

INDEX

DOC → [Config] [MIPS U] [MIPS K] [markdown]
[CR.md]

COURS → [1 (+code)] (+outils)] [2] [3] [4] [5] [6] [7] [8]
[9]

TME → [1] [2] [3] [4] [5] [6] [7] [8] [9]

CODE → [gcc + soc] [1] [2] [3] [4] [5] [6] [7] [8] [9]

A. Questions
A.1. Questions générales
A.2. Questions sur l'implémentation
B. Travaux pratiques
Questions
Etat du code par une lecture directe du registre READ du TTY par ...
Le problème et une solution possible
Mise en place d'une FIFO entre l'isr du TTY et la fonction `tty_read()`
Utilisation de la FIFO

3 - Gestion des Threads

La majorité des réponses aux questions sont dans le cours, c'est voulu. Les questions suivent à peu près l'ordre du cours, elles sont simples, mais vous avez quand même besoin de comprendre le cours pour y répondre :-). Quand une question vous demande si quelque chose est vrai ou faux, ne répondez pas juste "oui" ou "non", mais justifiez vos réponses avec une petite phrase. Le but de ces questions est d'évaluer vos connaissances, donc plus vous êtes précis, mieux c'est. Vous avez un corrigé que vous devez consulter pour vous autocorriger, mais pour qu'il soit utile, lisez-le après avoir cherché vous-même les réponses. Dans certains cas, ce ne sera pas simple, mais tentez quand même une réponse, même si vous savez que c'est faux, car ce sera plus simple de comprendre la réponse.

A. Questions

A.1. Questions générales

- Dites-en une phrase ce qu'est un processus informatique (selon Wikipédia)
 - Un processus est un programme en cours d'exécution. Dans cette définition, le programme c'est seulement le fichier qui contient le code.
 - Pour le système d'exploitation, c'est un conteneur de ressources permettant l'exécution d'un programme. C'est-à-dire, l'ensemble des ressources nécessaires : un espace d'adressage pour le code et les données, des fichiers, des ports réseaux et des threads, etc. Dans cette UE, on ne voit pas vraiment la notion de processus puisqu'on a une seule application, pas d'espace d'adressage virtuel, pas de système de fichiers, on est donc dans un cas très simplifié.
- Est-ce qu'un processus utilisateur s'exécute toujours dans le mode utilisateur du MIPS ?
 - Non, la majorité du code s'exécute en mode utilisateur (user), mais lorsqu'il fait un appel système, il entre dans le noyau pour exécuter les fonctions rendant le service avec le droit du noyau, et c'est toujours le processus utilisateur qui s'exécute, mais avec les droits du noyau.
- Nous avons vu qu'un processus utilisateur peut faire des appels système, c'est-à-dire demander des services au noyau du système d'exploitation. Est-ce qu'un processus peut faire des interruptions et des exceptions ?
 - Non pour les interruptions, les demandes d'interruption (IRQ pour Interrupt ReQuest) sont faites par les périphériques grâce à des signaux électriques binaires (2 états). Ces demandes ne peuvent pas être faites directement par le code de l'utilisateur. Toutefois, une IRQ est la conséquence d'une commande ou d'une configuration faite par le programme, alors on pourrait dire que les IRQ sont provoquées par les programmes.
 - Oui pour les exceptions, une exception est toujours la conséquence de l'exécution d'une instruction que le processeur ne peut pas ou ne sait pas faire.
- Un processus dispose d'un espace d'adressage pour s'exécuter, qu'y met-il ?
 - Le processus y met son code et ses données globales et, nous le verrons plus tard, ses données dynamiques dans des segments obtenus par `malloc()` ou `mmap()`. Le processus y met aussi les piles d'exécution de ces threads. Dans l'implémentation actuelle du système, ces piles sont dans les variables globales, mais dans un vrai système, elles seraient allouées dynamiquement.
- Dans un fichier exécutable, avant qu'il ne soit chargé en mémoire, on trouve le code du programme et les données globales. Est-ce qu'il y a aussi les piles d'exécution des threads ? Justifiez votre réponse.
 - Non, en principe non, puisque les piles sont créées à la volée à chaque création des threads. Mais, dans la version actuelle du système, qui n'a pas encore de service de mémoire dynamique, les piles des threads sont dans des variables globales de type `struct thread_s`, alors on peut se demander si elles sont dans le fichier. La réponse est non, parce que les variables globales thread de type `struct thread_s` ne sont pas initialisées. Elles sont donc dans une section `BSS` qui est définie (position et taille) dans le fichier, mais elle n'occupe pas de place pour les données puisque c'est le programme qui fait la mise à 0 du segment.
- Un thread de processus informatique représente un fil d'exécution de ce processus. Il est défini par une pile d'exécution pour ses fonctions, un état des registres du processeur et des propriétés comme un état d'exécution (RUNNING, READY, DEAD, et d'autres que nous verrons plus tard). Combien de threads a-t-on par processus au minimum et au maximum ?
 - On en a au moins 1 dont la fonction principale est `main()`. Le nombre maximum est défini dans le système d'exploitation pour notre cas, mais plus généralement, il est dépend de la quantité de mémoire disponible, car chaque thread utilise une pile qui peut être grande.
- Tous les threads d'un processus se partagent le même espace d'adressage, et donc le même code, les mêmes variables globales, les mêmes variables dynamiques (nous les verrons dans un prochain cours). Est-ce qu'ils se partagent aussi les piles ?
 - Non, chaque thread a sa propre pile. On peut se dire qu'ils partagent leur pile, si on imagine créer une variable `VA` locale dans la pile d'un thread `TA` et que l'on passe l'adresse de cette variable locale `VA` à un autre thread `TB`. C'est possible, mais ce n'est pas recommandé, car certains OS interdisent cette pratique. Pour faire communiquer deux threads, il faut passer par des variables globales. Nous verrons cela plus tard dans les prochaines semaines.
- Lorsque l'on crée un nouveau thread (un nouveau fil d'exécution du processus), il faut indiquer sa fonction principale, c'est-à-dire la fonction par laquelle qu'il doit exécuter. Est-ce que le nouveau thread pourra appeler d'autres fonctions ?

