Projet de Systèmes d'Exploitation

donné le 28 novembre 2011, à rendre le 10 janvier 2012

1 Préambule

Le thème de ce projet est l'exécution « contrôlée » d'un processus pour pouvoir faire tourner un programme pas à pas, inspecter ses variables. On peut le faire pour des bonnes raisons (un débogueur) où pour des mauvaises raisons (un virus). Parmi les outils très répandus qui offrent toutes ces facilités on retrouve le débogueur `gdb`, ainsi que la commande `strace`.

L'avant-projet vous a familiarisé avec le format ELF et avec l'appel système `ptrace` qui est utilisé par `gdb` et par `strace`. On vous demande maintenant d'implementer l'appel système `ptrace` dans un cadre simplifié. Le projet comporte toutes les étapes du développement logiciel : la compréhension de la base existante, la rédaction d'un cahier de charges, le développement proprement-dit, et la rédaction du rapport.

La base existante comprend un mini-émulateur de CPU et noyau. Sa partie d'analyse lexicale et le cœur de l'exécution vous sont fournis en état de fonctionnement.

2 Modalités

Des échéances seront indiqués pour chaque étape du projet. Vous travaillerez en quadrinôme : un chef de projet, un développeur, un développeur débutant et un stagiare. Chacun a son rôle. Vous devez rendre un compte-rendu commun sur la compréhension de la base existante, un cahier de charges individuel, le code développé dûment commenté, un rapport commun sur le code développé, et effectuer une soutenance avec présentation, démonstration et session de questions. Le format des fichiers rendus de type source doit être .cxx ou .h . Le format des fichiers rendus de type documentation doit être .pdf ou .txt. Le nom du fichier contenant le rapport doit contenir les noms de tous les auteurs (le nom du chef de projet en premier). Le tout doit être archivé dans une archive .tar.gz. Le nom de l'archive doit contenir les noms de tous les auteurs (le nom du chef de projet en premier). La liste des chefs des projets et des développeurs est affiché sur la page web du projet. Si vous estimez que vous devrez vous retrouver parmi eux envoyez un courriel à votre enseignant qui sera ravi de vous rajouter sur les listes.

Important Vous devez ABSOLUMENT respecter l'ordre des étapes et leur cahier de charges. Toute étape non-traitée sérieusement fait que les étapes suivantes ne seront pas du tout considérées, au dépens de votre note globale. N'hésitez pas à poser des questions, surtout par courrier électronique. Commencez à travailler tôt, pour avoir du temps, et ne bloquez pas trop sur un point. Faites beaucoup de tests pour mieux tout comprendre, et essayez de déduire un maximum avec un minimum d'investigation, en vous arrêtant lorsque vous avez assez compris pour avancer.

3 Projet– but du jeu et présentation

Vous devez télécharger depuis la page web du projet un paquetage contenant le code fourni.

3.1 But du jeu

Vous devez rajouter des facilités pour le déboguage, similaires à une partie de ce que `gdb` offre. Si vous n'avez pas déjà utilisé `gdb`, testez-le brièvement.

Soit le programme suivant, dans le fichier `petitProg.cxx`, compilé avec `-ggdb` et linkédité dans `petitProg.run` et puis lancé depuis le shell avec l'invocation `gdb petitProg.run` :

```
int main() {
    int a,b,c;
    a = 56;
    b = 200;
    c = a + b;
    a = 0;
    while(c > 1) {
```

```
        a += 1;
        c = c / 2;
    }
    return 0;
}
```

