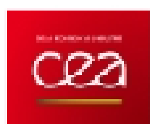
Introduction au cours de débogage

du séquentiel au HPC

Martial MANCIP

`Martial.Mancip_ala_MaisondelaSimulation_dot_fr`

Maison de la Simulation
CEA/Saclay, bât. 565, PC 190
91191 Gif-sur-Yvette CEDEX



Plan général du cours

Présenter les outils utilisables en séquentiel pour aller vers le parallèle et le HPC.

- introduction
- structure.pdf : compilation, symboles, mémoire
- system.pdf : les limites, les fichiers core, les nombres
- fichier.pdf : strace, les outils, le réseau
- gdb.pdf
- valgrind.pdf
- gdb_client-server.pdf

Plan

- programmation
- Les messages obscurs
- Les applications
- Le déboguage
- principe du débogage
- Les messages obscurs

site en anglais, assez complet.

programmation

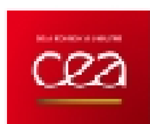
Comment ne pas faire de bogue ?

Au moins :

- concevoir,
- suivre des règles,
- gérer les versions,
- valider (tests unitaires, tests de non régression).

33^{ieme} ORAP, F. Bodin :

17 Règles de bonne pratiques de développement pour le HPC.
Bodin ORAP 33,
Article HPC Magazine : english version,version française.



Les applications

De plus en plus d'applications sont multi-langages.

(On peut déboguer une librairie qui tourne sous forme de script python.)

Utiliser les librairies et les interfaces automatiques ;
par exemple boost, scotch ou swig.

La parallélisation :

- MPI : passage de messages,
  difficulté sur plusieurs noeuds,
- OpenMP/threads : mémoire partagée,
  plus difficile à déboguer à cause de la désynchronisation des threads.

On découvre parfois des bogues avec l'augmentation du nombre de cœurs utilisés.

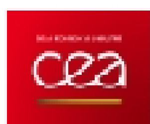
Le débogage

Ce que je dois faire lorsque mon code ne s'exécute pas comme attendu.

Lorsque

- Il ne démarre pas (ou très peu) : traces
- il ne calcule pas ce qui est attendu : versions, visu
- il ne s'arrête pas (il boucle à l'infini) : gdb, valgrind
- il s'arrête avec un message d'erreur obscur : lire le message, gdb
- il s'arrête et pas de messages,
  je ne sais pas où chercher !

Le binaire et les sources, les fichiers core, les outils...



principe du débogage

- Identifier rapidement la ligne du code source du programme où se produit le comportement suspect, sans recherche par sortie écran (printf, cout) ou fichier de sortie
- Contrôler le déroulement du programme en mode pas à pas, pratique pour arrêter le programme avant qu'il ne se plante
- contrôle de l'exécution d'un programme, arrêt conditionnel,
- Se concentrer sur les variables, en les explorant interactivement. supervision des variables, des fichiers ouverts

Les messages obscurs

- array out of bounds
- segmentation violation
- floatting point invalid
- stack overflow
- heap buffer overflow

On va étudier la structure des exécutables et la mémoire des processus.

La visualisation

Recherche d'un dysfonctionnement seulement après :

- versionnement : qu'est-ce qui a été modifié depuis la dernière version enregistrée ?
- tests unitaires : ma nouvelle fonction répond bien à ses spécifications ?
- valgrind ne rend pas d'erreur
- tests de simplification : un modèle conservatif conserve bien une solution constante ?

La visualisation

Utilisation des outils de visualisation « in-line ». Avec Catalyst (Paraview), libsim (Visit) ou DDT/Visit, par exemple.

La structure de compilation

La mémoire des programmes

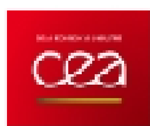
Martial MANCIP

Martial.Mancip_ala_MaisondelaSimulation_dot_fr

Maison de la Simulation

CEA/Saclay, bât. 565, PC 190

91191 Gif-sur-Yvette CEDEX



Plan

- Compilation
- Édition de liens
- Compilateurs
- Les librairies
- Table des symboles
- Mémoire virtuelle des processus
- traces

Compilation

du code au binaire

- pré-compiler ! -cpp, -E : des sources à conserver
- algorithmes ⇒ langages

Compilation

- librairies statiques et dynamiques,
  composants (couplages),
- paquets du système, chaîne de headers (/usr/include)
- Attention aux options d'optimisation !
  « -O3 » réduit les boucles
  mais l'enlever « supprime » parfois le bogue !

Édition de liens : ld

- regrouper (statique), relier (dynamique)
- vérifier les symboles : attention à l'ordre dans la liste !
- créer l'exécutable
- organiser la mémoire

Édition de liens : ld

cas du fortran 90 :
une partie de l'édition de lien est faite à la compilation !
fichiers de module = gestion des interfaces

encapsulation de ld avec le compilateur
(pour le f90 ou les mpif90/mpicc)

Compilateurs

fortran 90

- open source : gfortran g95 (modules en clair)
- propriétaires pgi, lf95 (très strict), ifort

C/C++ : ifc, gcc/g++

option bound-check : vérifie les tableaux

option trace => ptrace (suivi de processus pères/fils)

Les librairies

les librairies :

- .a (statique = uniquement pour cet exécutable)
- et .so (dynamique = partagées avec d'autres processus).

