Introduction

Qu'est ce qu'une sémaphore

Les sémaphores servent à éviter :

- les conditions de course (race conditions) et dépendances
- les corruptions de donnée
- les comportements non déterministes

Introduction d'un mécanisme de synchronisation.

Exécution concurrente de plusieurs tâches partageant des ressources communes et introduction de section critique.

Qu’est ce qu’une futex

Futex -> Fast Userspace Mutex

–> Pourquoi c’est mieux que les mutex ?

- La futex limite le nombre d’appels systèmes
- Un recours au kernel uniquement pour récupérer la page ou en cas de blocage.
- Amélioration de performance : moins de context-switch.
- Mémoire partagée.

Contexte

Code basé sur le repository fourni par M. Poquet et disponible sur GitHub.

V1 : Implémentation des mutex

Architecture fortement inspirée de l'implémentation XV6 des sémaphores

V2 : Implémentation des futex

Architecture inspirée du code d'exemple d'usage des opérations de mémoire exclusives de ARMv8-a

Malheureusement, code non testé sur RaspberryPi :(

Kernel vs User spaces

Contexte

Pour implémenter les appels kernels, nous devons ajouter plusieurs syscalls !

- **Sémaphores**
  - créer une sémaphore (sem_new)
  - détruire une sémaphore (sem_delete)
  - prendre un jeton (sem_p)
  - rendre un jeton (sem_v)
- **Futex**
  - récupérer la page des sémaphores (get_fut_page)
  - bloquer la tâche (fut_block)

Kernel Side :

sys.c

```
unsigned long sys_sem_new(unsigned int
count){
   sem_new(count);
}
void sys_sem_p(semaphore sem){
   sem_p(sem);
}
void sys_sem_v(semaphore sem){
   sem_v(sem);
}
void sys_sem_delete(semaphore sem){
   sem_delete(sem);
}
```

```
void *const sys_call_table[] = {
  sys_write,
  sys_fork,
  sys_exit,
  sys_sem_new,
  sys_sem_delete,
  sys_sem_p,
  sys_sem_v
};
```

sys.h

```
#define __NR_syscalls  9
```

User Side : user_sys.S


```
.set SYS_SEM_NEW_NUMBER, 3
.set SYS_SEM_DELETE_NUMBER, 4
.set SYS_SEM_P_NUMBER, 5
.set SYS_SEM_V_NUMBER, 6

.globl call_sys_sem_new
call_sys_sem_new:
   mov w8, #SYS_SEM_NEW_NUMBER
   svc #0
   ret

.globl call_sys_sem_delete
call_sys_sem_delete:
   mov w8, #SYS_SEM_DELETE_NUMBER
   svc #0
   ret
```

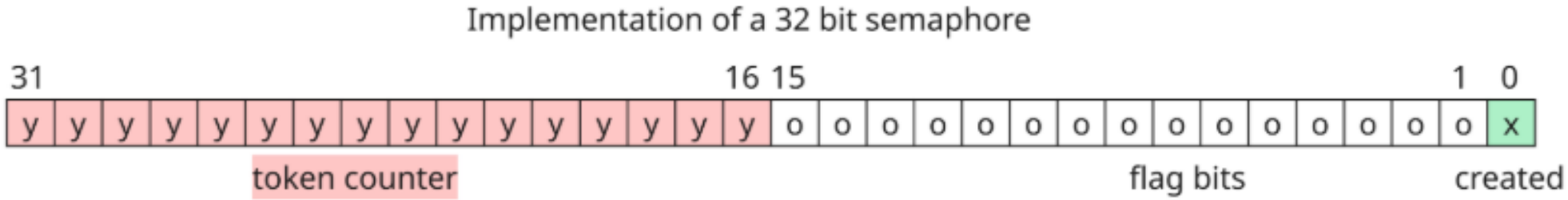
```
.globl call_sys_sem_P
call_sys_sem_P:
   mov w8, #SYS_SEM_P_NUMBER
   svc #0
   ret

.globl call_sys_sem_V
call_sys_sem_V:
   mov w8, #SYS_SEM_V_NUMBER
   svc #0
   ret
```

User Side : use_sys.h


```c
unsigned long call_sys_sem_new(unsigned int count);
void call_sys_sem_delete(unsigned long sem);
void call_sys_sem_P(unsigned long sem);
void call_sys_sem_V(unsigned long sem);
```

Sémaphores

Spécification



Une page mémoire dédiée au stockage des sémaphores

Initialisation


```c
// represents the content of the semaphore number s
#define SEM(s) * (unsigned long*) ( sem_page + s*32)

unsigned long sem_page;

void sem_table_init() {
    sem_page = allocate_kernel_page(); // already zeroed
}
```

Création et destruction de sémaphore


```
unsigned long sem_new(unsigned int count) {
    disable_irq();
    unsigned long sem = 0;
    while ((SEM(sem)) % 2) {
        sem++;
    }
    SEM(sem) = count << 16 | 1 ;
    enable_irq();
    return sem;
}

void sem_delete(unsigned long sem) {
    disable_irq();
    SEM(sem) = 0;
    enable_irq();
}
```

Modifications du scheduler


```
for (int i = 0; i < NR_TASKS; i++){
      p = task[i];
      if (p && p->state == TASK_RUNNING && p->counter > c) {
        c = p->counter;
        next = i;
             } else if (p && p->state == TASK_BLOCKED_SEM && sem_count(p-
>blocked_by) != 0) {
                sem_p(p->blocked_by);
                p->state = TASK_RUNNING;
            }
    }
```

