CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 6 : 3.7 프로시져

http://eslab.cnu.ac.kr

* Some slides are from Original slides of RBE

전달사항

중간고사 : 10월 31일 금요일 저녁 7시

다음주 읽기 숙제 : pp.290-301, 3.12절

강의 일정

주	날짜	강의실 (화)	날짜	실습실 (목)
1	9월 2일	Intro	9월 4일	리눅스 개발환경 익히기
2	9월 16일	정수	9월 11일	**소수 1**
3	9월 23일	소수2	9월 18일	GCC & Make,
4	9월 30일	어셈1 – 데이터 이동 제어문	9월 25일	Data lab
5	10월 7일	어셈2 – 제어문	10월 2일	**펜티엄 어셈블리**
6	10월 14일	어셈3 – 프로시져	10월 16일	GDB
7	10월 21일	시스템 보안	10월 23일	Binary bomb1
8	10월 28일	시험휴강	10월 30일	Binary bomb 2
9	11월 4일	프로세스 1	11월 6일	**Tiny shell 1**
10	11월 11일	**프로세스 2**	11월 13일	Tiny shell 2
11	11월 18일	시그널	11월 20일	Tiny shell 3
12	11월 25일	동적메모리 1	11월 27일	Malloc lab1
13	12월 2일	동적메모리 2	12월 4일	Malloc lab2
14	12월 9일	**기말고사**	12월 11일	Malloc lab3
15	12월 16일	**Wrap-up/종강**		

현상태 점검 : 우리는 지금..

IA32 어셈블리어

- 데이터 이동명령어
- 연산 명령어
- 제어 명령어

오늘의 주제

- 프로시져의 구현

프로시져 Procedure

프로시져를 사용할 때 벌어지는 일

- 파라미터 넘겨주기
- 실행 코드의 점프
- 지역변수들을 위한 저장장소 확보
- 프로시져 종료시에 할당받은 저장장소 돌려주기

IA32에서 파라미터 넘겨주기와 지역변수 공간은 프로그램 스택을 이용한다

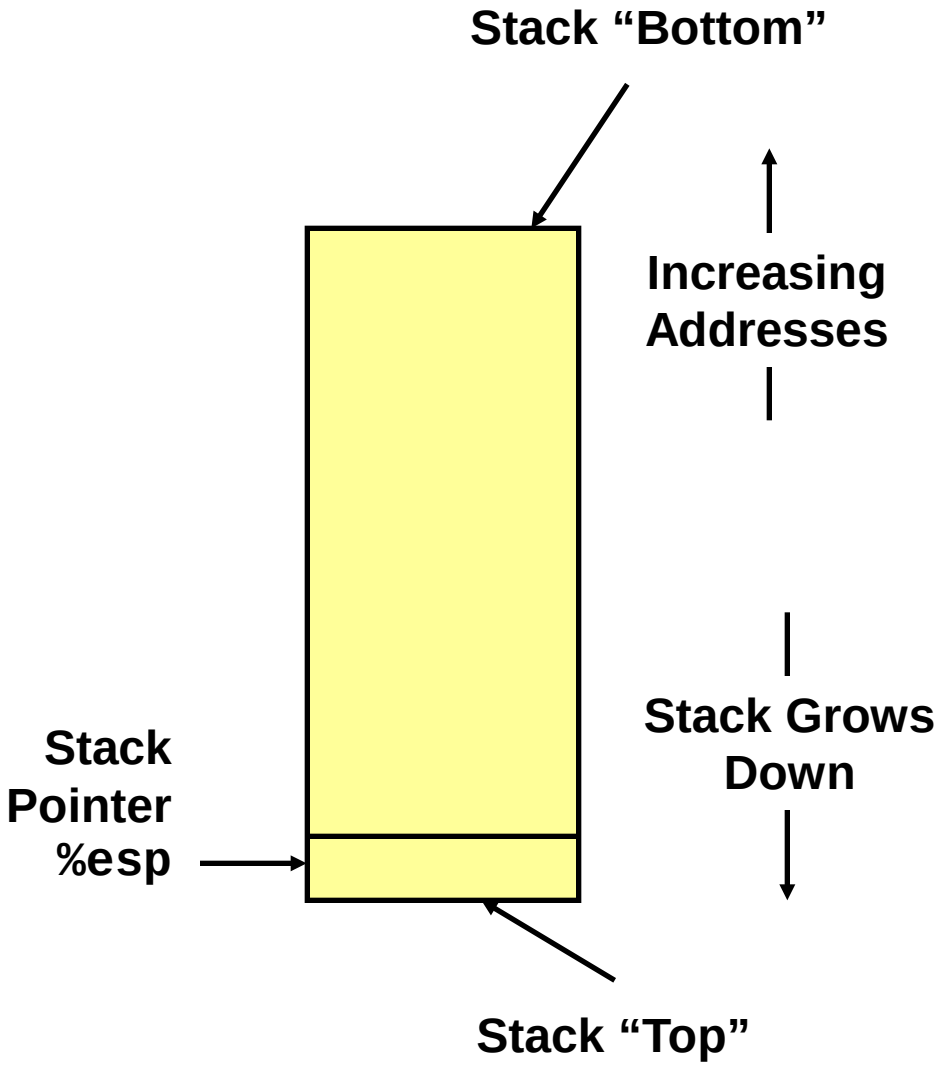
IA32 스택

스택은 메모리의 일부 영역이다

스택은 아래쪽으로 주소가 감소하며 커진다

%esp 레지스터는 최소 스택주소를 가리키고 있다.

