Communication and Synchronization of Concurrent Tasks (MF2008)

Lei FENG
2008.11.17

Embedded Control Systems Research Group
KTH (Royal Institute of Technology)
Stockholm, Sweden
Outline

• Brief review (independent tasks)
• Dependency of tasks
• Pitfalls of resource sharing
• Solutions to safe task interactions
• Implementation with Rubus RTOS
Brief Review

- Concurrent programming (3 types)
- Threads/tasks vs. programs
- Reentrant code (share CPU only)

```c
float ctrl(float p, ref, cur)
{
    ...
    return u;
}
```
Review: Thread States

- Ready
- Running
- Blocked/Waiting

- creation
- dispatching
- pre-emption
- termination
- synchronization
- scheduling
Review: Real-Time Scheduling

- Real-time systems, hard and soft
- Various timing parameters
- Real-time scheduling methods: FP, RM, EDF
- Sufficient (not necessary) condition of RM: \( \frac{C_1}{T_1} + ... + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \)
- Sufficient and necessary condition of EDF: \( \frac{C_1}{T_1} + ... + \frac{C_n}{T_n} \leq 1 \)
- How to realize concurrent programming?
- Foreground/background programming
- RTOS, and why is it more predictable?
Review: Rubus RTOS

- A scheduling hierarchy:
  
green > red > blue > background loop

```
redSetschedule bsRubusStart ...
```
Outline

• Brief review (independent tasks)
• Dependency of tasks
  – Partitioning and task interaction
  – Communication & synchronization
• Pitfalls of resource sharing (Chp. 4)
• Solutions to safe task interactions (Chp. 4)
• Implementation with Rubus RTOS
Partitioning Example

Trajectory planner → References → Gain vector → Actuator → Output

Model following control

State observer

Planner Thread → Controller Thread → Actuator Thread

Sensor Thread
Partitioning

• Why?
  – Different rates and triggering methods
  – Hardware and software constraints

• Affecting factors
  – Timing and reliability
  – *Task interactions*

• End-to-end task and end-to-end delay
Task Interaction with Itself

- PIT handler in Lab 2
- Get the \textit{refSpeed} from keyboard
- Calculate PWM command using \textit{refSpeed}
Task Interactions

• Completely independent tasks:
  – No resource sharing
  – Requires separate processors

• Functionally independent tasks with implicit dependency:
  – Shared software (non-reentrant code)
  – Shared hardware, CPU, memory, network

• Functionally dependent tasks:
  – Communication (data-flow)
  – Triggering (control-flow)
  – Timing relation (precedence constraint, synchronous in time, e.g. braking on all wheels)
Concurrent Resource Sharing

- The common thing for task dependencies
- Resource shared by at least two tasks
- Last lecture focuses on CPU sharing only
- This lecture considers CPU + memory/code sharing
Outline

• Brief review (independent tasks)
• Dependency of tasks
• Pitfalls of resource sharing
• Solutions to safe task interactions
• Implementation with Rubus RTOS
Unprotected Resource Sharing

- A **resource** is any soft-/hard-ware entity used by a thread to advance its execution.
- \( n=5 \)?

**Critical region (section):** a sequence of statements accessing the shared resource.
Unprotected Resource Sharing

- Critical section is not atomic
- Concurrent tasks are pre-emptive
- Unpredictable value (corrupted data)
More Complex Data Corruption

- \( n=1010, 1111, 0, \) and more?
- \( n=1100: \) both read+calculate, \( R1=1111, \)
  \( R2=0, \) store: \( R1.\text{low}, R2.\text{low}, R2.\text{high}, \)
  \( R1.\text{high} \)
- \( n=0010: \) B reads+calculates, \( R2=0, \) B stores
  \( R2.\text{low}, \) A reads \( (0100)+\text{calculates+stores} \)
  \( (1110), \) B stores \( R2.\text{high} \)
- Many more......

