

Systems Programming

Advanced Operating Systems Lecture 2

Systems Programming

- Why is systems-level programming different?
 - Systems programs interact with hardware
 - Systems programs have memory and data layout constraints
 - Systems programs strongly driven by bulk I/O performance
 - Systems programs maintain long-lived, concurrently accessed, state
- We are used to high-level programming, ignoring low-level details
→ the details matter when writing systems programs

Challenges

- Operating systems are evolving:
 - more power constrained
 - more real-time applications
 - more embedded
 - more concurrent
 - more safety critical
 - more security critical
- Are C and Unix the right programming model for the operating systems of the future?

Challenges: Power Constraints

- Many systems run on constrained hardware
 - May have limits on power consumption (e.g., battery powered)
 - May have to be physically small and/or robust
 - May have strict heat production limits
 - May have strict cost constraints
- Used to throwing hardware at a problem, writing inefficient – but easy to implement – software
 - Software engineering based around programmer productivity
 - Constraints differ in embedded systems – optimise for correctness, cost, then programmer productivity

Challenges: Ensuring Predictable Timing

- Real-time systems → scheduling theory can prove correctness, if system timing predictable
- Numerous sources of unpredictability
 - Timing variation due to dependence on algorithm input values → measure
 - Blocking due to resource access
 - Preemption by higher priority tasks or interrupt handlers
 - Processor cache improves average timing, with poor worst-case bounds
 - Virtual memory – address translation, paging, memory protection
 - Memory allocation and management – `malloc()` or garbage collector
- Avoid by defensive programming
 - Disable or avoid features that cause timing variation
 - Optimise for predictability, not raw performance

Challenges: Embedded Systems

- Constraints on embedded systems:
 - Must interact with hardware to manipulate their environment – custom device drivers and low-level hardware access in application code
 - Safety critical or simply hard to upgrade – strong correctness constraints
 - Often resource constrained, with a low-level programming model
- Issues differ from those inherent in traditional desktop application programming

Device Drivers

- Devices represented by bit fields at known address
 - Bit-level manipulation to access fields in control register
 - Code needs memory address and size of control register, layout, endianness, and meaning of bit fields within register
- C allows definition of bit fields and explicit access to particular memory addresses via pointers – needed for implementation of device drivers
- Illusion of portability – standard C does not specify:
 - Size of basic types (e.g., a char is not required to be 8 bits, an int is not required to be 32 bits, etc.)
 - Bit and byte ordering
 - Alignment or atomicity of memory access
 - Each environment defines these – e.g., `<stdint.h>` and `<limits.h>` – but type checking is limited
- Device drivers written in C a frequent source of bugs
- Other languages (e.g., Ada, Rust) provide strict definitions and allow stronger type checking

```
struct {
    short errors      : 4;
    short busy        : 1;
    short unit_sel     : 3;
    short done         : 1;
    short irq_enable   : 1;
    short reserved     : 3;
    short dev_func     : 2;
    short dev_enable   : 1;
} ctrl_reg;

int enable_irq(void)
{
    ctrl_reg *r = 0x80000024;
    ctrl_reg tmp;

    tmp = *r;
    if (tmp.busy == 0) {
        tmp.irq_enable = 1;
        *r = tmp;
        return 1;
    }
    return 0;
}
```

Example: hardware access in C

Sources of Bugs in Device Drivers (1)

Name	Description	Total faults	Device prot. violations	S/W protocol violations	Concurrency faults	Generic faults
USB drivers						
rtl8150	rtl8150 USB-to-Ethernet adapter	16	3	2	7	4
catc	el1210a USB-to-Ethernet adapter	2	1	0	1	0
kaweth	kl5kusb101 USB-to-Ethernet adapter	15	1	2	8	4
usb net	generic USB network driver	45	16	9	6	14
usb hub	USB hub	67	27	16	13	11
usb serial	USB-to-serial converter	50	2	17	13	18
usb storage	USB Mass Storage devices	23	7	5	10	1
IEEE 1394 drivers						
eth1394	generic ieee1394 Ethernet driver	22	6	6	4	6
sbp2	sbp-2 transport protocol	46	18	10	12	6
PCI drivers						
mthca	InfiniHost InfiniBand adapter	123	52	22	11	38
bnx2	bnx2 network driver	51	35	4	5	7
i810 fb	i810 frame buffer device	16	4	5	2	5
cmipci	cmi8338 soundcard	22	17	3	1	1
Total		498	189 (38%)	101 (20%)	93 (19%)	115 (23%)

Fix through device documentation and better language support for low-level programming?

Can we address these through improvements to the supporting infrastructure for device-drivers?

