Mémoire EDAC : spécifications fonctionnelles

Ce document constitue la spécification fonctionnelle du logiciel à réaliser. 3 versions successives du logiciel sont à implanter. Chaque version est spécifiée par un ensemble d'exigences et doit conduire à la production d'une distribution après vérification que 1) toutes les exigences soient vérifiées par au moins un test 2) que tous les tests unitaires et d'intégration se déroulent correctement.

Le logiciel à implanter est un mécanisme de détection/correction de la mémoire que l'on trouve typiquement dans les systèmes embarqués pour le spatial. Après un bref rappel sur les principes du mécanisme, ce document décrit les 3 versions à implanter ainsi que les exigences à respecter.

1. Mécanisme de protection mémoire EDAC

Le principe général d'un mécanisme de protection de la mémoire EDAC pour une application dans le spatial consiste à partitionner la mémoire en deux parties : une 1ère partie qui stocke les données de l'application et une seconde partie qui mémorise des bits de parité pour détecter et corriger les erreurs. Dans ce projet, on utilise une solution classique basée sur le code de Hamming.

L'objectif du projet consiste à mettre en œuvre un tel mécanisme dans l'espace noyau Linux ainsi que de vérifier son comportement grâce à un mécanisme d'injection de faute. On propose de réaliser ce mécanisme graduellement, en 3 étapes :

1. D'abord par une mise en œuvre en espace utilisateur, sous la forme d'une bibliothèque qui peut être utilisée par des applications multi-threadées.
2. Une seconde implantation mais en espace noyau sous la forme d'un pseudo-périphérique.
3. Enfin une troisième version complète la seconde version par un mécanisme de correction (ou *scrubber*) qui permet de corriger périodiquement les fautes sur les bits.

2. Version 1 : implantation du mécanisme EDAC en espace utilisateur

Pour la première version de ce mécanisme de protection mémoire, on souhaite proposer une mise en œuvre en espace utilisateur sous la forme d'une bibliothèque dont l'interface est la suivante :

```
#include <pthread.h>

#ifndef  PTHREAD_EDAC_H
#define  PTHREAD_EDAC_H  yes

typedef struct pthread_edac_t{
 . . .
} pthread_edac_t;


extern int pthread_edac_init(struct pthread_edac_t* edac,
     int memorysize);
extern int pthread_edac_write(struct pthread_edac_t* edac,
     unsigned char* data, int size, int offset);
extern int pthread_edac_read(struct pthread_edac_t* edac, unsigned
     char* data, int size, int offset);
```

```
extern int pthread_edac_destroy(struct pthread_edac_t* edac);

#endif
```

Du point de vue de l'utilisateur, cette bibliothèque permet :

- De créer une mémoire protégée par un mécanisme d'EDAC
- Il est possible de créer plusieurs mémoires EDAC au sein d'un même processus.
- Une mémoire EDAC peut être accédée par plusieurs threads simultanément.

Pour tester cette bibliothèque EDAC, il vous sera demandé d'écrire deux processus UNIX :

1. Un premier processus qui constitue une application utilisant 1 ou plusieurs mémoires EDAC.
2. Un second processus qui permet de simuler les fautes. Ce processus doit permettre d'injecter les fautes pendant l'exécution de l'application. Son rôle consiste à modifier certains bits des mémoires EDAC afin de simuler les fautes, c-à-d les inversions de bits liées au rayonnement solaire.

Afin de permettre aux 2 processus d'accéder aux mêmes mémoires EDAC, vous devrez employer un mécanisme d'IPC Unix (mécanisme de communication inter-processus). Le plus simple consiste à rendre accessible la mémoire EDAC aux deux processus grâce à un segment de mémoire partagé UNIX. Ces segments de mémoire sont mis en place via l'appel systèmes `shmat` et vous trouverez sur Internet de nombreuses documentations expliquant le fonctionnement de ce mécanisme système.

Dans cette première version, la détection et la correction des erreurs (ou *scrubbing*) s'effectuent lors d'une lecture de la mémoire. On parle de vérification paresseuse (ou *`lazy verification`*) car elle s'effectue au dernier moment. Lors d'une lecture, les données sont corrigées puis renvoyées à l'utilisateur. Après avoir expérimenté cette solution, on vous demande d'expliquer quel est le problème soulevé par cette mise en œuvre de la détection/correction.

Pour compléter la description de l'application à réaliser, vous trouverez ci-dessous un ensemble d'exigences. On vous demande de produire un jeu de tests unitaires qui couvre toutes les exigences ci-dessous. La couverture des exigences sera un élément important pour l'évaluation du travail. Par ailleurs, comme toute spécification, celle-ci peuvent être incomplète voire incohérente. C'est votre rôle de détecter ces erreurs et d'éclaircir avec le client les éventuelles ambiguïtés.

