Introduction to Embedded Systems

Chapter 10 Multitasking

Hao Zheng

Comp. Sci. & Eng.
U. of South Florida

Concurrency

source: wiki

Layers of Abstraction for Concurrency in Programs

Concurrent model of computation

dataflow, time triggered, synchronous, etc.

Multitasking

processes, threads, message passing

Processor

interrupts, pipelining, multicore, etc.

Uses of concurrency:

- Reacting to external events (interrupts)
- Exception handling (software interrupts)
- Creating the illusion of simultaneously running different programs (multitasking)
- Exploiting parallelism in the hardware (e.g. multi-core machines).
- Dealing with real-time constraints.

Threads

- An **imperative** program is a sequence of steps where data are transformed in each step.
- A **thread** is an imperative program.
- Threads maintain separate registers and stacks, but share memory space.
  - Running threads is cheaper by avoiding expensive MMU operations.
- Threads create an illusion of concurrency.
  - But are interleaved in hardware.
- After a thread is created, it is in *active* or *suspend* state.
- **Scheduler** chooses which threads to execute.

Thread Scheduling

- Without an OS, multithreading is achieved with interrupts. Timing is determined by external events.
- Generic OSs (Linux, Windows, OSX, …) provide thread libraries (like "pthreads") and provide no fixed guarantees about when threads will execute.
- Real-time operating systems (RTOSs), like QNX, VxWorks, RTLinux, Windows CE, support a variety of ways of controlling when threads execute (priorities, preemption policies, deadlines, …).
- Processes are collections of threads with their own memory, not visible to other processes. Segmentation faults are attempts to access memory not allocated to the process. Communication between processes must occur via OS facilities (like pipes or files).

Posix Threads (PThreads)

PThreads is an API (Application Program Interface) implemented by many operating systems, both real-time and not. It is a library of C procedures.

Standardized by the IEEE in 1988 to unify variants of Unix. Subsequently implemented in most other operating systems.

An alternative is Java, which typically uses PThreads under the hood, but provides thread constructs as part of the programming language.

Creating and Destroying Threads

```
#include <pthread.h>

void* threadFunction(void* arg) {
    ...
    return pointerToSomething or NULL;
}

int main(void) {
    pthread_t threadID;
    void* exitStatus;
    int value = something;
    pthread_create(&threadID, NULL, threadFunction, &value);
    ...
    pthread_join(threadID, &exitStatus);
    return 0;
}
```

Can pass in pointers to shared variables.

Can return pointer to something.
Do not return a pointer to an local variable!

Create a thread (may or may not start running!)

Becomes arg parameter to threadFunction.
Why is it OK that this is a local variable?

Return only after all threads have terminated.

Notes on Implementing Threads

- Threads may or may not begin running when created.
- A thread may be suspended between any two *atomic* instructions (typically, assembly instructions, not C statements!) to execute another thread and/or interrupt service routine.
  - This can lead to many serious problems.
- A **scheduler** needed to decide which threads to execute, when to execute, and for how long.
- Scheduling policies:
  - fairness,
  - priority
  - timing constraints.

Notes on Implementing Threads (cont’d)

- **Cooperative** multitasking does not interrupt a thread.
  - Threads required to release the processor.
  - A thread may run for long time starving other threads.
  - Most Oss use timer interrupts to pause threads.
  - A **jiffy** is the time interval when the system clock is interrupted.
- Small jiffy degrades performance, large one may violate time constraints. Usually in 1ms – 10 ms.
- Threads can often be given *priorities*, and these may or may not be respected by the thread scheduler.
- Threads may *block* on semaphores and mutexes (we will do this later in this lecture).

Typical Thread Programming Problem

"The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically."

Design Patterns, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley Publishing Co., 1995. ISBN: 0201633612):

Observer Pattern in C

```
// Value that when updated triggers notification
// of registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {…}
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {…}

// Procedure to update the value
void update(int newValue) {…}

// Procedure to call when notifying
void print(int newValue) {…}
```

Observer Pattern in C

```
// Value that when updated triggers notification of
// registered listeners.
int value;

// List of listeners. A li
// pointers to notify proc
typedef void* notifyProced
struct element {…}
typedef struct element ele
elementType* head = 0;
elementType* tail = 0;

// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {…}

// Procedure to update the value
void update(int newValue) {…}

// Procedure to call when notifying
void print(int newValue) {…}
```

```
typedef void* notifyProcedure(int);
struct element {
  notifyProcedure* listener;
  struct element* next;
};
typedef struct element elementType;
elementType* head = 0;
elementType* tail = 0;
```

