Debug && Cheatsheets

M1 - CHPS

Architecture Interne des Systèmes d'exploitations (AISE)

Jean-Baptiste Besnard
<jean-baptiste.besnard@paratools.com>

Julien Adam
<julien.adam@paratools.com>

Organisation

- Chaque session est découpée en deux parties. Un cours théorique le matin et une mise en pratique l'après-midi (TD) portant sur les connaissances vues le matin.
- Des QCMs sur les bases **importantes** au fil des semaines et portant sur un cours précédent. Le QCM aura toujours lieu durant la matinée
- Un Projet, date de rendu au **19/12/2023 23:59**
- Un Examen final le **22/12/2023 (APM)**

1. **Généralités sur les OS et Entrées-Sortie**
2. **Compilation, Bibliothèques et Layout Mémoire**
3. **Mémoire partie 2, Layout Binaire, Runtime**
4. **Virtualisation et Conteneurs**
5. **Programmation réseau et entrées/sorties avancées**
6. **Programmation Noyau**
7. **Scheduling et Temps-Réel**
8. **Examen Ecrit et Démos**

Type d'Examen	Coefficient
QCMs	10 %
Projet	40 %
EXAMEN	50 %

Cours et Corrections

https://github.com/besnardjb/AISE_24

Programme

1. ~~*Généralités sur les OS et Entrées-Sortie*~~
2. ~~*Compilation, Bibliothèques et Layout Mémoire*~~
3. ~~*Mémoire, Layout Binaire, Runtime*~~
4. ~~*Virtualisation et Conteneurs*~~
5. ~~*Programmation réseau et entrées/sorties avancées*~~
6. ~~*Programmation Noyau*~~
7. **Debug & Cheatsheets**
8. Examen Ecrit et Démos

Couteau suisse sous Linux

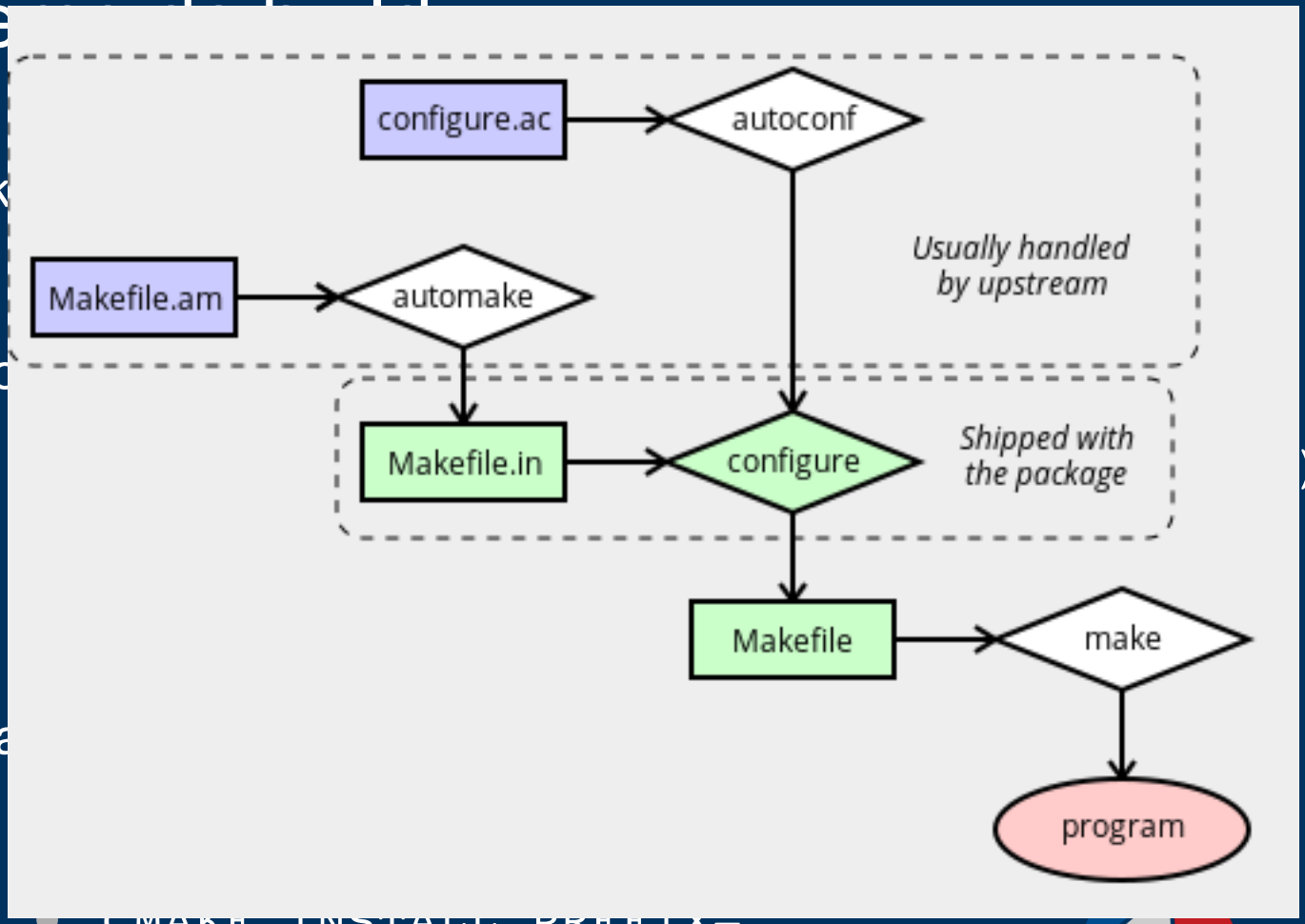
- Bases linux
  - cd, ls, mkdir…
  - cat, less, more, tail…
  - Shell, bash, zsh…
  - find, grep, sed…
- Gestion processus
  - top, htop
  - ps, kill, signal…
- Multiplexers (tmux, screen)
- Debuggers (au moins gdb)
- Réseau
  - ip, route, netstat
  - host, dnslookup
- Compilation & binaire
  - gcc, icc, xlc, clang…
  - readelf, nm
  - objdump, hexdump, xxd
- Gestion logicielle
  - Git, *svn*
  - Forges Gitlab, GitHub
  - Vim, emacs, nano…

Système de build

- Makefiles
  - `make`
- Autotools
  - `./autogen.sh && ./configure (puis Makefile)`
    - **`–prefix=`**
    - **`CC=`**
- CMake
  - `cmake .. (puis makefile)`
    - `CMAKE_INSTALL_PREFIX=`
    - `CMAKE_C_COMPILER=`