Attention : pour le débogage, on peut (doit) installer les versions des paquets « -dgb » des librairies.

ldd

outil 'file' : qu'est-ce qu'il y a dans ce binaire ?

Exemple sur mon `geeqie` :

ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=6905fe92bc180097534353b84003b742120cc299, stripped

ldd

```
linux-vdso.so.1 (0x00007fff31bb1000)
...
libgtk-x11-2.0.so.0 => /lib64/libgtk-x11-2.0.so.0 (0x00007f3d7aca1000)
libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f3d78f8f000)
libm.so.6 => /lib64/libm.so.6 (0x00007f3d78c89000)
...
libX11.so.6 => /lib64/libX11.so.6 (0x00007f3d77f56000)
libXcomposite.so.1 => /lib64/libXcomposite.so.1 (0x00007f3d76204000)
libGL.so.1 => /usr/lib64/nvidia-current/libGL.so.1 (0x00007f3d73ac6000)
...
libbz2.so.1 => /lib64/libbz2.so.1 (0x00007f3d72ed8000)
libXau.so.6 => /lib64/libXau.so.6 (0x00007f3d72cd4000)
libXdmcp.so.6 => /lib64/libXdmcp.so.6 (0x00007f3d72ace000)
libnvidia-glcore.so.346.72 =>
/usr/lib64/nvidia-current/libnvidia-glcore.so.346.72 (0x00007f3d6f966000)
libgraphite2.so.3 => /lib64/libgraphite2.so.3 (0x00007f3d6f744000)
```

Table des symboles

Les symboles : du code source au binaire
(parfois modifié par le compilateur)

```
> nm -a fichier_binaire
```

*-0.2cmCertains compilateurs rendent illisible le code dans la table

- mangle en C++

```
nm -a fich.o | xargs c++filt -_ > fich.demangle
```

  ou option « -C » de nm.

- exemple de ifort

  qui fait un mangle « maison » des symboles qui le rend plus difficile à déboguer sans 'idb'

Table des symboles

L'option '-g[1-3]' :
des informations pour le débogage en plus des symboles

La commande strip (ou l'option '-s') enlève la table des symboles (aussi -g0).

Mémoire virtuelle des processus

5 segments : text, data, bss, heap et stack.

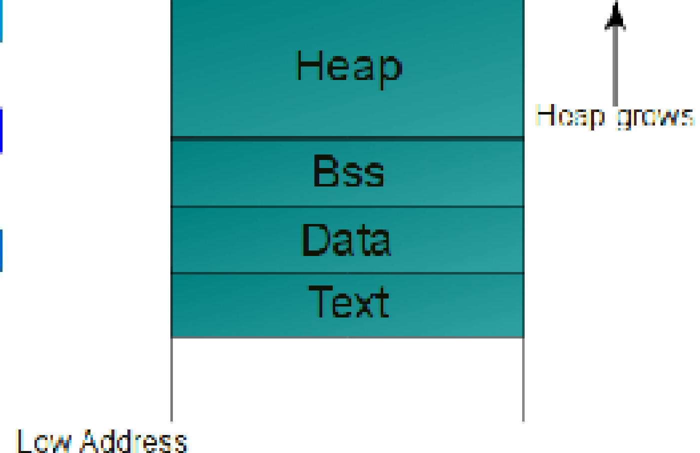
- segment text : segment code (source),
  en lecture seule,
  partagé pour les threads /
  processus pour les librairies en shared
  $\hookrightarrow$ ptrace (gdb), strace : pointeur d'instruction
- le segment data :
  variables statiques initialisées au lancement (save) et
  constantes globales du programme ;
  segment de taille constante en lecture seule,
  partagé pour les threads /
  processus pour les librairies en shared.

Mémoire virtuelle des processus

- Le segment bss « Block Started by Symbol » :
  variables statiques non initialisées,
  chaînes de caractères.
- Le segment tas (heap) : toutes les autres variables du programme (variables dynamiques).
  Le tas grandit vers les adresses mémoire plus grandes.

Mémoire virtuelle des p

- La pile (stack)

