



Exécution d'une application simple

Visite guidée commentée

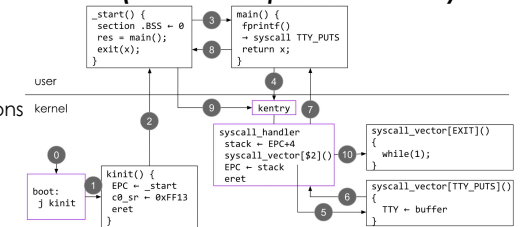
```
#include <libc.h>
int main(void) {
    fprintf(0, "[%d] app is alive\n", clock());
    return 0;
}
```

franck.wajsburt@lip6.fr

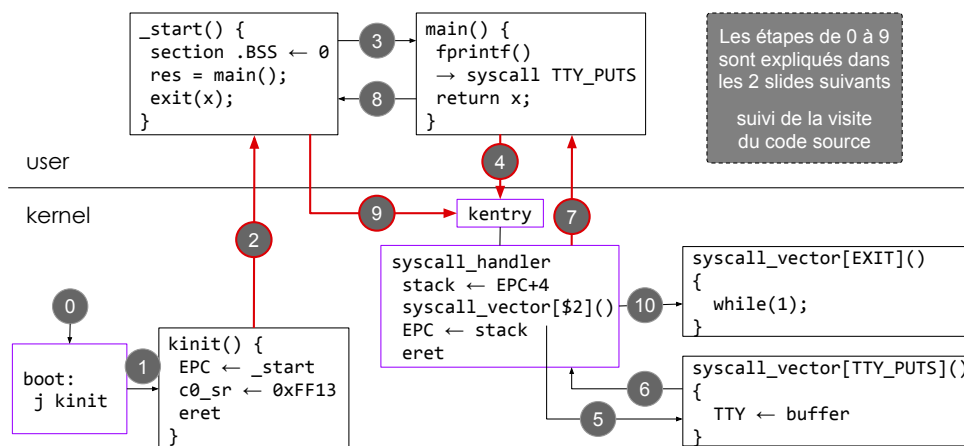
27 nov 2023

Un parcours de boot à exit (les étapes 0 à 4)

- Après l'activation du signal reset, le MIPS saute à l'adresse de boot `0xBFC00000`, le MIPS est en mode kernel, les interruptions sont masquées (le bit `c0_sr.ERL` est à 1).
- Le code de boot se contente d'initialiser le pointeur de pile en haut de la section `.kdata` puis il appelle la fonction `kinit()`
- Démarrage de l'application avec la fonction `_start()`, cette fonction prépare la mémoire utilisateur en initialisant les variables globales non initialisées (elles sont dans la section `.BSS`).
- Appel de la fonction `main()`, c'est la fonction principale de l'application (elle devrait recevoir des arguments de la ligne de commande, ici il n'y en a pas). La fonction `main()` peut demander l'arrêt de l'application par l'appel à la fonction `exit()` ou juste sortir par `return x`, et laisser `_start()` faire l'appel à `exit()`
- L'exécution de `fprintf()` définie dans la libc provoque l'exécution d'une instruction `syscall` qui déroute l'exécution de l'application vers l'adresse `kentry`, le point d'entrée unique du noyau (hormis `kinit()`).

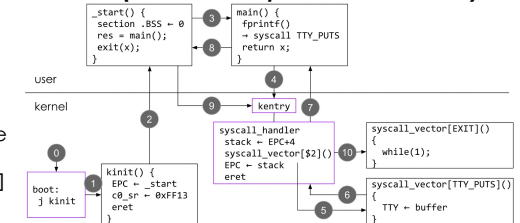


Un parcours de boot à exit (en 10 étapes)

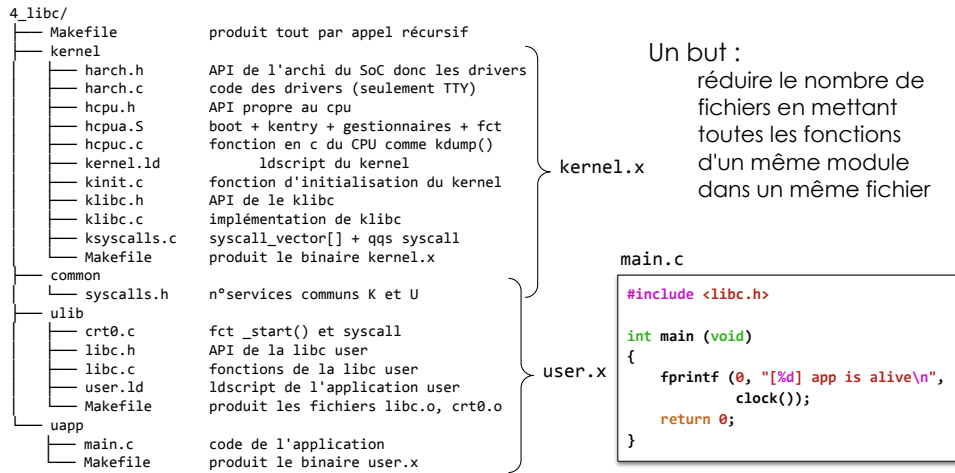


Un parcours de boot à exit (les étapes 5 à 10)

