

What tools are needed to generate a microkernel?

- microkernel maybe self supporting
 - initially they are built on a *host*
 - and *downloaded* to a *target*
- during open systems internals we will be developing a microkernel which has been written in: Modula-2, C and a very small amount of assembly language

Development host and facilities

- target system and development host are not necessarily same

- tools found on host include:
 - compilers
 - assemblers
 - linkers
 - editors
 - disassemblers
 - debuggers
 - simulators
 - emulators
 - libraries
 - build

Tools

- build
 - takes executables and places instructions and data *into* the target

- once the target is running it will have no link with the host
 - at this point the target is said to be *stand-alone*

- compiler
 - often termed a cross compiler if the target processor is different than the host
 - even if microprocessor is same the libraries will be different and this is called cross development

Tools

- linker may be specific to target processor
- assembler dependent on target processor
- debugger
 - some cross development systems allow remote debugging of a microkernel
- emulator
 - *hardware* and *software* tool which allows the designer to analyze the system executing at full speed
 - normally a critical component is substituted by a "plug" attached to the emulator

Tools

- simulator
 - *software* tool which allows designer to analyze the system behavior
 - does not run at full (target) speed

- advantages and disadvantages
 - emulator - very good for finding hardware bugs when software is running
 - simulator - very good for finding software bugs

Simulator

- functionality allows you to single step any section of code and single step backwards in time
 - examine simulated hardware events which cause software to take actions (interrupts)
 - devices can be modeled (DMA, interrupts)
 - same software as final system

open systems internals

- in this course we will be studying:
 - how to build a microkernel
 - key components that are at the center of microkernel
 - debugging techniques

open systems internals

- background reading:
 - D. Comer, *Operating System Design The XINU Approach*, Prentice-Hall (PC edition), 1988, ISBN 0-13-638313-0
 - A.M. Lister, *Fundamentals of Operating Systems*, 3rd Edition, The MacMillan Press Ltd, 1984
 - John O’Gorman, *Operating systems with Linux*, Palgrave, ISBN 0-333-94745-2, 2001
 - Lewin Edwards, *Embedded system design with a Limit Budget*, Newnes, ISBN 0750676094

- although these books do not directly address a microkernel, much of the content and practice can be applied

Development host

- our host system will be a UNIX clone (GNU/Linux) and target will be a naked PC.
 - all software will be written in C

Development tools in detail

- host and target systems may be same or completely different
 - so different that they might not have same microprocessor
 - or even same endianness!

- thus our development microkernel requires a different breed of tools
 - **cross development tools**

Cross development tools

- we might expect the following cross development tools:
 - assembler, linker, archiver, compiler, debugger, emulator, simulator, cross development libraries, bootstrap loader
 - a number of these are complex!

Assembler/archiver/linker family

- assembler takes in ASCII instructions and emits an object file
 - object file syntax might be completely different from the object file format found on native development system

- linker and archiver
 - read in an object file in a format and emit another object file or executable file

- **note** the executable file might again be very different from the native development system format

Compiler

- the one component which changes the least between the development microkernel environment and native host development!
- takes in source and generates ASCII assembler instructions
- compiler *in raw form* does not use any link libraries
 - but might require header files in C

Cross development libraries

- usually need special low level components to be rewritten for each different target microkernel
 - maybe some of the higher level libraries are generic (at source level) between different target systems
 - some even might be borrowed from the host. eg string library

Debugger

- hardest of all! why?
- debugger needs to run on both target and host at the same time
 - the two halves needs to communicate via remote procedure calls!

How does a debugger work?

- traditionally under a normal operating system (say UNIX) a debugger operates in the following way:
 - the debugger is executed and it prompts for a child process to debug
 - the user replies (normally with the name of an executable)
 - the debugger then starts the new executable (debuggee) running

- the **debuggee** starts by running some initialization code which in turn will call the **debugger**
 - which indicates that communication has been established

- after communication established the debugger can monitor values of variables, stack frames and insert break points

Action of a break point

- most microprocessors have break point instruction
 - when a break point is executed it typically causes an interrupt to occur
 - from this interrupt it is possible to find out the value of the program counter, stack, frame pointers

- debuggers exploit this break point functionality to probe the executable for data and stop it running at the users request
 - stop at a source code line number
 - stop at the start of a function
 - stop at the next line

- all achieved via the break point and debugger

microkernel development and the debugger

- we have already seen that the debugger requires the following:
 - a form of break point on the target microprocessor
 - the ability to examine the executable and find out the address of
 - a function
 - a variable

