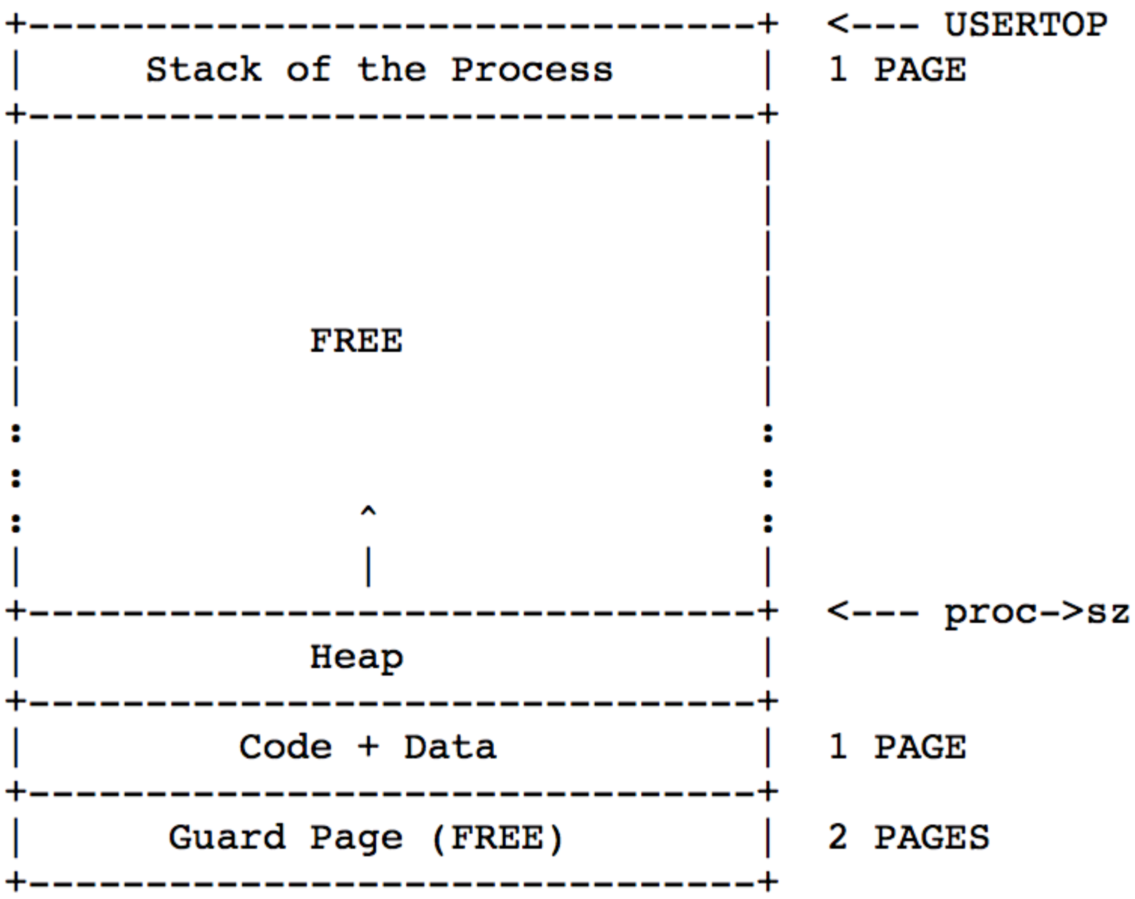
Kernel Threads

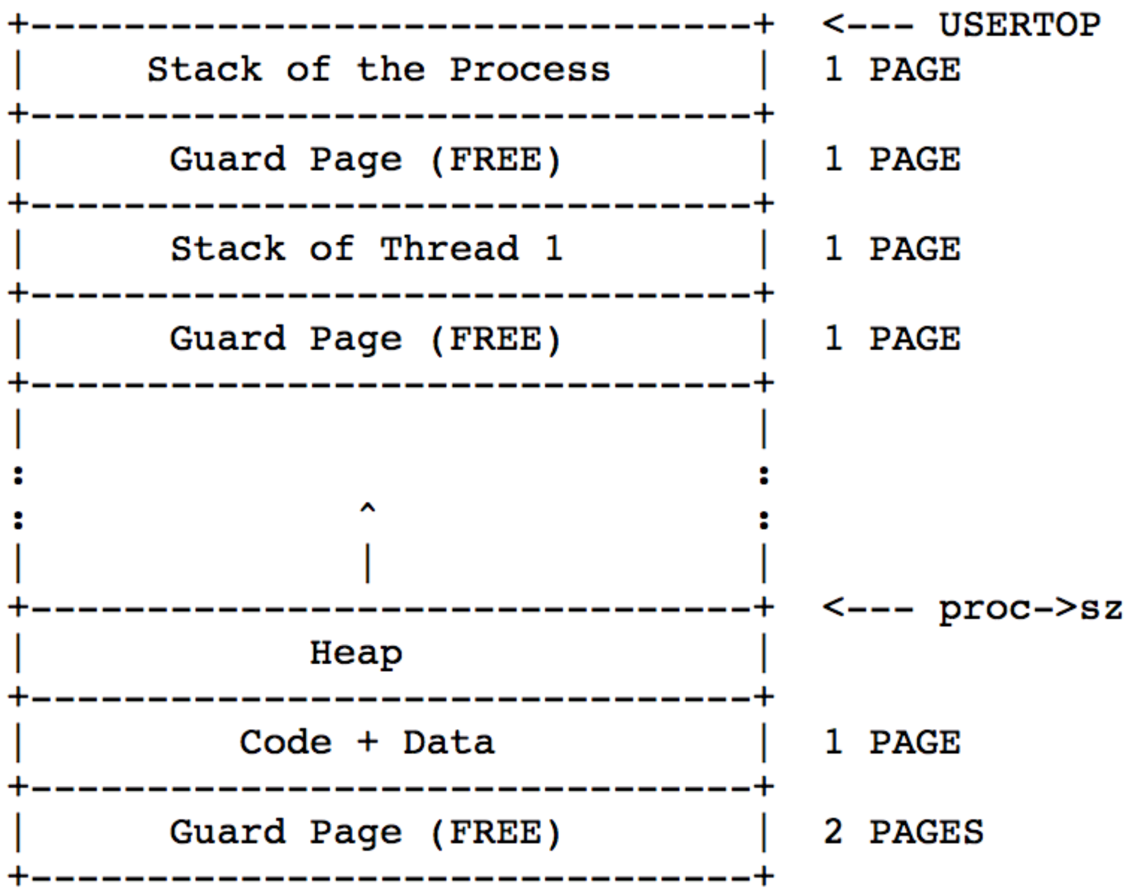
Zhewen Song

07/29/17

VM layout



```
+--------------------------------+  <--- USERTOP
|      Stack of the Process      |  1 PAGE
+--------------------------------+
|                                |
|                                |
|                                |
|                                |
|             FREE               |
|                                |
|                                |
:                                :
:                                :
:              ^                 :
|              |                 |
+--------------------------------+  <--- proc->sz
|             Heap               |
+--------------------------------+
|          Code + Data           |  1 PAGE
+--------------------------------+
|       Guard Page (FREE)        |  2 PAGES
+--------------------------------+
```



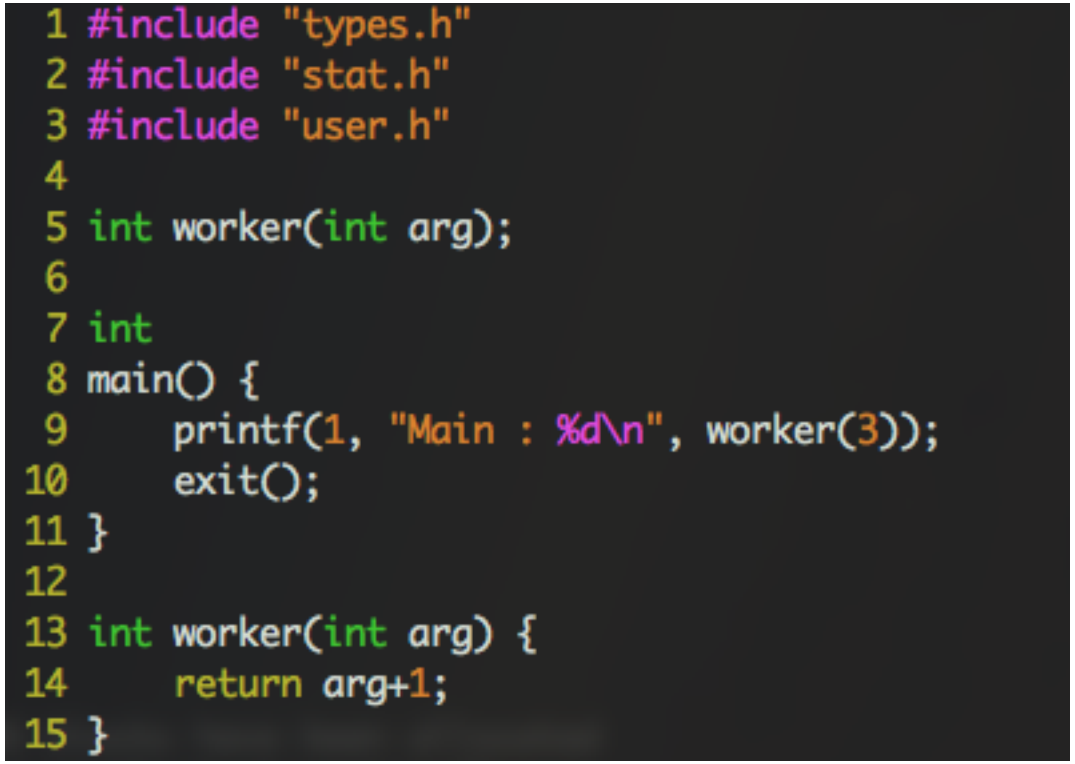
```
+--------------------------------+  <--- USERTOP
|      Stack of the Process      |  1 PAGE
+--------------------------------+
|       Guard Page (FREE)        |  1 PAGE
+--------------------------------+
|       Stack of Thread 1        |  1 PAGE
+--------------------------------+
|       Guard Page (FREE)        |  1 PAGE
+--------------------------------+
|                                |
:                                :
:              ^                 :
|              |                 |
+--------------------------------+  <--- proc->sz
|             Heap               |
+--------------------------------+
|          Code + Data           |  1 PAGE
+--------------------------------+
|       Guard Page (FREE)        |  2 PAGES
+--------------------------------+
```

Threads in xv6

- You may treat thread as a process which shares the code, data, heap, file descriptors, etc. with the parent process. They are actually the same from scheduler's perspective, i.e. the thread is also placed into `ptable`, just as normal process.
- When we call `clone(void(*fcn)(void *), void *arg)` at user space, a thread is created, and the thread will execute the routine `fcn(void *arg)`. But how does the thread know which routine to run?

