Techniques et Outils d'Ingénierie Logicielle

•••

Jean-Baptiste Besnard

Temporalité (Peut bouger car auto-adaptatif!)

- Cours 1:
  - Maitriser un shell et ses automatisations
  - Avoir un environnement de développement productif
  - Savoir compiler un programme (lib, makefile, gcc, search paths)
  - Comprendre les phases de la compilation
- Cours 2:
  - Cycle de vie d'un programme
  - Documentation
  - Gestion de code source (Versionning) == GIT
  - Principe de forge (Issue, MR, fork, Pull)
  - Notion d'open Source
  - Gestion des conflits de code-source
- **Cours 3:**
  - **Débogage de programmes**
  - **Principe et structure des débogueurs**
  - **Utilisation de GDB**
  - **Structure d'un binaire**
  - **Structure d'un binaire lors de l'éxécution**
  - **Débogage mémoire**
  - **Gestion d'erreur**

De S9 à 12 en alternance cours/TP principalement le Mercredi matin. Double TP en S11.

Pour Récupérer Les Slides et Exemples

git clone https://github.com/besnardjb/TOI24

Layout Mémoire d’un Programme

Processus Cannonique

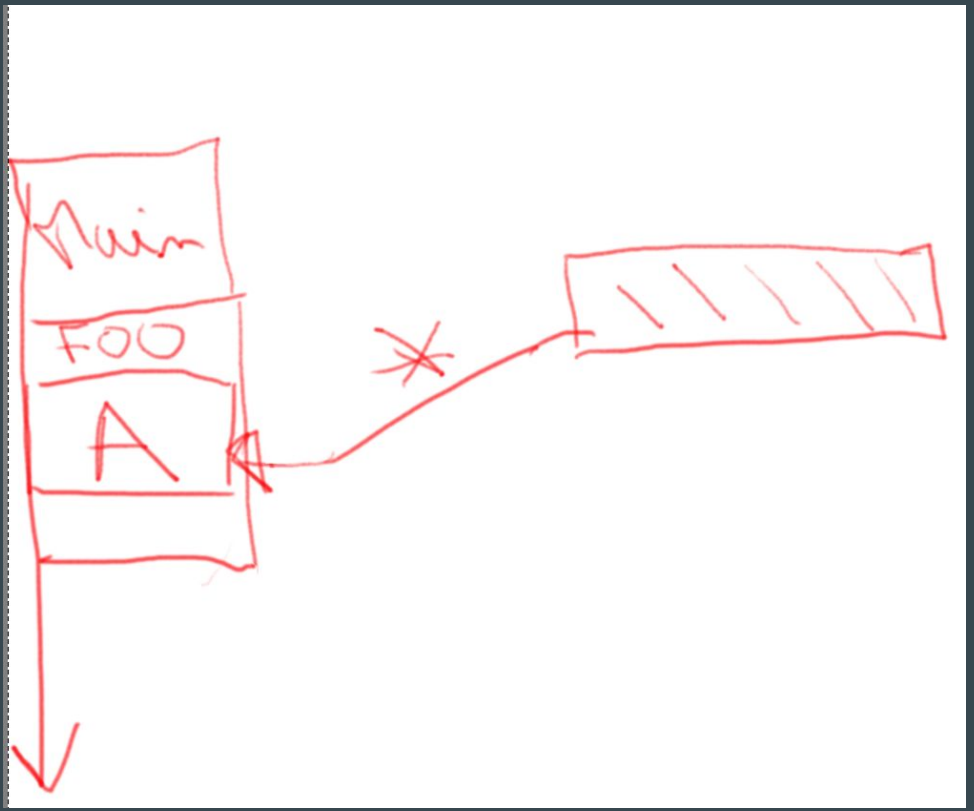
La Pile

- La pile descend vers le bas
- Elle est de taille finie
- Chaque thread a sa pile
- On y empile les **frames** d'appel de fonctions
  - Registres et addresse de retour
  - Variables locales

