Documentation : Compatibilité POSIX de Marcel

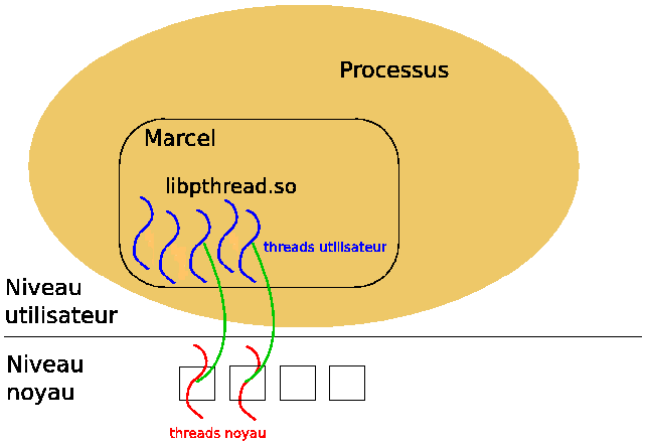
S.Jeuland

1 Contexte

Marcel est une bibliothèque de threads utilisateur qui se décline en plusieurs « flavors » suivant la plate-forme et les besoins visés. Afin de pouvoir exploiter les machines SMP, Marcel est capable d'utiliser un ordonnancement à deux niveaux basé sur des threads noyaux fournis par le système. Avec le système Linux, Marcel est également capable d'utiliser les activations permettant de remédier à la limitation classique des bibliothèques de threads utilisateurs, à savoir les appels systèmes bloquants. Marcel a également un bon ordonnancement sur machines NUMA à l'aide d'un mécanisme puissant permettant au programmeur d'exprimer la structure de son application. Toutes ces « flavors » se basent sur un même noyau solide de gestion de threads et sont spécialisées lors de la compilation.

Marcel a pour avantage de ne pas passer par le noyau lors de l'appel d'un certains nombre de fonctions. Ceci permet de gagner beaucoup de temps et de souplesse dans l'exécution des programmes. L'équipe Runtime aimerait faire fonctionner plusieurs applications existantes sur Marcel. Ces applications, qui tournent sur système d'exploitation Linux/UNIX, utilisent une bibliothèque appelée libpthread qui est la librairie de threads normalisée POSIX et sur laquelle s'appuie la très grande part des applications qui utilisent des threads. C'est pour cela que l'on veut avoir une couche de compatibilité POSIX pour Marcel. C'est ici qu'intervient le stage Marcel-POSIX.

FIG. 1 – Gestion des threads par Marcel



1.1 Objectif

L'objectif du stage est de fournir une nouvelle bibliothèque qui comportera les mêmes fonctions que la bibliothèque libpthread du système. Ses fonctions feront appel à des fonctions de Marcel. Lors de l'exécution des applications, la bibliothèque libpthread du système est remplacée par celle de marcel. Cette nouvelle bibliothèque suit la norme POSIX, qui est une norme imposant des règles précises portant sur l'implémentation des fonctions UNIX.

1.2 Travail demandé

Mon travail lors de ce stage a été de donner à l'ordonnanceur Marcel d'une part une sémantique POSIX, et d'autre part une gestion de signaux. J'ai donc dû écrire une partie des fonctions de la bibliothèque libpthread, tout en suivant la norme POSIX. Ces fonctions de gestion de threads utilisateurs (qui sont des threads créés par l'application elle-même et non par le noyau) ont pour la plupart la particularité de commencer par la chaîne de caractère «*pthread_*». Certaines fonctions font appel à des fonctions de l'ordonnanceur Marcel et d'autres non. On peut par exemple y retrouver des fonctions portant sur des attributs de

threads, des mutexes, des barrières, etc. De plus, j'ai implémenté la plupart des fonctions qui gèrent les signaux Linux, ainsi que leur environnement.

1.3 Finalité du projet

L'équipe Runtime souhaitait faire tourner l'ordonnanceur Marcel sur des applications courantes qui utilisent des threads et des signaux dans un environnement Unix/Linux. En effet, cela valoriserait le travail de l'équipe de recherche si les utilisateurs pouvaient lancer diverses applications avec l'ordonnanceur Marcel. De plus, la compatibilité POSIX pour Marcel permettrait d'étendre le spectre d'application des travaux de recherche sur des logiciels utilisateur précompilés selon la norme POSIX. La compatibilité POSIX apporterait aussi l'avantage de faciliter l'utilisation de l'ordonnanceur Marcel puisque l'on aurait plus besoin de changer le code source des applications existantes afin qu'elles puissent tourner dessus. Il suffirait donc de lancer l'exécutable dans un environnement Marcel.

Il faut noter que la compatibilité POSIX pour Marcel a été demandée par l'entreprise Bull et le CEA. On retrouve la notion de transfert industriel qui existe à l'INRIA car le travail réalisé sur la compatibilité POSIX de Marcel trouvera tout de suite son application dans le milieu industriel.

