MI03 - TP2 : Réalisation d'un mini-noyau temps réel ARM

Thomas Legris, Aurélie Suzanne

Printemps 2013

Sommaire

Introduction

Ce TP avait pour objectif la réalisation d'un mini noyau temps réel sur une carte ARM ARMADEUS. Pour cela nous avons, comme au TP précédent, utilisé le langage C. Dans une première partie nous avons créé un ordonnanceur classique de tâches, puis nous avons mis en oeuvre cet ordonnancement pour notre mini noyau temps réel. Pour cela nous avons ajouté à ce noyau un système capable de gérer et commuter les tâches. Enfin nous avons fait quelques essais de notre noyau avec différentes tâches et en y ajoutant la gestion de sémaphores.

1 Ordonnanceur de tâches

Un noyau possède de nombreuses fonction. Il s'occupe notamment de la gestion des périphériques, des exceptions, ainsi que des tâches. Les deux premières fonctions sont très intéressantes, et très complexes. Cependant, nous avons dans ce TP privilégié le côté initialisation et organisation des tâches.

1.1 Qu'est-ce qu'une tâche ?

Une tâche (ou processus) est un programme informatique, découpé en plusieurs sections [1]. Un tâche est principalement composée de données et de code à faire exécuter par le processeur. Ainsi chaque tâche possède un cheminement et remplit donc une fonction induite par le programmeur. On notera que l'on utilisera indépendamment dans ce TP tâche et processus qui sont approximativement similaire, la tâche ayant pour simplifier un processus léger.

Hormis le code et les variables, il y'a deux structures de données très liées à une tâche. Ce sont la pile et le registre d'état processeur. Le registre d'état processeur est une carte de l'état du processeur durant l'exécution d'une tâche. Il est amené à changer régulièrement au cours du temps et est très différent d'une tâche à une autre. Ce registre permet entre autre de gérer des zones d'accès mémoire protégé. Concernant la pile, celle-ci possède toutes les variables non-globale, c'est-à-dire les variables temporaires (comme les variables crée dans une fonction). Cette pile permet donc d'avoir des variables dynamiques et permet également d'être caché et donc non-accessible partout durant l'exécution. La pile sert également à sauvegarder les adresses de retour de fonction, indispensable pour l'imbrication (et donc l'appel) de fonctions.

Pour qu'une tâche fonctionne, il lui faut un environnement propre, complètement dissocié de toutes autres tâches, pouvant s'exécuter tout à tour sur un même processeur (dans le cas des ordinateurs mono-coeur) . C'est là qu'intervient l'ordonnanceur.

1. voir rapport précédent

1.2 Rôle de l'ordonnanceur

L'ordonnanceur doit permettre à plusieurs tâches de s'exécuter sur un processeur unique. Le temps de calcul du processeur doit donc être partagé entre toutes ces tâches. Dans notre cas nous souhaitons que chaque tâche bénéficie d'un temps d'exécution de 10^{-3} secondes. L'ordonnanceur suit l'algorithme de *Round-robin*. C'est à dire qu'il n'y a pas de priorité entre les tâches, et que celles-ci s'exécutent de manière uniforme circulairement (L'ordre de base étant celui d'arrivé des tâches).

En effet, chaque tâche est à son arrivée mise dans une pile de tâches dans laquelle elle va attendre d'être exécutée. A chaque instant l'algorithme *Roun-robin* va déterminer la nouvelle tâche à exécuter par le processeur. Celle-ci sera la suivante à celle qui vient d'être éxécuté dans la pile, et si on atteint la fin de la pile on reviendra au début. On peut voir ce fonctionnement dans la figure suivante :

