Programmation pour cartes à puce
Cours 3 - Transactions

Halim Djerroud

révision : 0.1

Situation du problème

Le problème de la carte arrachée :

- Incohérences si lors d'une exécution de la carte, celle-ci est arrachée.
- Exemple : arrachement de la carte du ***TP Porte_monnaie*** pendant que celle-ci effectue un "crédit" ou un "débit".

Solution :

- Lorsque plusieurs opérations sont liées entre elles, on voudrait qu'elles s'effectuent toutes ou-bien qu'aucune ne s'effectue.
- Il faut donc rendre certaines opérations atomiques, indivisible : soit on fait **TOUT** ou alors **RIEN**.

Carte à puce : Taille des données

Dans le domaine de la carte à puce, lorsqu'on personnalise une donnée, il est nécessaire de connaître la taille de la donnée. Si l'on inscrit correctement la donnée mais pas la taille, la donnée n'est pas cohérente.

- Il est impératif que les deux opérations soient toutes les deux effectuées.
- Solution : Transaction

Définition d'une Transaction

Définition 1

Une transaction est une association d'opérations non indépendantes. Le but des transactions est de définir un procédé qui puisse, soit effectuer toutes les opérations liées entre elles, soit n'en effectuer aucune.

- Avant de se poser des questions sur les attaques, la première des sécurités est la robustesse du code.

Définition d'une Transaction

Définition 2

On dit qu'un code est robuste s'il fonctionne dans le maximum de situations, même inattendues.

- La plupart des piratage de cartes ont lieu à cause de code non robuste. En effet, l'une des techniques classiques de piratage est le débordement de tampon,buffer overflow qui consiste à donner des tailles incohérentes qui permettent d'avoir accès à des zones mémoires qui n'étaient pas prévues.

Définition d'une Transaction

Définition 3

Une transaction est donc composée d'une suite de d'actions qui doivent vérifier les propriétés d'**atomicité**, de **cohérence**, d'**isolation** et de **durabilité**, résumées par le vocable **ACID**.

- **"A" tomicité** : opérations indissociables les unes des autres, correspond à l'indivisibilité (on fait tout ou rien). Une transaction doit effectuer toutes ses mises à jour ou ne rien faire du tout. En cas d'échec, le système doit annuler toutes les modifications qu'elle a engagées. L'atomicité est menacée par :
  - Panne matériel (lecteur de carte, ordinateurs ...).
  - Très souvent l'**arrachement de la carte**.
  - Généralement par tout événement susceptible d'interrompre une transaction en cours.

Définition d'une Transaction (2)

- **"C" ohérence** : La transaction doit faire passer le système d'un état cohérent à un autre. En cas d'échec, l'état cohérent initial doit être restauré. La cohérence de la base peut être violée par un programme erroné.
- **"I" solation** : Les résultats d'une transaction ne doivent être visibles aux autres transactions qu'une fois la transaction validée, afin d'éviter les interférences avec les autres transactions.
- **"D" urabilité** : Dès qu'une transaction valide ses modifications, le système doit garantir qu'elles seront conservées durablement même en cas de panne.

Exemple : Transactions financières

En général, on crédite un compte mais pour pouvoir le créditer, il faut en débiter un autre. Deux cas de figure se présentent :

- Soit l'opération de crédit et l'opération de débit sont toutes les deux effectuées et donc, le transfert d'argent a été effectué correctement.
- Soit aucune des deux opérations n'est effectuée.

On ne peut pas créditer sans débiter car si c'était le cas, il y aurait une création d'argent et on ne peut pas débiter sans créditer car il y aurait alors une disparition d'argent.

Exemple typiques que peut poquer l'arrachement de cartes

- Arrachement arrachement lors de l'écriture de taille durant l'opération de personnalisation. Aura pour conséquence un problème du buffer overflow, c'est-à-dire que l'on dépasse la capacité de la mémoire donnant ainsi accès à des zones pouvant contenir des données privées.

Solution à l'arrachement des cartes

- Il faut cibler l'anti-arrachement des cartes à puce pour protéger les données en EEPROM. C'est donc l'écriture en EEPROM qu'il faut cibler et programmer sous transaction (changement d'état avec les propriétés ACID).
- Il n'y a pas de solution absolue à ce problème. La seule chose qu'on puisse faire est de minimiser les risques. On va se contenter du risque résiduel acceptable.

Solution à l'arrachement des cartes

- Dans ce cours, on se propose de procéder en 2 temps
  - **Engagement** de la transaction qui va consister à copier les données dans une mémoire auxiliaire
  - **Validation** de la transaction qui va consister à copier les données de la mémoire auxiliaire dans l'adresse de destination
- Pour cela, on a besoin de deux choses :
  - Une **mémoire auxiliaire** : mémoire tampon dans laquelle on va inscrire des données pour l'objet de la transaction.
  - Un **automate** qui va décrire ce que contient la mémoire tampon. Elle peut être :
    - Soit vide : `ETAT=VIDE`
    - Soit contenir des données : `ETAT = DATA`

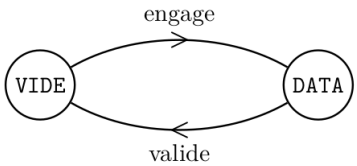
Solution à l'arrachement des cartes (2)

Lorsqu'on va engager la transaction, on va faire une transition de l'état VIDE à l'état *DATA* et lorsqu'on va valider la transaction, les données seront à l'adresse de destination et donc la mémoire tampon passera à l'état **VIDE**.

