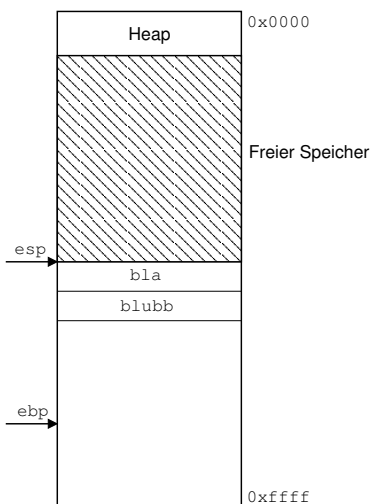


Funktionsweise von Funktionsaufrufen

- 1) Der Stack ganz „normal“. Ganz oben (die kleinen Adressen) ist das Datensegment (hier nicht extra aufgeführt) und der Heap. Unten ist der Stack. Er wächst nach oben (der esp, Stackpointer, wird also *dekrementiert*, wenn der Stack größer wird). Als letzte zwei Assemblerbefehle wurde folgendes (in *der* Reihenfolge) ausgeführt:

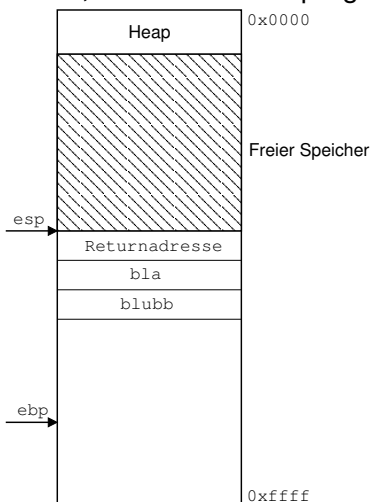
```
pushl blubb  
pushl bla
```



- 2) Nun kommt ein „call“ Assemblerbefehl:

```
call UnterProgramm
```

Dies veranlasst zunächst einmal den Prozessor, die momentane Programmadresse auf den Stack zu „pushl“en, damit das Unterprogramm später wieder zurückfindet.



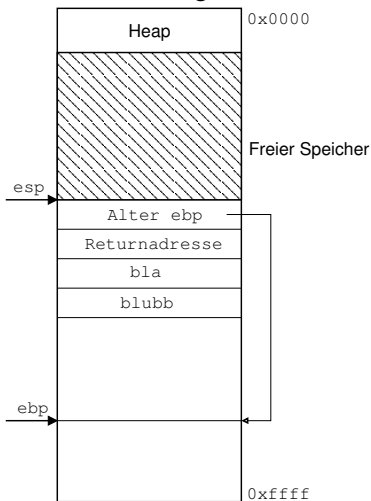
- 3) Wenn die Rücksprungadresse auf dem Stack ist, wird in das Unterprogramm gegangen. Man kann sich also den kompletten „call“ Befehl so vorstellen:

```
pushl RuecksprungAdresse  
jmp UnterProgramm
```

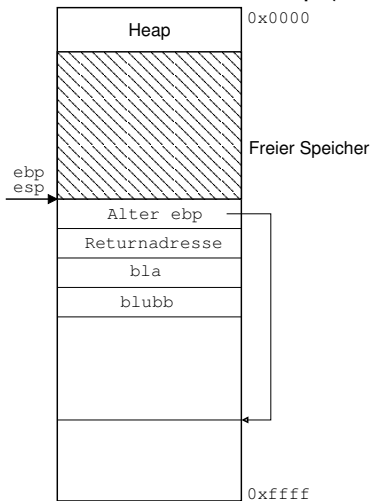
Sobald der „jmp“ ins Unterprogramm vonstatten gegangen ist, werden normalerweise immer diese Befehle ausgeführt (das ist wirklich *sehr* typisch für ein Unterprogramm, ein sogenannter “Stackframe“ wird erzeugt):

```
pushl %ebp  
movl %esp, %ebp
```

So sieht die Lage dann aus, nachdem „%ebp“ auf den Stack gepusht wurde:

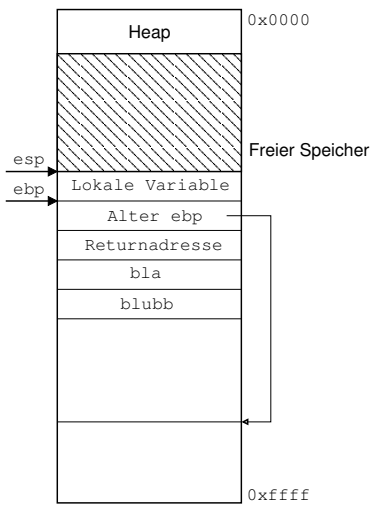


4) Nun wird der neue %ebp (der Base Pointer) auf den momentanen Stackpointer, %esp, gesetzt:



5) Jetzt sind wir im Unterprogramm! Hier können wir am Stack herumpfuschen, wie wir wollen, solange wir nicht mehr „popl“en, als „pushl“en wird nichts passieren. Sozusagen der Sandkasten für Unterprogramme. Das benutzen wir doch gleich, um eine lokale Variable auf den Stack zu schieben:

```
pushl LokaleVariable
```



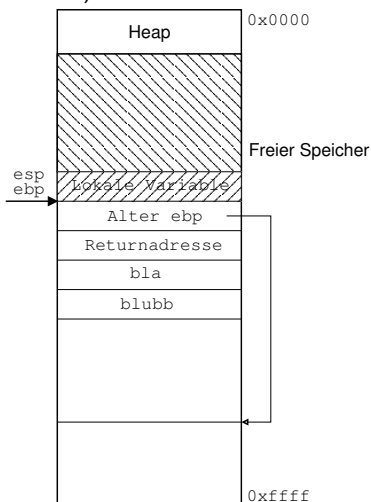
6) Irgendwann haben wir aber genug gemacht. Das Unterprogramm soll verlassen werden. Dazu dient die Kombination der Assemblerbefehle

```
leave  
ret
```

Der „leave“ Befehl führt zunächst einmal

```
movl %ebp, %esp
```

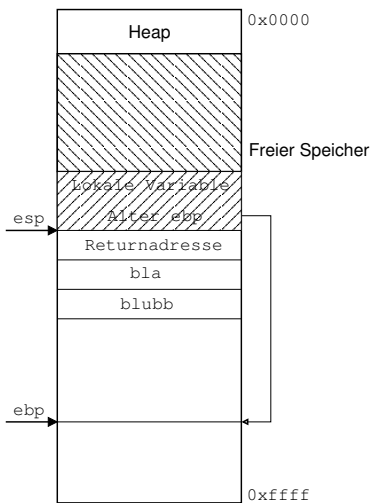
aus, und setzt damit also den Stack wieder auf den Anfang zurück (egal wie weit nach hinten wir ge-„push“ed hatten):



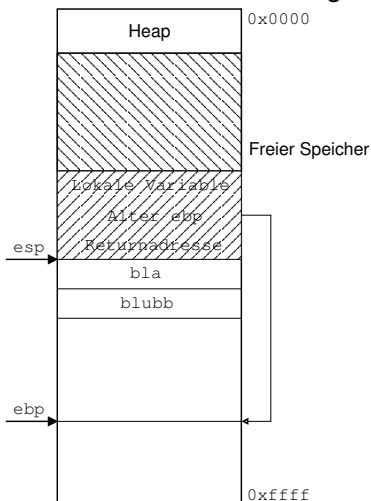
7) Dann führt - immernoch der „leave“ Befehl - folgendes aus:

```
popl %ebp
```

Damit wird also der alte Base-Pointer (den wir uns vorher gesichert hatten) restauriert:



- 8) Letztendlich wird dann noch der „ret“ Befehl abgearbeitet: Er „pop“ed die noch die Programmadresse, die der „call“ Befehl auf den Stack geschoben hatte, und springt zur alten Programmadresse (eigentlich einen Befehl nach der alten Programmadresse) zurück:



Bemerkung: Nur wenn man die Vorstellung davon hat, wie der Stack aufgebaut ist und was bei einem Funktionsaufruf („call“) passiert, kann man sich erklären, welche Position die einer Funktion übergebenen Parameter für Adressen haben: so liegen die übergebenen Parametervariablen „bla“ und „blubb“ in Diagrammen 4 und 5 (also *im* Unterprogramm) an folgenden Adressen:

```
bla    = 8(%esp)
blubb = 12(%esp)
```

Noch eine Bemerkung: Die Adresse, die ein „call“ als „Rücksprungadresse“ speichert, ist immer die, des nächsten Befehls nach dem „call“. Beispiel:

```
unterprogramm:
    ...
    ret

main:
    ...
moo:  call unterprogramm
kuh:  movl $9, %eax
```

Themengebiet: Funktionsaufrufe auf dem x86 - Der Stack (Version 3)
Seite: 5
Copyright: Johannes Bauer (www.johannes-bauer.com)

...
ret

In diesem Beispiel wird also bei dem „call unterprogramm“ die Adresse, an der das „movl \$9, %eax“ steht auf den Stack ge-„push“-ed (also die Adresse „kuh“, nicht „moo“!).

Ich hoffe, das trägt ein bisschen zum Verständnis bei. Bei Fragen bin ich zu erreichen unter JohannesBauer@gmx.de.