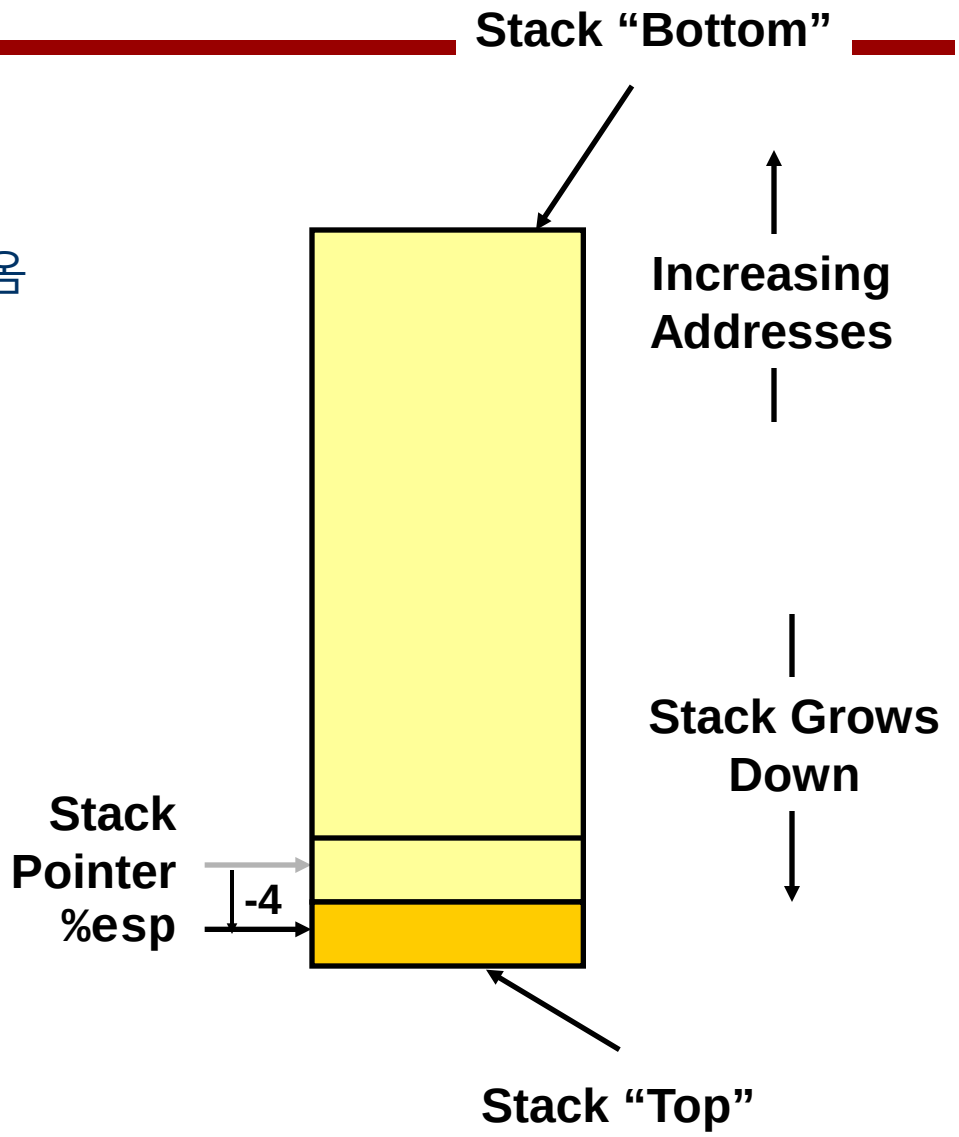
- 스택 top element 을 가리킨다



IA32 스택 Push

Pushing

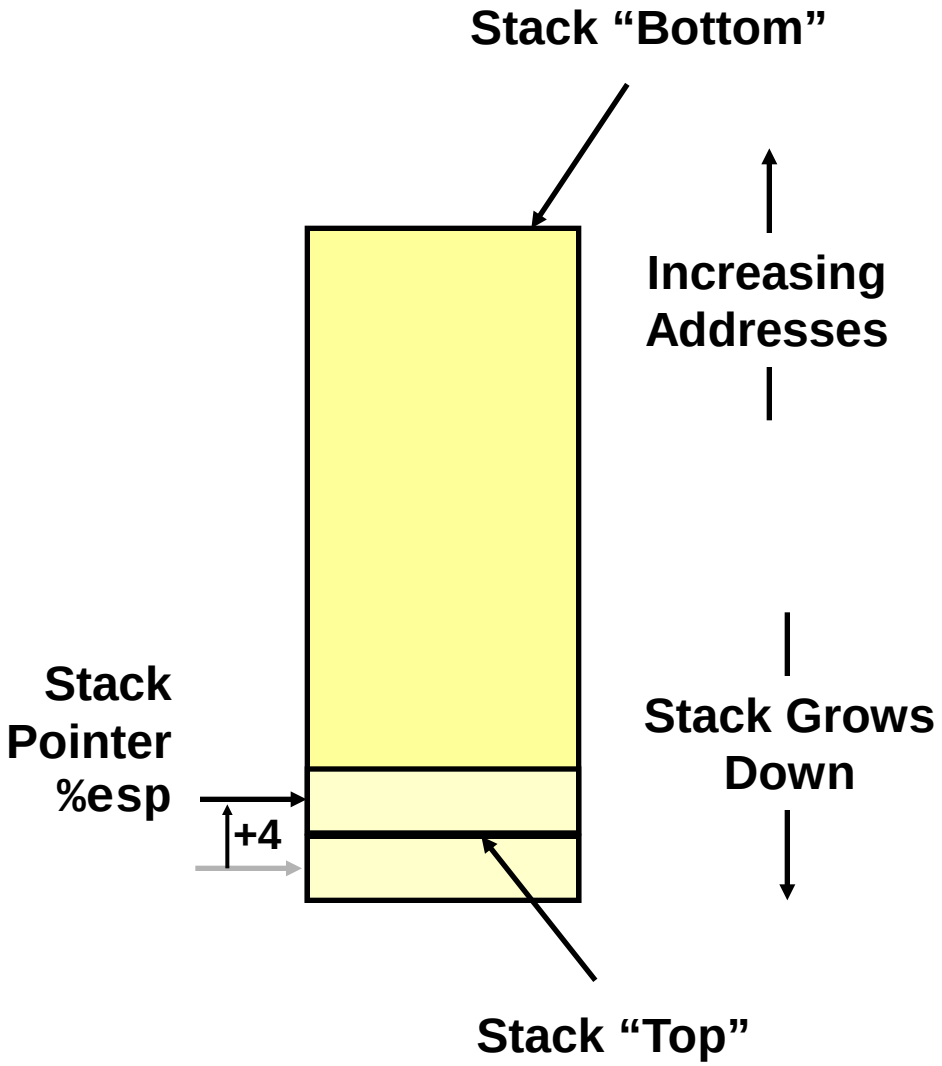
- `pushl` *Src*
- Src 로부터 오퍼랜드를 가져옴
- %esp 를 4 감소시킴
- %esp 가 가리키는 주소에 오퍼랜드를 기록함



IA32 스택 Pop

Popping

- `popl` *Dest*
- %esp 로부터 오퍼랜드를 읽어옴
- %esp 를 4 증가시킴
- 오퍼랜드를 *Dest*에 기록함



프로시져 호출 및 리턴

프로시져 호출

- `call label` 리턴 주소를 스택에 push;
  Jump to *label*

리턴 주소

- call 명령어의 바로 다음 명령어 주소가 됨
- 예

```
804854e: e8 3d 06 00 00      call   8048b90 <main>
8048553: 50                  pushl  %eax
```

  - 리턴주소 = 0x8048553

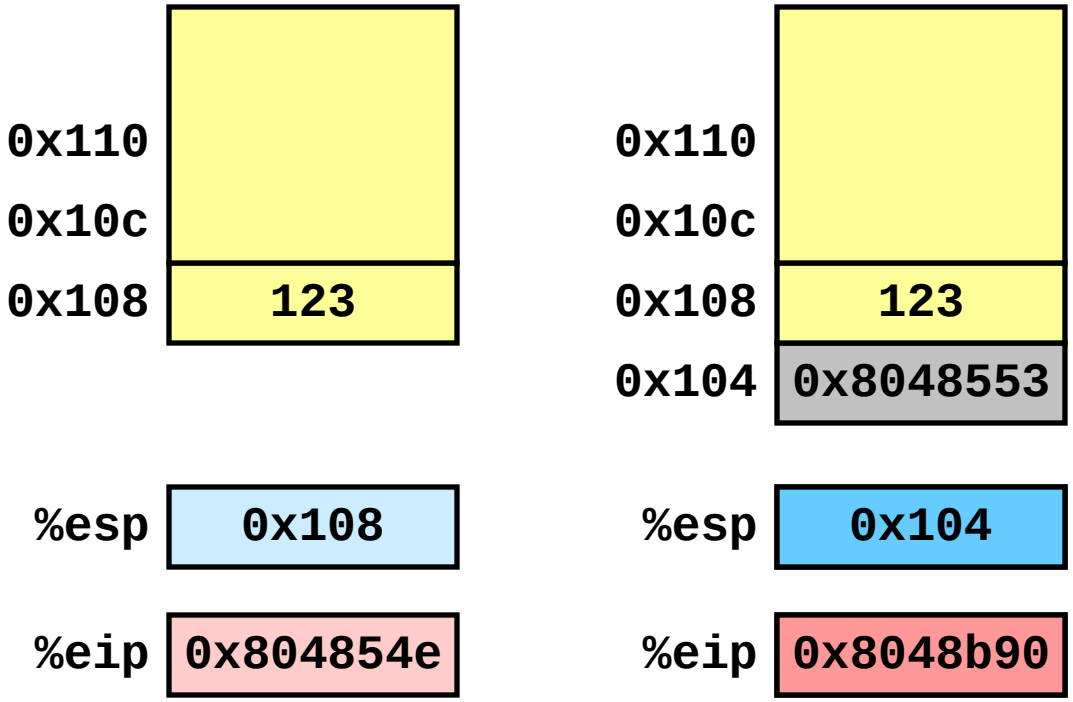
피호출 프로시져로부터 호출 프로시져로 리턴

- `ret` 스택에서 리턴 주소를 pop함;
  Jump to return address

프로시져 호출 예제

```
804854e:  e8 3d 06 00 00        call   8048b90 <main>
8048553:  50                    pushl  %eax
```