- Bien sûr, rien ne l'en empêche, c'est généralement le cas. Si vous appelez par exemple `printf()` dans votre thread, c'est un appel de fonction.
9. Est-ce qu'on peut créer deux threads avec la même fonction principale ?
- Bien sûr, plusieurs threads peuvent se partager le même code. Ils n'auront pas la même pile et donc il n'auront pas la même histoire.
10. Combien d'arguments la fonction principale d'un thread peut-elle prendre et de quel type ?
- Le prototype de la fonction principale d'un thread est imposé par la fonction `thread_create()`. Une fonction principale de thread prend un seul argument de type `void *` et elle rend un `void *`. Si on veut passer plusieurs arguments, il faut les mettre dans une structure et passer le pointeur sur cette structure.
 - Il y a quand même une exception pour le thread `main()` créé systématiquement au démarrage du processus. Comme vous le savez, la fonction `main()` prend jusqu'à 3 arguments : `int argc`, `char *argv[]` et un moins connu `char *arge[]` (aussi nommé `char *env[]`) qui contient les définitions de **variables d'environnement du shell**). Dans notre système `main()` n'a pas d'arguments, alors que normalement c'est grâce au shell que l'utilisateur peut définir les arguments `argc`, `argv` et `arge`, mais nous n'avons pas encore de shell. La fonction `main()` rend un `int` et non pas un `void *`.
11. Que se passe-t-il lorsqu'on sort de la fonction principale d'un thread ?
- Ça dépend de quel thread on parle, c'est soit le thread `main()` de l'application, soit un thread standard créé par l'application avec `thread_create()`.
 - Le thread `main()` est lancé par la fonction `_start()` qui est la toute première fonction de l'application. Quand on sort de `main()`, on retourne dans `_start()` qui doit stopper l'application en exécutant la fonction `exit()` avec la valeur de retour de `main()` en argument (un `int`).
 - Un thread standard est lancé par la fonction `thread_start()`. Cette fonction `thread_start()` lance la fonction principale de thread en lui donnant son argument (la fonction principale et son argument sont des arguments de `thread_start()`). Quand on sort de la fonction principale du thread, on revient dans `thread_start()`, laquelle va appeler `thread_exit()` avec la valeur de retour de la fonction principale du thread en argument (un `void *`).
12. L'exécution en temps partagé est un mécanisme permettant d'exécuter plusieurs threads à tour de rôle sur le même processeur. Comment s'appelle le service du noyau chargé du changement de thread ?
- C'est l'ordonnanceur ou, en anglais, le scheduler. Il s'appelle ainsi parce que son rôle est de donner le processeur à tous les threads de l'application en respectant une politique d'ordonnement.
13. La phase de changement de thread a une certaine durée, c'est un temps perdu du point de vue de l'application. Comment nomme-t-on cette phase pour indiquer que c'est un temps perdu ?
- C'est un *thread switching overhead cost*, ce qui signifie *frais de commutation*. C'est le temps que le noyau met pour sélectionner un nouveau thread (avec l'ordonnanceur), sauver le contexte du thread entrant et charger le contexte du thread entrant. Nous n'avons pas plusieurs processus dans notre application, cet *overhead* est donc assez court, car tous les threads partagent le même espace d'adressage, mais quand il y a plusieurs processus, le coût de changement de thread de 2 processus distincts est beaucoup plus cher, car il faut vider (*flush*) les caches, nous en parlerons au prochain cours.
14. Pour l'exécution en temps partagé, le noyau applique une politique, laquelle définit l'ordre d'exécution. Si les threads sont toujours prêts à être exécutés et que le noyau les exécute à tour de rôle de manière équitable, comment se nomme cette politique ?
- C'est une politique *round robin* ou *robin des bois*, ou à tour de rôle équitablement. Attention à ne pas la confondre avec la politique *fifo*, dans cette dernière ce que l'ordonnanceur, c'est le prochain thread entrant sera celui qui est sorti depuis le plus longtemps, ou dit autrement, quand un processeur sort (c.-à-d. perd le processeur), il sera le dernier à le regagner. Vous allez dire que c'est du *round robin*, mais non, dans la politique *fifo*, ce sont les threads eux-mêmes qui décident quand ils rendent le processeur, par un appel explicite à `thread_yield()` ou lorsqu'il demande une ressource indisponible. Il n'y a pas de recherche d'équité alors qu'avec la politique *round robin*, le noyau utilise un timer pour que chaque thread dispose du même temps d'exécution en imposant des `thread_yield()`.
15. Dans cette politique équitable, quelle est la fréquence type de changement de thread ? Donnez une justification.
- Ça dépend un peu de la fréquence du processeur. Il faut que la commutation soit assez rapide pour donner l'illusion du parallélisme (l'impression que tous les threads s'exécutent en même temps), mais pas trop à cause de l'*overhead* de changement de thread. La réponse est entre 10 et 100Hz. Plus la fréquence du processeur est élevée, plus la fréquence de commutation peut être rapide. À 1GHz, un processeur exécute 10 millions de cycles en 10ms (100Hz), si l'*overhead* de changement de thread est de 1000 cycles (un ordre de grandeur), l'*overhead* prend 0.01% du temps d'exécution du processeur, c'est négligeable.
16. Comment nomme-t-on la durée entre deux interruptions d'horloge ? Ici, c'est le temps d'une instance d'exécution d'un thread.
- C'est le **tick**. Un tick d'horloge est la durée entre deux IRQ du timer.
 - Dans l'état actuel du code, on fait une commutation de thread à chaque tick. Dans un système plus évolué, on a la notion de **quantum** qui correspond à un nombre fini de ticks, par exemple 1 quantum = 10 ticks. Ce quantum peut varier au cours du temps pour donner plus de temps à certains threads (au démarrage par exemple ou pour des tâches que l'on veut favoriser)
17. Le mécanisme de changement de thread (dont vous avez donné le nom précédemment) se déroule en 3 étapes, quelle que soit la politique suivie. Quelles sont ces étapes ?
- L'ordonnanceur réalise :
 - l'élection du thread entrant qu'il choisit parmi tous les threads prêts en suivant une politique explicite. Pour le *round robin*, l'élu sera celui qui attend depuis le plus longtemps.
 - La sauvegarde du contexte du thread sortant.
 - Le chargement du contexte du thread entrant.
18. Comment se nomme la fonction qui provoque la perte du processeur par le thread en cours au profit d'un nouveau thread ?
- C'est la fonction `thread_yield()`. `yield` signifie **cession**, le thread cède ou lâche le processeur.

19. Qu'est-ce qui provoque un changement de thread sans que le thread n'en fasse lui-même la demande ?

- C'est l'IRQ du timer pour respecter la politique *round robin*, mais pas seulement, on retire le processeur aux threads bloqués, parce qu'ils ont demandé une ressource au noyau, mais que cette ressource n'est pas disponible. Le thread peut aussi demander à rendre le processeur.

20. Dans le mécanisme de changement de thread, l'une des étapes est la sauvegarde du contexte, est-ce la même chose qu'un contexte de fonction ? Dites de quoi il est composé.

