GDB & Reverse

Master CHPS

Julien Adam <adamj@paratools.com>

Organisation du Cours

Cour MATIN, TD après-midi

- **10/01: Généralités sur les OS et Utilisation de base**
- **15/01 : La Chaine de Compilation et l'exécution d'un programme**
- **16/01 : Les I/Os POSIX et Introduction aux Sockets**
- **30/01 : Méthodes de communication Inter-Processus**
- **01/02 : Mémoire avancée (mmap, madvise, pages, TLB, …)**
- 06/02 : GDB et reverse et Q/A projets (journée TD)
- 13/02 : TD Déboggage (gdb, valgrind) && TD mesure du temps et profilage (perf, kcachegrind) et Q/A projets (journée TD)
- Un examen final (25 Mars matin)

Plan du jour

1. Bases du scheduling de processus
2. Debugging d’un programme
3. La « stackframe »
4. Temps dédié aux projets

Outils utiles

- Compilateurs (C): **`gcc`**, `icc`, `xlc`, `clang`, `pgcc`…
    - Dont intermédiaires : `cpp`, `as`, `ld/gold`
- Debuggers : **`gdb`**, `ddt`, `lldb`, `adb`
- Analyse binaire (*disassembling*) :
    - ELF : **`readelf`**, `hte`, `elfedit`, **`nm`**
    - Objets : **`objdump`**, `objcopy`
    - Conversion : `xxd`, `hexdump`, `base64`
- Bonus : `radare2`, `peda`
- Opcodes x86_64 : http://ref.x86asm.net/coder64.html
- Sources : https://github.com/gweodoo/aise.git

Bug d’un programme

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

Bug d’un programme

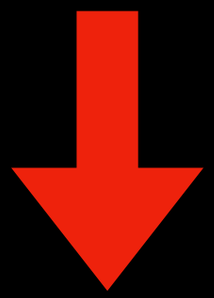
- Qu’est-ce qu’un programme qui « bug » ?
  - Crash (SEGV par exemple)
  - Résultat différent de ce qui est attendu
  - Interblocage (deadlock)
- **Idée : Suivre l’exécution du programme (flot & variables)**
- L’outil du débutant en debug : **`printf`**
  - Avantages : Simple, aucune connaissance à priori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

```
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 <u>défini</u> !**
- 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

- 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

- **<u>À chaque fois que le point d'arrêt est rencontré, le programme est suspendu</u>**

- 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 <u>toujours</u> 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
```

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*)

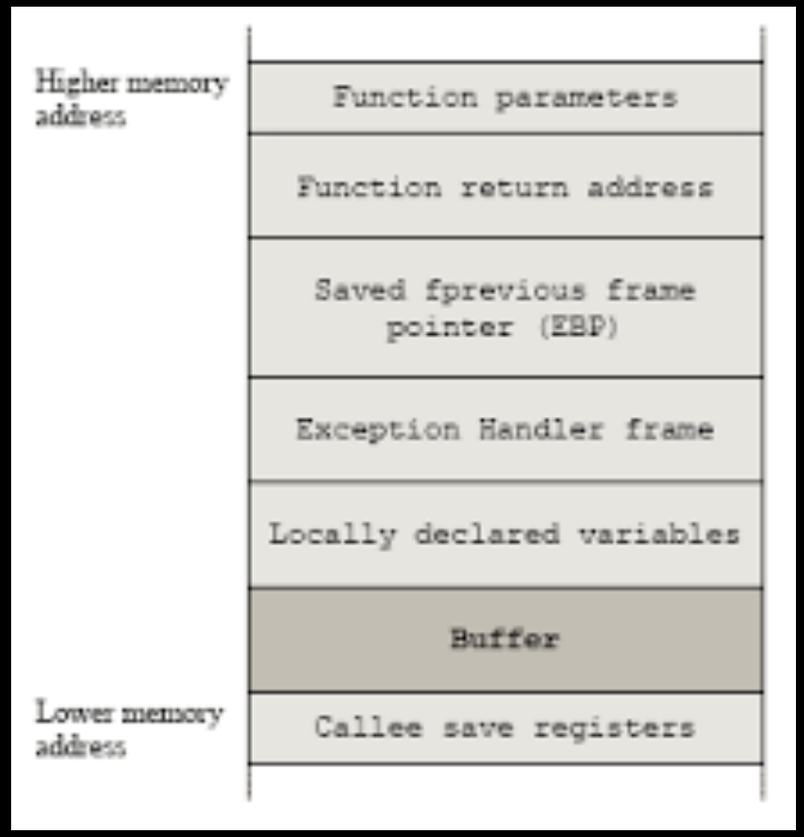
    - Le pointeur de pile **RBP** de la frame précédente