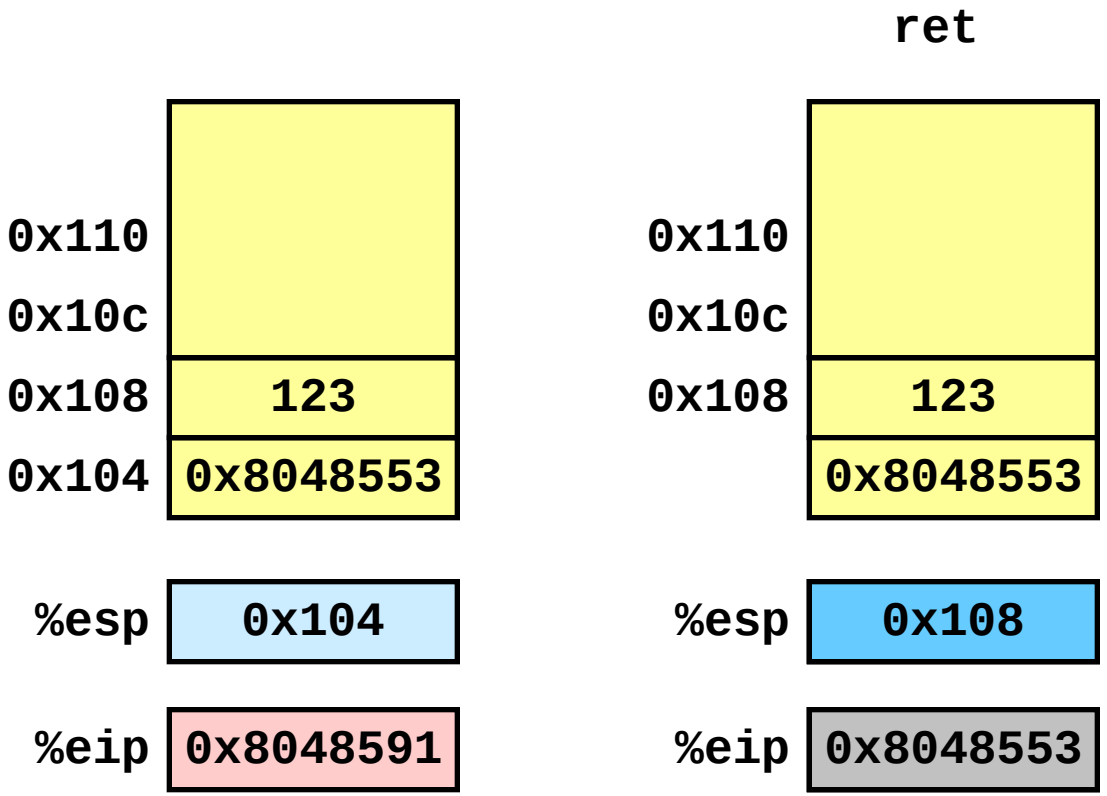
call 8048b90



%eip is program counter

프로시져 리턴 예제

```
8048591:   c3          ret
```



프로그래밍에서 스택의 역할

Recursion을 지원하는 프로그램언어

- e.g., C, Pascal, Java
- Code must be "*Reentrant*"(재진입가능)
  - 한 개의 프로시져가 동시에 여러 번 실행 가능하다(?)
- 각 프로시져 호출(진입)마다 별도의 상태정보를 저장해 주어야 한다.
  - Arguments
  - Local variables
  - Return pointer

프로시져의 상태정보는 제한된 수명을 갖는다

- 호출되었을 때 생성되고, 리턴시에 소멸된다

스택 프레임 Stack Frame

- 한 개의 프로시져 호출 시에 할당되는 스택영역

호출 상관도

예제 코드

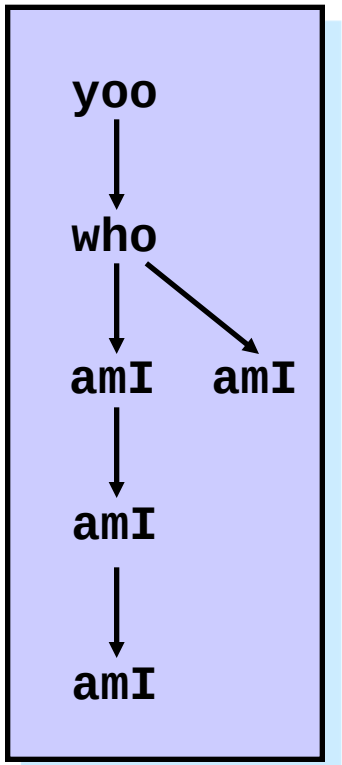
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

- amI( ) 는 재귀함수

Call Chain

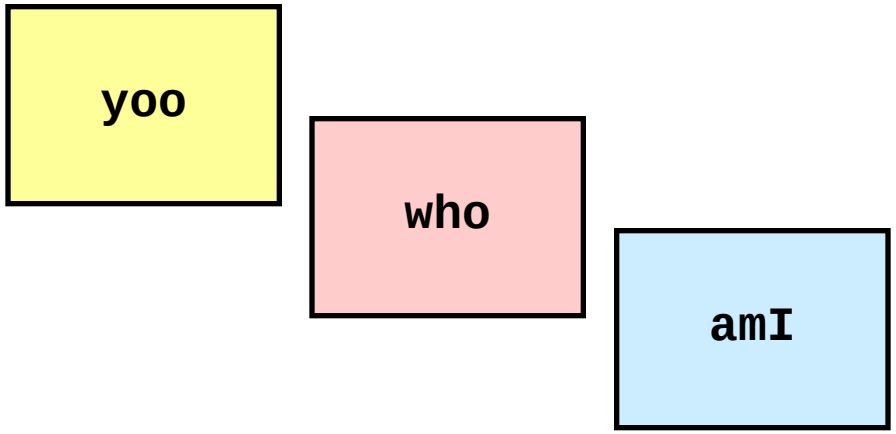


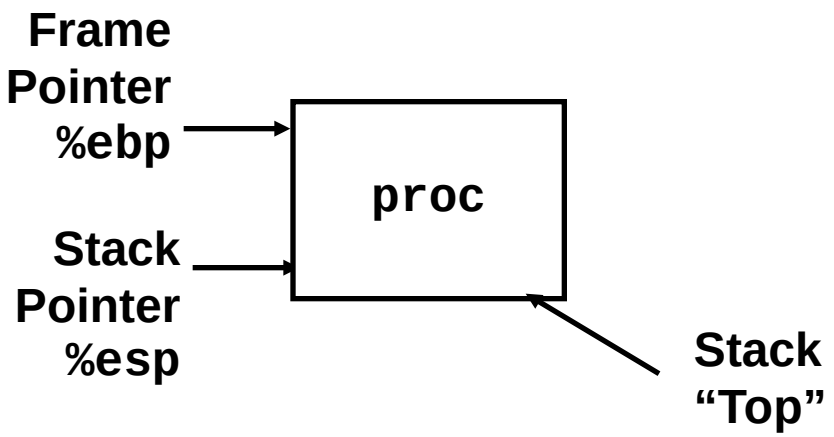
스택 프레임

Contents
- 지역변수(Local variables)
- 리턴정보
- 임시저장공간

Management
- 프로시져가 시작될 때 공간이 확보됨
  - “Set-up” code
- 리턴시에 공간을 돌려줌
  - “Finish” code

Pointers
- Stack pointer %esp
  스택 top을 가리킴
- Frame pointer %ebp
  현재 프레임의 시작을 가리킴



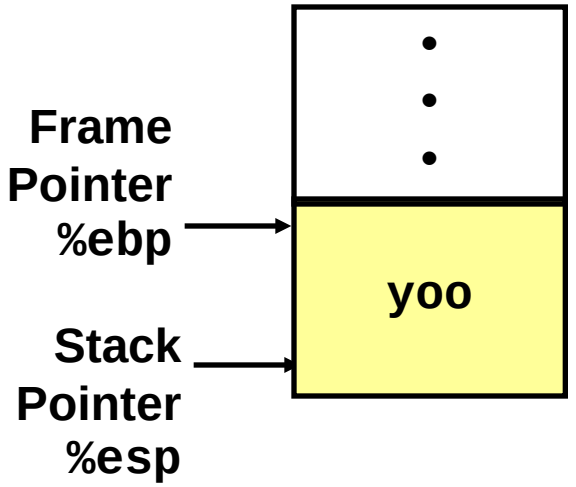


스택의 동작 (1)

```
yoo(…)
{
    •
    •
   who();
    •
    •
}
```

Call Chain

yoo



스택의 동작 (2)