L'exemple de dialogue avec gdb qui suit vous montre les **entrées** de l'utilisateur et les *réponses de gdb*.

```
(gdb) break petitProg.cxx:1
Breakpoint 1 at 0x80483ac: file petitProg.cxx at line 1
(gdb) run
Starting program: /.../petitProg.run
Breakpoint 1, main () at petitProg.cxx:1
1     int main() {
(gdb) step
3     a = 56;
(gdb) step
4     b = 200;
(gdb) break petitProg.cxx:9
Breakpoint 2 at 0x80483f3: file petitProg.cxx at line 9
(gdb) continue
Continuing.
Breakpoint 2, main () at petitProg.cxx:9
9     c = c / 2;
(gdb) print a
$1 = 1;
(gdb) display c
1: c = 256;
(gdb) display a
2: a = 1;
(gdb) step
7     while(c > 1) {
2: a = 1;
1: c = 128;
(gdb) continue
Continuing.
Breakpoint 2, main () at petitProg.cxx:9
9     c = c / 2;
2: a = 2;
1: c = 128;
(gdb) continue
Continuing.
Breakpoint 2, main () at petitProg.cxx:9
9     c = c / 2;
2: a = 3;
1: c = 64;
(gdb) delete 2
(gdb) continue
Continuing.
Program exited normally.
(gdb) quit
```

On comprend ainsi comment on peut suivre pas-à-pas l'exécution d'un programme, examiner (et pourquoi pas changer) les valeurs de ses variables, et le faire avancer plusieurs pas si désiré. Le paquetage que vous téléchargerez vous fournit un interpréteur de minilangage (qui sera présenté ci-après), avec un ordonnanceur et une notion de processus, le tout en état de fonctionnement. Votre tâche est de lui rajouter les fonctionnalités illustrées avec cet exemple de gdb. Dans ce projet on réalise un *support noyau pour le suivi de processus* (**trace**, en anglais) pour le minifragment de noyau très simpliste ainsi fourni, mais il n'y aura pas de notion de *autre processus* qui débogue. Ce sera un dialogue « direct » avec l'environnement – comme si vous aviez un shell spécial discutant directement avec le noyau – pour simplifier.

3.2 Fonctionnalités à rajouter – présentation générale

Vous devez rajouter les facilités pour
- l'exécution pas à pas – démarrage, arrêt, continuation
- l'inspection de variables – une fois (comme le `print` de `gdb`) ou rappelées à chaque pas (comme le `display` de `gdb`)
- les points d'interruption (comme les `break` de `gdb`)
- si tout cela est réalisé, des extensions supplémentaires pour aller plus loin

Pour ce faire, vous devez surtout comprendre les structures de données utilisées – la partie code d'exécution, malgré son nombre de lignes, n'est pas du tout essentielle pour le gros de votre travail.

3.3 Présentation du paquetage

Ce paquetage fournit la classe `ProcInfo` de l'espace de noms `ProcDebug`, qui permet la lecture, analyse, interprétation et exécution de programmes écrits dans un mini-langage très simple et facile. Le but principal est de rajouter des fonctionnalités au paquetage, de préférance dans une AUTRE CLASSE, dans laquelle vous pouvez mettre le contenu de `proj.cxx` (fourni dans le paquetage) qui est votre « point de départ ». Le fichier `ProcDebug.h` contient les déclarations importantes, et les 220 premières lignes de `ProcDebug.cxx` devront étre également étudiées.

Vous avez toute liberté de modifier n'importe quelle partie du code fourni dans `ProcDebug.*`, mais ne le faites que si vous êtes absolument sûrs de ne pas pouvoir faire autrement. Il pourrait éventuellement y avoir des choses à modifier dans ces 220 premières lignes de `ProcDebug.cxx`.

3.4 Premiers pas

Téléchargez le paquetage et décompressez-le. Allez dans `dirproj` et faites `make nom=proj`. Testez ensuite avec `./proj.run tst/tst1.0.m 0`, vous devez voir "Hello world".

3.5 Le mini-langage – première partie

Le paquetage, une fois téléchargé et compilé, vous permet donc aussitôt d'exécuter de programmes, dont vous avez également des exemples dans le sous-répertoire `tst`.