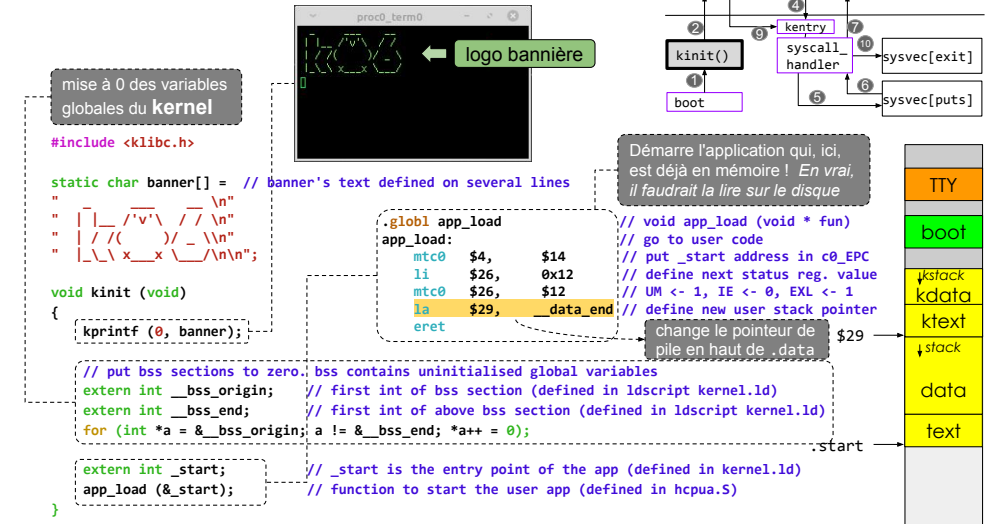
- `kentry` a décodé le registre de cause et fait appel au gestionnaire de `syscall` (`syscall_handler`) qui sauvegarde dans la pile les valeurs de registres lui permettant de revenir de l'appel système (dont `EPC+4`) et elle appelle la fonction présente dans la table `syscall_vector[]` à la case du n° de service
- La fonction `syscall_vector[SYSCALL_TTY_PUTS]()` envoie les octets du buffer dans le registre `WRITE` du TTY
- Au retour de la fonction précédente, on revient dans le gestionnaire de `syscall` qui rétablit la valeur des registres sauvegardés dans la pile et qui prépare le registre `EPC` pour l'exécution de l'instruction `eret` qui revient dans la fonction `main()`
- L'exécution de `return` permet de sortir de la fonction `main()` pour revenir dans la fonction `_start()`. L'application est terminée, il faut appeler `exit()`
- La fonction `exit()` exécute l'instruction `syscall` qui saute dans `kentry` comme à l'étape 4.
- Comme à l'étape 6, le gestionnaire de `syscall` appelle cette fois la fonction `syscall_vector[SYSCALL_EXIT]()` qui, ici, se contente d'arrêter l'exécution.



