15. Stack: Das EBP-Register

Im Aufgaben-Archiv `sp-ss2013-ue09.tgz` finden Sie die Dateien `15-test-ebp.c` und `15-stack-test.c`.

a) Übersetzen Sie das Programm `15-test-ebp.c` (mit `gcc -m32 15-test-ebp.c` im 32-Bit-Modus) und führen Sie es aus. Vergleichen Sie die Ausgabe mit den Informationen auf den Vorlesungsfolien. Schauen Sie sich die Speicheradressen der Funktionsparameter `x` und `y` sowie der lokalen Variablen `a` und `b` an.

Hinweis: Das Programm liest das Register `ebp` aus. Da C keinen direkten Zugriff auf Registerinhalte erlaubt, muss es ein wenig Inline-Assembler-Code verwenden – das Auslesen erfolgt über das Makro

```
#define get_ebp(v) __asm__ ("mov %%ebp, %0" : "=r"(v))
```

das den Inhalt von Register `ebp` in die Variable `v` kopiert.

b) Erzeugen Sie mit `gcc -m32 -S 15-ebp-test.c` die Assembler-Datei `15-ebp-test.s` und suchen Sie darin den Code der Funktion `f()` und `main()`.

c) Wenn Sie das Programm mit Debugging-Informationen kompilieren, können Sie auch den Debugger gdb verwenden, um sich den Assembler-Code von f() und main() ausgeben zu lassen:

Übersetzen Sie das Programm mit `gcc -m32 -ggdb 15-ebp-test.c` und starten Sie das erzeugte Binary `a.out` über `gdb` (`gdb a.out`). Dann lassen Sie sich mit `disas f` und `disas main` die beiden Funktionen ausgeben (disassemblieren). Über `run` können Sie das Programm im Debugger laufen lassen, mit `quit` verlassen Sie den Debugger.

d) Übersetzen Sie jetzt auch `15-stack-test.c`; führen Sie damit dieselben Tests wie in Aufgaben a) bis c) durch.

16. Threads, Signal-Handler und ihre Stacks

Im Aufgaben-Archiv finden Sie die Datei `16-thread-stacks.c`.

a) Das Programm erzeugt drei Threads, es definiert mit `signal()` einen Signal-Handler für das Signal `SIGALRM` und sorgt mit `alarm()` dafür, dass nach einer Sekunde ein Alarm-Signal an den Prozess geschickt wird. Außerdem führt es vor und nach dem Erzeugen der Threads das Kommando `cat /proc/PID/maps` aus. (Dazu benutzt es die `system()`-Funktion.) In einigen Funktionen gibt es zudem die aktuell gültige Stack-Adresse (über das Register `esp`) und Teile des Stacks aus.

Lesen Sie den Quellcode und vollziehen Sie nach, wie das Programm arbeitet. Es nutzt viele Makros, zu einigen davon finden Sie hier Erklärungen. Mehrzeilige Makro-Definitionen haben an den Zeilenenden immer das Zeilenumbruchzeichen `\`, damit der Parser weiß, dass die Makro-Definition noch in der nächsten Zeile fortgesetzt wird.

`get_esp` und `get_ebp` lesen die Inhalte der Register `esp` und `ebp` aus, wie in Aufgabe 15.

Das Register `eip` enthält den Instruction Pointer (Befehlszähler) – dieser lässt sich nicht mit demselben Trick auslesen, den wir für `esp` und `ebp` genutzt haben. (Der Assembler-Befehl `mov` akzeptiert `eip` nicht als Registernamen.) Darum nutzt `get_eip` einen Trick: Wenn Sie die Funktion `get_eip` aufrufen, liegt die Rücksprungadresse während der Bearbeitung von `get_eip` auf dem Stack und kann dort ausgelesen werden – das macht das Kommando

```
__asm__ __volatile__ ("movl 4(%%ebp), %0" : "=r" (eip));
```

`get_eip` gibt also die Adresse zurück, die den Befehl **hinter** dem `get_eip`-Aufruf enthält.