Systè

- Mak
  -
- Auto
  -
- CMa
  -
    - CMAKE_INSTALL_PREFIX=
    - CMAKE_C_COMPILER=



Système de build

- Makefiles
  - `make`
- Autotools
  - `./autogen.s`
    - **`–prefix=`**
    - **`CC=`**
- CMake
  - `cmake .. (p`
    - `CMAKE_IN`
    - `CMAKE_C_`

```
1   #                                               -*- Autoconf -*-
2   # Process this file with autoconf to produce a configure script.
3
4   AC_PREREQ([2.71])
5   AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])
6   AC_CONFIG_SRCDIR([main.c])
7   AC_CONFIG_HEADERS([config.h])
8
9   # Checks for programs.
10  AC_PROG_CC
11
12  # Checks for libraries.
13  # FIXME: Replace 'main' with a function in '-lcustom':
14  AC_CHECK_LIB([custom], [main])
15
16  # Checks for header files.
17
18  # Checks for typedefs, structures, and compiler characteristics.
19  AC_TYPE_SIZE_T
20
21  # Checks for library functions.
22  AC_CHECK_FUNCS([strdup])
23
24  AC_CONFIG_FILES([Makefile])
25  AC_OUTPUT
26
27
```

Environment-modules / lmod

- Facile de rajouter ses propres modules:
  - Exporter MODULEPATH (fonctionne comme PATH)
  - L'arborescence d'un module est analogue à chemin
  - La recette détaille la liste des modifications d'environnement nécessaire pour charger le module

```
adamj@saturn:~$ tree /usr/local/packages/Modules/modulefiles/openmpi*
/usr/local/packages/Modules/modulefiles/openmpi
├── 4.0.3-gcc10.2
├── 4.0.3-gcc8.3
├── 4.0.3-gcc9.3
├── 4.0.3-llvm10
├── 4.0.3-llvm9
└── 4.0.6-gcc10.2

0 directories, 6 files
```

Er

- t

```
adamj@saturn:~$ cat /usr/local/packages/Modules/modulefiles/openmpi/4.0.3-gcc10.2
#%Module#####################################
##
## openmpi
##

set _module_name  [module-info name]

module-whatis   "OpenMPI Version 4.0.3 built using GCC 10.2 compilers (64-bit) "

set _module_name  [module-info name]
set is_module_rm  [module-info mode remove]
set sys           [uname sysname]
set os            [uname release]
set mach          [uname machine]

conflict mpi-tor

if {[is-loaded mpi] && !$is_module_rm } { module unload mpi }

set     SWTOP   /packages
set     MPI     $SWTOP/openmpi/4.0.3-gcc10.2
setenv  MPI         $MPI
setenv  TAU_MPI "-mpiinc=$MPI/include -mpilib=$MPI/lib"

prepend-path    PATH    $MPI/bin
prepend-path    LD_LIBRARY_PATH $MPI/lib
#prepend-path    LD_LIBRARY_PATH /opt/torque/lib
prepend-path    MANPATH $MPI/share/man
prepend-path    C_INCLUDE_PATH $MPI/include
prepend-path    CPLUS_INCLUDE_PATH $MPI/include

module load gcc/10.2.0

#system /packages/Modules/track.sh $_module_name
```

Spack

```
Naos  ~
 spack list | wc -w
3618
```

- Conçu pour les problématiques HPC
- Gestionnaire de paquets en userspace
- Supporte de nombreuses recettes HPC:
  - Plusieurs souches MPI
  - hwloc
- Prend en charge les variantes compil' et les flags
- Simple à installer:
  - Télécharger les sources: https://github.com/spack/spack
  - `export PATH=$PWD/spack/bin:$PATH`
  - Intégration: `source ./spack/share/spack/setup-env.sh`

Spac

- Co
- Ge
- Su
  -
  -
- Pre

```
└─ spack
usage: spack [-hkV] [--color {always,never,auto}] COMMAND ...

A flexible package manager that supports multiple versions,
configurations, platforms, and compilers.

These are common spack commands:

query packages:
  list                  list and search available packages
  info                  get detailed information on a particular package
  find                  list and search installed packages

build packages:
  install               build and install packages
  uninstall             remove installed packages
  spec                  show what would be installed, given a spec
```

```
└─ spack find
==> 85 installed packages
-- linux-linuxmint20-haswell / gcc@9.3.0 ------------------------
autoconf@2.69         czmq@4.1.1          gmp@6.2.1           libmd@1.0.3         m4@1.4.19
automake@1.15         diffutils@3.7       hwloc@2.5.0         libpciaccess@0.16   mpc@1.1.0
automake@1.16.3       diffutils@3.8       hwloc@2.6.0         libsigsegv@2.13     mpfr@3.1.6
automake@1.16.5       docbook-xml@4.5     hwloc@2.6.0         libsodium@1.0.18    ncurses@6.2
berkeley-db@18.1.40   docbook-xsl@1.79.2  jansson@2.13.1      libtool@2.4.6       numactl@2.0.14
binutils@2.37         expat@2.4.1         libbsd@0.11.3       libxml2@2.9.12      openmpi@4.1.1
boost@1.77.0          flux-core@0.31.0    libedit@3.1-20210216 libyaml@0.2.5      openssh@8.7p1
bzip2@1.0.8           flux-sched@0.20.0   libelf@0.8.13       libzmq@4.3.4        openssl@1.1.1l
cmake@3.21.2          gcc@9.3.0           libevent@2.1.12     lua@5.3.5           openssl@1.1.1l
cmake@3.21.4          gdbm@1.19           libffi@3.3          lua-luaposix@35.0   osu-micro-benchmarks@5.7.1
cuda@11.4.0           gettext@0.21        libiconv@1.16       lz4@1.9.3           pcre@8.44
```

Python & pyenv

- Python est maintenant prédominant en environnement HPC
  - Codage simple pour des non-experts
  - Large panel de fonctionnalités
  - Gestionnaire de paquets inclus
  - Langage orienté objet
  - Python 2 déprécié, focus sur python3 !
- Installer un paquet : pip
  - `pip install mpi4py (via pypi.org)`
  - `pip install .`

Pyenv

- Toujours essayer de développer dans des « environnements »
- Un environnement crée une installation virtuelle d'une souche Python
- La gestion de l'environnement Python peut être cloisonné
  - Pip install virtualenv
- Créer un env: **python3 -m virtualenv ./build**
- Charger un env: **source ./build/bin/activate**
- Décharger: **deactivate**
- Supprimer un env: **rm -rf ./build/**
- Comment gérer plusieurs installations Python par dessus !
  - **Pyenv !**

The Art of Debugging

Bug d’un programme

- Qu’est-ce qu’un « bug » dans un programme?

