

Laboratorio di Sistemi Operativi  
Anno Accademico 2006-2007

**Software Development with uMPS**  
**Part 2**  
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#FDL>

## uMPS software development

### Software development with uMPS architecture:

Assembly language development is cumbersome: only ROM and specific HW interface libraries require to be developed in Assembly

C language (and GNU gcc compiler) is the preferred alternative

The use of a compiler requires a number of conventions to be defined:

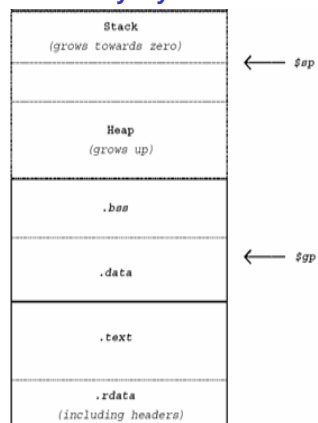
- memory layout and program structure
- register usage conventions
- object and executable file formats

© 2007 Mauro Morsiani

2

## uMPS software development

### A generic program memory layout:



© 2007 Mauro Morsiani

3

## uMPS software development

### Generic program memory layout explained:

one program in memory = four areas:

*.text* area: contains the code instructions, is typically read-only

*.data* area: contains the program static data (variables, arrays, etc.); it may be subdivided into:

*.data proper*: for data already initialized when program starts

*.bss (block started by symbol)*: for other variables and static data

*heap*: contains the program dynamic data (malloc'd structures allocated at runtime: it requires an OS to manage them)

*stack*: for procedure calls management

All these areas will have specific addresses in memory

The executable file may contain only *.text* and *.data* areas

© 2007 Mauro Morsiani

4

## uMPS software development

### uMPS program memory layout:

Actually, two types of programs may exist in uMPS:

kernel and kernel-like programs, starting with VM off

user programs running with VM on

uMPS memory layout for kernel programs:

*.text* area: starts at 0x2000.1000 (first frame in RAM is reserved to ROM handlers) and is *padded* to a multiple of 4KB

*.data* area: starts from the next 4KB frame right after *.text* area

*heap*: not available (no OS to manage it, remember?)

*stack*: starts from **RAMTOP** and grows down

© 2007 Mauro Morsiani

5

## uMPS software development

### uMPS program memory layout (cont'd):

uMPS memory layout for programs running with VM on:

*.text* area: starts at 0x8000.0000 (first address in virtual memory kuseg2 segment) and is *padded* to a multiple of 4KB

*.data* area: starts from the next 4KB frame right after *.text* area

*heap*: available if the OS supports it

*stack*: starts from 0xBFFF.FFFC (OS may decide to use another value) and grows down

The memory layout is provided to the compiler using `elf32*.x` configuration files

different files exist for big- and little-endian uMPS, and for kernel and user programs

© 2007 Mauro Morsiani

6

## uMPS software development

### uMPS register usage conventions:

Register Number	Register Name	Use
\$0	<i>\$0</i>	Hardwired to zero value.
\$1	<i>\$at</i>	Reserved for the assembler: used in computations and for holding temporary values.
\$2-\$3	<i>\$t0-\$t1</i>	Used for expression evaluations and for holding integer function results; may also be used to pass the static link during nested procedures call.
\$4-\$7	<i>\$a0-\$a3</i>	Used to pass the first four integer-compatible type actual arguments at procedure call; not preserved across procedure calls.
\$8-\$15, \$24-\$25	<i>\$t0-\$t9</i>	Temporary registers, used for expression evaluations: not preserved across procedure calls.
\$16-\$23	<i>\$s0-\$s7</i>	Saved registers: their values are preserved across procedure calls.
\$26-\$27	<i>\$k0-\$k1</i>	Reserved for kernel use.
\$28	<i>\$gp</i>	Global pointer.
\$29	<i>\$sp</i>	Stack pointer.
\$30	<i>\$s8 or \$t0</i>	A saved register (like <i>st0-s7</i> ), or used as frame pointer.
\$31	<i>\$ra</i>	Procedure call return address; also used for expression evaluation.

