Project 3 Walkthrough

Before that...

- reminder that optional Project 2 is due Friday 2 Nov.
- auto grading Project 1 prepared you for Project 2
- you need to develop your own test cases
- “it worked on my machine”

Project 3

work in teams of 2

You are responsible:

- form a team by finding someone you work well with.
- resolve any problems that arise during the partnership.

You must follow a pair programming methodology

http://www.extremeprogramming.org/rules/pair.html

You must follow a pair programming methodology

http://www.extremeprogramming.org/rules/pair.html

- sit side by side in front of the monitor sliding keyboard and mouse back and forth.
- while one person types the other observes, detects tactical coding errors, etc.
- roles swapped frequently.

Pair programming research shows increase software quality without impacting time to deliver

Difficulty

Project 1: moderately easy

Project 3: moderately hard

Likely to be the most challenging programming you have ever done.

Strategies to make it easier

Strategies to make it easier

The project's main difficulty is conceptualizing the solution. Once you overcome that hurdle, you will be surprised at how relatively simple the implementation is.

Brute force

A brute force all nighter (or several all nighters) has a low chance of success.

User level thread library

using this library:

- create threads
- destroy them
- allow threads to control scheduling

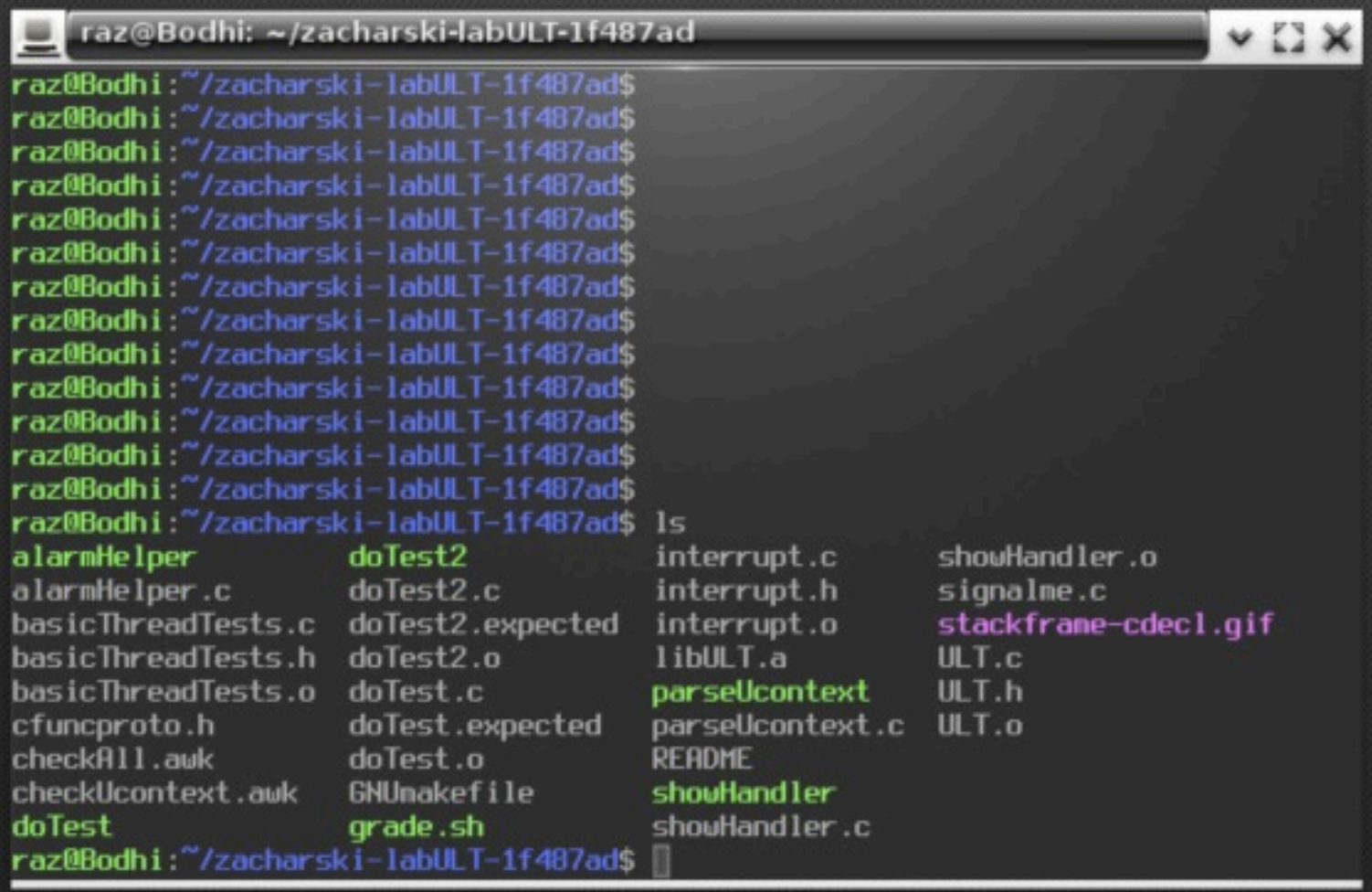
```
main(int argc, char ** argv)
{
    // Some initialization
    // Create threads
    // wait for threads to finish
    // exit
}

// "Main" procedure for thread i
root_i (...)
{
    // do some work
    // yield
    // repeat as necessary
    // return (implicit thread destruction)
}
```