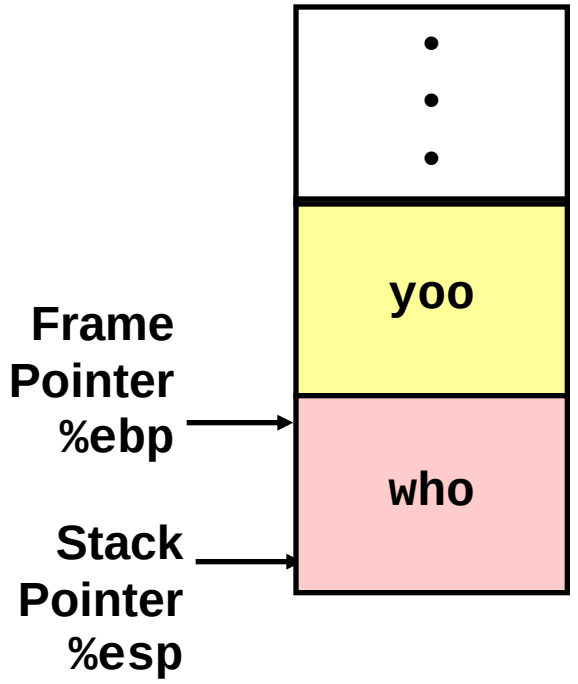
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

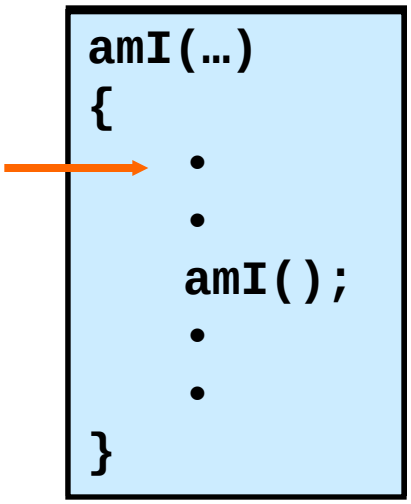
yoo

↓

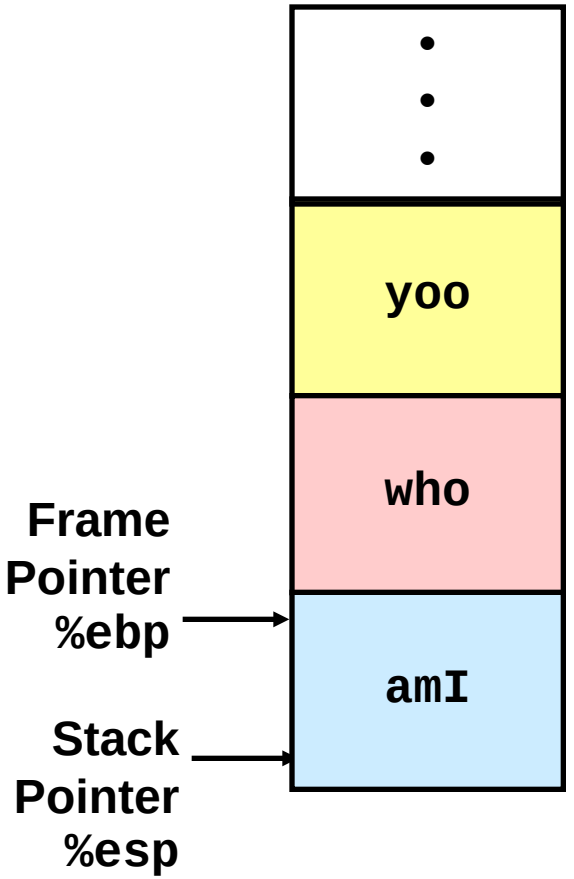
who



스택의 동작 (3)



Call Chain

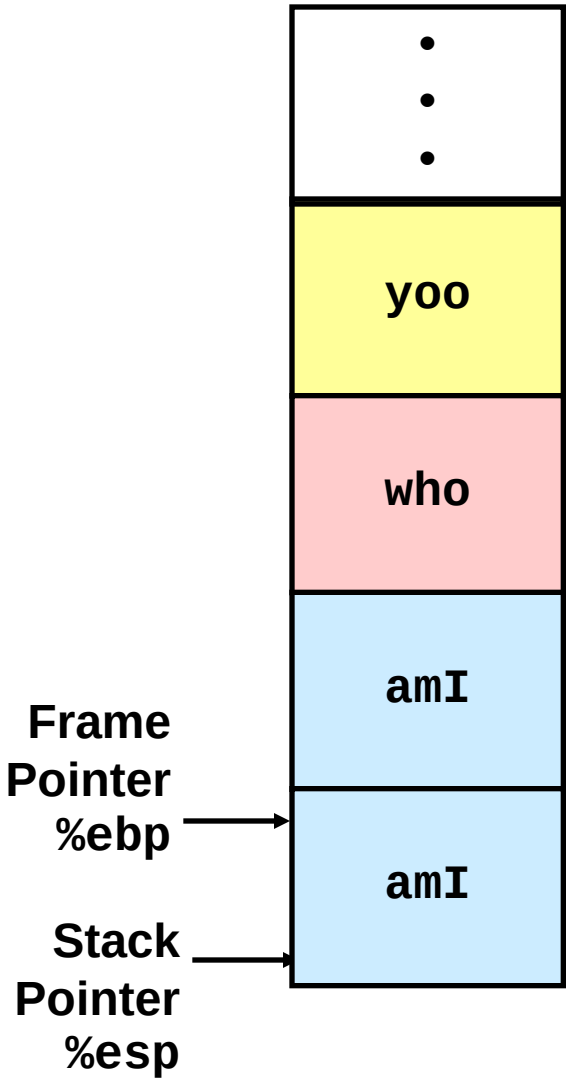


스택의 동작 (4)

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

yoo
↓
who
↓
amI
↓
amI

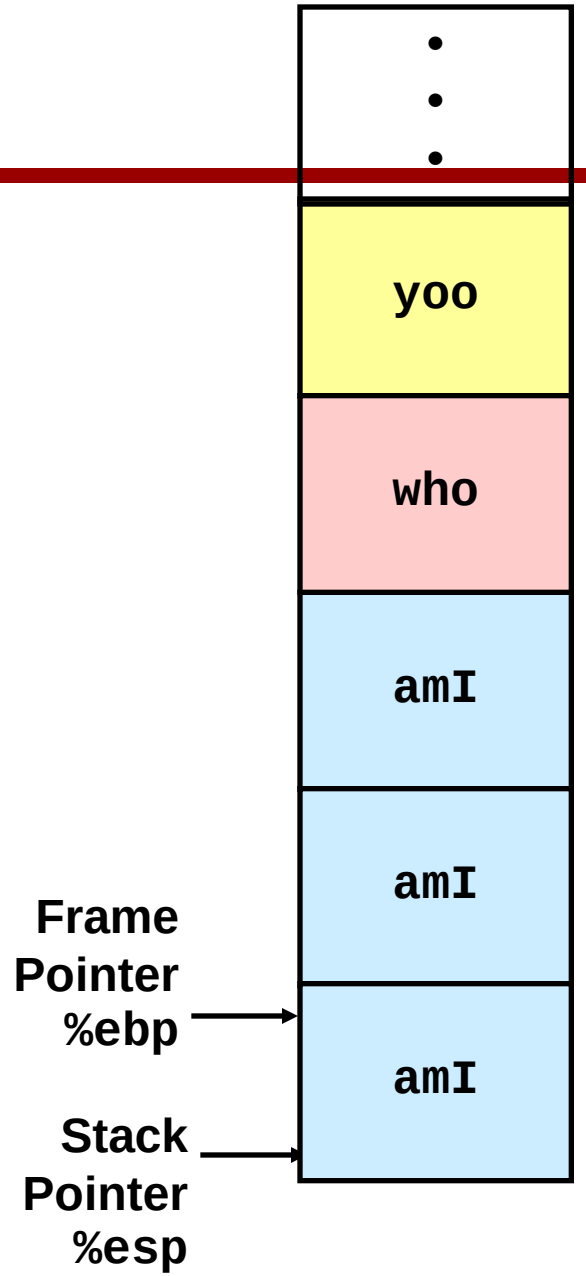


스택의 동작 (5)