Demo1 : X86 calling conventions

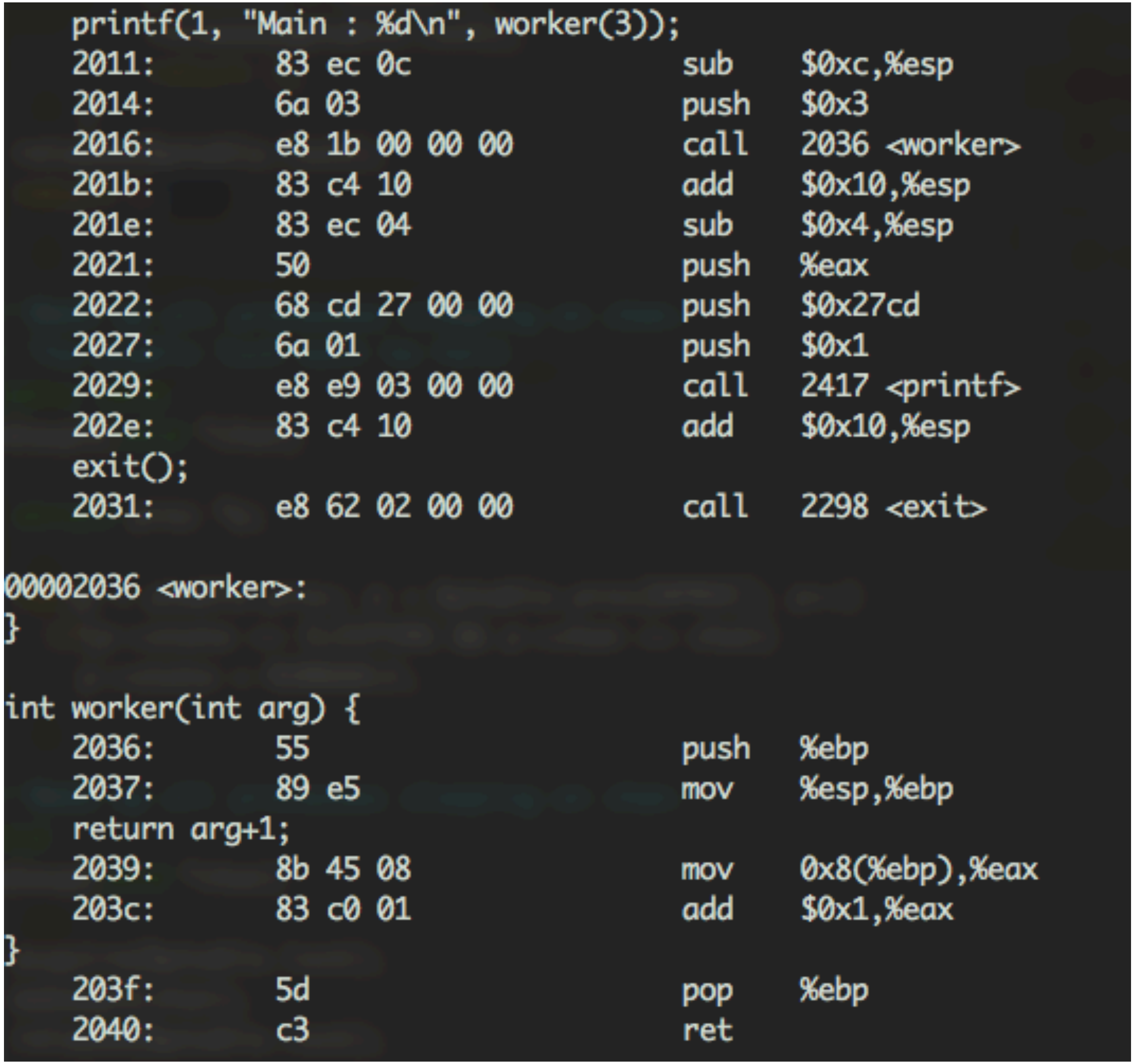
- Easy piece of code
- But do you really understand what happens under the hood?



```c
#include "types.h"
#include "stat.h"
#include "user.h"

int worker(int arg);

int
main() {
    printf(1, "Main : %d\n", worker(3));
    exit();
}

int worker(int arg) {
    return arg+1;
}
```

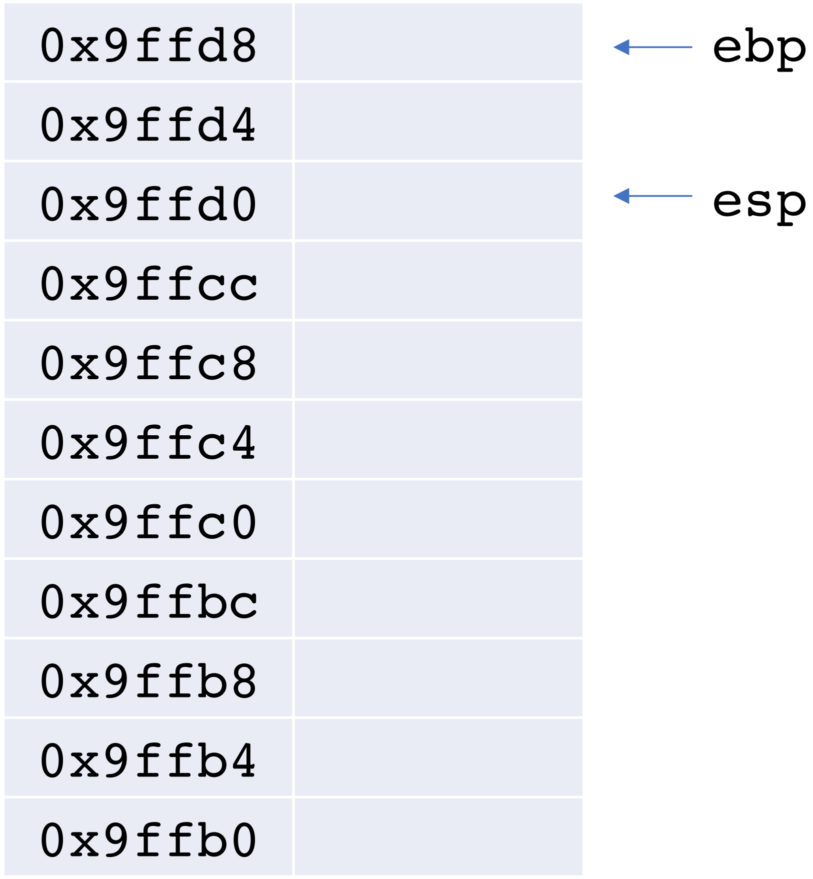
Useful tools

- `objdump` dumps the assembly
- `gdb` can step into assembly line by line
  - How to debug a user program?
    - `add-symbol-file fs/demo 0x2000`
  - How to get the register info?
    - `info register esp`
    - `i r esp` (for short)