Bug d’un programme

- Qu’est-ce qu’un programme qui « bug » ?
  - Crash (SEGV par exemple)
  - Résultat différent de ce qui est attendu
  - Interblocage (deadlock)
- Idée : Suivre l’exécution du programme (flot & variables)
- L’outil du débutant en debug : printf
  - Avantages : Simple, aucune connaissance à priori
  - Inconvénients : recompilation, scories…

Le « vrai » debugging

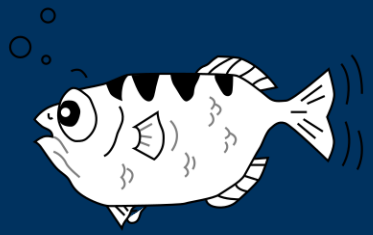
- Contrôlé par un outil tiers : le débugger
- Large panel de fonctionnalités :
  - Suivre une variable
  - Mettre en pause le programme
  - Insérer des « points d’arrêt » (breakpoint)
  - Exécution du programme instruction par instruction
  - Explorer le binaire
  - Explorer la mémoire
  - …

Exemple d’un debugger

```
└─▶ gcc segv.c -g && ./a.out
[1]    28842 segmentation fault (core dumped)  ./a.out
```

```
└─▶ gcc segv.c -g && gdb ./a.out
Reading symbols from ./a.out...done.
gdb >> run
Starting program: /home/adamj/Documents/cours/aise/Cours_5/debug/a.out
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.27-37.fc28.x86_64

Program received signal SIGSEGV, Segmentation fault.
----------------------------------------------------------------------------------------------------------------------------------[regs]
  RAX: 0x0000000000000000  RBX: 0x0000000000000000  RBP: 0x00007FFFFFFFD990  RSP: 0x00007FFFFFFFD978  o d I t s z a p c
  RDI: 0x0000000000000001  RSI: 0x00007FFFFFFFDA78  RDX: 0x00007FFFFFFFDA88  RCX: 0x00007FFFF7DD0718  RIP: 0x0000000000000000
  R8 : 0x00007FFFF7DD1D80  R9 : 0x00007FFFF7DD1D80  R10: 0x0000000000000007  R11: 0x0000000000000002  R12: 0x00000000004003B0
  R13: 0x00007FFFFFFFDA70  R14: 0x0000000000000000  R15: 0x0000000000000000
  CS: 0033  DS: 0000  ES: 0000  FS: 0000  GS: 0000  SS: 002B                    Error while running hook_stop:
Cannot access memory at address 0x0
0x0000000000000000 in ?? ()
gdb >> backtrace
#0  0x0000000000000000 in ?? ()
#1  0x00000000004004ac in main (argc=0x1, argv=0x7fffffffda78) at segv.c:4
```

GDB



- « GNU Debugger »
- Le debugging n'est pas magique, des informations supplémentaires sont ajoutées au binaire via le flag **`-g`**
- Extra informations dans les sections dédiées (voir format DWARF)
- Commande interactive. Un prompt est ouvert en attente de commandes utilisateur (ou d'un script)
- Lancer GDB : **`gdb ./a.out`** (ou **`file ./a.out`** au prompt)
- Arguments : option **`--args`** ou **`set args`** au prompt
- Démarrer le programme : **`run`**
- Quitter : **`q[uit]`** (ou Ctrl + D)
- Aide : `help <cmd>`
- `Spack install gdb`

```
En-têtes de section :
  [Nr] Nom                Type
  [ 0]                    NULL
  [ 1] .interp            PROGBITS
  [ 2] .note.ABI-tag      NOTE
  [ 3] .note.gnu.build-id NOTE
  [ 4] .gnu.hash          GNU_HASH
  [ 5] .dynsym            DYNSYM
  [ 6] .dynstr            STRTAB
  [ 7] .gnu.version       VERSYM
  [ 8] .gnu.version_r     VERNEED
  [ 9] .rela.dyn          RELA
  [10] .init              PROGBITS
  [11] .text              PROGBITS
  [12] .fini              PROGBITS
  [13] .rodata            PROGBITS
  [14] .eh_frame_hdr      PROGBITS
  [15] .eh_frame          PROGBITS
  [16] .init_array        INIT_ARRAY
  [17] .fini_array        FINI_ARRAY
  [18] .dynamic           DYNAMIC
  [19] .got               PROGBITS
  [20] .got.plt           PROGBITS
  [21] .data              PROGBITS
  [22] .bss               NOBITS
  [23] .comment           PROGBITS
  [24] .debug_aranges     PROGBITS
  [25] .debug_info        PROGBITS
  [26] .debug_abbrev      PROGBITS
  [27] .debug_line        PROGBITS
  [28] .debug_str         PROGBITS
  [29] .symtab            SYMTAB
  [30] .strtab            STRTAB
  [31] .shstrtab          STRTAB
```

GDB

- Afficher du contenu : `print` (ou `p`) de tout type. Le type doit être connu de GDB pour être affiché. Peut couvrir quasiment toute expression C (ex: **`a->b.c`**)
- La variable i : `print i`
- Le contenu du pointeur p : `print *p`
- Le contenu à l'adresse (de type T) : `print {T}addr`
- Rappel : VOID n'est pas un type défini !
- Briser une ambiguïté de portée : `print main::i`
- Variables GDB, préfixées par « `$` » : registres, retour de print…

Format	
a	Pointer.
c	Read as integer, print as character.
d	Integer, signed decimal.
f	Floating point number.
o	Integer, print as octal.
s	Try to treat as C string.
t	Integer, print as binary (*t* = „two“).
u	Integer, unsigned decimal.
x	Integer, print as hexadecimal.

```
gdb >> x $rsp
0x7fffffffdfe8: 0x00007ffff7ddf0b3
gdb >> x/10a $rsp
0x7fffffffdfe8: 0x7ffff7ddf0b3 <__libc_start_main+243>  0x7ffff7ffc620 <_rtld_global_ro>
0x7fffffffdff8: 0x7fffffffe0d8  0x100000000
0x7fffffffe008: 0x555555555231 <main>   0x5555555552b0 <__libc_csu_init>
0x7fffffffe018: 0xb0c75bfdb89320cb      0x5555555550e0 <_start>
0x7fffffffe028: 0x7fffffffe0d0  0x0
gdb >> x/10zg $rsp
0x7fffffffdfe8: 0x00007ffff7ddf0b3      0x00007ffff7ffc620
0x7fffffffdff8: 0x00007fffffffe0d8      0x0000000100000000
0x7fffffffe008: 0x0000555555555231      0x00005555555552b0
0x7fffffffe018: 0xb0c75bfdb89320cb      0x00005555555550e0
0x7fffffffe028: 0x00007fffffffe0d0      0x0000000000000000
```

GDB

- Le programme s'exécute comme s'il était hors de GDB. Un deadlock ou un SEGV peut donc se reproduire.
- Pour interrompre le programme pendant son exécution : **`Ctrl + C`**
- Principe de base : Une fois le programme stoppé, il est possible d'inspecter son contenu. Lorsqu'un programme est stoppé, il se trouve sur une instruction donnée, mappée dans l'espace mémoire du processus
- Idée : afficher la pile d'appels courante : **`bt`**
- Navigation entre les « frames » : **`up`** | **`down`** | **`frame #x`**

```
gdb >> bt
#0  0x00007ffff7bc78bd in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007ffff7bc0d05 in pthread_mutex_lock () from /lib64/libpthread.so.0
#2  0x000000000040070c in func (arg=0x7fffffffd96c) at threads.c:18
#3  0x00007ffff7bbe594 in start_thread () from /lib64/libpthread.so.0
#4  0x00007ffff78f1f4f in clone () from /lib64/libc.so.6
```