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

yoo
↓
who
↓
amI
↓
amI
↓
amI

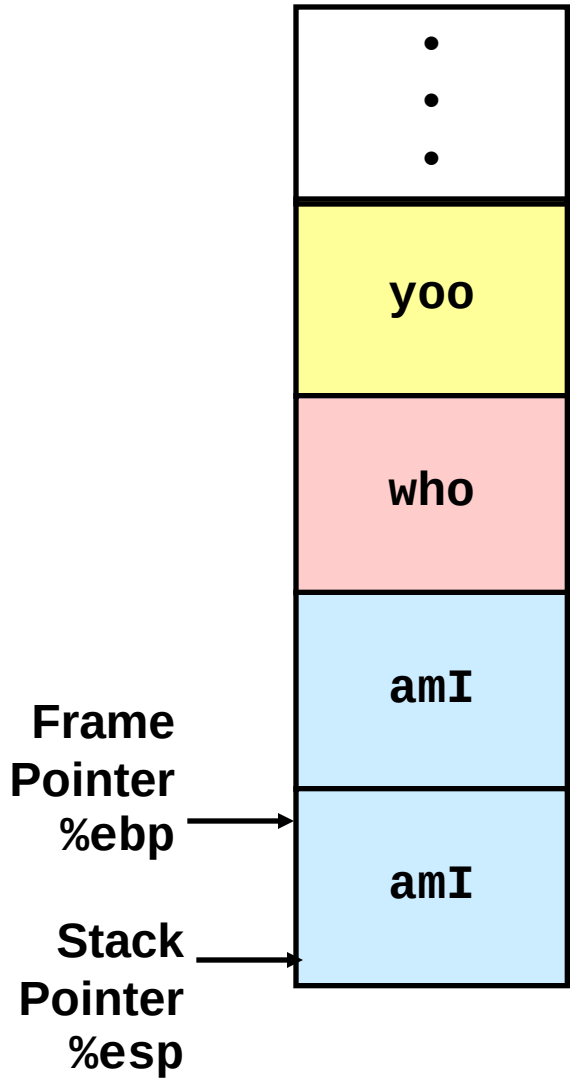


스택의 동작 (6)

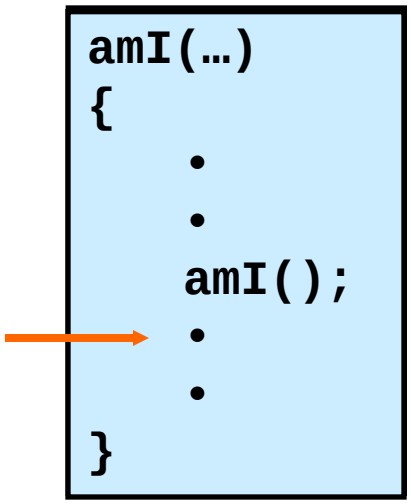
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

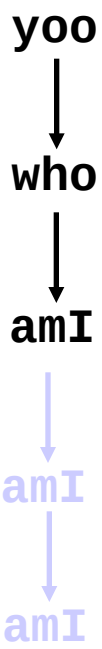
yoo
↓
who
↓
amI
↓
amI
↓
amI

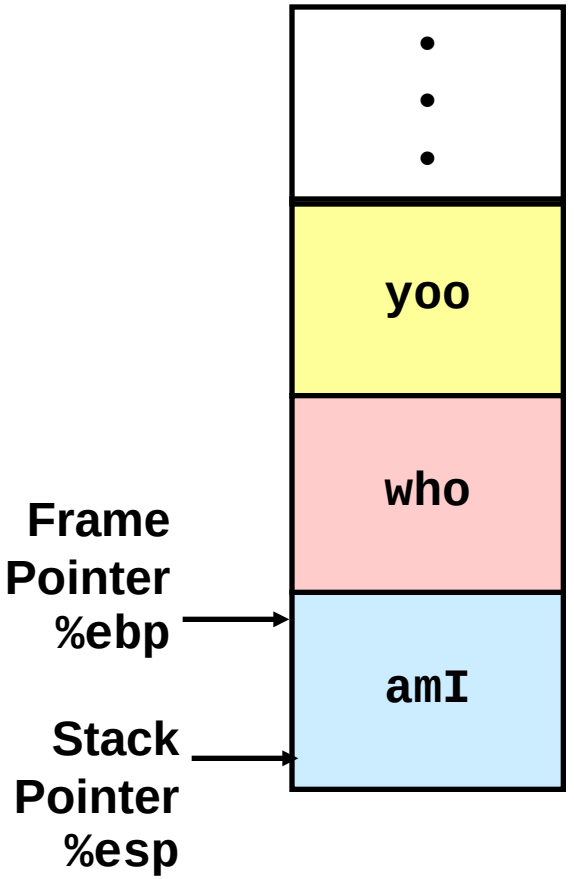


스택의 동작 (7)



Call Chain





스택의 동작 (8)

