Stack and Subprograms

Purpose of Stack

- Stack is a portion of memory which is used to
  - To save return address in subprogram call
  - To pass parameters to subprograms (including C function calls)
  - To allocate space for local variables for subprograms
  - To save registers to be preserved across subprogram calls
- In Linux
  - `ESP` is already set up by OS when a program starts
  - Storing an arbitrary value in `ESP` would disrupt the existing stack

Push and Pop

- ESP register
  - Stack pointer
  - Keeping track of top of the stack

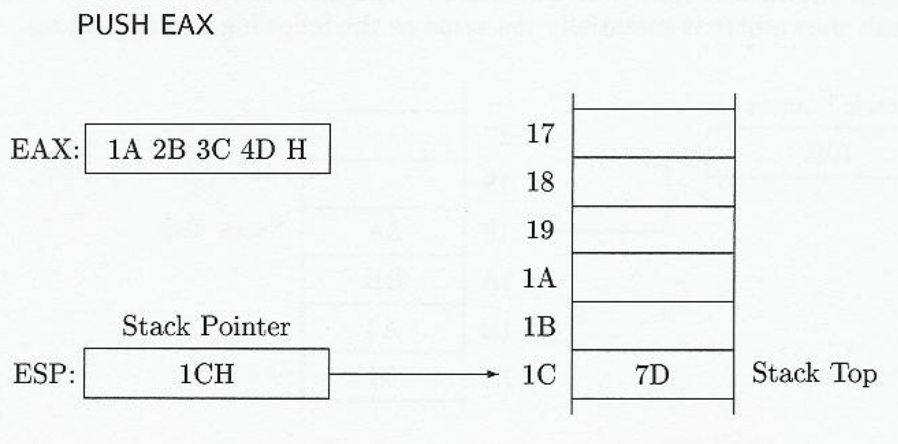


Figure 7-3. Before PUSH EAX

- PUSH intruction

```
PUSH EAX
  SUB ESP, 4
  MOV [ESP], EAX
```

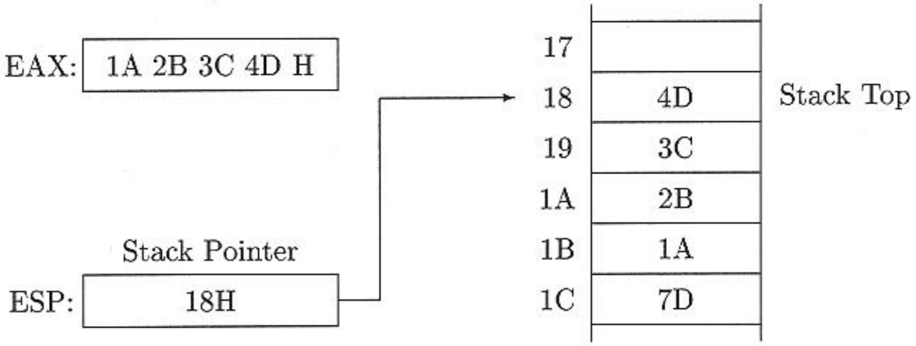


Figure 7-4. After PUSH EAX

Push and Pop

- POP instruction
  - POP EBX

    MOV EBX, [ESP]

    ADD ESP, 4

- PUSHA and POPA instruction

- The order of pops is the exact reverse of the order of the pushes

```
PUSH EAX
PUSH EBX
; computations
POP EBX
POP EAX
```

CALL and RET

- CALL
  - Pushes the address of the next instruction on the stack, and
  - Jumps to certain address
- RET
  - Pops off an address , and
  - Jumps to that address
- It is very important that one manage the stack correctly so that the right number is popped off by RET instruction

CALL and RET

```
;
        MOV ESP, 2000H ; initialize the stack
;
; Edlinas programs must initialize the stack
; Unix programs must not.
;
        IN EAX,[0]        ; Get a user input
        MOV EBX, EAX      ; EBX is where the subroutine works.
        CALL subpr        ; Leave for the subroutine.
        MOV EAX, EBX      ; Back now from the subroutine!
        OUT [1], EAX      ; Output the incremented value
        RET               ; Go back to Edlinas.
;
subpr:
        INC EBX           ; Subprogram does its job.
        RET               ; Go back to the main program.
```

Program 7.3

Calling Convention

- Call and return
  - A subprogram is invoked with a `CALL` instruction and returns via a `RET`
- Parameter passing
  - Parameters are pushed by the caller
  - Parameters on the stack are accessed using `EBP` by subprogram
  - Parameters are removed by the caller
- Local variables
  - Local variables are allocated on the stack
  - Local variables are accessed using `EBP` too
- Return value is passed via `EAX` register