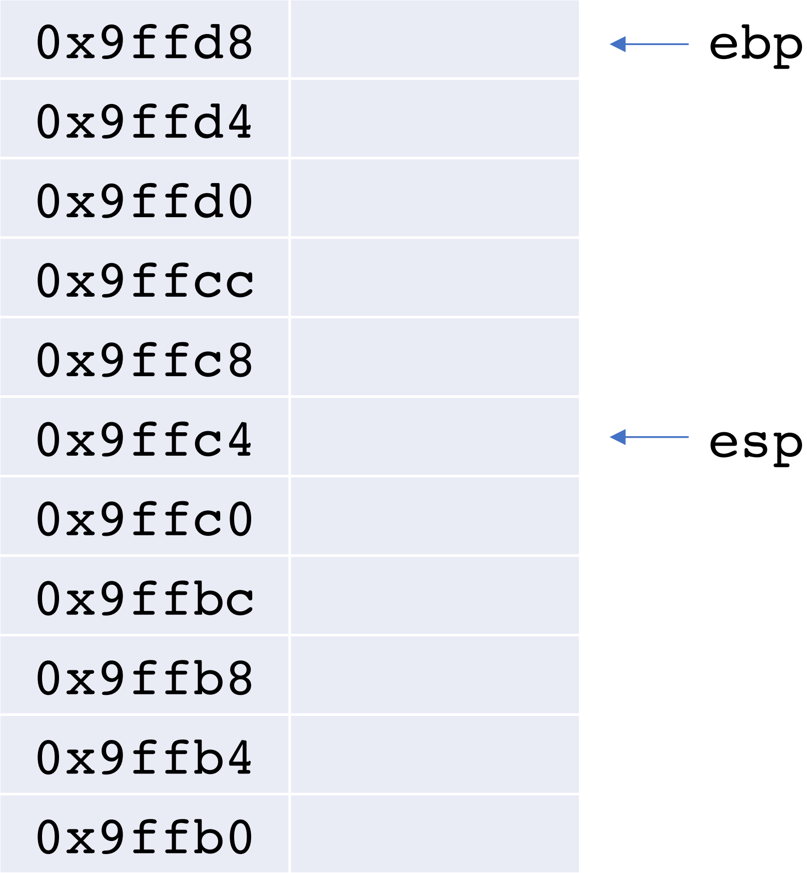
```
    printf(1, "Main : %d\n", worker(3));
    2011:       83 ec 0c                sub    $0xc,%esp
    2014:       6a 03                   push   $0x3
    2016:       e8 1b 00 00 00          call   2036 <worker>
    201b:       83 c4 10                add    $0x10,%esp
    201e:       83 ec 04                sub    $0x4,%esp
    2021:       50                      push   %eax
    2022:       68 cd 27 00 00          push   $0x27cd
    2027:       6a 01                   push   $0x1
    2029:       e8 e9 03 00 00          call   2417 <printf>
    202e:       83 c4 10                add    $0x10,%esp
    exit();
    2031:       e8 62 02 00 00          call   2298 <exit>

00002036 <worker>:
}

int worker(int arg) {
    2036:       55                      push   %ebp
    2037:       89 e5                   mov    %esp,%ebp
    return arg+1;
    2039:       8b 45 08                mov    0x8(%ebp),%eax
    203c:       83 c0 01                add    $0x1,%eax
}
    203f:       5d                      pop    %ebp
    2040:       c3                      ret
```

```
printf(1, "Main : %d\n", worker(3));
```



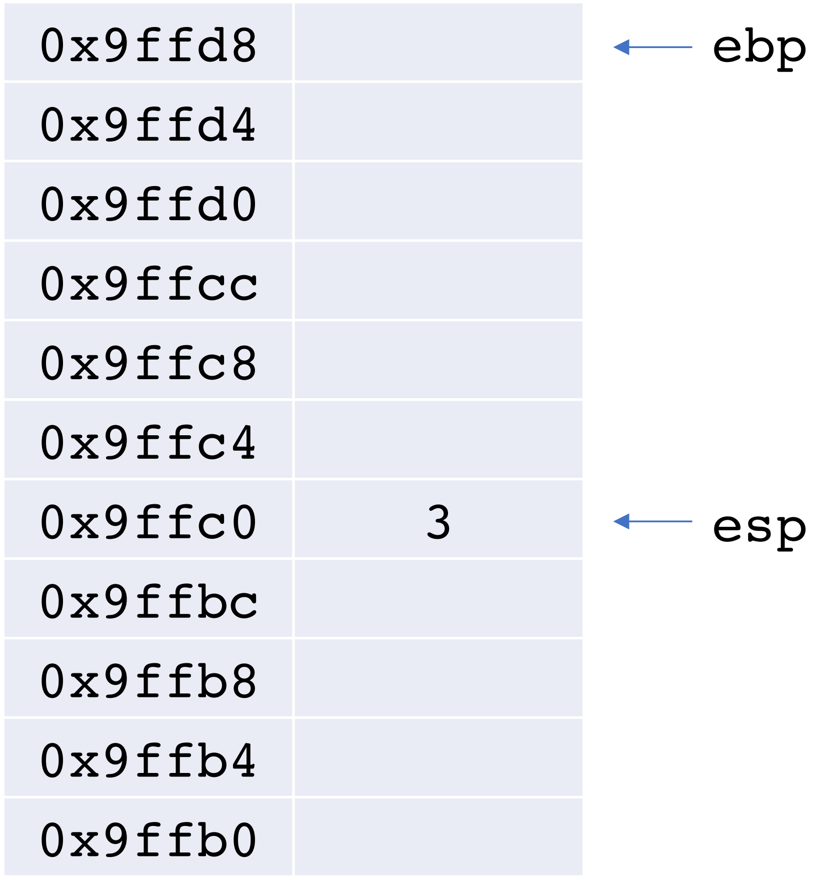
0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		← esp
0x9ffcc		
0x9ffc8		
0x9ffc4		
0x9ffc0		
0x9ffbc		
0x9ffb8		
0x9ffb4		
0x9ffb0		

```
sub     $0xc,%esp
```