GDB

- Il est possible de définir statiquement des instructions où l'on souhaite stopper le programme pour pouvoir l'inspecter : le « point d'arrêt » ou **breakpoint**
- Peut être une adresse mémoire (0x…), un nom de fonction (symbole, par extension) ou un tuple (fichier, numéro de ligne), si l'option **`-g`** est passé à la compilation
- À chaque fois que le point d'arrêt est rencontré, le programme est suspendu
- Mettre un breakpoint : **`break <ref>`**
- Supprimer un breakpoint : **`delete <ref>`**
- Activer / désactiver un breakpoint : **`enable`** | **`disable <ref>`**
- Reprendre une exécution normale après un breakpoint : **`continue`**
- À considérer : `les watchpoints` (triggers sur changement de contenu)

```
gdb >> info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400508 in main at sample.c:9
        breakpoint already hit 1 time
2       breakpoint     keep y   0x00000000004004f0 in mult at sample.c:5
        breakpoint already hit 1 time
```

GDB

- Comment régler la question : « *Ma fonction est appelée 100 fois, comment mettre un breakpoint seulement sur certaines occurrences ?* »
- Idée : Les conditions
- La condition doit toujours être une expression C valide et doit pouvoir être évaluée comme un booléen (vrai/faux)
- Le contexte de la condition est la frame unrollée et non sa parente. Les variables locales sont donc accessibles
- Exemple: `break mult if a == 10`

GDB

- Un breakpoint est souvent posé à priori, sans savoir ou le programme présente un bug. On peut donc se poser la question : « *Comment continuer le programme jusqu'au bug tout en inspectant chaque instruction ?* »
- Idée : le « pas-à-pas » (= *stepping*)
- Une fois le breakpoint atteint, il est possible d'avancer **bloc par bloc**, revenant à mettre un breakpoint sur chaque ligne (=**`tbreak`**)
- `step` : ligne de code suivante (peut prendre un nombre en argument)
- `next` : ligne de code suivante, sans descendre dans les fonctions appelées
- continue : reprendre une exécution normale
- Pour avoir un grain **par instruction**, utiliser les versions suffixées par « i » (**`stepi`**, **`nexti`**…)
- Terminer la fonction courante (jusqu'au **`return`**) : **`finish`**

GDB

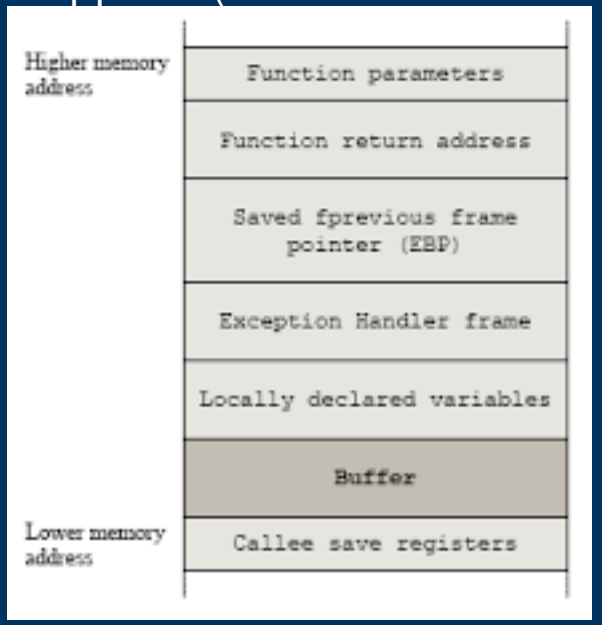
- Obtenir des informations: `info`
- Gestion des signaux : `handle`
- Quel type de variable ? : **`whatis`**
- Code machine ? **`Disassemble`**

```
gdb >> disassemble /m mult
Dump of assembler code for function mult:
7       {
   0x00000000004004e6 <+0>:     push   %rbp
   0x00000000004004e7 <+1>:     mov    %rsp,%rbp
   0x00000000004004ea <+4>:     mov    %edi,-0x4(%rbp)
   0x00000000004004ed <+7>:     mov    %esi,-0x8(%rbp)

8               return a * b;
   0x00000000004004f0 <+10>:    mov    -0x4(%rbp),%eax
   0x00000000004004f3 <+13>:    imul   -0x8(%rbp),%eax

9       }
   0x00000000004004f7 <+17>:    pop    %rbp
   0x00000000004004f8 <+18>:    retq

End of assembler dump.
gdb >>
```

```
gdb >> help info
Generic command for showing things about the program being deb

List of info subcommands:

info address -- Describe where symbol SYM is stored
info all-registers -- List of all registers and their contents
info args -- Argument variables of current stack frame
info auto-load -- Print current status of auto-loaded files
info auxv -- Display the inferior's auxiliary vector
info bookmarks -- Status of user-settable bookmarks
info breakpoints -- Status of specified breakpoints (all user-
info checkpoints -- IDs of currently known checkpoints
info classes -- All Objective-C classes
info common -- Print out the values contained in a Fortran COM
info copying -- Conditions for redistributing copies of GDB
info dcache -- Print information on the dcache performance
info display -- Expressions to display when program stops
info exceptions -- List all Ada exception names
info extensions -- All filename extensions associated with a s
info files -- Names of targets and files being debugged
info float -- Print the status of the floating point unit
info frame -- All about selected stack frame
info frame-filter -- List all registered Python frame-filters
info functions -- All function names
info guile -- Prefix command for Guile info displays
info handle -- What debugger does when program gets various si
info inferiors -- IDs of specified inferiors (all inferiors if
info line -- Core addresses of the code for a source line
info locals -- Local variables of current stack frame
info macro -- Show the definition of MACRO
info macros -- Show the definitions of all macros at LINESPEC
info mem -- Memory region attributes
info os -- Show OS data ARG
info pretty-printer -- GDB command to list all registered pret
info probes -- Show available static probes
info proc -- Show /proc process information about any running
info program -- Execution status of the program
info record -- Info record options
info registers -- List of integer registers and their contents
info scope -- List the variables local to a scope
info selectors -- All Objective-C selectors
info set -- Show all GDB settings
```

Rappel: Layout de la pile

- La pile est une superposition couches appelées « frame », toutes identiques. Une stackframe est créée à chaque fois qu'une nouvelle fonction est appelée (instruction x86 **`call*`**)
- Dans une stack-frame est stockée :
  - Les arguments de fonctions
  - L'adresse de retour RIP dans la fonction parente (pour *Return Instruction Pointer*)
  - Le pointeur de pile **RBP** de la frame précédente
  - Les variables automatiques (dites « locales »)
- Il existe deux pointeurs de pointeurs de pile
  - RBP : « Base pointer » = l'adresse où commence la frame courante
  - RSP : « Stack Pointer » = l'adresse qui suit la dernière adresse accessible pour la frame courante




```
gdb >> disas main
Dump of assembler code for function main:
   0x000000000040052e <+0>:     push   %rbp
   0x000000000040052f <+1>:     mov    %rsp,%rbp
   0x0000000000400532 <+4>:     sub    $0x10,%rsp
```