2 Cahier des charges fonctionnel

2.1 Compatibilité POSIX

Il existe deux types de compatibilités POSIX, la compatibilité API et la compatibilité ABI (Application Binary Interface). La compatibilité API (Application Programming Interface) porte sur tout ce qui est nom de fonctions, de variables, de structures utilisés dans les programmes et fonctions. La compatibilité ABI porte sur les valeurs des variables, de retours d'erreur, sur la taille des structures, l'ordre des champs de structures. Les compatibilités API et ABI se différencient aussi avec la notion de compilation. La compatibilité API d'un programme intervient avant la compilation alors que la compatibilité ABI intervient après, une fois qu'un exécutable est généré.

Dans le cadre du stage Marcel-POSIX, nous pouvons voir que les fonctions écrites ont trois niveaux de compatibilité POSIX. Les fonctions dont le prototype commence par « *marcel_* » n'ont *a priori* aucune compatibilité avec la norme POSIX, tant au point de vue de l'API que de l'ABI. C'est l'interface originelle de Marcel que l'on ne doit pas changer. Les fonctions « *pmarcel_* » quant à elles, doivent être compatibles API et les fonctions dont le prototype commencera par « *lpt_* » devront être compatibles API et compatibles ABI. Elles seront équivalentes avec les fonctions « *pthread_* » excepté le nom.

2.2 Utilisation des trois niveaux de compatibilité

- les fonctions marcel sont utilisées par les gens qui programment directement avec Marcel et qui n'ont pas besoin de la sémantique POSIX.
- les fonctions *pmarcel peuvent* être utilisées par les gens qui veulent utiliser Marcel avec une sémantique POSIX mais qui peuvent recompiler leur application. En effet, la compatibilité au niveau API (avant compilation) leur suffit.
- les fonctions de type *lpt* (et donc *pthread*) sont implémentées pour les gens qui veulent utiliser Marcel afin de faire tourner des programmes déjà compilés pour utiliser libpthread (la recompilation étant impossible pour des raisons légales ou pratiques).

2.3 Cahier des charges technique

- ne pas modifier le comportement des fonctions Marcel préexistantes car celles-ci sont utilisées par toute l'équipe Runtime.
- toutes les fonctions implémentées doivent avoir une sémantique SUSV3. La norme SUSV3 est une norme qui reprend le langage C et POSIX, et y ajoute des extensions bien connues. Les fonctions de type lpt (et pthread) devront être compatible API avec la norme SUSV3 et compatible ABI avec la bibliothèque libpthread de Linux, alors que les fonctions de type pmarcel pourront se contenter d'être compatible API avec SUSV3.
- la gestion de signaux Unix doit être fonctionnelle dans Marcel.
- l'indentation du code est de huit espaces par profondeur de bloc.
- faire tourner le maximum de tests de posixtests

2.4 Critères de validation du projet

L'équipe Runtime a fixé un seul critère de validation. Il est tel que la JVM (bibliothèque JAVA de Sun Microsystems) tourne sur Marcel. En effet, si la JVM tourne sur Marcel, il est probable que beaucoup d'autres applications utilisateur puissent tourner sur Marcel.

2.5 Plannification du projet

Au début du stage, j'avais prévu de passer deux semaines sur la compréhension de l'ensemble des modules et des fonctions de l'ordonnanceur Marcel, ainsi que sur l'apprentissage de toutes les fonctions de la libpthread et de la gestion de signaux. Ensuite, je m'étais fixé environ deux mois de travail sur l'implémentation de signaux, puis un mois sur les autres fonctions de la libpthread et deux semaines sur les tests. En réalité, contrairement à ce que j'avais imaginé, l'implémentation des signaux et du reste de la libpthread s'est faite plutôt en paralléle et s'est précédée d'ajouts de contrôle d'erreur aux fonctions préexistantes. L'implémentation des fonctions s'est en général terminée plus vite que prévu et les tests ont donc commencé plus tôt et se sont déroulés plus longtemps.

Le stage s'est déroulé comme suit :

- compréhension de l'ordonnanceur et de la norme SUSV3 : deux semaines

- ajout des contrôles d'erreurs des fonctions préexistantes : deux semaines
- implémentation d'une gestion de signaux et des fonctions de la libpthread : dix semaines
- programmes de tests et débuggage des fonctions implémentées : trois semaines
- validation et nettoyage de code (indentation, etc) : une semaine

3 Réalisation du projet

3.1 Analyse du cahier des charges et de ses contraintes

- ne pas modifier le comportement des fonctions Marcel préexistantes car celles-ci sont utilisées par toute l'équipe Runtime. Afin de ne pas modifier les fonctions Marcel, il suffit d'écrire des surcouches de ces fonctions avec la sémantique POSIX demandée. Dans la plupart des cas, ces surcouches feront appel aux fonctions Marcel préexistantes. Des macros permettent d'écrire des fonctions de sémantique Marcel, pmarcel et lpt de manière indépendante ou liée, si la sémantique est la même.
- toutes les fonctions implémentées doivent avoir une sémantique POSIX. Le comportement POSIX est défini par la norme SUSV3. Les fonctions de type lpt (pthread) devront être compatible ABI et API avec la norme SUSV3, alors que les fonctions de type pmarcel pourront se contenter d'être compatible API. Je devrai donc lire la documentation SUSV3 et donner sa sémantique aux fonctions que j'implémenterai.
- la gestion de signaux Unix doit être fonctionnelle dans Marcel. Je devrai implémenter une gestion de signaux avec les fonctions que je trouverai dans la documentation SUSV3. Leur sémantique devra être identique.
- l'indentation du code est de huit espaces par profondeur de bloc.