where “root_i” is a “root function” that the ith thread calls to start executing.

Pre-existing code

2,000 lines of code

git repository (see project write-up)


raz@Bodhi: ~/zacharski-labULT-1f487ad
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$
raz@Bodhi:~/zacharski-labULT-1f487ad$ ls
alarmHelper
alarmHelper.c
basicThreadTests.c
basicThreadTests.h
basicThreadTests.o
cfuncproto.h
checkAll.awk
checkUcontext.awk
doTest
doTest2
doTest2.c
doTest2.expected
doTest2.o
doTest.c
doTest.expected
doTest.o
GNUmakefile
grade.sh
interrupt.c
interrupt.h
interrupt.o
libULT.a
parseUcontext
parseUcontext.c
README
showHandler
showHandler.c
showHandler.o
signalme.c
stackframe-cdecl.gif
ULT.c
ULT.h
ULT.o
raz@Bodhi:~/zacharski-labULT-1f487ad$

Thread Context

Program counter, registers, local variables, stack, etc.

Program Context

- need to save and restore the context from the processor when switching threads.
- you will use two existing library calls:
  - getcontext
  - setcontext
- project writeup has link to man page

setcontext(2) - Linux man page

Name

getcontext, setcontext - get or set the user context

Synopsis

#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);

Description

In a System V-like environment, one has the two types *mcontext_t* and *ucontext_t* defined in <*ucontext.h*> and the four functions **getcontext**(), **setcontext**(), **makecontext**(3) and **swapcontext**(3) that allow user-level context switching between multiple threads of control within a process.

The *mcontext_t* type is machine-dependent and opaque. The *ucontext_t* type is a structure that has at least the following fields:

```
typedef struct ucontext {
    struct ucontext *uc_link;
    sigset_t         uc_sigmask;
    stack_t          uc_stack;
    mcontext_t       uc_mcontext;
    ...
} ucontext_t;
```

with *sigset_t* and *stack_t* defined in <*signal.h*>. Here *uc_link* points to the context that will be resumed when the current context terminates (in case the current context was created using **makecontext**(3)), *uc_sigmask* is the set of signals blocked in this context (see **sigprocmask**(2)), *uc_stack* is the stack used by this context (see **sigaltstack**(2)) and *uc_mcontext* is the machine-specific representation of the saved context, that includes the calling thread's machine registers.

The function **getcontext**() initializes the structure pointed at by *ucp* to the currently active context.

setcontext(2) - Linux man page

Name

getcontext, setcontext - get or set the user context

Synopsis

#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);

Description

In a System V-like environment, one has the two types *mcontext_t* and *ucontext_t* defined in <*ucontext.h*> and the four functions **getcontext**(), **setcontext**(), **makecontext**(3) and **swapcontext**(3) that allow user-level context switching between multiple threads of control within a process.

