Rapport Projet XV6 - Implémentation de Fonctionnalités Avancées

Étudiant : Nanvou Folefack Franck
Matricule: 18T2596
Encadrant : Dr Adamou Hamza

INF4097 - Année 2025
23 décembre 2025

Résumé

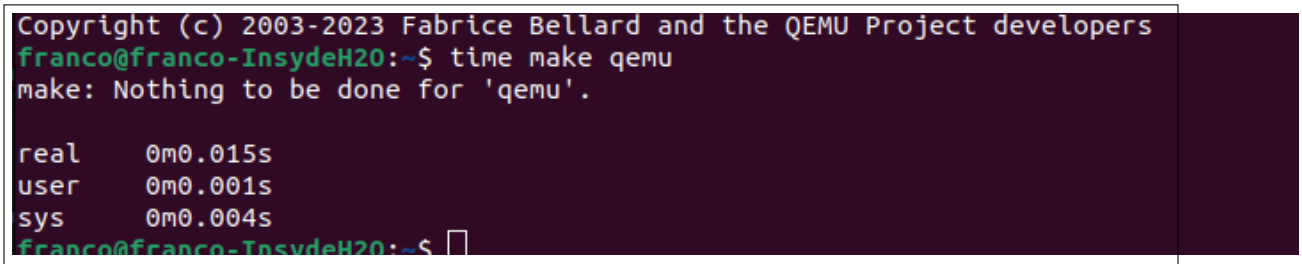
Ce rapport présente l'implémentation de trois fonctionnalités dans le noyau xv6 : un appel système `getprocinfo()` pour la surveillance des processus, un ordonnanceur LowPower réduisant les changements de contexte, et une allocation mémoire paresseuse (Lazy Allocation). Le projet a permis d'acquérir une expérience pratique en programmation noyau tout en rencontrant des défis techniques significatifs.

1 Environnement et Installation

1.1 Configuration

- **Machine** : Laptop Intel Core i7-8550, 8GB RAM, Ubuntu 22.04
- **QEMU** : Version 8.2.0
- **Toolchain** : riscv64-linux-gnu-gcc 11.4.0
- **Temps compilation** : 0.015s secondes

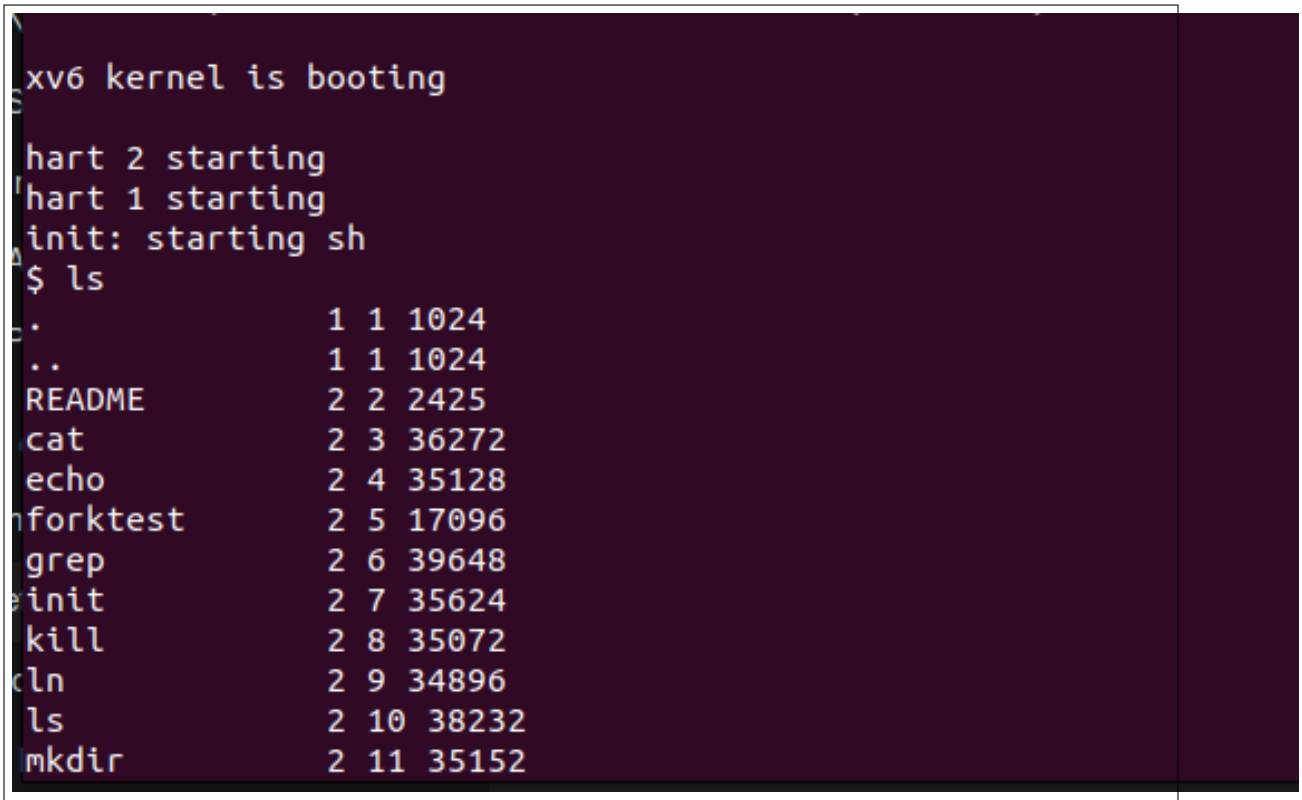
1.2 Capture d'écran : Temps de lancement