Summary cause of bugs found in Linux USB, Firewire (IEEE 1394), and PCI drivers from 2002–2008
[from L. Ryzhyk *et al.*, “Dingo: Taming device drivers”, Proc. EuroSys 2009, DOI 10.1145/1519065.1519095]

Device protocol violations are mis-programming of the hardware, software protocol violations and concurrency faults are invalid interactions with the rest of the Linux kernel

Sources of Bugs in Device Drivers (2)

- What causes software protocol violations and concurrency faults?
 - Misunderstanding or misuse of the device driver API, especially in uncommon code paths (e.g., error handling, hot-plug, power management)
 - Incorrect use of locks leading to race conditions and deadlocks
- Bad programming and poor documentation of kernel APIs and locking requirements?
- Or error-prone programming languages, concurrency models, and badly designed kernel APIs?

Type of faults	#
Ordering violations	
Driver configuration protocol violation	16
Data protocol violation	9
Resource ownership protocol violation	8
Power management protocol violation	8
Hot unplug protocol violation	5
Format violations	
Incorrect use of OS data structures	29
Passing an incorrect argument to an OS service	19
Returning invalid error code	7

Table 2. Types of software protocol violations.

Type of faults	#
Race or deadlock in configuration functions	29
Race or deadlock in the hot-unplug handler	26
Calling a blocking function in an atomic context	21
Race or deadlock in the data path	7
Race or deadlock in power management functions	5
Using uninitialised synchronisation primitive	2
Imbalanced locks	2
Calling an OS service without an appropriate lock	1

Table 3. Types of concurrency faults.

[from L. Ryzhyk *et al.*, “Dingo: Taming device drivers”, Proc. EuroSys 2009, DOI 10.1145/1519065.1519095]

Improving Device Drivers – Engineering

- Model device drivers in object-oriented manner
 - Device drivers generally fit some hierarchy
 - Use object-oriented language; encode common logic into a superclass, instantiated by device-specific subclasses that encode hardware details
 - May be able to encode protocol state machines in the superclass, and leave the details of the hardware access to subclasses (e.g., for Ethernet or USB drivers)
 - May be able to abstract some of the details of the locking, if hardware similar enough
 - Might require multiple inheritance or mixins to encode all the details of the hardware, especially for multi-function devices
- Implementation choices – device driver framework
 - Linux kernel implements this model in C, with much boilerplate
 - MacOS X uses restricted subset of C++ within kernel – simplifies driver development by encoding high-level semantics within framework, leaves only device-specific details to individual drivers

Apple, Inc. “I/O Kit Fundamentals”, 2007

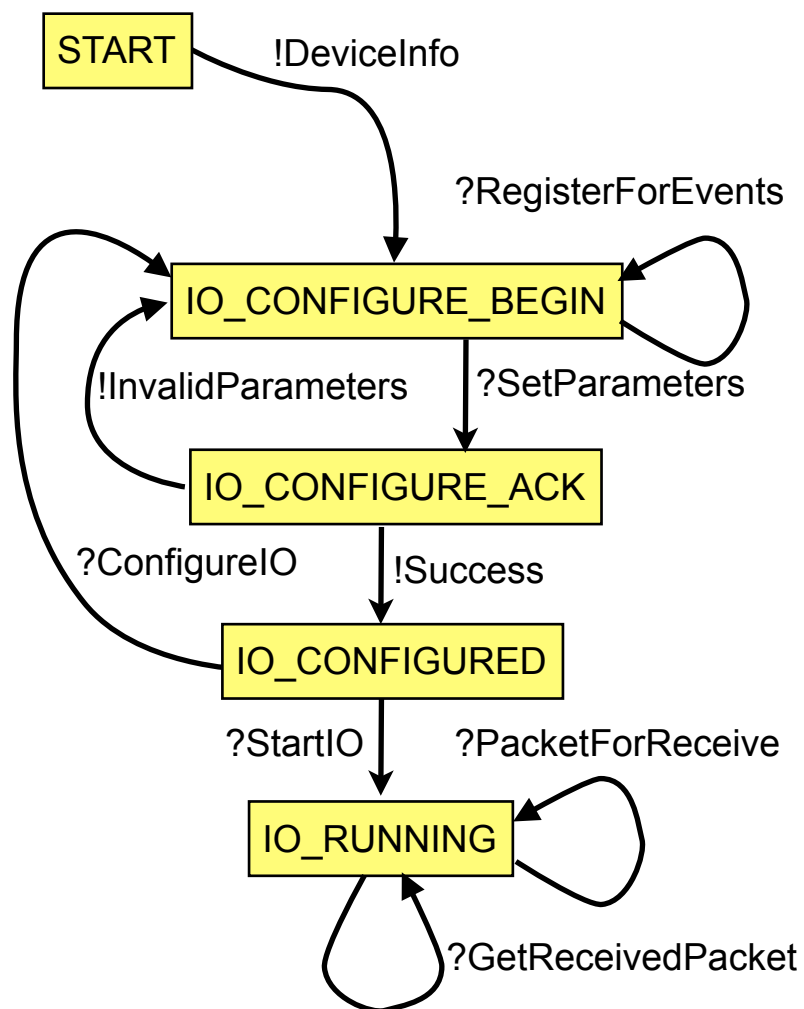
<http://developer.apple.com/library/mac/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/IOKitFundamentals.pdf>