Manipulation des Registres:

- **r* = 64 bits**
- **e* = 32 bits**
- **l* = 16 bits**

GDB : Threading

- Un des gros avantages d'un debugger est d'aider à gérer plusieurs flots d'exécution comme les pthreads.
- Lister les threads : **`info threads`**
- Par défaut, le thread #0 est celui actif
- Changer de thread: **`thread #x`**
- Exécuter une même commande sur plusieurs threads:

  `thread apply #x #y <cmd>`
- Par défaut, un breakpoint est mis à l'échelle d'un processus
- Un breakpoint thread-specific : **`break <ref> thread #x if …`**
- Attention, un thread doit exister pour être utilisé dans un breakpoint !

GDB : Multi-processing

- Par défaut, GDB ne « suit » pas les processus descendants du processus courant. Configurable via la variable **`follow-fork-mode`**
  - Child: Debug du processus créé
  - Parent: Debug du processus initial (défaut)
- Mais qu'arrive-t-il à l'autre processus ? Configurable via la variable detach-on-fork :
  - On : détache le processus non sélectionné (défaut)
  - Off : GDB suit les deux processus. Celui non sélectionné est mis en suspens (défaut)
- Que se passe-t-il si l'application invoque **`exec*()`** ? **`Follow-exec-mode`**

GDB : Scripting

- Souvent en HPC, il n'est pas possible ou pratique d'avoir un prompt interactif (ex: app avec 64 processus)
- GDB fournit une interface légère de scripting soumis via la ligne de commande : **`-x file | -command=file`**
- Possible aussi via stdin : **`gdb < file`**
- Ou chargeable depuis le prompt : **`source file`**
- Prologue :
  - Set breakpoint pending on : Mise en place de breakpoints sans connaitre le mapping actuel en mémoire. GDB n'avertit pas lorsque le breakpoint n'a pas pu être posé
  - Set pagination off : Éviter la pagination (« Type <return> to continue… »)
  - Set logging on: conserver la sortie (set logging file)
- Epilogue : **`run`**

Valgrind

- `Spack install valgrind`
- Framework d'instrumentation: fournit les méthodes pour permettre à des outils d'analyse dynamiques
- **Memcheck**: problème de gestion mémoire
  - Accès mémoire invalides
  - Utilisation de mémoire non-initialisée
  - Fuites
  - Double frees & Co
  - Overlap de memcpy