C'est un automate à deux états avec deux transitions qui sont justement les fonctions :

- **Engage** : transformer l'état de la mémoire tampon de vide vers un état qui contient des données.
- **Valide** : vide le tampon pour écrire des données à l'adresse de destination

Solution à l'arrachement des cartes (2)

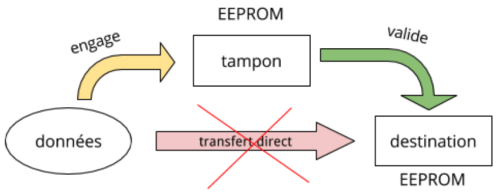


- L'état **VIDE** signifie qu'il n'y a pas de données dans la mémoire tampon et qu'il n'y a pas de transaction engagée.
- L'état **DATA** signifie qu'une transaction a été engagée, que les données se trouvent dans la mémoire tampon et qu'elles sont destinées à aller dans une adresse de destination bien spécifique.

Cas d'utilisation : Écriture en EEPROM

Lorsqu'on veut écrire dans l'EEPROM, on ne va pas appeler la fonction `eeprom_write_xx()`, mais plutôt ces deux fonctions :

- `Engage(n, d, e)` : écrire `n` octets de données accessibles à l'adresse `d` vers l'EEPROM à l'adresse `e`.
- Valide() : Valider une transaction en cours qui aurait été interrompue



S'il y a eu un arrachement pendant cette écriture, et s'il y a eu une transaction engagée mais interrompue à cause de l'arrachement, il faut appeler la fonction `Valide()` au `RESET` (remise sous tension).

Engager et valider

Que signifie "**engager**" ?

1. mettre l'état à vide (`ETAT←VIDE`) **(a)**
2. copier les données en mémoire tampon **(b)**
3. mettre l'état à data (`ETAT←DATA`) **(c)**
4. Engagement terminé **(d)**

Que signifie "**valider**" ?

1. si `ETAT=DATA` alors déplacer les données de la mémoire tampon vers la destination **(e)**
2. mettre l'état à vide (`ETAT←VIDE`) **(f)**

Remarque

Tout état différent de `DATA` est interprété comme vide.

Solution : Étude des différentes erreurs possibles

Si on arrache la carte en :

- **(a)** : On est dans le cas résiduel : si `ETAT=DATA` (au lieu de VIDE), erreur lors de la validation du prochain `RESET` car on va déplacer des données qui n'existent pas. Ce cas est peu probable si on choisit comme valeur pour `DATA` une donnée qui n'a pas plus de raison que les autres d'être écrite. Si on considère que l'écriture est aléatoire, on a une chance sur 256 que la donnée est `DATA`.
- **(b)** : L'état reste vide, transaction non engagée
- **(c)** : Deux cas :
  - si `ETAT=DATA` → souhaité
  - si `ETAT`≠`DATA` , `ETAT=DATA` , transaction non engagée
- **(d)** : Validé au prochain `RESET`
- **(e)** : Pendant le transfert des données, état reste data, validé au prochain RESET
- **(f)** : Deux cas :
  - si `ETAT`≠`DATA` → souhaité
  - si `ETAT=DATA` , validé au prochain `RESET`

Mise en œuvre : Structure de données

On va définir une structure qui va contenir la mémoire tampon et l'état de l'automate. Cette structure contiendra :

- un octet : état
- un entier : taille des données présentes dans la mémoire tampon
- un pointeur : adresse de destination des données dans l'`EEPROM`
- un tableau : les données de la mémoire tampon

Mise en œuvre : Macros pour les transactions

```
// Macros pour les transactions
#define DATA 0x1c
#define VIDE 0
#define MAX_TAMPON 64
```

Mise en œuvre : la structure transaction

Ensuite, on définit la structure transaction et on déclare une variable de ce type en EEPROM.

```
// définition de la structure pour les transactions
struct transaction {
    uint8_t etat ;
    uint8_t taille ; // taille des données de la mémoire tampon
    uint8_t *destination ;
    uint8_t tampon[MAX_TAMPON];
};
// déclaration d'un type transaction en EEPROM
struct transaction ee_trans EEMEM = {VIDE, 0, 0} ;
```

Mise en œuvre : Fonction Engage

```
// écrire n octets de données en source vers destination en EEPROM
void Engage(int n, void *source, void *destination){
    // mettre l'état à VIDE
    eeprom_write_byte(&ee_trans.etat, VIDE) ;
    // introduire les données dans ee_trans dans l'EEPROM
    eeprom_write_byte(&ee_trans.taille, n) ;
    eeprom_write_word((uint16_t*)&ee_trans.destination, (uint16_t)
    destination) ;
    eeprom_write_block(source, ee_trans.tampon, n) ;
    // mettre l'état à DATA
    eeprom_write_byte(&ee_trans.etat, DATA) ;
}
```

Mise en œuvre : Fonction Valide

```
// transfère les données de la mémoire tampon dans la destination
void Valide(){
    int i ;
    int taille_tampon ;
    // récupère la taille du tampon dans l'EEPROM
    taille_tampon = eeprom_read_byte(&ee_trans.taille) ;
    // récupère l'adresse de destination dans l'EEPROM
    uint16_t *dest = eeprom_read_word((uint16_t*)&ee_trans.destination) ;
    // si etat = DATA, on transfère les données
    if(eeprom_read_byte(&ee_trans.etat) == DATA){
        for(i = 0; i < taille_tampon ; i++){
            eeprom_write_byte((uint8_t*)&dest[i], eeprom_read_byte(&ee_trans
            .tampon[i])) ;
        }
    }
    // mettre l'état à VIDE
    eeprom_write_byte(&ee_trans.etat, VIDE) ;
}
```

Mise en œuvre : Recharger() et Depenser()

Enfin, dans les méthodes Recharger() et Depenser(), on remplace la ligne

```
eeprom_write_word(&solde, montant);
```

par ces deux lignes

```
Engage(2, &montant, &solde);
Valide() ;
```

Et on n'oublions pas de valider au RESET.