Eljily Mohamed
m3eljily@enib.fr

Novembre 2024

1 Introduction

Ce document fournit un guide étape par étape pour comprendre et vérifier chaque composant principal dans la construction progressive de notre système d'exploitation. Le projet est conçu de manière à ce que le fichier source `main.c` soit divisé en plusieurs sections, chacune encapsulée dans des macros `#ifdef ... #endif`. Chaque section comprend un code d'exemple qui teste une fonctionnalité spécifique de l'OS en cours de développement.

Pour tester une fonctionnalité particulière, nous définissons le label correspondant avec `#define` en haut de `main.c`. Ce label correspond au test souhaité et déterminera quel segment de code est inclus lors de la compilation. Cette approche permet aux développeurs de se concentrer sur une fonctionnalité à la fois, permettant ainsi des tests modulaires et un débogage tout au long du développement de l'OS.

2 Aperçu des Tests

Ci-dessous se trouve un aperçu des différents labels disponibles dans `main.c`, chacun représentant un cas de test différent :

- `MAIN_TEST` : Teste le fonctionnement d'un appel système.
- `MAIN_EX1` : Teste la configuration de la première tâche et le changement de tâche.
- `MAIN_EX2` : Similaire à `MAIN_EX1`, mais avec toutes les tâches exécutant le même code.
- `MAIN_EX3` : Teste la fonctionnalité des sémaphores.
- `MAIN_EX4` : Montre l'utilisation d'un sémaphore comme un mutex.
- `MAIN_EX5` : Teste la fonction de temporisation.
- `MAIN_EX6` : Similaire à `MAIN_EX5`, mais avec deux tâches temporisées.
- `MAIN_EX7` : Teste la terminaison d'une tâche.
- `MAIN_EX8` : Teste l'interface de fichiers virtuels.
- `MAIN_EX9` : Teste l'utilisation de la LED RGB comme dispositif.
- `MAIN_EX10` : Teste l'utilisation du bouton USR comme dispositif avec gestion des interruptions pour les pressions de bouton.

Chacun de ces labels nous permet de vérifier un aspect spécifique de la fonctionnalité du système d'exploitation, garantissant des tests robustes et un développement progressif. Cette documentation sert de guide pour comprendre chaque test, sa configuration, son objectif et les résultats attendus alors que nous construisons le système d'exploitation composant par composant.

3 TEST_MAIN

3.1 Questions et Réponses

3.1.1 a. Changements observés après l'exécution de l'instruction `svc`

Décrivez les changements observés dans les valeurs des registres *msp*, *lr* et *ipsr* après l'exécution de l'instruction `svc`.Expliquez la signification de ces changements dans le contexte des appels système.

Réponse :

Avant l'exécution de l'instruction `svc`, les valeurs par défaut des registres étaient :
- *ipsr* : `no fault` : mode utilisateur

Signification : Cela indique qu'il n'y a pas d'exceptions en cours.
- *msp* : `0x2003ffd0`

Signification : Cela pointe vers le sommet de la pile principale, où les données de la fonction en cours sont stockées.
- *lr* : `0x2eb <main+14>`

Signification : Ce registre contient l'adresse de retour après l'exécution de `main`, indiquant où le programme doit reprendre après un appel de fonction.

Après l'exécution de l'instruction `svc`, les valeurs par défaut des registres sont devenues :
- *ipsr* : `faults 0x11` : mode système

Signification : La valeur `0x11` indique qu'une exception a été déclenchée, signalant le passage en mode d'exécution des appels système.
- *msp* : `0x2003ffb0`

Signification : La modification de cette valeur indique que la pile a été ajustée pour gérer des données liées à l'appel système.
- *lr* : `0xfffffff9`

Signification : Cette valeur indique que le programme doit revenir à une adresse spécifique après l'exécution de l'appel système, signalant une transition dans le mode d'exécution.

