Périphériques Initiateurs

LU3IN031 Architecture des ordinateurs - 2
Matériel et Logiciel
B8

franck.wajsburt@lip6.fr
V4

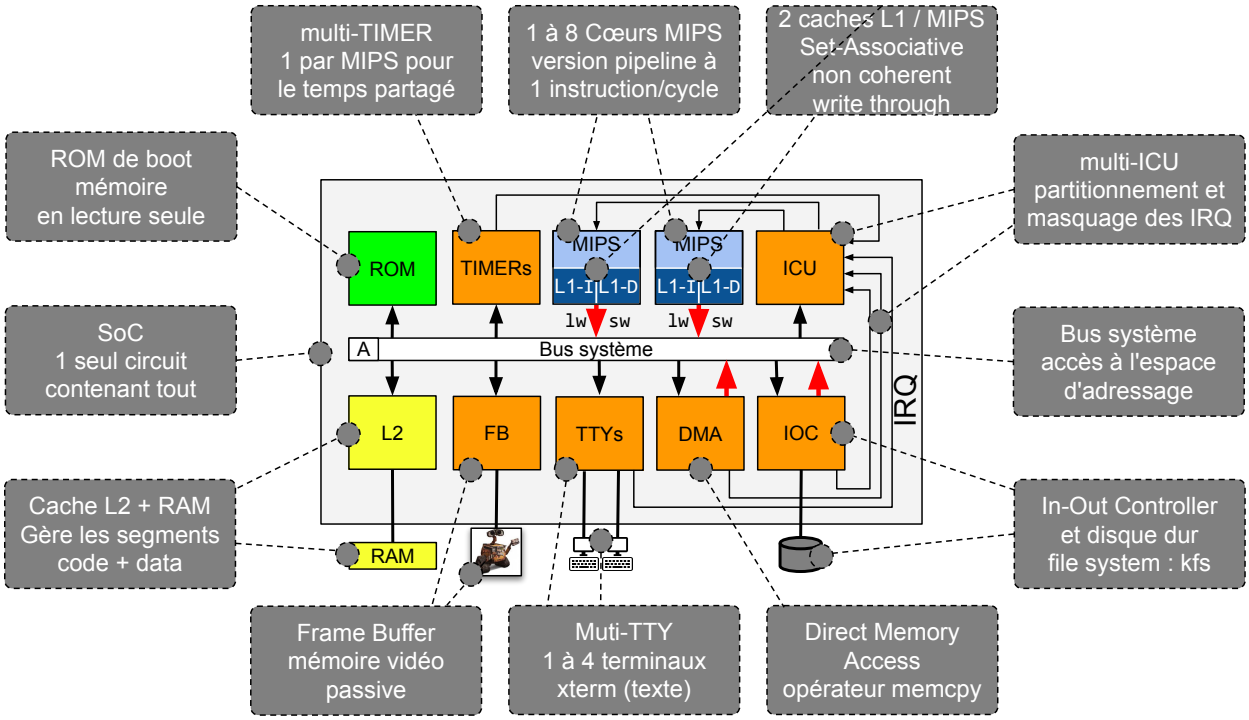
Objectifs de la séance

Almo-1 est une architecture avec 1 à 8 coeurs mais sans cohérence des caches L1, avec des contrôleurs de périphériques pour des fonctions internes (Timer, ICU et DMA) et avec des contrôleurs de périphériques pour des entrées-sorties (TTY, contrôleur vidéo FB et contrôleur de disque)
Comment faire tout fonctionner en même temps ?

- Revoir l'ensemble des composants
- Faire fonctionner plusieurs composants initiateurs en même temps
- Utiliser le contrôleur vidéo pour afficher des objets graphiques

