Laboratorio di Sistemi Operativi
Anno Accademico 2006-2007

Software Development with uMPS
Part 2

Mauro Morsiani

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

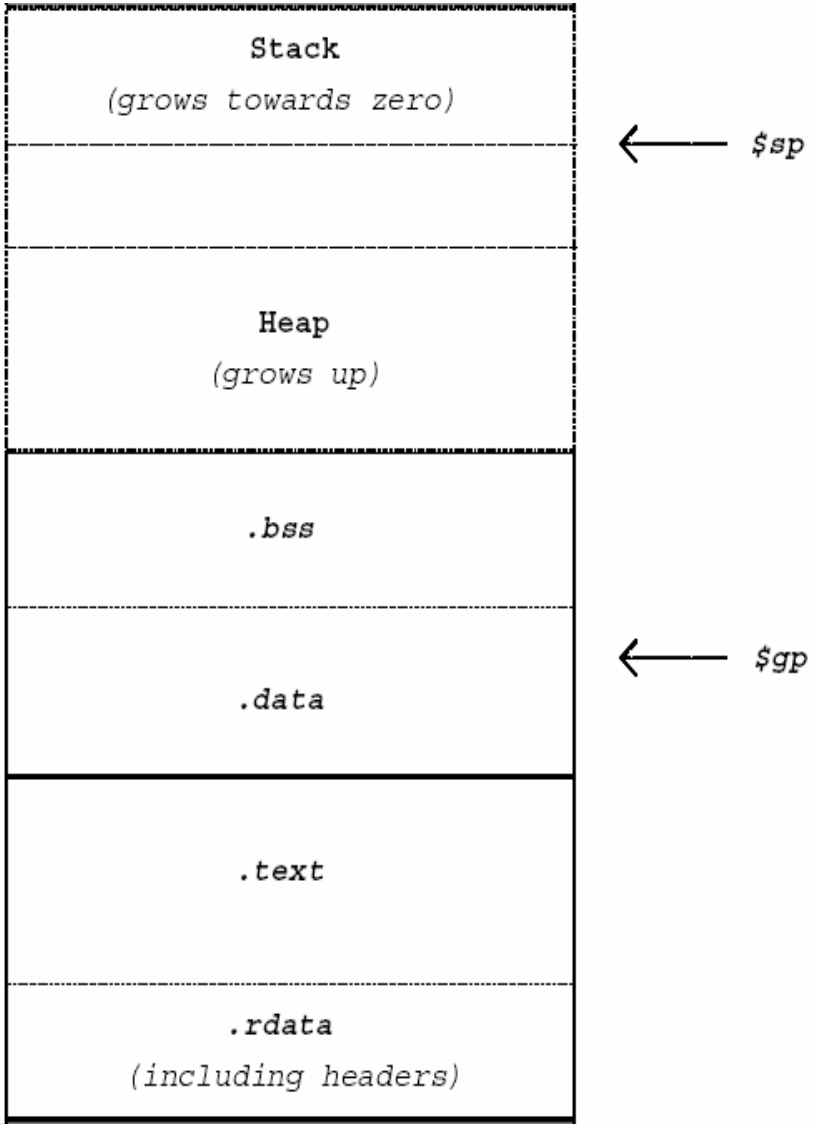
C language (and GNU gcc compiler) is the preferred alternative

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

uMPS software development

A generic program memory layout:



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

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

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

uMPS software development

uMPS register usage conventions:

Register Number	Register Name	Use
$0	*$0*	Hardwired to zero value.
$1	*$at*	Reserved for the assembler: used in computations and for holding temporary values.
$2-$3	*$v0-$v1*	Used for expression evaluations and for holding integer function results; may also be used to pass the static link during nested procedures call.
$4-$7	*$a0-$a3*	Used to pass the first four integer-compatible type actual arguments at procedure call; not preserved across procedure calls.
$8-$15, $24-$25	*$t0-$t9*	Temporary registers, used for expression evaluations: not preserved across procedure calls.
$16-$23	*$s0-$s7*	Saved registers; their values are preserved across procedure calls.
$26-$27	*$k0-$k1*	Reserved for kernel use.
$28	*$gp*	Global pointer.
$29	*$sp*	Stack pointer.
$30	*$s8 or $fp*	A saved register (like s0-s7), or used as frame pointer.
$31	*$ra*	Procedure call return address: also used for expression evaluation.