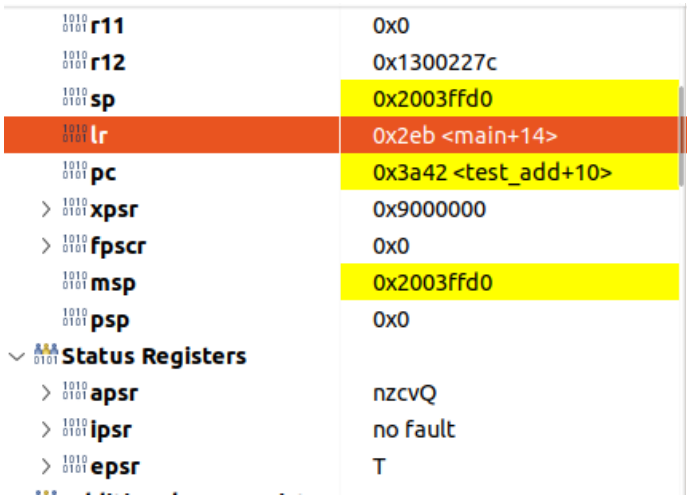


Figure 1: Avant l'exécution svc

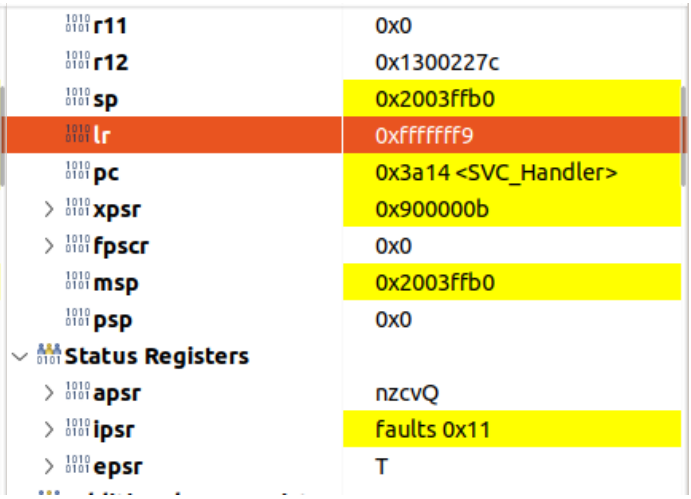


Figure 2: Après l'exécution de svc

3.1.2 b. Complétion des appels système

Compléter les appels système dans le fichier source `oslib.c`. Compléter la fonction `svc_dispatch()` dans le fichier `kernel/kernel.c` pour que les fonctions d'implémentation côté noyau soient correctement appelées. Les noms des fonctions qui implémentent un appel système côté noyau sont définis par `sys_appel_systeme`. Par exemple, l'implémentation de l'appel système `os_start` est la fonction `sys_os_start`. On ne demande pas de réaliser l'implémentation des appels système (pour le moment . . .).

Les appels systèmes considérés sont :

Appel Système	Numéro	Commentaire
os_alloc	1	Implémentation noyau : `malloc`
os_free	2	Implémentation noyau : `free`
os_start	3	
task_new	4	
task_id	5	
task_wait	6	
task_kill	7	
sem_new	8	
sem_p	9	
sem_v	10	

Table 1: Appels Systèmes et Implémentations

oslib.c :

Nous allons compléter les fonctions qui existent dans `oslib.c`,bibliothèque d'appels systèmes.
Réponse :

Je vais donner un exemple pour deux fonctions : une avec retour et une sans retour. Nous appliquerons la même logique pour les autres fonctions. Pour chaque appel système, nous envoyons `svc` suivi du numéro de l'appel système, et nous stockons le résultat de la fonction dans `r0` (s'il s'agit d'une fonction avec retour).

1. **Fonction avec retour :** `os_alloc`

```
void* os_alloc(unsigned int req)
{
    void* val;
    // Appel système pour allocation
    __ASM volatile ("svc 1\n\tmov %0, r0" : "=r" (val));
    return val; // Retourne le pointeur alloué
}
```

2. **Fonction sans retour :** `os_free`

```
void os_free(void *ptr)
{   // Appel système pour libération de mémoire
    __ASM volatile ("svc 2\n");
}
```

Voir le reste du code qui existe dans le fichier `oslib.c`.

Fonction Dispatcher dans le code système kernel

La fonction `svc_dispatch` est responsable du dispatching des appels système. Elle prend en entrée un numéro d'appel système `n` et un tableau `args[]` contenant les paramètres associés (jusqu'à 4). La fonction utilise un `switch` pour déterminer quelle fonction système appeler en fonction du numéro d'appel.

Le code de la fonction est le suivant :

```
int32_t svc_dispatch(uint32_t n, uint32_t args[])
{
    int32_t result = -1;
    void *buffer;
    switch(n) {
        case 0:  // Test function
            result = sys_add((int)args[0], (int)args[1]);
            break;
```

```
        case 1:  // os_alloc (malloc)
            buffer = malloc((size_t)args[0]);
            result = (buffer == NULL) ? -1 : (int32_t)buffer;
            break;
        case 2:  // os_free
            free((void *)args[0]);
            result = 0;
            break;
        case 3:  // os_start
            result = sys_os_start();
            break;
            dok les cas tanyin hnaya chouv code .....
        default:
            result = -1;
            break;
    }
    return result;
}
```

Dans ce code, chaque cas du `switch` correspond à un appel système différent. Le résultat de l'appel système est stocké dans la variable `result`.