Observer Pattern in C

```
// Value that
registered lis
int value;

// List of lis
// pointers to
typedef void*
struct element
typedef struct
elementType* h
elementType* t

// Procedure t
void addListen

// Procedure t
void update(in

// Procedure to call when notifying
void print(int newValue) {…}
```

```
// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {
  if (head == 0) {
    head = malloc(sizeof(elementType));
    head->listener = listener;
    head->next = 0;
    tail = head;
  } else {
    tail->next = malloc(sizeof(elementType));
    tail = tail->next;
    tail->listener = listener;
    tail->next = 0;
  }
}
```

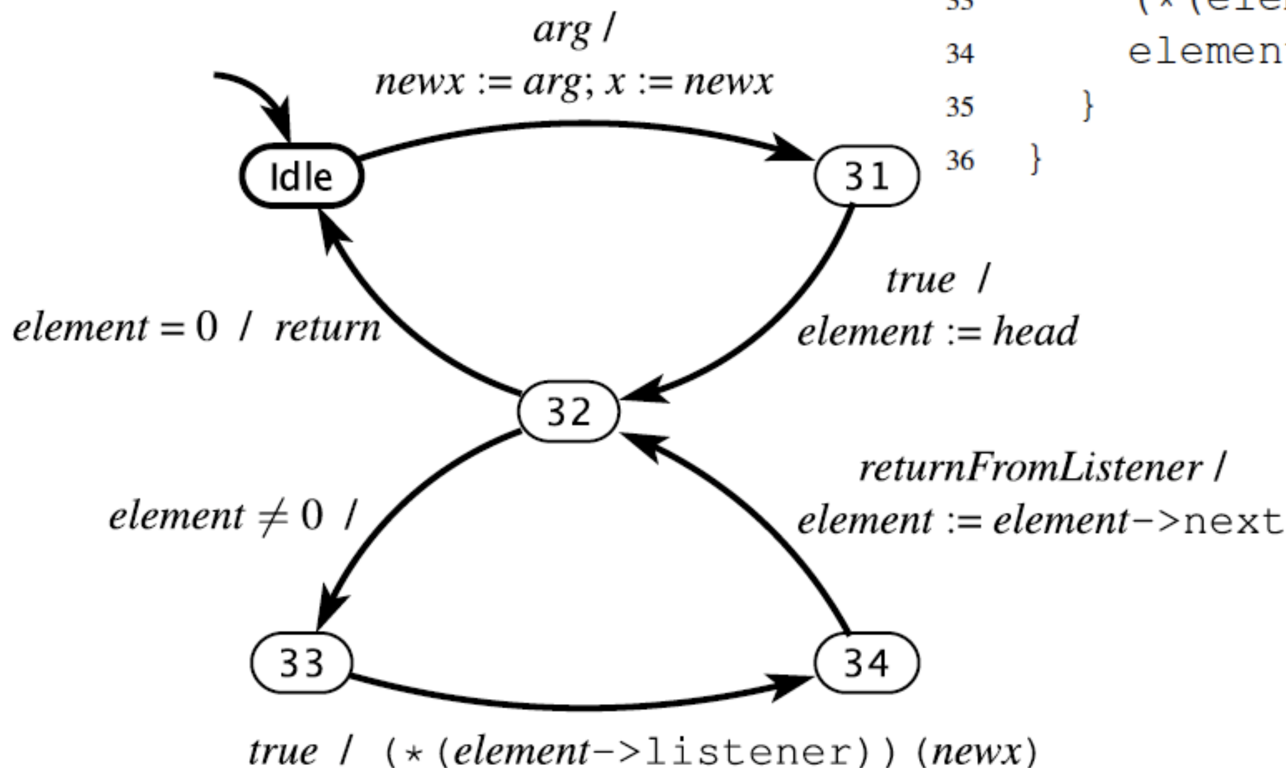
Observer Pattern in C

```
// Value that when updated triggers notification of
registered listeners.
int value;

// List of listeners. A linked list containing
// pointers to notify procedures.
typedef void* notifyProcedure(int);
struct element {…}
typedef struct
elementType* h
elementType* t

// Procedure t
void addLister

// Procedure t
void update(in

// Procedure t
void print(int
```

```
// Procedure to update the value
void update(int newValue) {
  value = newValue;
  // Notify listeners.
  elementType* element = head;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
}
```

Model of the Update Procedure

```
27  // Procedure to update x.
28  void update(int newx) {
29    x = newx;
30    // Notify listeners.
31    element_t* element = head;
32    while (element != 0) {
33      (*(element->listener))(newx);
34      element = element->next;
35    }
36  }
```



Sequential Program Example

```
// Example of notify procedure
void print(int arg) {
     printf(“%d”, arg);
}

int main(void) {
     addListener(&print);
     addListener(&print);
     update(1);
     addListener(&print);
     update(2);
     return 0;
}
```

Output: ???