0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8		
0x9ffc4		← esp
0x9ffc0		
0x9ffbc		
0x9ffb8		
0x9ffb4		
0x9ffb0		

push $0x3



0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8		
0x9ffc4		
0x9ffc0	3	← esp
0x9ffbc		
0x9ffb8		
0x9ffb4		
0x9ffb0		

`call   2036 <worker>`

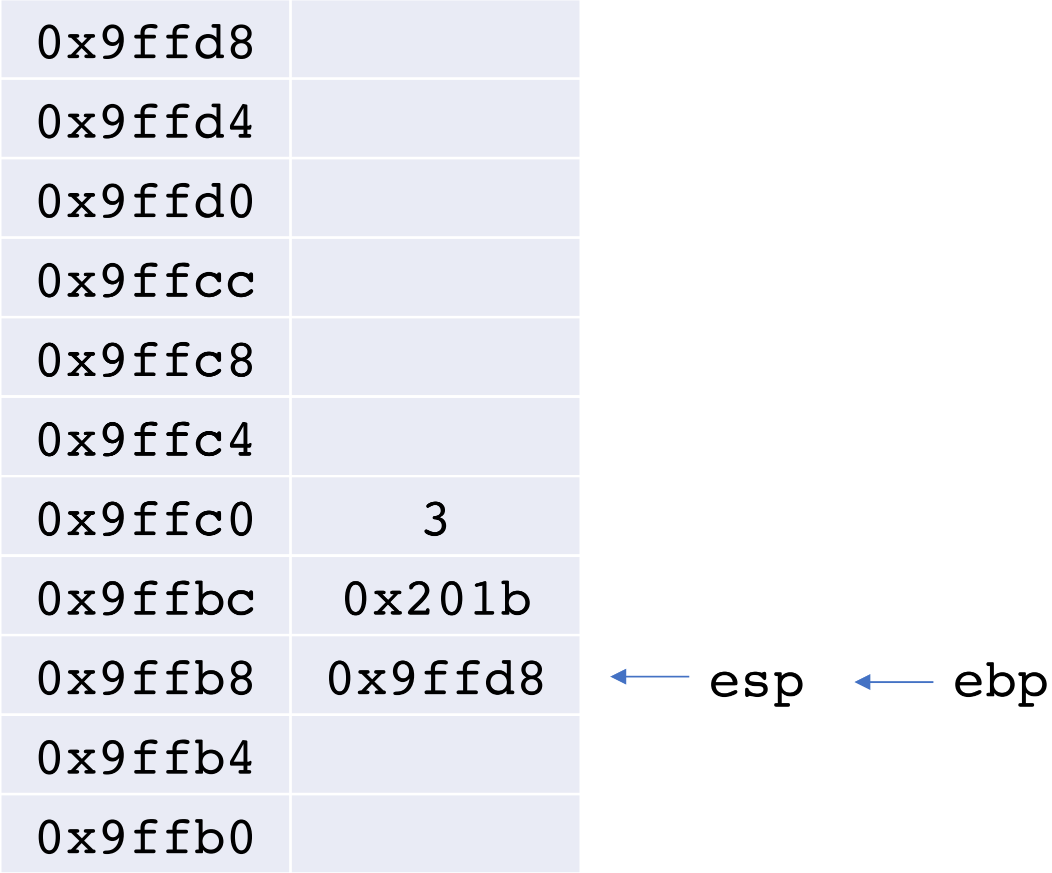
0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8		
0x9ffc4		
0x9ffc0	3	
0x9ffbc	0x201b	← esp
0x9ffb8		
0x9ffb4		
0x9ffb0		

Where does this addr come from?

push %ebp

0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8		
0x9ffc4		
0x9ffc0	3	
0x9ffbc	0x201b	
0x9ffb8	0x9ffd8	← esp
0x9ffb4		
0x9ffb0		

mov %esp,%ebp



0x9ffd8	
0x9ffd4	
0x9ffd0	
0x9ffcc	
0x9ffc8	
0x9ffc4	
0x9ffc0	3
0x9ffbc	0x201b
0x9ffb8	0x9ffd8
0x9ffb4	
0x9ffb0	