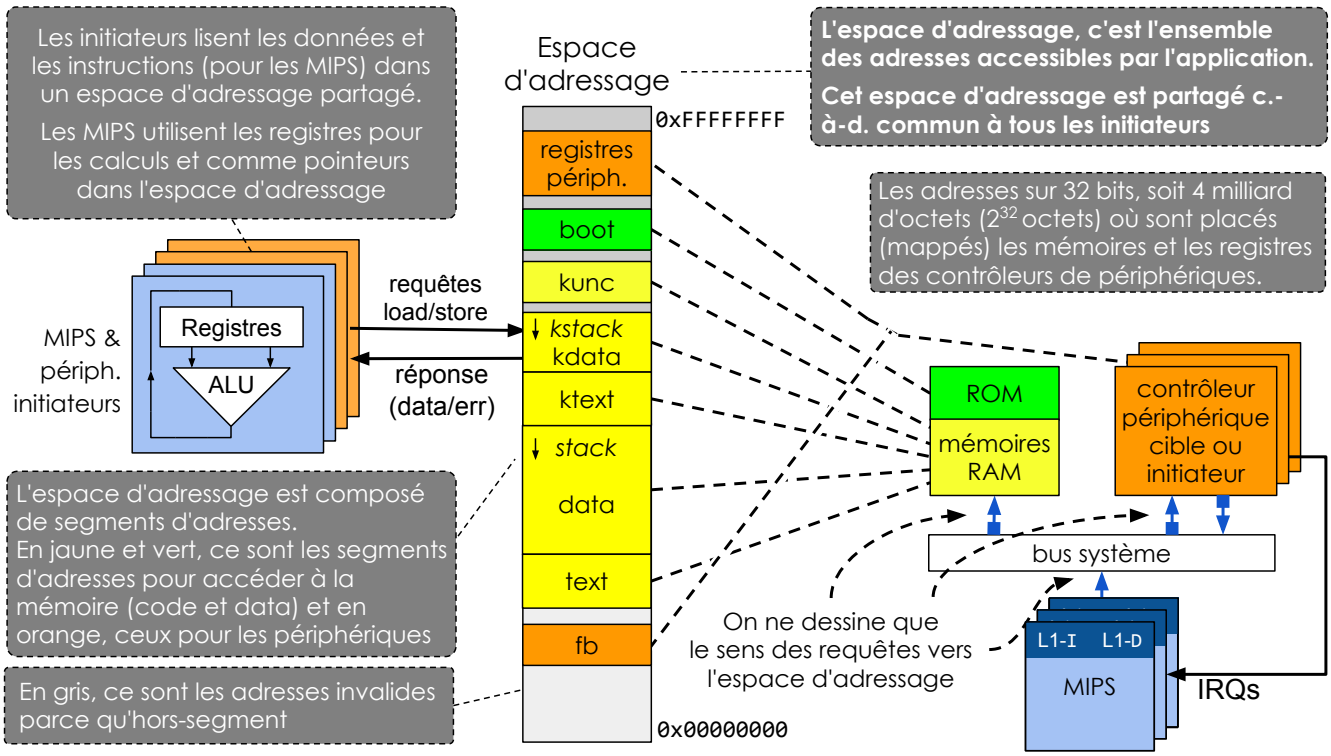
Le SoC Almo1

SoC - System-on-Chip - almo1

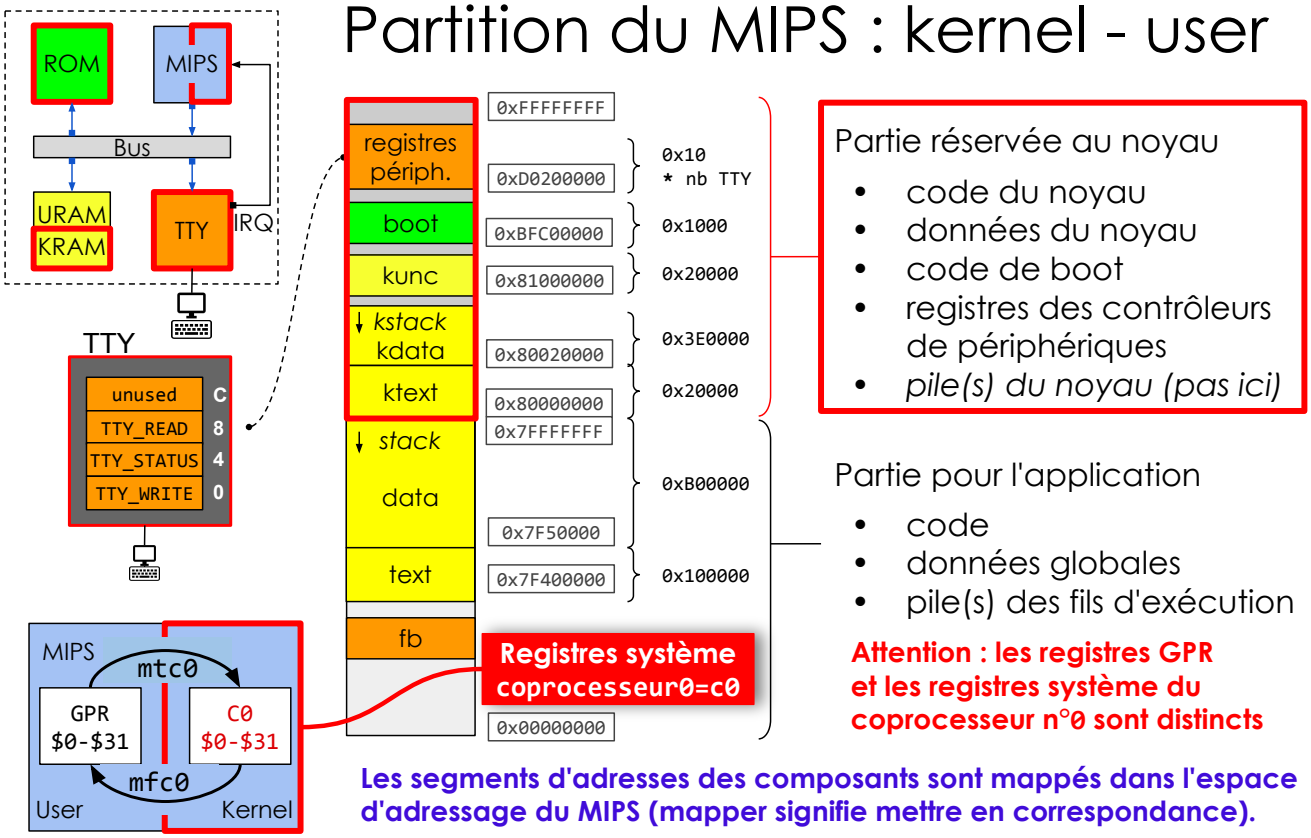


Partition des ressources
Kernel - User

Espace d'adressage du MIPS



Partition du MIPS : kernel - user



Partie réservée au noyau

- code du noyau
- données du noyau
- code de boot
- registres des contrôleurs de périphériques
- *pile(s) du noyau (pas ici)*

Partie pour l'application

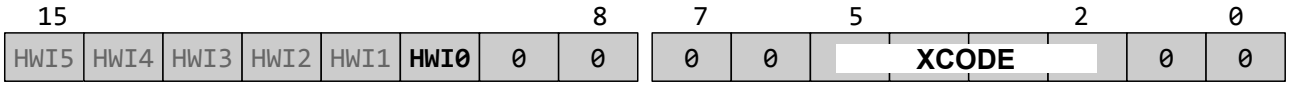
- code
- données globales
- pile(s) des fils d'exécution

Attention : les registres GPR et les registres système du coprocesseur n°0 sont distincts

Les segments d'adresses des composants sont mappés dans l'espace d'adressage du MIPS (mapper signifie mettre en correspondance).

Registres système : Status, Cause, EPC

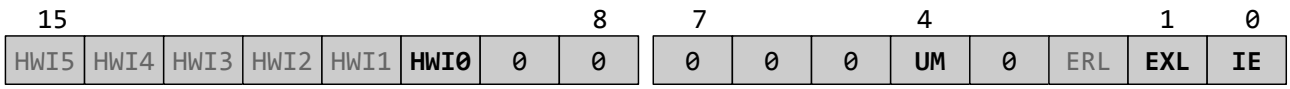
Le registre `c0_cause ($13)` contient la cause d'entrée dans le noyau (si IRQ, syscall ou except)



Etat des IRQ du MIPS

`0000` → IRQ ; `1000` → syscall ; `autre` → Exception

Le registre `c0_sr ($12)` contient le mode d'exécution du MIPS et les autorisations d'IRQ



Enable des IRQ du MIPS

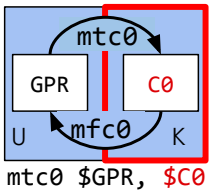
User Mode

EXcept Level

IRQ enable

Le registre `c0_epc ($14)` contient l'adresse de retour si c'est une IRQ ou l'adresse de l'instruction courante pour syscall et toutes les exceptions

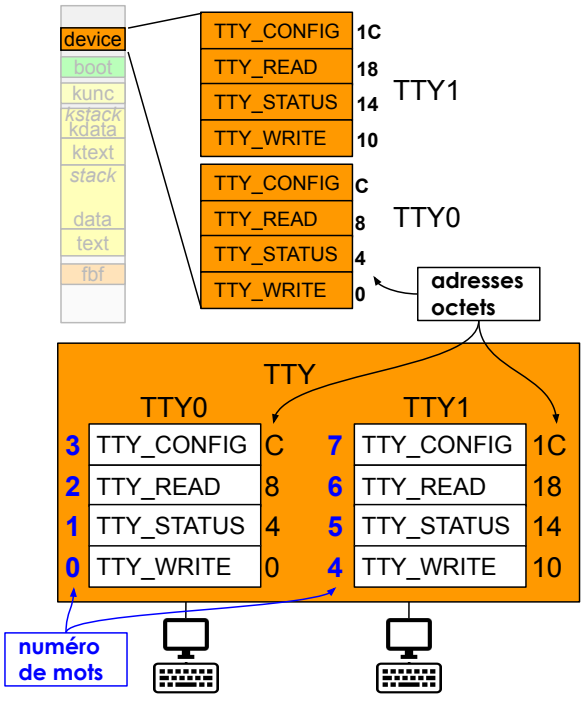
Le registre **`c0_cause`** du **MIPS** joue un rôle semblable au registre **`ICU_STATE`** de l'**ICU** mais pour les **IRQ** du **MIPS** et le registre **`c0_sr`** du **MIPS** joue un rôle semblable au registre **`ICU_MASK`** de l'**ICU.**




```
mtc0 $GPR, $C0
mfc0 $GPR, $C0
```

Contrôleurs de périphériques
Cibles et Initiateurs

Contrôleur de terminaux TTY



Tous les registres sont alignés sur des mots,
chaque terminal utilise un segment de 4 mots.

