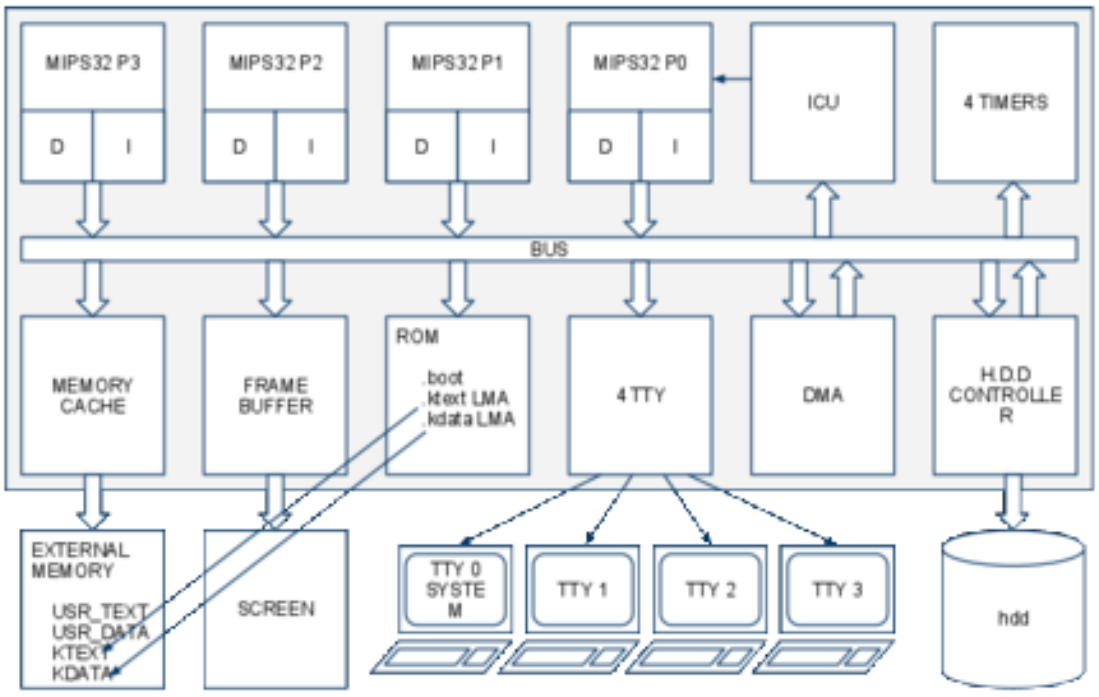
Boot Loader

Déplacement du code système vers la RAM

MI074 -3

Action du bootloader



Placement du code et des données

Le code et les données doivent être liés (ld) pour pouvoir être exécutés dans la ram, mais ils doivent être placés dans la rom par le loader.

On utilise le LMA (Load Memory Address)

```
#include "segmentation.h"
MEMORY
{
  boot   : ORIGIN = BOOT_BASE, LENGTH = BOOT_SIZE
  ktext  : ORIGIN = KTEXT_BASE, LENGTH = KTEXT_LMA_SIZE
  kdata  : ORIGIN = KDATA_BASE, LENGTH = KDATA_LMA_SIZE
}
SECTIONS
{
 .boot : { *(.boot) *(.boot.*) } > boot
 .ktext : AT ( KTEXT_LMA_BASE ) { *(.kentry) *(.text) __ktext_end = .; } > ktext
 .kdata : AT ( KDATA_LMA_BASE ) { *(.data*) *(.rodata*) *(.bss*) *(COMMON*) *(.scommon*)
                                  __kdata_end = .; } > kdata
}
```

Le Bootloader (synchro)

Un processeur et un seul (le CPU0) doit déplacer le système.
Les autres processeurs "attendent" qu'une variable globale change.
La variable globale est déclarée dans un .c

- Quelle est l'adresse de cette variable ?

```
.extern boot_signal   `
la      $26,    boot_signal
sw      $0,     ($26)
bne     $16,    $0, boot_wait
la      $26,    __boot_loader
jalr    $26
```

