IUT d'Aix-en-Provence
413, avenue Gaston Berger
13625 Aix-en-Provence cedex 1
Tel : 04 42 93 90 00
Fax : 04 42 93 90 90

Projet systèmes d'exploitation

Enseignant responsable : Andreea DRAGUT

Eloi Perdereau
Axel Prat
Geoffrey Cano
Abdennour Boutrig

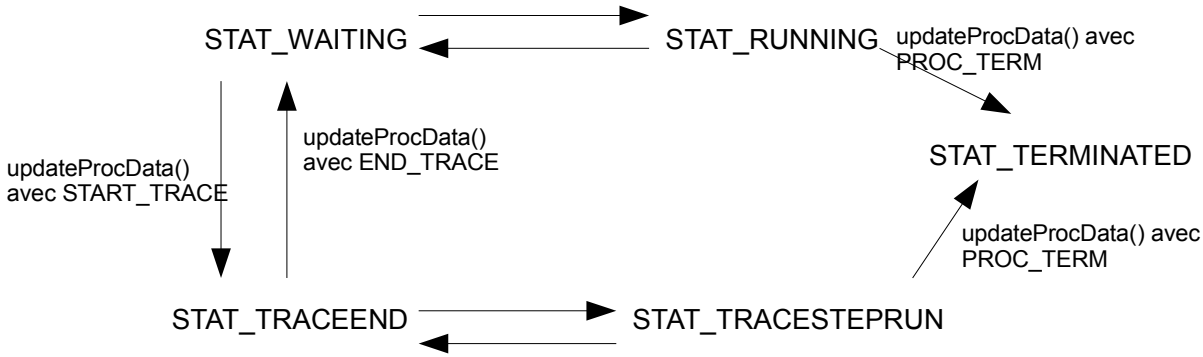
Mardi 10 janvier 2012

Sommaire

Explications sur le code initialement fourni

Dans proj.cxx, avant d'exécuter une instruction, l'ordonnanceur élit un processus avec electAProc(), et c'est la prochaine instruction de ce processus qui sera exécuté.

Diagramme des états initial :



STAT_TERMINATED = Processus terminé

STAT_WAITING = En attente que le l'ordonnanceur lui donne la main

STAT_RUNNING = En train d'exécuter une instruction du mini-langage

STAT_TRACEEND = En train d'être tracé

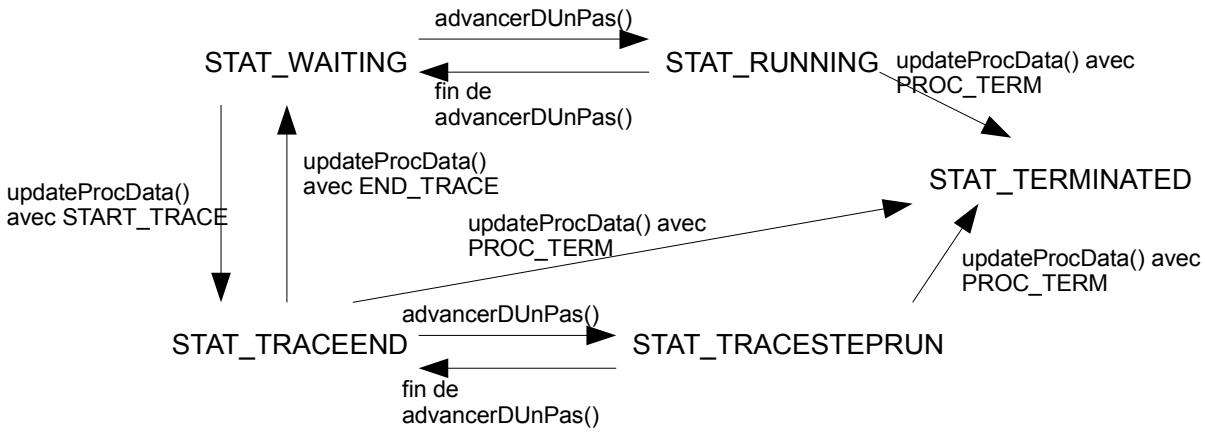
STAT_TRACESTEPRUN = Exécution d'une instruction du mini-langage en étant tracé

L'appel à updateProcData() avec comme type d'opération START_TRACE met le statut du processus à STAT_TRACE.
Il en est de même avec le type END_TRACE qui le met à STAT_WAITING.

L'appel à updateProcData() avec comme type d'opération PROC_TERM appel la fonction doTerminateProc() qui termine le processus et met son statut à STAT_TERMINATED.
Dans la version finale, un affichage a été rajouté dans doTerminateProc() si le processus est tracé.

Explications à propos du travail de groupe

Nouveau diagramme des états :



La fonction avancerDUnPas() est une sorte de wrapper de updateProcData() avec comme type d'opération ADVANCE_PROC, mais ici, non seulement on avance le processus d'une instruction, mais on bloque le signal SIGQUIT (et autres – voir l'optionnel sur les signaux). De plus, on modifie l'état du processus en STAT_RUNNING ou STAT_TRACESTEPRUN lors de l'exécution effective de l'instruction, selon si le processus est tracé ou non. (Le processus est tracé si, en rentrant dans avancerDUnPas(), son statut est STAT_TRACEEND, sinon, il ne l'est pas).

Par rapport au diagramme des états initial, on a rajouté le passage de l'état STAT_TRACEEND à STAT_TERMINATED directement. C'est uniquement le cas lorsqu'on exécute la commande « quit » du mini-débugger.