Pour chaque terminal

- `TTY_WRITE` 1 mot en écriture seule, le caractère ascii est mis dans l'octet de poids faible
  → sortie vers l'écran
- `TTY_STATUS` 1 mot en lecture seule, ≠ 0 s'il y a un caractère ascii en attente dans TTY_READ
- `TTY_READ` 1 mot en lecture seule, le caractère ascii tapé est dans l'octet de poids faible
  lire TTY_READ acquitte l'IRQ du TTY concerné
- `TTY_CONFIG` inutilisé dans cette version, mais permet la configuration p. ex. du débit d'échange avec le terminal externe.

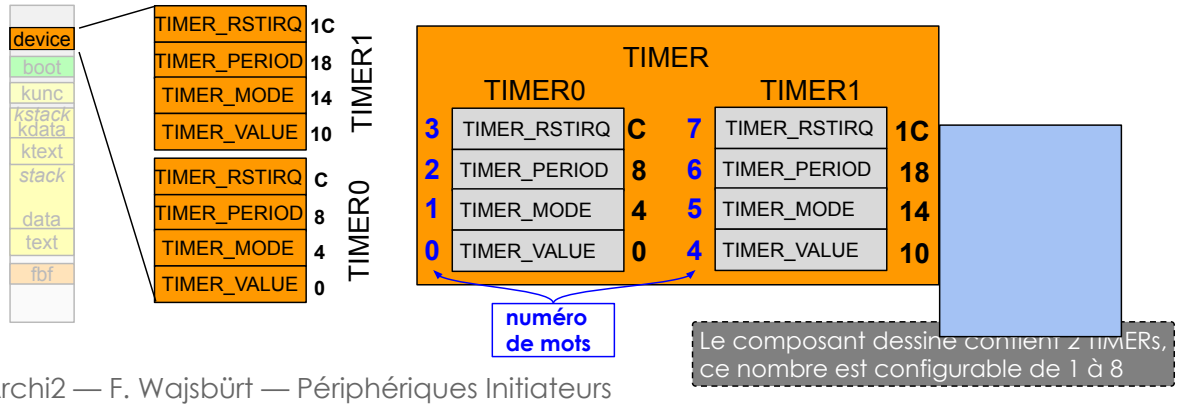
Chaque TTY lève une IRQ si un caractère est reçu par le TTY donc si son STATUS est ≠ de 0 (n TTY → n IRQ)

Le composant dessiné contient 2 TTYs, ce nombre est configurable de 1 à 4

Contrôleur TIMER

Le TIMER contient des compteurs de cycles qui peuvent lever des interruptions périodiques.
C'est un périphérique cible contrôlé par des accès en lecture / écriture dans ses registres.

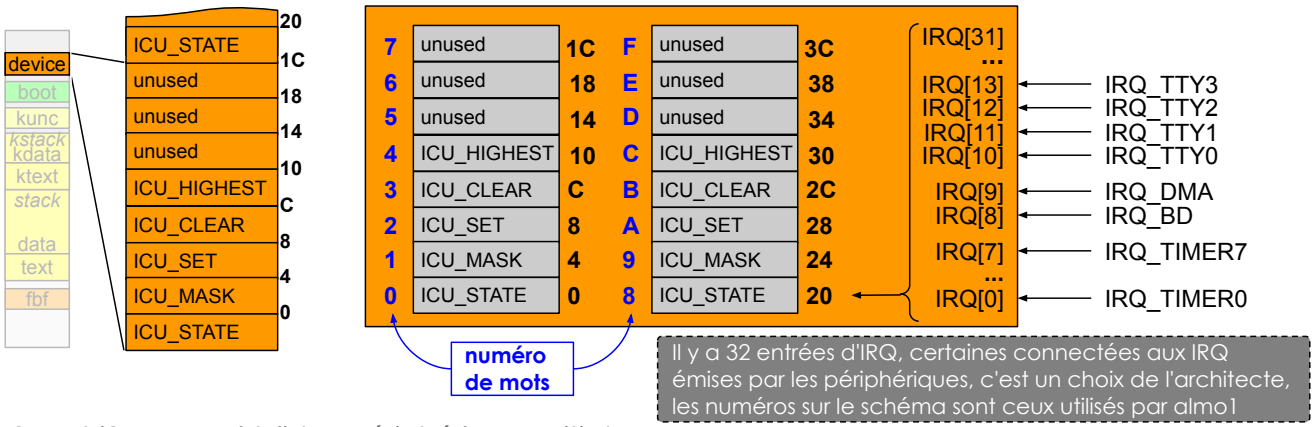
- TIMER_VALUE (lecture / écriture) +1 à chaque cycle
- TIMER_MODE (écriture seule) configure le mode de fonctionnement
  - Bit 0 : 1 → timer en marche (décompte) ; 0 → timer arrêté
  - Bit 1 : 0 → pas d'IRQ quand le compteur atteint 0
- TIMER_PERIOD (écriture seule) période demandée entre 2 IRQ
- TIMER_RESETIRQ (écriture seule) **écrire n'importe quoi à cette adresse acquitte l'IRQ**



Contrôleur ICU : Interrupt Controller Unit

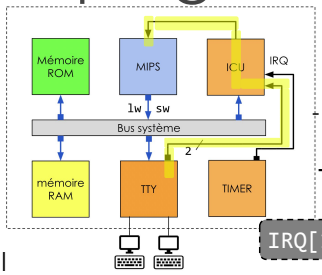
L'ICU est un concentrateur et un routeur de signaux d'IRQ. Chaque IRQ peut être masquée.
Il y a autant de jeu de registres que de MIPS (ici 2), chaque MIPS sélectionne les IRQ qu'il souhaite

- ICU_STATE (lecture seule) état des lignes IRQ (identique pour toutes les instances)
- ICU_MASK (lecture seule) masques des lignes IRQ (sélection des IRQ désirées par MIPS)
- ICU_CLEAR (écriture seule) commande de mise à 0 des masques d'IRQ
- ICU_SET (écriture seule) commande de mise à 1 des masques d'IRQ
- ICU_HIGHEST (lecture seule) **numéro de la ligne IRQ active et non masquée la plus prioritaire**
  Ici, c'est l'IRQ active dont l'index est le plus petit



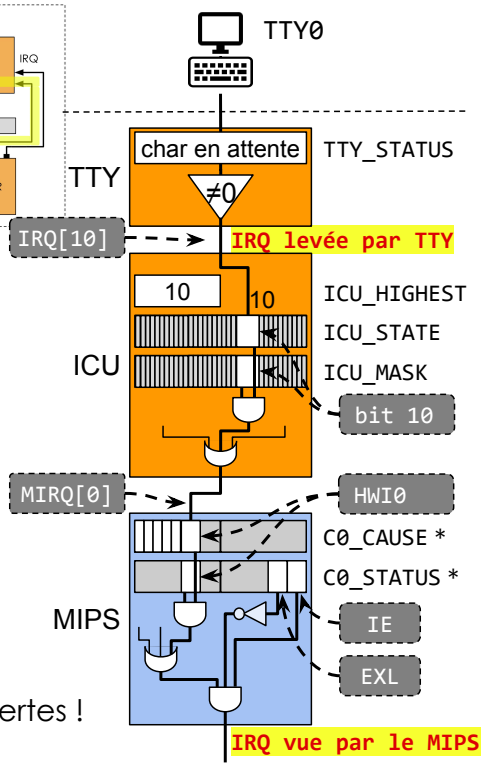
Niveau de masquage des IRQ

Une IRQ est émise par un contrôleur de périphérique peut être masquée par le noyau lorsqu'il exécute du code *critique*