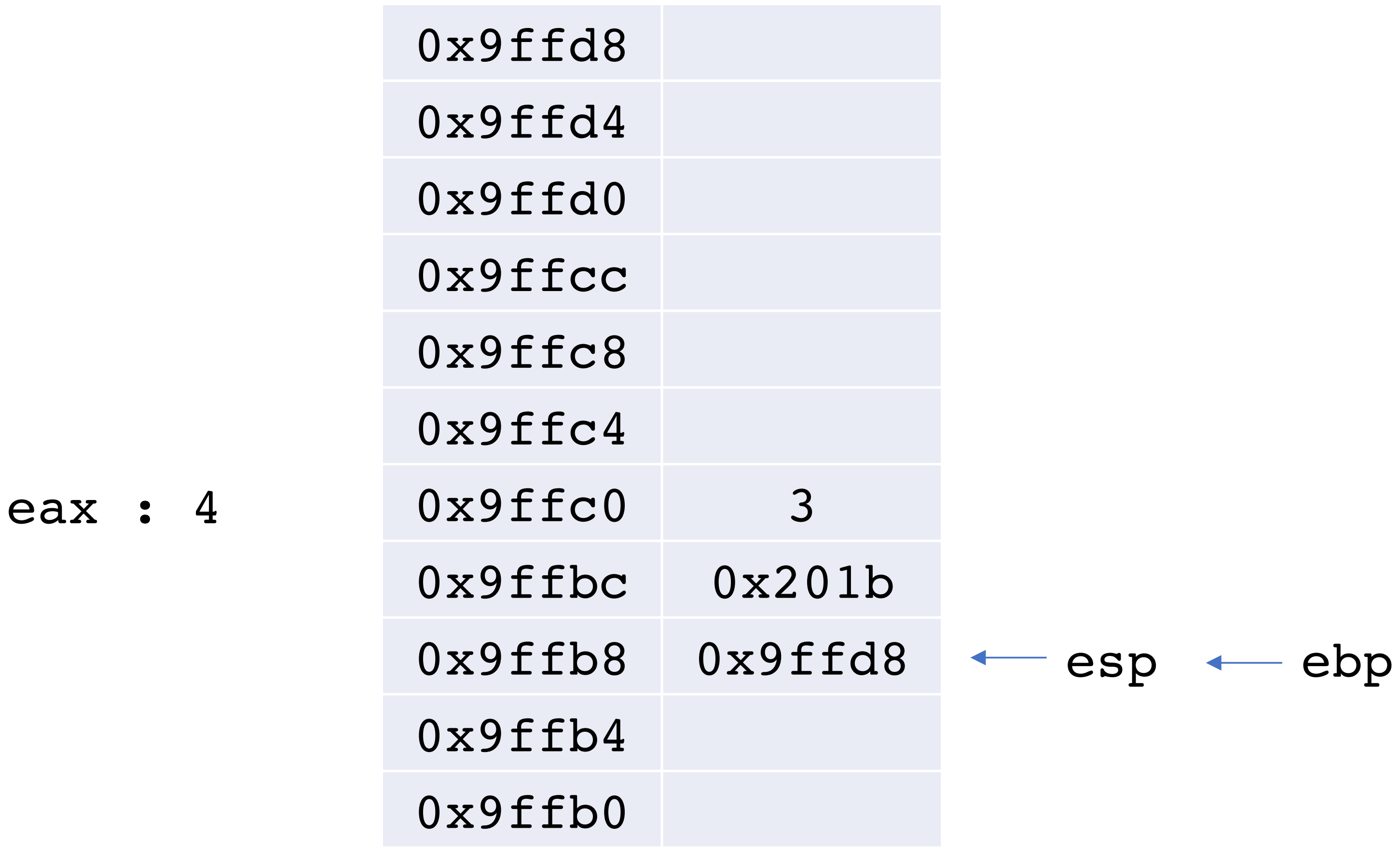
← esp ← ebp

```
mov     0x8(%ebp),%eax
```

eax : 3

0x9ffd8		
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8		
0x9ffc4		
0x9ffc0	3	
0x9ffbc	0x201b	
0x9ffb8	0x9ffd8	← esp ← ebp
0x9ffb4		
0x9ffb0		

```
add     $0x1,%eax
```



eax : 4

0x9ffd8	
0x9ffd4	
0x9ffd0	
0x9ffcc	
0x9ffc8	
0x9ffc4	
0x9ffc0	3
0x9ffbc	0x201b
0x9ffb8	0x9ffd8
0x9ffb4	
0x9ffb0	

pop %ebp

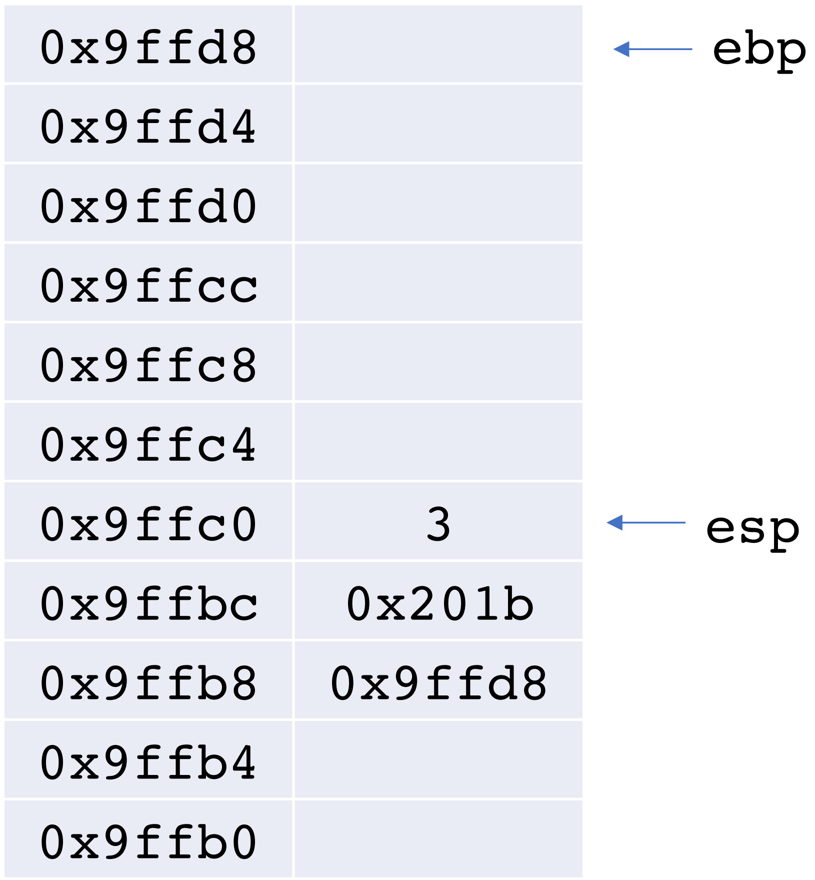
How does ebp know it should go back where it was?

eax : 4

0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8		
0x9ffc4		
0x9ffc0	3	
0x9ffbc	0x201b	← esp
0x9ffb8	0x9ffd8	
0x9ffb4		
0x9ffb0		

ret

eax : 4



0x9ffd8		ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8		
0x9ffc4		
0x9ffc0	3	esp
0x9ffbc	0x201b	
0x9ffb8	0x9ffd8	
0x9ffb4		
0x9ffb0		

Where are we when we return? Where is the return value stored?