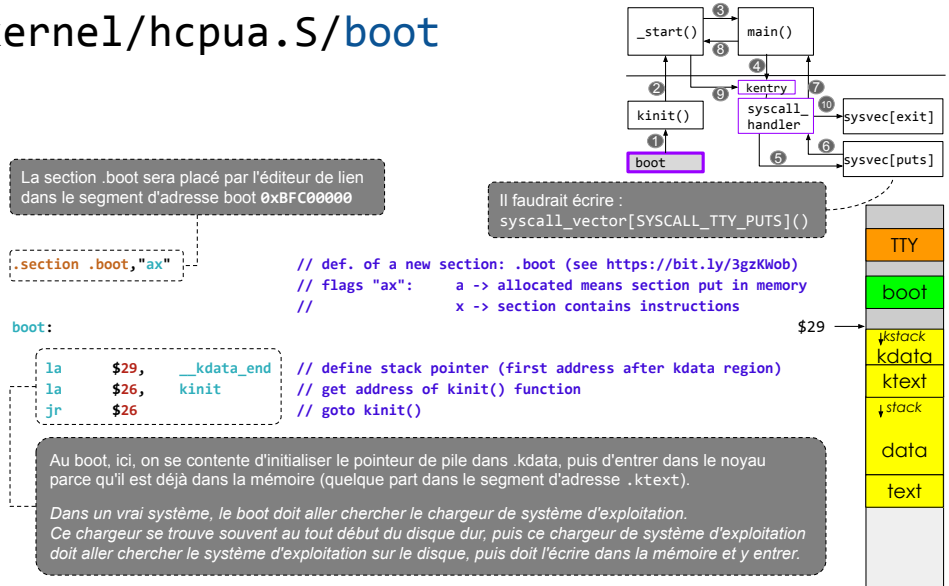
Ensemble des fichiers de cette étape



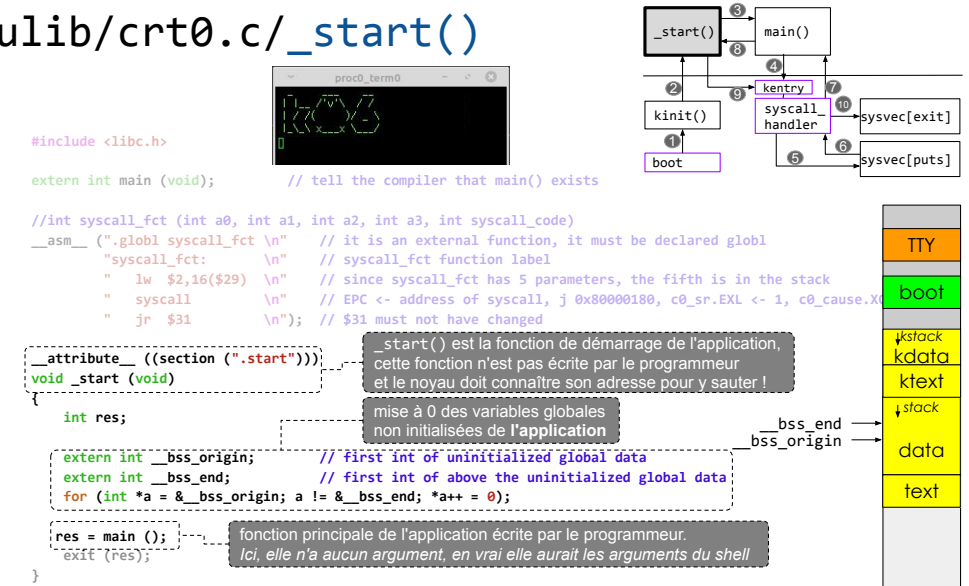
```
kernel/kinit.c/kinit()
```



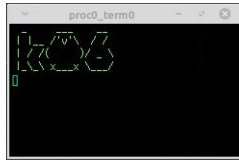
kernel/hcpua.S/boot



- ulib/crt0.c/_start()



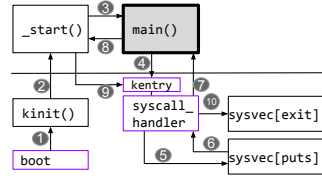
uapp/main.c/main()



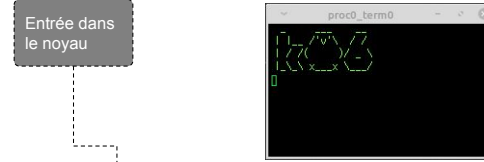
```
#include <libc.h>

int main (void)
{
    fprintf (0, "[%d] app is alive\n", clock());
    return 0;
}
```

Ici, on veut juste afficher un message avec la date (en cycles).
Ce sera quand même 2 appels système : clock() et fprintf()
Le code de ces fonctions est dans la libc, d'où l'inclure au début



kernel/hcpua.S/kentry



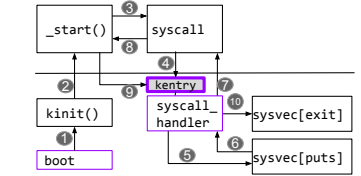
```
kentry:
    mfc0 $26, $13          // read CR (Cause Register)
    andi $26, $26, 0x3C    // apply cause mask (keep bits 2 to 5)
    li $27, 0x20           // 0x20 is the syscall code
    bne $26, $27, kpanic   // if not then that is not a syscall
```

syscall_handler:
[...]

L'entrée dans le noyau, il faut analyser le champ XCODE
du registre de cause c0_cause (\$13 du coprocesseur 0)
et, ici, si ce n'est pas le code de syscall, c'est la panique !

Le code du gestionnaire de syscall est juste après ...

Notez que le noyau peut utiliser les registres \$26 et \$27
sans les sauver avant car ces registres lui sont réservés.
Ce sont des registres temporaires pour le noyau, gcc ne
les utilise jamais, il ne contiennent rien pour l'application



ulib/libc.c/clock()



```
#include <syscalls.h> // kernel services

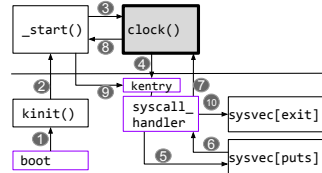
unsigned clock (void)
{
    return syscall_fct (0, 0, 0, 0, SYSCALL_CLOCK);
}
```

Le but de la fonction clock() est de préparer les arguments et
d'appeler la fonction syscall_fct() qui entre dans le noyau.

ici le code produit par gcc (un peu ré-ordonné
pour la lisibilité et avec des commentaires)
la fonction syscall_fct() est forcément écrite
en assembleur mais elle ne fait pas grand-chose

```
<clock>:
    27bdf0e0    addiu    sp,sp,-32      # nr=1 nv=0 na=5 mais 32?
    afbf001c    sw       ra,28(sp)      # sauve ra = $31
    24020005    li       v0,$2          # v0 = $2
    af200019    sw       v0,16(sp)      # 5e arg dans la pile
    00003825    move     a3,zero        # a3 = $7
    00003825    move     a2,zero        # a2 = $6
    00002825    move     a1,zero        # a1 = $5
    00002025    move     a0,zero        # a0 = $4
    0f000277    jal      7f4009dc <syscall_fct> # appel de la fct syscall
    8fbf001c    lw       ra,28(sp)      # restore $31
    27b00020    addiu    sp,sp,32      # restore le pt de pile
    03e00008    jr       ra             # sort
```

```
<syscall_fct>:
    8fa20010    lw       v0,16(sp)      # récupère $2 depuis la pile
    0000000c    syscall              # $4 à $7 inchangé -> syscall
    03e00008    jr       ra             # sort avec val de retour $2
```