Observer Pattern in C: Will this work in a multithreaded context?

```
// Value that wh
int value;

// List of liste
// pointers to n
typedef void* no
struct element {
typedef struct e
elementType* hea
elementType* tai

// Procedure to
void addListener

// Procedure to
void update(int

// Procedure to
void print(int newValue) {…}
```

```
// Procedure to add a listener to the list.
void addListener(notifyProcedure listener) {
  if (head == 0) {
    head = malloc(sizeof(elementType));
    head->listener = listener;
    head->next = 0;
    tail = head;
  } else {
    tail->next = malloc(sizeof(elementType));
    tail = tail->next;
    tail->listener = listener;
    tail->next = 0;
  }
}
```

Using Posix mutexes on the observer pattern in C

```
#include <pthread.h>
...
pthread_mutex_t lock;

void addListener(notify listener) {
  pthread_mutex_lock(&lock);
  ...
  pthread_mutex_unlock(&lock);
}

void update(int newValue) {
  pthread_mutex_lock(&lock);
  value = newValue;
  elementType* element = head;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
  pthread_mutex_unlock(&lock);
}

int main(void) {
  pthread_mutex_init(&lock, NULL);
  ...
}
```

However, this carries a significant deadlock risk. The update procedure holds the lock while it calls the notify procedures. If any of those stalls trying to acquire another lock, and the thread holding that lock tries to acquire this lock, deadlock results.

Handling Deadlock

- Deadlock is a state where the program cannot progress any further.
- The program must be aborted.
- To avoid deadlock
  - Disable interrupts (no locks used).
  - Use only one lock throughout the entire multi-threaded program. What if there are several shared resources?
  - Ensure that all locks are acquired in the same order among all threads – break module programming.

One possible “fix” with Insidious Error

```
#include <pthread.h>
...
pthread_mutex_t lock;

void addListener(notify listener) {
  pthread_mutex_lock(&lock);
  // add listener to the list...
  pthread_mutex_unlock(&lock);
}

void update(int newX) {
  pthread_mutex_lock(&lock);
  /* create a hardcopy the list
    of listeners */
10: pthread_mutex_unlock(&lock);
11: /* update listeners with ‘newX’ */
  elementType* element = headCopy;
  while (element != 0) {
    (*(element->listener))(newX);
    element = element->next;
  }
}
```

What is wrong with this?

Notice that if multiple threads call update(), the updates will occur in some order. But there is no assurance that the listeners will be notified in the same order. Listeners may be mislead about the “final” value.

10.3 Processes and Message Passing

- Processes are imperative programs with their own memory space.
  - *Threads share the same memory addr space.*
- Require a memory management unit (MMU) to protect a process from accidental access to its memory and provides address translation.
- Communication b/w processes uses pipes, files, or message passing.
- Message passing uses a carefully controlled section of shared memory only accessible through special expert written library calls.

A Simple Message Passing Application

```
void* producer(void* arg) {
   for (int i = 0; i < 10; i++) {
           send(i);
   }
   return NULL;
}

void* consumer(void* arg) {
   while(1) {
      printf("received %d\n", get());
    }
    return NULL;
}
```

A Simple Message Passing Application

```
int main(void) {
   pthread_t threadID1, threadID2;
   void* exitStatus;
   pthread_create(&threadID1, NULL, producer, NULL);
   pthread_create(&threadID2, NULL, consumer, NULL);
   pthread_join(threadID1, &exitStatus);
   pthread_join(threadID2, &exitStatus); return 0;
}
```

Difference from the multithreaded programs?

One Way to Implement Message Passing

```
/* Use a linked list as the buffer */
typedef struct element element_t;
element_t *head = 0, *tail = 0;

/* Size of the buffer */
int size = 0;

/* mutex lock */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* Signal for communication */
pthread_cond_t sent = PTHREAD_COND_INITIALIZER;
```

Message Passing: Send Operation

```
void send(int message) {
   pthread_mutex_lock(&mutex);
   /*
     Add message into the buffer
    */
I0:  pthread_cond_signal(&sent);
I1:  pthread_mutex_unlock(&mutex);
}
```

```
int get() {
   /* some initialization */
   pthread_mutex_lock(&mutex);
   while (size == 0)
     pthread_cond_wait(&sent, &mutex);
   /*
     Extract a message from the buffer
   */
   pthread_mutex_unlock(&mutex);
   return result;
}
```

- It is critical to keep the order of I0 and I1.
- Otherwise, a deadlock may occur.
- In the above code, the size can become unbounded!

Concurrency is Just Hard…

Sutter and Larus observe:

"humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations."

H. Sutter and J. Larus. Software and the concurrency revolution. ACM Queue, 3(7), 2005.

It is Threads that are Hard!

- Threads are sequential processes that share memory and are wildly non-deterministic.
- From the perspective of any thread, the *entire state of the universe can change between any two atomic actions* (itself an ill-defined concept).
- This can lead to insidious errors, races, and deadlocks.
- Problems can lurk for years, even in intensively used programs.
- The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design).