

Laboratorio di Sistemi Operativi
Anno Accademico 2006-2007

Software Development with uMPS
Part 3
Mauro Morsiani

Copyright © 2007 Mauro Morsiani
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found at:
<http://www.gnu.org/licenses/fdl.html#FOSS>

uMPS software development

Developing software with uMPS requires:

basic knowledge of UNIX environment and commands

knowledge of the uMPS architecture, GUI, and software development conventions

setup of an effective debugging environment

© 2007 Mauro Morsiani

2

uMPS software development

uMPS simulator main commands:

`umps`: the simulator itself

`umps-elf2umps`: to convert the output of the compiler to files the simulator will understand

`umps-objdump`: to analyze these files

`umps-mkdev`: to build disks and tapes for the simulator

uMPS software development

uMPS simulator-related files:

In the `support/` and `example*/` directory:

`*.rom.umps`: the ROM files

`*.core.umps`: the kernel to be loaded

`*.stab.umps`: the kernel *symbol table*

`*.aout.umps`: for programs other than kernel

other `*.umps` files (`term0.umps`, `printer0.umps...`): files associated to devices

`/etc/umpsrc` and `.umpsrc` (`ls -a` to see it): the simulator configuration file

`elf32*.x` files: configuration files for the cross-compiler

uMPS software development

uMPS other essential components:

some libraries (XForms, libelf) for building the simulator

`libumps.e` (and `libumps.o`) under `support/`: uMPS library for interfacing with ROM services, **CPO** registers and issue TLB-related and SYSCALL instructions

`crtso.o` and `crti.o`: kernel and program startup functions

`const.h` and `types.h` under `support/h`: some useful types and constants (eg. processor state definition)

a *cross-compiler* based on GNU gcc:

`mipsel-linux-gcc` for little-endian uMPS (on x86)

`mips-linux-gcc` for big-endian uMPS (on PPC)

the `make` utility

© 2007 Mauro Morsiani

5

uMPS software development

libumps: uMPS support library

`libumps` acts as a wrapper, allowing to:

- access ROM routines

- access special **CPO** registers

- issue TLB-related and SYSCALL instructions

`libumps` is composed by two parts:

- `libumps.e`: to be included in C programs source (see it for library description and details)

- `libumps.o`: to be linked with other object files to make an executable file

© 2007 Mauro Morsiani

6

uMPS software development

Common issues in uMPS development:

setup of critical registers (esp. `$gp`, `$sp`, `PC`, `CP0.Status`):
check values and bit masks

data structure corruption: it's easy to make coding mistakes
or forget to (re-)initialize data structures

overlapping of stack spaces among different processes

unwanted compiler optimizations:

- use `volatile` (esp. when accessing device registers)

- use subroutines

- do not optimize (no `-O` flags)

no `printf()`!

© 2007 Mauro Morsiani

7

uMPS software development

Breakpoint, Suspect and Trace: the debugger's tools of trade

Breakpoint: a position (an address) in the code; simulation stops when reaches it (may be referred to with a *symbol* + offset)

Suspect area: a memory range (a set of addresses) containing data (array, variables...) under exam; may be a *Read* suspect and/or a *Write* suspect (may be referred with a *symbol*)

Suspect: simulation stops when an access of the appropriate type (R, W) is made to the suspect area

Traced range: a range of memory addresses selected for showing addresses may be physical or virtual ones

only physical addresses may be traced in uMPS

© 2007 Mauro Morsiani

8

uMPS software development

Advanced uMPS debugging strategies

how to replace `printf()`:

- initialize a global character array and provide some basic access function able to write contents (copy chars) into it
- trace the array (= show it in the GUI)
- set a write suspect on the array (or a breakpoint on the access function)
- see `pltest.c` for an example

how to check internal variables and execution flow: use *debugging functions*

- define debugging functions and insert them into the code
- set breakpoints on debugging functions
- variables to be shown can be passed as parameters

uMPS software development

Debugging functions: an example

```
void debug(int where, int var1)
{
    return;
}
```

...

```
var_to_check = some_complex_calculation;
debug(10, (int)var_to_check);
```

...

then check **\$a0-\$a1** for values when breakpoint is reached

uMPS software development

Debugging functions: an example (cont'd)

```
if (some_condition) {  
    debug(14,TRUE);  
    ...  
} else {  
    debug(15,FALSE);  
    ...  
}
```

uMPS software development

General software development strategies:

- define your goals (make a top-down analysis)
- share opinions with other group members
- keep a log; printing helps
- take your time: practice makes perfect
- backup, backup, backup
- know your tools (or know how to know...)
- do not "fear the machine"
- read the manual! (and the documentation, and the newsgroup, and...)
- look at examples (and Google...)
- be creative and curious
- (when all else fails) ask for help: don't panic! ☺

uMPS software development

How to set up an effective debugging environment:

Basic UNIX tools:

command reference: `man` and `info`

show and search: `more/less`, `diff` and `grep`

editor: `vi`, `emacs`, `joe`, ...

compilation: `make` and `makefile`

compiler flags: `-v -E -S -c -o -ansi -pedantic -Wall`

backup: `cp` and `tar` (plus `mv` and `rm`)

log: `>&`, `script` and `history`

Advanced code development tools:

source control: `rcs`, `cvs`, `subversion`, ...