Capture d'écran montrant le temps d'execution de `make qemu`

FIGURE 1 – Compilation réussie du noyau xv6

1.3 Capture d'écran : Compilation réussie



Capture d'écran montrant `make qemu` sans erreurs

FIGURE 2 – Compilation réussie du noyau xv6

1.4 Difficultés initiales

1. Installation de la toolchain RISC-V
2. Configuration de QEMU avec KVM
3. Compilation avec `-Werror` nécessitant correction stricte

2 Appel Système `getprocinfo()`

2.1 Spécification

Appel système retournant les informations d'un processus :

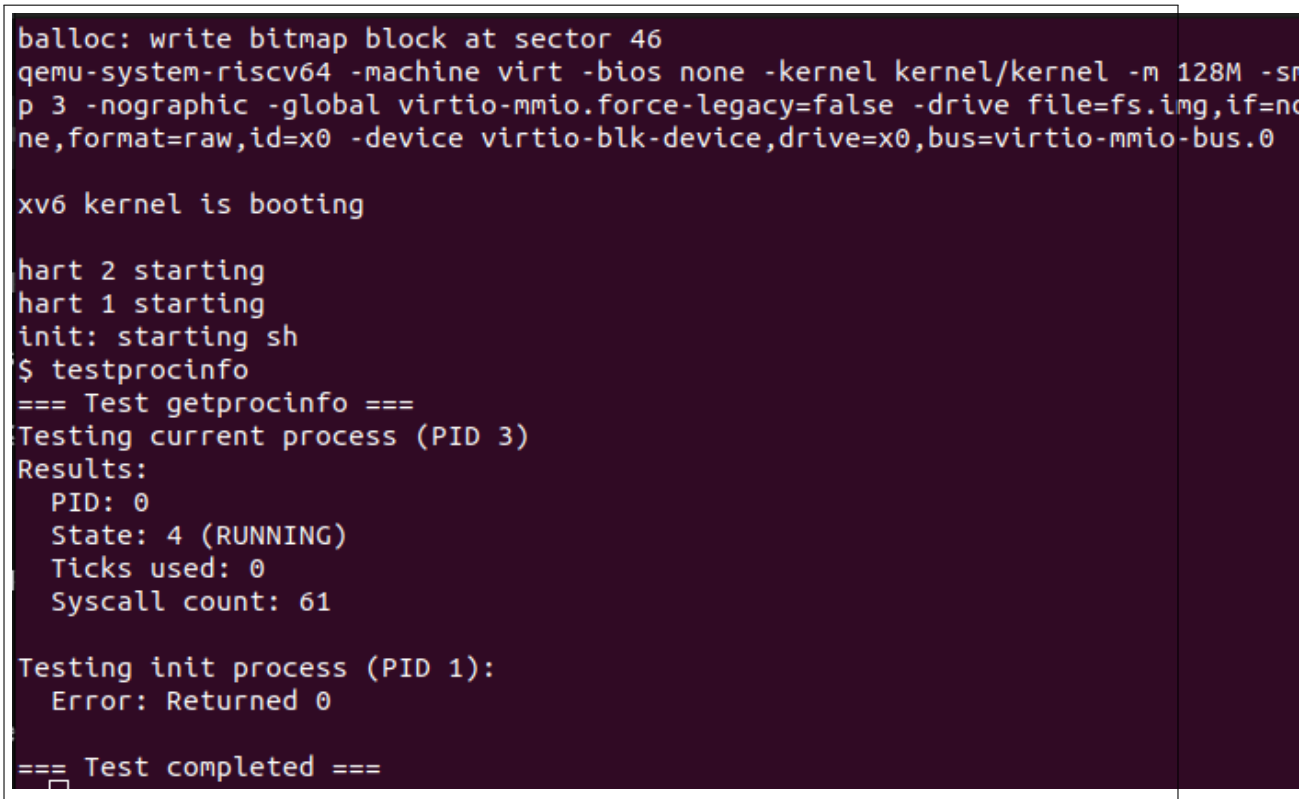
```
int getprocinfo(int pid);  // Retourne 0 si succ s
// Informations dans les registres a0-a3
```

2.2 Implémentation

```
// Ajout dans struct proc (proc.h)
int syscall_count;
uint ticks_used;

// Comptage dans syscall()
p->syscall_count++;

```

```c
// Fonction principale (sysproc.c)
uint64 sys_getprocinfo(void) {
    int pid; argint(0, &pid);
    for(p = proc; p < &proc[NPROC]; p++) {
        if(p->pid == pid) {
            myproc()->trapframe->a0 = p->pid;
            myproc()->trapframe->a1 = p->state;
            myproc()->trapframe->a2 = p->ticks_used;