microkernel development and the debugger

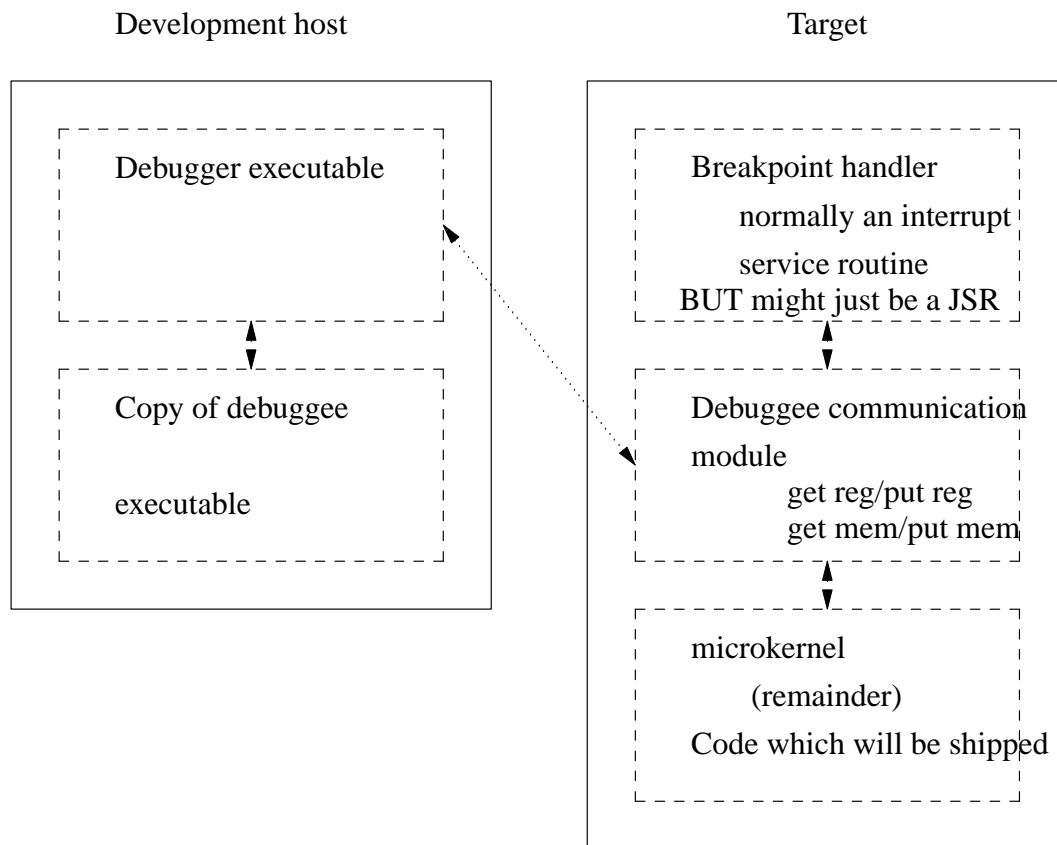


Requirements	
host	target
examine executable	set brk points
examine symbol table	respond to brk
understand target insts	
comm with target	comm with host

microkernel development and the debugger

- communicate with target requires
 - send break
 - send new register values
 - get/set memory contents

Anatomy of a microkernel debugger



Anatomy of a microkernel debugger

- note the communication link between host and target
 - this might be a RS232 cable
 - or a UNIX socket
 - or any other *digital pipe*

GNU binutils and GDB

- debuggers, assembler, linkers, archivers can be difficult to create for a cross development environment
- fortunately the GNU software foundation has written binutils package and GDB package
 - binutils consists of an assembler, linker, archiver and library of object file, executable file formats
- the assembler has been split into 3 components
 - front end which takes ASCII instructions and enters them internally as binary
 - middle stage which computes all references, labels etc
 - back end which writes out the appropriate object file format

Binutils

- the binutils allows a user to configure:
 - which object file format to use SRECORDS or elf, aout, etc
 - which front end to use, (if there isn't one for your microprocessor then you write it yourself!)
 - the assembler knows about the following instruction sets: alpha, VAX, 68k, 29k, m88k, [345]86, h8300, mips, sparc, h8500, hp300, smp, i860, i960, ns32k, ppc, tahoe, z8k

- to add the smp processor required 1101 lines of C (front end) and some configuration details
 - and a disassembler (242 lines of C)
 - endian ness, object file format, debugging information " .stabs "

GNU GDB package

- GNU distribute all source to all their tools
 - debugger is no exception, the source can be configured to operate on the host in a normal operating system environment to debug host processes
 - GDB uses the binutils object and executable file handling routines

- however it can also be configured to run on a host and debug a different (microkernel) target
 - you need to integrate the target GDB communication stub with your existing microkernel
 - agree on a communication method

Porting GDB to another target

- firstly complete the binutils port, make sure that the binutils can either
 - run on a new native target
 - or cross assemble, link etc

- configure GDB to use the same object code libraries as binutils
 - make sure the disassembler component exists

- configure GDB to understand target specific entities:
 - 32 bits in a register
 - function offset start
 - stacks grow downwards
 - break point instruction code and length

Porting GDB to another target

- whether the PC is decremented after a break point occurs
- the number of registers
- frame point register number
- program counter register number
- how function return values are implemented
- frame chain following function (up/down)

Installing GNU/Linux on an iPAQ

- those interested in installing GNU/Linux on an iPAQ might wish to follow this [link](http://people.via.ecp.fr/~clem/nist/ipaq.php#t3.3) `<http://people.via.ecp.fr/~clem/nist/ipaq.php#t3.3>`.