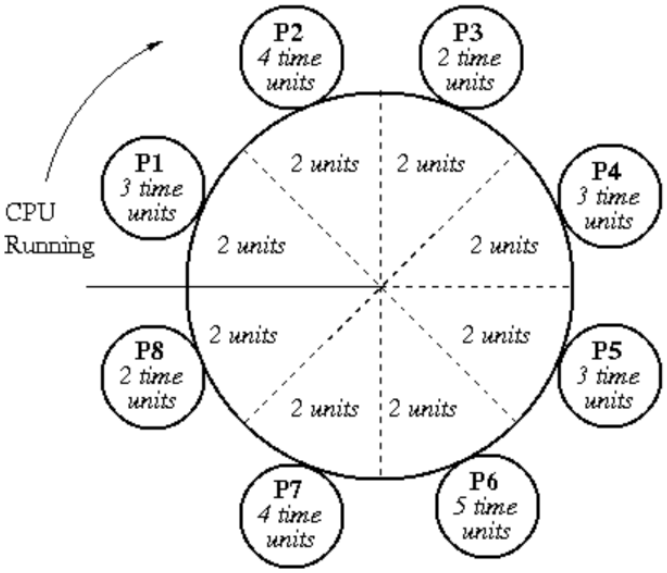

FIGURE 1.1 – Fonctionnement d'un algorithme Round Robin

Dans cette figure, chaque processus possède un temps d'exécution (admettons le temps d'exécution qu'il lui reste). Le CPU va faire jouer tour à tour chacun des processus pour 2 unités de temps, ce qui décrémentera le temps d'exécution des processus. Une fois la tâche éxécutée elle sortira de la file.

Dans la première partie de ce TP nous ne nous sommes intéressés qu'à coder l'ordonnanceur en langage C. C'est à dire que nous n'avons réalisé aucune interaction avec le noyau. Nous avons donc vérifié l'intégrité de notre code avec la console Linux.

1.3 Fonctions d'ordonnancement implémentées

L'algorithme d'ordonnancement nécessite plusieurs fonctions pour pouvoir fonctionner. Tout d'abord il lui faut initialiser la file où seront stockées les tâches. Notre file de tâche

a une taille constante qui est *MAX_ TACHES*, valeur prédéfinie. Elle possède également un pointeur vers la prochaine tâche appelé queue. Nous file est donc initialisée à 0 et la queue vaut *F_ VIDE*, valeur également prédéfinie.

Ensuite il nous faut pouvoir ajouter des tâches à la file. Cela se fait tout simplement en mettant dans la case queue du tableau la valeur de la tâche à ajouter. Puis, dans la case de cette dernière on met l'ancienne valeur de la case queue.

De la même façon, on veut pouvoir retirer une case de la file. Pour cela on fait presque le contraire : on met la valeur de la tâche à supprimer à *F_ VIDE* et on remplace la valeur de la case qui contient notre tâche (c'est à dire celle qui la précède) par la tâche qui suivait celle-ci.

Enfin, le coeur de l'algorithme réside dans le choix de la tâche suivante à exécuter. Ceci se fait en choisissant la prochaine tâche de la file, grâce à la queue. Tout ceci est grandement simplifié par notre structure, puisque queue pointe toujours sur la tache à exécuter. Il nous suffit donc d'exécuter la tâche queue et d'actualiser queue avec la valeur contenue dans le tableau à l'emplacement queue.

Nous possédons désormais un code fonctionnel pour ordonnancer correctement une série de tâche en *Round-robin*.

2 Gestion et commutation de tâches

2.1 Principe de l'appel à l'ordonnanceur

L'appel à l'ordonnanceur se fait de deux manières :

- ➞ Appel par interruption : Cette appel est effectué lors d'une interruption IRQ provoqué par notre Timer [1].
- ➞ Appel direct : Cette appel simule une interruption IRQ, elle permet de forcer la commutation de tâches. C'est pratique pour initialiser et pour effectuer des actions telles que l'endormissement ou le réveil d'un tâche. (Typiquement ce rôle est supporté par la fonction *schedule*)

2.2 Commutation de tâches

Comme nous l'avons vu précédemment, une tâche possède : des variables globales, une pile, un registre d'état et du code. Lorsque nous voulons changer de tâche, il faut absolument sauvegarder le registre d'état du processeur (*cpsr*) ainsi que les registres du processeur, car ces données sont amenées à changer. Pour cela, rien de plus simple, il suffit de stocker ces informations sur la pile de la tâche. En effet, la pile a l'avantage d'être dynamique, une fois notre programme en RAM, seul la connaissance des positions de piles est importante. Ainsi il faudra préalablement connaître les positions de chaque pile de chaque tâche (contenu dans

1. voir rapport TP2

notre cas dans la variable *context*).