Lors d'une frappe du clavier `TTY0` :

→ Le registre `TTY_STATUS` de `TTY0` devient non nul
→ Le contrôleur de `TTY` lève son `IRQ`
→ Le signal entre par la pin `IRQ[10]` de l'`ICU` (pour le SoC des TP)
→ Le bit `10` de `ICU_STATE` passe à `1`
→ Si c'est la seule `IRQ`, le registre `ICU_HIGHTEST` reçoit `10`
→ l'`ICU` fait un `AND` entre les bits `10` de `ICU_STATE` et `ICU_MASK`
→ puis un `OU` avec toutes les autres sorties de `AND` (ici non dessinés)
→ L'`IRQ` en sortie de l'`ICU` entre dans le MIPS (`MIRQ[0]`)
→ Le bit `HWI0` du registre de cause `C0_CAUSE` passe à `1`
→ Le MIPS fait un `AND` entre les bits `HWI0` de `C0_CAUSE` et `C0_STATUS`
→ puis un `OU` avec toutes les autres sorties de `AND` (ici non dessiné)
→ enfin on fait un `AND` avec les bits `IE` et `not EXL` de `C0_STATUS`.

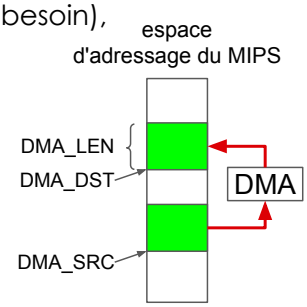


Pour voir une `IRQ`, il faut que toutes les « vannes » soient ouvertes !

* `c0_cause` et `c0_status` sont détaillés plus loin

Contrôleur DMA : Direct Memory Access

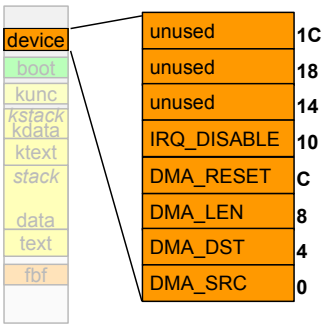
Le DMA copie la mémoire à partir de l'adr. DMA_SRC vers l'adr. DMA_DST de DMA_LEN octets.
Dans l'ordre, on commence par écrire les adresses SRC, DST et IRQ_DISABLE (si besoin),
puis on écrit LEN, ce qui provoque le démarrage de la copie par le DMA

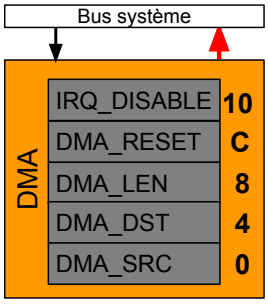


- DMA_IRQ_**DISABLE** (lecture/écriture) masquage de la ligne IRQ
- DMA_RESET (écriture seule) acquittement de la ligne IRQ
- DMA_LEN (écriture/lecture) taille en octets à déplacer
- DMA_DST (écriture seule) adresse de destination
- DMA_SRC (écriture seule) adresse source

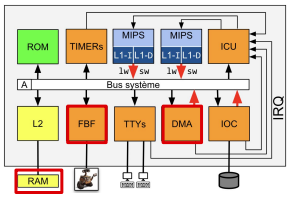
A la fin de l'opération,
le DMA lève un interruption
et LEN contient le nombre
d'octet non écrits.

S'il est différent de 0,
c'est qu'il y a une erreur.





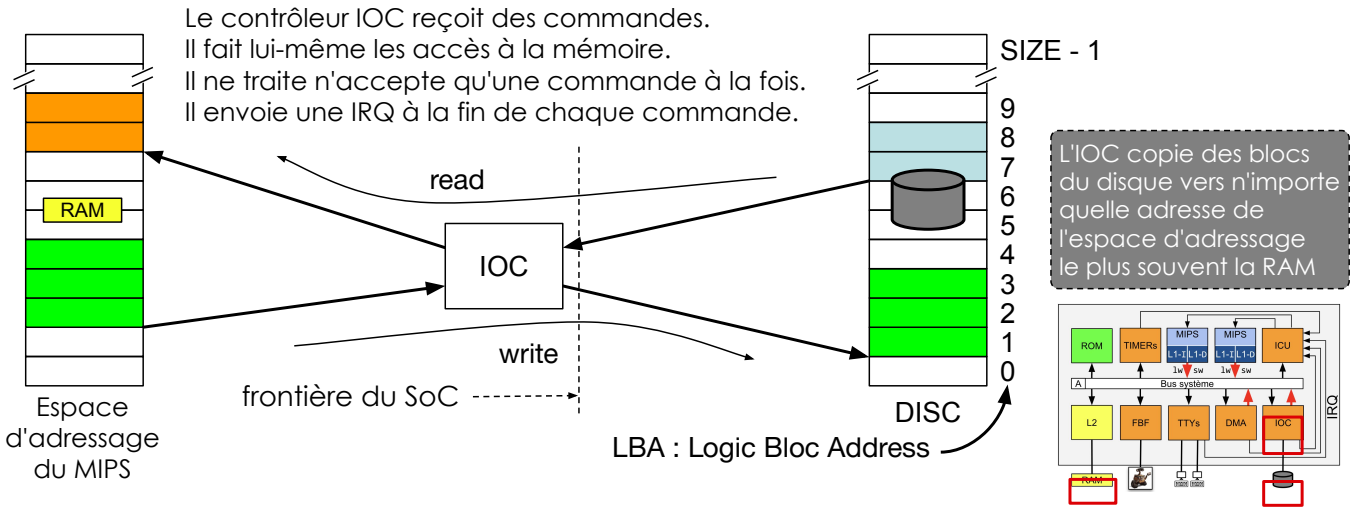
Le DMA peut copier
n'importe quelle adresse
RAM → RAM ou RAM → FBF



Contrôleur de disque IOC : In Out Controller

Le contrôleur de disque permet de lire (read) ou d'écrire (write) le disque.

- Les échanges se font par blocs. La taille d'un Bloc est une caractéristique intrinsèque du disque, c'est typiquement 512 octets (BLOCK_SIZE)
- Les opérations sont plus ou moins longues en fonction de la nature du disque.
  Sur un disque mécanique, il faut déplacer une tête de lecture avant de lire ou d'écrire, cela peut être long. Sur un disque flash, il n'y a pas cette latence.



Contrôleur de disque IOC : In Out Controller

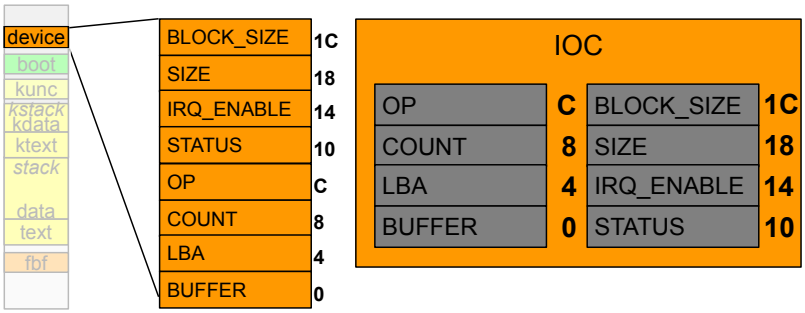
Le composant IOC est aussi appelé Block Device parce qu'il déplace des blocs du disque.