add $0x10,%esp

0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		← esp
0x9ffcc		
0x9ffc8		
0x9ffc4		
0x9ffc0	3	
0x9ffbc	0x201b	
0x9ffb8	0x9ffd8	
0x9ffb4		
0x9ffb0		

eax : 4

sub $0x4,%esp

eax : 4

0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		← esp
0x9ffc8		
0x9ffc4		
0x9ffc0	3	
0x9ffbc	0x201b	
0x9ffb8	0x9ffd8	
0x9ffb4		
0x9ffb0		

push %eax

eax : 4

0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8	4	← esp
0x9ffc4		
0x9ffc0	3	
0x9ffbc	0x201b	
0x9ffb8	0x9ffd8	
0x9ffb4		
0x9ffb0		

push $0x27cd

eax : 4

0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8	4	
0x9ffc4	0x27cd	← esp
0x9ffc0	3	
0x9ffbc	0x201b	
0x9ffb8	0x9ffd8	
0x9ffb4		
0x9ffb0		

push $0x1

0x9ffd8		← ebp	
0x9ffd4			
0x9ffd0			
0x9ffcc			
0x9ffc8	4		
0x9ffc4	0x27cd		
0x9ffc0	1	← esp	Oops, overwritten! Is it fine?
0x9ffbc	0x201b		
0x9ffb8	0x9ffd8		
0x9ffb4			
0x9ffb0			

```
eax : 4
```

call 2417 <printf>

eax : 4

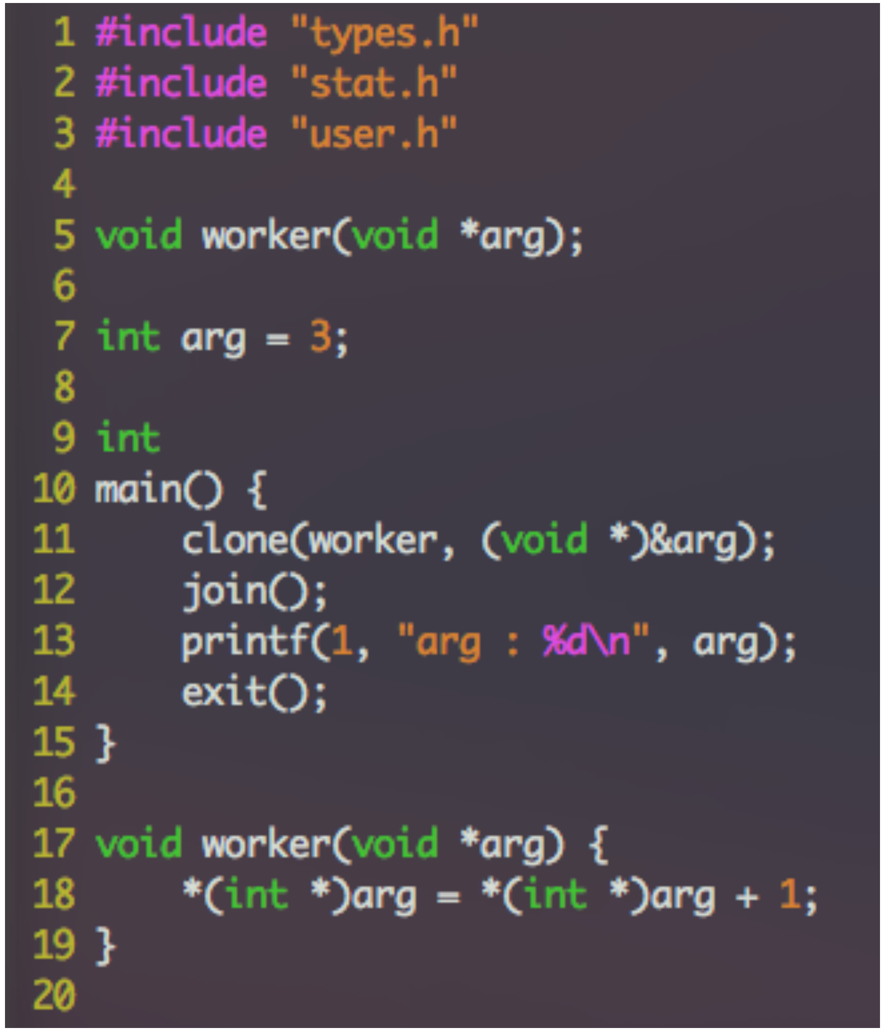
0x9ffd8		← ebp
0x9ffd4		
0x9ffd0		
0x9ffcc		
0x9ffc8	4	
0x9ffc4	0x27cd	
0x9ffc0	1	
0x9ffbc	0x202e	← esp
0x9ffb8	0x9ffd8	
0x9ffb4		
0x9ffb0		

Our old friend again!