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

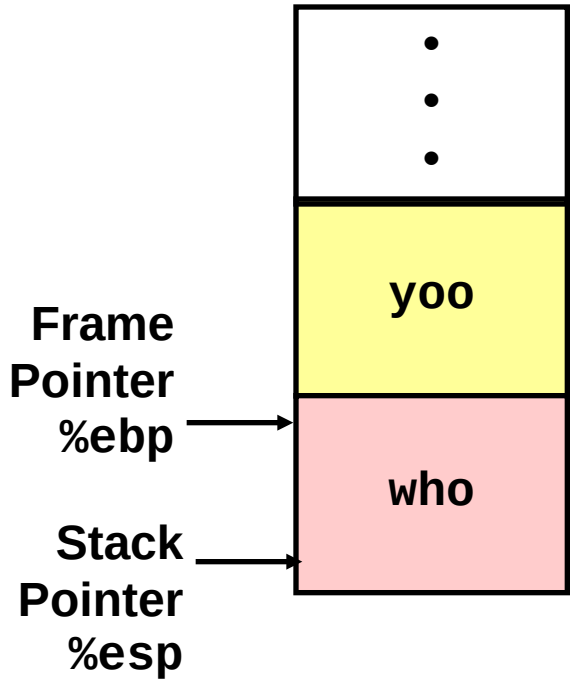
yoo

who

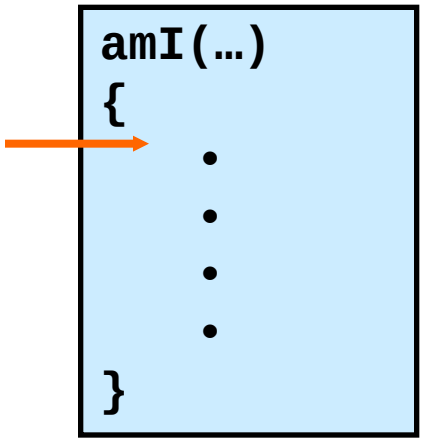
amI

amI

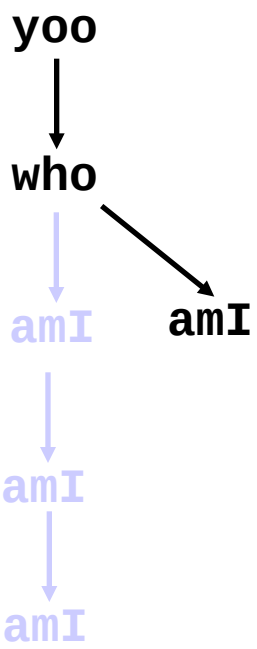
amI

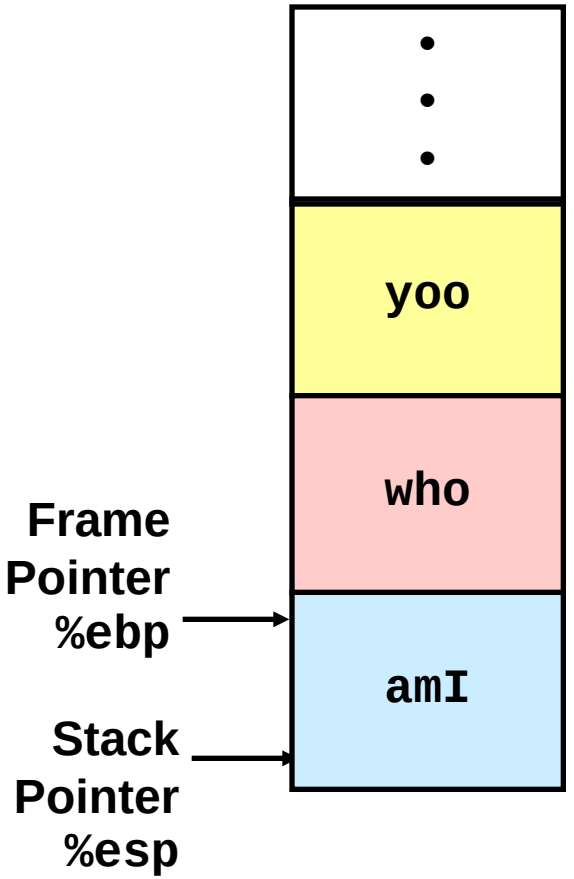


스택의 동작 (9)



Call Chain

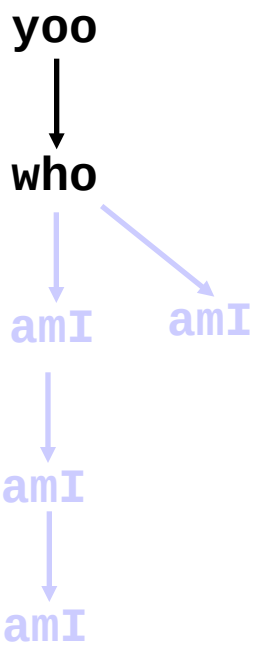


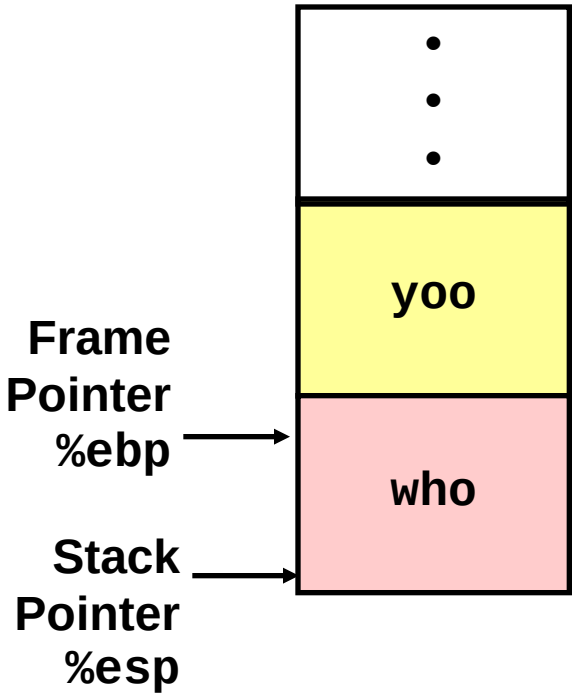


스택의 동작 (10)

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

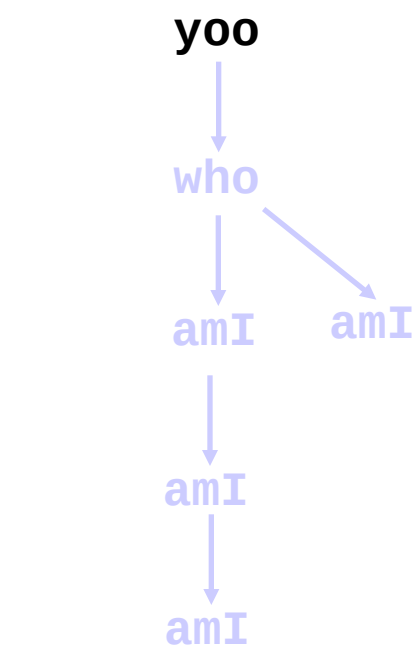


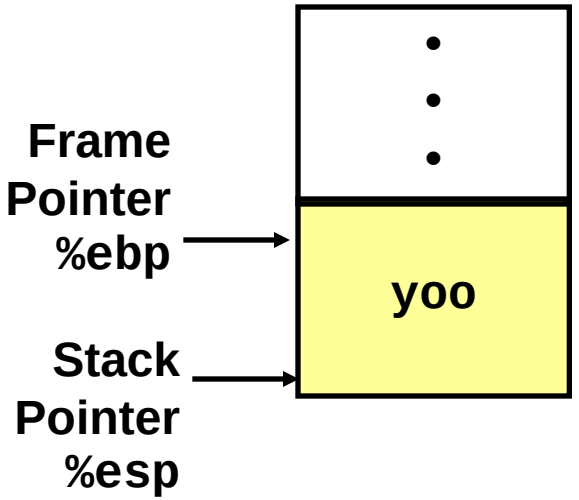


스택의 동작 (11)

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

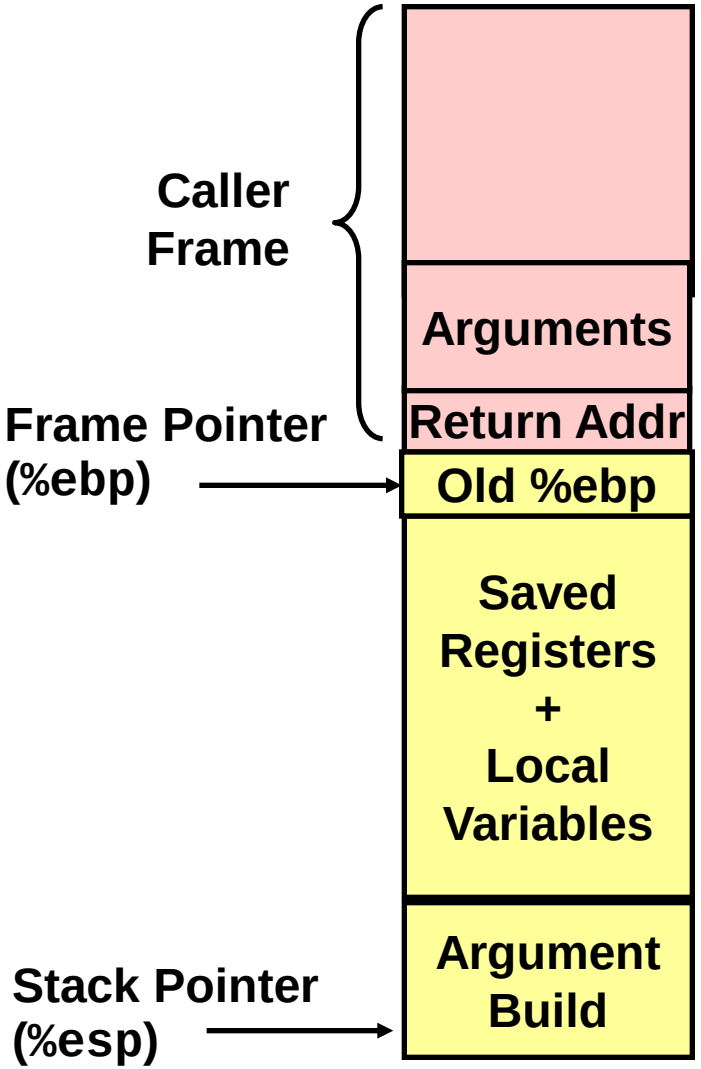
Call Chain





IA32/Linux 스택 프레임

- Current Stack Frame (“Top” to Bottom)
  - “callee”의 매개변수를 저장함
    - “Argument build”
  - 지역변수 Local variables
    - 레지스터로 부족할 때 사용
  - 레지스터 컨텍스트의 저장
  - Old frame pointer
- “Caller” Stack Frame
  - Return address
    - call 명령에 의해 푸쉬됨
  - “callee” 를 위한 매개변수



연습문제 1. call과 리턴

Register	Value
%eax	0x100
%ebx	0x00
%edx	0x3
%esp	0x108
%eip	0x8000

위와 같이 펜티엄의 레지스터들이 값을 저장하고 있을 때, call 0x8300 을 실행했을 때, 실행후 %eip 레지스터와 %esp의 값을 각각 쓰시오.