- four main phases
 - installing a new bootloader on the iPAQ
 - installing the linux kernel
 - configuring TCP/IP
 - installing the operating system across the TCP/IP network

iPAQ cross development tools

- the GNU C compiler, binutils and glibc can be configured to target the iPAQ
 - iPAQ uses a 200 Mhz SA1110 arm processor
 - typically has 90 Mb ram

- cross development tools are much preferred to slow native tools

Configuring tools

- we have to build/install following components:
 - binutils
 - unpack_headers
 - gcc
 - newlib
 - glibc

Ordering configuration

- build the cross development binutils
 - strongarm-linux-elf-as, strongarm-linux-elf-ld, etc
- unpack the kernel headers from strongarm-linux
 - contains function prototypes and system call definitions
- now build a cross C compiler
 - strongarm-linux-elf-gcc

Ordering configuration

- cross compiler, assembler and headers complete
 - now need minimal libraries
- build crt0.o
 - found in newlib package
- now build cross glibc
 - contains open, close, read, strcat, printf, etc

Detail: headers

- are initially unpacked into: /usr/local/strongarm-linux-elf/include
- defines system calls
- ```
#define SYS_open __NR_open
#define SYS_read __NR_read
#define SYS_write __NR_write
#define SYS_close __NR_close
```
- taken from: `bits/syscall.h`

## Use of system headers

- used so that gcc can build a library of functions
  - each of which maps onto a system call

```
int read (int fd, void *ptr, int len)
{
 return syscall(SYS_read, fd, ptr, len);
}
```

## crt0.o

- required as it is the first piece of user code executed
  - this code calls your `main` function
  - the source to this is sometimes assembler and sometimes C
- duty is to set up the arguments for `main` and environment for `main`

## crt0.c for the iPAQ

```
#include <stdlib.h>

extern char **environ;
extern int main(int argc, char **argv, char **envp);

void _start(int args)
{
 /*
 * The argument block begins above the current
 * stack frame, because we have no return
 * address. The calculation assumes that
 * sizeof(int) == sizeof(void *). This is
 * okay for i386 user space, but may be
 * invalid in other cases.
 */
 int *params = &args-1;
 int argc = *params;
 char **argv = (char **) (params+1);

 environ = argv+argc+1;
 exit(main(argc, argv, environ));
}
```

# Hello world

- remember hello world might be written:

```
#include <stdio.h>

int main (int argc, char *argv[],
 char *environ[])
{
 printf("hello world\n");
 return 0;
}
```

- many applications ignore the third parameter to main!

## Cross glibc (C libraries)

- required as they provide printf, open, read, close, etc
- they will in turn perform system calls and utilize the strongarm  
`#include <syscall.h>` file
- C libraries are extensive and take longer to build than the linux kernel!
- once all these pieces are installed we can build any C program (which only uses libc).

## C compiler driver

- the C compiler driver will perform a number of activities for users
  - preprocess the C source
  - compile the preprocessed source
  - link the object files and libraries to form an executable

- examine this with:

```
strongarm-linux-elf-gcc -v hello.c
```

- ```
cc1 -lang-c ... hello.c -o /tmp/ccuvJUp0.s  
as -o /tmp/ccsay5Hn.o /tmp/ccBNqUFj.s  
collect2 ... crt0.o -lgcc -lc /tmp/ccsay5Hn.o
```

Building more compilers

- the gcc package and associated front ends can be combined to produce a number of compilers
 - C, f77, ADA, Java, C++
 - a number of more front ends are in development: COBOL, Modula-2, Pascal

- we will build a cross development Modula-2 compiler

Cross development tutorial exercise

- using the cross development C compiler build
 - hello world from Colin Morris' lecture

uLinux Tutorial

- in this tutorial we will build a Linux kernel for the strong-arm microprocessor
 - create a minimal filesystem
 - create a tiny application

- then we will run a strong-arm simulator and see micro GNU/Linux boot and watch our application run

Create a tiny application

- open up an editor (emacs) and write the classic "hello world" program in C
- you need to add the following code after your code finishes to ensure that the simulator exits cleanly
 - add this code after your `printf`

Create a tiny application

```
fclose(stdout);  
fclose(stderr);  
usleep(1); /* this allows time for vmlinux to  
           flush its devices before we  
           terminate simulation */  
  
__asm__ __volatile__ (  
    "mov    r6, #0\n"  
    "mov    pc, r6\n"  
);
```

- save your code in a file `hello.c`

- compile it using the following commands:

```
strongarm -g -o init hello.c
```

Strong arm kernel

- now unpack the linux kernel sources, do this by opening a terminal and typing:

- `unpackstrongarmkernel`

- now we need to compile the sources, so type:

- `compilestrongarmkernel`

- this will take a few minutes

Minimal filesystem

- now we need to create a minimal GNU/Linux filesystem

- to do this type in the following:

- `unpacklinuxfs`

- now type the following to pack up the filesystem with your application

- `createulinuxfs`

Simulate an iPAQ

- now we can simulate strong-arm uLinux, type the following:

- `simstrongarm`

uLinux

- examine the file system contents (a copy will be in your current directory)
- your program is placed in `/bin/init` but normally this would be the GNU/Linux `init` program
 - `init` as its name suggests is the first system utility in user space to run