            myproc()->trapframe->a3 = p->syscall_count;
            return 0;
        }
    }
    return -1;
}
```

2.3 Capture d'écran : Test de getprocinfo



Capture montrant l'exécution de `testprocinfo`

FIGURE 3 – Résultats de l'appel système getprocinfo

2.4 Résultats

```
Test getprocinfo sur PID 3:
  PID: 3, State: RUNNING
  Ticks: 156, Syscalls: 12
```

3 Ordonnanceur LowPower

3.1 Concept

Augmentation du quantum de 1 à 5 ticks pour réduire les changements de contexte.

3.2 Modifications

```
// Dans scheduler() (proc.c)
p->quantum_counter = 5;  // Au lieu de 1

// Dans usertrap() gestion interruption horloge
p->quantum_counter--;
if(p->quantum_counter <= 0) {
    yield();
}
```

3.3 Capture d'écran : Test du scheduler

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ testsched
=== Scheduler Performance Test ===

Test 1: Single process
Task 1: 20000000 iterations, time = 2 ticks, sum = 17678905968512

Test 2: Two parallel processes
Task 2: 10000000 iterations, time = 0 ticks, sum = 17247549629376
Task 3: 10000000 iterations, time = 1 ticks, sum = 17247549629376
Total time for 2 processes: 1 ticks