    - 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 **<u>qui suit</u>** 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

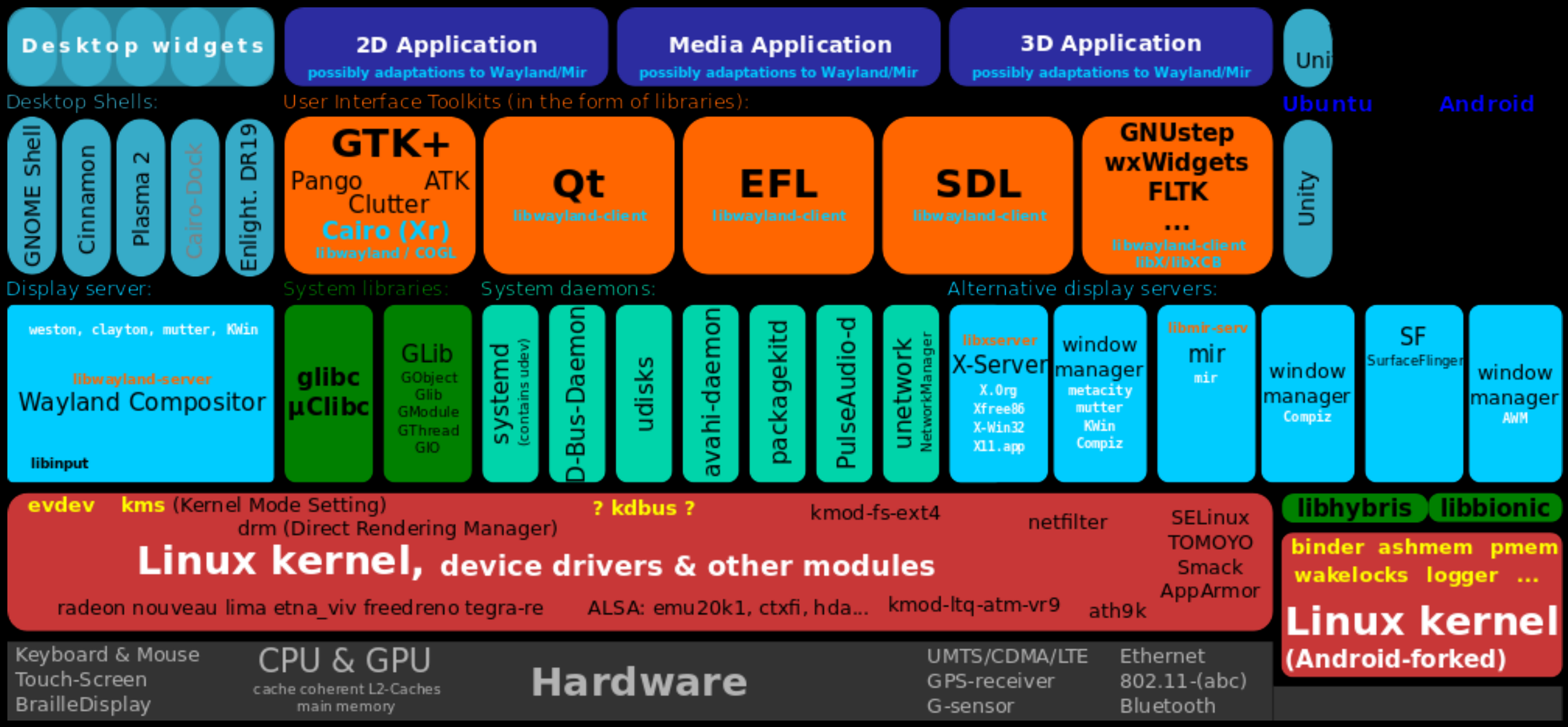
- Souvent en HPC, il n'est pas possible ou pratique d'avoir un prompt interactif (ex: app avec 64 processus)
- GDB fournit une interface légère de scripting soumis via la ligne de commande : **`-x file`** | **`-command=file`**
- Possible aussi via stdin : **`gdb < file`**
- Ou chargeable depuis le prompt : **`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`**