Numéro d'exigence	**Description**
V1-E1	Il est possible de créer plusieurs mémoires EDAC au sein d'un même processus.
V1-E2	Une mémoire EDAC peut être accédée par plusieurs threads simultanément: chaque lecture et écriture doit être atomique.
V1-E3	La fonction `pthread_edac_init` initialise la mémoire `edac`. L'argument `memorysize` spécifie la taille en caractère de la mémoire. En cas de succès, `pthread_edac_init` retourne 0. Elle retourne -1 en cas d'erreur et `errno` est alors initialisé avec l'erreur provoquée.

V1-E4	La fonction `pthread_edac_destroy` détruit la mémoire `edac`. En cas de succès, elle retourne 0. Elle retourne -1 en cas d'erreur et `errno` est alors initialisé avec l'erreur provoquée.
V1-E5	Les données sont accédées grâce à un offset. L'offset est un déplacement relatif vis-à-vis de l'adresse de la première donnée de la mémoire EDAC. Lors d'un appel à `pthread_edac_read` ou à `pthread_edac_write` la valeur de l'argument `offset` ne peut être ni négatif, ni supérieur à la taille de la mémoire EDAC -1
V1-E6	La fonction `pthread_edac_write` écrit dans la mémoire `edac` les `size` caractères pointés par `data` à partir de la position `offset` de la mémoire EDAC. Lors de l'écriture, les bits de parité du code de Hamming sont calculés et mémorisés. En cas de succès, `pthread_edac_write` retourne le nombre de caractères effectivement écrit. Elle retourne -1 en cas d'erreur et `errno` est alors initialisé avec l'erreur provoquée.
V1-E7	Si lors d'un appel à `pthread_edac_write` la valeur d'`offset` ne peut écrire `size` caractères car `offset+size>taille de la mémoire EDAC`, alors `pthread_edac_write` écrit le plus grand nombre possible de caractères et retourne le nombre de caractères effectivement écrit.
V1-E8	La fonction `pthread_edac_read` lit à la position `offset` depuis la mémoire `edac size` caractères et les mémorise dans la variable `data`. Lors de la lecture, les fautes sont détectées/corrigées. En cas de succès, `pthread_edac_read` retourne le nombre de caractères effectivement lus. Elle retourne -1 en cas d'erreur et `errno` est alors initialisé avec l'erreur provoquée.
V1-E9	Si lors d'un appel à `pthread_edac_read` la valeur d'`offset` ne peut lire `size` caractères car `offset+size>taille de la mémoire EDAC`, alors `pthread_edac_read` lit le plus grand nombre possible de caractères et retourne le nombre de caractères effectivement lus.

3. Version 2 : mise en œuvre du mécanisme EDAC en espace noyau

Pour cette nouvelle version, l'ensemble du service est implanté dans l'espace noyau Linux. Plusieurs processus ou threads peuvent donc accéder à une même mémoire EDAC via un fichier périphérique donné. La mémoire est donc accessible via un pilote pour lire et écrire les données ainsi que leurs bits de parité. Voici la nouvelle interface de ce service :

```
int open  (char* pathname, int flags, mode_t mode);
int creat (char *pathname, mode_t mode);
int read  (int fd, char* buff, int count);
int write (int fd, char* buff, int count);
int lseek (int fd, int offset, int whence);
int close (int fd);
int ioctl (int fd, int request, unsigned long arg);
```

Un processus utilisant `creat` (ou `open` avec le paramètre `flags` adéquat) permet d'initialiser la zone mémoire et les services associés. Une fois mise en place, la mémoire peut être ouverte en lecture, en écriture, ou en lecture/écriture selon les arguments passés lors de l'appel à `open`.

Les permissions (argument `mode`) sont gérées selon les règles usuelles avec Unix.

Une mémoire est écrite avec `write` et lue avec `read`. Comme pour la version 1, lecture et écriture sont réalisées à partir de la position de l'offset. Par défaut, `open` positionne l'offset à 0. `read/write` incrémentent l'offset au fur et à mesure des opérations de lecture/écriture. La fonction `lseek` permet de positionner l'offset à une valeur particulière.

Il est possible de définir plusieurs mémoires EDAC sur un même système : le mineur du pilote permet d'identifier les mémoires EDAC les unes des autres.

L'injection de faute est toujours réalisée par un processus indépendant de l'application qui accède à une mémoire EDAC via le driver et modifie les bits grâce à une opération `ioctl`. La commande `ioctl` permettant cette injection de faute est EDAC_FAULT.