Dans l'ordre, on commence par écrire : BUFFER, LBA, COUNT et IRQ_ENABLE (si besoin),
puis on écrit OP, ce qui provoque le démarrage de l'IOC

• BLOCK_SIZE	(lecture seule)	taille d'un bloc en octets
• SIZE	(lecture seule)	taille du disque en bloc
• IRQ_ENABLE	(lecture/écriture)	masquage de la ligne IRQ
• STATUS	(lecture)	état de l'IOC en fin d'opération (acquitte IRQ)
• OP	(écriture seule)	sens de la transaction (read du disque ou write du disque)
• COUNT	(écriture/lecture)	taille en blocs à déplacer
• LBA	(écriture seule)	adresse de destination (en bloc)
• BUFFER	(écriture seule)	adresse source (adr. octets alignée sur un bloc)

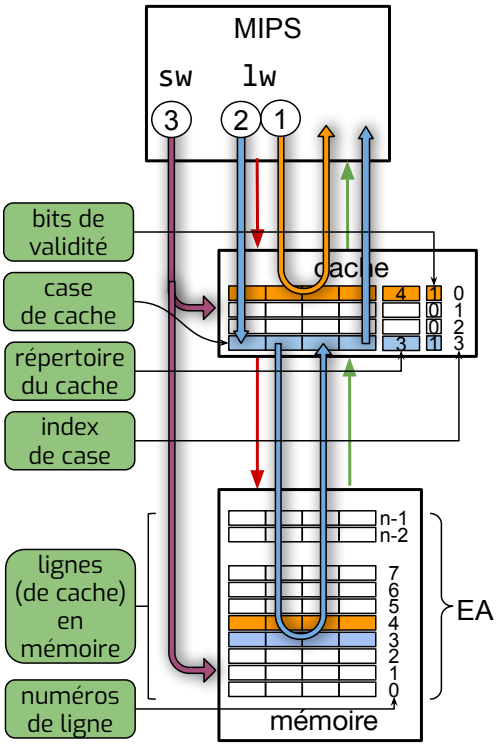
A la fin de l'opération, l'IOC lève une interruption et COUNT contient le nombre de BLOCK non écrits.

S'il est ≠ 0, c'est qu'il y a une erreur.



Cohérence des caches L1

Principe de fonctionnement du cache L1



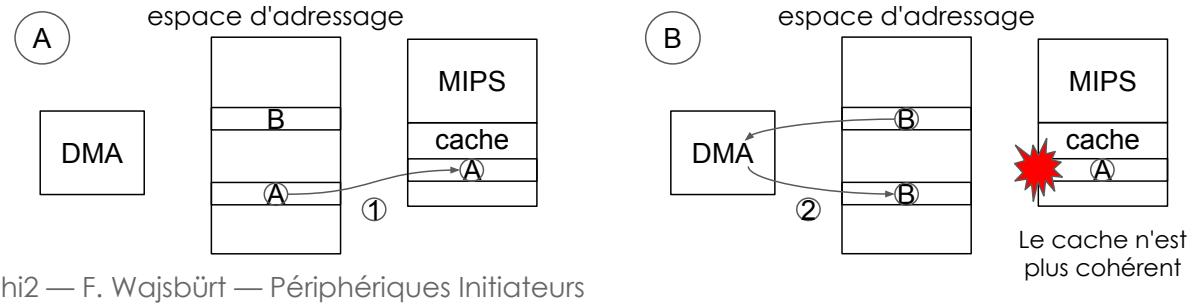
① Lecture d'une donnée déjà présente dans le cache
- Le MIPS exécute une instruction `lw`
- Le cache détermine le numéro de ligne où se trouve la donnée à partir de son adresse
- Et la ligne est déjà dans le cache
- La donnée est lue dans le cache et rendue au MIPS

② Lecture d'une donnée absente du cache
- Le MIPS exécute une instruction `lw`
- Le cache détermine le numéro de ligne
- Mais la ligne est absente du cache
- Le MIPS est gelé
- La ligne est demandée à la mémoire
- La ligne est rangée dans le cache
- La donnée est lue dans le cache et rendue au MIPS

③ Écriture d'une donnée : politique ***write through*** *cf.* slide 27
- Le MIPS exécute une instruction `sw`
- Le cache détermine le numéro de ligne où se trouve la donnée à partir de son adresse
- Si la ligne est déjà dans le cache, elle est mise à jour
- La donnée est envoyée vers la mémoire

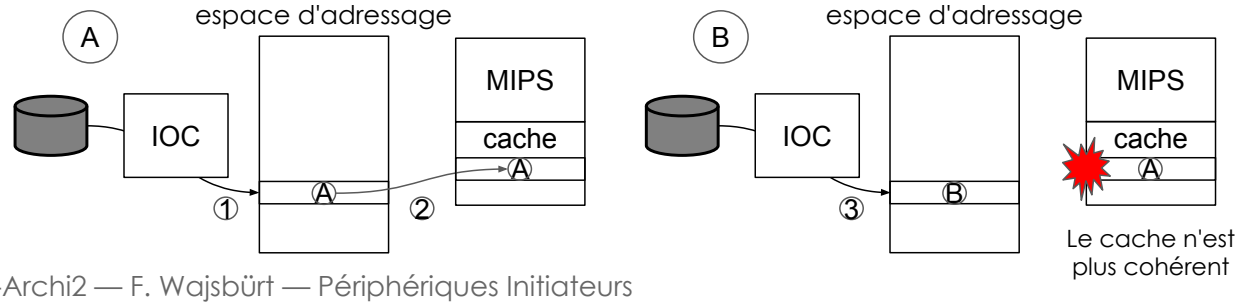
Problème de l'absence de cohérence

- Les caches L1 contiennent des copies de lignes de caches de la mémoire.
- Le cache est cohérent s'il contient des copies à jour, c.-à-d. les mêmes qu'en mémoire
- Si la plateforme ne contient d'un seul MIPS et qu'il est seul à faire des accès dans la mémoire alors les caches des instructions (`I`) et des données (`D`) sont toujours à jour,
  - Le cache `I` parce qu'il est en lecture seulement
  - Le cache `D` parce qu'il utilise une politique **write through** (écriture systématique en RAM)
- **Si le MIPS n'est pas le seul à faire des écritures en mémoire, il y a un problème de cohérence.**
  - Ⓐ Après ① le cache a la copie d'une ligne Ⓐ dans l'une de ses cases.
  - Ⓑ Si le DMA modifie la ligne en mémoire par ②, **alors la copie de la ligne présente dans le cache n'est plus à jour**, et si le MIPS lit cette ligne, il voit l'ancienne valeur Ⓐ



Problème de l'absence de cohérence

- C'est le même problème avec le contrôleur de disque (IOC)
  - Ⓐ Après les étapes ①② le cache a une copie de la ligne Ⓐ dans l'une de ses cases
  - Ⓑ Si l'IOC modifie la ligne en mémoire par ③, **alors la copie de la ligne présente dans le cache n'est plus à jour**, et si le MIPS lit cette ligne, il voit l'ancienne valeur Ⓐ
- Pour être sûr que le code (Application ou Kernel) utilise les données à jour il est absolument nécessaire de garantir la cohérence des copies de données
- Cette cohérence peut être
  - gérée automatiquement par le matériel, donc sans intervention du logiciel
  - gérée explicitement par le logiciel à chaque fois qu'il sait qu'il y a un risque