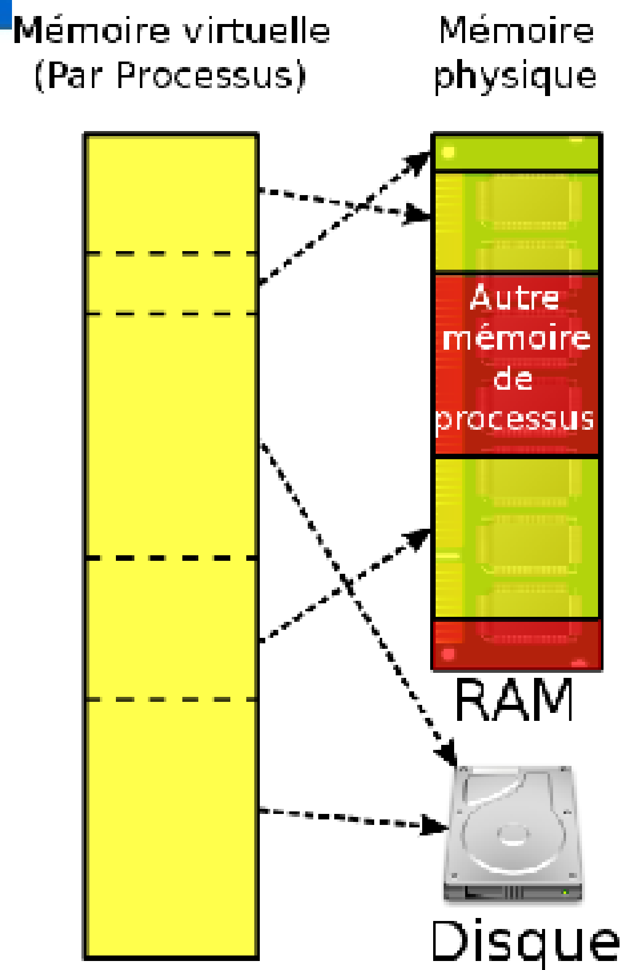


Mémoire virtuelle des processus

- La mémoire virtuelle

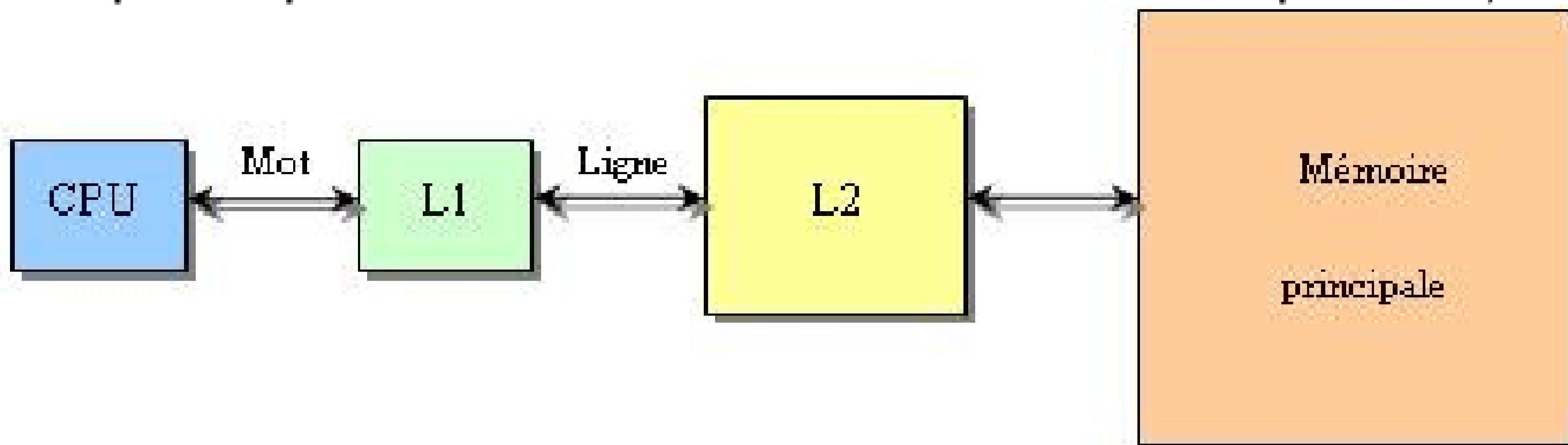
a changer tout ça !

source : wikipedia en



Mémoire virtuelle des processus

- la mémoire cache des processeurs : mémoire rapide (recopie de parties entières - boucle - du source si possible).



L’environnement d’exécution

Les limites systèmes

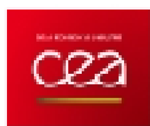
Martial MANCIP

Martial.Mancip_ala_MaisondelaSimulation_dot_fr

Maison de la Simulation

CEA/Saclay, bât. 565, PC 190

91191 Gif-sur-Yvette CEDEX



Plan

- Les limites systèmes
- Les plantages de programme
- Les fichiers core
- La précision numérique

Les limites systèmes

dépend du shell : ulimit ou limit

exemples :

```
limit vmemoryuse unlimited
ulimit -a
```

Les limites systèmes

```
coredumpsize   core file size          (blocks, -c)   0
datasize       data seg size           (kbytes, -d)   unlimited
filesize       file size               (blocks, -f)   unlimited
descriptors    open files                      (-n)   1024
memorylocked   max locked memory       (kbytes, -l)   64
memoryuse      max memory size         (kbytes, -m)   unlimited
stacksize      stack size              (kbytes, -s)   8192
cputime        cpu time               (seconds, -t)   unlimited
maxproc        max user processes              (-u)   63485
vmemoryuse     virtual memory          (kbytes, -v)   unlimited
```

plantages de programme

dépassement de pile (stack overflow)

- variables trop grandes
- boucle infinie sur un appel de fonction récursive

⇒ Ne jamais passer d'opération dans un appel de fonction !

dépassement de tampon (buffer overflow) :

- débordement vers un autre tableau
  (option de check-bounds)
- ou dans la mémoire d'un autre processus
  (interdit par le système :
  erreur de segmentation - « segmentation violation ») !