- Alors non, il faut vraiment faire attention au vocabulaire. Le contexte d'un thread et le contexte d'une fonction sont deux concepts très différents. Cela signifie que la question *Qu'est-ce qu'un contexte ?* n'a pas une seule réponse et pour être précis, il faut demander *contexte de quoi ?*.
- Le contexte d'une fonction est un segment d'adresse dans la pile d'exécution, dans lequel la fonction
 - sauvegarde la valeur des registres persistants afin des restaurer avant de retourner dans la fonction appelante ;
 - alloue ses variables locales ;
 - alloue la place pour les arguments des fonctions qu'elle appelle.
- Une fonction accède exclusivement à son propre contexte et à la partie des arguments du contexte de la fonction appelante.
- Le contexte d'un thread, c'est l'état des registres du processeur pendant que le thread s'exécute. Parmi les registres, il y a le registre `PC` (Program Counter) qui pointe vers l'instruction en cours d'exécution, le registre `SP` qui pointe sur la dernière case occupée dans la pile d'exécution du thread, le registre `CO_SR` (Status Register) qui indique essentiellement le mode d'exécution du MIPS et il y a tous les registres de travail du thread.

21. Où est sauvé le contexte d'un thread ? Que pouvez-vous dire de la fonction de sauvegarde ? (langage, prototype, valeur de retour, etc.)

- Pour notre système, c'est dans un tableau présent dans la structure de données du thread (`struct thread_s`).
- C'est une fonction en assembleur parce qu'elle est spécifique au processeur, on ne pourrait pas l'écrire en C.
- Elle prend en argument un pointeur vers le tableau de sauvegarde. C'est un prototype générique qui fait partie de la HAL (Hardware Abstraction Layer).
- Elle rend 1 quand elle vient juste de faire la sauvegarde du contexte du thread en cours.

22. Chaque thread dispose de sa propre pile d'exécution, doit-on aussi sauver la pile lors des changements de thread ?

- Non, elle reste en mémoire, mais lors des changements de thread, on change simplement le pointeur de pile, ainsi on change de pile.

23. Après qu'un thread a été élu et que son contexte a été chargé dans le processeur, donnez le nom de la fonction responsable du chargement et dites où elle retourne ? Attention, il y a deux cas. Vous avez une partie de la réponse dans le cours à partir du slide 23, et vous avez des commentaires dans le code de `kernel/kthread.c`. L'idée n'est pas de répondre de manière précise, mais de comprendre pourquoi il y a deux cas.

- C'est la fonction `thread_load()` qui se charge du chargement de la restauration du contexte du thread entrant (nouvellement élu).
- Quand on sort de la fonction `thread_load()`, il y a en effet 2 cas :
 - Le thread entrant n'a jamais été élu. Dans ce cas, le `jr $31` va nous faire entrer dans la fonction `thread_bootstrap()` dont le but est de lancer le thread en allant chercher les informations dans la structure `thread_s` du thread nouvellement élu, à savoir
 - la fonction de démarrage `start()` ou `thread_start()`
 - la fonction principale du thread (uniquement pour les threads standards, c'est inutile pour le thread `main()`, on sait que c'est `main()`)
 - l'argument du thread (uniquement pour les threads standards, c'est inutile pour le thread `main()`, ce sont en principe des arguments de la ligne de commande (on ne voit pas ça pour le moment)).
 - Le thread avait déjà été élu et donc il avait perdu le processeur et il avait appelé `thread_save()`.
 - En conséquence, on sortira de `thread_load()` par `thread_save()` et on revient dans `sched_switch()`.
 - Pour qu'on ne rentre pas dans une boucle sans fin, la valeur de retour de `thread_save()` après une restauration de contexte est 0 (alors que c'est 1 après une sauvegarde). On teste donc cette valeur de retour de `thread_save()` pour savoir ce qu'on doit faire ensuite.
 - On sort ensuite de `sched_switch()` et on revient dans `thread_yield()` (actuellement c'est le seul cas, mais nous verrons d'autres fonctions appelant `sched_switch()`).
 - Après, on revient dans un syscall ou dans l'ISR du timer, suivant l'événement qui avait abouti à la perte du processeur par le thread courant.

A.2. Questions sur l'implémentation

1. Quelles sont les fonctions de l'API utilisateur des threads et les états de threads ? Indiquer les changements d'état provoqué par l'appel des fonctions de cette API. Regardez les transparents pour répondre.

- Il y a 3 fonctions dans l'état actuel du code, il y en aura d'autres plus tard dans le module.
 - Création : `int thread_create (thread_t * thread, void *(*fct) (void *), void *arg);`
Le thread créé prend l'état `READY`
 - Cession : `int thread_yield (void);`
Le thread courant passe de l'état `RUNNING` (donné par l'ordonnanceur) à l'état `READY`
Le thread élu passe de l'état `RUNNING` (donné par l'ordonnanceur) à l'état `READY`
 - Terminaison : `int thread_exit (void *retval);`
Le thread courant passe de l'état `RUNNING` (donné par l'ordonnanceur) à l'état `DEAD`
- Nous verrons d'autres fonctions (p. ex. `mutex_lock()`) et d'autres états de threads (p. ex. `WAIT` ou `ZOMBI`)

2. La structure `thread_s` rassemble les propriétés du thread, sa pile et le tableau de sauvegarde de son contexte. Cette structure est, dans l'état actuel du code, entièrement dans le segment des données globales de l'application. Pouvez-vous justifier cette situation et en discuter ?

- Pour chaque thread, on a besoin d'une pile d'exécution, d'un tableau pour stocker le contexte (les valeurs de registres) et des propriétés (état, pointeur sur la fonction principale, etc.).
- La pile est nécessairement dans l'espace utilisateur parce que les fonctions utilisateurs vont y mettre leur contexte d'exécution.
- Le contexte du thread et ses propriétés pourraient être mis dans l'espace noyau, mais il faudrait avoir 2 structures par thread.