kernel/hcpua.S/syscall_handler

Gestionnaire
de syscalls
→ début

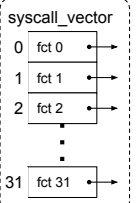
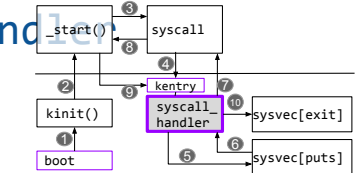
- alloue de la place dans la pile pour \$31, EPC, SR et les 5 arguments des services : \$4 à \$7 + \$2 (n° du service)
- trouve l'adresse du service dans syscall_vector[\$2]
- change le mode d'exécution du MIPS → UM=0 IE=0
- appelle le service avec un jal syscall_vector[\$2]

syscall_handler:

```
addiu $29, $29, -8*4      // context for $31 + EPC + SR + syscall_code + 4 args
mfc0 $27, $14             // $27 <- EPC (addr of syscall instruction)
mfc0 $26, $12             // $26 <- SR (status register)
addiu $27, $27, 4         // $27 <- EPC+4 (return address)
sw $31, 7*4($29)          // save $31 because it will be erased
sw $27, 6*4($29)          // save EPC+4 (return address of syscall)
sw $26, 5*4($29)          // save SR (status register)
sw $2, 4*4($29)           // save syscall code (useful for debug message)
mtc0 $0, $12              // SR <- kernel-mode w/o INT (UM=0 ERL=0 EXL=0 IE=0)

la $26, syscall_vector    // $26 <- table of syscall functions
andi $2, $2, SYSCALL_NR-1 // apply syscall mask
sll $2, $2, 2             // compute syscall index (multiply by 4)
addu $2, $26, $2          // $2 <- & syscall_vector[$2]
lw $2, ($2)               // $2 <- syscall_vector[$2]
jalr $2                   // call syscall function
```

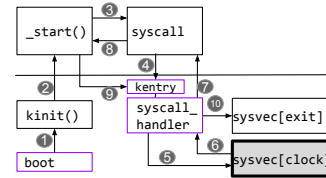
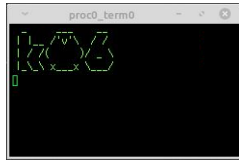
initialisation du vecteur de syscalls = tableau de pointeur de fonctions
ayant au plus 5 arguments : \$4 à \$7 et le 5° dans la pile (n°service)



```
void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = syscall_unknown,
    [SYSCALL_EXIT] = exit,
    [SYSCALL_TTY_GETS] = tty_gets,
    [SYSCALL_TTY_PUTS] = tty_puts,
    [SYSCALL_CLOCK] = clock,
};
```

kernel/hcpua.S/clock()

Gestionnaire de syscalls
→ service



```
.globl clock
clock:
    mfc0    $2, $9
    jr      $31
```

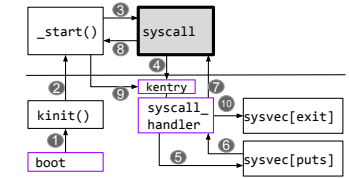
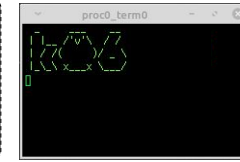
Le service clock() se contente de récupérer dans \$2 la valeur du compteur de cycles présents par défaut dans le registre \$9 du coprocesseur 0 du MIPS et sort !

clock() est nécessairement écrite en assembleur. .globl permet de la rendre visible hors de hcpua.o

ulib/crt0.c/syscall_fct

À la sortie du noyau, on revient dans la fonction syscall_fct() vue slide 40

\$2 contient la valeur de retour, il suffit de sortir pour revenir dans la fonction appelante ici on retourne dans clock()



```
#include <libc.h>
```

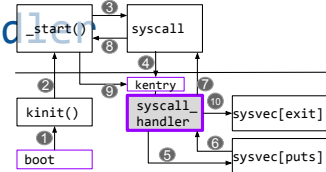
syscall_fct() est forcément écrite en assembleur. Notez qu'on peut inclure du code assembleur directement dans un code C. Cette fonction est ici la seule fonction écrite en assembleur que nous aurons, c'est pourquoi, nous avons choisi de ne pas créer un fichier .S spécifique pour syscall_fct() et la mettre dans le fichier crt0.c qui contient aussi la fonction _start(). Le fichier crt0.c contient en fait la fonction d'entrée dans l'application _start() et la fonction de sortie syscall_fct()