The *mcontext_t* type is machine-dependent and opaque. The *ucontext_t* type is a structure that has at least the following fields:

```
typedef struct ucontext {
    struct ucontext *uc_link;
    sigset_t         uc_sigmask;
    stack_t          uc_stack;
    mcontext_t       uc_mcontext;
    ...
} ucontext_t;
```

with *sigset_t* and *stack_t* defined in <*signal.h*>. Here *uc_link* points to the context that will be resumed when the current context terminates (in case the current context was created using **makecontext**(3)), *uc_sigmask* is the set of signals blocked in this context (see **sigprocmask**(2)), *uc_stack* is the stack used by this context (see **sigaltstack**(2)), and *uc_mcontext* is the machine-specific representation of the saved context, that includes the calling thread's machine registers.

The function **getcontext**() initializes the structure pointed at by *ucp* to the currently active context.

setcontext(2) - Linux man page

Name

getcontext, setcontext - get or set the user context

Synopsis

#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);

Description

In a System V-like environment, one has the two types *mcontext_t* and *ucontext_t* defined in <*ucontext.h*> and the four functions **getcontext**(), **setcontext**(), **makecontext**(3) and **swapcontext**(3) that allow user-level context switching between multiple threads of control within a process.

The *mcontext_t* type is machine-dependent and opaque. The *ucontext_t* type is a structure that has at least the following fields:

```
typedef struct ucontext {
    struct ucontext *uc_link;
    sigset_t         uc_sigmask;
    stack_t          uc_stack;
    mcontext_t       uc_mcontext;
    ...
} ucontext_t;
```

need to allocate a struct ucontext in memory and pass pointer to a call to getcontext

with *sigset_t* and *stack_t* defined in <*signal.h*>. Here *uc_link* points to the context that will be resumed when the current context terminates (in case the current context was created using **makecontext**(3)), *uc_sigmask* is the set of signals blocked in this context (see **sigprocmask**(2)), *uc_stack* is the stack used by this context (see **sigaltstack**(2)), and *uc_mcontext* is the machine-specific representation of the saved context, that includes the calling thread's machine registers.

The function **getcontext**() initializes the structure pointed at by *ucp* to the currently active context.

setcontext(2) - Linux man page

Name

getcontext, setcontext - get or set the user context

Synopsis

#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);

Description

In a System V-like environment, one has the two types *mcontext_t* and *ucontext_t* defined in <*ucontext.h*> and the four functions **getcontext**(), **setcontext**(), **makecontext**(3) and **swapcontext**(3) that allow user-level context switching between multiple threads of control within a process.

The *mcontext_t* type is machine-dependent and opaque. The *ucontext_t* type is a structure that has at least the following fields:

```
typedef struct ucontext {
    struct ucontext *uc_link;
    sigset_t         uc_sigmask;
    stack_t          uc_stack;
    mcontext_t       uc_mcontext;
    ...
} ucontext_t;
```

Later you call setcontext with that pointer to copy that state to the processor

with *sigset_t* and *stack_t* defined in <*signal.h*>. Here *uc_link* points to the context that will be resumed when the current context terminates (in case the current context was created using **makecontext**(3)), *uc_sigmask* is the set of signals blocked in this context (see **sigprocmask**(2)), *uc_stack* is the stack used by this context (see **sigaltstack**(2)), and *uc_mcontext* is the machine-specific representation of the saved context, that includes the calling thread's machine registers.

The function **getcontext**() initializes the structure pointed at by *ucp* to the currently active context.

setcontext(2) - Linux man page

Name

getcontext, setcontext - get or set the user context

Synopsis

#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);

Description

In a System V-like environment, one has the two types *mcontext_t* and *ucontext_t* defined in <*ucontext.h*> and the four functions **getcontext**(), **setcontext**(), **makecontext**(3) and **swapcontext**(3) that allow user-level context switching between multiple threads of control within a process.

The *mcontext_t* type is machine-dependent and opaque. The *ucontext_t* type is a structure that has at least the following fields:

```
typedef struct ucontext {
    struct ucontext *uc_link;
    sigset_t         uc_sigmask;
    stack_t          uc_stack;
    mcontext_t       uc_mcontext;
    ...
} ucontext_t;
```

Look in sys/ucontext.h for more info (on Ubuntu /usr/include/sys/ucontext.h)

with *sigset_t* and *stack_t* defined in <*signal.h*>. Here *uc_link* points to the context that will be resumed when the current context terminates (in case the current context was created using **makecontext**(3)), *uc_sigmask* is the set of signals blocked in this context (see **sigprocmask**(2)), *uc_stack* is the stack used by this context (see **sigaltstack**(2)), and *uc_mcontext* is the machine-specific representation of the saved context, that includes the calling thread's machine registers.