⇒ attention à strcpy ! ! (pas de vérification)

plantages de programme

dépassement de tas (heap buffer overflow) :
trop de mémoire allouée par rapport à la limite.

Les fichiers core

copie de la mémoire à l'instant du plantage :

coredumpsize à “unlimited”

core.$PID

Permet de récupérer

- le stack,
- la mémoire globale,
- la mémoire locale (parfois).

cf cours gdb.

Les fichiers core

```
> gdb test_core.e -c core.20735
[...]
Reading symbols from /lib64/libm.so.6...
Reading symbols from
/usr/lib/debug/lib64/libm-2.9.so.debug...done.
[...]
Core was generated by './test_core.e'.
Program terminated with signal 11, Segmentation fault.
(gdb) i s
#0 0x0000000000401911 in test_core_MP_mycall (myglobal=0x64b420)
                                    at test_core.f90 :8
#1 0x0000000000401955 in MAIN_ () at test_core.f90 :16
#2 0x0000000000402f4e in main ()
```

La précision numérique

Gestion des exceptions FPE (Floating Point Exceptions).

Les différentes erreurs sur les flottants :

- Division par 0 : produit un core ou un NaN.
- Défaut d'initialisation : propage les NaN (voir les options d'init).
- Floating Point Underflow (FPU) :
  on a divisé par un nombre trop grand
  ou lorsque l'on a multiplié deux nombres trop "éloignés".

.

La précision numérique

Des variables d'environnement ou des options de compilations sont disponibles.

Possible de détecter les NaN avec isNaN() en fortran90.

En général :

$x \sim= x$ `répondra « vrai » si x vaut NaN.`

La précision numérique

CADNA est un wrapper c++ sur C de recherche d'instabilité numérique.

- Self-validation detection
- Mathematical instabilities detection
- Branching instabilities detection
- Intrinsic instabilities detection
- Cancellation instabilities detection

utilisation gdb : « break instability »

Problèmes de fichier

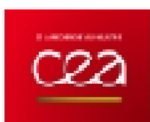
Martial MANCIP

Martial.Mancip_ala_MaisondelaSimulation_dot_fr

Maison de la Simulation

CEA/Saclay, bât. 565, PC 190

91191 Gif-sur-Yvette CEDEX



Plan

- tracer les appels systèmes

Plan

- tracer les appels systèmes
- fichiers ouverts d'un processus

Plan

- tracer les appels systèmes
- fichiers ouverts d'un processus
- Appel réseaux : lister les connections

tracer les appels systèmes

strace : trace les appels systèmes de fonctions et signaux
affiche le résultat dans la sortie d'erreur

$\equiv$ option « s » de **htop** récent

tracer les appels systèmes

exemple d'appel :

strace ls -la 1> out 2> err_$(date +%F_%H :%M)

2015-09-15_09 :06

```
prompt> strace ls intro.pdf
execve("/bin/ls", ["ls", "auto", "intro.aux", [...],"intro.tex "], [/* 127 vars */]) = 0
brk(0) = 0x186b000
[...]
open("/login/BIN/tls/x86_64/librt.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
stat("/login/BIN/tls/x86_64", 0x7fff656c1770) = -1 ENOENT (No such file or directory)
open("/lib/librt.so.1", O_RDONLY) = 3
read(3, "\177ELF\1[...]"..., 832) = 832
close(3) = 0
[...]
```

tracer les appels systèmes

> out_file_$(date +%F_%H :%M) 2>&1

C'est complexe, mais cela peut s'avérer très utile :

- pour vérifier que les fichiers sont bien trouvés et ouverts
- entre deux itérations d'un même code, avec l'outil `diff`

fichiers et processus

fichiers ouverts d'un processus : lsof -p #PID

> xterm less /PATH/fichier.tex&
[32605]
> /usr/sbin/lsof -p 32605

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

less 32605 mylogin cwd DIR 8,6 4096 4727131 /PATH/

[...]

less 32605 mylogin 4r REG 8,6 1088 2580499 /PATH//fichier.tex

htop : le super-top ! (lsof : touche 'l')

fichiers et processus

Quel utilisateur/processus utilise ce fichier ?
/sbin/fuser -v fichier/device

```
> /sbin/fuser -v system.pdf
               USER      PID     ACCESS   COMMAND
 system.pdf :  mylogin   10577   F....    acroread
```

cnrs cea

Appel réseaux : lister les connections

lister les sockets ouvertes :

```
netstat -latup
```

exemple :

> netstat -latup | grep 3892

(Tous les processus ne peuvent être identifiés, les infos sur les processus non possédés ne seront pas affichées, vous devez être root pour les voir toutes.)