Parameter Passing

- Parameters to a subprogram may be passed on the stack
  - Parameters are pushed by the caller in the reverse order that they appear in the call
  - Parameters on the stack are not popped off by the subprogram, instead they are accessed from the stack itself
  - Parameters on the stack are accessed using indirect addressing with base register `EBP` (not `ESP`) by subprogram, like `[EBP+8]`
  - Parameters are removed after `RET` instruction, i.e. also by the caller
    - To support varying number of arguments
    - `ADD` or `POP` instructions

Accessing Parameters with ESP

		high
ESP+4	Parameter	
ESP	Return address	
		low

(a) Subprogram 호출 직후의 스택

		high
ESP+8	Parameter	
ESP+4	Return address	
ESP	Subprogram data	low

(b) Subprogram이 스택을 사용하는 경우

Accessing Parameters with EBP

			high
ESP+12	EBP+8	Parameter	
ESP+8	EBP+4	Return address	
ESP+4	EBP	Saved EBP	low
ESP		Subprogram data	

(c) EBP를 이용한 파라미터 접근

General Caller and Callee Form with Parameter Passing

Caller:

```
    push  dword 1  ; pass 1 as parameter
    call  subprogram
    add   esp, 4   ; remove parameter from stack
```

Callee:

```
  subprogram:
    push  ebp      ; save original EBP value on stack
    mov   ebp, esp ; new EBP == ESP

  ; subprogram code

    pop   ebp      ; restore original EBP value
    ret
```

Local Variables on the Stack

- The stack can be used as a convenient location for local variables
  - The stack is exactly where C program stores normal (automatic) variables
  - To make subprogram reentrant
  - Local variables are stored right after the saved `EBP` value in the stack, by subtracting the number of bytes required from `ESP`
  - Indirect addressing with base register `EBP` is used to access local variables

General Caller and Callee Form with Local Variables

Callee:

```
subprogram:
 push  ebp       ; save original EBP value on stack
 mov   ebp, esp ; new EBP == ESP
 sub   esp, LOCAL_BYTES  ; # of bytes needed by locals

; subprogram code

 mov   esp, ebp ; deallocate locals
 pop   ebp       ; restore original EBP value
 ret
```

Example 1

```
main()
{
   …
   i = func(1, 2);
   …
}
```

```
  …
  push  dword 2 ; 2nd parameter
  push  dword 1 ; 1st parameter
  call  func   ; call subprogram
  add   esp, 8 ; remove parameters
  …
```

```
int func(int x, int y)
{
    int a, b, c;
    …
    return c;
}
```

```
func:
   push   ebp        ; save original EBP values
   mov    ebp, esp   ; new EBP <- ESP
   sub    esp, 12    ; allocate space needed
                     ; for local variables
                     ; (a, b, and c)
; subprogram code

   mov    eax, [EBP-12]
   mov    esp, ebp   ; deallocate local vars
   pop    ebp        ; restore original EBP
   ret
```

Stack Frame for Example 1

ESP+24	EBP+12	2	high
ESP+20	EBP+8	1	
ESP+16	EBP+4	Return address	
ESP+12	EBP	Saved EBP	
ESP+8	EBP−4	Local variable a	
ESP+4	EBP−8	Local variable b	
ESP	EBP−12	Local variable c	low

Example 2

```c
void calc_sum(int n, int *sump)
{
    register int i;
    int sum = 0;

    for (i=1; i<=n; i++)
        sum += i;

    *sump = sum;

    return;
}
```

```asm
calc_sum:
    push   ebp
    mov    ebp, esp
    sub    esp, 4               ; make room for sum

    mov    dword [ebp-4], 0 ; sum = 0
    mov    ebx, 1               ; ebx == i
for_loop:
    cmp    ebx, [ebp+8]         ; is i <= n ?
    jnle   end_for

    add    [ebp-4], ebx
    inc    ebx
    jmp    for_loop

end_for:
    mov    ebx, [ebp+12] ; ebx = sump
    mov    eax, [ebp-4]  ; eax = sum
    mov    [ebx], eax    ; *sump = sum

    mov   esp, ebp
    pop   ebp
    ret
```

Stack Frame for Example 2

ESP+16	EBP+12	sump	high
ESP+12	EBP+8	n	
ESP+8	EBP+4	Return address	
ESP+4	EBP	Saved EBP	
ESP	EBP−4	sum	low

ENTER and LEAVE instruction

Callee:

```
subprogram:
  enter LOCAL_BYTES, 0     ; # of bytes needed by locals

; subprogram code

  leave
  ret
```