Ici nous avons aborder un sujet intéressant : le fait qu'une tâche possède plusieurs piles. En effet pour chaque mode processeur, il lui faut une pile attribuée. Pourquoi ? Parce que chaque mode possède ses propres registres et son propre registre d'état processeur. Néanmoins il est possible de communiquer entre certains registres de certains modes. Ici nous avons deux mode, System et IRQ. Le mode IRQ correspond au déclenchement d'une interruption ici le *timer*. Tandis que le mode System est le mode normal d'exécution des tâches pour nous.

Le fait qu'une tâche puisse "switcher" entre différents modes permet :

- ➞ De sauvegarder l'état de la tâche avant interruption, et donc de pouvoir y revenir par la suite.
- ➞ Permet d'effectuer un traitement sans rapport avec la tâche (typiquement dans notre cas : le changement de tâche).

2.3 Soucis rencontrés pour le noyautage

Ce TP initialement présenté sous la forme de texte à trous, nous a pris beaucoup plus de temps que nous le pensions. En effet nous avons mis beaucoup de temps à comprendre exactement ce qu'il fallait faire et à s'adapter au fait de jouer avec des entités si bas-niveau en C.

Parmi les principaux oublis et erreurs nous pouvons citer :

- ➞ ne pas oublier d'effectuer un appel à *schedule* dès que l'on modifie l'exécution d'une tâche
- ➞ réaliser une fonction de sortie qui boucle à l'infini et ne sorte pas
- ➞ ne modifier le code initial sous aucun prétexte
- ➞ ne pas rajouter de code en dehors des commentaires (en particulier sur l'appel du mode *irq*)
- ➞ être extrêmement précautionneux dans le code arm
- ➞ faire très attention aux gestions de pile
- ➞ revérifier le code des TP précédent sur lequel on s'appuie et sur lequel le moindre bug n'est plus toléré

3 Partage de ressources

Lors de l'accès à une ressource, il se peut qu'une tâche soit déjà en train de l'utiliser. Résultat, il peut y avoir un conflit et il faut absolument bloquer l'utilisation simultané d'une ressource. C'est pour ça que nous nous intéressons au partage de ressources.

3.1 Action sur les tâches

On doit pouvoir avoir accès à deux actions sur une tâche pour faire du partage de ressources :

- ➞ Faire dormir une tâche : Pour cela on passe simplement la tâche dans l'état suspendu, avant de faire appel à l'ordonnanceur. Lorsque l'on effectue ces opérations, on les protège par un *lock* qui nous assure qu'aucune autre fonction ne pourra modifier en même temps que nous les valeurs sur lesquelles on travaille.
- ➞ Réveiller une tâche : Ce processus se fait de la même façon que pour endormir une tâche, sauf qu'au lieu de la passer à l'état suspendu, on la passe à l'état exécutée.

Ici, le mécanisme est donc simple, chaque tâche peut être suspendue ou en exécution. Si le statut est *exécution*, l'ordonnanceur, lors du tour de la tâche, va exécuter la tâche. C'est le comportement normal. Cependant, si la tâche a un statut *suspendu*, l'ordonnanceur va l'ignorer et passer à la suivante. Pour cela on a ajouté dans la fonction *scheduler* une boucle qui continu tant qu'elle n'a pas atteint une tâche dont le statut n'est pas suspendu.

Il y'a néanmoins un risque ici, c'est que si toutes les tâches sont suspendues, l'ordonnanceur peut se retrouver bloqué. Il faudrait prévoir une tâche de fond non suspendable, ce qui permettrait de relancer les interruptions (qui pourrait libérer une ressource et donc une tâche).

3.2 Exclusion mutuelle

Nous cherchons donc désormais un moyen de permettre à deux tâches ayant besoin des mêmes ressources (ou travaillant sur les mêmes ressources) de pouvoir être exécutées, ceci à des intervalles de temps différents, même si on les lance simultanément. Nous allons tout d'abord travailler sur le principe de l'exclusion mutuelle dans lequel on interdit à une tâche l'accès à une zone.