- une dans l'espace utilisateur pour la pile et peut-être d'autres informations (que nous verrons plus tard)
 - une dans l'espace noyau, pour le contexte du thread et ses propriétés.
 - Lors de la création d'un thread, il faudrait allouer deux structures dynamiquement, mais c'est impossible dans l'état actuel du code, car nous n'avons pas d'allocateur de mémoire dynamique.
 - Nous avons donc fait un choix simplificateur et une seule structure entièrement dans l'espace utilisateur. C'est un choix temporaire.
3. Le tableau de sauvegarde du contexte d'un thread est initialisé avec des valeurs qui seront chargées dans les registres du processeur au premier chargement du thread. Tous les registres n'ont pas besoin d'être initialisés avec une valeur. Seuls les registres `$c0_sr` (`$12` du coprocesseur système), `$sp` (`$29` des GPR) et `$ra` (`$31` des GPR) ont besoin d'avoir une valeur choisie. Pourquoi ?
- Lorsque l'ordonnanceur élit un thread pour la première fois,
 - il va débiter par `thread_bootstrap()` parce que c'est l'adresse de cette fonction qui a été mise dans `$31`,
 - aucun registre GPR ne contient encore rien de significatif, sauf bien sûr `$sp` le pointeur de pile,
 - `thread_bootstrap()` appelle `thread_launch()` pour sauter dans la fonction `thread_start()` avec l'instruction `eret`, il faut que `$c0_sr` soit tel que l'exécution d'`eret` nous amène en mode user, interruptions autorisées.
4. `$c0_sr` est initialisé avec `0x413`, dite pourquoi.
- `0x413` → `HWI0=1` ; `UM=1` ; `EXL=1` et `IE=1`
 - `HWI0=1` et `IE=1` permettent d'autoriser les interruptions
 - `UM=1` permet de dire que le mode futur sera le mode user
 - `EXL=1` permet d'imposer le mode kernel interruptions masquées, quels que soient les bits `UM` et `IE`.
5. La fonction `sched_switch()` appelle d'abord l'électeur de thread qui choisit le thread entrant (qui gagne le processeur), puis `sched_switch()` sauve le contexte du thread sortant (qui perd le processeur) et charge le contexte du thread entrant, enfin `sched_switch()` change l'état du thread entrant à `RUNNING`. `sched_switch()` est appelée par `thread_yield()`. Pouvez-vous expliquer pourquoi avoir créé `sched_switch()` ? Ce n'est pas évident au premier abord, mais il y a une raison.

```
void sched_switch (void) { //
    int th_curr = thread_current_idx; // n° du thread courant dans thread_tab
    int th_next = sched_elect (); // demande le numéro du prochain thread
    if (th_next != th_curr) { // Si c'est le même thread, ne rien faire
    !
        if (thread_save (thread_tab[th_curr]->context)) { // sauve le ctx du thread sortant et rend
        1
            thread_current_idx = th_next; // mise à jour de thread_current_idx
            thread_load (thread_tab[th_next]->context); // chargement de contexte & sortie par jr
        $31
        } // donc de thread_save(), mais qui rend 0
    }
    thread_tab[thread_current_idx]->state = TH_STATE_RUNNING; // the thread choisi est dans l'état
}
int thread_yield (void) {
    thread_tab[thread_current_idx]->state = TH_STATE_READY; // état futur du thread sortant
    sched_switch (); // changement de threads (ou pas)
    return 0;
}
```

- Si vous avez trouvé, vraiment bravo !
- Un principe de fonctionnement de l'ordonnanceur, c'est que l'ordonnanceur ignore la raison pour laquelle un thread perd le processeur.
- Son travail, c'est d'élire un nouveau thread entrant, sauve le contexte du thread sortant et charge le contexte du thread entrant. Il change également l'état du thread entrant à `RUNNING` parce qu'il sait que c'est le bon état (il vient de charger un contexte et donc le thread entrant est nécessairement le `RUNNING` thread).
- Actuellement, un thread a deux manières de perdre le processeur : `thread_yield()` et `thread_exit()`:

```
int thread_yield (void) {
    thread_tab[thread_current_idx]->state = TH_STATE_READY; // l'état passe de RUNNING à READY
    (le thread cède juste le proc)
    sched_switch (); // demande à l'ordonnanceur de
    trouver un autre thread READY
    return 0;
}
void thread_exit (void *value_ptr) {
    thread_tab[thread_current_idx]->state = TH_STATE_DEAD; // l'état passe à DEAD
    sched_switch (); // demande à l'ordonnanceur de
    trouver un autre thread READY
}
```

- Comme vous pouvez le voir, dans ces fonctions, on commence par changer l'état du thread et après on appelle `sched_switch()`.
 - Pour le moment, c'est un simple changement d'état avant l'appel à `sched_switch()`, mais plus tard, il y aura d'autres opérations, pour la gestion des listes d'attente des ressources partagées ou la gestion des terminaisons de threads dont d'autres attendent la terminaison (avec `thread_join`).
6. Quand un thread est élu pour la première fois, à la sortie de `thread_load()`, on appelle la fonction `thread_bootstrap()`. Retrouvez dans les transparents du cours les étapes qui vont mener à l'exécution de la fonction principale du thread élu, et expliquez-les.
- J'ai déjà décrit les étapes dans d'autres réponses, mais je vais le refaire.
 - `thread_bootstrap()` appelle `thread_launch()` qui appelle `thread_start()` ou `_start` qui appelle la fonction principale de thread.
 - `thread_bootstrap()` se contente de changer l'état du thread élu à `RUNNING` (`sched_switch()` donnera aussi cet état quand le thread sera réélu, mais pour le moment c'est la première fois que le thread est choisi alors on n'est pas revu dans `sched_switch()`), puis `thread_bootstrap()` appelle `thread_launch()` en lui donnant 3 arguments : la fonction principale du

thread, son argument et la fonction de démarrage du thread. Dans le cas du thread `main()`, les deux premiers arguments sont NULL parce la fonction de démarrage du thread `main` `_start()` sait ce qu'il faut faire (lancer `main()`)

- `thread_launch()` c'est juste l'appel à `eret` après avoir initialisé `c0_EPC` avec l'adresse de la fonction de démarrage du thread.
- `thread_start()` ou `_start()` on est dans le code de l'application et on appelle la fonction principale du thread.

7. Un thread peut perdre le processeur pour 3 raisons (dans la version actuelle du code), quelles sont ces raisons ?

- Soit c'est lui qui demande par l'appel explicite à `thread_yield()`, soit c'est une interruption d'horloge, soit c'est `thread_exit()`

8. Quand un thread **TS** perd le processeur pour une raison X à la date **T**, il entre dans le noyau par `kentry`, puis il y a une séquence d'appel de fonction jusqu'à la fonction `thread_load()` du thread entrant **TE**. Lorsqu'on sort de ce `thread_load()`, on est dans le nouveau thread **TE**. Plus tard, le thread **TS** sera élu à son tour et gagnera à nouveau le processeur en sortant lui aussi d'un `thread_load()`. En conséquence, on sortira de la séquence des appels qu'il y avait eu à la date **T**. Expliquez, en vous appuyant sur la description du comportement précédent, pourquoi on ne sauve pas les registres temporaires dans le contexte des threads.