Corruption de Pile

- Pour chaque fontion les variables sont cote à cote il est donc possible d'accidentellement les faire se modifier (dépassement de tampon) ou Stack-Overflow
- Si la pile devient trop grande: Stack overflow
- Il est possible de corompre l'addresse de retour de frame pour appeler un autre code
- Les corruptions de pile sont un problème de sécurité et difficiles à déboguer

Le Tas



- Zone mémoire large
- Gérée par un allocateur:
  - malloc
  - free
- Une variable de la pile peut contenir une addresse du tas
- Espage paginé (4 KB)

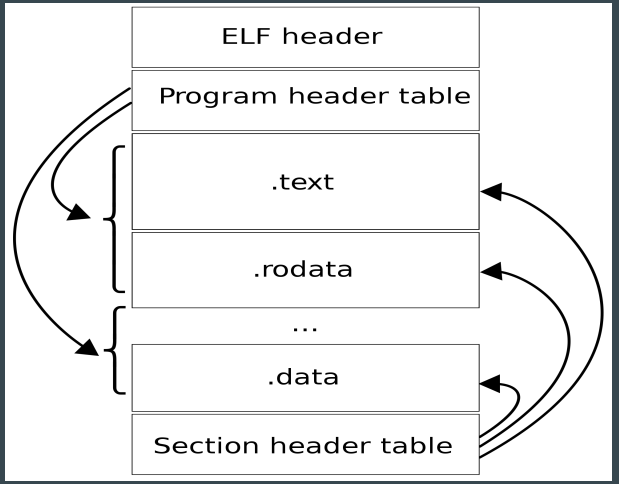
Corruption mémoire et le Tas

- Dans le tas il est souvent plus difficile de voir les corruptions:
  - Pages par blocs de 4k (pas de segfault pour overflow +1-1 par exemple).
  - Larges zones et gestion de l'allocateur en interne sur les tampons
  - Il est possible de modifier des zones d'une autre variable accidentellement
  - Enfin, les fuites mémoires existent aussi

Le Format Elf

Format ELF

- ELF = *Executable and Linkable Format*
- Décrit comment un binaire doit être représenté pour être compris par le lanceur de processus (`ld-linux.so`)
- Un programme contient beaucoup d'informations. Pour rester cohérent, il est segmenté en plusieurs sections
- Séquence magique : « **`7F 45 4C 46`** » = 7F « ELF »
- L'entête du ELF contient toutes les informations nécessaires à l'architecture (32/64 bits, endianness, ABI, type de fichier, jeu d'instruction...)




```
En-tête ELF:
  Magique:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Classe:                            ELF64
  Données:                           complément à 2, système à octets
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  Version ABI:                       0
  Type:                              EXEC (fichier exécutable)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Adresse du point d'entrée:         0x4003b0
  Début des en-têtes de programme :  64 (octets dans le fichier)
  Début des en-têtes de section :    6360 (octets dans le fichier)
  Fanions:                           0x0
  Taille de cet en-tête:             64 (octets)
  Taille de l'en-tête du programme:  56 (octets)
  Nombre d'en-tête du programme:     9
  Taille des en-têtes de section:    64 (octets)
  Nombre d'en-têtes de section:      27
  Table d'indexes des chaînes d'en-tête de section: 26
```

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Format ELF

- **Program header** : Stocke les informations nécessaires à la création de l'image du processus. Structure le programme d'un point de vue mémoire

- **Section header** : Regroupe les informations nécessaires au bon fonctionnement du programme. Structure le programme d'un point de vue fonctionnel