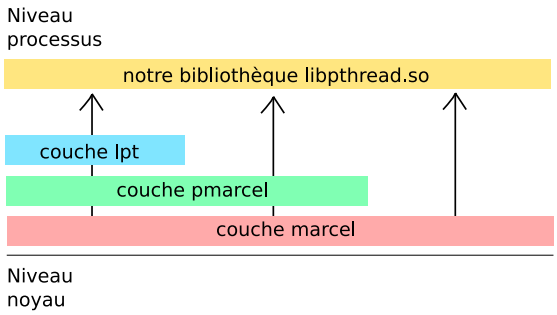
3.2 Conception générale

Tout d'abord, le langage utilisé est le langage C car nous travaillons directement sur le système d'exploitation Linux.

Les fonctions de la libpthread que nous devons implémenter sont des surcouches POSIX des fonctions de la bibliothèque Marcel. Elle sont compatible

ABI avec la norme POSIX. Comme je l'ai écrit précédemment, il existe un intermédiaire entre les fonctions Marcel et celle de la libpthread : il s'agit des fonctions pmarcel qui elles ont la possibilité de disposer que de l'API POSIX. En résumé, les fonctions de la libpthread (que nous appelons avec le suffixe lpt), appellent des fonctions de type pmarcel qui appellent celles de Marcel. Il existe des cas où la sémantique pmarcel est identique à celle de la sémantique de pthread. Dans ce cas on ne passe pas par les fonctions de type lpt.

FIG. 2 – Schéma des couches de compatibilité POSIX



Les différentes fonctions se regroupent en famille de plusieurs modules. Par exemple, les fonctions liées aux signaux ont leur propre module (*marcel_signal.c*) alors que les fonctions de gestion de sémaphores sont ailleurs (*marcel_sem.c*). Tous ces modules sont regroupés dans un dossier « source » alors que les headers, quant à eux, sont dans le dossier « include ». Une fois les fonctions de type lpt implémentées, nous leur donnons un nouvel alias qui portera le nom réel de la fonction utilisée dans la libpthread. Lors de la compilation de Marcel, un nouveau répertoire nommé « build » est crée. Il contient notre bibliothèque « libpthread.so ». C'est celle-ci qui sera utilisée au lancement d'un programme. Afin d'utiliser Marcel au lieu d'utiliser l'ordonnanceur du système, nous avons écrit un script de démarrage appelé « pm2-libpthread » qui charge l'environnement Marcel (dont notre libpthread) avant de lancer le programme.

Un environnement de gestion de signaux a été complètement implémenté. Il a fallu stocker de manière globale les différentes informations comme les signaux en suspens, les masques de signaux, etc.

J'ai pu rencontrer quelques difficultés dans ce projet. Au début, il fut difficile

de se plonger dans un environnement inconnu. Il a fallu se plonger dans le code afin de le comprendre. Ensuite, j'ai rencontré quelques difficultés pour implémenter les fonctions de gestion de signaux à partir de zéro. Enfin, il m'a pris un peu de temps pour apprendre les détails de la norme POSIX et l'appliquer méticuleusement au code.

3.3 Conception détaillée

3.3.1 Le thread

L'ordonnanceur Marcel gère la mise en place des threads (fils d'exécutions) sur les processeurs. Le thread est l'élément central du projet. C'est autour de lui que s'articulent la majorité des fonctions implémentées. Il est nécessaire de stocker diverses informations relatives à chaque thread. Dans *marcel_descr.h* est implémentée la structure de threads *marcel_task_t* qui contient toutes les informations. Les champs que j'ai ajoutés à la structure préexistante concernent majoritairement les signaux (masque courant, signaux en suspens) et les états d'annulation du thread (annulation activée/désactivée, annulation immédiate/déferrée). La structure *marcel_t* est très utilisée, c'est l'identité du thread et un pointeur sur la structure *marcel_task_t* qui contient les informations de ce même thread.

3.3.2 Factorisation du code

Dans ce projet, j'ai beaucoup utilisé des macros qui servent à définir soit : des fonctions marcel, des fonctions pmarcel, des fonctions à la fois marcel et pmarcel. La macro *DEF_MARCEL* (respectivement *DEF_POSIX*) prend en paramètre cinq champs :

- le type de retour
- le nom de la fonction sans le suffixe *marcel_* (respectivement *pmarcel_*)
- les paramètres de la fonction
- les même paramètres sans leur type
- le code complet de la fonction

Exemple pour l'interface Marcel (les deux codes sont équivalents) :

- *DEF_MARCEL(int,exit,(marcel_t thread),(thread), { ... code ... })*

– *int marcel_exit(marcel_t thread) { ... code ... };*

La macro *DEF_MARCEL_POSIX* crée à la fois les fonctions des interfaces marcel et pmarcel. De plus, afin de renommer nos fonctions de type pmarcel et lpt en fonctions commençant par le suffixe « pthread » (comme dans la libpthread), nous utilisons aussi des macros