- Quand un thread rend le processeur, il le reprendra plus tard et reviendra précisément dans la fonction où il l'avait perdu (sauf si c'est une sortie définitive avec `thread_exit()` bien sûr).
- Quand on entre dans une fonction C, on sait que l'on peut modifier les registres temporaires, car ils ne contiennent rien pour la fonction appelante. S'ils contiennent quelque chose d'important, la fonction appelante doit sauver leur valeur avant d'entrer dans la fonction appelée.
- Par contre, la fonction appelante suppose que les registres persistants conservent leur valeur, c.-à-d. qu'ils ne sont pas modifiés par la fonction appelée.
- C'est vrai pour toutes les fonctions, c'est donc vrai aussi pour la fonction `thread_save()`. Elle peut modifier les registres temporaires, mais pas les registres persistants.
- C'est donc seul les registres persistants qu'elle sauve et qui seront restaurés par la fonction `thread_load()` qui sortira de `thread_save()` sans modification des registres persistants.

9. Dans le cours, nous suivons l'exécution du code au démarrage (vers le slide 37), nous pouvons voir que la fonction `kinit()` fait 3 choses importantes : (1) initialiser à `0` la section `BSS` (contenant les variables globales non explicitement initialisées dans le programme), (2) demander à l'architecture de s'initialiser et (3) lancer la première (et ici seule) application. Où sont définis les symboles `__bss_origin`, `__bss_end`, `__main_thread`, `_start` et quel est leur type ?

```
void kinit (void)
{
    kprintf (banner);
// 1
    extern int __bss_origin, __bss_end;
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);

// 2
    arch_init(20000); // init architecture ; arg=tick

// 3
    extern thread_t _main_thread; // thread struct pour main()
    extern int _start; // _start() point d'entrée app.
    thread_create_kernel (&_main_thread, 0, 0, (int)&_start);
    thread_load (_main_thread.context);

    kpanic();
}
```

- C'est le fichier `kernel.ld` qui définit la position de `__bss_origin` et `__bss_end` dans la section `.kdata`. Ce sont des adresses qui dépendent des variables globales.
- C'est aussi le fichier `kernel.ld` qui définit les adresses `_main_thread` et `_start`. Par convention, `_main_thread` est au tout début de la section `.data` de l'utilisateur et `_start` est au tout début de la section `.text` de l'utilisateur. Cette convention est nécessaire pour que le kernel sache comment lancer le premier thread de l'application.

10. Dites ce que sont les arguments `2` et `3` de `thread_create_kernel()` dans le code de `kinit()` et pourquoi, ici, on les met à `0` ?

- `thread_create_kernel()` est la fonction qui crée le thread
 - Le premier argument est un pointeur vers la structure `thread_t` à remplir.
 - Le quatrième argument est l'adresse de la fonction `_start` de démarrage du thread `main()`.
 - Le deuxième, c'est normalement l'adresse de la fonction de principale du thread `main()`, ça devrait être l'adresse de la fonction `main()`. Le problème c'est qu'on ne peut pas connaître l'adresse de la fonction `main()`, elle est quelque part dans la section `.text`. Le fait ne pas savoir où est `main()` n'est pas important, car on appelle `_start()` qui appelle `main()`.
 - Le troisième argument, c'est normalement l'argument de la fonction principale du thread, mais par pour le thread `main()` qui doit prendre en principe les arguments de la ligne de commande du shell (ici rien). Là encore, ce n'est pas important, la fonction `_start()` saura trouver les arguments.
 - Comme on n'a pas besoin de arguments 2 et 3, on met 0.

11. Dans la fonction `kinit()`, que se passe-t-il quand on sort de `thread_load()` et pourquoi avoir mis l'appel à `kpanic()` ?

- Quand on sort de `thread_load()`, on entre dans la fonction `_start()` (après un passage par `thread_bootstrap()` et `thread_launch()`). Or on ne sort jamais de `_start()`, on sort de l'application avec `exit()`.
- On n'exécute rien après `thread_load()`, mais si ça devait se produire alors c'est un kernel panic !

12. Dans quelle pile s'exécute la fonction `kinit()` ? Dans quelle section est-elle ? Pourquoi n'est-elle que temporaire ?

- `kinit()` utilise une pile temporaire en haut du segment `.kdata`. Dès qu'on entre dans une application, on utilise la pile de l'application et on ne revient plus jamais sur la pile de `kinit()`.
- Dans le cas général, il y a toujours une application et un thread en cours, et le processeur utilise la pile courante.

- Dans la version actuelle du code, il n'y a qu'une pile par thread qui est utilisée à la fois par les fonctions utilisateur et par le kernel lors des syscall ou des ISR, mais bientôt, nous aurons 2 piles par thread, une pour le code utilisateur et une pour le code noyau.
13. Pour le chargement de thread `main()` avec `thread_load (_main_thread.context)`, on initialise les registres `$16` à `$23`, `$30`, `$c0_EPC`, est-utile ? Si oui pourquoi ? Sinon, pourquoi faire ces initialisations ?
- Non, ça ne sert à rien, le contexte restauré ne contient rien dans ces registres, **mais** quand appelle `thread_load()`, on ne veut pas savoir si c'est pour la première fois pour ce thread ou si c'est une vraie restauration. C'est le `jr $31` qui retournera dans `sched_switch()` ou ira dans `thread_launch()`. Alors, on accepte de restaurer des registres inutilement, c'est juste la première fois.
14. Dans le deuxième TME2, vous avez dû modifier le code `syscall_handler` (gestionnaire de syscalls) pour le rendre interruptible. En effet, lorsque l'application demande un service au noyau, mais que le noyau ne peut pas le rendre immédiatement (comme la lecture d'une touche du clavier), si vous restez bloqué dans l'appel système en attendant la donnée et que les interruptions sont masquées, alors le noyau ne peut pas gérer les IRQ (pour le TME 2, il ne pouvait pas gérer l'IRQ du timer pour gérer le dépassement du temps de jeu). Pour rendre l'appel système interruptible, vous aviez dû mettre `0x401` dans le registre `c0_sr` dans le gestionnaire de syscall, avant d'appeler la fonction de service. Nous allons changer cette politique et considérer que les appels système ne sont plus interruptibles. Quelle(s) conséquence(s) voyez-vous pour les appels système ?
- Si les appels système ne sont plus interruptibles, ils ne doivent plus être bloquants !
15. Que doit-faire le noyau si un thread lui demande une ressource qu'il n'a pas ? Il ne peut pas attendre la ressource, alors il a deux possibilités, les voyez-vous ? Mettez-vous à sa place si vous devez gérer des ressources, par exemple des places dans un restaurant, que vous avez des clients qui se présentent et que toutes les places sont occupées. Que faites-vous ?
- Deux possibilités :
 1. Soit le noyau abandonne le service et rend une erreur au thread pour l'informer que le service ne peut pas être rendu, et donc que le thread doit retenter sa chance plus tard ou faire autre chose. Pour l'analogie, vous dites à votre client de partir et de, s'il veut, revenir plus tard ou pas.
 2. Soit le noyau demande un changement de thread avec `thread_yield()` pour faire quelque chose d'utile pour un autre thread. Pour l'analogie, vous dites à votre client d'attendre et vous allez faire autre chose. Le client attend et tente de rentrer dès qu'un autre client sort du restaurant. Notez que dans cette analogie, il n'y a pas de file d'attente, le dernier client arrivé sera peut-être le premier servi.
 - Dans la version actuelle du code, le thread qui attend la ressource reste dans l'état READY et donc il sera élu par l'ordonnanceur quand son tour viendra pour tester à nouveau la ressource et la prendre si elle est disponible, sinon il subit à nouveau un `thread_yield()`.
 - Ce comportement sera modifié en introduisant un état `WAIT` pour les threads afin de l'ordonnanceur ne donne pas le processeur à un thread dont la ressource attendue n'est pas disponible.