Regardons le mini-langage, petit à petit. Le classique

```
PROGRAM
PRINT @ "Hello world\n"
ENDPROGRAM
```

a l'air plutôt sympatique. Par rapport au C/C++, remarquez l'absence de '`;`'. Avons-nous droit aux variables ? Oui :

```
PROGRAM
NEW   @ a : 5
NEW   @ b : 10
PRINT @ "\"a\" vaut ",a," et 'b' vaut ",b,"\n"
ENDPROGRAM
```

qui donne

```
"a" vaut 5 et 'b' vaut 10
```

On commence à comprendre : '`@`' sépare le mot-clé (« verbe ») du reste de l'instruction, et le '`:`' est l'opérateur d'assignation. Combien de tels verbes y a-t-il dans notre mini-langage ? Très peu, en voici une bonne partie :
- `PROGRAM` et son companion `ENDPROGRAM`, une seule fois
- `NEW` pour déclarer et initialiser une variable. Il n'y a pas vraiment de pointeurs à proprement parler (enfin, si, mais on verra cela plus tard)
- `READ` pour lire au clavier : `READ @ a` lit *dans* `a` – pas de notion d'adresse de `a` (il n'y pas d'appel de fonction dans notre mini-langage). À gauche du '`:`' on met une "left-value" – concepte bien connu du C/C++.

– `COMPUTE` pour calculer une expression comportant strictement un et un seul opérateur arithmétique, ou de comparaison.
– `COPY` pour copier le contenu d'une variable dans une autre
– `PRINT` pour afficher à l'écran.

Avec ce vocabulaire on peut déjà écrire des choses plus compliquées, comme par exemple le test `tst/tst2.2.m`

```
PROGRAM
NEW     @ a : 0
NEW     @ b : 0
NEW     @ c : 0
PRINT   @ "Entrez deux nombres entiers... "
READ    @ a
READ    @ b
COMPUTE @ c : a + b
PRINT   @ "La somme de ",a," et ",b," vaut ",c,"\n"
COMPUTE @ c : a / b
PRINT   @ "Le quotient de ",a," par ",b," vaut ",c,"\n"
ENDPROGRAM
```

qui, lancé avec `./proj.run tst/tst2.2.m 0`, donne ce à quoi on s'attend bien

```
Entrez deux nombres entiers... 10
2
La somme de 10 et 2 vaut 12
Le quotient de 10 par 2 vaut 5
```

on peut mettre autant d'éspaces qu'on veut, l'important est de ne pas oublier les opérateurs. `COMPUTE` n'effectue qu'une seule opération, sans paranthèses. Si vous rentrez zéro pour `b`, l'environnement se rend compte au moment de l'exécution du `COMPUTE` et arrête le tout.

```
Entrez deux nombres entiers... 3
0
La somme de 3 et 0 vaut 3
RUN ERROR Division by zero in '/', tst/tst2.2.m:10
```

Arrêtons-nous ici pour le langage, et décrivons en détail ce que vous devez réaliser. D'ailleurs, un conseil très important : le long du travail, utilisez au maximum les exemples fournis, et préoccupez-vous le moins possible d'écrire d'autres programmes en ce mini-langage, sauf si vous avez tout fini. Vous disposez d'un nombre assez grand d'exemples, suffisants pour la plupart des cas à tester. Pour comprendre ce qui ne va pas lors d'erreurs de syntaxe pour vos éventuels programmes, exécutez avec un niveau de verbosité 3 et référez-vous à la fin de ce document, pour la syntaxe exacte.

4 Travail à faire – étapes, délais

Le travail demandé est découpé en étapes, que vous devez suivre scrupuleusement, sous peine de ne pas voir compter votre travail qui suivrait une étape manquante. Essayez de respecter les délais. Si vous avez un retard de plus d'une semaine pour le développement votre enseignant ou un autre chef de projet peut prendre le rôle de chef de votre projet. (Vous devrez l'annoncer à votre enseignant et vous devrez payer votre nouveau chef de projet avec des points déduits de votre note finale.)