- Le reste du ELF est composé de blocs d'instructions, indexées dans l'une et/ou l'autre de ces tables

```
En-têtes de section :
  [Nr] Nom              Type            Adr              Décala.Taille ES Fan LN Inf Al
  [ 0]                  NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp          PROGBITS        0000000000400238 000238 00001c 00   A  0   0  1
  [ 2] .note.ABI-tag    NOTE            0000000000400254 000254 000020 00   A  0   0  4
  [ 3] .note.gnu.build-id NOTE           0000000000400274 000274 000024 00   A  0   0  4
  [ 4] .gnu.hash        GNU_HASH        0000000000400298 000298 00001c 00   A  5   0  8
  [ 5] .dynsym          DYNSYM          00000000004002b8 0002b8 000048 18   A  6   1  8
  [ 6] .dynstr          STRTAB          0000000000400300 000300 000040 00   A  0   0  1
  [ 7] .gnu.version     VERSYM          0000000000400340 000340 000006 02   A  5   0  2
  [ 8] .gnu.version_r   VERNEED         0000000000400348 000348 000020 00   A  6   1  8
  [ 9] .rela.dyn        RELA            0000000000400368 000368 000030 18   A  5   0  8
  [10] .init            PROGBITS        0000000000400398 000398 000017 00  AX  0   0  4
  [11] .text            PROGBITS        00000000004003b0 0003b0 000181 00  AX  0   0 16
  [12] .fini            PROGBITS        0000000000400534 000534 000009 00  AX  0   0  4
  [13] .rodata          PROGBITS        0000000000400540 000540 000010 00   A  0   0  8
  [14] .eh_frame_hdr    PROGBITS        0000000000400550 000550 000044 00   A  0   0  4
  [15] .eh_frame        PROGBITS        0000000000400598 000598 000118 00   A  0   0  8
  [16] .init_array      INIT_ARRAY      0000000000600e40 000e40 000008 08  WA  0   0  8
  [17] .fini_array      FINI_ARRAY      0000000000600e48 000e48 000008 08  WA  0   0  8
  [18] .dynamic         DYNAMIC         0000000000600e50 000e50 0001a0 10  WA  6   0  8
  [19] .got             PROGBITS        0000000000600ff0 000ff0 000010 08  WA  0   0  8
  [20] .got.plt         PROGBITS        0000000000601000 001000 000018 08  WA  0   0  8
  [21] .data            PROGBITS        0000000000601018 001018 000004 00  WA  0   0  1
  [22] .bss             NOBITS          000000000060101c 00101c 000004 00  WA  0   0  1
  [23] .comment         PROGBITS        0000000000000000 00101c 000058 01  MS  0   0  1
  [24] .symtab          SYMTAB          0000000000000000 001078 0005a0 18     25  41  8
  [25] .strtab          STRTAB          0000000000000000 001618 0001c0 00      0   0  1
  [26] .shstrtab        STRTAB          0000000000000000 0017d8 0000f9 00      0   0  1
Clé des fanions :
  W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes), I (info),
  L (ordre des liens), O (traitement supplémentaire par l'OS requis), G (groupe),
  T (TLS), C (compressé), x (inconnu), o (spécifique à l'OS), E (exclu),
  l (grand), p (processor specific)
```

```
En-têtes de programme :
  Type           Décalage Adr. vir.          Adr.phys.          T.Fich.  T.Mém.   Fan Alignement
  PHDR           0x000040 0x0000000000400040 0x0000000000400040 0x0001f8 0x0001f8 R E 0x8
  INTERP         0x000238 0x0000000000400238 0x0000000000400238 0x00001c 0x00001c R   0x1
      [Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000400000 0x0000000000400000 0x0006b0 0x0006b0 R E 0x200000
  LOAD           0x000e40 0x0000000000600e40 0x0000000000600e40 0x0001dc 0x0001e0 RW  0x200000
  DYNAMIC        0x000e50 0x0000000000600e50 0x0000000000600e50 0x0001a0 0x0001a0 RW  0x8
  NOTE           0x000254 0x0000000000400254 0x0000000000400254 0x000044 0x000044 R   0x4
  GNU_EH_FRAME   0x000550 0x0000000000400550 0x0000000000400550 0x000044 0x000044 R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x000e40 0x0000000000600e40 0x0000000000600e40 0x0001c0 0x0001c0 R   0x1
```

Format ELF

- `.text` : **code exécutable**
- `.data / .bss` : **données globales**
- `.rodata` : **constantes**
- `.tdata/.tbss` : Section de données thread-specific (TLS)
- `.got` : Table globale permettant d'avoir un accès indirect aux symboles globaux
- `.got.plt` : GOT pour fonctions dynamiques
- `.rel[a].*` : Symbole repositionnable, à résoudre avant le début du programme

