Low-level Programming

Colin Perkins

http://csperkins.org/teaching/2003-2004/rtes4/lecture19.pdf

UNIVERSITY
of
GLASGOW

Lecture Outline

- Administrivia
- Lessons to learn from the programming assignment
- Interrupt and timer latency
- Memory issues
  - Protection
  - Virtual memory
  - Allocation, locking, leaks and garbage collection
  - Caching
- Power, size and performance constraints
- System longevity
- Case studies

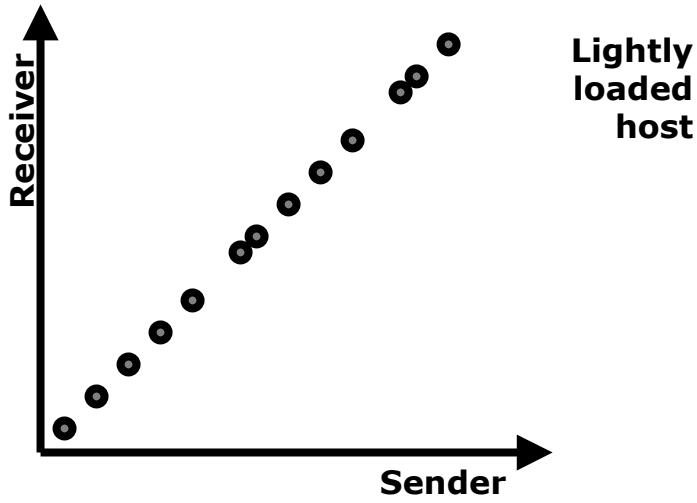
Administrivia

- Return of assignments:
  - Problem sets 1 & 2 should be returned this week
  - Programming assignment will be returned after Easter
- Revision lecture: 27th April, 3pm

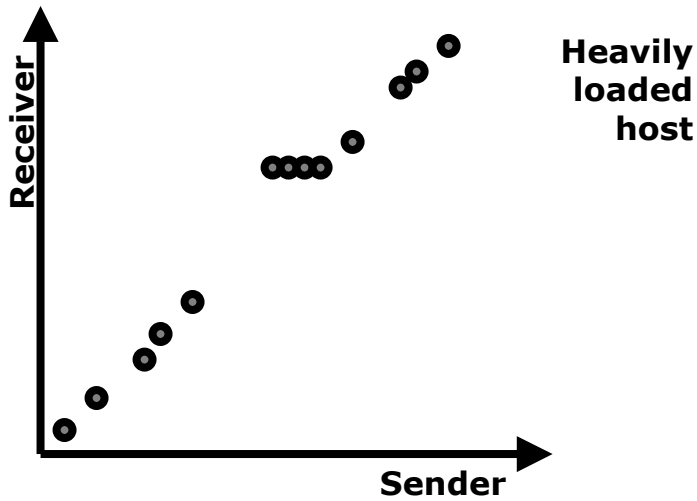
- Examination:
  - 80% of grade derived from exam mark
    - Answer 3 out of 4 questions
    - 2 questions from each part of the course
  - Sample exam distributed today
  - Sample answers will be provided and discussed in the revision lecture

Programming Assignment

- Lightly loaded hosts show little jitter
- Inter-packet spacing is usually 20ms
  - Do see occasional 10ms spacing
- Scheduling granularity is 10ms, but process gets woken up on the tick after it's deadline has passed



- Heavily loaded hosts show much more variability, packets being delayed and arriving in bursts
- Linux is not suitable for VoIP with 10ms packet duration
  - Would be suitable with 20ms packet duration on lightly loaded hosts
  - On heavily loaded hosts, jitter buffer would be too large for interactive use



Programming Assignment: Lessons to Learn

- A general purpose operating system is not optimised for real-time use; it has a different purpose:
  - Fair sharing of resources
  - Good average case performance; at expense of worst case
  - Protection from untrusted users

  all of which worsen real-time performance!