4.1 Schéma de conception diagramme des états - à finir avant le 5.12.2011

Les processus se trouvent chacun dans un état – en attente dans la file, en exécution, en attente entrées/sorties, etc. Pour notre but, on considère également des états comme « en cours de déboguage » – ceux avec « `TRACE` » dans leur noms. Voici la liste complète du paquetage.

```
enum ProcStatus {
    STAT_WAITING, STAT_RUNNING,
    STAT_IOWAIT,
    STAT_MUTEXWAIT, STAT_MUTEXGRAB, STAT_NOMUTEX,
    STAT_SYS, STAT_TERMINATED,
    STAT_TRACEEND, STAT_TRACESTEPRUN
};
```

Considérant seulement `STAT_WAITING`, `STAT_RUNNING`, `STAT_TERMINATED`, `STAT_TRACEEND` et `STAT_TRACESTEPRUN`, concevez le diagramme de changement d'un état vers un autre (flêches et conditions) pour un processus quelconque dans notre mini-environnement.

4.2 Interruption d'exécution normale – à finir avant 17.12.2011

Le fichier `proj.cxx` (dans `dirproj`) contient un exemple très simple de « moteur » de notre environnement (ou plutôt centre de commande). Le but de cette étape et des suivantes est son extension graduelle, le mettant éventuellement dans une classe si le cœur vous en dit.

Rajoutez

- la possibilité d'appuyer sur `Ctrl \` (donc d'envoyer un `SIGQUIT`) afin d'interrompre l'exécution en cours,
- une manière de la reprendre de là où elle a été interrompue – comme la commande `continue` de `gdb`.

Contraintes à respecter, dans cette étape et dans toutes les étapes suivantes.

1. n'interrompez pas l'instruction du mini-langage qui est en cours ! Ces étapes sont atomiques du point de vue de l'ordonnanceur fourni, et doivent le rester telles quelles. Vous devez donc vous protéger de l'arrivée de ce signal, noter son arrivée, et la prendre en compte entre deux instructions du mini-langage
2. n'empêchez pas l'ordonnanceur d'élire un processus : c'est en manipulant les états du processus « visé » (à vous de décider comment – regardez ces 220 premières lignes de `ProcDebug.cxx` pour mieux comprendre), que vous réalisez ce travail
3. manipulez judicieusement les états, conformément au diagramme rédigé au point précédent. N'hésitez pas à contacter l'enseignant en cas de doutes.

Idéalement vous allez construire un micro-interpréteur très simple de quelques commandes de déboguage, un peu comme `gdb`. Par exemple, une fois le `SIGQUIT` pris en compte, un prompt spécial, mettons `mDbg>`, peut être affiché, et la commande **`continue`** être comprise pour continuer l'exécution :

```
shell> proj.run petitProg.m 0
.....
Ctrl+\
Interrupted at line 4
mDbg> continue
Continuing from line 4
.....
Program ended
mDbg> quit
shell>
```

La partie « interface » doit vous prendre un temps minimal de réalisation, elle n'est aucunément importante de par son aspect, mais uniquement de par les fonctionnalités offertes, qui vous sont précisément décrites ici.

Enfin, pour vous faciliter le travail, il existe une donnée membre booléenne de la classe `ProcInfo` qui s'appelle `qDoSleepAfterEachInstruction`, qui, lorsqu'elle est mise à vrai, fait un `sleep(1)` à la fin de l'exécution de chaque instruction.

Pour le moment, faites abstraction de la possibilité de lancer plusieurs processus, afin d'avancer dans votre travail. Autrement dit, ne vous préoccupez pas de la manière de *choisir* le processus à interrompre. On verra cela plus tard. Pour les tests, ne lancez qu'un seul `tst.m` (ou programme de votre choix).