© 2007 Mauro Morsiani

7

## uMPS software development

### uMPS register usage conventions (cont'd):

Some interesting registers:

**\$v0**: contains the return value when C functions return

**\$a0-\$a3**: contain the first four parameters in a C function call

**\$sp**: stack pointer

**\$fp**: frame pointer (in the stack)

**\$gp**: global pointer to the *GOT* (*Global Offset Table*), a special data structure built by the compiler in the *.data* area for holding addresses and offsets to access global data structures, function tables and so on

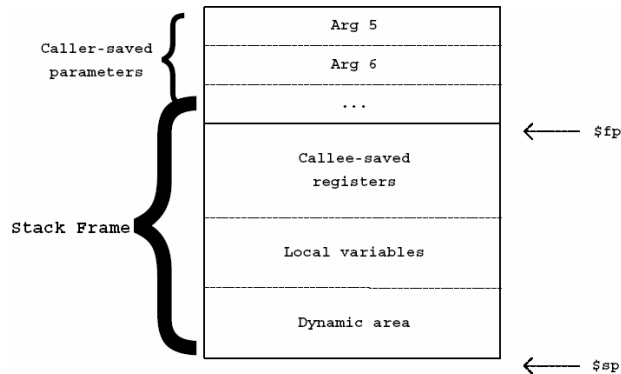
**\$k0-\$k1**: reserved to ROM handlers usage

© 2007 Mauro Morsiani

8

## uMPS software development

### uMPS stack frame layout:



© 2007 Mauro Morsiani

9

## uMPS software development

### Compilation is a complex process:

Four compilation stages:

- pre-process
- compile
- assemble
- link

Usually governed by `Makefile` and `make`

GNU gcc compiler final output: an ELF (*Executable and Linking Format*) executable file

ELF requires a OS to be loaded: too complex for uMPS kernel or bootstrap ROM

a different file format is required

© 2007 Mauro Morsiani

10

## uMPS software development

### Who converts ELF to uMPS file format(s)?

`umps-elf2umps` takes an ELF binary (executable or object file) in input and produces:

- \* `.rom.umps`: ROM files
- \* `.aout.umps`: user programs using VM
- \* `.core.umps`: the format used for kernel files
- \* `.stab.umps`: a *symbol table* (a text file describing the symbols' names and addresses in the executable) associated to the matching `.aout.umps` or `.core.umps` file

© 2007 Mauro Morsiani

11

## uMPS software development

### The uMPS `.aout` format:

File Offsets	
data area padded to multiple of 4kB (no bss included)	.data File Start Offset
text area padded to multiple of 4kB	0x00B0
	0x00A0
\$GP Start Value	0x00A0
data File Size	0x0024
data File Start Offset	0x0020
data Memory Size	0x001C
data Start Address	0x0018
text File Size	0x0014
text File Start Offset	0x0010
text Memory Size	0x000C
text Start Address	0x0008
Program Start Address	0x0004
uMPS .aout Magic Number	0x0000

© 2007 Mauro Morsiani

12

## uMPS software development

### uMPS .aout header explained:

Field Name	File Offset	Explanation
<code>.aout Magic File No.</code>	0x0000	Special identifier used for file type recognition.
<code>Program Start Addr.</code>	0x0004	Address (virtual) from which program execution should begin. Typically this is 0x8000.0000.
<code>.aout Start Addr.</code>	0x0008	Address (virtual) for the start of the <code>.aout</code> area. It is fixed to 0x8000.0000.
<code>.aout Memory Size</code>	0x000C	Size of the memory space occupied by the <code>.aout</code> section.
<code>.aout File Start Offset</code>	0x0010	Offset into <code>.aout</code> file where <code>.aout</code> begins. Since the header is part of <code>.aout</code> , this is always 0x0000.0000.
<code>.aout File Size</code>	0x0014	Size of <code>.aout</code> area in the <code>.aout</code> file. Larger than <code>.aout</code> Mem. Size since its padded to the nearest 4KB block boundary.
<code>.data Start Addr.</code>	0x0018	Address (virtual) for the start of the <code>.data</code> area. The <code>.data</code> area is placed immediately after the <code>.aout</code> area at the start of a 4KB block, i.e. <code>.aout Start Addr. + .aout File Size</code> .
<code>.data Memory Size</code>	0x001C	Size of the memory space occupied by the full <code>.data</code> area, including the <code>.bss</code> area.
<code>.data File Start Offset</code>	0x0020	Offset into the <code>.aout</code> file where <code>.data</code> begins. This should be the same as the <code>.aout File Size</code> .
<code>.data File Size</code>	0x0024	Size of <code>.data</code> area in the <code>.aout</code> file. Different from the <code>.data</code> Mem. Size since it doesn't include the <code>.bss</code> area but is padded to the nearest 4KB block boundary.
<code>SGP Start Value</code>	0x00A8	Starting value for <code>SGP</code> , computed during linking. It is usually loaded by <code>start.c</code> into <code>SGP</code> at program start time.
<code>.aout</code>	0x00B0	The program's <code>.aout</code> area.
<code>.data</code>	<code>.aout File Size</code>	The program's <code>.data</code> area.