- *DEF_PTHREAD* renomme les fonctions « pmarcel_nomfonction » en « pthread_nomfonction » (ex : *pmarcel_getschedparam* en *pthread_getschedparam*).
- *DEF_C* renomme les foncions « pmarcel_nomfonction » en « nomfonction » (ex : *pmarcel_sleep* en *sleep*).
- *DEF_LIBPTHREAD* renomme les fonctions « lpt_nomfonction » en « pthread_nomfonction » (ex : *lpt_sigmask* en *pthread_sigmask*).
- *DEF_LIBC* renomme les foncions « lpt_nomfonction » en « nomfonction » (ex : *lpt_sigaction* en *sigaction*). On utilise les fonctions de type lpt quand la sémantique de la fonction de la libpthread est différente de celle que nous avons implémenté en pmarcel. Par exemple, les structures lpt peuvent être différentes des structures pmarcel.

3.3.3 Accès concurrents

Nos threads peuvent partager le même espace de mémoire (variables globales, modifications de la structure d'un autre thread dans le cas d'un envoi de signal par exemple). Afin d'assurer la fiabilité des données lors des accès concurrents, nous devons verrouiller les données sensibles à l'aide de mutex. Nous effectuons donc un *mutex_lock* avant l'écriture ou la lecture d'une variable partagée, puis un *mutex_unlock* après. Dans l'ordonnanceur Marcel, il existe de petits verrous très légers (les spinlocks) qui tournent en boucle tant qu'il n'ont pas l'accès à la variable. Ils sont beaucoup utilisés et il faut faire attention qu'ils puissent prendre le verrou assez vite car ils consomment de la ressource processeur. Cela veut dire que lorsque l'on possède le verrou, il faut le relâcher vite

3.3.4 La gestion de signaux

Toute la gestion de signaux a été implémentée dans les modules *marcel_signal.c* et *marcel_signal.h*. Ces modules gèrent tout ce qui concerne les signaux linux. La mise en place d'un traitant, le stockage des signaux en suspens, la gestion

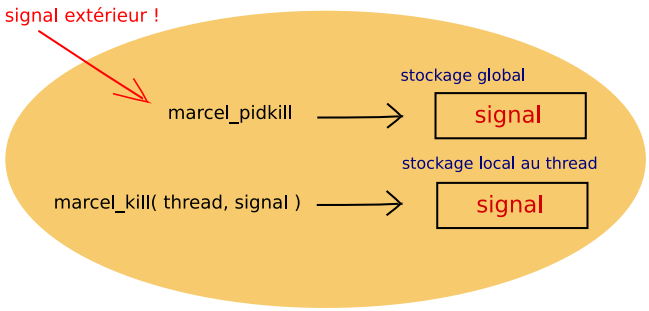
des masques de signaux, l'appel du traitant (ou du traitement par défaut) et les autres fonctions liées aux signaux y sont implémentées.

Que ce soit au niveau du processus entier (global), ou sur chaque thread (local), il est nécessaire de stocker des informations comme les signaux en attente, les signaux bloqués. Pour le processus entier, ces informations sont stockées de manière globale (variable globale dans le module), sinon pour chaque thread, de manière locale (dans la structure de thread).

La fonction *sigaction* permet d'associer une action précise (appel de traitant, etc) lors de la réception d'un signal donné. Elle prend en paramètre un numéro de signal et une structure *struct sigaction*. Cette structure permet de stocker différentes informations comme le traitant à appeler lors de la délivrance d'un signal, ou encore son masque. Un masque permet de bloquer la délivrance des signaux contenus dans celui-ci. Nous avons implémenté la fonction *sigaction* de telle sorte que la structure *struct sigaction* entrée en paramètre soit stockée au sein de notre tableau global. Quand un signal reçu sera délivré (avec la fonction *marcel_deliver_sig*), c'est cette struture *struct sigaction* qui sera utilisée afin de déterminer quelle action accomplir (appeler un traitant, ignorer le signal, action par défaut...). Une structure sigaction différente est stockée pour chaque signal.

La fonction *pthread_kill* permet d'envoyer un signal à un thread donné en paramètre. Après avoir pris un mutex sur le champ des signaux en attente du thread ciblé (verrouillage), il ajoute à ce champ le signal placé en paramètre, puis relâche le mutex. Dans le cas d'un signal externe au processus qui est reçu, c'est le noyau qui s'en charge. Une fonction spécifique à l'ordonnanceur Marcel s'en aperçoit et appelle la fonction *marcel_pidkill* qui ajoute le signal au champ des signaux en suspens. Ce champ est global au processus.

FIG. 3 – Arrivée d'un signal



La fonction *marcel_sigtransfer* permet de transférer un signal stocké au niveau global dans un thread. Le signal sera donc stocké localement dans le tableau des signaux en suspens du thread. C'est à partir de ce thread que le signal sera réellement délivré.