4 TEST_EX1

Une tâche est une entité de code qui peut s'exécuter indépendamment du reste du programme. Elle est caractérisée par le code qui est exécuté (la fonction passée en paramètre à l'appel système `task_new`) et un contexte processeur (valeur des registres).

Pour garder une trace des informations concernant une tâche, on lui associe, côté noyau, un descripteur de tâche (Task Control Block ou TCB) défini dans le fichier `kernel/kernel.h` par :

```
typedef struct _Task {
    struct _Task *next;
    struct _Task *prev;
    uint32_t id;
    TaskState state;
    uint32_t *stack;
    uint32_t *sp;
    int32 priority;
} Task;
```

Regarde le document du prof, il a bien expliqué ça et le reste aussi.

4.0.1 Création de tâches

- Allouer dynamiquement un descripteur pour la tâche créée ainsi que la pile (les deux sont alloués en une seule fois).
- Initialiser les champs appropriés du descripteur de tâche.
- Insérer le descripteur dans la liste des tâches prêtes, dont le premier nœud est pointé par la variable globale `tsk_running`.

Exemple : Liste des Tâches après Création

Lorsque trois tâches sont créées, la structure des tâches est comme suit :

`tsk_running`
`next` → `next` → `next`
`descript_tâche_1` → `descript_tâche_2` → `descript_tâche_3`

Retour de la Fonction

La fonction renvoie :

- l'identifiant de tâche (`id`) si l'allocation a réussi,
- -1 si la structure n'a pas pu être allouée.

Initialisation de la Tâche

- Utiliser l'allocateur mémoire intégré à la libc associée au compilateur, `malloc`.
- Initialiser le champ `id` en utilisant une variable globale de manière à ce que chaque tâche ait un identifiant unique.
- Initialiser le champ `delay` à 0.

Chaînage des Blocs

Le chaînage des blocs est effectué en utilisant les fonctions de manipulation de listes circulaires fournies dans `kernel/list.[ch]`. On utilise les pointeurs :

- `prev` : pointeur vers la tâche précédente,
- `next` : pointeur vers la tâche suivante.

Allocation et Positionnement de la Pile

- Le champ `splim` doit pointer en bas du bloc mémoire alloué pour la pile.
- Le pointeur de pile (`sp`) doit pointer en haut du bloc alloué pour la pile.

L'empilement d'une valeur est effectué par :

- décrémentation de 4 octets du pointeur de pile,
- écriture de la valeur à empiler (4 octets) à l'adresse pointée.

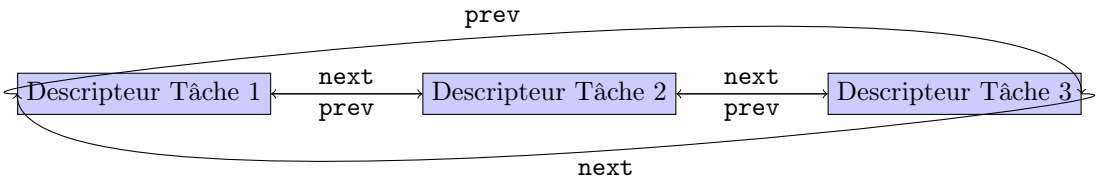
Création de la Tâche avec un Contexte Minimal

Les tâches sont créées avec l'état `READY` et doivent disposer d'un contexte minimal sur la pile. Cela inclut :

- `xPSR` : valeur initiale du registre d'état (bit T).
- `PC` : adresse de la fonction à exécuter par la tâche.
- `CONTROL` : la tâche s'exécute en mode `Thread unprivileged`.
- `EXC_RETURN` : code de retour de l'exception, permettant à la tâche de retourner en mode `Thread unprivileged` et d'utiliser le pointeur de pile `psp`.

Exemple : Création de 3 tâches et gestion des pointeurs `prev` et `next`

Lorsque 3 tâches sont créées, le chaînage des descripteurs dans la liste des tâches ressemble à ceci :



Dans cet exemple, chaque tâche est reliée aux autres via les pointeurs `prev` et `next`, formant une liste circulaire doublement chaînée.