Almo1 : pas de cohérence automatique

Solution matérielle par suppression du problème

- Ne pas utiliser le cache pour les données partagées entre initiateurs (processeurs ou périphériques) → segment KUNCached

Solutions logicielles

- Utiliser l'instruction `cache 17, imm($rs)` qui invalide la ligne dont l'adresse est `$rs + imm`, si celle-ci est dans le cache

  17 est une opération qui demande l'invalidation du cache L1 (il existe d'autres opérations possibles mais pas forcément implémentées dans almo-1
  https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf page 85

- Utiliser l'instruction `ll` au lieu de `lw` parce que `ll` n'est pas caché dans cette version

```
int uncached_load (volatile int * addr);
```

Une solution pour résoudre un problème est de le supprimer, c'est radical mais ce n'est pas toujours possible ou efficace

Parcours du buffer partagé et exécution de l'instruction `cache` sur une adresse de chaque ligne, c'est assez long…

```
void dcache_buf_invalidate(void *buf, unsigned size) {
      while (size > 0) {
            cache 17, 0(buf)
            size -= cache_line_size
            buf += cache_line_size
      }
}
```

```
uncached_load:
    ll   $2,  0($4)  // ll is an UNCACHED LOAD
    jr   $31
```


Conséquence de l'absence de cohérence

On ne peut pas exécuter l'application utilisateur sur plusieurs MIPS parce que les structures de données du noyau sont accédées par tous les threads quel que soit le MIPS sur lequel il s'exécute.

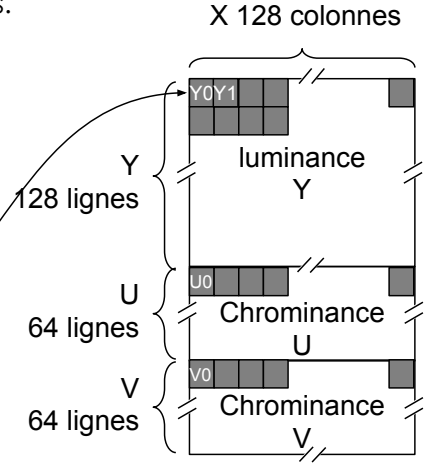
Si un thread attend le changement d'état d'une variable du noyau fait par un thread s'exécutant sur un autre MIPS, il ne peut pas utiliser son cache puisque le cache ne sera pas mise à jour lorsque que la variable changera

Sauf si on n'utilise pas le cache pour les données du noyau !
Et ça, c'est vraiment trop cher sur les performances…

Contrôleur Vidéo

Périphérique FB : Frame Buffer

- Le composant Frame Buffer est un périphérique **cible**.
  - C'est une mémoire de pixels accessible en écriture (*et en lecture*)
  - Les valeurs de pixels écrit dans cette mémoire sont envoyées vers l'écran.
  - Sur la plateforme l'image est configurable : ici 128 x 128 pixels.
- Chaque pixel occupe 2 octets avec un codage YUV422
  - 1 octet pour la luminance (intensité lumineuse) Y
  - 1 octet pour la chrominance (couleur) U et V.
- YUV422 signifie que chaque pixel a une valeur propre de Y mais chaque valeur de chrominance est commune à 2 pixels
  Il existe aussi le YUV420 où la chrominance est pour 4 pixels
- L'image occupe 128 x 128 x 2 = 32 kiB = 16kiB + 16KiB
  - le 1er octet est la luminance du pixel (0,0) [Y0]
  - le 2nd octet est le pixel (1,0) [Y1], etc. jusqu'au pixel (127,0)
  - puis l'octet 128 contient le pixel (0,1) , etc.

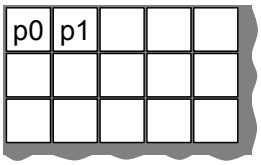


- Les tableaux de chrominance sont situés après
  - Le premier octet du premier tableau contient la chrominance U des pixels Y0 et Y1.
  - Le premier octet du second tableau contient la chrominance V des pixels Y0 et Y1.

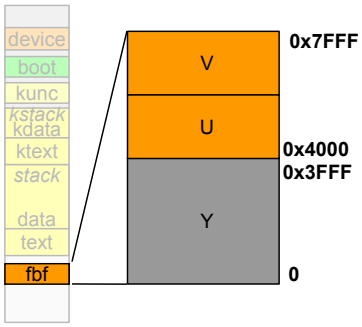
Périphérique FB : Frame Buffer

Pour chaque pixel, l'écran recalcule les composantes Rouge, Verte et Bleue

- Pour le pixel p0
  - `R = (y0 - 0.00004*u0 + 1.140*v0)`
  - `G = (y0 - 0.395*u0 - 0.581*v0)`
  - `B = (y0 + 2.032*u0 - 0.0005*v0)`
- Pour le pixel p1
  - `r = (y1 - 0.00004*u0 + 1.140*v0)`
  - `v = (y1 - 0.395*u0 - 0.581*v0)`
  - `b = (y1 + 2.032*u0 - 0.0005*v0)`



p0 et p1 partagent la même chrominance

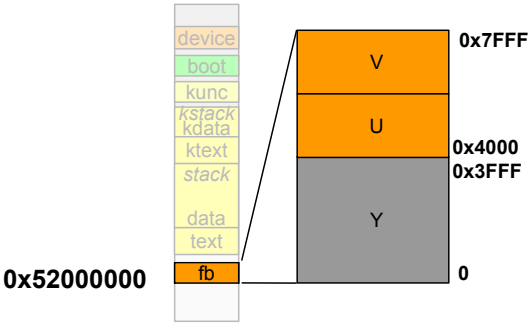


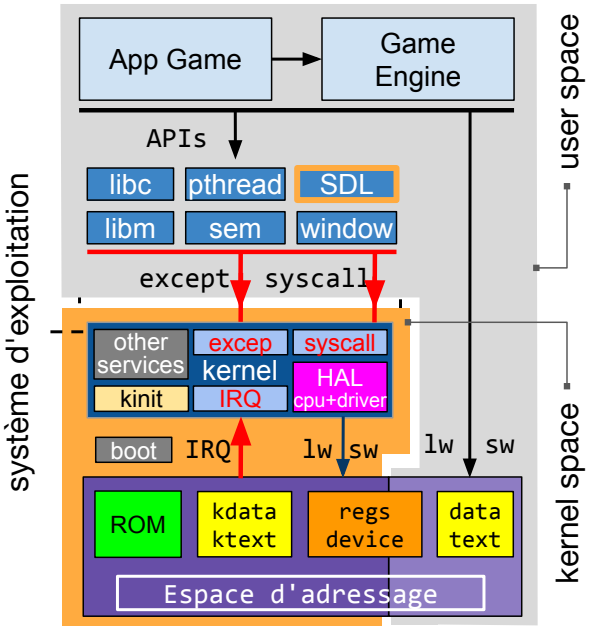
L'application manipule naturellement le RGB, il faudra calculer les YUV (on utilise un codage en virgule fixe pour limiter le temps de calcul)