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

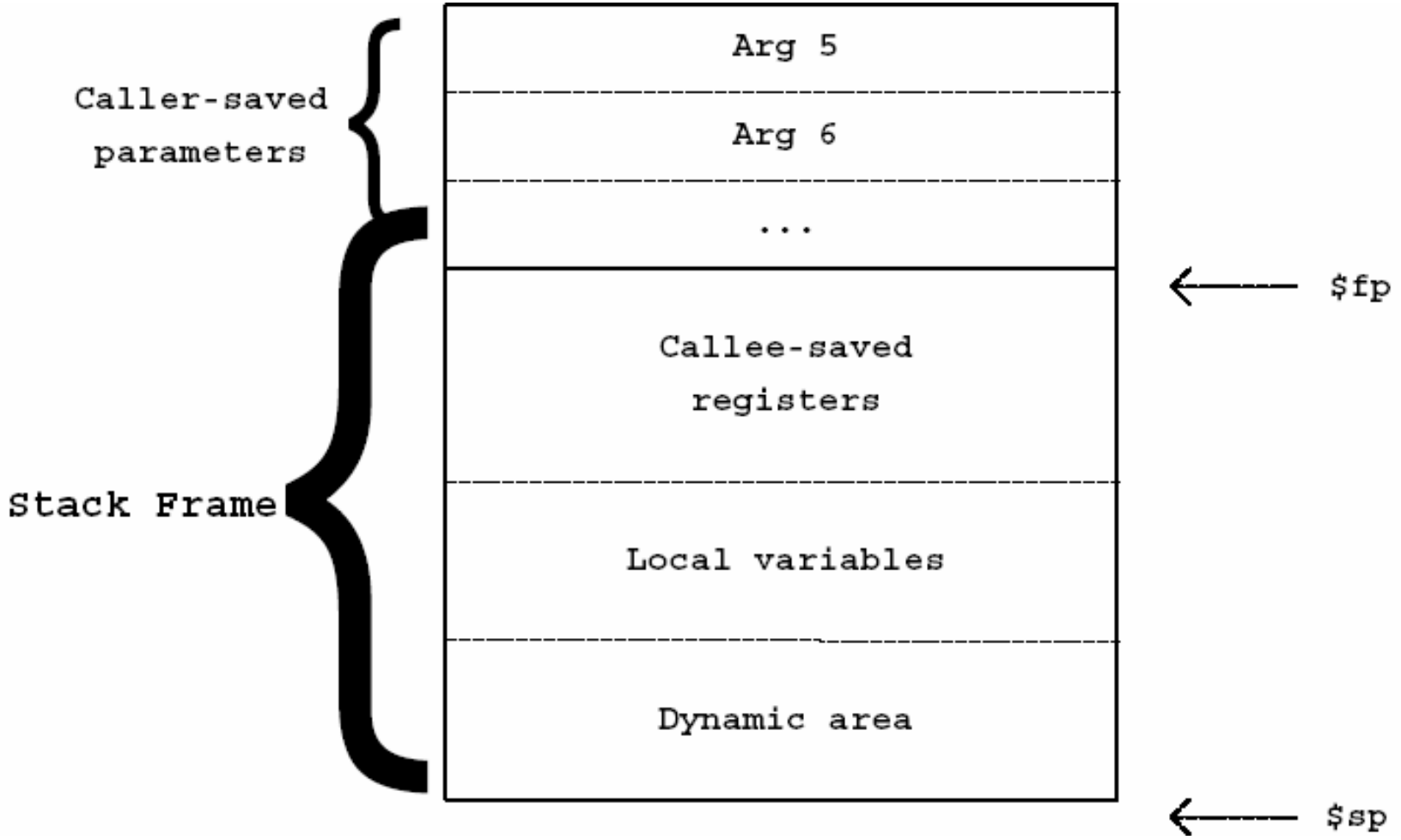
$fp: frame pointer (in the stack)

$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

uMPS software development

uMPS stack frame layout:



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

uMPS software development

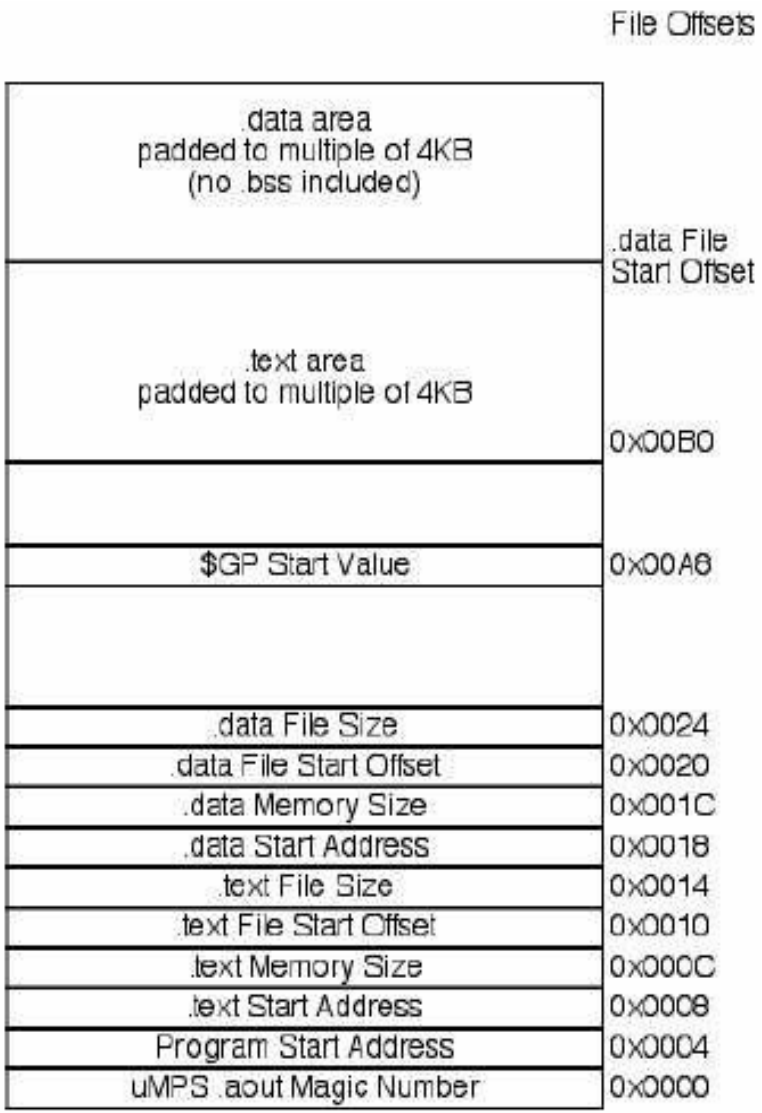
Who converts ELF to uMPS file format(s)?

`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

uMPS software development

The uMPS .aout format:



uMPS software development

uMPS .aout header explained:

Field Name	File Offset	Explanation
.aout Magic File No.	0x0000	Special identifier used for file type recognition.
Program Start Addr.	0x0004	Address (virtual) from which program execution should begin. Typically this is 0x8000.00B0
.text Start Addr.	0x0008	Address (virtual) for the start of the **.text** area. It is fixed to 0x8000.0000
.text Memory Size	0x000C	Size of the memory space occupied by the **.text** section.
.text File Start Offset	0x0010	Offset into .aout file where **.text** begins. Since the header is part of **.text**, this is always 0x0000.0000
.text File Size	0x0014	Size of **.text** area in the .aout file. Larger than **.text** Mem. Size since its padded to the nearest 4KB block boundary.
.data Start Addr.	0x0018	Address (virtual) for the start of the **.data** area. The **.data** area is placed immediately after the **.text** area at the start of a 4KB block, i.e. **.text** Start Addr. + **.text** File Size.
.data Memory Size	0x001C	Size of the memory space occupied by the full **.data** area, including the **.bss** area.
.data File Start Offset	0x0020	Offset into the .aout file where **.data** begins. This should be the same as the **.text** File Size.
.data File Size	0x0024	Size of **.data** area in the .aout file. Different from the **.data** Mem. Size since it doesn't include the **.bss** area but is padded to the nearest 4KB block boundary.
$GP Start Value	0x00A8	Starting value for **$GP**, computed during linking. It is usually loaded by `_start()` into **$GP** at program start time
.text	0x00B0	The program's **.text** area
.data	**.text** File Size	The program's **.data** area

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*

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()`?

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

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.
`void LDST(state_t *statep)`	**LDST**
`void FORK(unsigned int entryhi, unsigned int status, unsigned int pc, state_t *statep)`	**FORK**
`void PANIC()`	**PANIC**
`void HALT()`	**HALT**

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

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.
`void LDST(state_t *statep)`	**LDST**
`void FORK(unsigned int entryhi, unsigned int status, unsigned int pc, state_t *statep)`	**FORK**
`void PANIC()`	**PANIC**
`void HALT()`	**HALT**

uMPS software development

libumps library functions for CP0 register access:

C usage	CP0 Register
`unsigned int getINDEX()`	**Index**
`unsigned int getENTRYHI()`	**EntryHi**
`unsigned int getENTRYLO()`	**EntryLo**
`unsigned int getSTATUS()`	**Status**
`unsigned int getCAUSE()`	**Cause**
`unsigned int getRANDOM()`	**Random**
`unsigned int getEPC()`	**EPC**
`unsigned int getBADVADDR()`	**BadVAddr**

uMPS software development

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

C usage	CP0 Register
`unsigned int setINDEX(unsigned int)`	**Index**
`unsigned int setENTRYHI(unsigned int)`	**EntryHi**
`unsigned int setENTRYLO(unsigned int)`	**EntryLo**
`unsigned int setSTATUS(unsigned int)`	**Status**
`unsigned int setCAUSE(unsigned int)`	**Cause**

uMPS software development

libumps library functions for TLB-related instructions:

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