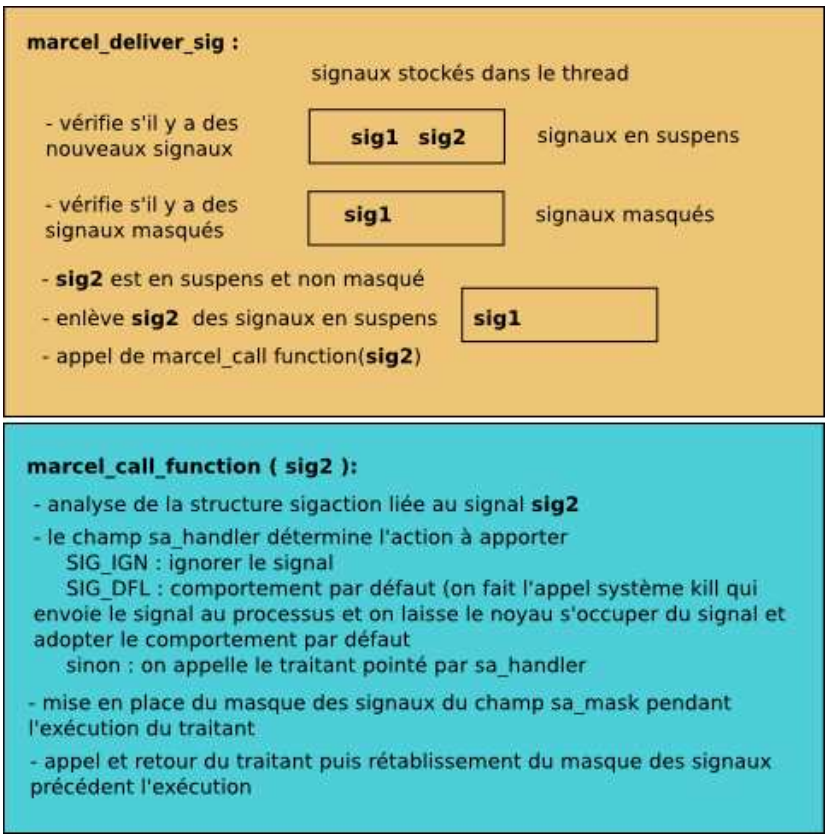
Délivrance d'un signal :

Dans divers endroits, la fonction *marcel_deliver_sig* est appelée. Elle se charge de regarder s'il y a bien des signaux dans le champ des signaux en suspens du thread. S'il y en a un et que celui-ci n'est pas masqué (le champ des signaux masqués est stocké dans la structure du thread), il sera bien délivré puis le signal sera effacé.

C'est la fonction *marcel_call_function* qui est appelée et qui permet d'associer au signal délivré l'action adéquate. Cette action est stockée dans le champ *sa_handler* de la structure *struct sigaction* stockée de manière globale au processus. De manière générale, trois comportements différents sont attendus :

- le signal est ignoré, donc *marcel_call_function* retourne.
- le comportement par défaut est adopté (nous appelons l'appel système *kill* qui se chargera lui-même de la gestion du signal délivré au niveau noyau).
- un traitant est appelé avec son masque de signaux adéquat (ce masque est stocké dans le champ *sa_mask* de la structure *struct sigaction*). Nous mettons en place le masque adéquat, puis nous appelons le traitant demandé et lorsqu'il retourne, nous rétablissons le masque précédent.

FIG. 4 – Délivrance d'un signal



Flags sigaction

Lors de la mise en place d'un traitant, la structure *struct sigaction* contient un champ contenant des « flags. » Ces « flags » influent sur la façon dont on traite le signal. Le plus important est le « flag » *SA_SIGINFO*. Il nous demande de stocker plus d'informations concernant le signal reçu. Ces informations sont stockées dans le champ *sa_siginfo* et le traitant n'est plus stocké dans le champ *sa_handler* mais dans le champ *sa_sigaction*. Tout au long du module nous regardons si le flags *SA_SIGINFO* est mis afin de savoir si nous devons prendre

en compte le champ *sa_siginfo* et choisir comme traitant de signal *sa_sigaction* à la place du champ *sa_handler*.

Fonctions de gestion de champs de signaux

Les ensembles de signaux en suspens et des signaux masqués sont de type *marcel_sigset_t*. Nous avons implémenté les fonctions qui gèrent les modifications de ces ensembles. *sigaddset* permet d'ajouter un signal à un ensemble, *sigismember* permet de vérifier si un signal est bien présent dans un ensemble. Nous avons utilisé un champ de 32 bits où un bit représente la présence d'un signal (iéme bit mis à un si le signal numéro i est présent). Ces fonctions simples (basées sur des opérations logiques) ont été implémentées avec des macros. Cela évite d'effectuer un appel de fonction.

Les fonctions de type sigwait

sigtimedwait, *sigwaitinfo* et *sigwait* qui attendent l'arrivée d'un signal. Elles prennent en paramètre un ensemble de signaux de type *marcel_sigset_t* et un pointeur. Lorsqu'un signal appartenant à cet ensemble est en suspens, celui-ci ne sera pas délivré mais les informations sur ce signal seront stockés à l'adresse du pointeur. Ces informations seront stockées, soit sous la forme d'une structure *siginfo_t* (*sigtimedwait*, *sigwaitinfo*) ou d'un entier (*sigwait*). Le thread appelant l'une de ces trois fonctions regarde si le signal est déjà en suspens, et dans le cas contraire, s'enregistre dans une file d'attente de signal (ainsi que l'adresse où les informations devront être stockées), et s'endort en attendant que le signal voulu arrive.