© 2007 Mauro Morsiani

13

## uMPS software development

### uMPS file formats compared:

`.rom` file format is just bare code: no file headers, `.text`, `.data`...

`.aout` and `.core` file formats are the same

`.aout` program layout will use virtual memory addresses (0x8000.0000 and so on)

`.core` program layout will use physical memory addresses (0x2000.1000 and so on)

`.core .data` area is provided with a zero-filled `.bss`

© 2007 Mauro Morsiani

14

## uMPS software development

### C language development in uMPS:

To develop software in C language for the uMPS architecture, some C conventions must be satisfied:

Register usage (as shown)

Memory layout (as shown)

Who calls `main()`?

The linker expects a `__start()` function to exist

It is provided in:

`crtso.o` for kernel and programs running with VM off

`crti.o` for user programs running with VM on

But who calls `__start()`?

© 2007 Mauro Morsiani

15

## uMPS software development

### The ROM strikes once again:

The Bootstrap ROM is able to access the `.core` file in memory (or to load it from tape) and to start the kernel

The ROMs are also responsible for exception and TLB-refill handling

Is there something else the ROMs can perform to help us?

Yes: handling the start of new processes and the kernel program termination

© 2007 Mauro Morsiani

16

## uMPS software development

### ROM support routines:

accessible only in Kernel mode

using physical addresses

invoked by raising a BREAK exception

C usage	ROM Service/Instr.
<code>void LDST(state_t *statep)</code>	<b>LDST</b>
<code>void FORK(unsigned int entryhi, unsigned int status, unsigned int pc, state_t *statep)</code>	<b>FORK</b>
<code>void PANIC()</code>	<b>PANIC</b>
<code>void HALT()</code>	<b>HALT</b>

© 2007 Mauro Morsiani

17

## uMPS software development

### libumps: uMPS support library

libumps acts as a wrapper, allowing to:

access ROM routines

access special **CP0** 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

18

## uMPS software development

### libumps library kernel support functions:

`unsigned int SYSCALL(unsigned int number, unsigned int arg1, unsigned int arg2, unsigned int arg3)`: generates a SYSCALL exception

`unsigned int STST(state_t * statep)`: saves the current status of the processor

C usage	ROM Service/Instr.
<code>void LDST(state_t *statep)</code>	<b>LDST</b>
<code>void FORK(unsigned int entryhi, unsigned int status, unsigned int pc, state_t *statep)</code>	<b>FORK</b>
<code>void PANIC()</code>	<b>PANIC</b>
<code>void HALT()</code>	<b>HALT</b>

© 2007 Mauro Morsiani

19

## uMPS software development

### libumps library functions for CP0 register access:

C usage	CP0 Register
<code>unsigned int getINDEX()</code>	<b>Index</b>
<code>unsigned int getENTRYHI()</code>	<b>EntryHi</b>
<code>unsigned int getENTRYLO()</code>	<b>EntryLo</b>
<code>unsigned int getSTATUS()</code>	<b>Status</b>
<code>unsigned int getCAUSE()</code>	<b>Cause</b>
<code>unsigned int getRANDOM()</code>	<b>Random</b>
<code>unsigned int getEPC()</code>	<b>EPC</b>
<code>unsigned int getBADVADDR()</code>	<b>BadVAddr</b>

© 2007 Mauro Morsiani

20

## uMPS software development

### libumps library functions for CP0 register access (cont'd):

C usage	CP0 Register
<code>unsigned int setINDEX(unsigned int)</code>	<b>Index</b>
<code>unsigned int setENTRYHI(unsigned int)</code>	<b>EntryHi</b>
<code>unsigned int setENTRYLO(unsigned int)</code>	<b>EntryLo</b>
<code>unsigned int setSTATUS(unsigned int)</code>	<b>Status</b>
<code>unsigned int setCAUSE(unsigned int)</code>	<b>Cause</b>

© 2007 Mauro Morsiani

21

## uMPS software development

### libumps library functions for TLB-related instructions:

C usage	CP0 Instruction
<code>void TLBWR()</code>	TLB-Write-Random
<code>void TLBWI()</code>	TLB-Write-Index
<code>void TLBR()</code>	TLB-Read
<code>void TLBP()</code>	TLB-Probe
<code>void TLBCLR()</code>	TLB-Clear

© 2007 Mauro Morsiani

22