The function **getcontext**() initializes the structure pointed at by *ucp* to the currently active context.

Task 1

finish implementing parseUcontext.c

Changing thread context

when creating a thread

- copy thread context from existing thread
- change 3 things

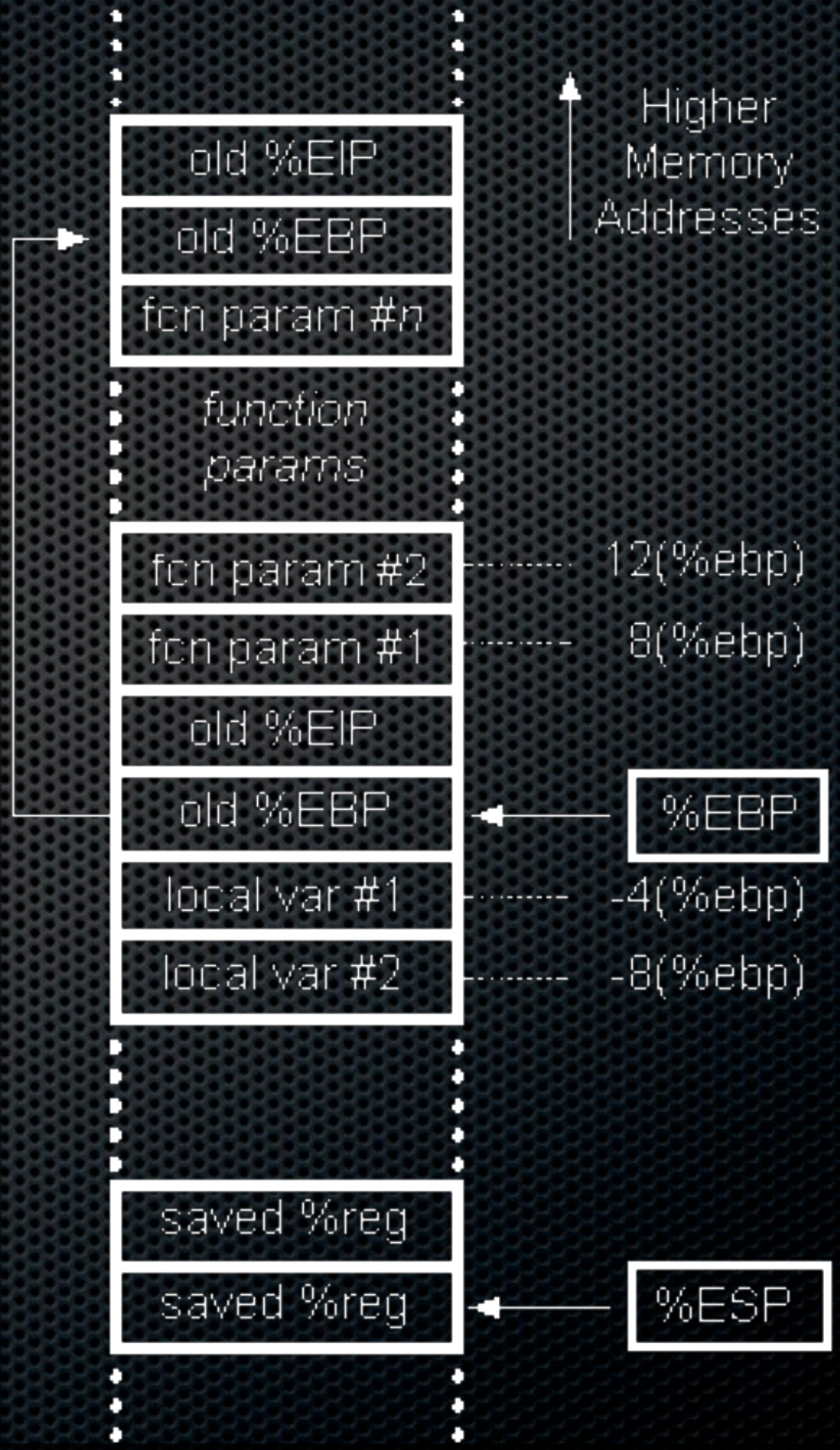
Change 3 things

- change the program counter to point to the function the thread should run
- allocate and initialize a new stack
- change the stack pointer to point to the top of the new stack