Reentrant and Recursive Subprograms

- A reentrant subprogram must satisfy the following properties
  - It must not modify any code instructions
  - It must not modify global data (data in the `data` and `bss` section). All variables are stored on the stack
- Advantages to writing reentrant code
  - A reentrant subprograms can be called recursively
  - A reentrant program can be shared by multiple processes
  - Reentrant subprograms work much better in multi-threaded programs

Recursive Subprograms

```
int fib(int k)
{
        int d;

        if(k < 2)
              d = 1;
        else
              d = fib(k-1) + fib(k-2);

        return d;
}
```

```
main()
{
       int x, y;
       ;
       printf("Enter a number:  ");
       scanf("%d", &x);
       ;
       y = fib(x);
       ;
       printf("%d\n", y);
}
```

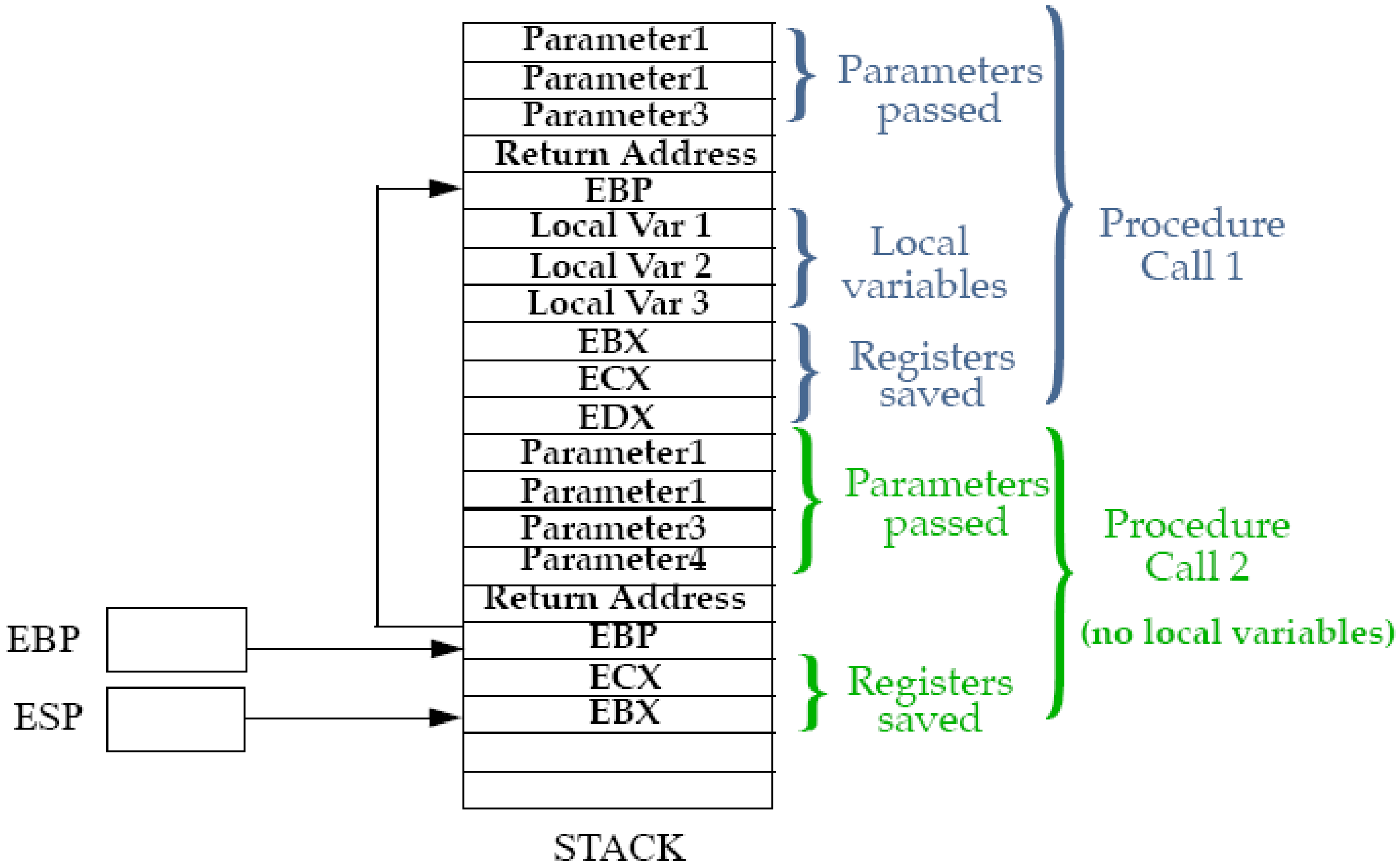
Recursion

- In Assembly

```
        global fib
fib:    PUSH ECX             ; Gives us access to ECX
        MOV EAX, [ESP + 8]
                             ; Access parameter
        CMP EAX, 2           ; if k < 2
        JAE ELS              ;
        MOV EAX, 1           ; Put the return value in EAX
        JMP DUN
ELS:    DEC EAX              ; Get k-1
        MOV ECX, EAX         ; EAX will be overwritten!
        PUSH EAX             ; Pass k-1 as parameter
        CALL fib             ; Get back fib(k-1) in EAX
        ADD ESP, 4           ; Get rid of parameter
        DEC ECX              ; Get k-2
        PUSH ECX             ; Pass k-2 as parameter
        MOV ECX, EAX         ; Put fib(k-1) into ECX
        CALL fib             ; Get back fib(k-2) in EAX
        ADD ESP, 4           ; Get rid of parameter
        ADD EAX, ECX         ; Add the two partial results
        POP ECX              ; Restore the original value
        RET
```

Nested Call Frames

- One call frame created per each subprogram call



Subprogram Calls in Assembly Program

- ***Caller (Before Call) :***
  - Push arguments, last to first
  - `CALL` the function
- ***Callee:***
  - Save caller's `EBP` and set up callee's stack frame (or `ENTER` instruction)
  - Allocate space for local variables
  - Save registers as needed (or `PUSHA` instruction)
  - Perform the task
  - Store return value in `EAX`
  - Restore registers (or `POPA` instruction)
  - Restore caller's stack frame (or `LEAVE` instruction)
  - Return
- ***Caller (After Return) :***
  - POP arguments, get return value in `EAX`

Multi-module Programs

- A Multi-module program is one composed of more than two object files
- In order for module A to use a label defined in B, the `extern` directive must be used
- Labels cannot be accessed externally by default. If a label can be accessed from other modules, it must be declared global in its module by `global` directive

main.asm