Enfin, le *`scrubbing`* (vérification et correction des fautes) est toujours implanté en mode *lazy* : le *`scrubbing`* est déclenché uniquement lors d'une opération `read`. Ainsi, lors d'un appel à `read`, les fautes sont détectées avant de retourner les données demandées/corrigées à l'appelant.

Enfin une mémoire EDAC à par défaut une taille de 1024 caractères. La taille peut être modifiée par `ioctl` mais seulement avant toute opération de lecture ou d'écriture avec la commande EDAC_SETMEMORYSIZE. La taille de la mémoire peut être consultée à tout moment avec la commande `ioctl` EDAC_GETMEMORYSIZE.

La sémantique de ces fonctions est décrite dans le tableau ci-dessous qui regroupe l'ensemble des exigences du logiciel à réaliser pour la version 2.

Numéro d'exigence	**Description**
V2-E1	`creat` (ou lors d'un appel à `open` équivalent, c-à-dire pour `mode` avec O_CREAT) permet de créer une mémoire EDAC identifiée par le nom `pathname` et dont la taille par défaut est de 1024 caractères. `creat` retourne le descripteur de fichier si la mémoire est correctement initialisée.
V2-E2	`creat` (ou lors d'un appel à `open` équivalent) retourne -1 lors d'un appel pour une mémoire EDAC déjà initialisée. `errno` est initialisée avec l'erreur détectée.
V2-E3	`open` permet l'ouverture d'une mémoire EDAC identifiée par le nom `pathname`. `open` retourne le descripteur de fichier permettant d'écrire et de lire dans la mémoire.
V2-E4	`open` et `creat` retournent -1 lors d'un appel pour un `pathname` inexistant. `errno` est initialisée avec l'erreur détectée.

V2-E5	Le paramètre `flags` pour `open` est l'un des éléments O_RDONLY, O_WRONLY ou O_RDWR qui réclament respectivement l'ouverture de la mémoire en lecture seule, écriture seule, ou lecture/écriture. `open` retourne -1 lors d'un appel avec une option différente des valeurs O_RDONLY, O_RDWR et O_WRONLY.
V2-E6	`close` permet de terminer les opérations sur une mémoire EDAC pour le descripteur `fd`. `close` retourne 0 si la fermeture s'est déroulée correctement.
V2-E7	`close` retourne -1 lors d'un appel si le paramètre `fd` ne correspond à aucun fichier ouvert. `errno` est initialisée avec l'erreur détectée.
V2-E8	`write` tente d'écrire `count` caractères depuis `buff` vers la mémoire EDAC. Si les caractères ont pu être écrits normalement, `write` retourne le nombre de caractères écrit et -1 sinon. `errno` est initialisée avec l'erreur détectée si -1 est retournée.
V2-E9	`write` retourne -1 lors d'un appel si `count` est négatif. `errno` est initialisée avec l'erreur détectée.
V2-E10	`write` retourne -1 lors d'un appel si le paramètre `fd` ne désigne pas une mémoire ouverte. `errno` est initialisée avec l'erreur détectée.
V2-E11	Si lors d'un appel à `write` la valeur d'`offset` ne peut écrire `count` caractères car `offset+count>taille de la mémoire EDAC`, alors `write` écrit le plus grand nombre possible de caractères et retourne le nombre de caractères effectivement écrit.
V2-E12	`read` tente de lire `count` caractères depuis `buff` depuis la mémoire EDAC. Si les caractères ont pu être lus normalement, `read` retourne le nombre de caractères lus et -1 sinon. `errno` est initialisée avec l'erreur détectée si -1 est retournée.
V2-E13	`read` retourne -1 lors d'un appel si `count` est négatif. `errno` est initialisée avec l'erreur détectée.
V2-E14	`read` retourne -1 lors d'un appel si le paramètre `fd` ne désigne pas une mémoire ouverte. `errno` est initialisée avec l'erreur détectée.
V2-E15	Si lors d'un appel à `read` la valeur d'`offset` ne peut lire `count` caractères car `offset+count>taille de la mémoire EDAC`, alors `read` lit le plus grand nombre possible de caractères et retourne le nombre de caractères effectivement lus.
V2-E16	L'argument `whence` supporte les trois valeurs suivantes : **SEEK_SET** L'offset est positionné au `offset` ième caractère. **SEEK_CUR** L'offset est incrémenté depuis sa valeur actuelle d'`offset` caractères. **SEEK_END** L'offset est positionné à la taille de la mémoire EDAC moins