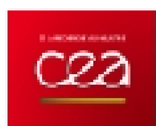
tcp 0 0 gagarin.local :53317 lhr08s04-in-f22.1e100 :https ESTABLISHED 3892/chrome

tcp 0 0 gagarin.local :46322 lhr14s19-in-f0.1e100. :https ESTABLISHED 3892/chrome

tcp 0 0 gagarin.local :40379 lhr14s22-in-f21.1e100 :https ESTABLISHED 3892/chrome

tcp 0 0 gagarin.local :50205 stackoverflow.com :http ESTABLISHED 3892/chrome

tcp 0 0 gagarin.local :56180 wi-in-f147.1e100.net :https ESTABLISHED 3892/chrome



Appel réseaux : lister les connections

socket avec port :

```
netstat -latupen | grep :#PORT
netstat -na | grep 11111
lsof -i :11111
```

L’outil gdb

le débogueur de base

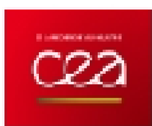
Martial MANCIP

`Martial.Mancip_ala_MaisondelaSimulation_dot_fr`

Maison de la Simulation

CEA/Saclay, bât. 565, PC 190

91191 Gif-sur-Yvette CEDEX



Plan

- bibliothèques et symboles
- éditeurs graphiques
- démarrer gdb
- sources de doc
- actions pour déboguer
- arrêter les processus
- analyser l'état de la mémoire
- modifier l'exécution
- se situer
- avancer dans l'exécution
- appeler une fonction, modifier

Plan

Utilisation avancée

- mode Text User Interface
- recommencer avec les scripts gdb
- points d'arrêt conditionnels
- Watchpoints
- commandes sur les breakpoints
- attacher un processus

bibliothèques et symboles

- symboles de débogage “-g”
- optimisation de la zone *text* avec “strip”

⇒ fonctions systèmes :
paquets de symboles à rajouter
(exemple : libc6-dbg)

⇒ récupérer les symboles de vos bibliothèques :

objcopy –only-keep-debug monexeavecsymboles messymboles.dbg

éditeurs

Plus simple les GUI de débogage ?
⇒ suivi des fichiers/lignes de code

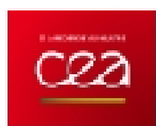
Tout environnement de développement (eclipse,...) ? !

open-source : kgdb, ddd, ...

commercial : Alinéa ddt, RogueWave TotalView

emacs gud-mode : M-x gdb -> “gdb -f monprog”

mode terminal avancé : section gdb-TUI plus loin



démarrer

```
> gdb programme
```

Attacher un fichier core

```
> gdb programme -c corefile
```

Attacher un processus

```
> gdb programme nPID
```

Ajouter des arguments à passer au programme :

gdb –args executable ses_arguments

ou en interactif

(gdb) set args MYARGS

(gdb) r[un]

déboguer

arrêter, observer, analyser

modifier

avancer, revenir en arrière ?

en parallèle : attraper un process

en openMP : changer de thread

À l'aide !

`> man gdb`

`> info gdb` ('C-h i' sous emacs)

(gdb) help

(gdb) help info
(gdb) help info stack
(gdb) h i s

arrêter

Attention aux symboles (commande `nm`) !

(gdb) break main

en fortran 90, on a souvent :

(gdb) break MAIN_

Dans un même fichier

(gdb) break ligne

(gdb) break fonction

Une fonction d'un module gfortran :

(gdb) break module_MP_fonction

breakpoints

On peut utiliser Control-z pour s'arrêter,
mais le breakpoint est plus précis !

```
(gdb) break 28
Breakpoint 1 at 0x401b29 : file test_infinity1.F90, line 28.
(gdb) b 43
Breakpoint 2 at 0x401b48 : file test_infinity1.F90, line 43.
```

breakpoints

(gdb) i[nfo] b[reakpoints]

```
Num Type Disp Enb Address What
1 breakpoint keep y 0x0000000000401b29 in inf_MP_infinity
                                        at test_infinity1.F90 :28
2 breakpoint keep y 0x0000000000401b48 in MAIN_
                                        at test_infinity1.F90 :43
breakpoint already hit 1 time
```

analyser

- Voir les 4 prochaines lignes du code :

(gdb) l[ist] 4

```
10 INTEGER : : i,j
11 REAL : : r
12
13 WRITE(*,*) "Appel de la fonction infinity avec la taille de boucle :",loopsize
```

- Voir une variable

(gdb) p[rint] value

- Son adresse en mémoire

(gdb) p *value

On peut utiliser les "casts" du C :

(gdb) p *(int *)value

.

analyser

- Un élément d'une structure

  (gdb) print donnees_tot_.part_g_

- Type d'une variable/structure :

  (gdb) p varp

  $12 = (const NC_var *) 0x90c6f28

  (gdb) pt varp

  ```
  type = struct {
          size_t xsz ;
          size_t *shape ;
      } *
  ```

- examiner une valeur pointée

  (gdb) x/d &static_b

  0x600ce4 <static_b> : 2

  (gdb) x/d 0x600ce8

  0x600ce8 <static_f_c.2549> : 3

analyser

La liste des variables globales

(gdb) info var[iables]

La liste des variables locales

(gdb) i[nfo] loc[als]

analyser

La liste des fonctions

(gdb) i[nfo] fun[ctions]

```
(gdb) i functions sin.*
All functions matching regular expression "sin.*" :
File rtld.c :
static void print_missing_version(int, const char *, const char *) ;

Non-debugging symbols :
0x00002b8f3936eea0 asinh
0x00002b8f39375710 sin
```