```
   extern   get_int, print_sum

   call get_int
   …
   call print_sum
```

sub.asm

```
   global   get_int, print_sum

get_int:
   …

print_sum:
   …
```

Interfacing Assembly with C

- Assembly routines are usually used with C for the following reasons
  - Direct access is needed to H/W features that are difficult or impossible to access from C
  - The routine must be as fast as possible and the programmer can hand-optimize the code better than the compiler can
- Most of the C calling convention have already been specified

Calling Assembly from C Function

- Saving registers
  - C assumes that a function maintains the values of the following registers: `EBX`, `ESI`, `EDI`, `EBP`, `CS`, `DS`, `SS`, `ES`
  - The `EBX`, `ESI`, `EDI` values must be preserved because C uses these registers as register variables
- Label of function
  - Most C compilers prepend a single underscore (_) character at the beginning of the names of function and global/static variables
- Passing parameters
  - The arguments of a function are pushed on the stack in the reverse order that they appear in the function call
- Returning values
  - All integral types (`char`, `int`, `enum`, …) are returned in the `EAX` register
  - If smaller than 32-bits, they are extended to 32-bits when stored in `EAX`
  - 64-bit values are returned in the `EDX:EAX` pair
- Other calling conventions
  - `cdecl` versus `stdcall` calling convention
  - In gcc,
    `void f ( int ) __attribute__ ((cdecl))`

Calling assembly from C

- We can write function in assembly which we can call from a C program

half.asm

```
        global half ; or .globl in Edlinas
half:   PUSH ECX          ; ECX is used, so push it.
        MOV ECX, [ESP + 8]
                          ; Stack top + 2 integers
        MOV EAX, 0
AGN:    INC EAX           ; Count the subtractions
        SUB ECX, 2        ; Repeatedly subtract 2
        JG AGN            ;
        JZ DUN            ; It comes out even
        MOV EAX, 0        ; It's not even.
DUN:    POP ECX
        RET
```

collatz.c

```
main()
{
    int count,x,y;
    count = 0;

    printf("Enter a number:  ");
    scanf("%d", &x);

    while(x != 1)
    {
        count = count + 1;
        y = half(x); /* This is the subroutine call!  */
        if(y != 0)
            x = y;
        else
            x = 3*x + 1;
        printf("\n x = %d.", x);
    }
    printf("\n There were %d iterations.\n\n.",count);
}
```

Command Line Arguments

% ./prog arg1 arg2 arg3

stack

		0	
	BFEEC4A0	argv[3]	→ arg3\0
	BFEEC49C	argv[2]	→ arg2\0
	BFEEC498	argv[1]	→ arg1\0
	BFEEC494	argv[0]	→ ./prog\0
		• • •	
EBP+12	BFEEC414	BFEEC494	; pointer to argument vector
EBP+8	BFEEC410	00000004	; # of arguments
EBP+4	BFEEC40C	001D8DE6	; return address of main
EBP, ESP	BFEEC408	BFEEC468	; old EBP ← current top of stack