- `.init` : prologue
- `.fini` : épilogue
- `.dynamic` : données utiles au loader pour charger les bibliothèques dynamiques
- `.dynstr` : Chaîne de noms des symboles globaux
- `.dynsym` : Table des symboles globaux
- `.symtab` : table de symbole
- `.c/dtors` : Stockage des routines `pre-main()`

Layout Mémoire: Relocation

```
cat /proc/self/maps
```

```
7fa293df8000-7fa293df9000 r--p 00000000 00:18 20589119                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293df9000-7fa293e19000 r-xp 00001000 00:18 20589119                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293e19000-7fa293e21000 r--p 00021000 00:18 20589119                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293e22000-7fa293e23000 r--p 00029000 00:18 20589119                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293e23000-7fa293e24000 rw-p 0002a000 00:18 20589119                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa293e24000-7fa293e25000 rw-p 00000000 00:00 0
7ffc71429000-7ffc7144a000 rw-p 00000000 00:00 0                          [stack]
7ffc715f0000-7ffc715f4000 r--p 00000000 00:00 0                          [vvar]
7ffc715f4000-7ffc715f6000 r-xp 00000000 00:00 0                          [vdso]
```

Types d’instrumentation

Instrumentation par le Compilateur

- Compiler le code instrumenté:
  - Ajoute les sondes lors de la compilation
  - Plus rapide que d’autres méthodes
  - Permet une instrumentation sélective
- Nécécite une recompilation:
  - -fsanitize=address
  - -fsanitize=threads
- Outils tels que ASAN présents dans les compilateurs modernes

Instrumentation au Runtime

- Intercepter les appels du code:
  - Appel ptrace: utilisé par GDB
  - Bibliothèques instrumentées:
    - efence: malloc debugger
- Pas de recompilation
- Supporte les threads
- Nécéssite les symboles de débug (compiler avec -g)
- Peut nécéciter un code non optimisé (-O0)

Virtualisation de l’éxecution

- Intercepte l’ensemble des opérations
  - Accès mémoire
  - Allocations
- Dans le cas de valgrind:
  - Exécution très lente (x1000)
  - Pas de support des threads (exécution séquentielle)
- Pas besoin de recompiler

Utilisation de Valgrind

Fonctions de Valgrind

- Détection des Fuites de Mémoire : Identifie les blocs mémoire alloués mais jamais libérés.
- Détection des Accès Mémoire Invalides : Signale les accès à des zones mémoire non allouées ou libérées.
- Détection des Race Condition : Repère les situations où plusieurs threads accèdent à la même ressource simultanément sans synchronisation.
- Profiling : Collecte des données sur l'utilisation de la mémoire et les performances du programme.

Approche de Valgrind

- Instrumentation : Valgrind modifie le binaire du programme à la volée pour insérer son propre code d'analyse.
- Exécution : Le programme instrumenté est exécuté par Valgrind.
- Analyse Dynamique : Valgrind surveille l'exécution du programme, collectant des informations sur les accès mémoire, les allocations et les libérations de mémoire, etc.
- Génération de Rapports : Valgrind produit des rapports détaillés sur les erreurs de mémoire, les fuites de mémoire, etc.

- Points Forts de cette Méthode
  - Précision : Valgrind surveille chaque instruction du programme, offrant une détection précise des erreurs de mémoire.
  - Indépendance de la Plateforme : Valgrind fonctionne sur plusieurs architectures et systèmes d'exploitation grâce à son approche de "virtualisation" de l'exécution du programme.
- Limitations
  - Overhead : L'instrumentation du code peut entraîner une surcharge significative des performances, rendant parfois l'exécution du programme plus lente.
  - Certaines Erreurs Peuvent Échapper à la Détection : Malgré sa puissance, Valgrind peut ne pas détecter certaines erreurs de mémoire, notamment les cas complexes ou les comportements conditionnels.