Appel de fonction :

```
(gdb) call sin(3.14159)
$1 = 2
```

analyser

table des symboles

```
(gdb) maint info symtab
{ objfile /home/login/[..]/ORCHIDEE_HEAD/modipsl/bin/orchidee_ol
((struct objfile *) 0x82b1cc8)

{ symtab stomate_data.f90 ((struct symtab *) 0x8ea4198)
dirname
/home/login/[..]/ORCHIDEE_HEAD/[..]/ORCHIDEE/src_stomate
[...]}

{ symtab readdim2.f90 ((struct symtab *) 0x8e94d68)
dirname /home/login/[..]/ORCHIDEE_HEAD/[..]/ORCHIDEE_OL
[...]}
```

se situer

"stack frame"

(gdb) info stack ("i s") ≡ backtrace ("bt")

```
(gdb) i s
#0 0x00002b921eb993f0 in write () from /lib64/libc.so.6
#1 0x00000000004178c9 in _g95_flush_stream ()
#2 0x0000000000402177 in write_formatted_sequential ()
#3 0x0000000000402316 in write_record ()
#4 0x000000000040250d in _g95_st_write_done ()
#5 0x0000000000401b1c in inf_MP_infinity
(loopsize=0x7fff8c46de88)
at test_infinity1.F90 :25
#6 0x0000000000401b68 in MAIN_ () at test_infinity1.F90 :49
#7 0x000000000040a9be in main ()
```

se situer

```
(gdb) up
#1 0x00000000004178c9 in _g95_flush_stream ()
(gdb) down
#0 0x00002b921eb993f0 in write () from /lib64/libc.so.6
```

se situer

```
(gdb) i s
#0 0x0851c1f3 in ncx_get_float_double
        (xp=0x9031308, ip=0x10dc7000) at ncx.c :1192
#1 0x0851eb53 in ncx_getn_float_double
        (xpp=0x52bf6024, nelems=1761, tp=0x10dc7000)
            at ncx.c :3457
#2 0x08525ef5 in getNCvx_float_double
        (ncp=0x90c6b60, varp=0x90c6f28, start=0x52bf60a0,
         nelems=842400, value=0x10dc6708)
            at putget.c :3645
```

avancer

La touche ”entrée” rappelle la dernière commande.

- cont : continuer
- pas-à-pas sommaire :
  step : avancer d’une ligne de code
  next : avancer d’une fonction
- pas-à-pas détaillé
  stepi : avancer d’une instruction
  nexti : avancer d’une instruction ou d’une fonction
- retour en arrière ? Totalview

modifier

modification d'une variable

```
(gdb) print loopsize
$5 = (int4 * const) 0x7fff28b89598
(gdb) p *loopsize
$6 = 10000000
(gdb) p *loopsize=4
$7 = 4
(gdb) p *loopsize
$8 = 4
```

Vecteur :

```
(gdb) p *loopsize@4
$9 = {4,10000000,10000000,10000000}
```

modifier

(gdb) info fun
All defined functions :
[...]
File memory_segment.c :
void f(void) ;
(gdb) r
Starting program : /home/login/PROG/memory_segment-g

Breakpoint 1, main (argc=1, argv=0x7fffffffe088) at
memory_segment.c :48
48 int local_main_a = 10 ;

Appel de la fonction f :
(gdb) call f()

gdb-TUI

gdb dans un terminal.
Passage en mode Text User Interface / mode standard :

- ‘C-x C-a’
- ‘C-x a’
- ‘C-x A’

On utilise alors les flêches.
Mode avec suivi des lignes des sources
avec une (‘C-x 1’) ou deux (‘C-x 2’) fenêtres.

Pour changer de fenêtre, faire ‘C-x o’.

Pour une utilisation simplifier de gdb, utilisez le mode
SingleKey ‘C-x s’.

recommencer avec les scripts gdb

Toutes ces étapes doivent parfois être recommencées plusieurs fois si l'on a dépassé le problème.

On doit alors retaper beaucoup d'appels de commandes gdb.

Heureusement, on peut enregistrer ces commandes avec la commande « script file_out » !

Ex de fichier de commandes :

```
info stack
up 7
info locals
p im
```

recommencer avec les scripts gdb

En ligne de commande :

```
gdb -x fichier_script mon_exe
```

En interactif :

(gdb) source commandes_gdb

points d’arrêt conditionnels

```
(gdb) condition 4 j==51500
(gdb) i b
Num Type Disp Enb Address What
2 breakpoint keep y 0x0000000000401b48 in MAIN_
                                        at test_infinity1.F90 :43
             breakpoint already hit 1 time
3 breakpoint keep y 0x0000000000401b58 in MAIN_
                                        at test_infinity1.F90 :48
             breakpoint already hit 1 time
4 breakpoint keep y 0x0000000000401b1c in inf_MP_infinity
                                        at test_infinity1.F90 :26
             stop only if j == 51500
             breakpoint already hit 1 time
```

points d'arrêt conditionnels

```
(gdb) c
Continuing.
Valeur de j : 51497
Valeur de j : 51498
Valeur de j : 51499
Valeur de j : 51500
Breakpoint 4, inf_MP_infinity (loopsize=0x7fff28b89598)
                              at test_infinity1.F90 :26
                             26 j=j+1
```

points d’arrêt conditionnels

```
(gdb) i b
Num Type Disp Enb Address What
[...]
4 breakpoint keep y 0x0000000000401b1c in inf_MP_infinity
                                    at test_infinity1.F90 :26
            stop only if j == 51500
            breakpoint already hit 2 times
```

```
(gdb) condition 4 j>51600
```

```
(gdb) condition 4
Breakpoint 4 now unconditional.
```