4.3 Exécution pas à pas – à finir avant 17.12.2011

Rajoutez maintenant la commande **`step`** au débogueur, pour ne faire qu'un seul pas. De nouveau, faites attention aux états, n'interrompez pas l'ordonnanceur, etc. – exactement les mêmes contraintes. Restez similairement dans la situation où un seul programme est lancé dans le mini-environnement. Affichez bien entendu au moins les numéros des lignes exécutées, au fur et à mesure.

4.4 Affichage du contenu des variables – à finir avant 17.12.2011

La table de symboles est disponible dans la donnée-membre `symbolTable` de la classe `ProcData`, celle qui regroupe tous les renseignements sur un processus (comme dans le TP sur l'ordonanceur). Cette table de symboles contient les identifiants ainsi que leur valeurs en cours – donc en permanence mise à jour. C'est là que les `READ` ou `COMPUTE` écrivent et de là que les `PRINT`, etc. lisent. Vous disposez également de `Name2ProcSymbolIndex` pour retrouver un indice dans `symbolTable` à partir du nom d'une variable.

Rajoutez la possibilité d'afficher le contenu de variables dont on précise le nom (disponible une fois interrompue l'exécution). Rajoutez cette possibilité de deux manières : soit une seule fois, comme le **`print`** de `gdb`, soit réaffichée après chaque pas, dans le mode d'exécution pas à pas, comme le **`display`** de `gdb`.

Vous pouvez vous inspirer du code de la fonction `displayProcInfo` de la classe `ProcInfo`. Cette fonction est activée lors d'une exécution en niveau de verbosité 4 ou 5 de `proj.run` tel qu'il est fourni dans le paquetage. [1]

Les mêmes contraintes sont toujours en vigueur.

4.5 Points d'interruption – à finir avant 5.01.2012

Rajoutez maintenant des points d'interruption (*breakpoints* en anglais), spécifiables au débogueur (similairement une fois interrompue l'exécution) sous la forme `<nom de fichier>:<numero de ligne>`, tout comme la commande **`break`** de `gdb`.

La fonction `findCrtInstruction()` de la classe `ProcInfo` pourrait servir pour savoir où vous vous trouvez lors de chaque tour de boucle d'exécution. On doit pouvoir les poser, ou les enlever.

Respectez les mêmes contraintes.

4.6 Terminaison, redémarrage, lancement pas-à-pas – à finir avant 6.01.2012

Rajoutez la possibilité de terminer le programme (même s'il est seulement interrompu), ainsi que celle de le redémarrer. Rajoutez également une option pour démarrer le programme en mode pas-à-pas dès le début.

4.7 Facilités supplémentaires optionnelles – à finir avant 7.01.2012

Maintenant que avez réalisé tout cela, vous pouvez choisir une ou plusieurs tâches restantes, sans contrainte d'ordre : on considère tout ceci comme la dernière étape du projet. Si vous avez bien travaillé jusqu'ici et que tout fonctionne, votre rapport brille et vous êtes satisfaits, sachez que vous pouvez avoir une très bonne note tout en vous arrêtant ici. Autrement, voici les possibilités (dont les cinq dernières vous invitent à vous plonger dans tout le code du paquetage).

1. découpez votre débogueur en deux processus communiquant par un pipe. Utilisez des sémaphores afin de pouvoir choisir quel processus interrompre, lorsqu'on en lance plusieurs dans le mini-environnement
2. rajoutez des options de traçabilité pour le reste des fonctionnalités du mini-langage, décrites ci-après (accès mémoire, boucle while, accès mémoire partagée, mutex, fork)
3. faites remonter des codes précis d'erreur pour la terminaison d'un processus
4. faites en sorte qu'il puisse y avoir plusieurs segments de mémoire partagée, et des ensembles de sémaphores comme en Unix

1. À titre d'information, dans `ProcInstruction` qui représente une instruction, la donnée-membre `leftvalue` (ainsi que la donnée-membre `operand`) est justement un indice dans `symbolTable` (respectivement un `vector` d'indices), installé au moment du parsing par `parseProg()` de la classe `ProcData` ; pour les constantes données dans le programme, des symbols nommés `AnnymSym0`, `AnnymSym1`, etc. sont automatiquement crées.