GDB : Scripting

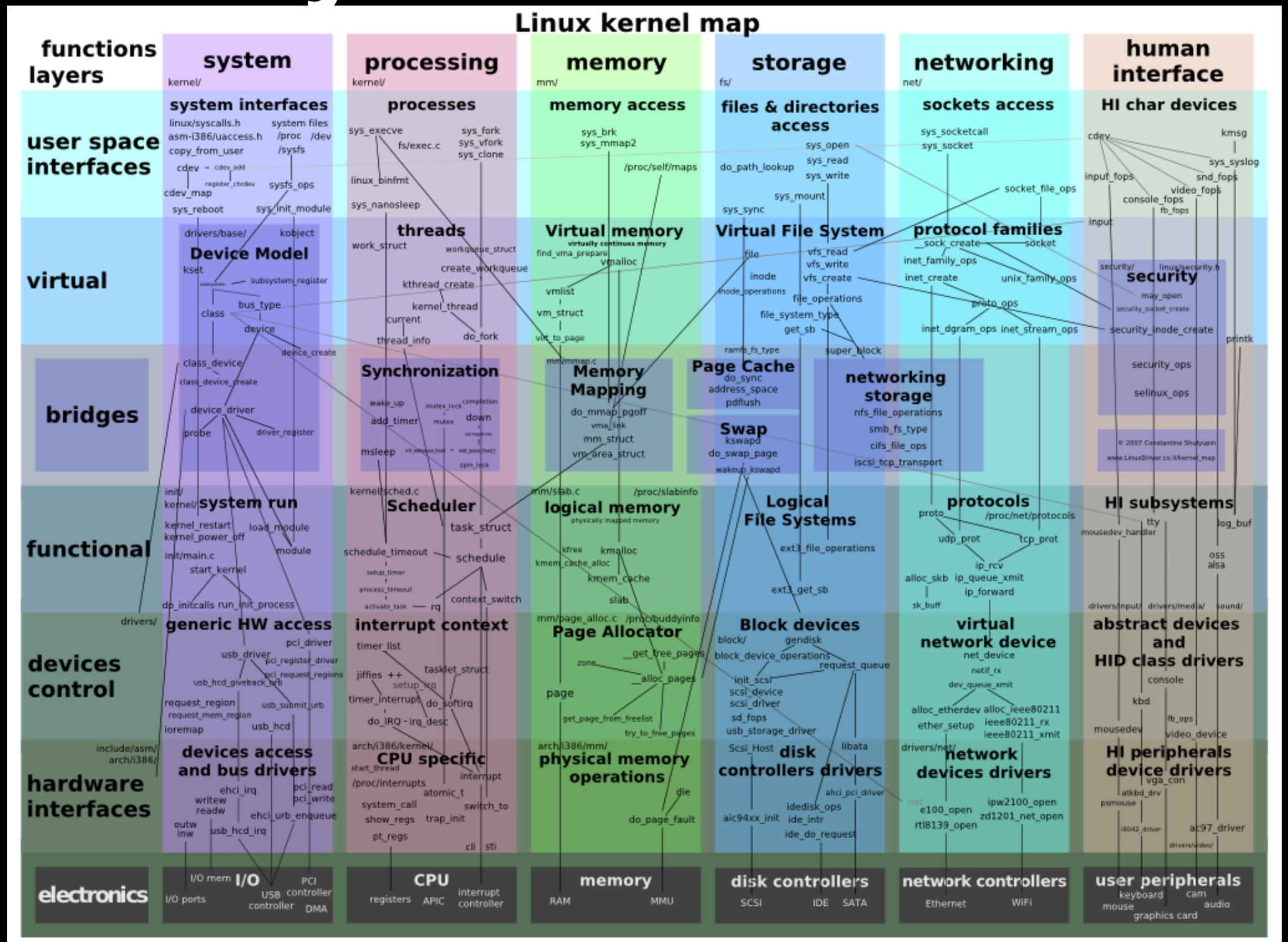
```
set breakpoint pending on
set pagination off
set logging file my_out
set logging on

set $var = 0
break func if a == 32
    command 1
    print a
    continue
end

break sample.c:9 if it == 6
    command 2
    print y
    continue
end

break main
    command 3
    backtrace
    continue
end

run

set logging off
quit
```