- Some problems can be mitigated:
  - Recompile kernel with 1ms clock interrupt
  - Run as root, using high priority POSIX real-time scheduling
  - Lock pages into memory, to avoid memory paging delays
  - Run on a lightly loaded system, forbid access to hardware devices unless audited for latency

  …essentially, close the system

"You Can Put Racing Stripes on a Bulldozer,
But it Won't Go Any Faster"

…the RTLinux manifesto

Interrupt and Timed Task Latency

- Standard Linux has ~600μs interrupt handler latency, and hard to get scheduled tasks to execute within 20,000μs of their scheduled time
  - High frequency timer interrupt can reduce scheduling latency to ~1000μs
- RTLinux claims a maximum 15μs interrupt handler latency, scheduled tasks execute within 35μs of their scheduled time

- Why such a difference?
  - Preemptable microkernel, with single address space
    - No context switch, user-to-kernel mode, overhead
  - No virtual memory or memory protection
    - No paging delays
    - No delays while page tables adjusted
  - Device drivers designed with minimal non-preemptable sections
    - Light-weight, prioritised, threads fire in response to interrupts

Memory Protection/Context Switch Overhead

- Many embedded systems use a single flat address space
  - Applications, shared libraries, kernel, devices all visible
  - A system or library call is equivalent to a function call

- Makes system calls, interrupts, very fast and predictable
  - No context switch to kernel mode
  - No adjustment of MMU page tables
- Consequences:
  - No isolation between applications, or between applications and the kernel
  - A change to one part implies that the entire system has to be revalidated; difficult as systems become larger

⇒ Some systems offer limited protection

  - Read only mapping of program/system text; IRQ vectors
  - Optional full memory protection

Memory Protection

- Consequences of offering memory protection:
  - Unpredictable latency
    - May take longer to task switch to/from a protected task
  - Memory overhead
    - Protection provided on a per-page basis, leads to wastage
    - Overhead of maintaining the page tables and protection maps
  - Code overhead
    - Operating system is required to trap illegal access and recover system to a safe state
- Which is easiest: proving the system correct, or writing handlers to safely recover from all possible failures?

Virtual Memory: Address Translation

- Two aspects to virtual memory:
  - Address translation
  - Paging to disk

- Address translation (or "virtual contiguity") is the act of making a fragmented block of physical memory appear to be a single contiguous block
  - Useful in dynamic systems, since it enables requests for large blocks of memory to be allocated when there is no physically contiguous block available
  - Adds overhead, since the system has to manage address translation tables
    - Uses memory, increases context switch time
    - Complicates DMA device access
- Often better to pre-allocate static pools of memory for real-time tasks; allocate from them to the application
  - Manage the sub-division of address space within the application

Virtual Memory: Locking and Paging

- Disk based virtual memory is supported by many systems that run both real time and non-real time tasks
- Should be obvious that paging to disk is harmful to the real time tasks
- System will provide control to prevent paging:
  - POSIX: **`mlock(addr, len)`** and **`mlockall()`**
    - Privileged, but would have significantly improved performance in the exercise when under load
  - Windows NT can specify that all memory owned by a particular thread is locked
  - LynxOS has a mode where pages of higher priority tasks are locked in memory, but new allocations can page out memory belonging to lower priority tasks

Memory Leaks and Garbage Collection

- An embedded system has to run for a long period of time, without user intervention
- Real-time garbage collection – with predictable latency, at controlled times – **is** possible…
- …but most embedded systems still programmed in C

- Problematic in small or long-lived systems…
  - Most C programs have memory leaks due to programmer error
  - Better to pre-allocate all memory in static buffers, to avoid the chance of a memory leak
  - Be **very** careful to free resources (memory) after use
  - Do you **always** check for out of memory errors? And recover gracefully?
    - Remember the recovery code cannot allocate memory
    - This may include the stack frame needed to make a function call!

What is a Small System?