5. rajoutez la possibilité de lire/écrire dans des fichiers, par groupes de `sizeof(int)` à la fois (étendant la syntaxe de `READ` et `PRINT`, etc.)
6. éliminez le busy-waiting, implémentant des files supplémentaires et rajoutant une primitive `WAIT`
7. rajoutez la notion de signal au mini-langage, par exemple avec une primitive `SIGNAL` et son compagnon `ENDSIGNAL` pour donner le code du traitant, une primitive `SIGNALMASK` pour la gestion, etc.

5 Documentation détaillée du mini-langage

Le mini-langage travaille en nombre entiers uniquement, ne manipule pas de chaînes de caractères (autrement que pour l'affichage, où l'on dispose de quelques séquences d'échappement usuelles), mais par contre contient trois fonctionnalités système : un `FORK`, un segment de mémoire partagée et un sémaphore binaire. Les fichiers `tst/tst4.1.m` et `tst/tst4.2.m` implémentent l'exo 04 du TP sur les sémaphores et mémoire partagée.

Le mini-langage ne comporte par contre pas de signaux, ni de primitive `WAIT`, laquelle peut néanmoins être simulée avec les outils présents. On pourrait ainsi écrire également le coiffeur endormi (on remplacerait les signaux par du busy-waiting).

Une propriété très importante est l'atomicité de CHAQUE instruction simple, de par la construction du paquetage.

On peut par exemple émuler en partie la fonctionnalité du `WAIT` utilisant la mémoire partagée : les processus se mettent d'accord sur un emplacement, et comme les lectures et écritures sont atomiques par construction de l'environnement, on met par exemple la valeur 1, et le père attend à ce qu'elle devienne zéro. Le fils y mettra zéro à sa fin [2]. Les exemples du `tst` fournis avec le paquetage illustrent cette méthode.

5.1 Éléments du mini-langage

- nombres (uniquement entiers),
- operations arithmetiques usuelles
- convention C pour les boolens (0 - faux, != 0 vrai)
- operations de comparaison usuelles
- pour faire la negation : 'non a' se calcule en faisant a == 0
- le ET logique est l'addition de nombres positifs ou zero
- le OU logique est la multiplication de nombres positifs ou zero
- variables (noms comme en C/C++, mais sans '_'), uniquement de type entier
- tableaux avec la syntaxe a$2 pour l'equivalent a[2] en C/C++ (a$b etant permis egalement, mais bien entendu 2$b étant interdit)
- constantes chaînes de caractères, pour le `PRINT` uniquement, avec des séquences d'échappement comme \n, \", etc.

L'identificateur '_' est reserve pour les ressources partagées comme la mémoire partagée (un seul segment, disponible pour tous) ou le sémaphore (un seul mutex, également disponible pour tous). Il n'y a pas de requête d'obtention d'une telle ressource, ou de sa destruction.

5.2 Syntaxe du mini-langage

Les espaces blancs ne comptent pas du tout

```
PROGRAM
... instructions (simples ou complexes)...
ENDPROGRAM
```

Chaque programme/processus à simuler comporte des instructions comme suit (à raison d'une seule instruction simple par ligne, leur ≪ verbe ≫ tout en majuscules)

2. par contre, si le fils fini accidentellement, par exemple à cause d'une division par zéro, ce schéma d'émulation ne fonctionne plus

5.2.1 Instructions simples

```
NEW     @ <Variable>       : <nbrOuAutreVariableDejaDefinie>
READ    @ <Variable>
COMPUTE @ <Variable>       : <nbrOuVar> <oper> <nbrOuVar>
COPY    @ <Variable>       : <nbrOuVar>
STORE   @ <Var>$<nbrOuVar> : <nbrOuVar>
LOAD    @ <nbrOuVar>       : <Var>$<nbrOuVar>
PRINT   @ <nbrOuVarOuString>,<nbrOuVarOuString>,...
FORK    @ <Var>
MUTEX   @ _ : _P
MUTEX   @ _ : _V
```

Le `FORK` dans le père stockera le pid du fils dans la variable fournie, et dans le fils y mettra zéro.