Le problème d'accès à une ressource suit donc toujours le même schéma : Une tâche A entre dans une zone, une tâche B tente alors de rentrer dans la même zone mais se voit refuser l'accès et donc s'endort. Une fois que A a fini, on doit réveiller B qui peut désormais s'exécuter.

Il est possible de faire de l'exclusion mutuelle avec les deux fonctions précédentes. Nous cherchons à implémenter un système où un producteur produit des nombres, pendant que le consommateur cherche à les manger. Pour cela, lorsque notre producteur produit un nombre on lui demande de réveiller le consommateur, puis de s'endormir jusqu'à ce que celui-ci le réveille. Lorsque le consommateur est réveillé, on lui demande de lire le chiffre, puis de réveiller le producteur et de s'endormir jusqu'à ce que celui-ci le réveille. L'un et l'autre s'appelent donc mutuellement, et dans un monde merveilleux ils continueraient ainsi à l'infini.

Néanmoins notre monde n'est pas parfait, ce qui rend cette façon de procéder est très dangereuse. En effet, on peut arriver à un *deadlock* dans lequel chaque tâches attend le réveille de l'autre. C'est à dire que l'on aurait les deux tâches qui se seraient mise en sommeil et attendraient de l'autre qu'elle la réveille. La preuve que même un ordinateur ne vit pas dans le meilleur des mondes se trouve figure 3.1.

Cette situation *deadlock* intervient parce que rien ne garantit que l'envoi d'un signal de réveil ne soit pas interrompue. En effet pour que cet algorithme marche, chaque tâche doit être prête à un moment précis, si la tâche B qui devait envoyer le réveille, effectue son envoie avant que la tâche A s'endorme, ils s'attendrons indéfiniment.

3.3 Sémaphores

3.3.1 Description des sémaphores

Puisque le monde parfait n'existe pas, les hommes ont trouvé une réponse au problème précédent : les sémaphores. Ce mécanisme permet de gérer les signaux indépendamment de l'exécution d'une tâche. Pour ce faire, une sémaphore garde en mémoire toutes les tâches qui attendent un signal. Globalement une sémaphore possède trois caractéristiques : *Init*, P et V.

Init est la valeur à laquelle on initialise le sémaphore, c'est à dire le nombre de tâches qui pourrons accéder en même temps à la ressource. P est une opération (ou fonction) qui reste en attente tant que la ressource n'est pas disponible et va prendre une ressource (décrémentation du nombre de ressources de 1) lorsqu'elle sera exécutée. Elle va ainsi permettre à notre tâche de s'exécuter. Enfin, V correspond au contraire, elle va libérer une ressource, on l'utilisera lorsque notre fonction aura fini de s'exécuter.

Il existe deux types principaux de sémaphores :
- ➞ Les sémaphores privés : ils permettent de synchroniser les tâches.
- ➞ Les sémaphore binaire (Mutex) : ils permettent d'attribuer l'accès exclusif à une ressource.

Le fonctionnement de ces deux sémaphores est le même, la seule distinction tient dans le fait que le mutex est toujours initialisé à un, tandis que les sémaphores privés peuvent être initialisés à de multiples valeurs. En effet, il peut y avoir plusieurs tâches qui peuvent accéder en même temps à la même ressource, mais ce nombre doit pouvoir être limité, et c'est à cela que servent les sémaphores privés.

3.3.2 Implémentation des sémaphores

Nous avons donc créer un nouveau fichier (sem.c) qui peut traiter ces sémaphores. Celles-ci sont gérées à l'aide d'une file FIFO. En effet, les premières tâches à être mises en attente sur une ressource seront aussi les premières à être exécutée. Nous possédons ainsi un tableau de sémaphores, chaque sémaphores étant elle-même déterminée par une valeur (le nombre de ressources) et une file des tâches en attente.

Nous avons cinq fonctions principales : *s_init, s_cree, s_close, s_wait et s_signal.* Les trois premières très simples servent jste à initialiser le tout, ou bien à ajouter ou supprimer des tâche des différentes files. *s_wait* correspond à ce que l'on avait appelé P, et qui va donc vérifier si la valeur de la sémaphore est supérieure à 0 ou non. Si c'est le cas, pas de problème on décrémente valeur et laisse la tâche s'exécuter à la sortie. Sinon on tombe en sommeil profond dans *s_wait* juqu'à ce que la valeur soit supérieure à 0 et que l'on puisse en sortir. *s_signal* correspond quand à lui à S et va incrémenter valeur, et actualiser la pile des tâches

qui attendent en enlevant celle qui vient d'être jouée et donnant la main à la prochaine.