Stack

on Intel chips

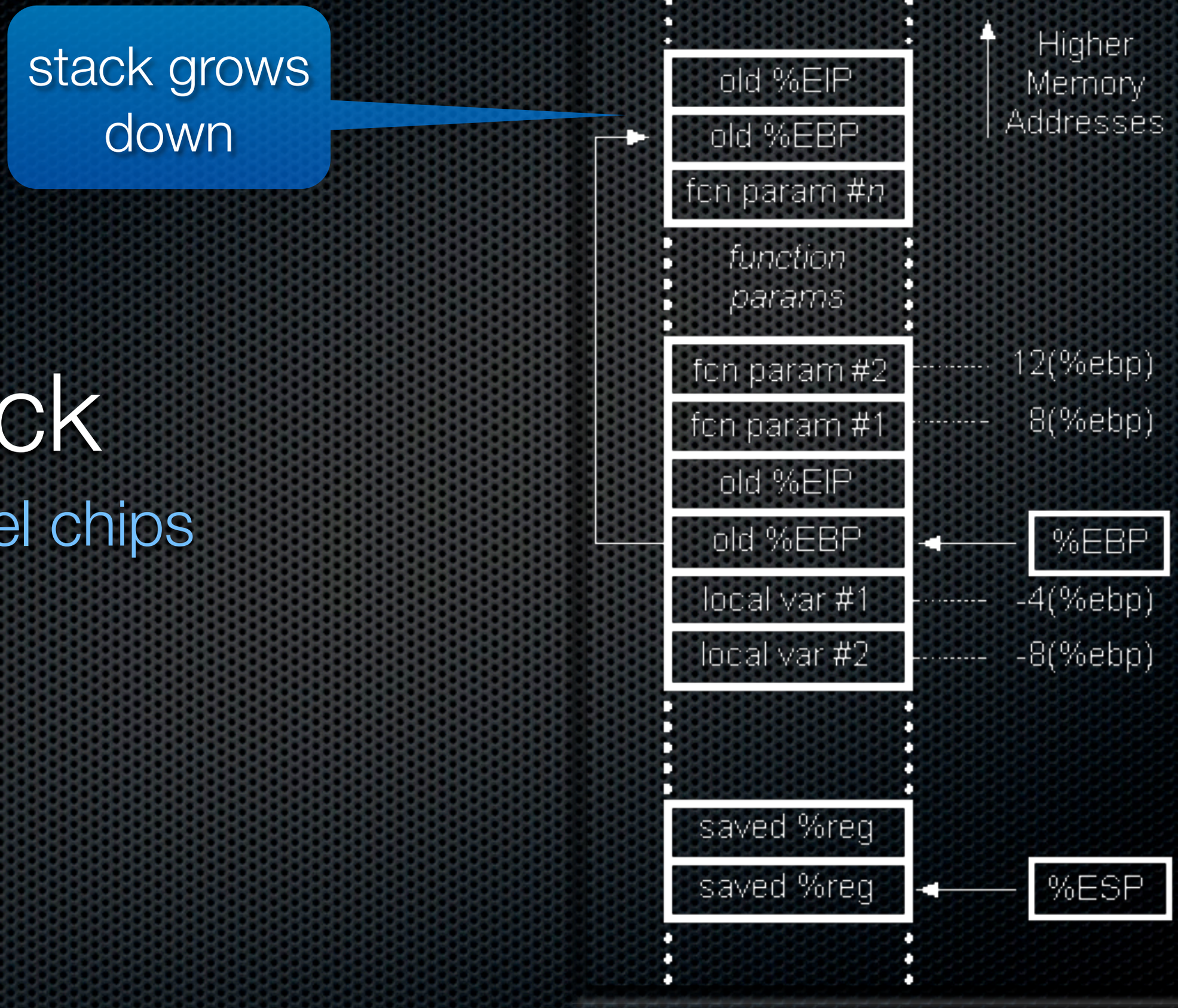
x86



Stack

on Intel chips