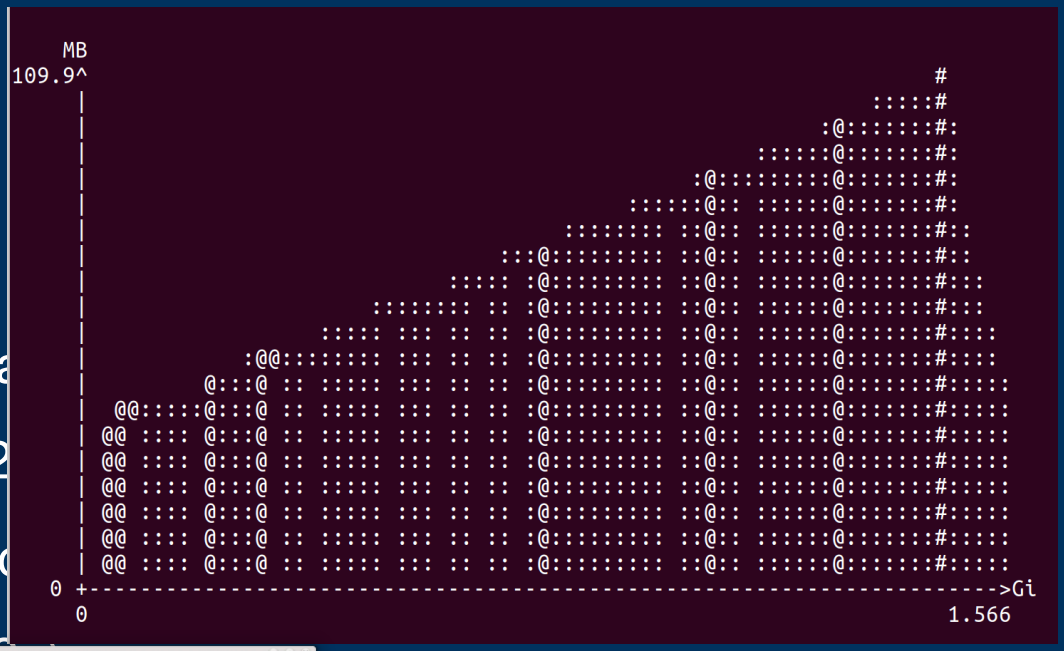
```
==1275213==
==1275213== HEAP SUMMARY:
==1275213==     in use at exit: 100 bytes in 1 blocks
==1275213==   total heap usage: 3 allocs, 2 frees, 2,148 bytes allocated
==1275213==
==1275213== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1275213==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpre
==1275213==    by 0x1092BB: main (in /home/adamj/a.out)
==1275213==
==1275213== LEAK SUMMARY:
==1275213==    definitely lost: 100 bytes in 1 blocks
==1275213==    indirectly lost: 0 bytes in 0 blocks
==1275213==      possibly lost: 0 bytes in 0 blocks
==1275213==    still reachable: 0 bytes in 0 blocks
==1275213==         suppressed: 0 bytes in 0 blocks
==1275213==
==1275213== For lists of detected and suppressed errors, rerun with: -s
==1275213== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

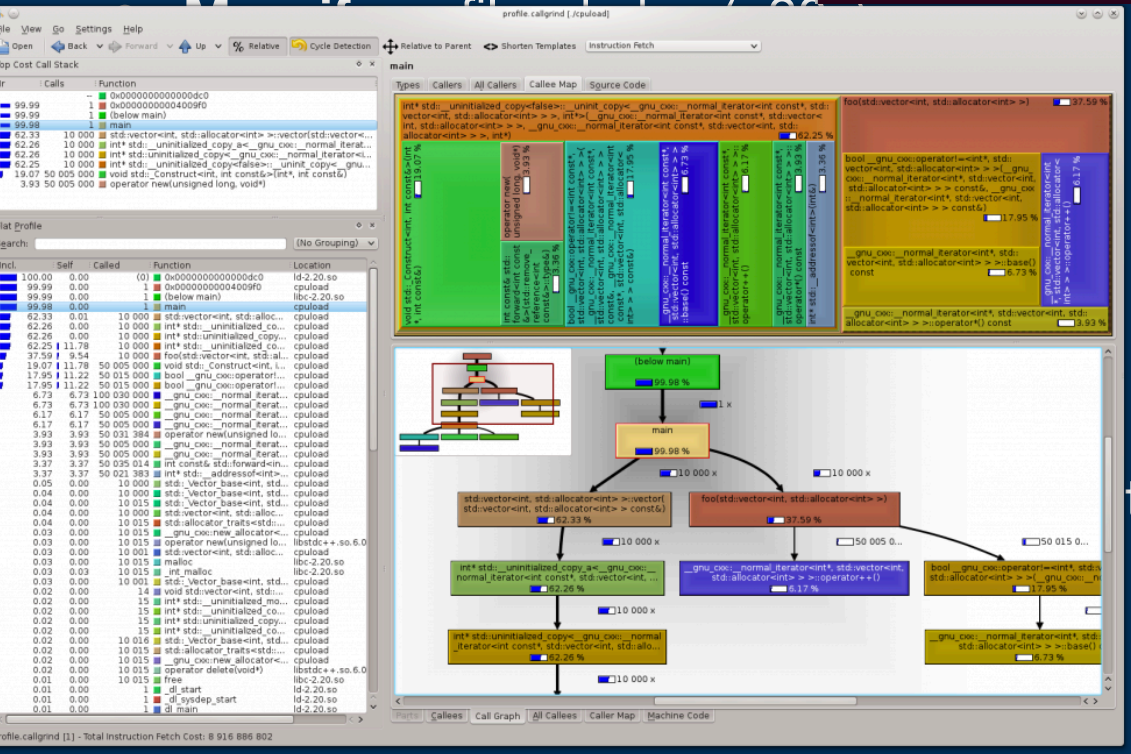
Valgrind

- Mais attention, instrumentation = overhead
- Codes sous memcheck ~20x plus lent
- **Cachegrind**: profiler d'accès aux caches (~70x)
- **Massif**: profiler de tas (~20x)
- **Callgrind**: graphe d'appel
- Helgrind: Threads
- …
- Certains outils ont une sortie analytique directe, d'autres ont besoin d'outils de visualisation pour être exploitable

Valgrind

- Mais attention, instrumenta
- Codes sous memcheck ~2
- **Cachegrind**: profiler d'acc





directe, d'autres ont besoin

table

Perf

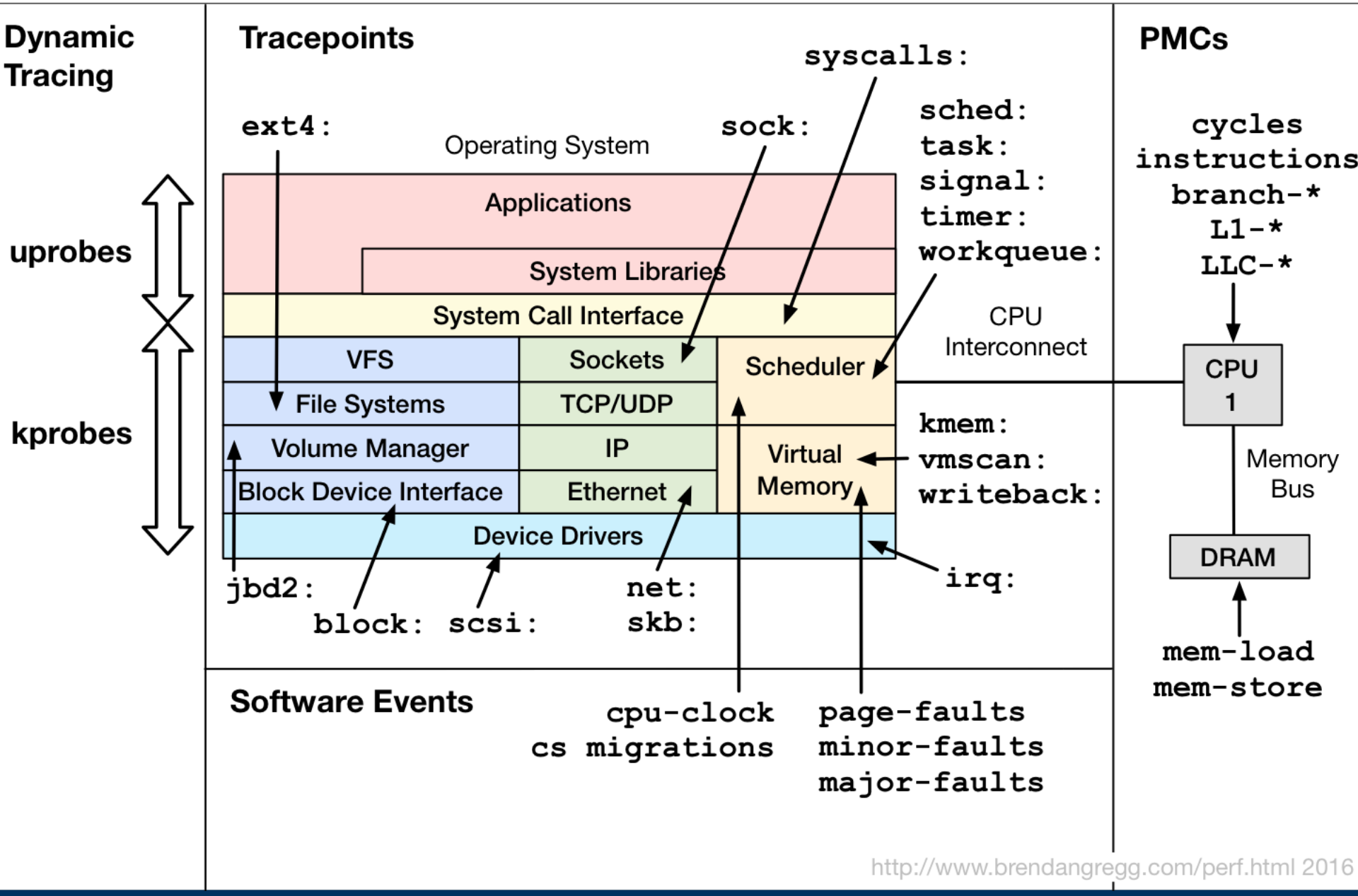
- Disponible depuis Linux kernel version 2.6.31
- Trace les compteurs relativement aux piles
- Très utilisé pour compter les cycles d'un app
- Mais peut faire beaucoup plus
- Une seule CLI: « perf »
- Supporte le « tracing » et le « sampling »

➡`yum install perf`

➡apt-get install linux-tools-common linux-tools-generic linux-tools-`uname -r`

➡git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

Linux perf_events Event Sources



Perf

- Se base beaucoup sur ptrace()
- Limitation des processus
- Limitation des capacités allouées par root

```
$ perf record -a

Error:
You may not have permission to collect system-wide stats.