Improving Device Drivers – State Models

- An ad-hoc device driver model is common
 - Many bugs due to poor specification and documentation of the model
 - Use of object-oriented languages can improve this somewhat, but need careful integration into C-based kernels
- Possible to formalise design as a state machine
 - Make underlying state machine visible in the implementation – MacOS X I/O Kit models incoming events, but not the states, allowable transitions, or generated events
 - Could formally define full state machine in source code, allow automatic verification that driver implements the state machine for its device class, and model checking of the state machine
 - Can be implemented within existing languages, by annotating the code
 - Fits better with more sophisticated, strongly-typed, languages, that can directly model system

Improving Device Drivers – State Models

Example: the Singularity operating system from Microsoft Research



```

contract NicDevice {
  out message DeviceInfo(...);
  in message RegisterForEvents(NicEvents.Exp:READY
c);
  in message SetParameters(...);
  out message InvalidParameters(...);
  out message Success();
  in message StartIO();
  in message ConfigureIO();
  in message PacketForReceive(byte[] in ExHeap p);
  out message BadPacketSize(byte[] in ExHeap p, int
m);
  in message GetReceivedPacket();
  out message ReceivedPacket(Packet * in ExHeap p);
  out message NoPacket();

  state START: one {
    DeviceInfo! → IO_CONFIGURE_BEGIN;
  }
  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents? →
      SetParameters? → IO_CONFIGURE_ACK;
  }
  state IO_CONFIGURE_ACK: one {
    InvalidParameters! → IO_CONFIGURE_BEGIN;
    Success! → IO_CONFIGURED;
  }
  state IO_CONFIGURED: one {
    StartIO? → IO_RUNNING;
    ConfigureIO? → IO_CONFIGURE_BEGIN;
  }
  state IO_RUNNING: one {
    PacketForReceive? → (Success! or BadPacketSize!)
      → IO_RUNNING;
    GetReceivedPacket? → (ReceivedPacket! or
      NoPacket!)
      → IO_RUNNING;
    ...
  }
}
    
```

Listing 1. Contract to access a network device driver.

G. Hunt and J. Larus. "Singularity: Rethinking the software stack", ACM SIGOPS OS Review, 41(2), April 2007. DOI:10.1145/1243418.1243424

Improving Device Drivers – Discussion

- Focus on low-level implementation techniques and high-performance in many device driver models
- Not necessarily appropriate in embedded systems?
- Raising level of abstraction can reduce error-prone boilerplate, allow compiler to diagnose problems

Challenges: Correctness and Security

- Systems may be safety or security critical
- Might need to run for many years, in environments where failures either cause injury or are expensive to fix
 - Medical devices, automotive or flight control, industrial machinery
 - Can you guarantee your system will run for 10 years without crashing?
 - Do you check all the return codes and handle all errors?
 - Fail gracefully?
- Security vulnerabilities in networked systems
 - Privacy and confidentiality – both of data in transit, and against attacks from the network
 - Any networked service is a potential security risk

Evolving Systems Software

- How to address these challenges?
 - Alternative programming models – better languages and tools
 - Alternative kernel designs and system architectures

Alternative Programming Models

- Move away from C as an implementation language
 - Lack of type- and memory-safety leads to numerous bugs and security vulnerabilities
 - Limited support for concurrency – race conditions, locking problems – makes it unsuitable for modern machine architectures
- Move towards architectures with a minimal kernel, and strong isolation between other components of the operating system
 - The monolithic part of a kernel is a single failure domain; this needs to be reduced to a minimum → microkernel architecture
 - Easier to debug and manage components when they're isolated from each other, and communicate only through well-defined channels