-

Utilisation Principale

- Sans recompiler mais avec overhead pour débug:
  - Débogage des accès mémoire et des allocations
  - Corruption de stack
  - Fuites mémoires
  - Pas de thread et gros overhead
- Alternative: ASAN (flag GCC/LLVM -fsanitize=address)
  - Doit recompiler
  - Moins d'overhead
  - Support des threads

Usage:

valgrind --tool=memcheck ./a.out

Profilage Simple avec Kcachegrind

Utilisation Principale

- KCachegrind est un outil de profilage graphique pour visualiser les données de profilage générées par des outils comme Valgrind et Callgrind.
- Fonctionnalités de KCachegrind
  - Visualisation Graphique : Affiche les données de profilage sous forme de graphiques interactifs.
  - Analyse des Performances : Permet d'identifier les parties du code qui consomment le plus de temps CPU ou de mémoire.
  - Navigation Facile : Permet de naviguer facilement à travers le code source pour localiser les goulots d'étranglement.
- Tous les inconvénients de Valgrind
  - Lent
  - Pas de threads

Utilisation Principale

Générer les données de profilage avec Valgrind :

$ valgrind --tool=callgrind ./my_program

Ouvrir les données dans KCachegrind :

$ kcachegrind

Lancez KCachegrind et ouvrez le fichier de données généré. Analyser les performances et identifier les opportunités d'optimisation dans le code source.

Utilisation de ASAN

Address SANitizer (ASAN)

- Qu'est-ce que AddressSanitizer (ASAN) ?
  - AddressSanitizer (ASAN) est un outil de détection des erreurs de mémoire développé par Google.
  - Conçu pour détecter les problèmes de mémoire courants tels que les débordements de tampon, les fuites de mémoire, etc.
- Fonctionnalités d'ASAN
  - Détection des Dépassements de Tampon : Identifie les tentatives d'accès à des zones mémoire en dehors des limites allouées.
  - Détection des Fuites de Mémoire : Signale les blocs mémoire alloués qui ne sont jamais libérés.
  - Détection des Accès aux Zones Mémoire Non Initialisées : Repère les accès à des variables non initialisées.

Utilisation

- Étapes pour Utiliser ASAN
  - Intégration dans le Processus de Compilation :
    - Ajouter le drapeau de compilation -fsanitize=address lors de la compilation du programme.
    - Par exemple, avec GCC : gcc -fsanitize=address -o my_program my_program.c
  - Exécution du Programme :
    - Exécuter le programme comme d'habitude, sans aucune modification supplémentaire.
  - Détection des Erreurs de Mémoire :
    - Pendant l'exécution, ASAN surveille l'accès à la mémoire et détecte les erreurs de mémoire telles que les débordements de tampon, les fuites de mémoire, etc.
  - Production de Rapports d'Erreurs :
    - ASAN produit des rapports détaillés sur les erreurs détectées, y compris les emplacements exacts dans le code source.

Utilisation

Compilation avec ASAN :