x86

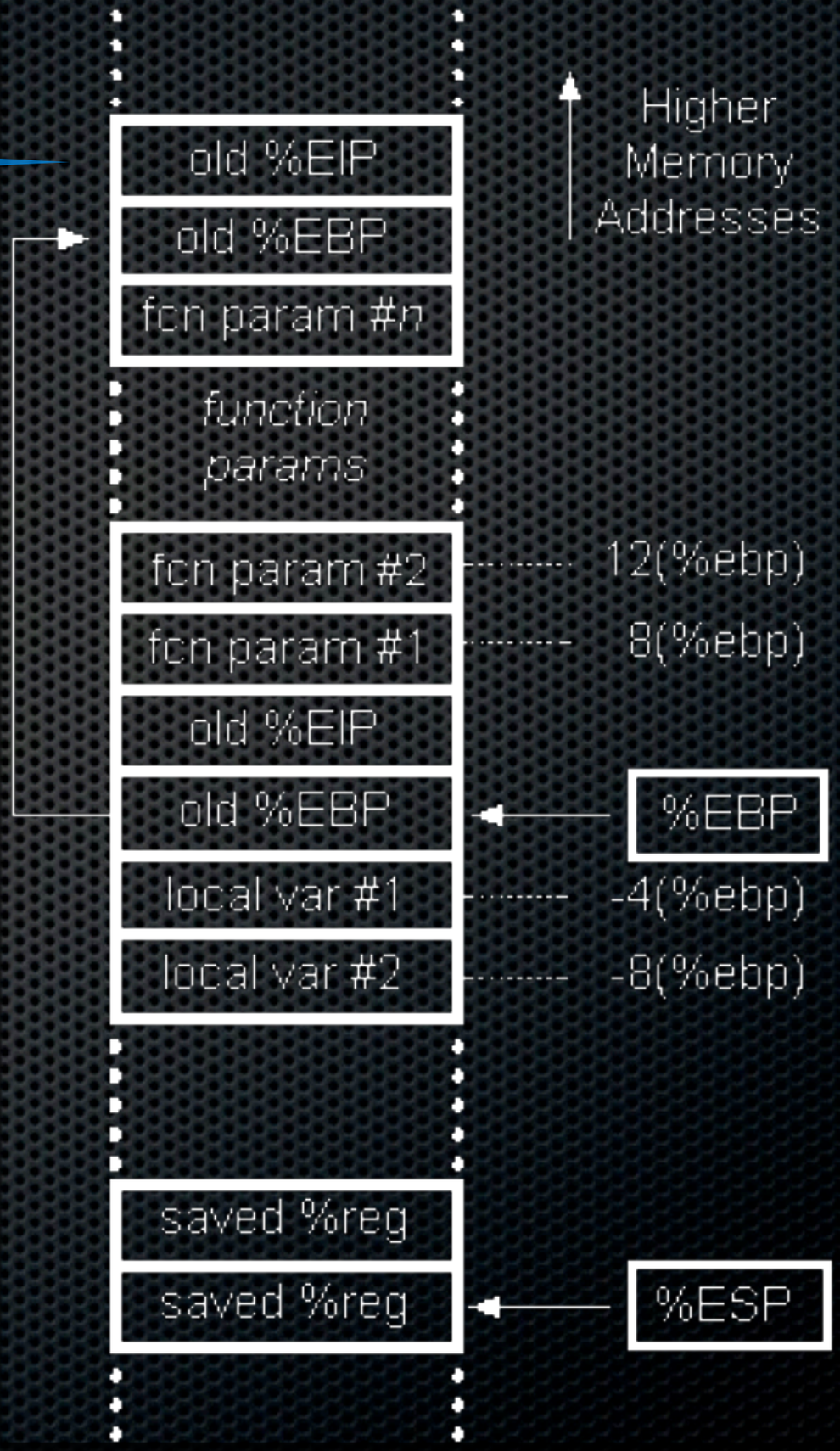


Instruction pointer (aka program counter)