Type- and Memory-Safe Languages

- Type safe language → protects its abstractions
 - Undefined behaviour prohibited by compiler/type system
 - The language specification can require that array bounds are respected, specify the error response to violation, etc.
 - More sophisticated type systems can catch more complex errors – e.g., enforce a socket is connected, check that an input string is correctly escaped to avoid SQL injection, ...
- Requires both compile- and run-time checking
 - The *type system* specifies legal properties of the program “for proving the absence of certain program behaviours”
 - Some properties can be statically checked by a compiler: a faulty program will not compile until the bug is fixed
 - Some properties require run-time checks: failure causes a controlled error
 - Doesn't guarantee system works correctly, but ensures it fails in a predictable and consistent way
- Doesn't require byte-code virtual machine; can have efficient implementation

```
-->cat tst.c
#include <stdio.h>

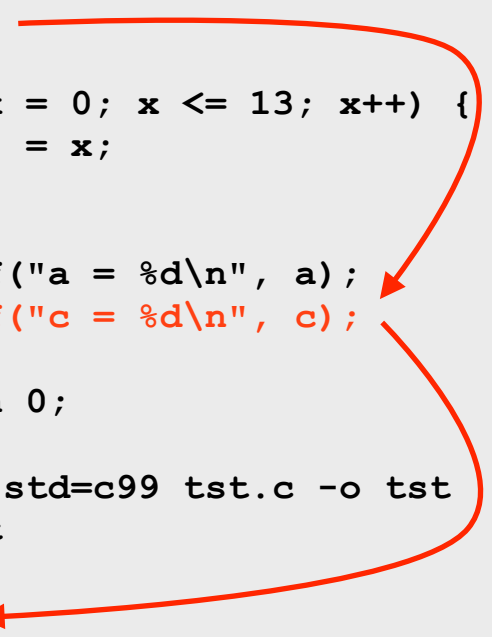
int main()
{
    int x;
    int a;
    int b[13];
    int c;

    a = 1;
    c = 2;

    for (x = 0; x <= 13; x++) {
        b[x] = x;
    }

    printf("a = %d\n", a);
    printf("c = %d\n", c);

    return 0;
}
-->gcc -std=c99 tst.c -o tst
-->./tst
a = 1
c = 13
-->
```



Modularity and Microkernels

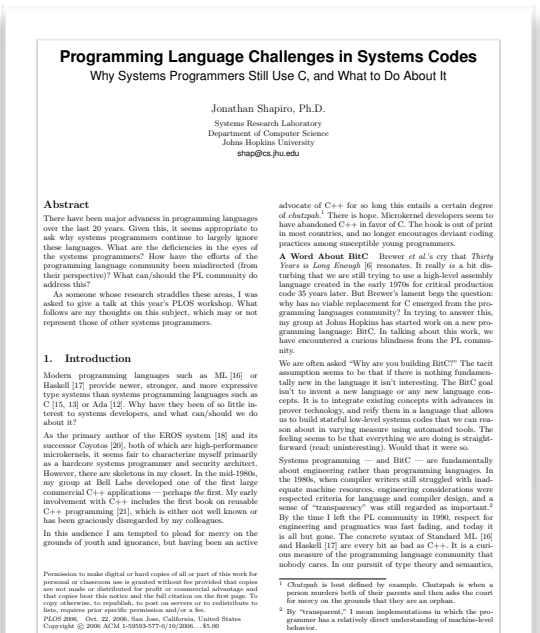
- Desirable to separate components of a system, so failure of a component doesn't cause failure of the entire system
- Microkernel operating system
 - Strip-down monolithic kernel to essential services; run everything else in user space communicating via message passing API
 - This includes devices drivers, network stack, etc.
 - Kernel just managing process scheduling, isolation, and message passing
 - Widely used in embedded systems, where robustness and flexibility to run devices for unusual hardware are essential features
 - But typically poor performance: frequent context switches expensive, due to need to cross kernel-user space boundary, manage memory protection, etc.

Strongly Isolated Systems

- A possible solution:
 - Microkernel that enforces all code written in a safe language (e.g., by only executing byte code, no native code)
 - This includes device drivers and system services running outside the microkernel
 - Type system prevents malicious code obtaining extra permissions by manipulating memory it doesn't own – done entirely in software; no need to use MMU to enforce process separation
 - A software isolated message passing process architecture – loosely coupled and well suited to multicore hardware
 - Example: the Singularity operating system from Microsoft Research
- Relies on modern programming language features
 - Combination is novel, but individual pieces are well understood

Discussion and Further Reading

- Systems software has unique constraints
- Correctness, robustness, security, performance
- Low-level programming model was necessary for efficiency – are there alternative models for modern systems?



- Further reading:
 - J. Shapiro, “Programming language challenges in systems codes: why systems programmers still use C, and what to do about it”, Proc. 3rd workshop on Programming Languages and Operating Systems, San Jose, CA, October 2006. DOI:10.1145/1215995.1216004
 - G. Hunt and J. Larus. “Singularity: Rethinking the software stack”, ACM SIGOPS OS Review, 41(2), April 2007. DOI:10.1145/1243418.1243424
 - Read these before the tutorial next week – come prepared to discuss

