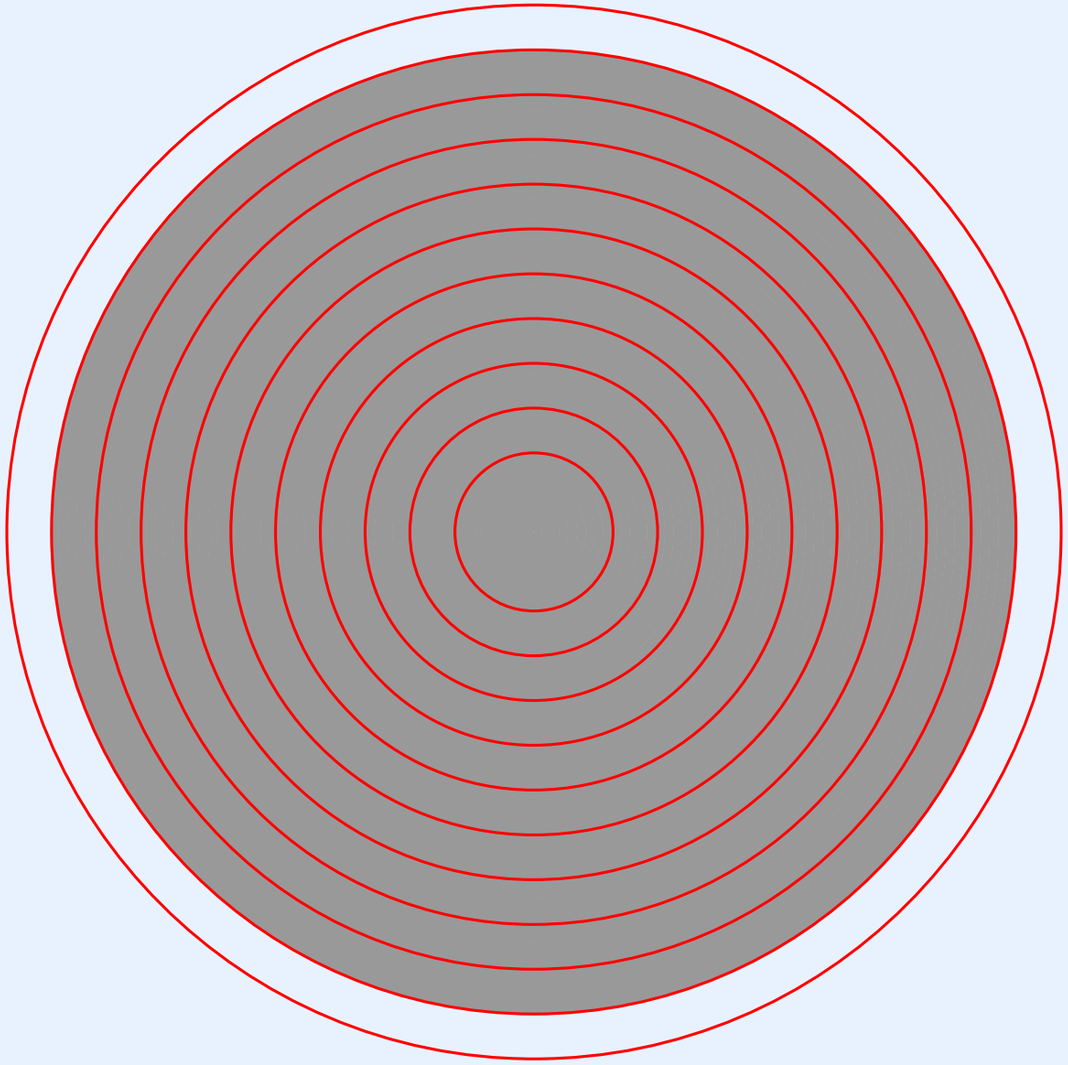
Module XV

Meta-Considerations: System Configuration And System Initialization

Relation Of Configuration And Initialization To The Hierarchy



System Configuration

Motivation For Configuration

- Hardware is modular: we build a computer by choosing
  - Processor
  - Memory size
  - Storage size and type
  - A set of I/O devices
- The goal: design an operating system that can run on as many hardware configurations as possible
- Achieving the goal: make operating system software *configurable*