Au cours de ce TP nous avons pu traiter deux problèmes avec les sémaphores : celui de notre exclusion mutuelle avec les producteurs/consommateurs, ainsi que celui très connu des philosophes qui veulent manger.

3.3.3 Le producteur/consommateur

Ce problème fonctionne désormais grâce à deux sémaphores. L'une est initialisée à un, et la seconde à 0 (*mySem* et *mySem2* respectivement). En effet le producteur fera le P de *mySem* au début et le S de *mySem2* une fois un nombre crée. Le consommateur lui fera l'inverse. Nous pouvons ainsi voir la bon fonctionnement de tout ceci sur la figure 3.2.

Afin de vérifier que tout cela fonctionnait bien, nous avons même crée deux consommateurs, voir figure 3.3.

3.3.4 Le problème des philosophes

Enfin le dernier problème que nous avons traité au cours de ce TP est celui des Philosophes qui à table oscillent entre manger et penser puisque le nombre de fourchettes est limité. Nous avons commencé à traiter ce problème qui est tout d'abord un problème d'algorithmique, mais nous n'avons pas eu le temps de le mener à terme.

Conclusion

Ce TP a donc été un TP très chronophage, surtout la partie sur la gestion et commutation de tâches que l'on a eu du mal à prendre en main. Ce TP est pourtant une version très simplifiée de ce que pourrait être un noyau temps réel, effectivement nous n'avons pas eu à gérer des algorithme d'ordonnancement à priorité, ni de tâches qui dépendent d'autres tâches ce qui aurait très rapidement complexifié notre problème. On a ainsi pu percevoir la difficulté que représente l'environnement temps réel.

Mais quel bonheur de voir *noyauter* et *sémaphorer* tout ce système !