gcc -fsanitize=address -o my_program my_program.c

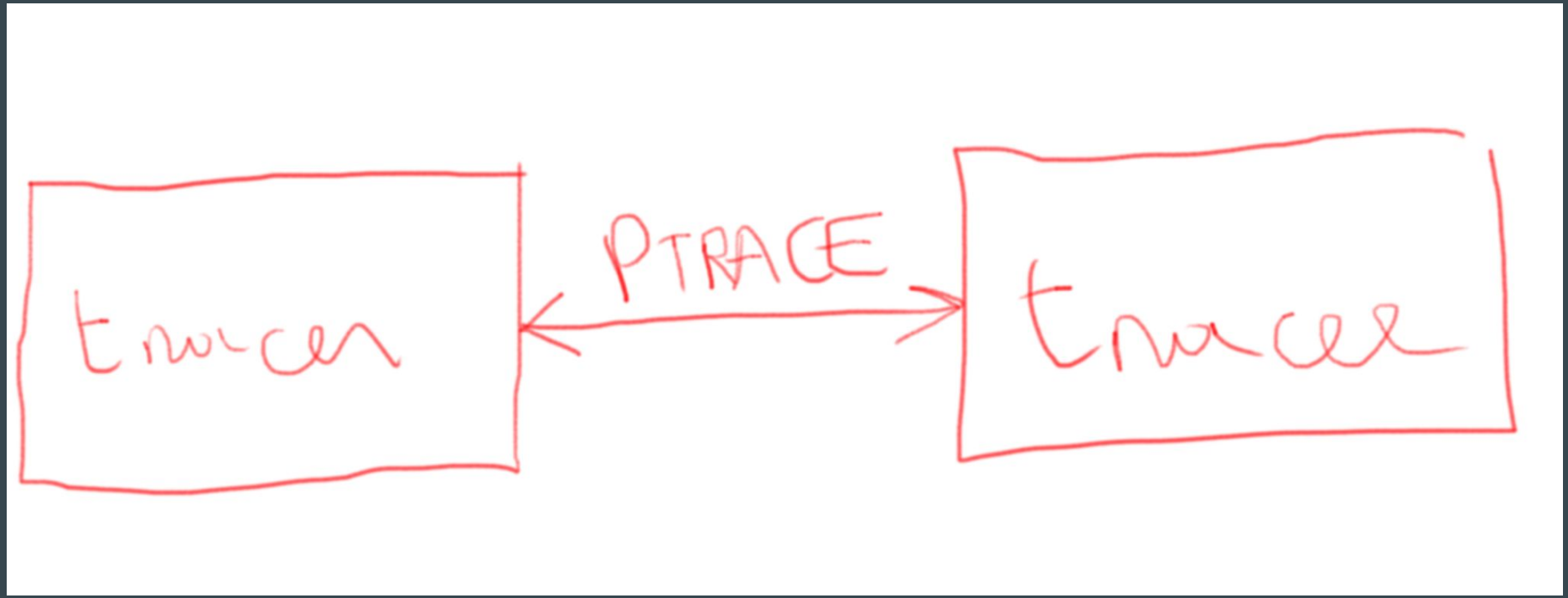
Exécution du Programme :

./my_program

ASAN détecte les erreurs de mémoire pendant l'exécution et produit des rapports d'erreurs.

Débogage: Le Débugger

Appel Système Ptrace



Erreurs et Bogues Communs

- Le débogage est crucial pour détecter et corriger une variété d'erreurs dans les programmes.

Les Deadlocks

- Déboguer les Deadlocks
  - Les deadlocks surviennent lorsque deux ou plusieurs processus ou threads restent bloqués indéfiniment, s'attendant mutuellement à libérer des ressources.
  - Le débogage permet d'identifier les points de synchronisation mal conçus ou mal utilisés.

Le Segmentation Fault

- Déboguer les Segmentation Faults (Segfaults)
  - Les segfaults se produisent lorsqu'un programme tente d'accéder à une zone mémoire non autorisée.
  - Le débogage permet de localiser précisément l'instruction responsable de l'accès mémoire incorrect.

Les Erreurs Logiques

- Déboguer les Erreurs Logiques
  - Les erreurs logiques se produisent lorsque le comportement du programme ne correspond pas à ce qui est attendu, mais sans provoquer de crash ou d'erreur système.
  - Le débogage est essentiel pour repérer et corriger ces erreurs subtiles qui peuvent altérer le bon fonctionnement du programme.

La Corruption Mémoire

- Moins Utile pour la Corruption de Mémoire
  - La corruption de mémoire peut entraîner des comportements imprévisibles et des crashes, mais elle est souvent difficile à détecter et à corriger via le débogage traditionnel.
  - Des outils spécifiques comme Valgrind sont généralement plus adaptés pour détecter la corruption de mémoire.
- En résumé:
  - Le débogage est un outil polyvalent pour identifier et résoudre une gamme variée d'erreurs dans les programmes.
  - Comprendre les différents types d'erreurs et les techniques de débogage appropriées est essentiel pour développer des logiciels robustes et fiables.
-

Appel Système Ptrace

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

Ptrace :

- Fonction système Unix pour le débogage.
- Fonctionnalités :
  - Permet l'observation et le contrôle des processus.
  - Lecture/écriture de mémoire.
  - Manipulation des signaux.
- On peut tracer un de ses fils ou s’attacher à un processus existant

Les Informations de Débug (DWARF)

- Format Dwarf:
  - Utilisé pour le débogage.
- Caractéristiques:
  - Fournit des informations de débogage telles que les noms de variables, les types de données, etc.
  - Peut être stocké dans des fichiers séparés liés aux exécutables (paquets -dbg) et dans l'exécutable (-g)
- Utilisations:
  - Essentiel pour les outils de débogage, tels que gdb.
  - Facilite la localisation et la correction des erreurs dans le code.
- Avantages:
- Réduit la taille des exécutables en séparant les informations de débogage.
- Améliore l'efficacité du débogage en fournissant des informations détaillées sur le code.