Configuration And Binding Times

- A designer must choose how / when to specify each part of a configuration
- Examples (listed in order from *early binding* to *late binding*)
  - Source code creation and configuration time
  - Preprocessing time
  - Compile time
  - Link time
  - Load time
  - Operating system startup time
  - Run time
- The tradeoff: earlier binding provides more efficiency; later binding provides more flexibility

The General Trend

- Industry has moved from early binding to late binding
- Examples
  - Device-specific I/O to device-independent I/O
  - Static program loading to dynamic loading
  - Physical memory to virtual memory
  - Pre-configured device drivers to dynamically-loaded drivers
  - Pre-linked libraries to dynamically-loaded libraries
  - Monolithic kernel to dynamically-loaded kernel modules

Xinu Configuration

- Optimized for efficiency (i.e., uses early binding)
- When configuring the system, fix
  - The specific set of devices
  - Interrupt vector assignments, if needed
- By compile time, fix
  - The processor architecture (e.g., instruction set, registers)
  - Device names, and possibly bus addresses
  - Sizes of internal operating system data structures (number of processes, semaphores, etc.)

Xinu Configuration
(continued)

- By link time, fix
  - All code, including library functions
  - Drivers for devices
  - Addresses for global kernel variables
- Post link time
  - Transform the executable program into a bootable image
  - Add additional headers, if needed

Xinu Configuration
(continued)

- At system startup
  - Find the size (and locations) of free memory blocks
  - Initialize each device
  - Determine whether a real-time clock is present (optional)
  - Allocate additional kernel objects (disk and network buffers)
  - Start network processes (and possibly other background processes)

Xinu Configuration
(continued)

- At runtime, allow processes to allocate the following dynamically
  - Buffer pools
  - Message ports
  - Semaphores
  - Processes
  - Slots used for network communication

The Xinu Configuration Program

- Xinu uses a separate program named *config* that
  - Runs before the operating system is compiled and generates source code
  - Reads input from a text file named *Configuration*
  - Assigns major and minor device numbers
  - Produces two output files: *conf.h* and *conf.c*
- File *conf.h*
  - Defines device data structures, names, and constants
  - Allows the user to define additional constants and override system defaults
- File *conf.c*
  - Contains initialization code for the entire device switch table

The Format Of The Configuration File

- File *Configuration* is a text file that is divided into three sections
- The sections are separated by a percent sign on a line by itself

device type declaration section

%

device specification section

%

other configuration constants

- Note: items in the third section are appended to the end of *conf.h*

Device Type Declarations

- Allow designers to assign a name to each *type* of device
- Specify a set of default driver functions for each device type
- Document how a set of device driver functions are related
- Motivations
  - Show how a set of functions constitute a device driver
  - Provide a name for each set of driver functions
  - Allow multiple device declarations to refer to the name rather than repeating details

An Example Device Type Declaration

- Consider a device driver for a serial device
- Xinu uses the name *tty*
- The type is defined once and then used with all serial devices
- The type declaration must specify a driver function for each high-level I/O operation
- As an example, suppose
  - The driver function for *read* is named *ttyread*
  - The driver function for *write* is named *ttywrite*
  - The driver function for *getc* is named *ttygetc*
  - The driver function for *putc* is named *ttyputc*
  - ... and so on

An Example Device Type Declaration
(continued)

- The syntax used to declare the *tty* type in file *Configuration* is

```
tty:
    on uart
        -i ttyinit      -o ionull       -c ionull
        -r ttyread      -g ttygetc      -p ttyputc
        -w ttywrite     -s ioerr        -n ttycontrol
        -intr ttyhandler                -irq 11
```