Figures

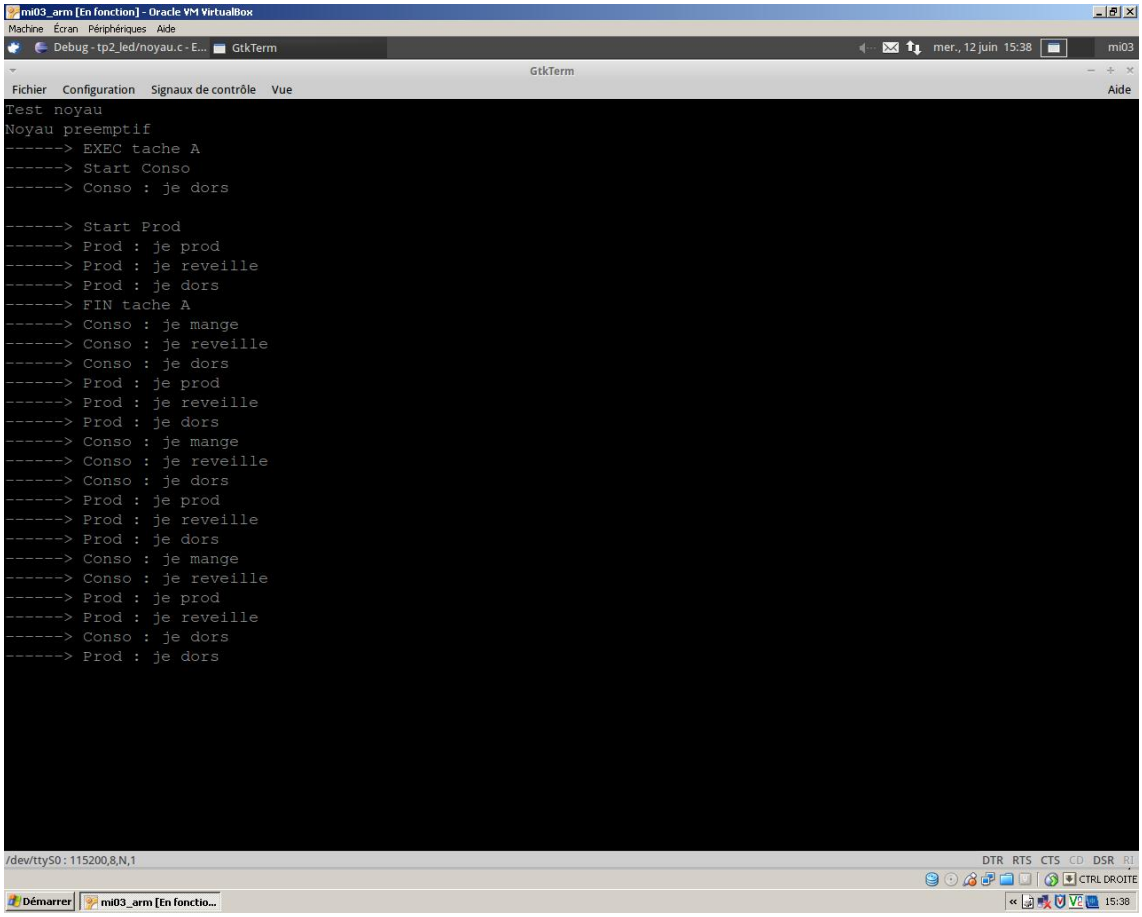

FIGURE 3.1 – Deadlock sur de l'exclusion mutuelle avec producteur consommateur

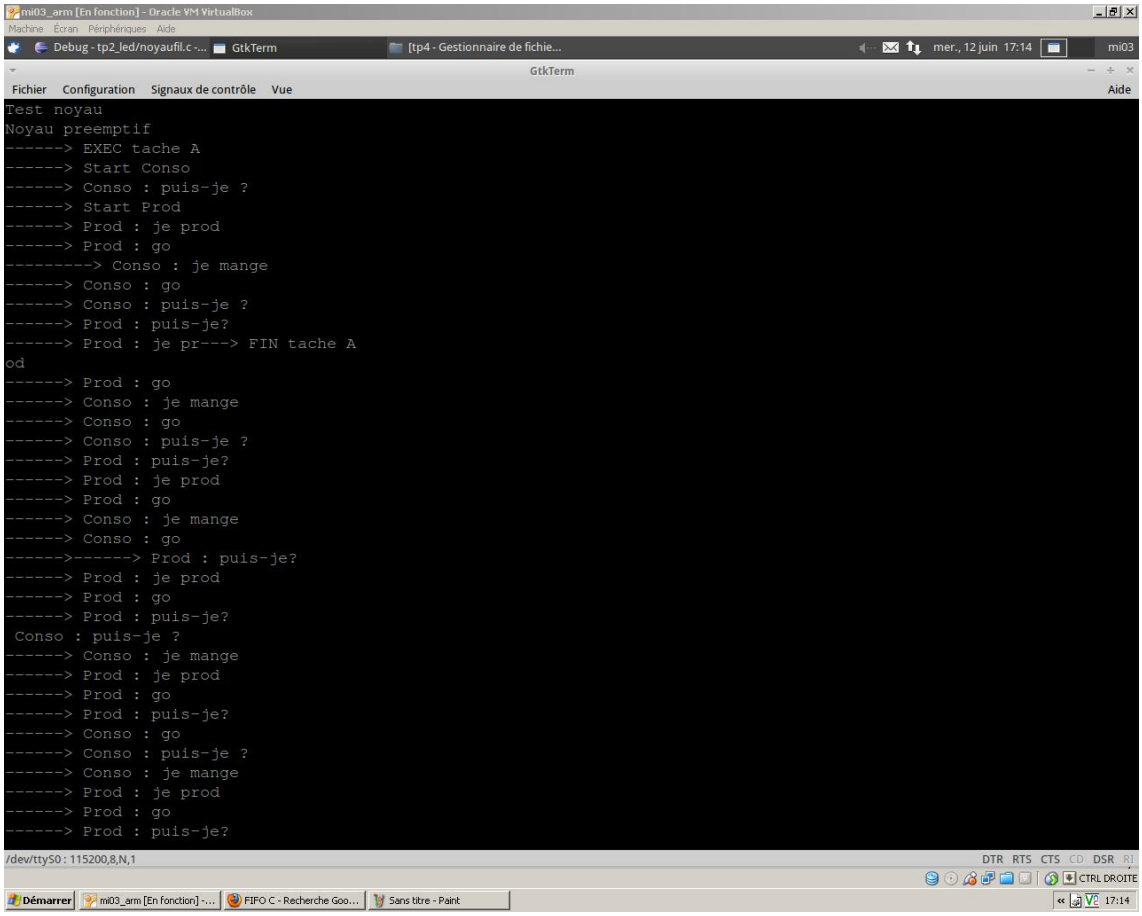