Sémaphores

```
void sem_p(unsigned long sem) {
        disable_irq();
    if (SEM(sem) >> 16) {
        SEM(sem) -= 1 << 16;
        enable_irq();
    }
    else {
        current->state = TASK_BLOCKED_SEM;
        current->blocked_by = sem;
        enable_irq();
        schedule();
    }
}
```

Sémaphores

```
void sem_v(unsigned long sem) {
    disable_irq();
    SEM(sem) += 1 << 16;
    enable_irq();
}
```

Futaphores

Similarités

Initialisation

```
unsigned long fut_page;
void fut_table_init() {
    fut_page = allocate_kernel_page();
    FUT(fut_page, 0) = (1 << 16) | 1;
}
```

Mémoire Partagée

```
 unsigned long get_fut_page(){
   unsigned long virt_addr = current->mm.user_pages[current-
>mm.user_pages_count-1].virt_addr + PAGE_SIZE;
   map_page(current, virt_addr, fut_page - VA_START);
   return virt_addr;
 }
```

Similarités

Modification du scheduler

```
for (int i = 0; i < NR_TASKS; i++){
  /*...*/
  else if (p && p->state == TASK_BLOCKED_FUT &&
           fut_count(fut_page,p->blocked_by) != 0) {
      int blocked = fut_pasm(p->blocked_by, fut_page);
  if (!blocked) p->state = TASK_RUNNING;
  }
}
```

Atomicité

CAS <Xs>, <Xt>, [<Xn>]

Compare and Swap reads 64-bits from memory, and compares it against Ws. If the comparison is equal, Xt is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

Atomicité

CAS <Xs>, <Xt>, [<Xn>]

Compare and Swap reads 64-bits from memory, and compares it against Ws. If the comparison is equal, Xt is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

Seulement sur ARMv8.1 :(

Désillusion et Exclusivité

LDXR <Xd>, [<Xn>]

Loads a 32/64-bit value from memory address [Xn] into register Xd, and marks the memory location as exclusively accessed by the core.

STXR <Ws>, <Xt>, [Xn]

Attempts to exclusively store a 32/64-bit value from register Xt to memory address [Xn].

Returns success/failure in Ws.

```
.globl fut_vasm
fut_vasm :
  mov x3,#32
  mul x0,x0,x3
    add x1,x0,x1
fut_v_try :
    LDXR x0,[x1]
    ADD x0, x0, 0b1000000000000000     # 1 shift 16
    STXR w2,x0,[x1]
    CMP w2,#0
    BNE fut_v_try
    ret
```

```
.globl fut_pasm
fut_pasm :
  mov x3,#32
  mul x0,x0,x3
    add x1,x0,x1
fut_p_try :
    LDXR x0,[x1]
  mov x2,x0,lsr #16
    CMP x2,#0
    BNE fut_p_true
    MOV x0, #1  // blocking
    RET
```

```
fut_p_true :
    SUB x0, x0, 0b1000000000000000
    STXR w2,x0,[x1]
    CMP w2,#0
    BNE fut_p_try
    mov x0,#0  // not blocking
    RET
```

Tests

tests statiques

Tâches non préemptives, scheduling manuel via le syscall yield()

Cette étape permet de vérifier le fonctionnement des fonctions des blocages, retraits et dépôts de jeton.

tests statiques (simplifié)


```
    unsigned long semaphore =
call_sys_sem_new(2);
    int pid = call_sys_fork();
    if (pid == 0) {
        call_sys_write("...");
        call_sys_sem_P(semaphore);
        call_sys_write("...");
        user_delay(100000000);
        call_sys_write("...");
        call_sys_sem_V(semaphore);
        call_sys_write("...");
        yield();
    } else {
        call_sys_write("...");
        call_sys_sem_P(semaphore);
        call_sys_write("...");
        yield();
        call_sys_write("...");
        call_sys_sem_V(semaphore);
        call_sys_write("...");
        yield();
    }
}
```

tests dynamiques

Tâches préemptives : les tâches peuvent se faire interrompre à tout moment.

- beaucoup de delays ajoutés dans les processus.
- les interruptions sont modifiées pour appeler plus souvent le scheduler.

Cette étape permet de vérifier le réel échange du token des mutex/futex entre les différentes tâches.

Le but est de **simuler un système le moins déterministe possible**.

tests dynamiques (simplifié)


```
    unsigned long page =
call_sys_fut_get_page();
    unsigned long futex = fut_new(page,
2);
    int pid = call_sys_fork();
    if (pid == 0) {
        page = call_sys_fut_get_page();
    }
    for (int i = 0; i < 3; i++) {
        switch (pid) {
        case 0:
            call_sys_write("...");
            fut_p(page, futex);
            call_sys_write("...");
            user_delay(1000000000);
            call_sys_write("...");
            fut_v(page, futex);
            call_sys_write("...");
            user_delay(1000000000);
            break;
        default:
            call_sys_write("...");
            fut_p(page, futex);
            call_sys_write("...");
            user_delay(1000000000);
            call_sys_write("...");
            fut_v(page, futex);
            call_sys_write("...");
            user_delay(1000000000);
            break;
        };
    }
}
```