Cadre de pile pour chaque tâche

Le cadre de pile (`Stack Frame`) est organisé comme suit :

Adresse	Éléments de la pile
+0x00	`xPSR`
+0x04	`PC` (adresse de la fonction de la tâche)
+0x08	`LR`
+0x0C	`R12`
+0x10	`R3`
+0x14	`R2`
+0x18	`R1`
+0x1C	`R0`
+0x20	`R11`
+0x24	`R10`
+0x28	`R9`
+0x2C	`R8`
+0x30	`R7`
+0x34	`R6`
+0x38	`R5`
+0x3C	`R4`
+0x40	`EXC_RETURN`
+0x44	`CONTROL`

Explication du code `sys_task_new`

La fonction `sys_task_new` :

- Alloue dynamiquement un descripteur de tâche et une pile avec une taille multiple de 8 octets.
- Initialise les champs de la tâche : identifiant (`id`), état initial (`TASK_READY`), et délai (`delay` à 0).
- Configure le cadre de pile pour la sauvegarde du contexte de la tâche, en plaçant les registres essentiels dans la pile :
  - `xPSR` : configuration du bit d'état,
  - `PC` : adresse de la fonction de la tâche,
  - `CONTROL` et `EXC_RETURN` : pour le mode `Thread unprivileged`.
- Insère la nouvelle tâche dans la liste des tâches prêtes via `tsk_running`.

5 Réentrance

5.1 Qu'est-ce que la réentrance ?

La réentrance désigne la capacité d'une fonction à être interrompue dans son exécution, puis appelée à nouveau avant la fin de son exécution précédente, sans entraîner de comportement indéfini. Une fonction est réentrante si elle satisfait les conditions suivantes :

- **Absence de variables globales modifiables** : Une fonction réentrante ne doit pas modifier ou dépendre de variables globales (ou locales statiques) sans protection explicite.
- **Pas d'accès partagé non sécurisé** : Elle ne doit pas accéder à des ressources partagées sans synchronisation appropriée.
- **Utilisation de la pile pour les données locales** : Les données locales doivent être allouées sur la pile, ce qui garantit que chaque appel dispose de son propre espace mémoire.
- **Indépendance de l'état externe** : La fonction ne doit pas dépendre d'un état externe non protégé.

5.2 Explications pour MAIN_EX2

Dans cet exemple, la fonction `code()` est appelée par plusieurs tâches (`tache 1`, `tache 2`, `tache 3`). Lorsque cet exemple est testé, le comportement est identique à celui de `MAIN_EX1`, bien que toutes les tâches partagent la même fonction de traitement. Ce comportement peut être expliqué comme suit :

1. **Contexte indépendant pour chaque tâche** : Chaque tâche dispose de sa propre instance de la fonction `code()`, car les données locales (comme `id`) sont allouées sur la pile. Il n'y a donc pas d'interférence entre les tâches.
2. **Absence d'accès partagé** : La fonction `code()` ne dépend pas de variables globales ou statiques non protégées. Si elle accède à des ressources partagées, elles sont correctement synchronisées.
3. **Fonction correctement conçue** : La fonction `code()` respecte les principes de la réentrance, ce qui permet à plusieurs tâches de l'utiliser en parallèle sans provoquer de comportement imprévisible.

5.3 Pourquoi le comportement est-il identique ?

Même si plusieurs tâches utilisent la même fonction `code()`, chaque tâche utilise une pile distincte pour stocker ses données locales. Cela garantit qu'il n'y a pas de conflit ou de chevauchement dans l'utilisation de la mémoire entre les tâches. Par conséquent, le comportement de l'exemple `MAIN_EX2` est identique à celui de `MAIN_EX1`, bien que le code soit plus compact et réutilisable.

5.4 Conclusion

L'exemple `MAIN_EX2` démontre l'importance de la réentrance dans les environnements multitâches. En respectant les principes de la réentrance, une seule fonction peut être utilisée simultanément par plusieurs tâches, simplifiant ainsi la conception logicielle et réduisant les duplications inutiles.

6 Sémaphores (MAIN_EX3)

Un sémaphore est une structure utilisée pour gérer l'accès aux ressources partagées entre plusieurs tâches. Il se compose d'un compteur de jetons (`count`) et d'une liste de tâches bloquées en attente (`waiting`). Le sémaphore est défini dans le fichier `kernel/kernel.h` par :

```
typedef struct _Semaphore {
    int32_t count;       // Compteur de jetons disponibles
    Task *waiting;       // Liste des tâches en attente (FIFO)
} Semaphore;
```

Le compteur `count` indique le nombre de jetons disponibles. Si `count` devient négatif, cela signifie que des tâches sont bloquées en attente de jetons. Les tâches bloquées sont insérées dans la liste `waiting` en respectant l'ordre FIFO.

Trois appels système sont fournis pour gérer les sémaphores :

- `sem_new(init)` : Crée un nouveau sémaphore avec une valeur initiale.
- `sem_p(sem)` : Prend un jeton (*P operation*) et bloque la tâche si aucun jeton n'est disponible.
- `sem_v(sem)` : Libère un jeton (*V operation*) et débloque une tâche en attente, le cas échéant.