Les fonctions *marcel_distribwait_sigext* et *marcel_distribwait_thread* permettent, avant l'ajout d'un signal dans le set des signaux en suspens, de vérifier qu'un thread n'est pas en train de faire un *sigwait* (par exemple) sur un set contenant ce signal. Si c'est le cas, les informations concernant ce signal seront stockées à l'adresse donnée par le thread en attente, ce dernier sera enlevé de la file d'attente, et le signal ne sera pas mis dans le set des signaux en suspens. Une fois qu'un signal dans le set a bien été reçu, le thread appelant *sigtimedwait*, *sigwaitinfo* et *sigwait* retourne après avoir stocké les informations à l'adresse du

pointeur. Si aucun signal ne vient dans un temps donné, *sigtimedwait* retourne une fois le temps écoulé.

Voici les grandes lignes de la manière dont a été implémentée la gestion de signaux. Ce fut le module le plus difficile et le plus long à écrire en raison des intéractions avec le noyau, de la gestion des accès concurrents, des variables de stockage qu'il a fallu déterminer et de la norme précise qu'il a fallu respecter.

3.3.5 Les autres modules

Le module marcel_threads.c

Ce module gère tout ce qui est relatif à l'exécution d'un thread. Il contient des fonctions liées aux attributs de threads comme la politique d'ordonnancement la priorité (qui est stockée dans une structure *struct marcel_sched_param*).

La principale difficulté de ce module concerne les conversions des politiques d'ordonnancement POSIX qui sont différentes de celles de Marcel. Il en est de même pour les priorités. Dans POSIX, les politiques ont pour nom *SCHED_FIFO*, *SCHED_RR* et *SCHED_OTHER*. Dans Marcel, l'équivalent de *SCHED_FIFO* et de *SCHED_RR* est le même, excepté que dans un cas la préemption est désactivée et pas dans l'autre. Pour déterminer une politique POSIX à partir de celle de Marcel, on doit regarder dans quel intervalle de priorité Marcel on se trouve (car *SCHED_OTHER* a toujours 0 comme priorité). On regarde ensuite si la préemption est activée afin de différencier une politique *SCHED_FIFO* d'une politique *SCHED_RR*.

De plus, des contrôles d'erreur ont été ajoutés à des fonctions préexistantes comme *pthread_join* ou *pthread_detach* et j'ai dû implémenter aussi des fonctions d'annulation comme *pthread_cancel* et *pthread_testcancel*. *pthread_cancel* teste si un thread peut être annulé (en regardant si son champ d'*état* d'annulation est mis à *« ENABLE »* et que son champ de *type* d'annulation est mis à *« ASYNCHRONOUS »*). Si le thread peut être annulé, il se termine. Sinon, le champ « cancelled » du thread est mis à 1, et lorsque *pthread_testcancel* sera appelé et si les état et type d'annulation seront adéquats (« ENABLE » et « ASYNCHRONOUS »), le thread s'arrêtera.

Le module marcel_attr.c

Ce module gère tous les attributs de threads. Ce sont ces attributs qui donneront les propriétés d'un nouveau thread créé avec un appel de *pthread_create*. Nous avons deux types de champs : les champs POSIX (qui sont dans la structure *pthread_attr_t*) et les autres (qui ne le sont pas). Ci-dessous, nous retrouvons les différents champs POSIX.

- *int __detachstate ; // détachement d'un thread*
- *int __schedpolicy ; // politique d'ordonnancement d'un thread*
- *struct marcel_sched_param __schedparam ; // paramètre d'un thread (priorité le plus souvent)*
- *int __inheritsched ; // héritage d'un thread*
- *int __scope ; // permet de déterminer si le thread créé sera un thread système (dans le noyau) ou un thread utilisateur*
- *size_t __guardsize ;*
- *int __stackaddr_set ; // tout ce qui est en relation avec la pile*
- *void *__stackaddr ; // adresse de la pile d'exécution du thread*
- *size_t __stacksize ; // taille de la pile d'exécution du thread*

Les fonctions de ce module sont principalement des mutateurs et des accesseurs aux champs de l'attribut entré en paramètre. On peut avec ces fonctions choisir le type d'héritage, la priorité, la politique d'ordonnancement et le type de thread à créer (utilisateur ou non). Ces paramètres deviendront effectifs lors de l'appel à *pthread_create*. C'est dans le module *marcel_glue_pthread.c* que cette fonction est implémentée. Si le champ « *__inheritsched* » de l'attribut est mis à *PTHREAD_SCHED_INHERIT*, le reste de la structure *pthread_attr_t* sera ignoré. On récupère alors les paramètres du thread courant (le père) qu'on met dans une nouvelle structure *marcel_attr_t*, et c'est cette dernière qui sera entrée en paramètre de la fonction *marcel_create*. De même, si le champ *__scope* est mis à *PTHREAD_SCOPE_SYSTEM*, on ajoute des informations dans la structure *marcel_attr_t* qui permettent de créer un thread système, avant d'appeler *marcel_create*. Si aucun attribut est entré en paramètre de la fonction *pthread_create*, on associe au nouveau thread, les attributs par défaut.