FIGURE 3.2 – Sémaphores des producteurs consommateurs

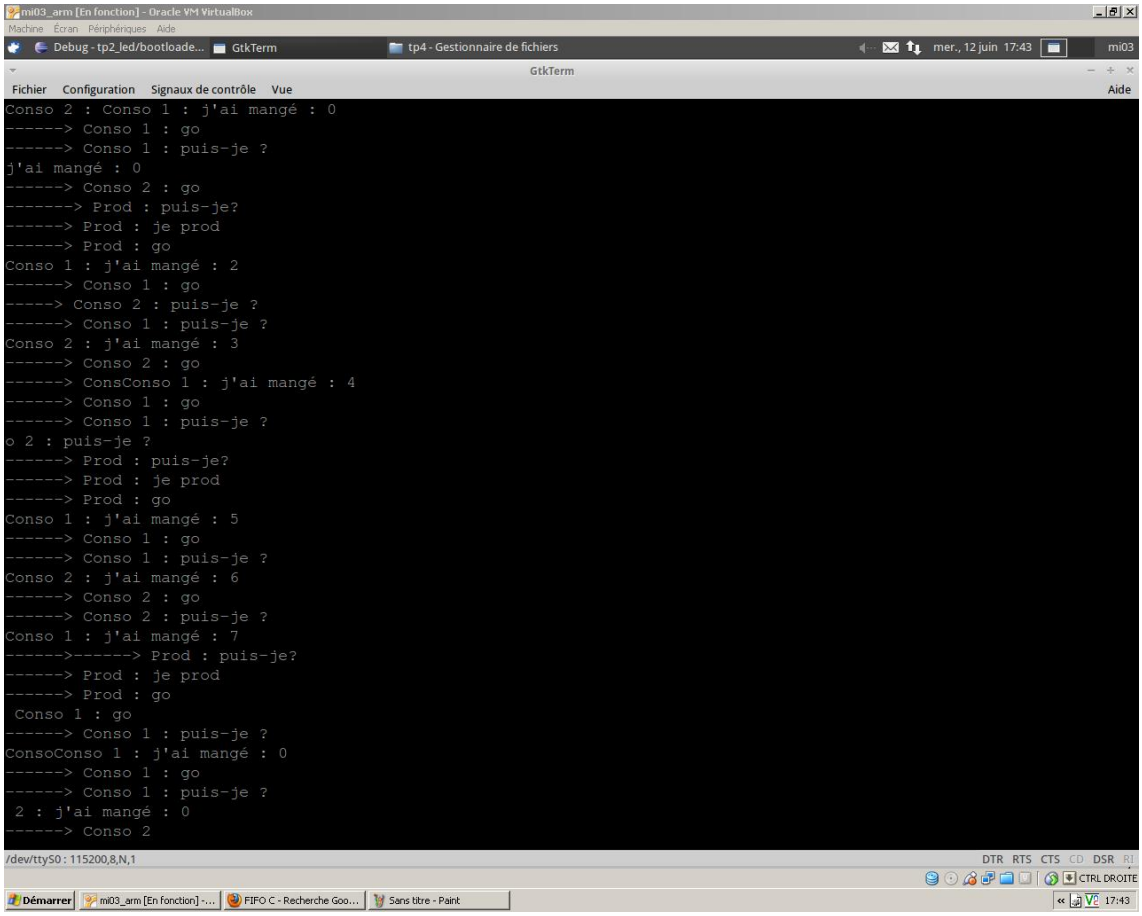

FIGURE 3.3 – Sémaphores avec deux consommateurs