	offset caractères.
V2-E17	lseek positionne l'offset de la mémoire EDAC à la valeur offset. En cas de succès, lseek renvoie la nouvelle valeur de l'offset et -1 sinon. errno est initialisée avec l'erreur détectée si -1 est retournée.
V2-E18	lseek retourne -1 lors d'un appel si offset est strictement supérieur à la taille de la mémoire EDAC ou si offset est négatif. errno est initialisée avec l'erreur détectée.
V2-E19	lseek retourne -1 lors d'un appel si le paramètre fd ne désigne pas une mémoire ouverte. errno est initialisée avec l'erreur détectée.
V2-E20	lseek retourne -1 lors d'un appel si la combinaison offset et whence conduit à une nouvelle valeur d'offset strictement supérieur à la taille de la mémoire EDAC ou strictement négatif. errno est initialisée avec l'erreur détectée.
V2-E21	Les seules valeurs possibles de l'argument request d' ioctl sont EDAC_GETMEMORYSIZE , EDAC_SETMEMORYSIZE et EDAC_FAULT.
V2-E22	Lors d'un appel à ioctl avec EDAC_GETMEMORYSIZE, ioctl renvoie la taille de la mémoire EDAC.
V2-E23	Lors d'un appel à ioctl avec EDAC_SETMEMORYSIZE, si la mémoire EDAC a déjà été utilisée (appel de read ou de write), alors ioctl renvoie -1. errno est initialisée avec l'erreur détectée.
V2-E24	Lors d'un appel à ioctl avec EDAC_SETMEMORYSIZE et à condition qu'aucune lecture/écrite n'ait été exécutée, alors la taille de la mémoire est initialisée à la valeur de l'argument arg. En cas de succès, ioctl renvoie 0 et -1 sinon. errno est initialisée avec l'erreur détectée si -1 est retournée.
V2-E25	Lors d'un appel à ioctl avec EDAC_FAULT, une faute est insérée sur le numéro de bits passé via l'argument arg. En cas de succès, ioctl renvoie 0 et -1 sinon. errno est initialisée avec l'erreur détectée si -1 est retournée.
VZ-E26	Lors d'un appel à ioctl avec EDAC_FAULT, le numéro de bits constitue un déplacement relatif vis-à-vis du premier bits de la mémoire EDAC. Ainsi, si la commande ioctl EDAC_FAULT est invoquée avec la valeur n, alors on modifie le kième bit (k = reste de n modulo 8) de l'offset n modulo 8. Par exemple, la demande de modification du bits 1021 déclenchera une modification à l'offset 1021 modulo 8, soit l'offset 127. A cette position, c'est le 5 ième bits qui devra être inversé (en effet, le reste de 1021 modulo 8 est égal à 5)

V2-E27	`ioctl` retourne -1 lors d'un appel si le paramètre `fd` ne désigne pas une mémoire ouverte. `errno` est initialisée avec l'erreur détectée.

4. Version 3 : correction de faute implantée par timer et tasklet périodique noyau

L'implantation de la version 3 est strictement identique à celle de la version 2 en dehors de la correction des fautes. En dehors d'`ioctl`, l'API est identique et l'injection de faute est toujours un processus extérieur.

Pour supprimer l'inconvénient de la correction des fautes lors de la lecture des données (en mode *lazy),* on propose cette fois-ci d'implanter la correction avec des timers et tasklet noyau. Le principe consiste à exécuter une analyse périodique de la mémoire EDAC afin de corriger lors de cette analyse toutes fautes détectées.

La période de correction peut être spécifiée par un appel à `ioctl` avec la valeur `EDAC_PERIOD`. On suppose une valeur par défaut de 2000 tics de l'horloge système pour cette correction périodique.

En dehors de V2-E21, toutes les exigences de la version 2 doivent être assurées pour la version 3. Par ailleurs, les nouvelles exigences ci-dessous doivent également être vérifiées :

Numéro d'exigence	**Description**
V3-E1	La période par défaut du timer noyau pour la correction périodique est par défaut de 2000 tics système.
V3-E2	Les seules valeurs possibles pour l'argument `request` d' `ioctl` sont EDAC_PERIOD, EDAC_GETMEMORYSIZE, EDAC_SETMEMORYSIZE et EDAC_FAULT.
V3-E3	`ioctl` modifie la période du *`scrubber`* lorsque `request` vaut `EDAC_PERIOD`. La période est donnée en nombre de tics système via l'argument `arg`. En cas de succès, `ioctl` retourne 0 ou -1 sinon. `errno` est initialisée avec l'erreur détectée.
V3-E4	Pour la commande `EDAC_PERIOD`, tout appel à la fonction `ioctl` avec une valeur <=0 pour l'argument `arg` retourne -1. `errno` est initialisée avec l'erreur détectée.