Stack

on Intel chips

x86

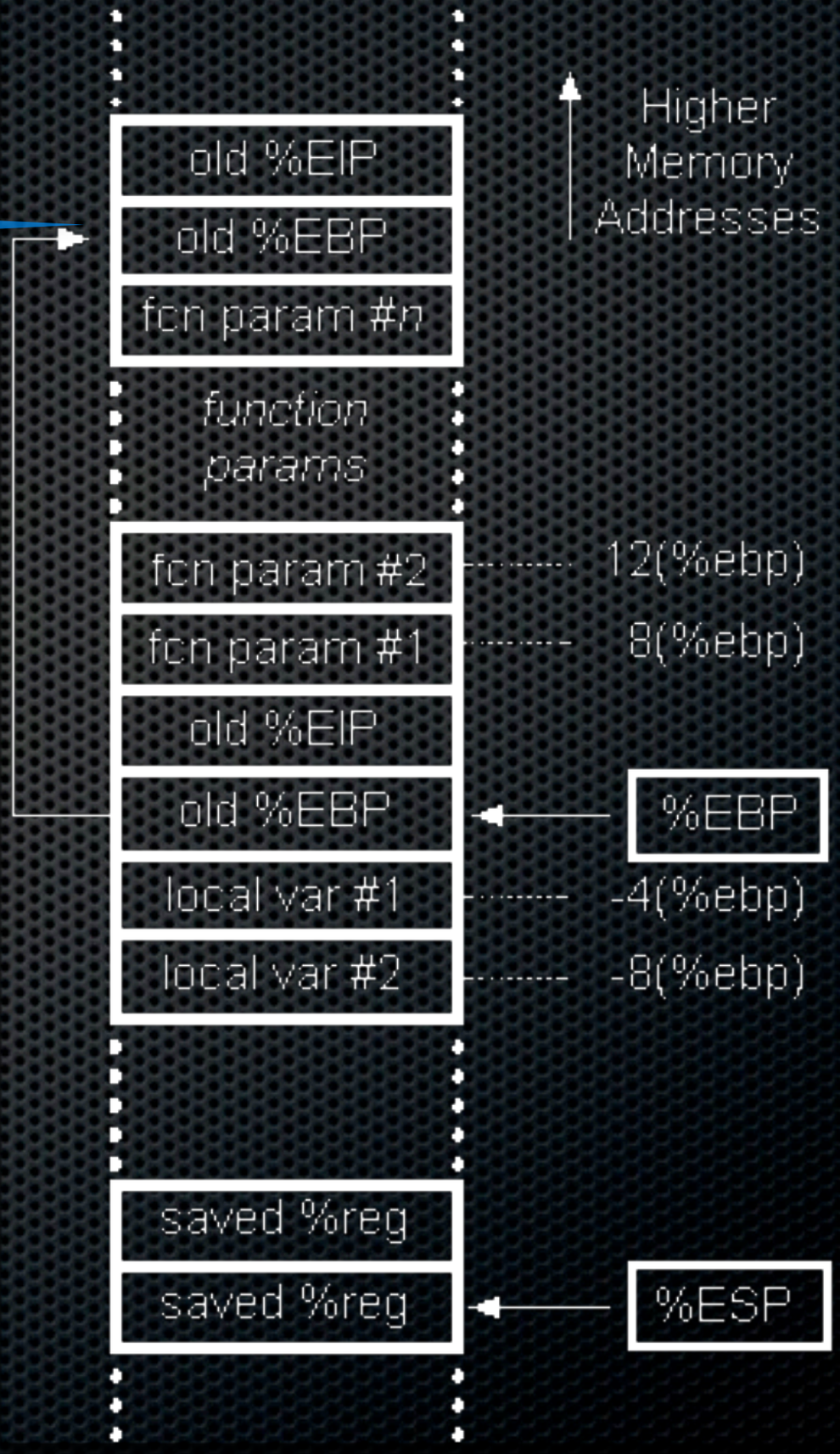


Stack

on Intel chips

x86

frame pointer

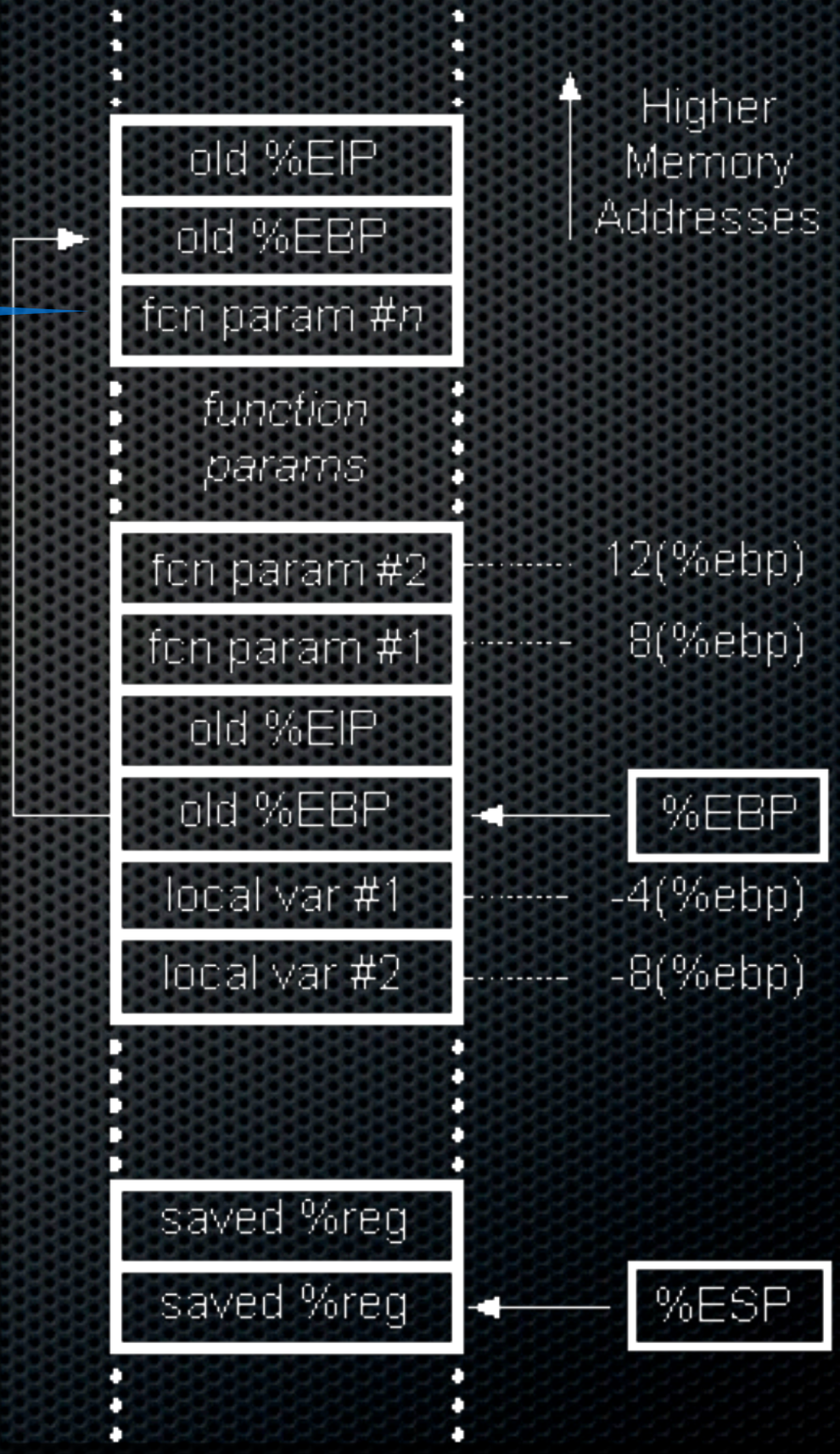


parameters pushed from right to left

Stack

on Intel chips

x86



You are implementing an API

- Tid ULT_Yield(Tid tid): suspend caller and run thread tid
  - ULT_ANY
  - ULT_SELF
  - returns tid of thread executed or:
    - ULT_INVALID
    - ULT_NONE (no threads available)

You are implementing an API

- Tid ULT_CreateThread(void (*fn)(void *), void *arg): create a new thread. It will either return the tid of the new thread or
  - ULT_NOMORE: library can't create more threads
  - ULT_NOMEMORY: couldn't allocate memory for the stack.

You are implementing an API

- Tid ULT_DestroyThread(Tid tid): destroy the thread.

On programming and logistics

Logistics

- works in teams of 2
- grading will be done on ubuntu 12.04
  - Bodhi Linux
- read and reread the project description
- start creating a road map of the C files.

Logistics cont'd

- there are very few lines of code to write.
- hacking doesn't work