Consider tweaking /proc/sys/kernel/perf_event_paranoid,
which controls use of the performance events system by
unprivileged users (without CAP_SYS_ADMIN).

The current value is 1:

  -1: Allow use of (almost) all events by all users
>= 0: Disallow raw tracepoint access by users without CAP_IOC_LOCK
>= 1: Disallow CPU event access by users without CAP_SYS_ADMIN
>= 2: Disallow kernel profiling by users without CAP_SYS_ADMIN
```

Perf

- Stats globales du programme:

```
$perf stat date
Thu May 24 17:44:55 CEST 2018

 Performance counter stats for 'date':

          0.889236      task-clock:u (msec)       #    0.311 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
               156      page-faults:u             #    0.175 M/sec
            452575      cycles:u                  #    0.509 GHz
            211872      instructions:u            #    0.47  insn per cycle
             47041      branches:u                #   52.900 M/sec
              4193      branch-misses:u           #    8.91% of all branches

       0.002858312 sec
```

```
$perf stat -d date
Thu May 24 17:46:55 CEST 2018

 Performance counter stats for 'date':

          0.888090      task-clock:u (msec)       #    0.639 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
               156      page-faults:u             #    0.176 M/sec
                 0      cycles:u                  #    0.000 GHz
            211872      instructions:u
             47041      branches:u                #   52.969 M/sec
              4175      branch-misses:u           #    8.88% of all branches
             55889      L1-dcache-loads:u         #   62.932 M/sec
              6000      L1-dcache-load-misses:u   #   10.74% of all L1-dcache hits
   <not supported>      LLC-loads:u
   <not supported>      LLC-load-misses:u

       0.001389281 seconds time elapsed
```

Perf

- Plusieurs exécutions pour lisser le bruit: -r <rep>

```
Performance counter stats for 'date' (1000 runs):

        0,285749      task-clock (msec)         #    0,643 CPUs utilized            ( +-  0,06% )
               2      context-switches          #    0,007 M/sec                    ( +-  0,18% )
               0      cpu-migrations            #    0,000 K/sec
             186      page-faults               #    0,651 M/sec                    ( +-  0,00% )
         855 254      cycles                    #    2,993 GHz                      ( +-  1,39% )  (0,00%)
         810 631      instructions              #    0,95  insn per cycle           ( +-  0,04% )
         158 770      branches                  #  555,629 M/sec                    ( +-  0,03% )
           6 739      branch-misses             #    4,24% of all branches          ( +-  0,07% )
         216 810      L1-dcache-loads           #  758,743 M/sec                    ( +-  0,03% )
    <not counted>     L1-dcache-load-misses                                        ( +-  4,20% )  (23,72%)
    <not counted>     LLC-loads                                                    (0,00%)
    <not counted>     LLC-load-misses                                              (0,00%)

      0,000444485 seconds time elapsed                                          ( +-  0,11% )
```

perf list

- Liste les évènements disponibles:

$ perf list -v # List all events you can use (-v for verbose desc.)

$ perf list -v float # List all events linked to FP operations

$ perf list -v l3 # List all events linked to the L3 cache

perf record

Récupère les stats d’un code

```
$ perf record -g date # Record cycles for « date »
```

« -g » capture les call stacks

Résultats dans « perf.data »

```
$perf record -g -e L1-dcache-load-misses date
```

Selection d’events avec « -e », il vaut mieux limiter leur nombre pour éviter un salin trop important

perf report

```
$ perf record -g date
$ perf report --stdio --sort parent
```

```
100.00%   100.00%  [other]
          |
          |--37.76%--_dl_sysdep_start
          |          dl_main
          |          |
          |           --29.08%--_dl_relocate_object
          |                     _dl_lookup_symbol_x
          |
          |--20.32%--_dl_addr
          |
          |--20.32%--0x2342
          |
          |--13.85%--0x1c33a
          |          _dl_map_object
          |          open_path
          |          open_verify
          |
          |--4.46%--strlen
          |
          |--2.05%--_dl_start_user
          |          _dl_start
          |          async_page_fault
          |
           --1.06%--_dl_start
                     |
                      --0.75%--async_page_fault
```

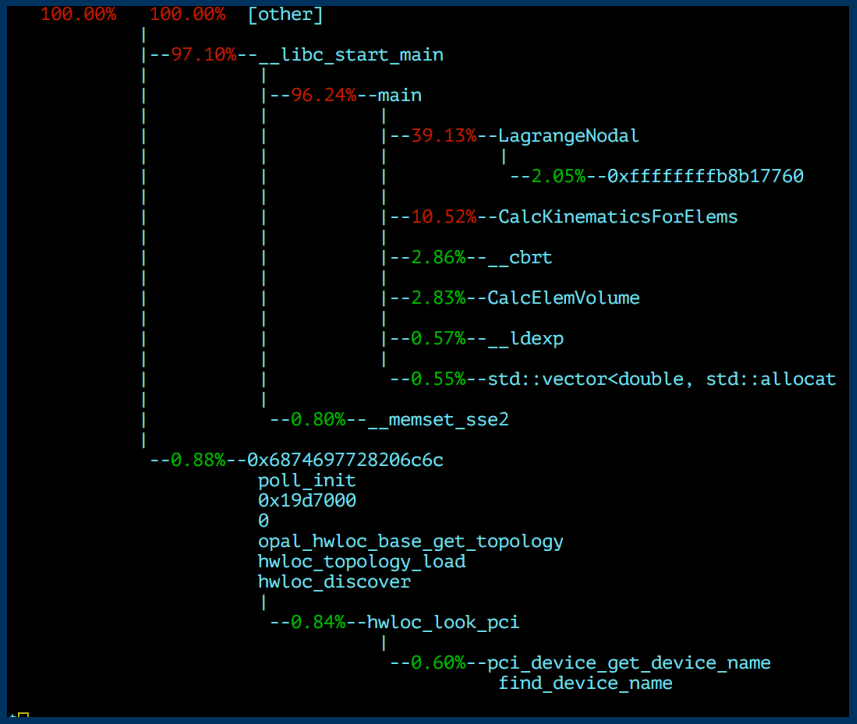
perf report « lulesh »

```
$ perf record  -e cycles -g ./lulesh2.0 -i 50
$ perf report  -g --sort parent
```

```
Samples: 3K of event 'cycles:u', Event count (approx.): 3025875782
  Children      Self  Parent symbol
-  100,00%   100,00%  [other]
   - 97,10% __libc_start_main
      - 96,24% main
         - 39,13% LagrangeNodal
              0xffffffffb8b17760
           10,52% CalcKinematicsForElems
           2,86% __cbrt
           2,83% CalcElemVolume
           0,57% __ldexp
           0,55% std::vector<double, std::allocator<double> >::resize
        0,80% __memset_sse2
   - 0,88% 0x6874697728206c6c
        poll_init
        0x19d7000
        0
        opal_hwloc_base_get_topology
        hwloc_topology_load
      - hwloc_discover
         - 0,84% hwloc_look_pci
            - 0,60% pci_device_get_device_name
                 find_device_name
```

perf report « lulesh »