Les modules nptl_barrier.c.m4, nptl_mutex.c.m4 et nptl_cond.c.m4

Un fichier m4 est un préprocesseur générique que nous utilisons afin de dupliquer du code. On écrit les fonctions à l'aide du préfixe « prefix » qui peut être remplacé par *marcel*, *pmarcel* et *lpt* (au choix). A la compilation, le fichier C est créé avec les fonctions adéquates.

J'ai totalement implémenté le fichier *nptl_barrier.c.m4*. Il gère l'implémentation des barrières de synchronisation. Lorsque qu'un thread appel un *pthread_barrier_wait* et arrive à une barrière de type *pthread_barrier_t*, un compteur est incrémenté, puis le thread s'endort. Quand le n-ième thread arrive à cette barrière (toujours dans la fonction *pthread_barrier_wait*), si le champ *init_count* de la structure *pthread_barrier_t* vaut n, cela veut dire qu'on a atteint le nombre de threads que l'on voulait à la barrière. A ce moment, on réveille tous les threads qui attendent à la barrière. La fonction *pthread_barrier_wait* a été implémentée en deux parties : une partie « begin » et une partie « end », cela permet au thread qui est sensé s'endormir de faire autre chose, par exemple du calcul, en attendant que la barrière se termine. Dans ce cas, il faut appeler *pthread_barrier_wait_begin* et *pthread_barrier_wait_end* séparément.

Dans le module *nptl_mutex.c.m4*, j'ai surtout ajouté de la gestion d'erreur et remis en route la fonction temporelle *pthread_mutex_timedlock*. Dans cette fonction, j'ai crée une boucle de temporisation. J'initialise une variable *timeout* en millisecondes qui dépend de la structure temporelle entrée en paramètre. Un appel à la fonction existante *ma_schedule_timeout(timeout)* après un appel *ma_set_current_state(MA_TASK_INTERRUPTIBLE)* a pour effet d'endormir le thread courant pendant la durée *timeout*. Si le thread est réveillé avant que la durée du *timeout* soit écoulée, la durée restante sera retournée. On refait la boucle tant que la durée restante est supérieure et différente de zéro. Une fois sorti de la boucle, en testant si la durée qui restait est égale à zéro, on sait si on a eu le temps de prendre le mutex. On le prend s'il restait du temps à s'écouler. Sinon on retourne l'erreur *ETIMEDOUT* si le temps s'est écoulé avant qu'on ait pu prendre le mutex. J'ai réutilisé la même méthode de temporisation dans les fonctions temporelles de *nptl_cond.c.m4* (*pthread_cond_timedwait*) et de *pthread_rwlock.c* (*pthread_rwlock_timed(wr/rd)lock*).

Le module marcel_sched_generic.c

Ce module regroupe des fonctions temporelles telles *sleep*, *usleep* et *nanosleep*. On utilise aussi la fonction déja implémentée *ma_schedule_timeout* après avoir initialisé la valeur de *timeout* à partir des données entrées en paramètre. La fonction *ma_schedule_timeout* retourne quand le temps d'endormissement est écoulé ou alors si un signal a été reçu. Ceci est exactement le comportement attendu d'un fonction de type *sleep*.

Le module pthread_todo.c

Ce module regroupe les fonctions non implémentées. Lors de l'appel de ces fonctions, un message d'erreur est affiché et la fonction retourne. Au fur et à mesure que j'ai implémenté des fonctions, le nombre de fonctions dans *pthread_todo.c* a diminué.

Le fichier libpthread.map

Afin de créer notre bibliothèque libpthread.so, les fonctions implémentées doivent être reliées avec la bonne version de la *« glibc »* (bibliothèque C). C'est à ceci que sert le fichier *libpthread.map*. Il en existe un par architecture. Plusieurs blocs représentant chaque version de la « *glibc* » sont écrits dans ce fichier. Il faut placer les noms de fonctions dans les bons blocs, afin que ceux-ci soient bien reconnus lorsque les applications utilisent notre librairie.

Le header pthread.h

Si l'on veut recompiler nous-même les programmes avec notre interface *pmarcel* (API POSIX et ABI non POSIX), nous avons besoin que l'application appelle directement les fonctions *pmarcel*. J'ai donc écrit un fichier *pthread.h*. En modifiant le script de compilation, il est possible d'utiliser notre header *pthread.h* au lieu du header */usr/include/pthread.h*. Il est essentiellement composé de *#define pthread_divers pmarcel_divers*.

3.3.6 Conclusion de la conception détaillée

Je vous ai présenté la manière dont a été implémentée la bibliothèque libpthread.so. Il reste sûrement des détails que je n'ai pas évoqués mais une grande partie des choix d'implémentation utilisés a été expliquée ci-dessus.