Syntaxe: `__asm__("une string avec des \n entre instructions, ici, décomposée comme pour banner")`

```
//int syscall_fct(int a0, int a1, int a2, int a3, int syscall_code)
__asm__(
    ".globl syscall_fct\n" // it is an external function, it must be declared globl
    "syscall_fct:\n" // syscall_fct function label
    "lw $2,16($29)\n" // since syscall_fct has 5 parameters, the fifth is in the stack
    "syscall\n" // EPC -> address of syscall, j 0x80000180, c0_sr.EXL <- 1 c0_cause.XCODE <- 8
    "jr $31\n"); // $31 must not have changed
```

kernel/hcpua.S/syscall_handler

Gestionnaire de syscalls
→ fin

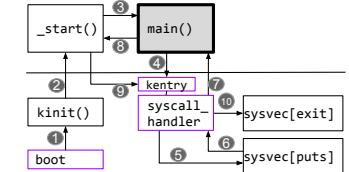
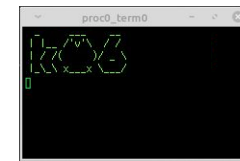


```
jalr $2 // call service function

lw $26, 5*4($29) // get old SR
lw $27, 6*4($29) // get return address of syscall
lw $31, 7*4($29) // restore $31 (return address of user syscall function)
mtc0 $26, $12 // restore SR
mtc0 $27, $14 // restore EPC
addiu $29, $29, 8*4 // restore stack pointer
eret // return : jr EPC with c0_sr.EXL <- 0
```

- La valeur de retour du service est dans \$2
- on restore \$31, EPC, SR
- on restore le pointeur de pile
- On sort avec eret ≈ jr EPC et effacer le bit c0_sr.EXL

uapp/main.c/main()

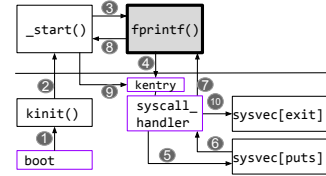
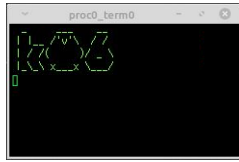


```
#include <libc.h>
```

```
int main(void)
{
    fprintf(0, "[%d] app is alive\n", clock());
    return 0;
}
```

- Au retour de clock() qui ne contenait que l'appel à la fonction syscall_fct().
- On revient dans main() et on va entrer dans fprintf() définie dans libc.c

ulib/libc.c/fprintf()



```
int fprintf(int tty, char *fmt, ...)
{
    int res;
    char buffer[PRINTF_MAX];
    va_list ap;
    va_start(ap, fmt);
    res = vsnprintf(buffer, sizeof(buffer), fmt, ap);
    va_end(ap);

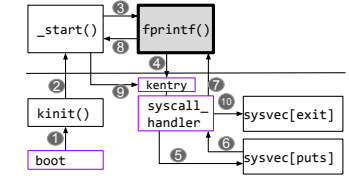
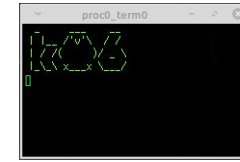
    res = syscall(tty, (int)buffer, res, 0, SYSCALL_TTY_PUTS);
    return res;
}
```

fprintf() est une **fonction variadique** * (avec un nombre variable d'arguments)

- Dans la déclaration, on met ... à la fin
- On déclare une variable `ap` de type `va_list` (type builtin défini par gcc)
- `va_start()` permet de dire à gcc quel est le dernier argument explicite et donc où commence la liste des arguments variables `ap`
- On pourrait directement utiliser `ap`, mais ici, on choisit de le passer en paramètre de la fonction `vsnprintf()` dont le dernier argument est de type `va_list`
- à la fin il faut fermer la liste `ap` avec `va_end()`

L'accès aux arguments de la liste `ap` est réalisé avec `va_arg()`, ici, dans `vsnprintf()`

ulib/libc.c/fprintf()



```
int fprintf(int tty, char *fmt, ...)
{
    int res;
    char buffer[PRINTF_MAX];
    va_list ap;
    va_start(ap, fmt);
    res = vsnprintf(buffer, sizeof(buffer), fmt, ap);
    va_end(ap);

    res = syscall_fct(tty, (int)buffer, res, 0, SYSCALL_WRITE);
    return res;
}
```

à la fin il faut fermer la liste `ap` avec `va_end()`

Au retour de `vsnprintf()`, le tableau `buffer[]` contient les caractères à envoyer au terminal n°`tty`, `res` contient le nombre de caractères dans la `buffer[]`

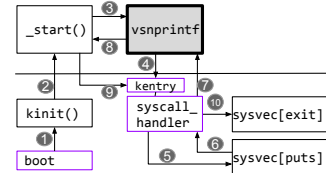
Pour écrire ces caractères, il faut demander au noyau par un appel système `WRITE` :

- `$4` : n°`tty`
- `$5` : adresse du `buffer[]`
- `$6` : nombre de caractères à afficher

En résumé :

Variadique : type `va_list` `ap` ; ouverture `va_start(ap, arg)` puis `N` lectures `va_arg(ap, type)` et fermeture `va_end(ap)`

ulib/libc.c/vsnprintf()