Explication du code rajouté :

Dans le fichier proj.cxx, après l'initialisation on déroute "tous" les signaux vers TraiterSig() (avec DerouterSignaux() de nsFctShell) et SIGQUIT vers la fonction LancerDbg(), puis on boucle en exécutant les instructions de ces processus avec la fonction avancerDUnPas() tant qu'il reste des processus non-terminés.
procInfo et newProc2Run sont désormais déclarés en global car on les utilise dans les traitants.

Pour le débugger, une classe a été crée : MiniDbg
Déclarée dans include/MiniDbg.h et définie dans dirproj/MiniDbg.cxx, cette classe représente (avec la fonction lancerDbg()) toute la partie débugger du mini-langage.
Elle possède 5 données membres :

- *m_ProcInfo* est la ProcInfo associée aux mini-processus
- *m_Proc* est l'indice du mini-processus courant dans procData de m_ProcInfo
- *m_Cmd* représente la commande tapée dans le prompt du mini-débugger
- *m_VarAAfficher* dans laquelle sont stockées les variables à afficher en mode pas à pas
- *m_Break* dans laquelle sont stockés les points d'interruptions (n° de ligne)
- *m_GoOut* sert pour sortir des imbrication de LancerDbg() (si plusieurs SIGQUIT)

Dans la fonction LancerDbg, on lance la trace pour le processus interrompu (newProc2Run), puis on créer le mini-débugger avant d'afficher son prompt.

La fonction Prompt() de MiniDbg s'occupe de vérifier la ligne tapée dans l'invite de commande, puis d'appeler les fonctions qui permettent de les exécuter.
Chaque commande du mini-débugger possède sa propre fonction Gerer* ().

La commande « continue » du mini-langage est exécutée dans la fonction GererContinue() qui avance pas à pas le programme tracé avec avancerDUnPas() tant que le processus n'est pas terminé.

Pour l'avancement pas à pas, on appel la fonction avancerDUnPas(), puis on affiche toutes les variables à afficher qui sont stockées dans le vecteur m_VarAAffichier.

GererPrint() appel AfficherVar() qui affiche nom de la variable entrée précédemment ainsi que sa valeur (ceci si elle existe dans le symbolTable, c'est à dire si la fonction findExistentSymbol() ne renvoie pas -1).
Pour le « display », avant d'ajouter la variable au vecteur m_VarAAfficher, on l'affiche avec AfficherVar(). Cet appel sert aussi de condition à l'ajout de la variable, ainsi, si une variable n'est pas trouvée, elle ne sera pas dans le vecteur.

La fonction GererBreak(), après maintes et maintes vérification ne fait qu'ajouter au vecteur m_Break, la ligne du fichier où il faudra interrompre le programme.
Cette interruption se fait dans la fonction GererContinue(), où avant d'exécuter une instruction, on vérifie si la prochaine instruction à exécuter n'est pas le prochain point d'interruption.

Concernant le redémarrage d'un processus, on charge dans l'instruction « père » du programme (proGram) la sauvegarde que l'on avait faite dans le constructeur de ProcInfo (proGramInit), puis on met 1 dans nextLineNumber (le début du programme, première ligne).
Enfin, on met le statut à STAT_TRACEEND car si on passe par cette fonction, c'est qu'on a eu un prompt pour ce processus qui est donc tracé.
Enfin on appelle GererStep() si step avait été précisé ou GererContinue () si ce n'est pas le cas.

Optionnel :
Notion de signaux au mini-langage

On a rajouté deux données membre à ProcData : sigMask et hanDler.

sigMask est un masque qui contient tous les signaux à bloquer pour l'exécution d'une instruction (initialisé avec SIGQUIT uniquement, mais peut être modifié dans le mini-langage par l'instruction SIGADD).
hanDler est de type ProcInstruction* qui aura le type DO_SIGNAL, et contiendra dans son bodyInstr les instructions du mini-traitant.

Lors du parsing du programme avec parseProg() qui est récursif, si l'instruction à parser commence par SIGNAL, on parse le traitant (toujours en récursif). Un peu comme pour PROGRAMME et WHILE ; lorsqu'on a fini de parser, on affecte l'instruction résultante à hanDler.

Dans la fonction doTheInstruction(), l'exécution de SIGADD et SIGDEL ne fait qu'ajouter, respectivement enlever le signal au masque sigMask.
On remarque que l'exécution d'une instruction DO_SIGNAL ne fait rien.

Lors de l'envoi d'un signal différent de SIGQUIT (et des signaux non déroutables), la fonction TraiterSig() est exécutée. Dans celle-ci, on exécute les instructions qui sont dans le bodyInstr de hanDler les unes après les autres.
(on ne gère pas les breaks, l'envoie d'un signal SIGQUIT termine le traitant).

Cahier des charges individuel

1. Eloi

Structure du débugger
- Déclaration de la classe MiniDbg
- Définition des fonctions traitantes dans proj.cxx

Points d'interruption
Au moins un optionnel

Rajouté :

Afficher les breaks, les processus ou les variables à afficher
Possibilité de supprimer des points d'interruptions ou des variables à afficher
Gérer plusieurs mini-processus

2. Axel

Terminaison, redémarrage, lancement pas-à-pas
Reprise de l'exécution d'un processus tracé

3. Abdennour

Exécution pas à pas

4. Geoffrey

Affichage du contenu des variables

Rajouté :

Modifier la valeur d'une variable