- The first line declares the type name *tty*
- The phrase *on uart* specifies the underlying hardware, and allows a single type to be used with multiple brands of hardware
- The items that begin with a minus sign are keywords

Keywords Used In Type Specification And Their Meanings

Keyword	Meaning
-i	specifies the driver function that performs init
-o	specifies the driver function that performs open
-c	specifies the driver function that performs close
-r	specifies the driver function that performs read
-w	specifies the driver function that performs write
-s	specifies the driver function that performs seek
-g	specifies the driver function that performs getc
-p	specifies the driver function that performs putc
-n	specifies the driver function that performs control
-intr	specifies the driver function that handles interrupts
-csr	specifies the control and status register address
-irq	specifies the interrupt vector number to use

Device Specification Section

- The second section of file *Configuration* specifies actual devices, and has one entry for each device in the system
- An entry specifies
  - A unique name for the device
  - The type of the device (using type names declared in the previous section)
  - A set of device driver functions or values, if they differ from the default
- Example 1: on the Galileo, the *CONSOLE* device is specified:

```
CONSOLE is tty  on uart  csr 0001770 -irq 0052
```

- Example 2: on the BeagleBone Black, the *CONSOLE* device is specified:

```
CONSOLE is tty  on uart  csr 0x44E09000 -irq 72
```

Overriding Individual Items

- An entry in the device specification may override any default in the type
- Example 1: use *mygetc* for the *CONSOLE* device, and specify a CSR address and irq

```
CONSOLE is tty  on uart  csr 0001770 -irq 0052 -g mygetc
```

- Example 2: specify *CONSOLE* and *SERIAL1* to both be *tty* devices, but give each a unique CSR and IRQ; only use *mygetc* for *CONSOLE*

```
CONSOLE is tty  on uart  csr 0001770 -irq 0052 -g mygetc
SERIAL1 is tty  on uart  csr 0001370 -irq 0054
```

Minor Numbers And Device Control Blocks

- When run, the *config* program
  - Assigns each device a unique *major device number*
  - Assigns each device a *minor device number* that is unique within all devices of the same type
  - Defines a constant that specifies the number of devices of each type
- Generates *conf.h* and *conf.c* files
- Motivation
  - Each major device number defines a row in the device switch table
  - Minor device numbers allow a programmer to declare an array of control blocks for each device type, and use the minor number of a device as an index

Example Major and Minor Device Numbers And Constants

device name	device identifier	device type	minor number
CONSOLE	0	tty	0
ETHERNET	1	eth	0
COM2	2	tty	1
ETHER2	3	eth	1
PRINTER	4	tty	2

- Generated constants

```
#define Ntty   3
#define Neth   2
```

System Initialization

Starting A Computer

- General idea: the OS is *not* the first piece of software that runs
- A typical boot scenario at power-up
  - The hardware performs some basic initialization
  - The fetch-execute cycle begins executing ROM code
  - ROM code completes hardware checks and hardware initialization
  - ROM code identifies a boot device, finds an executable image, and loads a copy into memory
  - ROM code sets the hardware registers for kernel mode and physical address space
  - ROM code branches to entry point of the image

The Initial Image

- Is *not* usually an operating system
- Is known as a *bootstrap* program and has the following capabilities
  - Knows about some devices, and contains code to use them
  - Is configured with a set of devices to try
- An example bootstrap strategy
  - If a CD-ROM contains an executable image, load and run it
  - If a USB drive contains a bootable image, load and run it
  - If the hard disk contains an executable image, load and run it
  - If the Ethernet card can boot over the network, try booting
  - If the above fails, display a message for the user and halt

An Example That Illustrates Bootstrap Complexity

- The lab contains Galileo boards, and when a board boots, we want to download an operating system image over the network
- The Galileo hardware
  - Does not perform network bootstrap automatically
  - Includes a configurable bootstrap program (*grub*), but the version of grub cannot be configured to perform network booting
- Good news: the Galileo bootstrap *can* be configured to boot a file from the internal flash storage