Watchpoints

```
(gdb) watch i
Hardware watchpoint 5 : i
(gdb) i b
Num Type Disp Enb Address What
[...]
4 breakpoint keep y 0x0000000000401b1c in inf_MP_infinity
                                        at test_infinity1.F90 :26
              breakpoint already hit 11 times

5 hw watchpoint keep y i
```

Watchpoints

```
(gdb) c
Continuing.

Hardware watchpoint 5 : i
Old value = 0
New value = 1

inf_MP_infinity (loopsize=0x7fff28b89598) at test_infinity1.F90 :21
21 DO WHILE ( i < loopsize )
```

Il s'arrête dès que la variable i a changé ;
à la ligne suivant la modification : après
20 i = 1

commandes sur les breakpoints

La structure **commands ... end** permet de programmer le dernier break(watch)point enregistré.

- Test **if-else-end**
- Condition == != < >
- Impression **p[rint] printf**
- Changement de variable **set var variable=valeur**
- continuation **cont**
- tout autre commande de gdb !

commandes sur les breakpoints

```
(gdb) break toto.c :38
(gdb) commands
(gdb)     silent
(gdb)     set x = y + 4
(gdb)     printf "x is %d\n",x
(gdb)     cont
(gdb) end
```

attacher un processus

- top ou **htop** ?
- ps -Aef | grep monprog
- ps ax -o euser,pid,s,%cpu,%mem,args | grep -v grep | grep “theprocess”
- Attacher :

```
> gdb monprog PID
```

Voir aussi l’option gdb *follow-fork-mode* en cas de clonage d’un processus.

Machine virtuelle valgrind

Memcheck

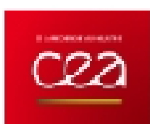
Martial MANCIP

`Martial.Mancip_ala_MaisondelaSimulation_dot_fr`

Maison de la Simulation

CEA/Saclay, bât. 565, PC 190

91191 Gif-sur-Yvette CEDEX



Plan

- Les outils
- lancement
- options
- rapport final
- lancer le débogueur
- suppressions

Les outils

L'outil par défaut est « Memcheck » qui analyse la gestion de la mémoire :
absence d'initialisation, erreur de pointeurs
allocation dynamique non libérée (trous = leak error),
débordement de tableaux (dynamique et non statique)

Il y a plusieurs autres outils :

- cachegrind : analyse du cache
- callgrind : graphe d'appel à l'exécution, profiling
- helgrind : débogueur de threads.
- Massif : optimiseur de mémoire

voir le site valgrind.
Et le tutoriel de openclassrooms.

lancement

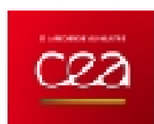
```
> valgrind monprogramme.e
```

Une erreur : ==#PID== du process, description, puis stack

```
==3227== Invalid read of size 4
==3227==    at 0x400658 : g (in PROGRAMMES/C/memory_segment.e)
==3227==    by 0x4005E9 : f (in PROGRAMMES/C/memory_segment.e)
==3227==    by 0x4006FD : main (in PROGRAMMES/C/memory_segment.e)

==3227==  Address 0x51b9040 is 0 bytes inside a block of size 4 free'd
==3227==    at 0x4C25A9E : free (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==3227==    by 0x400653 : g (in PROGRAMMES/C/memory_segment.e)
==3227==    by 0x4005E9 : f (in PROGRAMMES/C/memory_segment.e)
==3227==    by 0x4006FD : main (in PROGRAMMES/C/memory_segment.e)
```

Si la table des symboles est présentes, on aura aussi les lignes de code.



options

Quelques options bien utiles.

- taille de la pile affichée : « –num-callers=10 »
- les trous de mémoire :

  « –leak-check=yes » « –leak-resolution=med|high »

  « –show-reachable=yes »
- fixe un nombre maximum d'erreurs : « –error-limit=yes »
- les sous-processus : « –trace-children=yes »
- les fichiers ouverts : « –track-fds=yes »
- les non initialisations : « –undef-value-errors=yes »

rapport final

```
==2119== FILE DESCRIPTORS : 11 open at exit.
==2119== Open file descriptor 10 : stomate_forcing.nc
==2119== at 0x1BA08E65 : open64 (in /lib/tls/libc-2.3.3.so)
==2119== by 0x84E146E : netcdf_MP_nf90_create_ (netcdf_file.f90 :44)
==2119== by 0x80FC651 : sechiba_MP_sechiba_main_ (sechiba.f90 :267)
==2119==
==2119== Open file descriptor 2 : /dev/pts/2
==2119== <inherited from parent>
==2119==
==2119== Open file descriptor 1 : /dev/pts/2
==2119== <inherited from parent>
==2119==
==2119== Open file descriptor 0 : /dev/pts/2
==2119== <inherited from parent>
```