%eip
%esp

swap 예제 – 다시보기

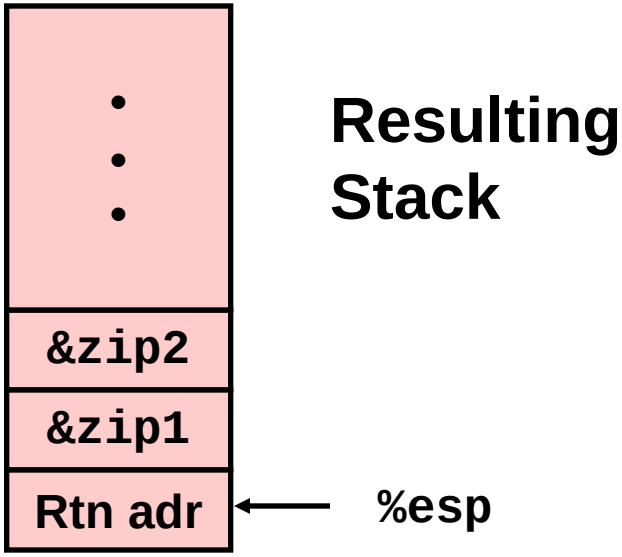
```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Calling swap from `call_swap`

```
call_swap:
    . . .
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    . . .
```



swap 예제 – 다시보기

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

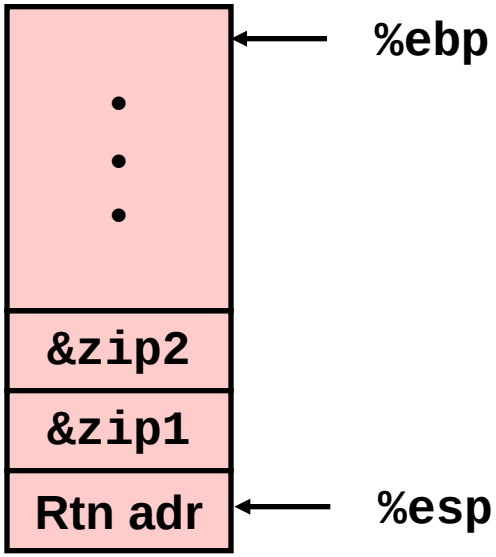
Set Up

Body

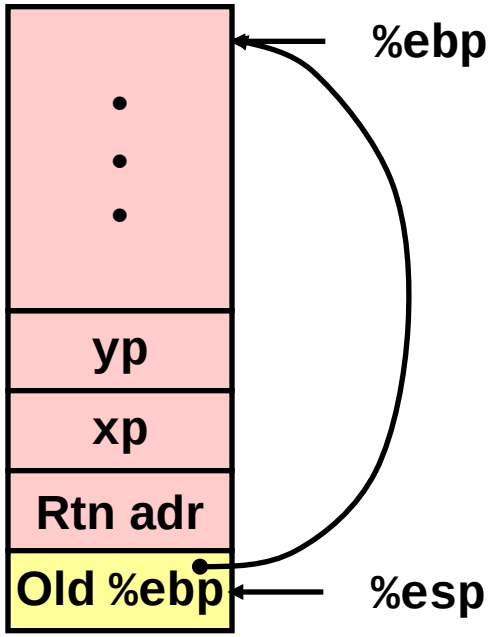
Finish

swap Setup #1

Entering Stack