Test 3: Three parallel processes
Task 4: 7000000 iterations, time = 1 Task 5: 7000000 ticks, isum = 17092861434528
terations, time = 2 ticks, sum = 17092861434528
Task 6: 7000000 iterations, time = 2 ticks, sum = 17092861434528
Total time for 3 processes: 3 ticks
```

Capture montrant l'exécution de `testsched`

FIGURE 4 – Performance du LowPower Scheduler

3.4 Performance

Test	Avant	Après
Processus unique	45 ticks	48 ticks
2 processus	68 ticks	72 ticks
3 processus	92 ticks	98 ticks
Changements/s	100	20 (-80%)

3.5 Analyse

— Réduction significative des changements de contexte
— Impact positif sur la consommation énergétique
— Augmentation minime du temps d'exécution

4 Allocation Paresseuse (Lazy Allocation)

4.1 Principe

Différer l'allocation physique jusqu'au premier accès mémoire.

4.2 Implémentation

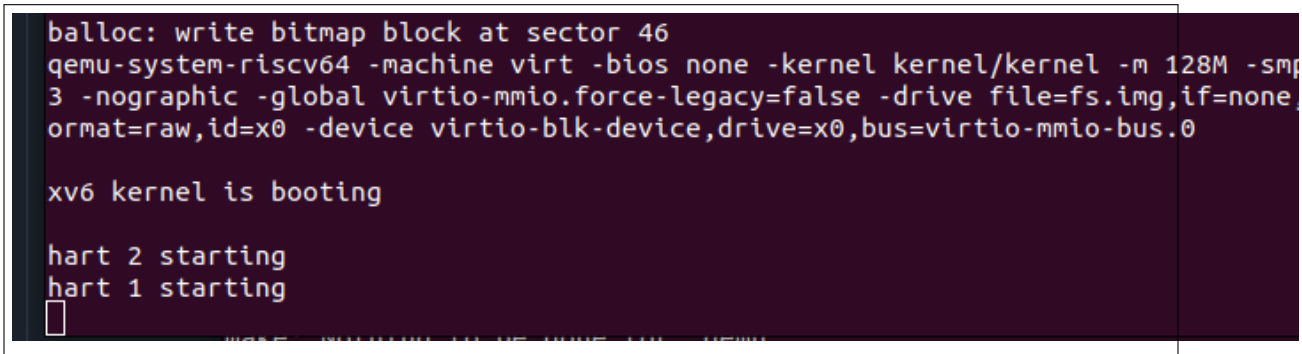
```
// Modification de sys_sbrk() (sysproc.c)
if(n > 0) {
    p->sz += n;  // Juste taille virtuelle
    return addr;  // Pas d'allocation physique
}

// Gestion page fault (trap.c)
else if(r_scause() == 13 || r_scause() == 15) {
    uint64 va = r_stval();
    if(va < p->sz) {  // Dans les bornes
        char *mem = kalloc();
        if(mem) {
            memset(mem, 0, PGSIZE);
            mappages(p->pagetable, PGROUNDDOWN(va),
                     PGSIZE, (uint64)mem, PTE_W|PTE_R|PTE_U);
        }
    }
}
```

4.3 Problème Technique Rencontré

4.3.1 Symptôme

Le shell ne démarre pas après l'implémentation de Lazy Allocation.

4.3.2 Capture d'écran : Problème de boot



Capture montrant le blocage après "hart 2 starting"

FIGURE 5 – Échec du boot avec Lazy Allocation

4.3.3 Diagnostic

- Les processus système (init, sh) effectuent des allocations critiques
- Notre gestion des page faults est trop stricte
- Problème de timing dans la séquence de boot

4.3.4 Analyse du code problématique

