Module XXIII

Subsystem Initialization
And
Memory Marking

Self-Initializing Modules

- The goal
  - Build operating system functions in *modules*
  - Allow each module to contain multiple functions
  - Allow processes to call the functions in arbitrary order
  - Avoid having the operating system call a module initialization function explicitly
- Advantage: keeping the operating system unaware of modules means the linker can include modules that are called directly and omit modules that are not used
- Examples of modules in Xinu
  - Buffer pools
  - High level message passing
- Question: how can we make modules self-initializing?

Possible Approaches For Self-Initialization

- Approach 1: Use a global variable
- Approach 2: Create an operating system function that modules can use to initialize automatically

Using A Global Variable For Initialization

- Declare a global variable with an initial value
- Example:

```
int32  needinit = 1;
```

- Write a module initialization function, *func_init*, that
  - Tests the global variable
  - Performs initialization if the variable still has its initial value
  - Sets the global variable to a new value
- Insert code at the beginning of each module function to call *func_init*

Example Module Initialization Using A Global Variable

```
int32 needinit = 1;                  /* Non-zero until initialized    */
... declarations for other global data structures

void  func_init(void) {
        if (needinit != 0) {         /* Initialization is needed      */
              ... code to perform initialization
              needinit = 0;
        }
        return;
}
void  func_1(... args) {
        if (needinit) func_init();  /* Initialize before proceeding  */
        ... code for func_1
        return;
}
void  func_2(... args) {
        if (needinit) func_init();  /* Initialize before proceeding  */
        ... code for func_2
        return;
}
```

The Problem Of Concurrent Execution

- Multiple processes can call module functions concurrently
- Therefore, multiple processes can call the initialization function concurrently
- We need mutual exclusion to ensure correctness
- The obvious choice is a semaphore because it
  - Only affects processes using the module
  - Eliminates global disable/restore
- However
  - Using a semaphore makes self-initialization more difficult
  - An initialization function must use disable/restore to create a semaphore

Initialization With Mutual Exclusion

```
int32 needinit = 1;                 /* Non-zero until initialized    */
sid32 mutex;                        /* Mutual exclusion semaphore ID */
void  func_1(... args) {
        intmask mask;
        mask = disable();           /* Disable during initialization */
        if (needinit) func_init();  /* Initialize before proceeding  */
        restore(mask);              /* Restore interrupts            */
        wait(mutex);                /* Use mutex for exclusive access*/
        ... code for func_1
        signal(mutex);              /* Release the mutex             */
        return;
}
```

- Note: all other functions in the module must be structured the same way as this example

Initialization With Mutual Exclusion
(continued)

- The module init function must use disable/restore when creating the semaphore

```
void  func_init(void) {
        intmask mask;
        mask = disable();
        if (needinit != 0) {        /* Initialization is still needed*/
              mutex = semcreate(1); /* Create the mutex semaphore    */
              ... code to perform other initialization
              needinit = 0;
        }
        restore(mask);
        return;
}
```

Self-Initialization And Reboot

- Recall
  - Global variables are allocated in the data section
  - The data segment is only initialized when the operating system is first loaded into memory
- A problem occurs if the operating system restarts *without* reloading
  - Global variables retain the values they had before the reboot
  - Using a global variable will not work
- Example: using

```
int32 needinit = 1;
```

means that if the module is initialized when the system first runs, *needinit* will be 0 on subsequent restarts

Accession Numbers

- An alternative to using a global variable as a Boolean
- The operating system defines a global variable *boot* that is initialized to zero and is incremented each time the system restarts
- Each module defines a global variable *modinit* that is initialized to zero and is incremented each time the module has been initialized
- If *modinit* is less than *boot*, the module has not been initialized after the most recent reboot

A Potential Problem With Accession Numbers

- Consider an embedded system with the following properties
  - The hardware has a small integer size
  - The system runs forever without being downloaded again
  - The system restarts frequently
- Consequences
  - The accession counter can wrap around
  - Module initialization will fail

Goals For Module Initialization

- Make modules self-initializing (do not insert explicit initialization calls into the operating system)
- Allow in-memory restarts
- Handle the problem of wrap-around
- Make the system efficient
- Is it possible to meet all the constraints?

The Xinu Memory Marking System

- Meets all the constraints
- Requires each module to declare a location to be used as its *memory mark*

```
memmark   L;
```

- Provides a function

```
mark(L);
```

that a module uses to mark location *L*, and a function

```
notmarked(L)
```

that a module uses to test whether *L* has been marked since the last reboot

- The memory marking system guarantees that *notmarked(L)* will return 1 after the operating system is restarted until *mark(L)* is called

An Example Use Of Memory Marking

- Suppose a module requires initialization
- To use memory marking, a programmer
  - Declares a single location, *X*, to be used for memory marking
  - Defines a module initialization function as illustrated above
  - Inserts a call to mark(X) at the end of the initialization function
  - Inserts a call to test notmarked(X) at the beginning of each function
  - Have the function call the module initialization function if *X* has not been marked