Thread A

\[
\begin{align*}
\ldots \\
n &:= n + 0\times1010 \\
\ldots 
\end{align*}
\]

\( n=0\times0101 \)

Thread B

\[
\begin{align*}
\ldots \\
n &:= n - 0\times0101 \\
\ldots 
\end{align*}
\]
Importance of Mutual Exclusion

- The resource is shared by at least two preemptive threads
- Critical region may be shared global variable, non-reentrant code, devices
- The operations of the critical region is not atomic (must allow interleaving)
- Unpredictable, difficult to debug
- Need **mutual exclusion** to protect a shared resource from concurrent access
- The protected resource is mutually exclusive resource
How to Ensure Mutual Exclusion

- Disallow pre-emption (by disabling interrupts or limiting preemption level)
- Serialize resource access by separating the threads that access the same mutually exclusive resource (via offline scheduling)
- Utilize OS supported “protection techniques”: semaphores, message queues, etc.
An Old Fashioned Approach

- Disallow **ALL** pre-emption
- Negative effect to other important interrupts

---

**Thread A**

...  
disable interrupts;  
n:=n+1;  
enable interrupts;  
...  

**Thread B**

...  
disable interrupts;  
n:=n-1;  
enable interrupts;  
...
A Naïve Approach

- Use a Boolean `free`
- A new shared resource created!
- Both threads may find `(free==1)`

```plaintext
Thread A
...
while (!free);
free=0;
n:=n+1
free=1;
...
```

```
Thread B
...
while (!free);
free=0;
n:=n-1;
free=1;
...
```

Diagram:
- `free`
- `n`
How to Improve It?

- Checking and locking should be atomic
- Replace the inefficient polling with “blocking” and “awakening” mechanism
Semaphore

- Semaphores: (Sec. 4.2)
  - Invented by Dijkstra in 1965
  - Common in OS
  - Many variants. We are using *binary semaphores* only in this course and Rubus OS

- Adopt the improvements

- Typical system calls:
  - Lock(S) or Wait(S): lock the resource. If it is already locked, then wait (or is blocked)
  - Unlock(S) or Signal(S): Unlock the resource, and wake up all waiting (or blocked) threads
If you make a mistake, signal E eight times
Binary Semaphore

- Two values: *Free* (1), *Locked* (0)
- Must have an initial value
- Thread A calls Lock(s):
  - If s is free (1), then s is locked (0) and A continues
  - If s is already locked (0), then s remains locked and A is blocked
- Thread A calls Unlock(s):
  - If s is free (1), then s remains free and A continues
  - If s is blocked (0), then s is free (1), A continues, and all threads blocked by s becomes ready
Example of Using Semaphore

- Semaphore status is managed by OS

```
Thread A
...  
Lock(s);  
\text{n:=n+1}  
Unlock(s);  
...  

Thread B
...  
Lock(s);  
\text{n:=n-1;}  
Unlock(s);  
...
```

Diagram:
- Thread A
  - Lock(s)
  - \text{n:=n+1}
  - Unlock(s)
  - ...

- Thread B
  - Lock(s)
  - \text{n:=n-1}
  - Unlock(s)
  - ...

- Semaphore S

- Semaphore status: n

- Synchronization:
  - Lock(s), B is blocked
  - Unlock(s)
Changing Thread States With Semaphores

- Creation
- Dispatching
- Pre-emption
- Termination

- Ready → Running
- Running → Pre-emption
- Blocked/Waiting → Termination
- A different thread called Unlock(s)
- Lock(s) when s is already locked by another thread
Semaphore for Synchronization

- Task synchronization using the semaphore
- Example: thread B must follow thread A

Thread A
Do job A;
Unlock(s);

Thread B
Lock(s);
Do job B;

A
Do job A
B
Do job B

(Locked)
Pro and Cons of Semaphore

+ Efficient
+ Applicable for both mutual exclusion and task synchronization
+ Widely available in virtually all OS

- A “blocking” mechanism that, if not used judiciously, can cause serious problems, e.g.,
  - Priority inversions
  - Deadlocks
  - Unmanageable system complexity (Hard to understand or debug)
Priority Inversion

- Def: A high priority task is ready to execute but blocked by the execution of a lower priority task
- Use FP scheduling, priority order: T1>T2>T3
- Unbounded blocking duration

Diagram:

- Task T1
- Task T2
- Task T3
- Blocking duration from T2 to T3
Priority Inheritance Protocol

- A task that blocks a high priority task inherits the priority of the blocked task for the duration of the blocking period.
- Priority of Ti is i, i=1, 2, 3.
- Reduced blocking time of T1.
Deadlock Caused by Shared Resources

- Multiple threads “lock” each other and no one can proceed
- Deadlock occurs when each one (thread) holds one resource (semaphore) and waits for the other one
- Priority inheritance cannot avoid it
- **Starvation**: a thread never gets CPU time to execute, because of overload and scheduling
Solutions of Deadlock