```
$ perf report  -g --sort parent --stdio
```



perf top

```
$ yes > /dev/null&
[1] 13755
$ perf top -p 13755
```

```
Samples: 94K of event 'cycles:u', Event count (approx.): 53656556765
Overhead  Shared Object      Symbol
  32,84%  libc-2.17.so       [.] _IO_file_xsputn@@GLIBC_2.2.5
  22,74%  libc-2.17.so       [.] fputs_unlocked
  13,46%  libc-2.17.so       [.] __strlen_sse2
  12,45%  libc-2.17.so       [.] __GI___mempcpy
   2,60%  yes                [.] 0x00000000000015af
   2,59%  yes                [.] 0x00000000000015c5
   2,55%  yes                [.] 0x00000000000015a0
   2,52%  yes                [.] 0x00000000000012f0
   2,15%  yes                [.] 0x0000000000001590
   2,13%  yes                [.] 0x0000000000001628
   2,11%  yes                [.] 0x000000000000159c
   0,44%  yes                [.] 0x00000000000015c8
   0,39%  yes                [.] 0x0000000000001595
   0,39%  yes                [.] 0x00000000000015d5
   0,38%  yes                [.] 0x000000000000162f
   0,04%  libc-2.17.so       [.] _IO_do_write@@GLIBC_2.2.5
For a higher level overview, try: perf top --sort comm,dso
```

Trace de programme en temps réel

perf top -a

```
# perf top -a -e syscalls:sys_enter_open
```

Trace global (à l'échelle du système) du syscall **`open`** (kernel tracepoint)

REQUIRES ROOT

```
syscalls:sys_enter_accept4                [Tracepoint event]
syscalls:sys_enter_access                 [Tracepoint event]
syscalls:sys_enter_acct                   [Tracepoint event]
syscalls:sys_enter_add_key                [Tracepoint event]
syscalls:sys_enter_adjtimex               [Tracepoint event]
syscalls:sys_enter_alarm                  [Tracepoint event]
syscalls:sys_enter_bind                   [Tracepoint event]
syscalls:sys_enter_brk                    [Tracepoint event]
syscalls:sys_enter_capget                 [Tracepoint event]
syscalls:sys_enter_capset                 [Tracepoint event]
syscalls:sys_enter_chdir                  [Tracepoint event]
syscalls:sys_enter_chmod                  [Tracepoint event]
syscalls:sys_enter_chown                  [Tracepoint event]
syscalls:sys_enter_chroot                 [Tracepoint event]
syscalls:sys_enter_clock_adjtime          [Tracepoint event]
syscalls:sys_enter_clock_getres           [Tracepoint event]
```

Perf est **beaucoup** plus performant en root, car un grand nombre d'events ne sont disponibles qu'avec un utilisateur privilégié

Perf et programmes MPI

Perf est orienté processus. Pour fonctionner avec un code MPI, il est nécessaire de séparer la sortie dans des fichiers séparés. Ici, un simple script fait l'affaire.

```
#!/bin/sh
H=`hostname`
P=$$
perf record -o perf-${H}-${P}.data  $@
```

```
$ srun -n 8 script.sh -g ./matmult
```

```
$ perf report -i <one-perf-file>.data -g
```

Flamegraphs

```
$ git clone https://github.com/brendangregg/FlameGraph
$ cd FlameGraph
$ perf record -F 99 -a -g -- sleep 60
$ perf script | ./stackcollapse-perf.pl > out.perf-folded
$ ./flamegraph.pl out.perf-folded > perf-kernel.svg
```

Astuce: fournir un script qui fait ça pour nous à chaque appel:

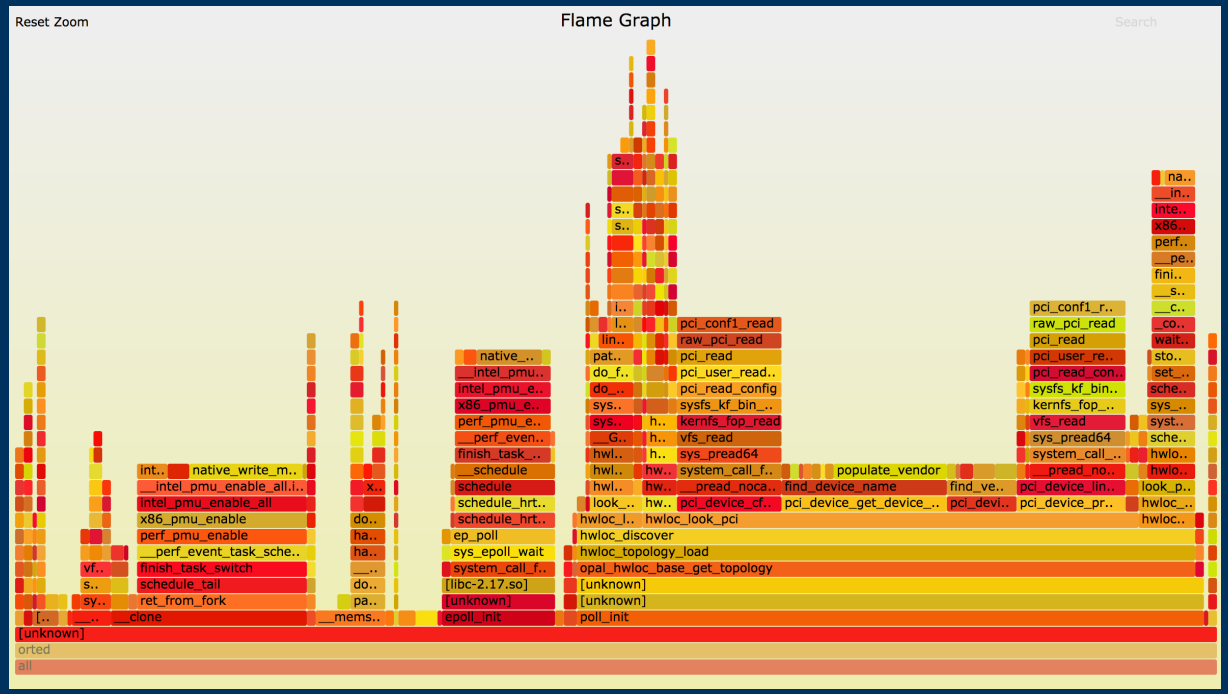
```
#!/bin/sh

FL_PREFIX=WHERE_YOU_CLONED_FLAMEGRAPH

perf script $@ | $FL_PREFIX/stackcollapse-perf.pl | $FL_PREFIX/flamegraph.pl
```

Flamegraphs

#You can now generate flamegraphs for all your metrics !

```
$ perf record -g ./lulesh2.0 -i 50 -p -s 100
```

```
$ flg > out.svg
```



Haute interactivité dans un navigateur