5.2.2 Les opérateurs

Les <oper>ateurs sont $+$, $-$, $*$, $/$, $\%$, $>$, $<$, $>=$, $<=$, $==$, $!=$ (comme en C/C++), et les calculs sont, on le rappelle, tous faits en nombres entiers. Il y a egalement l'opérateur $ pour les tableaux, similaire au [] du C/C++ ainsi : `a$2` correspond à `a[2]`.

Pour le `STORE` ou le `LOAD` on peut également utiliser '_' pour designer la mémoire partagée, par exemple :

```
MUTEX @ _ : P
STORE @ _$2 : alpha
MUTEX @ _ : V
```

5.2.3 Instruction complexe

```
WHILE  @ <nombre> (<nbrEntOuNomVar> <oper> <nbrEntOuNomVar>) REPEAT
    <instruction>
    <instruction>
    ...
    <instruction>
ENDWHILE @ <memeNombreQueCeluiDeCeWHILE>
```

Le nombre qui étiquette un `WHILE` sert non seulement à aider au parsing mais aussi à mieux se reperer dans le programme. On peut bien entendu imbriquer des `WHILE`[3] à condition de respecter l'appariement des nombres-étiquettes. Pour réaliser un 'if', on fait un `WHILE` dont la condition devient fausse avant la première arrivée à son `ENDWHILE`. Le 'else' d'un 'if' est un second `WHILE` qui suit, avec la condition inversée ET une variable vraie si on n'est PAS DU TOUT rentré dans le premier `WHILE` (celui du 'if').

5.3 Remarques

Il ne peut pas y avoir de variable non-initialisée, car le `NEW` se fait toujours avec une valeur. Par contre, la zone mémoire "tas" (`heapMemory`) où l'on accède avec l'opérateur $ n'est PAS INITIALISÉE (et la mémoire partagée non plus). De même, il n'y a pas d'opérateur '&' (adresse d'une variable). L'unité de la `heapMemory` (ainsi que de la mémoire partagée) est l'entier (et non pas l'octet).

Il faut enfin remarquer, du point de vue de la syntaxe, que le seul endroit où les paranthèses apparaissent est dans le `WHILE`. Un `COMPUTE` n'a qu'un seul opérateur et exactement un, qui est binaire, prenant deux opérandes.

3. la structure mémoire qui représente un programme dans `ProcData` avec les `ProcInstruction` est un arbre

5.4 Exemple pour le FORK, illustrant également le 'if'

```
PROGRAM
NEW @ pid : 5
NEW @ qFils : 1
FORK @ pid
NEW @ qPere : pid
NEW  @ waitFils : 1
STORE @ _$1 : waitFils
WHILE @ 1 (qPere) REPEAT
  COPY @ qFils : 0
  PRINT @ "Bonjour, je suis le pere, et mon fils a le pid ",pid,"\n"
  COPY @ qPere : 0
  WHILE @ 12 (waitFils) REPEAT
    LOAD    @ waitFils : _$1
  ENDWHILE @ 12
  PRINT @ "######  Le pere et le fils ont fini  #########\n"
ENDWHILE @ 1
COMPUTE  @ pid : pid == 0
COMPUTE  @ qFils : qFils * pid
WHILE @ 2 (qFils)  REPEAT
  PRINT @ "Bonjour, je suis le fils\n"
  PRINT @ "Le fils a fini.\n"
  COPY  @  qFils : 0
  STORE @ _$1 : qFils
ENDWHILE @ 2
ENDPROGRAM
```