6.1 Implémentation des Appels Systèmes

6.1.1 `sem_new`

La fonction `sem_new` initialise un sémaphore en allouant dynamiquement sa mémoire et en initialisant ses champs. Son code est :

```
Semaphore *sys_sem_new(int32_t init) {
    Semaphore *sem = (Semaphore *)malloc(sizeof(Semaphore)); // Allocation dynamique
    if (sem == NULL) {
        return NULL; // Allocation échouée
    }
    sem->count = init; // Initialisation du compteur
    sem->waiting = NULL; // Liste des tâches vide
    return sem; // Retourne le sémaphore créé
}
```

6.1.2 `sem_p`

La fonction `sem_p` décrémente le compteur du sémaphore. Si aucun jeton n'est disponible (`count < 0`), la tâche courante est bloquée.

```
int32_t sys_sem_p(Semaphore *sem) {
    --sem->count;
    if (sem->count < 0) {
        Task *task;
        tsk_running = list_remove_head(tsk_running, &task); // Retirer de la liste des tâches en cours
        sem->waiting = list_insert_tail(sem->waiting, task); // Ajouter à la liste des tâches en attente
        task->status = TASK_WAITING;
        sys_switch_ctx(); // Commutation de contexte
    }
    return sem->count;
}
```

6.1.3 `sem_v`

La fonction `sem_v` incrémente le compteur. Si des tâches sont en attente, la première tâche dans la liste est débloquée.

```
int32_t sys_sem_v(Semaphore *sem) {
    if (sem == NULL) {
        return -1; // Sémaphore invalide
    }
    __disable_irq(); // Désactiver les interruptions
    sem->count++;
    if (sem->count <= 0) {
        Task *unblocked_task;
        sem->waiting = list_remove_head(sem->waiting, &unblocked_task); // Débloquer la première tâche
        if (unblocked_task != NULL) {
            unblocked_task->status = TASK_READY;
            tsk_running = list_insert_tail(tsk_running, unblocked_task);
```

```
        }
    }
    __enable_irq(); // Réactiver les interruptions
    return 0;
}
```

6.2 Test du Fonctionnement des Sémaphores

L'application suivante met en œuvre deux tâches synchronisées par un sémaphore :

1. **Tâche 2** : Attend une pression sur le bouton `USER` de la carte. Lorsqu'un appui est détecté, elle ajoute un jeton au sémaphore `sem`.
2. **Tâche 1** : Attend un jeton sur le sémaphore `sem`. Lorsqu'elle reçoit un jeton, elle incrémente et affiche une variable `cpt`.

Voici un exemple de code pour ces tâches :

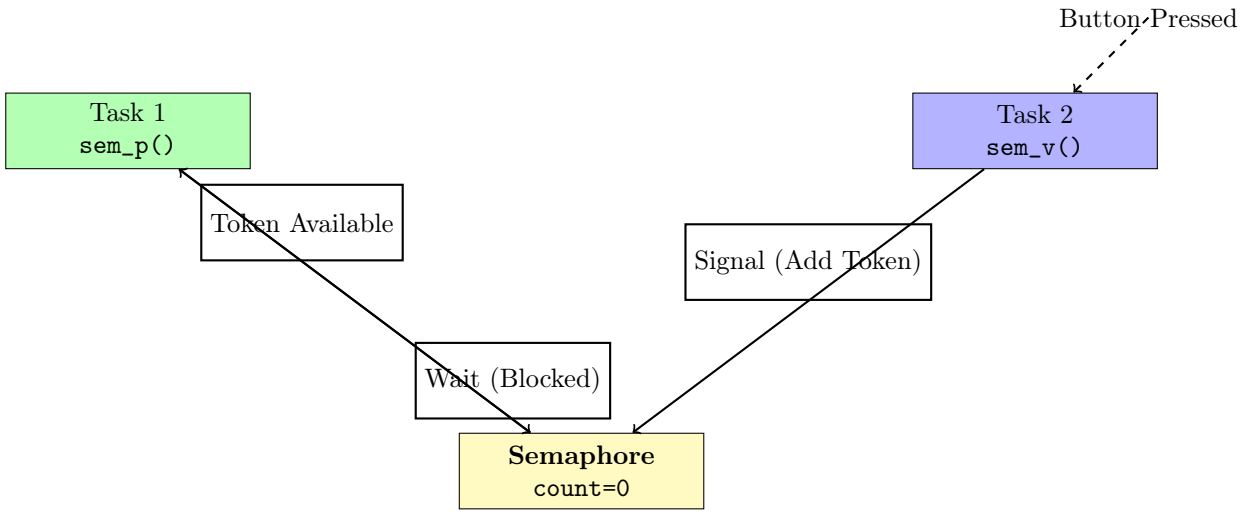


Figure 3: Semaphore Synchronization Between Task 1 and Task 2

```
void task1() {
    while (1) {
        sem_p(sem); // Attendre un jeton
        cpt++;
        printf("Button pressed, count: %d\n", cpt);
    }
}

void task2() {
    while (1) {
        if (button_pressed()) { // Détection d'appui
            sem_v(sem); // Ajouter un jeton
        }
    }
}
```