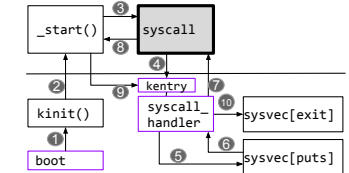
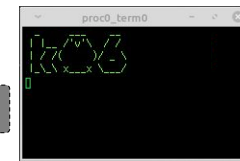
```
static int vsnprintf(char *buffer, unsigned size, char *fmt, va_list ap)
{
    char arg[16]; // buffer used to build the argument
    char *tmp; // temporary pointer used to build arguments
    [...]
    while (*fmt) { // for all char in fmt
        [...]
        switch (*fmt) { // study the different cases
            [...]
            case 's': // case %s (string)
                tmp = va_arg(ap, char *); // tmp points to this string argument
                tmp = (tmp) ? tmp : "(null)"; // replace "" by "(null)"
                goto copy_tmp; // go to copy tmp in buffer
            [...]
        }
    }
    abort();
    *buf = '\0'; // put the ending char 0
    res = (int)((unsigned)buf - (unsigned)buffer); // compute the nb of char to write
    return res;
}
```

Dans ce slide, il n'y a pas le code de `vsnprintf()`, mais vous pouvez lire le code de la libc si cela vous intéresse :-)

Le principe consiste à parcourir la chaîne `fmt` et à chaque `%` on regarde le caractère suivant pour connaître le type de l'argument de la liste `ap` dont il faut prendre la valeur : `s`:char*, `d`:int, `x`:int, etc.

`va_arg(ap, type)` permet de prendre la variable suivante dans la liste `ap` et type informe gcc du type de cet argument.

ulib/crt0.c/syscall_fct()



Nouvel appel de `syscall_fct()`

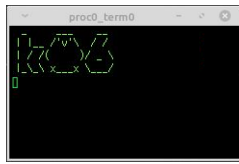
```
#include <libc.h>

extern int __bss_origin; // first int of uninitialized global data
extern int __bss_end; // first int of above the uninitialized global data
extern int main(void); // tell the compiler that main() exists

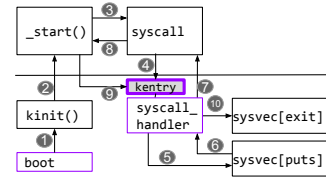
//int syscall_fct(int a0, int a1, int a2, int a3, int syscall_code)
asm ("__globl syscall_fct\n" // it is an external function, it must be declared globl
     "syscall_fct:\n" // syscall_fct function label
     "lw $2,16($29)\n" // since syscall has 5 parameters, the fifth is in the stack
     "syscall\n" // EPC <- address of syscall, j 0x80000180, c0_s.EXL <- 1, c0_cause.XCODE <- 8
     "jr $31\n" // $31 must not have changed
```

kernel/hcpua.S/kentry

Nouvelle entrée dans kentry



```
kentry:
    mfc0 $26, $13          // read CR (Cause Register)
    andi $26, $26, 0x3C    // apply cause mask (keep bits 2 to 5)
    li $27, 0x20           // 0x20 is the syscall code
    bne $26, $27, kpanic   // if not then that is not a syscall
```



kernel/harch.c/tty_write()

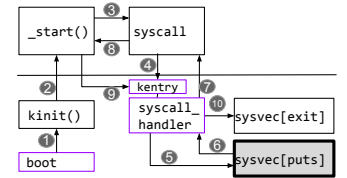
Description de la composition des registres du TTY par une struct C

```
struct tty_s {
    int write;           // tty's output address
    int status;          // tty's status address something to read if not null)
    int read;            // tty's input address
    int unused;          // unused address
};
```

extern volatile struct tty_s __tty_regs_map[NTTYS]; // NTTYS is a #define

```
int tty_write(int tty, char *buf, int count)
{
    int res = count;
    tty = tty % NTTYS; // to be sure that tty is an existing tty
    while ((count != 0) && (*buf != 0)) {
        __tty_regs_map[tty].write = *buf;
        count--;
        buf++;
    }
    return res - count;
}
```

L'accès à un registre du TTY est simplifié par cette déclaration de structure C

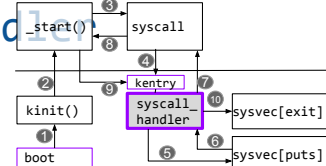


__tty_regs_map est un label défini dans le fichier **ldscript** **kernel.ld**. ce label est égal à l'adresse du premier registre du TTY dans l'espace d'adressage du MIPS (pour ce SoC). Cette déclaration permet de dire au compilateur que cette adresse est de type **struct tty_s** et qu'elle est allouée ailleurs (**extern**). Il y a autant de jeu de registres qu'il y a de TTY, d'où le tableau de struct

kernel/hcpua.S/syscall_handler

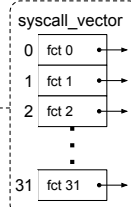
Gestionnaire de syscalls → début

Cette fois l'appel système est SYSCALL_TTY_PUTS



```
syscall_handler:
    addiu $29, $29, -8*4    // context for $31 + EPC + SR + syscall_code + 4 args
    mfc0 $27, $14           // $27 <- EPC (addr of syscall instruction)
    mfc0 $26, $12           // $26 <- SR (status register)
    addiu $27, $27, 4       // $27 <- EPC+4 (return address)
    sw $31, 7*4($29)        // save $31 because it will be erased
    sw $27, 6*4($29)        // save EPC+4 (return address of syscall)
    sw $26, 5*4($29)        // save SR (status register)
    sw $2, 4*4($29)         // save syscall code (useful for debug message)
    mtc0 $0, $12            // SR <- kernel-mode w/o INT (UM=0 ERL=0 EXL=0 IE=0)

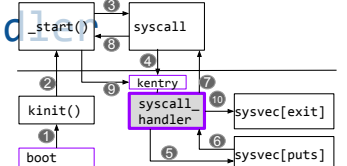
    la $26, syscall_vector // $26 <- table of syscall functions
    andi $2, $2, SYSCALL_NR-1 // apply syscall mask
    sll $2, $2, 2           // compute syscall index (multiply by 4)
    addu $2, $26, $2        // $2 <- & syscall_vector[$2]
    lw $2, ($2)             // $2 <- syscall_vector[$2]
    jalr $2                 // call service function
```



```
void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = syscall,
    [SYSCALL_EXIT] = exit,
    [SYSCALL_READ] = tty_read,
    [SYSCALL_WRITE] = tty_write,
    [SYSCALL_CLOCK] = clock,
};
```