B. Travaux pratiques

Pour la suite de la séance, récupérez l'archive du tp3 et placez là à côté des tp1 et tp2. Le code est fonctionnel, vous pouvez le tester. Je ne vous fais pas modifier, ou pire écrire, la gestion des threads, mais je vous invite à lire le code, c'est très commenté. Les principaux fichiers modifiés sont `kernel/hcpua.S` pour les fonctions `thread_load()`, `thread_save()` et `thread_launch()` (`app_launch()` a disparu, elle n'est plus utile). Des fichiers sont nouveaux : `kernel/kthread.h` qui contient le code de `thread_create_kernel()`, `thread_yield()`, `thread_exit()`, `sched_switch()` et quelques autres. `common/thread.h` qui contient les prototypes de fonctions communes au noyau et à l'utilisateur et `ulib/thread.c` qui contient aussi les fonctions `thread_create()`, `thread_yield()`, `thread_exit()` mais avec des appels système. Je vous invite vraiment à lire le code, c'est un bon exercice de lire le code des autres, croyez-moi. Pour lire le code, vous devez suivre les appels lors de l'entrée dans l'application ou les interruptions d'horloge, ce n'est pas une lecture linéaire du fichier (même si ce n'est pas inutile pour voir une vue d'ensemble).

Vous pouvez voir la différence entre les fichiers du TME B2 et du TME B3

```

01_gameover/                                01_threads
|-- common                                  |-- common
|   |-- syscalls.h                          |   |-- syscalls.h
|-- kernel                                  |-- kernel
|   |-- harch.c                             |   |-- harch.c
|   |-- harch.h                             |   |-- harch.h
|   |-- hcpua.S                             |   |-- hcpua.S
|   |-- hcpuc.c                             |   |-- hcpuc.c
|   |-- hcpu.h                             |   |-- hcpu.h
|   |-- kernel.ld                           |   |-- kernel.ld
|   |-- kinit.c                             |   |-- kinit.c
|   |-- klibc.c                             |   |-- klibc.c
|   |-- klibc.h                             |   |-- klibc.h
|   |-- ksyscalls.c                         |   |-- ksyscalls.c
|   |-- Makefile                           |   |-- kthread.c
|-- Makefile                               |-- Makefile
|-- tags                                   |-- tags
|-- uapp                                  |-- uapp
|   |-- main.c                             |   |-- main.c
|   |-- Makefile                           |-- Makefile
|-- ulib                                  |-- ulib
|   |-- crt0.c                             |-- crt0.c
|   |-- libc.c                             |-- libc.c
|   |-- libc.h                             |-- libc.h
|   |-- Makefile                           |-- Makefile
|   |-- user.ld                             |-- user.ld

```

Questions

1. En utilisant le mode debug et le fichier `label0.s`, donner une estimation de l'overhead de changement de thread
 - Il faut compter le nombre de cycles entre l'entrée dans le noyau (`kentry`) due à une IRQ du Timer et l'appel à `thread_load()` (il manque les cycles utilisés par `thread_load()`, on peut aussi prendre comme borne supérieure, le premier appel de la première fonction appelée dans le nouveau thread).
 - Pour une mesure précise, il faut utiliser le fichier `trace0.s` et compter le temps entre `kentry` et l'instruction `eret` lors du traitement d'une IRQ du Timer.

Etat du code par une lecture directe du registre READ du TTY par `tty_read()`

Pour la partie pratique, vous allez changer la manière de lire les caractères du TTY pour la rendre plus efficace. Tous les changements seront faits dans le fichier `kernel/harch.c`. Commençons par comprendre le code proposé qui est fonctionnel, mais qui a un problème que nous allons résoudre.

uapp/main.c

- Le code ci-dessous contient l'application donnée pour ce TP. Nous avons 3 threads : `main`, `T0` et `T1`.
- `main` et `T1` se contente d'afficher des messages sur le TTY0 et d'attendre (l'attente active (DELAY) est là pour ralentir l'affichage des messages).
- `T0` lit le clavier de TTY1

```

/*-----*/
|_|_/'v'\_|_/\date      2022-02-22
|_|_/'/'\_|_/\copyright  2021-2022 Sorbonne University
|_|_\'x\'x\_|_/\https://opensource.org/licenses/MIT
/*-----*/

#include <libc.h>
#include <thread.h>

#define DELAY(n) for(int i=n;i--;) __asm__("nop");

thread_t t0, t1;

void * t0_fun (void * arg)
{
    char buf[64];
    for (int i = 0;; i++) {
        fprintf (1, "entrez un truc : ");
        fgets (buf, sizeof(buf), 1);
        fprintf (1, "%s\n", buf);
    }
    return NULL;
}

void * t1_fun (void * arg)
{
    for (int i = 0;; i++) {
        fprintf (0, "[%d] t1 is alive (%d) : %s\n", clock(), i, (char *)arg);
        DELAY(1000000);
    }
    return NULL;
}

int main (void)
{
    thread_create (&t1, t1_fun, "bonjour");
    thread_create (&t2, t2_fun, NULL);
    for (int i = 0;; i++) {
        fprintf (0, "[%d] app is alive (%d)\n", clock(), i);
        DELAY(1000000);
    }
    return 0;
}