Les Informations de Débug (DWARF)

- Utilisation de GDB :
  - Débogueur en ligne de commande pour les programmes C, C++, et autres langages.
- Fonctionnalités principales :
  - Exécution pas à pas du code.
  - Inspection des variables et de la mémoire.
  - Suivi des appels de fonctions.
- Commandes courantes :
  - break : Définit un point d'arrêt.
  - run : Démarre l'exécution du programme.
  - print : Affiche la valeur d'une variable.
- Avantages :
  - Puissant pour le débogage en profondeur.
  - Disponible sur la plupart des systèmes Unix-like.

Lancement avec GDB

Lancement d'un Programme avec GDB

Lancement d'un Programme avec GDB

- Ouvrir un terminal.
- Naviguer vers le répertoire contenant le binaire du programme à déboguer.
- Lancer GDB avec la commande :
  - gdb <nom_du_programme>
  - gdb --args ./a.out args # Cas avec arguments inline
- Pour lancer le programme avec des arguments :
  - run <arguments>

GDB exécutera le programme jusqu'à ce qu'il rencontre un point d'arrêt ou qu'il se termine.

Attach avec GDB

- Qu'est-ce que l'attachement dans GDB ?
  - L'attachement permet à GDB de se connecter à un processus en cours d'exécution.
  - Utile pour déboguer des programmes déjà lancés ou des processus distants.
- Trouver l'identifiant du processus que vous souhaitez attacher en utilisant la commande ps ou pgrep.
- Lancer GDB :
  - gdb
  - Utiliser la commande attach suivi de l'identifiant du processus :
    - attach <PID>
- GDB se connectera au processus en cours d'exécution et vous pourrez commencer le débogage.

Commandes de Base de GDB

- run : Lancer le programme.
- break : Placer un point d'arrêt.
- continue : Reprendre l'exécution après un point d'arrêt.
- step : Exécuter la prochaine instruction, en entrant dans les fonctions appelées.
- next : Exécuter la prochaine instruction, sans entrer dans les fonctions appelées.
- print : Afficher la valeur d'une variable.
- quit : Quitter GDB.

Commandes d’info

- backtrace: affiche un backtrace
- list: affiche le code autours du point courant
- disass: Désassemble le code autours du PC
- info breakpoints : Afficher tous les points d'arrêt définis.
- info locals : Afficher les variables locales dans la fonction courante.
- info args : Afficher les arguments passés à la fonction courante.
- info registers : Afficher les valeurs des registres.
- display <variable> : Afficher continuellement la valeur d'une variable à chaque arrêt.

Commandes de Navigation

- step : Exécuter la prochaine instruction, en entrant dans les fonctions appelées.
- next : Exécuter la prochaine instruction, sans entrer dans les fonctions appelées.
- finish : Exécuter jusqu'à la fin de la fonction courante.
- until : Exécuter jusqu'à la fin de la boucle ou jusqu'à la ligne spécifiée.
- jump <ligne> : Sauter à une ligne spécifiée.

Commande sur Breakpoint

La commande "command" de GDB permet de définir des actions à effectuer lorsqu'un point d'arrêt est rencontré.

Syntaxe :

command breakpoint_num
> commande1
> commande2
> ...
> end

Commande sur Breakpoint

Définir une commande à exécuter au point d'arrêt numéro 1 :

```
command 1
> print "Point d'arrêt atteint!"
> backtrace
> end
```

À chaque fois que le point d'arrêt numéro 1 est atteint, GDB affichera "Point d'arrêt atteint!" suivi d'une trace de la pile (backtrace).