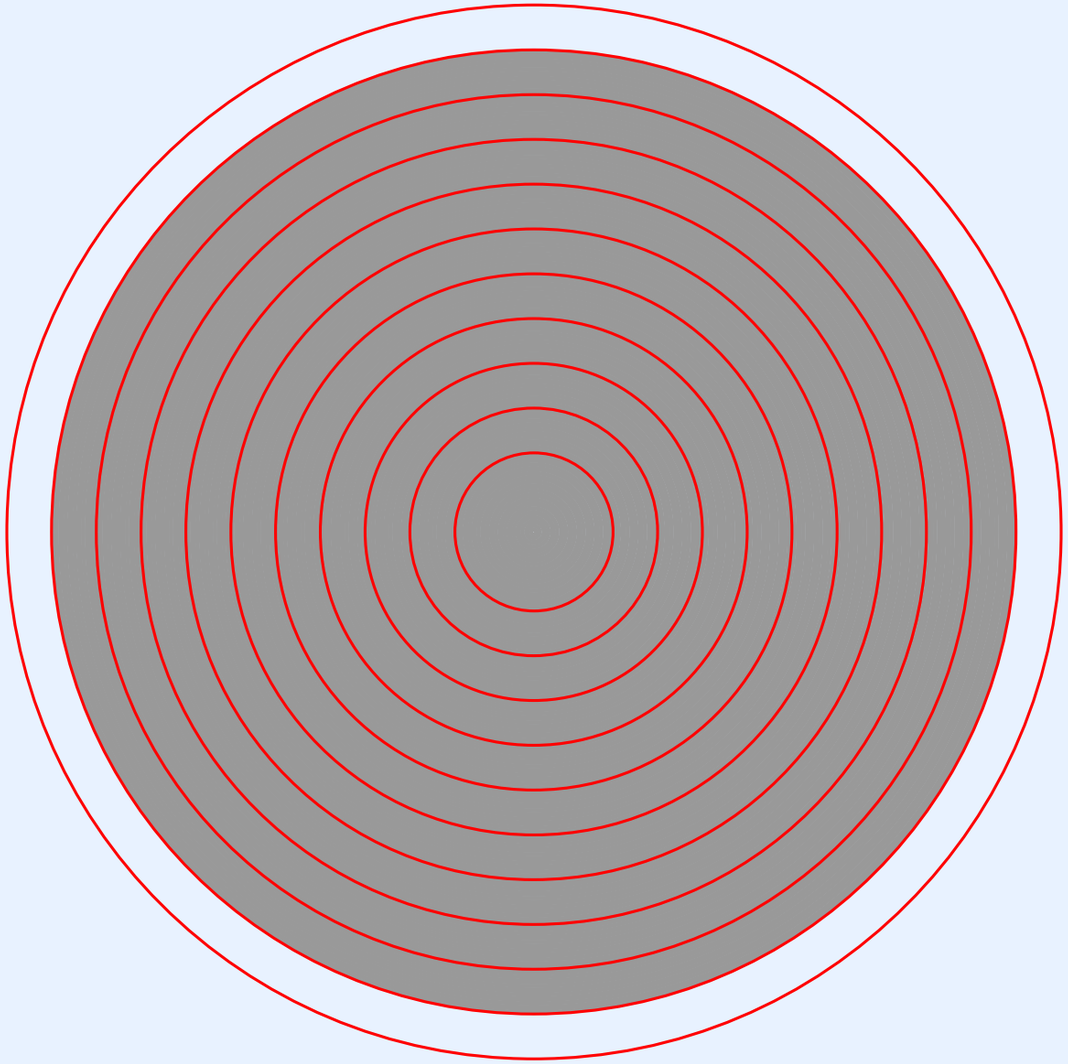
Module XXV

Meta-Consideration: System Configuration

Relation Of Configuration To The Hierarchy



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 functions to device-independent I/O functions
  - 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

Microkernel Design

- General idea
  - Start with a minimal operating system kernel (microkernel)
  - Design other operating system functions as independent modules
  - Boot the microkernel and only load other modules as needed
  - Possibly: unload a module when no longer needed
- Several microkernel systems have been built
- The current conclusion: microkernels fill niche roles
  - Most modules are needed all the time
  - Dynamically loaded libraries solve many problems for which microkernels were originally intended

Tradeoffs Between Early And Late Binding

- Early binding
  - More efficient
  - Lower startup delay
  - Less flexible
- Late binding
  - More flexible
  - A single system can run on a range of hardware
  - Some hardware resources may go unused

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 applications, library functions, and shell commands
  - Drivers for all 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 the device switch table, device names, and constants
  - Allows the user to define additional constants and override system defaults
- File *conf.c*
  - Generates 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

Driver Definition

- Question: what is a device driver?
- Answer: a set of functions that provide an interface to a device, including a function the operating system calls to initialize the device, upper-half functions applications call to perform I/O and lower-half functions invoked when the device interrupts
- In Xinu, one can only tell which functions a driver uses by looking at the Configuration file

Outside the Configuration file, one finds individual functions; only the Configuration file specifies which functions have been selected to form a given device driver.

Data Associated With A Device

- In addition to a set of functions, a device driver may need additional data
  - The address on the bus assigned to the device's Control and Status Registers (CSRs)
  - The interrupt Request number (IRQ) assigned to the device
- For embedded systems, CSR and IRQ values are assigned statically when the hardware is designed
- For larger systems, the operating system must use a bus protocol at startup to find the CSR and IRQ values for each device
- Consequence: the Xinu Configuration file allows, but does not require, a user to specify IRQ and CSR values

Keywords Used In Type Specification And Their Meanings

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

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
SERIAL2	2	tty	1
PRINTER	3	tty	2
ETHERNET2	4	eth	1

- Generated constants

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

Modern OS Device Configuration

- Operating systems have placed increasing emphasis on runtime configuration of devices
- There are two basic paradigms
  - Adaptation (used by embedded systems): 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 new devices

Runtime Device Configuration Requirements

- Hardware must be able to detect and report the presence of a new device
  - Each device must follow a standard for identification
  - The device and processor must agree on a protocol that allows the processor to interrogate the device without knowing the device type or details
- The operating system must be capable of loading drivers dynamically

An Example Of Dynamic Configuration: USB Devices

- The hardware uses a single interrupt vector for the USB *host controller*
- A device driver for the host controller is configured at startup
- 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, but may make choices for the user
- Examples
  - 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 a Wi-Fi interface plus an Ethernet interface that connects to the Internet, should other Wi-Fi users be allowed to connect to the Internet by sending packets through the computer?
- Manual configuration allows an owner to specify devices and avoid loading drivers dynamically, but requires more effort
- What is the correct balance?
- Vendors try to separate polices from configuration, but the list of policy decisions seems to grow longer and longer

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
- An operating system is not the first piece of software that runs
- Simply booting an operating system may involve multiple bootstrap programs

Questions?