Our Solution

- Configure the Galileo bootstrap (*grub*) to boot a file from the internal flash storage
- Place a more advanced version of *grub* on the flash (a version that has *multiboot* to check several devices)
- Unfortunately, even the multiboot version of *grub* cannot boot over the network, so use it to boot a file from the SD card
- We wrote a program that can boot over the network and placed it on the SD card (we named the program *xboot*)
- *Xboot* is designed to contact a server on the local network, download the user's Xinu image from the server, and jump to the entry point
- The cute twist: *xboot* is actually a version of Xinu in which the main program downloads an image and branches to it

Memory Occupancy During Bootstrap

- Where should a bootstrap program reside?
- If the goal is to boot an operating system, the bootstrap cannot occupy the locations that the OS will occupy
- Two approaches have been used
  - Self-relocating code: the bootstrap starts in the standard location, but moves a copy of itself to high memory and then branches to the copy
  - The bootstrap is bound to high memory addresses: the bootstrap program is compiled and linked to run at a high memory address (beyond the memory into which the operating system is loaded)
- *Xboot* uses the second approach

When Xinu Begins

- Startup code is written in assembly language to
  - Initialize the hardware
  - Zero the bss segment
  - Initialize the co-processor, if one is present
  - Create an environment suitable for C (e.g., sets the stack pointer)
  - Invoke the C initialization function, *nulluser*

When Xinu Begins

- *Nulluser* invokes function *sysinit* (also written in C) to initialize the OS
  - Performs any remaining platform initialization
  - Initializes memory management hardware and free list
  - Initializes each operating system module
  - Initializes each I/O device and its driver
  - Transforms from a sequential program to a concurrent system
    * Enables interrupts
    * Creates a new process to execute *main*
    * Lets the current computation become the NULL process

The Transformation From Program To Process

- The transformation is the most significant aspect of initialization
- It occurs in Xinu function *sysinit*
- The entire transformation is surprisingly simple and elegant
  - Fill in the process table entry for process 0
  - Make the state *PRCURR*
  - Set *currpid* to zero
  - Create and resume a process for main program
- When *resume* is called, scheduling and context switch proceed as usual — a concurrent system is executing

Subsystem Initialization

Self-Initializing Modules

- Goal
  - Build a *module* consisting of multiple functions
  - Allow processes to call the functions in any order
  - Avoid having the operating system call a module initialization function
- Advantage: the operating systems does not need to be aware of all modules — the linker only puts modules in the operating system that are used
- Example modules
  - Buffer pools
  - High level message passing
- Question: how can self-initializing modules be implemented?

Self-Initializing Modules

- Two approaches can be used
  - Use a global variable
  - Create an operating system function to aid with initialization

Using A Global Variable For Initialization

- Declare a global variable with an initial value
- Write an initialization function for the module that
  - Tests the global variable
  - Performs initialization if the variable still has its initial value
  - Sets the global variable to a new value
- Program each function in the module to call the initialization function

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
- The initialization function must use disable/restore when creating 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 on the module are structured the same way as this one

Initialization With Mutual Exclusion (continued)

- The init function uses the semaphore for mutual exclusion

```
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 section is only initialized when the operating system image is downloaded into memory
  - If the operating system reboots, global variables retain whatever value they had before the reboot
- Consider a module that uses global variable

```
int32 needinit = 1;
```

- If the module is initialized, variable *needinit* will have the value zero during subsequent reboots

Using Accession Numbers To Handle Reboot

- The operating system defines a global variable *boot* that starts at zero and is incremented during each reboot
- Each module defines a global variable *minit* that starts at zero and is incremented each time the module has been initialized
- If *minit* 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 without being downloaded again
  - The system reboots frequently
- Consequences
  - The reboot counter can wrap around
  - Module initialization will fail

The Xinu Memory Marking System

- Solves the problem of wrap-around
- Uses two new system calls
- Each module declares a location to be used as its *memory mark*
  - *memmark* L;
- The module then uses the two system calls
  - *mark(L)*

    to mark location *L*, and
  - *notmarked(L)*

    to test whether *L* has been marked since the last reboot
- The memory marking system guarantees that *notmarked(L)* will return 1 after reboot until *mark(L)* is called

An Example Use Of Memory Marking