Ce test vérifie que les tâches sont correctement synchronisées à l'aide du sémaphore. Lors de la détection d'un appui, la tâche 2 libère un jeton, débloquant ainsi la tâche 1.

6.3 Observation des Commutations de Contexte

Lors des appels `sem_p` et `sem_v`, les commutations de contexte doivent être observées pour garantir que :

- Une tâche est bloquée si aucun jeton n'est disponible (`sem_p`).
- Une tâche en attente est débloquée dès qu'un jeton devient disponible (`sem_v`).

7 Sémaphores et Mutex (MAIN_EX4)

7.1 Différences entre un Semaphore et un Mutex

Les sémaphores et les mutex sont des mécanismes de synchronisation utilisés pour gérer l'accès aux ressources partagées.

7.1.1 Semaphore

Un sémaphore est un compteur qui contrôle l'accès à une ou plusieurs ressources partagées. Plusieurs tâches peuvent obtenir des jetons simultanément, permettant un accès parallèle aux ressources.

7.1.2 Mutex

Un mutex (*mutual exclusion*) garantit qu'une seule tâche accède à une ressource partagée à la fois. Contrairement aux sémaphores, un mutex ne peut être libéré que par la tâche qui l'a acquis, assurant une exclusion mutuelle stricte.

7.2 Pourquoi Utiliser un Mutex?

Un mutex est idéal pour garantir qu'une seule tâche accède à une ressource à la fois. Il empêche les accès concurrents erronés et simplifie la gestion de la synchronisation.

7.3 Application dans MAIN_EX4

Dans `MAIN_EX4`, un mutex est utilisé pour protéger les sections critiques. Chaque tâche doit acquérir le mutex avant d'y accéder, garantissant un accès exclusif.

7.3.1 Pseudocode

```
#ifdef MAIN_EX4

Semaphore *mutex = NULL;

void code() {
    uint32_t id = task_id();
    while (1) {
        sem_p(mutex);
        printf("tache %d\r\n", (int)id);
        sem_v(mutex);
        for (int k = 0; k < 50000; k++);
    }
}

void dummy() {
    while (1);
}

int main() {
    mutex = sem_new(1);
    task_new(code, 512);
    task_new(code, 512);
    task_new(code, 512);
```

```
    task_new(dummy, 128);
    os_start();
    return 0;
}

#endif
```

7.4 Résumé

L'exemple montre l'utilisation d'un mutex pour assurer l'exclusion mutuelle et éviter les conflits lors de l'accès aux ressources partagées.

8 Exercice Principal 5 : Implémentation de `task_wait`

8.1 Explication

La fonction système `task_wait` permet à une tâche de se mettre en pause pendant une durée spécifiée. Cela évite l'utilisation inutile du processeur avec des boucles d'attente et améliore l'efficacité.

8.2 Étapes

1. La tâche est retirée de la liste des tâches prêtes (`ready list`) et ajoutée à la liste des tâches en sommeil (`sleeping list`).
2. Un compteur de délai est initialisé avec la durée spécifiée.
3. Lors de chaque interruption du minuteur (`timer interrupt`), le compteur est décrémenté.
4. Lorsque le compteur atteint zéro, la tâche est déplacée à nouveau dans la liste des tâches prêtes.

8.3 Pseudocode

```
// Appelée par une tâche pour démarrer un délai
function task_wait(duration):
    current_task = get_running_task()
    current_task.delay = duration
    remove_from_ready_list(current_task)
    add_to_sleeping_list(current_task)

// Gestionnaire d'interruption du minuteur
function timer_interrupt_handler():
    for task in sleeping_list:
        task.delay -= SYS_TICK
        if task.delay <= 0:
            remove_from_sleeping_list(task)
            add_to_ready_list(task)
```

