRs cannot block, if message queue is full, error might be returned
- Messages are sent to message queue in the following ways
  - Not block (ISR and task)
  - Block with a timeout (task only)
  - Block forever (task only)
Message Queue Operations: Receiving

- Messages are received from message queue in the following ways
  - Not block (ISR and task)
  - Block with timeout (task only)
  - Block forever (task only)

- Messages could be read from the queue in the following ways
  - Destructive (message gets removed after read).
  - Non-destructive (message does not get removed after read)

Typical Message Queue Use

- Non-interlocked, one-way data communication
  - Simple, activities of sending task and receiving task is not synchronized, no acknowledgement required from receiving task, ISR use this type (should not block)

- Interlocked, one-way data communication
  - Sending task requires an acknowledgement from the receiving task
  - Reliable mechanism for task communication
Typical Message Queue Use

- Interlocked, two-way data communication
  - Full duplex or tightly coupled communication
  - Bi-directional communication (client-server)
  - Two message queues are required

- Broadcast communication
  - Send a copy of the same message to multiple tasks
  - One-to-many-tasks

VxWorks Message Queue (1)

- VxWorks message queues are the primary inter-task communication mechanism with a single CPU
- VxWorks message queues enable tasks to exchange information in a flexible, secure, formal, and efficient manner.
- Message queues allow a variable number of messages, each of variable length, to be queued
- RTOS provides routines for message queue control (defined in msgQLib.h):
  - (http://www.vxdev.com/docs/vx55man/vxworks/ref/msgQLib.html)
    - msgQCreate()
    - msgQDelete()
    - msgQSend()
    - msgQReceive()

- Options used include:
  - MSG_Q_FIFO
  - MSG_Q_PRIORITY
  - NO_WAIT
  - WAIT_FOREVER
  - MSG_PRI_URGENT
  - MSG_PRI_NORMAL
  - etc.
VxWorks Message Queue (2)

- Message is placed in the queue according to the priority

VxWorks Message Queue (3)

- Full-Duplex Communication
**VxWorks Message Queue (4)**

- Client-Server

![Diagram of VxWorks Message Queue]

**Example: experiment with VxWorks MQ In Group**

- create message queue (up to ten, length 100 bytes, priority):
  \[
  mQ = \text{msgQCreate}(10, 100, 1)
  \]

- create a string:
  \[
  str = \text{“my message\n”}
  \]

- write to the queue (NO_WAIT, MSG_PRI_NORMAL):
  \[
  \text{msgQSend}(mQ, str, strlen(str)+1, 0, 0)
  \]

- create an empty buffer:
  \[
  buffer = \text{calloc}(20, 1)
  \]

- read it back from the queue (No_WAIT):
  \[
  \text{msgQReceive}(mQ, buffer, 20, 0)
  \]

- check the results:
  \[
  \text{printf}(buffer)
  \]
• Virtual I/O device managed in pipeDev
• Built on top of vxWorks message queues
• Standard I/O system interface (read/write)
• Similar to named pipes in UNIX (UNIX host)

Creating a Pipe

• STATUS pipeDevCreate (name, nMessages, nBytes)
  • name Name of pipe device. By convention use “pipe/yourName”
  • nMessages max number if messages in pipe
  • nBytes max size of bytes in each message

• Returns OK if successful, ERROR otherwise
Example Pipe Creation

- pipeDevCreate ("/pipe/myPipe", 10, 100)
  Value = 0 = 0x0

- devs

<table>
<thead>
<tr>
<th>drv</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>/null</td>
</tr>
<tr>
<td>1</td>
<td>/tyCo/0</td>
</tr>
<tr>
<td>1</td>
<td>/tyCo/1</td>
</tr>
<tr>
<td>4</td>
<td>columbia</td>
</tr>
<tr>
<td>2</td>
<td>/pipe/myPipe</td>
</tr>
</tbody>
</table>

Value = 0 = 0x0

- devs() command displays available device list
- drv column indicates the number of driver managing the device

Reading/Writing to a Pipe

- To access an existing pipe, first open it with open()
- read(): pends if pipe is empty
- write(): pends if pipe is full
- close()

fd = open ("pipe/myPipe", O_RDWR, 0);
write (fd, msg, len)
read (fd, msg, len)
close (fd)
Queues vs. Pipes

- Message Queues Adv:
  - Timeout capable
  - Message prioritization
  - Faster
  - show()

- Pipes Adv:
  - Use standard I/O interface (open(), close(), read(), write())
  - Can perform redirection via ioTaskStdSet()
  - Allows tasks to wait for data on a combination if several pipes, sockets, or serial devices

- Both maybe written to/from an ISR

Queues and Pipes in-class lab