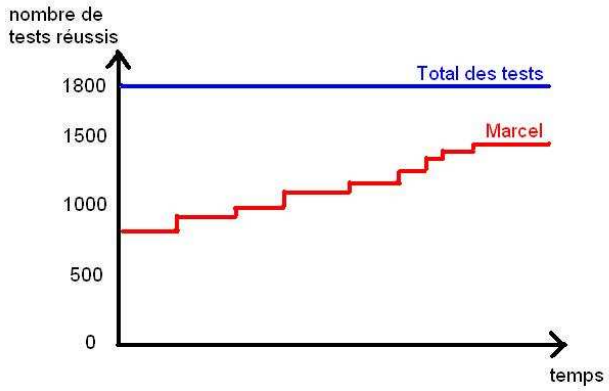
4 Plan de test de conformité

4.1 Batterie de test Posixtests

Afin de pouvoir vérifier la compatibilité POSIX des fonctions que nous avons implémentées, nous avons utilisé une suite d'environ 1800 tests (la suite posixtests). Ces tests vérifient le comportement, les valeurs de retour, les codes d'erreur que renvoient les fonctions et les comparent à la norme POSIX. Ces tests ont permis de relever un bon nombre d'erreurs d'implémentation et de les corriger. Certains comportements ne sont pas supportés par l'ordonnanceur Marcel (verrous partagés, etc). En conséquence les tests correspondants ne sont pas réussis.

Sur l'ensemble des tests environ 1400 réussissent (environ 1580 pour la NPTL, la librairie du système), les autres étant des fonctions non implémentées ou des options non supportées. Lors du lancement de la première batterie de tests, nous avions environ 800 tests réussis, ce qui nous donne une augmentation assez conséquente. Je pense que l'utilisation de cette suite de tests a été déterminante dans le projet.

FIG. 5 – Evolution des tests POSIX réussis



De plus, j'ai écrit en parallèle quelques petits programmes de tests en java, afin de vérifier que la « JVM Sun Microsystem » commençait bien à touner un petit peu sur Marcel. Ces tests, qui utilisaient plusieurs threads lancés en parallèle, ont été considérés comme réussis.

4.2 Compilation des tests POSIX avec Marcel

En changeant le script de compilation, il est possible de recompiler tous les tests POSIX avec le header *pthread.h* de Marcel qui appelle des fonctions de type pmarcel. Cela permet aux utilisateurs d'utiliser l'ABI Marcel tout en gardant la sémantique (l'API) POSIX, s'ils disposent du code source. Nous obtenons des résultats légèrement meilleurs en procédant ainsi.

5 Validation et résultats (début octobre)

Le critère de validation était de faire tourner la JVM de Sun Microsystem sur Marcel. Au fur et à mesure que les tests avançaient, nous avons testé le lancement d'un petit jeu de monopoly écrit en java qui comportait plusieurs

threads. C'est dès lors que le monopoly s'est mis a tourner que nous avons constaté que Java commençait à tourner sur Marcel. La grande surprise a été de lancer Mozilla Firefox avec Marcel et de constater qu'il fonctionnait, même en utilisant des applications *flash* et *Java* avec lancement de plusieurs threads (ouverture d'un chat écrit en java par exemple).

Nous avons observé de bons résultats. En effet, nous avons vu que l'ordonnanceur Marcel pouvait lancer des applications tel Mozilla Firefox, Thunderbird ou la Sun Microsystem JVM. La grande satisfaction a été de voir aussi tourner OpenOffice.org 2.0.3 sur Marcel, résultat auquel on ne s'attendait pas. On peut donc affirmer que l'objectif de ce stage a été atteint. Il aurait été intéressant de tester Marcel sur Apache, mais faute de temps, cela n'a pu se faire.

6 Conclusion

Un bon nombre d'applications normées POSIX tournent maintenant sur Marcel. Cela crée un tournant pour Marcel qui peut maintenant être utilisé directement avec des logiciels précompilés. Cela ouvre de nouvelles perspectives de tests et de développement car le problème qui a longtemps existé, c'est à dire que Marcel ne peut pas être testé directement avec des binaires, n'est plus d'actualité. L'équipe Runtime poursuit actuellement son travail en utilisant Marcel avec des logiciels de parallélisme normés POSIX comme OpenMP par exemple.

7 Glossaire

- thread : Un thread est un fil d'exécution, c'est à dire une séquence de code en train d'être exécutée par un processeur. Il peut y avoir plusieurs threads au sein d'un même processus.
- thread système : thread de niveau noyau qui peut être exécuté par un processeur
- CEA (Commissariat de l'Energie Atomique) : Centre de recherche sur l'énergie nucléaire (déchets, sûreté et sécurité), la recherche technologique et la santé.
- Bull : L'entreprise Bull conçoit et développe des serveurs, des logiciels et des services pour des environnements ouverts intégrant les technologies

avancés.

8 Bibliographie

- Site internet de l'INRIA : *http ://www.inria.fr/*
- Site Internet de l'équipe Runtime : *http ://runtime.bordeaux.inria.fr/*
- Site Internet du projet pm2 : *http ://gforge.inria.fr/project/pm2/*
- Site Internet de la norme SUSV3 : *http ://www.unix.org/version3/*
- Site Internet des tests POSIX : *http ://sourceforge.net/projects/posixtests/*
- Site Internet de l'entreprise Bull : *http ://www.bull.com/fr/*
- Site Internet du CEA : *http ://www.cea.fr*