8.4 Figure

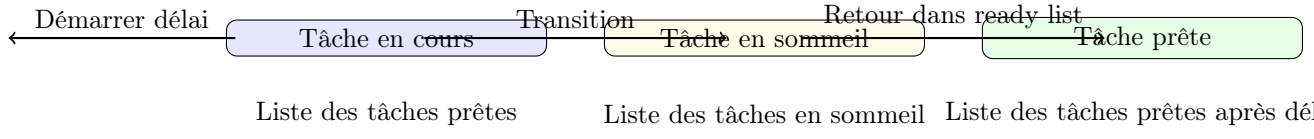


Figure 4: Transitions d'état lors de l'utilisation de `task_wait`

9 Exercice Principal 6 : Délais Simultanés

9.1 Explication

Cet exercice étend la fonctionnalité de `task_wait` pour plusieurs tâches. Chaque tâche gère son propre délai indépendamment, ce qui démontre la robustesse du planificateur de tâches en sommeil.

9.2 Étapes

1. Chaque tâche appelle `task_wait` avec une durée différente.
2. Le planificateur gère chaque tâche dans la liste des tâches en sommeil.
3. Une fois le délai terminé, les tâches sont déplacées dans la liste des tâches prêtes dans l'ordre approprié.

9.3 Pseudocode

```
// Exemple de tâche avec différents délais
function task1():
    while true:
        perform_task1_operations()
        task_wait(50)  // Délai de 50 ms

function task2():
    while true:
        perform_task2_operations()
        task_wait(100)  // Délai de 100 ms
```

9.4 Figure

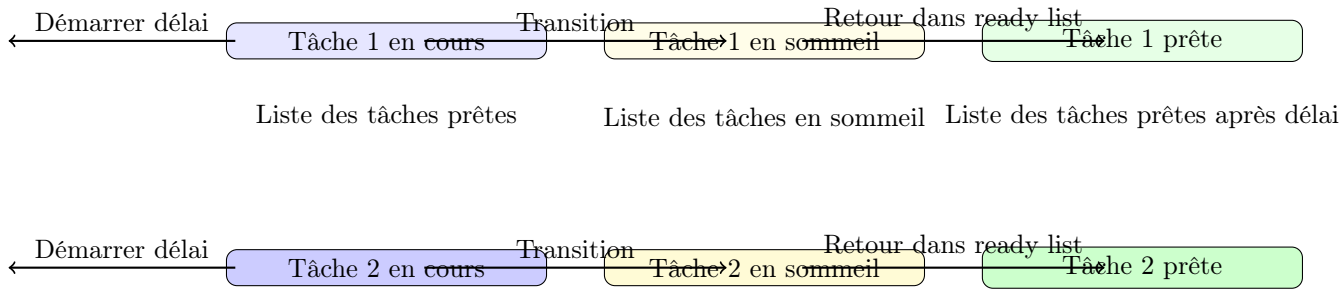


Figure 5: Gestion de plusieurs délais avec `task_wait`

10 Exercice Principal 7 : Terminaison de Tâche

10.1 Explication

La fonction système `task_kill` permet de terminer une tâche explicitement. Une terminaison implicite se produit lorsqu'une tâche atteint la fin de sa fonction.

10.2 Étapes

1. La tâche est retirée de toutes les listes (`ready`, `sleeping`, `blocked`).
2. Les ressources associées à la tâche sont libérées.
3. Si une tâche termine son exécution, `task_kill` est automatiquement appelée.

10.3 Pseudocode

```
// Terminaison explicite
function task_kill(task):
    if task in ready_list:
        remove_from_ready_list(task)
    if task in sleeping_list:
        remove_from_sleeping_list(task)
    if task in blocked_list:
        remove_from_blocked_list(task)
    free_task_resources(task)

// Terminaison implicite
function task_end():
    task_kill(get_running_task())
    schedule_next_task()
```

11 Exercice Principal 8 : Système de Fichiers Virtuel (VFS)

11.1 Explication

Le Système de Fichiers Virtuel (VFS) fournit une abstraction pour accéder aux périphériques matériels à travers une interface standardisée. Cela permet une interaction uniforme avec les fichiers et périphériques.

11.2 Fonctions Implémentées

- `open`: Ouvre un fichier ou périphérique et retourne un descripteur.
- `read`: Lit des données depuis un fichier ou périphérique.
- `write`: Écrit des données dans un fichier ou périphérique.
- `close`: Ferme un fichier ou périphérique.