Das `PEEK`-Makro liest den (als `unsigned int` interpretierten) Inhalt der Speicheradresse `addr` aus. Dazu castet es `addr` zunächst mit `(unsigned *)` in einen Zeiger auf ein `unsigned int` und greift dann mit `*` auf den gespeicherten Inhalt zu.

Das Makro `print_regs(name)` erzeugt über `#name` einen String `"name"`, so dass also z. B. der Aufruf `print_regs(main)` zu `printf ("main" ": esp = …)` expandiert wird. In C darf man mehrere String-Konstanten einfach aneinander hängen; so ist etwa `"ab" "cd" "ef"` eine alternative Schreibweise für `"abcdef"`.

b) Übersetzen Sie das Programm mit `gcc -m32 16-thread-stacks.c -lpthread` (Die `pthread`-Bibliothek müssen Sie explizit mit angeben.) und führen Sie es aus. Ignorieren Sie zunächst die angezeigten Inhalte von `/proc/PID/maps`; Sie können dazu auch die beiden Aufrufe des Makros `show_mem_map` vorübergehend auskommentieren.

Vergleichen Sie die angezeigten Inhalte der Register `esp`, `ebp` und `eip` in den Funktionen `f1`, `f2` und `main`. `f1` läuft einmal (in einem Thread), `f2` läuft zweimal (in zwei weiteren Threads). Sie können daran erkennen, dass alle Threads (auch die beiden Instanzen, die `f2` ausführen) mit separaten Stacks arbeiten, selbst wenn sie (wie bei `f2`) denselben Code ausführen.

c) Betrachten Sie nun zusätzlich die beiden Ausgaben von `/proc/PID/maps`: Diese zeigen die Speichernutzung des Prozesses vor und nach dem Erzeugen der Threads an. (Falls Sie in Aufgabe b) die Aufrufe von `show_mem_map` auskommentiert haben, müssen Sie diese zunächst wieder aktivieren und das Programm neu übersetzen.) Die tabellerische Ausgabe enthält folgende Spalten:

```
address           perms offset  dev   inode   pathname
08048000-08056000 r-xp 00000000 03:0c 64593   /usr/sbin/gpm
bfb6e000-bfb8f000 rw-p 00000000 00:00 0       [stack]
```

– `address` gibt den Adressbereich an, der in dieser Zeile beschrieben wird, die letzte Adresse ist jeweils exklusive.

– `perms` gibt die Zugriffsrechte an (`rwx` = lesen, schreiben, ausführen).

– wenn der Speicher Inhalte einer Datei enthält, gibt `offset` an, ab welcher Position der Dateieinhalt hier eingeblendet ist; dann steht in der Spalte `pathname` ein Dateiname, und es handelt sich um ein `mmap`-Mapping dieser Datei in den Speicher.

– `dev` und `inode` identifizieren bei `mmap`-Mappings das Dateisystem und darauf den Inode, also die Datei.

Die zweite Ausgabe der Memory Map ist umfangreicher als die der ersten, weil für die drei erzeugten Threads zusätzliche Speicherbereiche genutzt werden. Identifizieren Sie die hinzugekommenen Speicherbereiche und vergleichen Sie diese mit den von den Threads ausgegebenen Registerinhalten (`esp`, `ebp`): Wo liegen die Stacks der Threads? Welchen Stack verwendet der Signal-Handler?

d) Übersetzen Sie das Programm mit der Debug-Option `-g`:

```
gcc -m32 -g 16-thread-stacks.c -lpthread
```

Dann können Sie mit `objdump -S a.out` ein Assembler-Listing erzeugen, das neben den Assembler auch die jeweiligen C-Befehle aus der Quelldatei enthält.

Suchen Sie die verschiedenen Funktionen aus dem C-Programm; diese Übersicht wird später nützlich sein, wenn Sie bei einem Programm Speicheradressen identifizieren wollen (also: wenn Sie herausfinden wollen, zu welchem Stück Code eine Adresse gehört).

17. Mehr zu Funktionsaufrufen

Schauen Sie sich in der PDF-Datei `Stack-Erklaerung/u8-a4.pdf` im Aufgabenarchiv die Folien 8.2–8.15 an. Sie beschreiben detailliert, wie sich der Stack beim Aufruf von Funktionen verändert.