Real-time Operating Systems and Systems Programming

Lecture 2
Understanding Memory (Stack)

ARVUTITEADUSE INSTITUUT
DEPARTMENT OF COMPUTER SCIENCE

Memory

- Processor registers (hidden in C)
- RAM
- Devices (hard disk)
- Internet?
- People??
- Books???

Stack

- Simple data structure
- Efficient implementations
- FILO (as opposed to FIFO)
- Operations: Push, Pop
- Important to us due to call stack
- Often supported in hardware

C implementation

```
typedef struct {
    int size;
    int items[STACKSIZE];
} STACK;
```

```
void push(STACK *ps, int x)
{
    if (ps->size == STACKSIZE) {
        fputs("Error: stack
overflow\n", stderr);
        abort();
    } else
        ps->items[ps->size++] = x;
}
```

```
int pop(STACK *ps)
{
    if (ps->size == 0){
        fputs("Error: stack
underflow\n", stderr);
        abort();
    } else
        return ps->items[--ps->size];
}
```

Hardware implementation

- Special stack register (can be read/written)
  - We will name it %esp in further examples
- Assembly instructions to manipulate it

The Dreadful Assembly

Short introduction to assembly

- Mainly moves data around (mov series)
- Jumps, conditional jumps (jmp series)
- Arithmetics
- Management (push, pop, call, return)
- Examples from IA32
  - Word = 16 bit due to ancient history
  - Double word for 32 bits

Registers

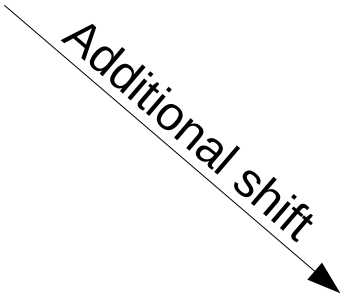
- 8 registers for 32bit values
- General purpose: %eax %ecx %edx %ebx %edi %esi (Historical names, would be simpler)
- Fun registers: %esp %ebp (Stack pointer & Frame pointer)
- Can be addressed also in smaller segments
- %eax[%ax[%ah[ ] %al[ ]]

Aside: C numeric constants

- Decimal: 10; -10
- Octal: 037 0431 (leading 0)
- Hexadecimal: 0xf1 0xdada (leading 0x)
- Unsigned: 10u, 0xafU
- Long: 10l 10L; Short: 10s 10S
- Floating-point: 0.04 4e-2 10.0 1e2

Operands

- Instructions have operands (arguments)
- Immediate
  - Constant values
  - $1024, $-10, $0xdeadbeef
- Register
  - %eax, %al
- Memory
  - 24(%eax, %edx, 1) ~ *Immediate(reg_b, reg_i, scale)*



Examples

```
movl $0x5040, %eax
movl %ebp, %esp
movl (%edi, %ecx), %eax
movl $-17, (%esp)
```

Stack operations

%eax = 0x123 %edx = 0 %esp = 0x108

pushl %eax

%eax = 0x123 %edx = 0 %esp = 0x104

popl %edx

%eax = 0x123 %edx = 123 %esp = 0x108

Procedures

- Call involves passing both data and control from one code to another.
- Must store local variables and arguments, deallocate them on exit

Stack frame

- Uses frame pointer to keep track of previous frame
- Stack pointer tracks “top” of stack