- `Y0 = 0.299*R0 + 0.587*G0 + 0.114*B0`
  `Y1 = 0.299*R1 + 0.587*G1 + 0.114*B1`
- `U0 = (-0.14713*(R0+R1) - 0.28886*(G0+G1) + 0.436*(B0+B1))/2`
- `V0 = (0.615*(R0+R1) - 0.51499*(G0+G1) - 0.10001*(B0+B1))/2`

La gestion du FB est réalisée au niveau USER

La manipulation des pixels est très fréquente.
Il n'est pas raisonnable de faire des syscalls pour chaque pixel.
Pour régler ce problème, on donne la mémoire du frame buffer à une application





Affichage d'une image en séquentiel

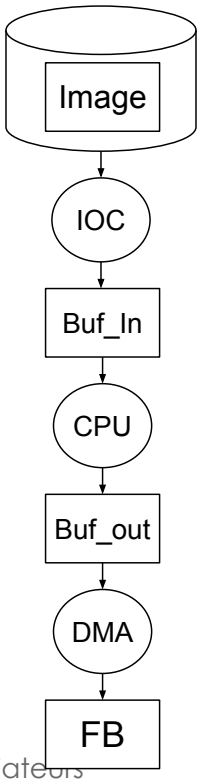
On veut afficher une image après l'avoir transformée.

Les 3 composants initiateurs vont travailler l'un après l'autre.

Dans le programme de test, nous faisons ces 3 étapes en boucle autant de fois que d'images sur le disque.

Faire N fois
commander IOC
Transformer l'image
commander DMA

Il faut attendre qu'une étape se termine avant de passer à la suivante.



Le disque contient l'image

L'IOC fait le transfert en mémoire

Buffer en mémoire

Transformation

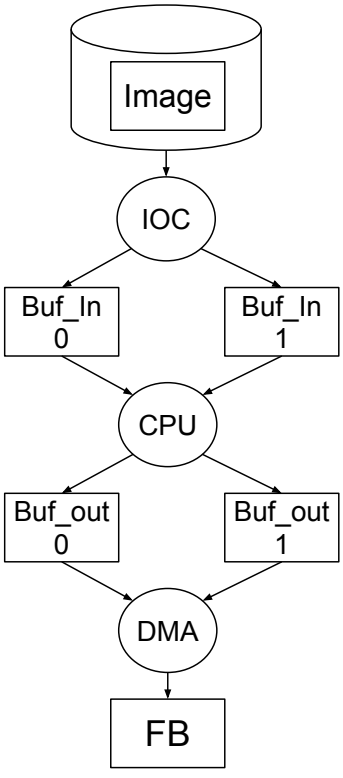
Buffer en mémoire

Le DMA transfer l'image dans le FB

Fonctionnement en pipeline

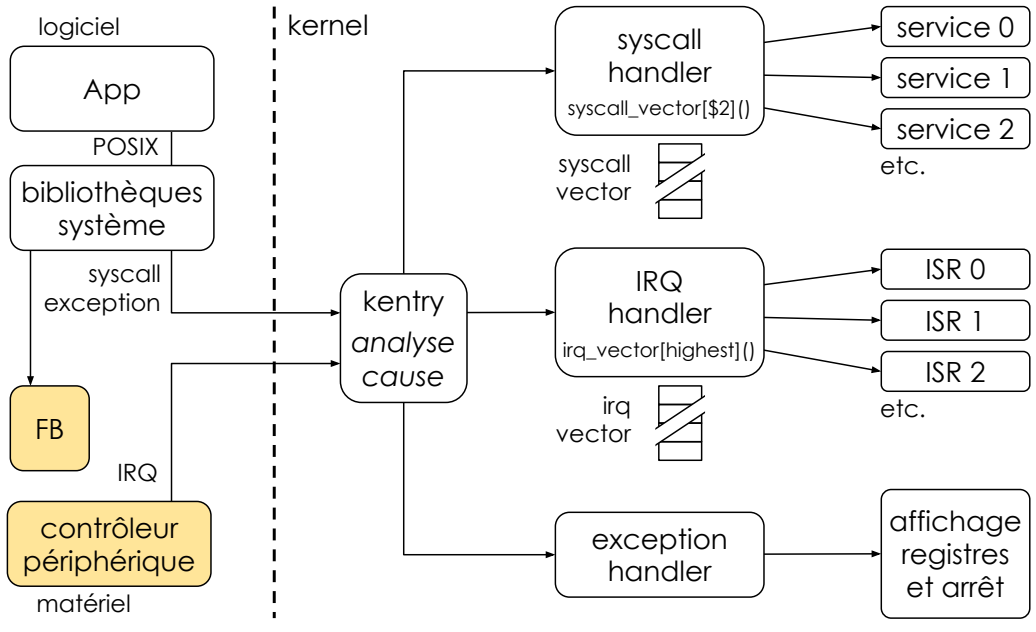
En utilisant 2 jeux de buffer en alternance on peut faire travailler en parallèle les trois composants : IOC, CPU et DMA.

Lec disk → Buf_In 0	Buf_In 0 → cpu → Buf_out 0	Buf_out 0 → dma → FB		
	Lec disk → Buf_In 1	Buf_In 1 → cpu → Buf_out 1	Buf_out 1 → dma → FB	
		Lec disk → Buf_In 0	Buf_In 0 → cpu → Buf_out 0	Buf_out 0 → dma → FB



Introduction SDL

Illustration avec le kernel



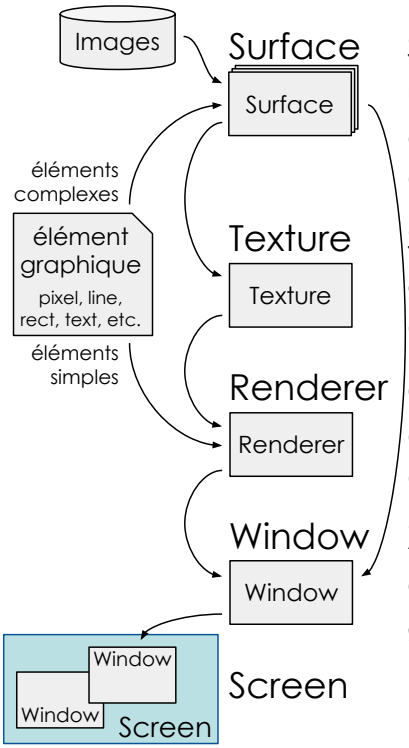
- Création des threads
- Scheduler des threads
- Synchronisation des threads
- Pilote de périphériques
- Allocation de mémoire

- *Création des processus*
- *Système de fichiers*
- *Accès réseaux*

Introduction à SDL

- Simple DirectMedia Layer https://wiki.libsdl.org/SDL2
  - Objectif
    Fournir un accès aux opérations multimédias (graphismes 2D, son, événements) indépendant de la plateforme.
  - Mode d'exécution
    En grande partie en mode User sauf pour la réservation des ressources ou pour les événements (souris, clavier, etc.)
  - Fonctions de base SDL
    - Création d'une fenêtre
    - Graphismes 2D
    - Gestion des événements
    - Gestion du son

Surface → Texture → Render → Window → Screen



Structure avec une zone de mémoire contenant des pixels, utilisée pour le chargement d'images et la manipulation des éléments graphiques
Opérations exécutées par le CPU