You may be running on a Z80 processor, with 64kbytes RAM…

- The QNX 4.x microkernel is approximately 12kbytes in size
- The VRTX microkernel is typically 4-8kbytes in size

For comparison…

```
-->uname -srm
FreeBSD 4.9-STABLE i386
-->cat tst.c
int main()
{
        return 0;
}
-->gcc tst.c -o tst
-->ls -l tst
-rwxrwx---  1 csp  csp  4206 Mar 16 21:15 tst
-->strip tst
-->ls -l tst
-rwxrwx---  1 csp  csp  2704 Mar 16 21:15 tst
-->
```

…on Linux, the binary is 50% larger

Effects of Cache

- You may be running on a more modern processor…
  - PowerPC 405CR embedded processor
    - 32 bit RISC processor, compatible with desktop PowerPC
    - 133MHz or 266MHz clock speed
    - CodePack™ compression of executables
    - Likely has several megabytes of memory
  - Relatively cheap, comparatively high performance, low power

- Has a small cache, which you may want to disable:
  - Processor and memory speeds are closely matched
    - Compare to desktop processor, with order magnitude difference
  - Simpler to predict memory access times without the cache
  - Cache improves average response times, but introduces unpredictability

Power, Size and Performance Constraints

- Many embedded systems are battery powered or run in power sensitive environments
- What influences power consumption?
  - Power consumption $\propto$ (clock speed)^2
  - Memory size
  - Processor utilization
- May have to be physically small and/or robust
- May have strict heat production limits
- May have strict cost constraints
  - That processor is slower, but 10¢ cheaper, the production run is 1 million, you paid your salary for the next couple of years…
- We're used to throwing hardware at a problem, and writing inefficient – but easy to implement – software
  - Software engineering based around programmer productivity
  - The constraints may be different in the embedded world…

System Longevity

Real time embedded systems are often safety critical…

- Medical devices
- Automotive or flight control
- Railway signalling
- Industrial machinery

…or just difficult to upgrade

- CD or DVD player
- Washing machine
- Microwave oven

- May need to run for several years, in an environment where failures either kill people, or are incredibly expensive to fix
  - Do you check all the return codes and handle all errors?
  - Fail gracefully?
  - Can you guarantee your system will run for 10 years without crashing?

Case Studies

- VxWorks
- QNX
- Symbian

Case Study: VxWorks

- Monolithic kernel
- Implements most of POSIX with real time extensions
- Proprietary APIs to control more advanced features
  - Message queues with timeouts
  - Control of priority inheritance on semaphores
  - User processes can enable/disable interrupts
- Defaults to a single address space, with address translation
  - Processes can request memory protection, if desired
  - Processes can control which regions of memory are cached

- Focus on hard real time, deeply embedded systems
  - Runs on the Mars rovers, Pathfinder
    - Pathfinder had problems due to uncontrolled priority inversion causing some tasks to miss their deadlines
    - Caused system to repeatedly reset to safe state
    - Enough debugging code left in that the problem could be resolved, and new code uploaded

Case Study: QNX

- Pure microkernel system
  - Many optional components, scales for 12kbytes to run on high end SMP machines with gigabytes of memory
- Native support for threads with a single address space
  - Memory protection optional
- Message passing abstraction for inter-task communication
  - Very efficient, due to single address space
  - Tasks inherit priority of the messages
  - Messages can be blocking, variable sized, or fixed size non-blocking
- Network stack, TCP/IP
- Full GUI, web browsers, Java, etc

- Focus on real time embedded, but user-facing, systems

Case Study: Psion/Symbian

- Psion series 5mx – precursor to Symbian mobile phones
- 16M RAM, 16M ROM
- 36MHz ARM710 processor
- Preemptive multitasking, GUI, C++
- Software: agenda, word processor, spreadsheet, address book, email, web browser, calculator, jotter, sketch, voice notes, Java
- Runs for ~1 month on 2 AA batteries
- Mine has run for almost 5 years without rebooting…
  - Small, efficient, power-aware and robust code

- Focus on telephony and soft real time systems
  - Often run under a hard real time OS using a two-level scheduler

Summary

By now you should…

- Be thinking about the system issues, and how features that improve general purpose performance hinder real time jobs
- Be thinking about the constraints on embedded systems, and differences in how they are engineered
- Know a little about different systems that are available

Tomorrow: summary and overview of the module