```
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 are structured the same as this one

An Example Use Of Memory Marking
(continued)

- The initialization function creates a semaphore and marks the location used for the module

```
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;
}
```

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;
extern  sid32   mkmutex;
typedef int32   memmark[1];     /* Declare a memory mark to be an array */
                                /*   so user can reference the name     */
                                /*   without a leading &                */

/*------------------------------------------------------------------------
 *  notmarked  -  Return nonzero if a location has not been marked
 *------------------------------------------------------------------------
 */
#define notmarked(L)            (L[0]<0 || L[0]>=nmarks || marks[L[0]]!=L)
```

- 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)

```
/*------------------------------------------------------------------------
 *  mark  -  Mark a specified memory location
 *------------------------------------------------------------------------
 */
status  mark(
          int32 *loc                    /* Location to mark             */
        )
{
        /* If location is already marked, do nothing */

        if ( (*loc>=0) && (*loc<nmarks) && (marks[*loc]==loc) ) {
                return OK;
        }

        /* If no more memory marks are available, indicate an error */

        if (nmarks >= MAXMARK) {
                return SYSERR;
        }

        /* Obtain exclusive access and mark the specified location */

        wait(mkmutex);
        marks[ (*loc) = nmarks++ ] = loc;
        signal(mkmutex);
        return OK;
}
```

Perspective On Memory Marking

- It occupies almost no extra space — one pointer 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;
```

Modern OS Configuration

- Operating systems have placed increasing emphasis on runtime configuration
- There are two basic paradigms
  - Adaptation: a system checks for one of several devices at startup and selects an appropriate device driver (e.g., the system recognizes any of four NICs)
  - Dynamic device configuration: a system permits devices to be plugged in or disconnected while the system is running
- The Internet makes it easier to locate and download driver software for dynamically connected devices

Runtime Device Configuration Requirements

- Hardware that can detect and report the presence of a new device
  - A standard protocol that a new device uses
  - A protocol that allows the processor to interrogate a device without knowing the device type
- An operating system capable of loading drivers dynamically

An Example: USB Devices

- The hardware uses a single interrupt vector for the USB *host controller*
- A device driver for the host controller is configured statically
- The driver for the host controller acts as a dispatcher
- When a new device appears, the host controller software
  - Polls the device over the USB to determine which device connected
  - Loads a driver for the device
  - Records the location of the driver
- When one of the USB devices interrupts, the host driver dispatches the interrupt to the driver for that device

The Balancing Act

- Automated configuration is handy
- Manual configuration allows an owner to specify exactly which devices will be used and avoid loading drivers dynamically
- What is the correct balance?
- Users often need some level of control
- When a computer with a printer connects to a network, should others on the network be allowed to use the printer?
- If a computer has an Ethernet interface connected to the Internet and a Wi-Fi interface, should other Wi-Fi devices be allowed to connect to the Internet by sending packets through the computer?

Summary

- System configuration
  - Permits a single operating system to run on multiple hardware configurations
  - Adapts to details such as
    * Peripheral devices
    * Memory size
  - Tradeoff: later binding increases flexibility, but reduces performance

Summary
(continued)

- System initialization is complex and hardware-dependent
  - An operating system is not the first piece of software that runs
  - The operating system must complete initialization and create an environment needed by C before calling a C function
  - The most significant aspect of initialization occurs when a C program transforms itself into an operating system that has concurrent processes
- The Xinu memory marking system allows modules to self-initialize

Questions?