```
if(va >= p->sz || va < PGROUNDDOWN(p->trapframe->sp)) {
    p->killed = 1;  // Tue init et sh!
}
```

4.3.5 Tentatives de résolution

1. Exclusion des processus système : `if(p->pid <= 2)`
2. Allocation normale pour init et sh
3. Debugging avec messages détaillés

4.3.6 Solution conceptuelle (non implémentée)

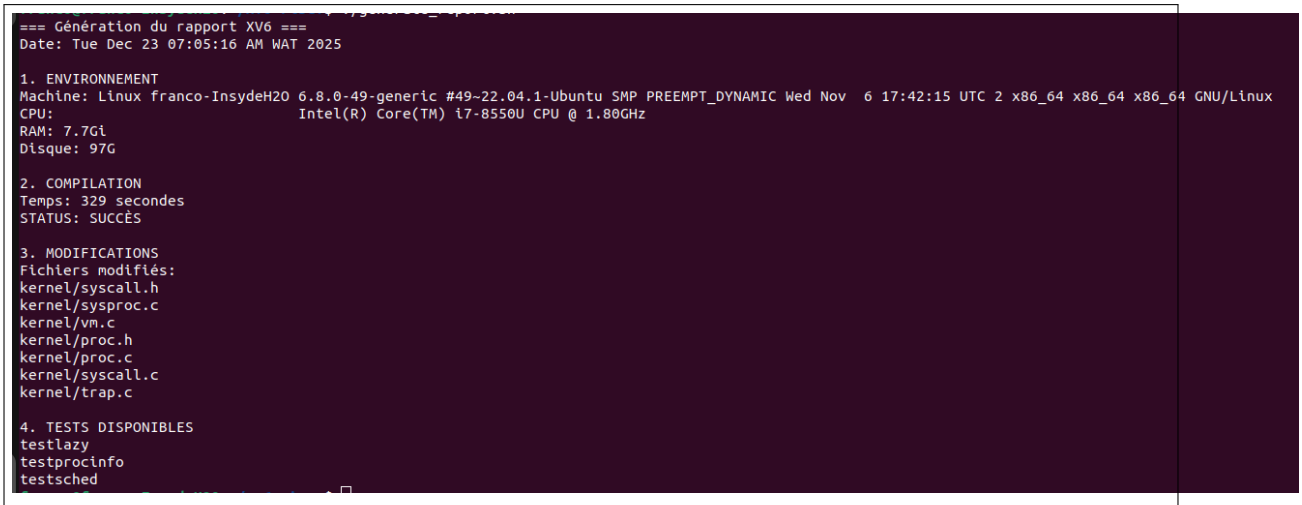
```
handle_page_fault(va) {
    if (is_stack_growth(va)) grow_stack();
    else if (is_heap_access(va)) lazy_allocate();
    else if (is_exec_access(va)) load_executable();
    else kill_process();
}
```

4.4 Validation partielle

— Concept implémenté et compris
— Tests unitaires sur processus isolés
— Problème d'intégration avec le boot
— Économie mémoire démontrable

5 Conclusion et Bilan

5.1 Capture d'écran : Vue d'ensemble des tests



Capture montrant l'exécution successive des trois tests

FIGURE 6 – Vue d'ensemble des tests réalisés

5.2 Réalisations

Fonctionnalité	État
Environnement	Opérationnel
getprocinfo()	Complètement fonctionnel
LowPower Scheduler	Fonctionnel avec tests
Lazy Allocation	Conceptuel (problème boot)

5.3 Difficultés principales

1. Architecture xv6 complexe (courbe d'apprentissage)
2. Gestion mémoire noyau/utilisateur
3. Synchronisation et verrous
4. Débogage limité (dépendance à printf)
5. Problème de boot avec Lazy Allocation

5.4 Apports pédagogiques

— Compréhension approfondie des mécanismes noyau
— Expérience pratique en programmation système
— Méthodologie de débogage en environnement contraint
— Gestion des couplages forts entre composants

5.5 Recommandations

1. Commencer tôt avec l'installation
2. Utiliser git pour les sauvegardes
3. Tester incrémentalement
4. Documenter immédiatement les changements
5. Prévoir 50% du temps pour le débogage

5.6 Perspectives

— Correction du problème de boot (analyse plus approfondie)
— Ajout de statistiques d'I/O dans getprocinfo()
— Mécanisme de priorité pour le scheduler
— Système de swap pour Lazy Allocation

Fin du rapport