rapport final

Résumé des erreurs

```
==2119== ERROR SUMMARY : 617 errors from 3 contexts (suppressed : 25191 from 17)
==2119==
==2119== 1 errors in context 1 of 3 :
==2119== Use of uninitialised value of size 8
==2119== at 0x806CF15 : _g95_maxval1_r8 (in /home/login/[..]/bin/orchidee_ol)
==2119== by 0x80F0F12 : intersurf_MP_intersurf_main_2d_ (intersurf.f90 :291)
==2119== by 0x805FF85 : MAIN_ (dim2_driver.f90 :791)
==2119==
==2119== 308 errors in context 2 of 3 :
==2119== Use of uninitialised value of size 8
```

rapport final

Suppressions : “erreurs” déjà référencées que l’on ne veut plus détailler.

–2119– supp : 19 Ugly strchr error in /lib/ld-2.3.3.so

–2119– supp : 60 U8_kill_trailing

–2119– supp : 60 C8_kill_trailing

–2119– supp : 90 U7_kill_trailing

–2119– supp : 3819 C7_kill_trailing

–2119– supp : 31 U6_kill_trailing

–2119– supp : 6438 C6_kill_trailing

–2119– supp : 6810 C6_strlen

rapport final

Mémoire finale

```
==2119== IN SUMMARY : 617 errors from 3 contexts (suppressed : 25191 from 17)
==2119==
==2119== malloc/free : in use at exit : 0 bytes in 0 blocks.
==2119== malloc/free : 0 allocs, 0 frees, 0 bytes allocated.
```

lancer le débogueur

Ajout de l'option « –db-attach=yes »

==3706== —- Attach to debugger ? — [Return/N/n/Y/y/C/c] —- y

==3706== starting debugger with cmd :

/usr/bin/gdb -nw /proc/3719/fd/1024 3719

[...]

0x0000000000400658 in g ()

(gdb) i s

#0 0x0000000000400658 in g ()

#1 0x00000000004005ea in f ()

#2 0x00000000004006fe in main ()

On peut changer le débogueur par défaut :

« –db-command=<command> »

lancer le débogueur

Pour quitter gdb

```
(gdb) q
A debugging session is active.
        Inferior 1 [process 3719] will be detached.
Quit anyway ? (y or n) y
Detaching from program : /proc/3719/fd/1024, process 3719
==3706==
==3706== Debugger has detached.
                    Valgrind regains control. We continue.
Variable local_main_a apres f : 10
```

suppressions

Gestion des suppressions automatiques

« –gen-suppressions=yes »

« –suppressions=fichier_de_suppressions »

exemple de suppression « Addr4 »

(utilisation d'un pointeur désalloué)

```
{   bug_g_f
     Memcheck :Addr4
     fun :g
     fun :f
     fun :main
}
```

« bug_g_f » est le nom que l'on donne à celle-ci.

suppressions

On peut construire un fichier de suppressions initial :

« –gen-suppressions=all »

« –log-file=logfile.log »

Alors on « récupère » les suppressions avec le script suivant

```
cat ./logfile.log | gawk -f ./parse_valgrind_suppressions.awk > suppfile
```

Débogue parallèle

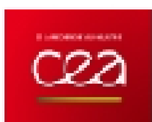
Martial MANCIP

`Martial.Mancip_ala_MaisondelaSimulation_dot_fr`

Maison de la Simulation

CEA/Saclay, bât. 565, PC 190

91191 Gif-sur-Yvette CEDEX



Plan

- gérer les threads
- gdb server
- exemple de bogues
- valgrind parallèle

gérer les threads

Les codes OpenMP sont basés sur des threads. On doit donc déboguer un processus et ses threads.

- **thread NumThread** : changer de thread
- **info threads** : la liste des threads d'un processus
- **thread apply [Num(s)Thread | all] args** : appliquer une commande à un thread, à un sous-ensemble (24-42) ou à tous
- breakpoints spécifique à un thread :

```
(gdb) break frik.c :13 thread 28 if bartab > lim
```

GDB server

- lancement de gdbserver sur le noeud de calcul

```
> gdbserver host :2345 hello_world
```

- Côté hôte gdb se lance avec le même exécutable de la machine cible

```
> gdb hello_world
```

- puis nécessite de se connecter à distance dans gdb

(gdb) target remote the-target :2345

exemple de bogues

1. bogue de barrière :
   send/receive $\Rightarrow$ lancer gdb sur process
2. envoie asynchrone :
   isend + modification du buffer juste après l'écriture de l'envoie
3. erreur “numérique” :
   bogue de mise à jour (sur root_prc) sans broadcast
4. même bogue qu'en séquentiel

valgrind en parallèle

parfois, il faut “attaquer” le démon MPI :

```
$ valgrind mpd &
$ /usr/local/bin/mpirun -np 3 mon_exe
```

```
valgrind mpirun -np 2 monserveur -np 3 monclient
```

```
–6949– errormgr : 3 supplist searches, 26 comparisons during search
–6949– errormgr : 3 errlist searches, 3 comparisons during search
==6948==
==6948== Process terminating with default action of signal 2 (SIGINT)
```