```

- ulib/libc.c

- La fonction `fgets()` est dans la **libc**, c'est une fonction bloquante du point de vue de l'utilisateur. Il l'appelle pour lire `count` caractères sur le TTY n° `tty` et les enregistre dans `buf`.
- `fgets()` demande 1 caractère à la fois et elle le revoit sur l'écran (c'est un *loopback*) pour que l'utilisateur sache que son caractère a été pris en compte.
- Il y a quelques subtilités dues au fait que lorsque vous taper sur **enter**, le clavier envoie deux caractères '\r' (`13` = *carriage return*) et '\n' (`10` = *line feed*), on jette `\r`. En outre, on gère le `back space` et le `delete` (à qui on donne le même comportement pour simplifier). Je vous laisse essayer de comprendre pour le plaisir.
- Quand `fgets()` appelle `read()`, cela fait l'appel système `SYSCALL_READ`.

```
int read(int fd, void *buf, int count)
{
    return syscall_fct( fd, (int)buf, count, 0, SYSCALL_READ);
}

int write(int fd, void *buf, int count)
{
    return syscall_fct( fd, (int)buf, count, 0, SYSCALL_WRITE);
}
```

```

int fgets (char *buf, int count, int tty)
{
    // to make the backspace, we use ANSI codes : https://www.wikiwand.com/en/ANSI_escape_code
    char *DEL = "\033[D \033[D"; // move left, then write ' ' and move left
    int res = 0;
    count--; // we need to add a NUL (0) char at the end
    char c=0;

    while ((count != 0) && (c != '\n')) { // as long as we can or need to get a char

        read (tty, &c, 1); // read only 1 char
        if (c == '\r') // if c is the carriage return (13)
            read (tty, &c, 1); // get the next that is line feed (10)

        if ((c == 8) || (c == 127)) { // 8 = backspace, 127 = delete
            if (res) { // go back in the buffer if possible
                write (tty, DEL, 7); // erase current char
                count++; // count is the remaining place
                buf--; // but is the next address in buffer
                res--;
            }
            continue; // ask for another key
        } else
            write (tty, &c, 1); // loop back to the tty

        *buf = c; // write the char into the buffer
        buf++; // increments the writing pointer
        count--; // decrements the remaining space
        res++; // increments the read char
    }
    *buf = 0; // add a last 0 to end the string

    return res; // returns the number of char read
}

```

kernel/ksyscall.c

- Je ne met pas toutes les étapes de l'appel du gestionnaire de syscall, vous avez ici le vecteur de syscall qui montre bien que l'on appelle la fonction du noyau `tty_read()`.

```

void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = unknown_syscall, /* default function */
    [SYSCALL_EXIT] = exit,
    [SYSCALL_READ] = tty_read,
    [SYSCALL_WRITE] = tty_write,
    [SYSCALL_CLOCK] = clock,
    [SYSCALL_THREAD_CREATE] = thread_create_kernel,
    [SYSCALL_THREAD_YIELD] = thread_yield,
    [SYSCALL_THREAD_EXIT] = thread_exit,
    [SYSCALL_SCHED_DUMP] = sched_dump,
};

```

kernel/harch.c

- Le thread tente de lire le clavier en lisant `status`, en cas d'échec il cède le processeur avec `thread_yield()`, en sachant qu'on lui rendra plus tard.
- En cas de succès, il enregistre le caractère lu dans le buffer et décrémente le nombre de caractères attendus, si c'est le dernier, il sort.
- Notez qu'il n'y a pas de loopback (c'est-à-dire de renvoi du caractère vers l'écran. C'est une opération complexe, on ne peut pas tout renvoyer (par exemple les flèches), c'est à la fonction système de faire ce travail.

```

int tty_read (int tty, char *buf, unsigned count)
{
    int res = 0; // nb of read char
    tty = tty % NTYS; // to be sure that tty is an existing tty
    int c; // char read

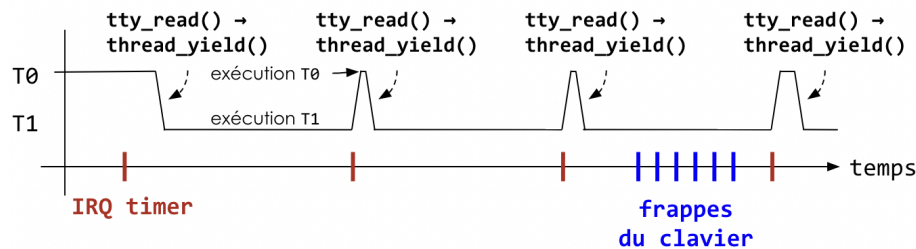
    while (count-- > 0) {
        while (!__tty_regs_map[ tty ].status) { // wait for a char from the keyboard
            thread_yield(); // nothing then we yield the processor
        }
        c = __tty_regs_map[ tty ].read; // read the char
        *buf++ = c;
        res++;
    }
    return res; // return the number of char read
}

```

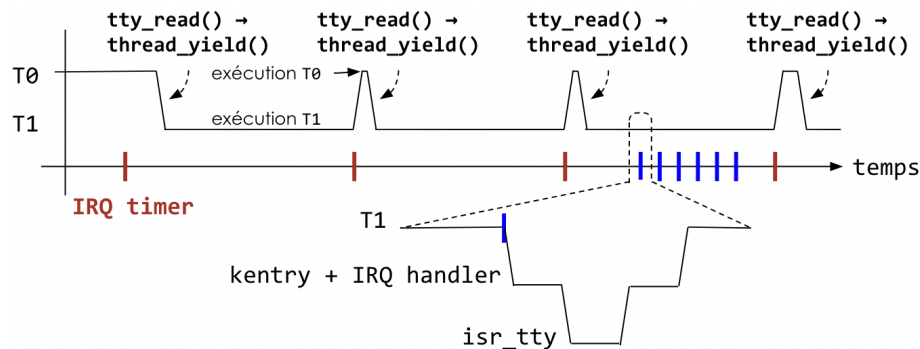
Le problème et une solution possible

Le code proposé à un problème. Pour le comprendre, nous allons partir d'un exemple illustré par le schéma ci-dessous :

- T0** appelle `tty_read()` qui cède le processeur à **T1** en l'absence de frappes.
- Le thread **T0** demande des lectures à chaque fois qu'il a le processeur, **T1** prend le temps qui lui est donné jusqu'à l'IRQ du TIMER.
- Si l'utilisateur frappe beaucoup de touches pendant que **T0** n'a pas le processeur. Les caractères lus doivent être stockés quelque part dans le contrôleur de TTY pour ne pas les perdre. Mais si cette mémoire est trop petite, on risque de perdre des caractères.

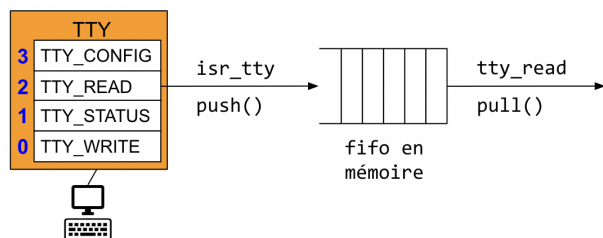


L'idée va être d'utiliser l'IRQ du TTY pour réagir à chaque frappe du clavier pendant l'exécution de **T1** pour lire le clavier et stocker les caractères dans une file d'attente. Sur le schéma ci-dessous est représentée l'exécution de l'isr du TTY qui vole des cycles à **T1** pour lire le caractère reçu par le contrôleur de TTY.