- Réponse:
  c'est une variable de la RAM qui est initialisée à 0
  et qui vaudra le nombre de processeur à la fin du __boot_loader

code de boot complet

```
#include <segmentation.h>
#include <config.h>

#define STACK_SIZE \
        CONFIG_BOOT_STACK_SIZE

.section .boot ,"ax",@progbits
.extern boot_signal
.extern __boot_loader
.extern __do_init

.ent    boot
.align  2

boot:
   li      $26,   0
   mtc0    $26,   $12              # Status Register
   mfc0    $16,   $0               # CPU_ID
   la      $27,   RAM_BASE+RAM_SIZE   # top
   li      $26,   (STACK_SIZE)
   mult    $16,   $26
   mflo    $29
   subu    $29,   $29,   $27
   addiu   $29,   $29,   -1*4
   # for __do_init/__boot_loader  argument
   la      $26,   boot_signal
   sw      $0,    ($26)
   # goto to boot_wait if procid != 0
   bne     $16,   $0, boot_wait
   la      $26,   __boot_loader
   jalr    $26

call_do_init:
   or      $4,    $0,    $16
   la      $26,   __do_init
   jr      $26

boot_wait:
   lw      $27,   0($26)
   sub     $5,    $27,   $16
   bgez    $5,    call_do_init
   j       boot_wait

.end boot
```

Fonction __boot_loader (__attribute__)

On doit placer cette fonction dans la section .boot :

il suffit de la déclarer avant sa définition avec un attribut section:

void __boot_loader(void) **__attribute__ ((section(".boot")));**

Les __attribute__ s'utilisent pour les fonctions ou les variables:

http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Variable-Attributes.html#Variable Attributes
http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Variable-Attributes.html#Function Attributes

Les autres (extrait):

- aligned(*b*) : variable alignée sur une frontière de *b* bytes
- paked : variable "tassée" pour occuper le moins de place
- noinline : fonction à ne jamais inliner
- noreturn : fonction dont on ne sort jamais (sauf exit)

Fonction __boot_loader (var ldscript)

On veut copier une partie de la rom vers la mémoire.

- On connait les adresses de début, mais les adresses de fin dépendent du remplissage.
- Les adresses de fin sont __ktext_end et __kdata_end

Pour récupérer ces adresses dans le code C:

- extern unsigned __ktext_end; // on déclare le symbole
- (unsigned) &__ktext_end // valeur typée du symbole

Fonction __boot_loader (le code)

```
#include <segmentation.h>
#include <config.h>

extern unsigned boot_signal;

void __boot_loader(void) __attribute__ ((section(".boot")));

void __boot_loader()
{
    unsigned i;
    unsigned size;
    extern unsigned __ktext_end;
    extern unsigned __kdata_end;

    size = ((unsigned) &__ktext_end - KTEXT_BASE) >> 2;
    for (i = 0; i < size; i++)
        *((unsigned *) KTEXT_BASE + i) = *((unsigned *) KTEXT_LMA_BASE + i);

    size = ((unsigned) &__kdata_end - KDATA_BASE) >> 2;
    for (i = 0; i < size; i++)
        *((unsigned *) KDATA_BASE + i) = *((unsigned *) KDATA_LMA_BASE + i);

    boot_signal = CONFIG_CPU_NR;
}
```

Insertion de code assembleur

On peut inclure du code assembleur dans du code C:

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

```
asm volatile (
  assembler template
  : output operands              /* optional */
  : input operands               /* optional */
  : list of clobbered registers /* optional */
);
```

- assembler template:
  une chaine de caractères, les instructions sont séparées par \n.
- output et input operands:
  permet d'associer les variables et les registres
- registers:
  permet d'informer les compilo des registres modifiés

insertion de code assembleur

Dans le code assembleur, les registres remplacés par des variables sont notés par %0, %1, etc et ils seront associés aux variables suivant leur index.

```
static inline bool_t cpu_spinlock_trylock(uint_t * lock)
{
  register bool_t state = 1;  // preload 1 as return value (hyp lock is busy)
  asm volatile (
          "lw     $2, (%1)    \n"   // load lock value in $2
          "bnez  $2, 20       \n"    // forgive whenever lock is busy
          "ll       $2, (%1)    \n"   // load lock value in $2 and try to register in memory
          "bnez  $2, 12       \n"    // forgive whenever lock is busy
          "sc     %0, (%1)    \n"    // sc returns in %0 <- 1 if free, 0 if busy
          "xori    %0, %0,  1 \n"    // trylock returns 0 if free, 1 if busy
          : "=r" (state)
          : "r" (lock)
          : "$2");
  return state;
}
```

affichage de la date de démarrage

```
void __do_init(unsigned cpu_id)
{
   unsigned register val;
   asm volatile ("mfc0 %0, $9\n" : "=r" (val));

   tty_puts(cpu_id, "Hello World at ");
   tty_putx(cpu_id, val);
   tty_puts(cpu_id, " !\n");
   while (1);
}
```