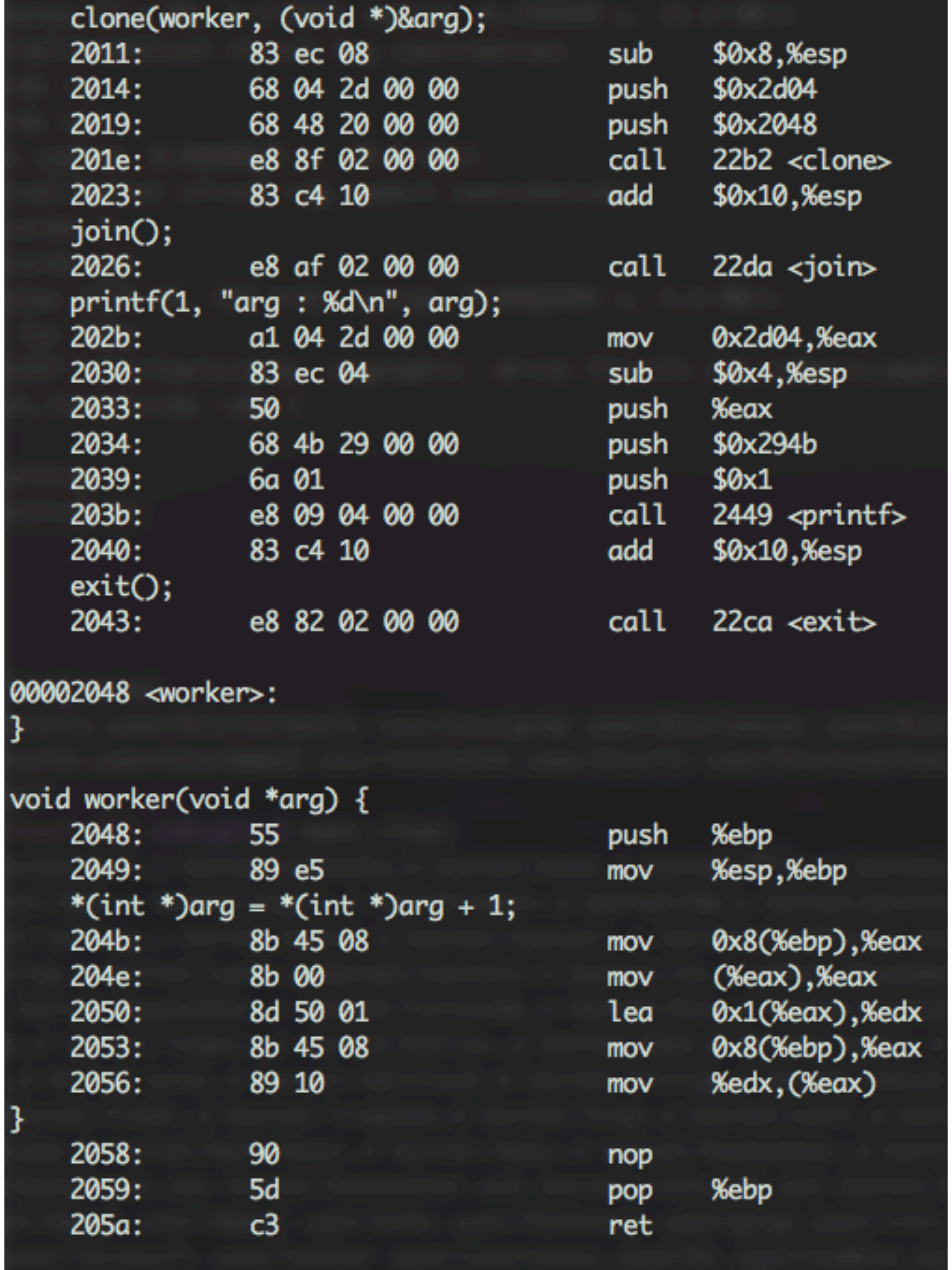
What we know:

- The arguments of the function are pushed into call stack in the reverse order.
- The return address, i.e. the address of the next instruction after return is pushed into call stack at last. Thus, we will always get this address when call stack pops, i.e. when function returns.
- In the function, the arguments can be retrieved from “pointer arithmetic” of `%ebp`, like `0x8(%ebp)`.
- The return value of a function gets stored in the general purpose register, e.g. `%eax`.
  - Some reminiscence: how does `fork()`return 2 values?

Demo2



```c
#include "types.h"
#include "stat.h"
#include "user.h"

void worker(void *arg);

int arg = 3;

int
main() {
    clone(worker, (void *)&arg);
    join();
    printf(1, "arg : %d\n", arg);
    exit();
}

void worker(void *arg) {
    *(int *)arg = *(int *)arg + 1;
}

```



```
    clone(worker, (void *)&arg);
    2011:       83 ec 08                sub    $0x8,%esp
    2014:       68 04 2d 00 00          push   $0x2d04
    2019:       68 48 20 00 00          push   $0x2048
    201e:       e8 8f 02 00 00          call   22b2 <clone>
    2023:       83 c4 10                add    $0x10,%esp
    join();
    2026:       e8 af 02 00 00          call   22da <join>
    printf(1, "arg : %d\n", arg);
    202b:       a1 04 2d 00 00          mov    0x2d04,%eax
    2030:       83 ec 04                sub    $0x4,%esp
    2033:       50                      push   %eax
    2034:       68 4b 29 00 00          push   $0x294b
    2039:       6a 01                   push   $0x1
    203b:       e8 09 04 00 00          call   2449 <printf>
    2040:       83 c4 10                add    $0x10,%esp
    exit();
    2043:       e8 82 02 00 00          call   22ca <exit>

00002048 <worker>:
}

void worker(void *arg) {
    2048:       55                      push   %ebp
    2049:       89 e5                   mov    %esp,%ebp
    *(int *)arg = *(int *)arg + 1;
    204b:       8b 45 08                mov    0x8(%ebp),%eax
    204e:       8b 00                   mov    (%eax),%eax
    2050:       8d 50 01                lea    0x1(%eax),%edx
    2053:       8b 45 08                mov    0x8(%ebp),%eax
    2056:       89 10                   mov    %edx,(%eax)
}
    2058:       90                      nop
    2059:       5d                      pop    %ebp
    205a:       c3                      ret
```

Some general guidance

- For `clone()`, it’s just analogous `fork()` and `exec()`combined
  - The `fcn` routine never returns. It’s just like `exec()`. That’s why we need a fake return address, i.e. `0xffffffff`, which is of course an invalid virtual address. OS will trap and thus kill this thread.
- For `join()`, it’s just analogous to `wait()`
- If you can understand the difference between process and thread, then you can easily figure out what to modify based on the given codes in xv6.

About condition variables

- cond_wait(cond_t *cv, lock_t *lk)
  - Assert lk is required
  - Put the caller to sleep on specific channel!
  - Release lk
  - Wait for lk to be released again
  - Reacquire lk

- sleep(void *chan, struct spinlock *lk)

About condition variables

- `cond_signal(cond_t *cv)`
  - Wake up the matching thread
- `wakeup(void *chan)`