Mise en place d'une FIFO entre l'isr du TTY et la fonction `tty_read()`

Le caractère lu est mis dans une structure FIFO (First In First Out). Le schéma ci-dessous illustre le fonctionnement de la FIFO. Une fifo simple a un écrivain et un lecteur. L'écrivain écrit des données avec une commande `push()` tant que la FIFO n'est pas pleine. Si, il y a deux comportements possibles : l'écrivain attend de la place ou alors l'écrivain jette la donnée, ça dépend de ce qu'on veut. Ici, on jettera, parce qu'on n'a pas le moyen de ralentir le flux de données (les frappes du clavier). Le lecteur lit les données avec `pull()` tant que la FIFO n'est pas vide.



Pour implémenter la FIFO, nous allons utiliser un tableau circulaire et des pointeurs. Il y a une structure et 2 fonctions d'accès.

```

/*
 * Simple fifo (1 writer - 1 reader)
 * - data      buffer of data
 * - pt_write  write pointer for L fifos (0 at the beginning)
 * - pt_read   read pointer for L fifos (0 at the beginning)
 *
 * data[] is used as a circular array. At the beginning (pt_read == pt_write) means an empty fifo
 * then when we push a data, we write it at pt_write, then we increment pt_write % fifo_size.
 * The fifo is full when it remains only one free cell, then when (pt_write + 1)%size == pt_read
 */
struct tty_fifo_s {
    char data [20];
    int  pt_read;    // points to the cell to read
    int  pt_write;   // points to the cell to write
};

/**
 * \brief   read from the FIFO
 * \param  fifo    structure of fifo to store data
 * \param  c        pointer on char to put the read char
 * \return  1 on success, 0 on failure
 */
static int tty_fifo_pull (struct tty_fifo_s *fifo, int *c)
{
    if (fifo->pt_read != fifo->pt_write) {
        *c = fifo->data [fifo->pt_read];
        fifo->pt_read = (fifo->pt_read + 1)% sizeof(fifo->data);
        return 1;
    }
    return 0;
}

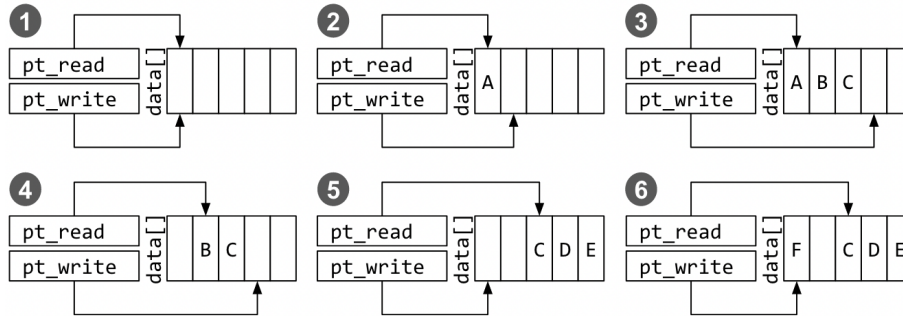
/**
 * \brief   write to the FIFO
 * \param  fifo    structure of fifo to store data
 * \param  c        char to write
 * \return  1 on success, 0 on failure
 */

```

```
static int tty_fifo_push (struct tty_fifo_s *fifo, int c)
{
    // écrire le code de push en vous inspirant de pull
}
```

Les schémas ci-dessous le comportement de la FIFO.

1. A l'initialisation comme la structure est dans les variables globales, les pointeurs `pt_read` et `pt_write` sont à 0. La fifo est vide puisque `pt_read == pt_write`.
2. On écrit `A` et on incrémente `pt_write`, `pt_read` ne bouge pas puisque l'on ne lit pas.
3. On écrit `B` et `C`.
4. On lit `A` et on incrémente `pt_read`, on peut lire parce que `pt_read != pt_write` et donc la FIFO n'est pas vide.
5. On écrit `D` et `E`. Lors de l'incrément de `pt_write` on revient à 0 à cause du modulo `size`.
6. On écrit `F` et ce sera fini parce que la fifo est pleine (`(pt_write + 1)%size == pt_read`), si on veut écrire à nouveau, il faut lire.



Utilisation de la FIFO

Pour utiliser la FIFO, vous allez devoir :

- créer une fifo par TTY, donc un tableau de `struct tty_fifo_s` de taille `NTTYS`.
- Vous allez devoir changer le code de `tty_read()` qui doit désormais lire la fifo.
- Créer une fonction `tty_isr(int tty)` qui lit le registre `READ` du `tty` en argument et écrit le caractère lu dans la FIFO du `tty`.
- Faire le **binding** des lignes d'interruption des TTY. C'est-à-dire modifier `arch_init()` pour
 - démasquer les lignes IRQ 10, 11, 12 et 13 dans le masque de l'ICU, ce sont les entrées de l'ICU utilisées par les TTY0 à TTY3.
 - initialiser les cases 10, 11, 12 et 13 des deux tableaux du vecteur d'interruption : `irq_vector_isr[]` et `irq_vector_dev[]`

Pour la correction, je vous mets le code sans le ranger dans des fonctions, afin de vous aider, mais pas trop...

```
int pt_write_next = (fifo->pt_write + 1) % sizeof(fifo->data);
if (pt_write_next != fifo->pt_read) {
    fifo->data[fifo->pt_write] = c;
    fifo->pt_write = pt_write_next;
    return 1;
}
return 0;
```

```
static struct tty_fifo_s tty_fifo [NTTYS];
```

```
{
    struct tty_fifo_s *fifo = &tty_fifo[ tty%NTTYS ];
    int c = __tty_regs_map[tty].read;
    tty_fifo_push (fifo, c);
}
```

```
{
    int res = 0;
    tty = tty % NTTYS;
    int c;
    struct tty_fifo_s *fifo = &tty_fifo[ tty%NTTYS ];

    while (count-->0) {
        while (tty_fifo_pull (fifo, &c) == 0) {
            thread_yield();
        }
        *buf++ = c;
        res++;
    }
    return res;
}
```

```
for (int tty = 1; tty < NTTYS; tty++) {
    icu_set_mask (0, 10+tty);
    irq_vector_isr [10+tty] = tty_isr;
    irq_vector_dev [10+tty] = tty;
}
```