kernel/hcpua.S/syscall_handler

Gestionnaire de syscalls → fin

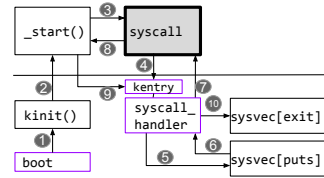


```
jalr $2 // call service function

lw $26, 5*4($29) // get old SR
lw $27, 6*4($29) // get return address of syscall
lw $31, 7*4($29) // restore $31 (return address of syscall function)
mtc0 $26, $12 // restore SR
mtc0 $27, $14 // restore EPC
addiu $29, $29, 8*4 // restore stack pointer
eret // return : jr EPC with c0_sr.EXL <- 0
```


ulib/crt0.c/syscall_fct()

sortie de la fonction
syscall_fct()



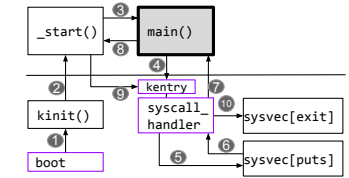
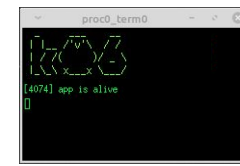
```
#include <libc.h>

extern int __bss_origin; // first int of uninitialized global data
extern int __bss_end; // first int of above the uninitialized global data
extern int main (void); // tell the compiler that main() exists

//int syscall_fct (int a0, int a1, int a2, int a3, int syscall_code)
__asm__ (".globl syscall_fct \n" // it is an external function, it must be declared globl
        "syscall_fct: \n" // syscall_fct function label
        " lw $2,16($29) \n" // since syscall has 5 parameters, the fifth is in the stack
        " syscall \n" // EPC <- address of syscall, j 0x80000180, c0_sr.EXL <- 1, c0_cause.XCODE <- 8
        " jr $31 \n"); // $31 must not have changed
```

uapp/main.c/main()

sortie de la fonction
main()

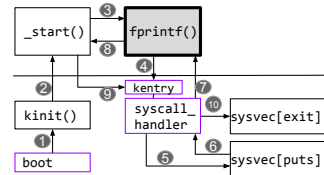


```
#include <libc.h>

int main (void)
{
    fprintf (0, "[%d] app is alive\n", clock());
    return 0;
}
```

ulib/libc.c/fprintf()

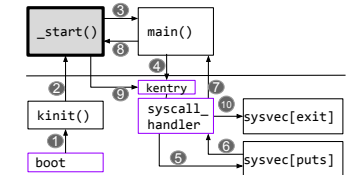
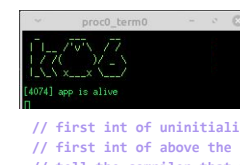
sortie de la fonction
fprintf()



```
int fprintf(int tty, char *fmt, ...)
{
    int res;
    char buffer[PRINTF_MAX];
    va_list ap;
    va_start (ap, fmt);
    res = vsnprintf(buffer, sizeof(buffer), fmt, ap);
    va_end(ap);
    res = syscall_fct( tty, (int)buffer, res, 0, SYSCALL_WRITE);
    return res;
}
```

ulib/crt0.c/_start()

retour dans la fonction
start()



```
#include <libc.h>

extern int __bss_origin; // first int of uninitialized global data
extern int __bss_end; // first int of above the uninitialized global data
extern int main (void); // tell the compiler that main() exists

//int syscall_fct (int a0, int a1, int a2, int a3, int syscall_code)
__asm__ (".globl syscall_fct \n" // it is an external function, it must be declared globl
        "syscall_fct: \n" // syscall_fct function label
        " lw $2,16($29) \n" // since syscall has 5 parameters, the fifth is in the stack
        " syscall \n" // EPC <- address of syscall, j 0x80000180, SR.EXL <- 1, c0_cause.XCODE <- 8
        " jr $31 \n"); // $31 must not have changed

__attribute__ ((section (".start")))
void _start (void)
{
    int res;

    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);
    res = main ();
    exit (res);
}
```

Puisque la fonction main() n'a pas fait appel à exit() et que exit() est la seule manière de sortir d'un programme utilisateur, alors c'est la fonction start() qui doit le faire avec la valeur rendue par main(). On ne revient jamais de exit() !