Pause ?

Système Linux



Système Linux



Isolation et privilèges

- Pour assurer une isolation des ressources du système (I/O, net, CPU), les processus sont rangés en deux groupes : **privilégiées ou non**

- Les processus privilégiés court-circuitent toutes les vérifications de permissions (programmes root par exemple)

- Les autres doivent être « élus » pour passer d'un niveau de privilèges à un autre. C'est ce qu'on appelle les anneaux de protections.

- Le niveau ultime est l'accès aux ressources hardwares, uniquement depuis le noyau.

- Afin d'éviter un modèle « tout ou rien », le noyau Linux range les accès aux ressources par groupes : les **capacités** (=capabilities). Un programme peut ainsi recevoir un ensemble de capacités nécessaires à sa tache sans être un programme privilégié, générateur de failles de sécurité.

```
cap_audit_control
cap_audit_read
cap_audit_write
cap_block_suspend
cap_chown
cap_dac_override
cap_dac_read_search
cap_fowner
cap_fsetid
cap_ipc_lock
cap_ipc_owner
cap_kill
cap_lease
cap_linux_immutable
cap_mac_admin
cap_mac_override
cap_mknod
cap_net_admin
cap_net_bind_service
cap_net_broadcast
cap_net_raw
cap_setfcap
cap_setgid
cap_setpcap
cap_setuid
cap_sys_admin
cap_sys_boot
cap_sys_chroot
cap_syslog
cap_sys_module
cap_sys_nice
cap_sys_pacct
cap_sys_ptrace
cap_sys_rawio
cap_sys_resource
cap_sys_time
cap_sys_tty_config
cap_wake_alarm
```

```
└─▸ getcap `which ping`
/usr/bin/ping = cap_net_admin,cap_net_raw+p
```

SUID

- Un moyen de transfert de droits entre utilisateurs.
- Un programme Suid est exécuté « au nom de l'initiateur du transfert ».
- **Tous** les droits de l'utilisateur initial sont propagés à l'utilisateur destinataire (« déguisement »)
- Pourquoi ? Considérer le programme **`passwd`**
- Cela peut-il poser un problème ?
- Mettre le « sticky bit » sur un programme : **`chmod +s`**

```
└─▸ ls -l ./a.out
-rwsrwxr-x. 1 adamj adamj 8296  5 févr. 21:51 ./a.out
```

Ordonnancement

- **`SCHED_FIFO`** : Premier arrivé, premier servi
- **`SCHED_RR`** : principe du « Fair share » (round-robin)
- **`SCHED_OTHER`** : politique par défaut, souvent RR
  - Définition d'une *epoch* : unité de calcul d'un code avant sa préemption (=*timeslice*)
- **`SCHED_IDLE`**: Hint sur un code CPU-bound (type « background »)
- **`SCHED_BATCH`**: Groupage de processus
- « Surcoûche » de priorité ajoutée au-dessus de ces modèles (statique, dynamique, temps-réel)
- **Routines : `sched_set/getscheduler()`**
- **`man sched`**

- Linux est un OS à temps partagé
- Concept de <u>préemption</u> = interrompre un processus au début de n'importe quelle instruction pour pouvoir réutiliser la ressource pour un autre fil d'exécution
- **<u>Mode noyau : non interruptible</u>**