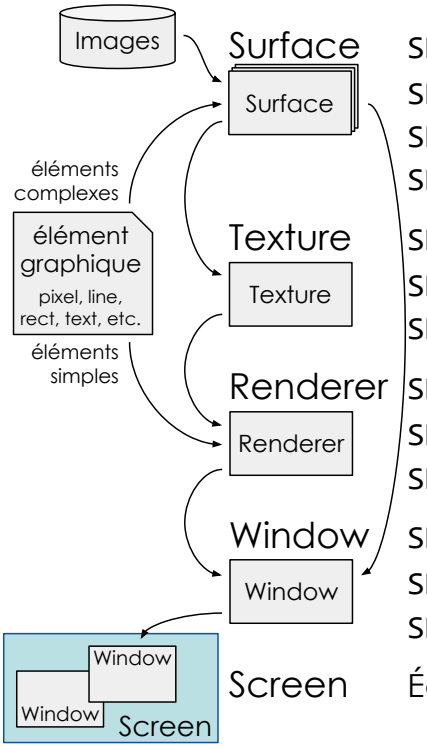
Structure utilisée pour stocker des images ou des éléments graphiques destinés à être créés dans un renderer.
Opérations généralement accélérées par le matériel.

objet proposant des opérations pour remplir un tampon de rendu temporaire avec textures ou éléments graphiques
Opérations généralement accélérées par le matériel

Zone dans l'écran
conteneur visuel pour le tampon de rendu d'un renderer
gestion des événements clics de souris ou entrées clavier

Écran dans lequel sont placées les Windows

Surface → Texture → Render → Window → Screen



Surface
- `SDL_CreateRGBSurface` : Crée une surface
- `SDL_LoadBMP` : Charge une image au format BMP
- `SDL_FillRect` : Remplit un rectangle avec une couleur
- `SDL_BlitSurface` : Copie une partie d'une surface sur une autre

Texture
- `SDL_CreateTexture` : Crée une texture vide
- `SDL_CreateTextureFromSurface` : Création à partir d'une surface
- `SDL_RenderDrawTexture` : dessin de texture sur un renderer

Renderer
- `SDL_CreateRenderer` : Création un renderer pour une window
- `SDL_RenderFillRect` : Remplit un rect. avec la couleur actuelle
- `SDL_RenderPresent` : Affiche le contenu du tampon de rendu

Window
- `SDL_CreateWindow` : Crée une nouvelle fenêtre
- `SDL_GetWindowSurface` : Récupère la surface associée
- `SDL_GetWindowID` : Récupère l'identifiant unique d'une fenêtre

Screen
- Écran dans lequel sont placées les Windows

SDL API minimaliste : fonctions

But : une app ko6 doit pouvoir s'exécuter sur SDL/Linux

Initialisation et Nettoyage

- `int SDL_Init( u32 flags)` : Initialise SDL (`flags SDL_INIT_VIDEO` : init vidéo & événements)
- `void SDL_Quit( void)` : Nettoie SDL et libère les ressources associées
- `char * SDL_GetError()` : renvoie un message correspondant à l'erreur de la dernière fonction

Création et Destruction

- `SDL_Window * SDL_CreateWindow( char *name, int x, int y, int w, int h, u32 flags)`
  Crée une nouvelle fenêtre SDL (`flags SDL_WINDOW_SHOWN`)
- `void SDL_DestroyWindow( SDL_Window *w)` : Détruit une fenêtre SDL et libère les ressources

Obtention de la Surface associée à une window

- `SDL_Surface *SDL_GetWindowSurface( SDL_Window * window)` : Récupère la surface

Opérations avec la Surface

- `u32 SDL_MapRGB( SDL_PixelFormat *format, u8 r, u8 g, u8 b)`
  Convertit les couleur `RGB` dans le format utilisée par la surface
- `int SDL_FillRect( SDL_Surface * dst, SDL_Rect * rect, u32 color)`
  Remplit un rectangle sur une surface avec une couleur spécifique.
- `int SDL_UpdateWindowSurface(SDL_Window * window)` MAJ une window à partir de sa surface.

SDL API minimaliste : structures

adaptation pour ko6

```
typedef struct SDL_PixelFormat {
    u32 yr, yg, yb;             ///< Y = yr*R + yg*G + yb*B (fixed virgule)
    u32 ur, ug, ub;             ///< U = ur*R + ug*G + ub*B (fixed virgule)
    u32 vr, vg, vb;             ///< V = vr*R + vg*G + vb*B (fixed virgule)
} SDL_PixelFormat;

typedef struct SDL_Surface {
    SDL_PixelFormat *format;    ///< R  used to convert RGB in pixel format
    int w, h;                   ///< R  dimension (should be same as window)
    void *pixels;               ///< RW pixel array (array of color)
} SDL_Surface;

typedef struct SDL_Rect {
    u32 x, y, w, h;             ///< position and dimension
} SDL_Rect;

typedef struct SDL_Window {
    int x, y, w, h;             ///< window position & dimension
    SDL Surface *surface;       ///< window's surface
} SDL_Window;

#define SDL_INIT_VIDEO 0
#define SDL_WINDOW_SHOWN 0
```


SDL API minimaliste : exemple

```
#include <SDL2/SDL.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    // création de la fenêtre et de la surface
    if (SDL_Init(SDL_INIT_VIDEO) < 0) { printf(" SDL_INIT_VIDEO %s\n", SDL_GetError()); return 1;}
    SDL_Window *window = SDL_CreateWindow( NULL, 0, 0, 800, 600, SDL_WINDOW_SHOWN);
    if (window == NULL) { printf("SDL_CreateWindow %s\n", SDL_GetError()); SDL_Quit(); return 1;}
    SDL_Surface *surface = SDL_GetWindowSurface(window);
    if (surface == NULL) { printf("SDL_GetWindowSurface %s\n", SDL_GetError());
        SDL_DestroyWindow(window);
        SDL_Quit();
        return 1;
    }
    // Dessin d'un carré bleu au centre de la surface
    SDL_Rect blueSquare = {300, 200, 200, 200}; // Position et taille du carrÃ©
    Uint32 blue = SDL_MapRGB(surface->format, 0, 0, 255); // Couleur bleue
    SDL_FillRect(surface, &blueSquare, blue);
    // Mise à  jour la fenêtre pour afficher les changements
    SDL_UpdateWindowSurface(window);

    sleep (5); // mise en pause c'est juste pour cet exemple
    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

Gestion des événements

Le principe d'un jeu utilisant SDL, c'est de créer une boucle sans fin

```
tant que (running)
    event = événement venant des périphériques d'entrée
     switch (event)
        case EVENT1 : action1;
        case EVENT2 : action2;
        …
    calculer la nouvelle frame
    afficher la frame
    attendre (pour réduire le nombre de frames par seconde)
```


En TME

L'idée est de manipuler la plateforme complète :

1. Le composant IOC (Block Device) dans le code lit une image depuis le disque
2. Le CPU déplace cette image (pourrait la mettre en couleur ou la modifier)
3. Le DMA déplace l'image modifiée et la place sur le frame buffer FBF

Et de faire ça en boucle :

a. avec un seul processeur sans paralléliser les étapes 1. 2. et 3.
b. en parallélisant l'IOC, le CPU et le DMA

Après, je vais vous proposer de tester un programme utilisant la SDL very lite