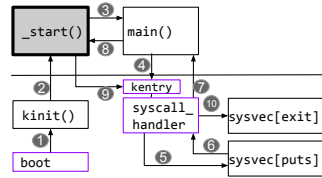
ulib/libc.c/exit()

dernier appel système avec la valeur de retour de l'application



```
void exit (int status)
{
    syscall_fct( status, 0, 0, 0, SYSCALL_EXIT);
}
```

// never returns



kernel/hcpua.S/syscall_handler

Gestionnaire de syscalls → début

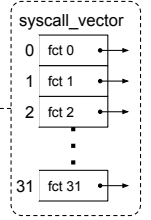
Cette fois l'appel système est SYSCALL_EXIT



syscall_handler:

```
addiu $29, $29, -8*4 // context for $31 + EPC + SR + syscall_code + 4 args
mfc0 $27, $14 // $27 <- EPC (addr of syscall instruction)
mfc0 $26, $12 // $26 <- SR (status register)
addiu $27, $27, 4 // $27 <- EPC+4 (return address)
sw $31, 7*4($29) // save $31 because it will be erased
sw $27, 6*4($29) // save EPC+4 (return address of syscall)
sw $26, 5*4($29) // save SR (status register)
sw $2, 4*4($29) // save syscall code (useful for debug message)
mtc0 $0, $12 // SR <- kernel-mode w/o INT (UM=0 ERL=0 EXL=0 IE=0)

la $26, syscall_vector // $26 <- table of syscall functions
andi $2, $2, SYSCALL_NR-1 // apply syscall mask
sll $2, $2, 2 // compute syscall index (multiply by 4)
addu $2, $26, $2 // $2 <- & syscall_vector[$2]
lw $2, ($2) // $2 <- syscall_vector[$2]
jalr $2 // call syscall function
```

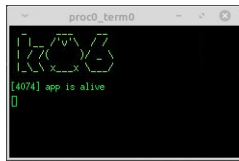


```
void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = unknown_syscall,
    [SYSCALL_EXIT] = exit,
    [SYSCALL_READ] = tty_read,
    [SYSCALL_WRITE] = tty_write,
    [SYSCALL_CLOCK] = clock,
};
```

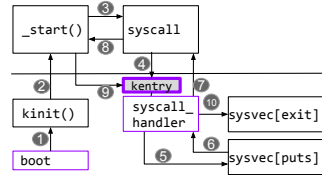
et donc : jal syscall_vector[SYSCALL_EXIT] = exit()

kernel/hcpu.S/kentry

Dernière entrée dans kentry



```
kentry: // kernel entry
mfc0 $26, $13 // read CR (Cause Register)
andi $26, $26, 0x3C // apply cause mask (keep bits 2 to 5)
li $27, 0x20 // 0x20 is the syscall code
bne $26, $27, kpanic // if not then that is not a syscall
```



kernel/klibc.c/exit()

On affiche un message, on est dans le noyau : kprintf() et clock() sont appelées directement sans syscall



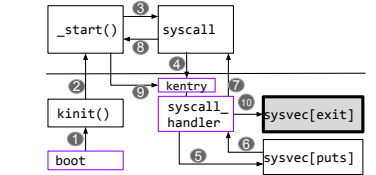
```
void exit (int status)
{
    kprintf (0, "\n\n[4d] EXIT status = %d\n", clock(), status);
    while (1); // infinite loop
}
```

Là c'est la fin ! on fige le simulateur !!!

Le simulateur propose un mode debug pour de suivre la trace d'exécution détaillée des instructions trace0.s et la séquence d'appel des fonctions label0.s le 0 c'est le n° du processeur, si on en a plusieurs, il y a trace1.s, label1.s, etc.

> make debug
> less label0.s

label0.s



K 12:	<boot>	../kernel/hcpua.S
K 37:	<kinit>	../kernel/kinit.c
K 61:	<kprintf>	../kernel/klibc.c
K 118:	<vsnprintf>	../kernel/klibc.c
K 1359:	<tty_write>	../kernel/harch.c
K 2671:	<app_load>	../kernel/hcpua.S
U 2695:	<start>	../ulib/crt0.c
U 2131:	<main>	../uapp/main.c
U 2154:	<clock>	../ulib/libc.c
U 2193:	<syscall>	../ulib/crt0.c
K 2213:	<kentry>	../kernel/hcpua.S
K 2227:	<syscall_handler>	../kernel/hcpua.S
K 2288:	<clock>	../kernel/hcpua.S
U 2353:	<fprintf>	../ulib/libc.c
U 2487:	<vsnprintf>	../ulib/libc.c
K 3146:	<syscall>	../ulib/crt0.c
K 3157:	<kentry>	../kernel/hcpua.S
K 3162:	<syscall_handler>	../kernel/hcpua.S
K 3206:	<tty_write>	../kernel/harch.c
U 3424:	<exit>	../ulib/libc.c
U 3440:	<syscall>	../ulib/crt0.c
K 3442:	<kentry>	../kernel/hcpua.S
K 3447:	<syscall_handler>	../kernel/hcpua.S
K 3477:	<exit>	../kernel/klibc.c
K 3504:	<clock>	../kernel/hcpua.S
K 3538:	<kprintf>	../kernel/klibc.c
K 3565:	<vsnprintf>	../kernel/klibc.c
K 4224:	<tty_write>	../kernel/harch.c

mode date label file