11.3 Pseudocode pour `open`

```
function open(path, flags):
    if path == "/dev":
        return create_file_descriptor(F_IS_DEVDIR)

    device = dev_lookup(path)
    if device == NULL:
        return -1  // Périphérique introuvable

    file_obj = create_file_object(path, flags, device)
    fd = allocate_file_descriptor(file_obj)

    if device.init:
        if device.init(file_obj) == -1:
            free_file_descriptor(fd)
            return -1

    return fd
```

12 Exercice Principal 9 : Pilote de LED

12.1 Explication

Le pilote de LED permet de contrôler les LEDs RGB en utilisant une fonction `write` qui met à jour l'état des LEDs. Chaque bit du tampon de données (buffer) représente l'état d'une LED spécifique (par exemple, la LED rouge, verte, ou bleue).

12.2 Pseudocode pour Écriture dans les LEDs

```
function dev_write_leds(file_obj, buffer, length):
    if length != sizeof(uint32_t):
        return -1  // Taille de buffer invalide

    led_value = buffer_to_uint32(buffer)
    leds(led_value)  // Mise à jour du matériel LED
    return length
```

12.3 Figure

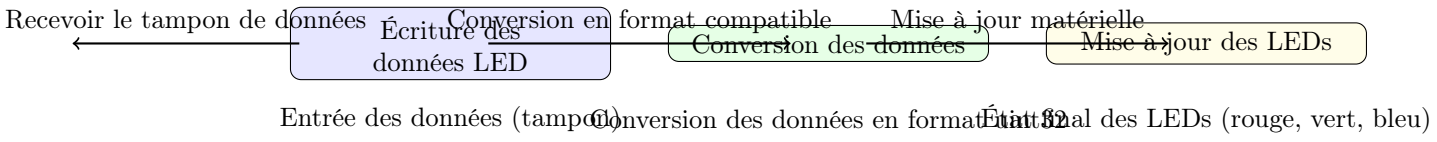


Figure 6: Fonctionnement du pilote de LED

13 Exercice Principal 10 : Gestion des Interruptions de Boutons

13.1 Explication

Cet exercice implémente la gestion d'événements asynchrones liés à l'appui sur un bouton. Les tâches en attente de l'événement sont bloquées jusqu'à ce que l'interruption soit déclenchée par le bouton.

13.2 Étapes

1. Une tâche attend un événement lié à l'appui sur le bouton. Pendant cette attente, elle est bloquée.
2. Lorsque l'utilisateur appuie sur le bouton, une interruption est déclenchée.
3. Le gestionnaire d'interruption débloque la tâche en attente et elle reprend son exécution pour traiter l'événement.

13.3 Pseudocode

```
// Fonction de la tâche en attente d'un appui sur le bouton
function task_wait_for_button():
    while true:
        wait_on_button_event()  // Bloque jusqu'à ce que le bouton soit pressé
        handle_button_press()   // Traite l'appui sur le bouton

// Gestionnaire d'interruption du bouton
function button_interrupt_handler():
    signal_button_event()  // Débloque la tâche en attente de l'événement
```

13.4 Explication du Pseudocode

- La fonction `task_wait_for_button` est exécutée en boucle par une tâche spécifique. Cette fonction utilise un mécanisme de blocage pour attendre un événement d'interruption déclenché par un bouton.
- Lorsqu'un bouton est pressé, une interruption est déclenchée. Le gestionnaire associé (`button_interrupt_handler`) appelle la fonction `signal_button_event` pour débloquer la tâche.
- Une fois débloquée, la tâche peut traiter l'appui sur le bouton, par exemple en exécutant une fonction `handle_button_press`.

13.5 Figure

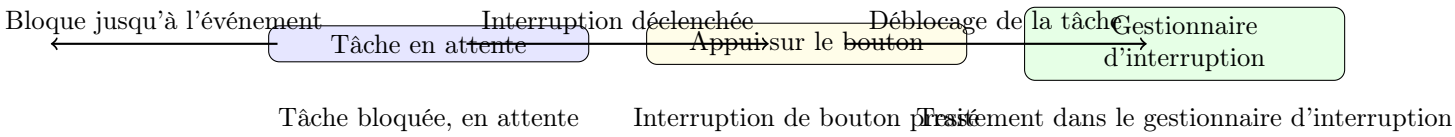


Figure 7: Workflow de la gestion des interruptions de boutons

13.6 Tests Réalisés

Les scénarios suivants ont été testés pour valider la gestion des interruptions :

- Une tâche a été créée pour attendre un événement d'appui sur un bouton.
- L'utilisateur a pressé le bouton, ce qui a déclenché une interruption.
- La tâche en attente a correctement repris son exécution après l'interruption et a traité l'événement.

13.7 Résultats

Les tests ont confirmé que :

- La tâche restait bloquée en attendant un événement.
- Le gestionnaire d'interruption a correctement débloqué la tâche en attente.
- Les tâches ont traité les événements d'appui sur le bouton sans conflit ni retard.