An Example Use Of Memory Marking
(continued)

```
memmark loc;                          /* Memory mark for the module      */
sid32   mutex;                        /* Mutual exclusion semaphore      */

void  func_1(... args) {

        if (notmarked(loc)) {         /* Test whether initialized        */
                func_init();          /* Initialize the module           */
        }
        wait(mutex);                  /* Use mutex for exclusive access */
        ... code for func_1
        signal(mutex);                /* Release the mutex               */
        return;
}
```

- Other functions in the module must be structured the same as this one

An Example Use Of Memory Marking
(continued)

- The initialization function uses disable/restore to guarantee that only one process marks the location

```
void  func_init(void) {
        intmask mask;

        mask = disable();
        if (notmarked(loc)) {
               mutex = semcreate(1); /* Create the mutex semaphore   */
               ... code to perform other initialization
               mark(loc);
        }
        restore(mask);
        return;
}
```

Goals For A Memory Marking System

- Absolute reliability: marking should not use a probabilistic approach even if p, the probability of an accurate answer has the property that $p \rightarrow 1$
- Efficiency
  - The *mark* function should only take a few instructions
  - The *notmarked* function should only take a few instructions
- Small marks: a memory mark location (i.e., a variable declared *memmark*) is only the size of an integer
- Location independence
  - An arbitrary location in memory can be used as a memory mark (i.e., type *memmark*)
  - The locations of memory marks do not need to be registered with the memory marking system before being used

The Idea

- Keep
  - An array of marked locations, *marks*
  - An integer count of how many locations are marked, *nmarks*
- Each item in the array stores the address of the marked location
- Each marked location contains an index into the *marks* array
- A location *X* is marked if and only if the following conditions hold
  - *X* contains integer *i*
  - $0 \le i < nmarks$
  - marks[i] contains the address of *X*

The Implementation Of Memory Marking

- Memory marking declarations and the definition of *notmarked*

```
/* mark.h - notmarked */

#define MAXMARK 20              /* Maximum number of marked locations   */

extern  int32   *(marks[]);
extern  int32   nmarks;
typedef int32   memmark[1];     /* Declare a memory mark to be an array */
                                /*   so user can reference the name     */
                                /*   without a leading &                */
```

- Note the clever use of a typedef to declare a memmark as an array of a single integer, which means a reference to the name is an address

Xinu Code For Memory Marking (Part 1)

```
/* mark.c - markinit, mark */

#include <xinu.h>

int32   *marks[MAXMARK];                /* Pointers to marked locations */
int32   nmarks;                         /* Number of marked locations   */
sid32   mkmutex;                        /* Mutual exclusion semaphore   */

/*------------------------------------------------------------------------
 *  markinit  -  Called once at system startup
 *------------------------------------------------------------------------
 */
void    markinit(void)
{
        nmarks = 0;
        mkmutex = semcreate(1);
}
```

- The operating system calls *markinit* each time the system reboots

Xinu Code For Memory Marking (Part 2)

```
/*------------------------------------------------------------------------
 *  notmarked  -  Return nonzero if a location has not been marked
 *------------------------------------------------------------------------
 */
syscall notmarked(memmark loc)
{
        intmask mask;                   /* Saved interrupt mask         */

        mask = disable();

        /* See if the location has been marked */

        if (loc[0]<0 || loc[0]>=nmarks || marks[loc[0]] != loc) {
                restore(mask);
                return FALSE;
        }
        return TRUE;
}
```

Xinu Code For Memory Marking (Part 3)

```
/*------------------------------------------------------------------------
 *  mark  -  Mark a specified memory location
 *------------------------------------------------------------------------
 */
syscall mark(
          int32 *loc                    /* Location to mark             */
        )

{
        intmask mask;                   /* Saved interrupt mask         */
        mask = disable();
        /* If location is already marked, do nothing */

        if ( (*loc>=0) && (*loc<nmarks) && (marks[*loc]==loc) ) {
                restore(mask);
                return OK;
        }
        /* If no more memory marks are available, indicate an error */
        if (nmarks >= MAXMARK) {
                restore(mask);
                return SYSERR;
        }
        /* Obtain exclusive access and mark the specified location */
        marks[ (*loc) = nmarks++ ] = loc;
        restore(mask);
        return OK;
}
```

Perspective On Memory Marking

- It occupies almost no extra space — an integer (the mark) and a pointer (in the *marks* array) per module (plus a mutex ID if the module needs mutual exclusion)
- It decouples modules from the operating system (sysinit does not need to call each module’s initialization function explicitly)
- It is extremely elegant
- A single line C expression tests whether location *L* is marked

```
(L[0]<0 || L[0]>=nmarks || marks[L[0]]!=L)
```

- A single assignment does all the work of marking a location *loc*

```
marks[ (*loc) = nmarks++ ] = loc;
```

Summary

- In addition to core pieces, an operating system contains a set of optional modules
- Having the operating system initialize each module means building knowledge of the modules into the OS
- Using a global variable does not work for embedded systems that reboot often
- The Xinu memory marking system offers an elegant, efficient mechanism modules can used to self-initialize

Questions?