• Four necessary conditions
  1. Mutual exclusion: the forks, semaphores
  2. Hold and wait of multiple resources
  3. No pre-emption: one does not put down the fork
  4. Circular wait

• Three approaches
  1. Prevention: break any of the 4 conditions
     1) Cond.2: reserve all resources with one sem.
     2) Cond.3: preemptive resource allocation
     3) Cond.4: order resource allocation (must take fork then knife)
  2. Avoidance: run-time control
  3. Detection & recovery
Priority Ceiling Protocol (Optional)

- An extension of priority inheritance protocol
- Assign a *priority ceiling* to every resource
- Thread T can lock a semaphore S if S is not locked *and T’s priority is higher than the ceiling of all other locked semaphores*
- Avoid deadlock by avoiding cyclic waiting
- Compared with priority inheritance protocol,
  - The worst-case blocking time is shorter
  - Slightly more complex to implement
- Improvement: *immediate priority inheritance*
Example of Priority Ceiling

- Two people with priorities 1 (high), 2 (low)
- Two resources have priority ceiling 1
- If P1 starts before P2, P1 cannot be blocked
- If P2 starts and gets one resource (fork), P1 may preempt it and try to get knife
- But the ceiling of knife is 1. P1 does not have *higher* priority
- P1 is blocked. P2 continues
Message Queues

- FIFO or other scheduling methods
- When Q is not full or empty, normal operation
- When Q is full, sender wants to write
  - block the sender until empty space is available
  - sender continues, overwrites or discards
- When Q is empty, receiver wants to read
  - block the receiver until new message comes in
  - receiver continues without new message
Implementation of Message Queues

- Define the queue yourself, just another shared memory space
- Need one semaphore for mutual exclusion and probably more for synchronization
- Use OS services, easy to use (semaphores are unnecessary) but less flexible
- Mailbox in some OS services
Summary of Task Communication

• Blocking vs. nonblocking communication
  – Blocking: semaphore, queue
  – Nonblocking: queue

• Asynchronous vs. synchronous
  – Asynchronous: the sender proceeds regardless of if the message has been received. (queue)
  – Synchronous (rendezvous): the sender proceeds only when the message has been received. (semaphore, queue, acknowledgement message)
Summary of Task Communication

• With and without shared memory
  – Shared memory: semaphore, queue
  – No shared memory: remote procedure calls (RPC), call the thread function

• Direct and indirect naming
  – Direct: Use the other thread’s name
  – Indirect: message queue, mailbox

• Topology
  – 1-to-1, 1-to-*, *-to-1, *-to-∗
Outline

• Brief review (independent tasks)
• Dependency of tasks
• Pitfalls of resource sharing
• Solutions to safe task interactions
• Implementation with Rubus RTOS
Semaphores in Rubus RTOS

- **Blue services**
- Binary semaphore (Locked, Free) defined in Rubus VS
- **Lock/Wait:**
  - `blueSemTimedLock`: blocking/waiting for a duration
  - `blueSemTryLock`: nonblocking
- **Unlock/Signal:**
  - `blueSemUnlock`: nonblocking
- **Mutex**: Mutual Exclusive Semaphore using the Priority Ceiling Protocol
Message Queues in Rubus RTOS

- Basic and blue services
- Defined by Rubus VS (limited capacity)
- Basic queues: support asynchronous communication between red and blue threads
  - Write: `bsQueuePut`, nonblocking
  - Read: `bsQueueGet`, nonblocking
- Blue queues:
  - Write: `blueMsgSend`, nonblocking
  - Read: `blueMsgTimedReceive`, nonblocking/blocking by setting `timeout` value
Mailboxes in Rubus RTOS (Optional)

- Basic service
- Message queue with *unlimited* length
- Support communication between *red* and *blue* threads
- Both write and read are nonblocking
Concurrency Issues So Far

- Implementation techniques
  - Background/foreground programming
  - Use RTOS

- Design issues:
  - Real-time scheduling, cyclic executive
  - Inter thread communication, resource protection

- Challenging problems:
  - Overload, starvation (CPU time)
  - Mutual exclusion (Memory)
  - Priority inversion
  - Deadlock
Summary of Lecture 4

• Partitioning and task interaction
• Mutual exclusion, why and how?
• Semaphores for resource protection and signaling
• Priority inversion
• Priority inheritance protocol
• Deadlock, reasons and 3 types of solutions
• Message queues
• Categories and concepts of different inter task communication methods