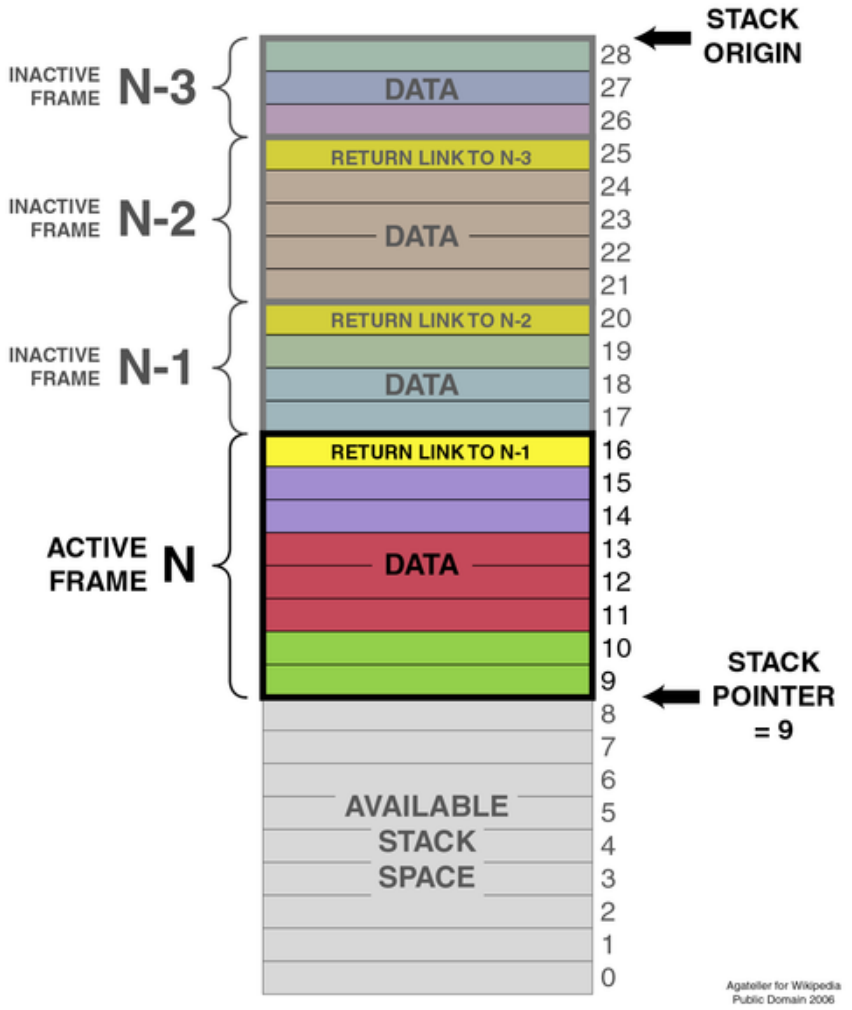


Image source: Wikipedia

One frame contains

- Address of last %ebp
  - Current frame pointer points to it (data accessed in relation to it)
- Saved registers
- Local variables (out of registers; array; &)
- Any temporary data
- Argument building area
- Return address (only if not active frame)

Transfer of control

- For procedure calls, processor supports the following instructions:
  - **call *Label*** / **call **Operand*** – calls procedure
  - **leave** – prepares stack for return
  - **ret** – return from call

1. Prepare stack
2. Call procedure

Call instruction

- Can start executing from an address or a label
- Pushes return address to stack (return address is next instruction from the call)
- Jumps to called address (= set program counter to the start of a procedure)

Ret instruction

- Pop an address from stack
- Go to the address (copy it to Program Counter)
- *To use properly, stack pointer must point to the "bookmark" address that* call instruction *stored.*
- For preparation, leave instruction is used

```
Leave:
movl %ebp, %esp
popl %ebp
    # note: %ebp == stack frame
```

Recap

- *call* pushes return address to stack, jumps

- new procedure saves old stack frame to stack
- Copies current stack pointer to frame pointer
- ...
- Copy frame pointer to stack pointer
- Restore old frame pointer
- Return to stored bookmark

Register conventions

- %eax, %edx, %ecx – Caller save
  - Procedures can overwrite them as want, but must restore them after return, as they may get overwritten
- %ebx, %esi, %edi – Callee save
  - Procedures can overwrite them only if they save them and restore them before returning
- %eax is the return register

What reflects to C?

- Automatic variables live on stack
- Function arguments are copied to stack before calling (call by value)
- Using pointers as arguments to functions can make calls by reference
- Uninitialized variables contain garbage
- Pointers to freed stack contain garbage
- Writing over a stack frame pointer is Not Good
- Writing over the return address is worse

Buffer overflow exploitation

- When a buffer overflows, it is possible to write over the return pointer to point within the buffer itself
- The buffer gets executed
- Newer C implementations protect stack for desktop compilation

How to remove variable from stack?

- Easy, declare it as *static*.
  - static int i = 0xf00;
- Moves the variable to heap
  - *the next lecture will cover this*

Nice hack using the stack

- The state of a program is defined by:
  - CPU register content (easy to save and restore as we saw)
  - Stack content
  - Heap content
- You could save the program state by saving the above

Long Jump

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

- setjmp ja longjmp can aid interrupt handling
- setjmp() saves the *stack* into env buffer, for use by longjmp() function. The env is usable only once.
- setjmp() returns 0 on the first call and a different value on the second call after longjmp() has been called. It can return "twice"!

Long Jump (2)

```
void longjmp(jmp_buf env, int val);
```

- longjmp() restores the saved environment. After longjmp() the program behaves like setjmp() would have returned the value val. longjmp() cannot send 0 since it will be replaced with 1.

```
jmp_buf env;
if ((val=setjmp(env)) == 0)
   printf("Now we have set long jump\n");
else
   printf("Long jump has returned value
%d\n",val);
..........…
longjump(env, 3);
```

long.c