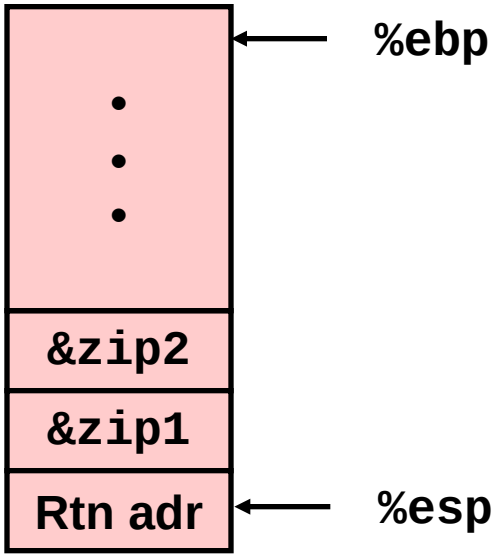
Resulting Stack



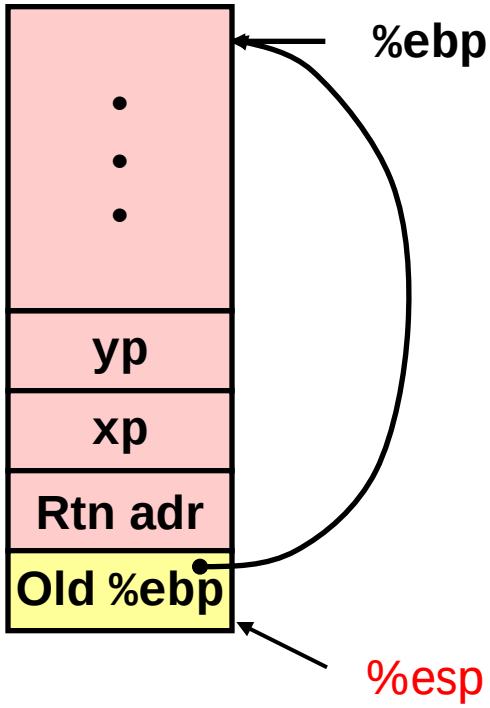

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



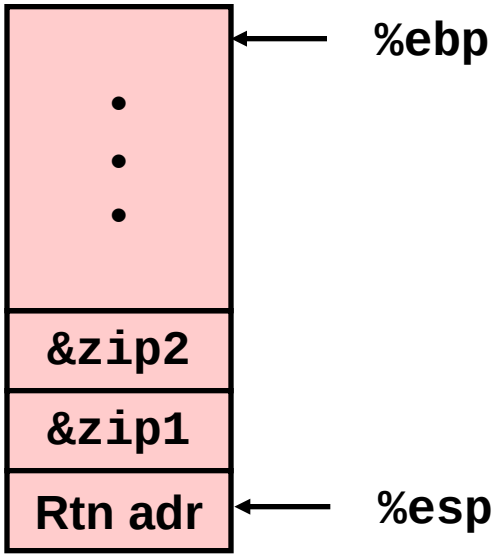
Resulting Stack



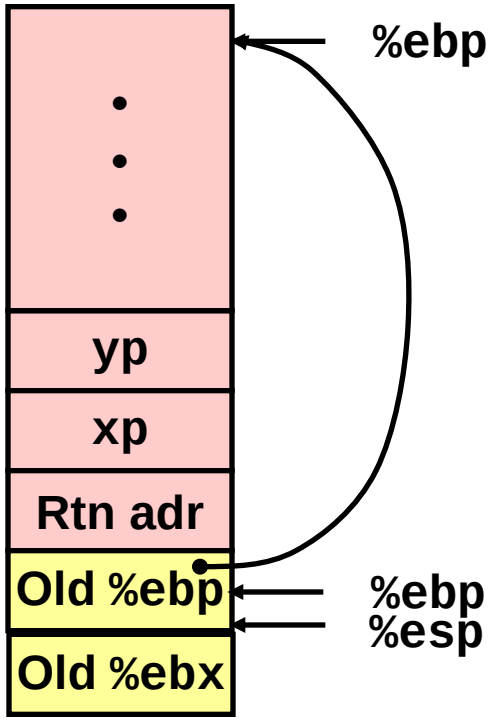

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

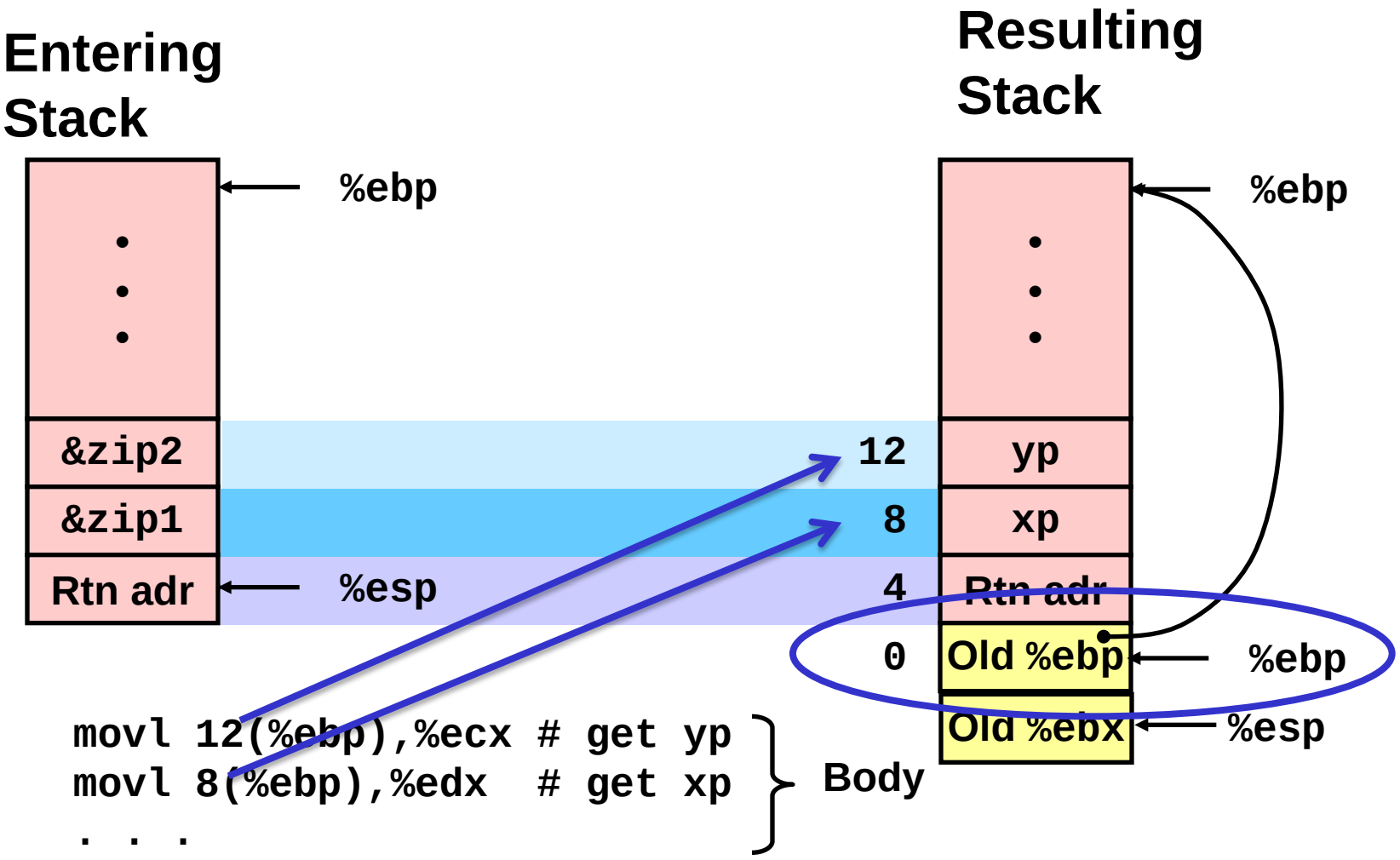


Resulting Stack

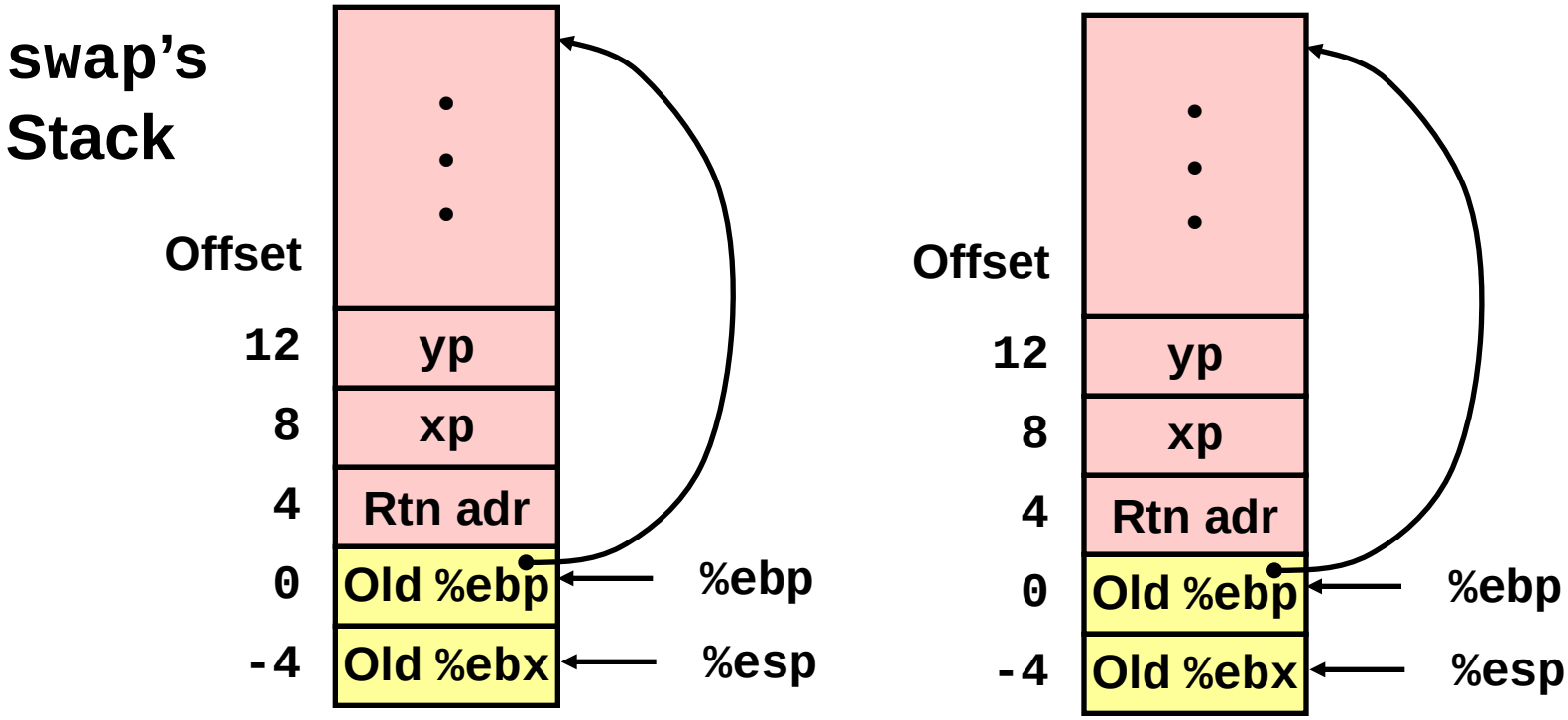



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup 의 결과



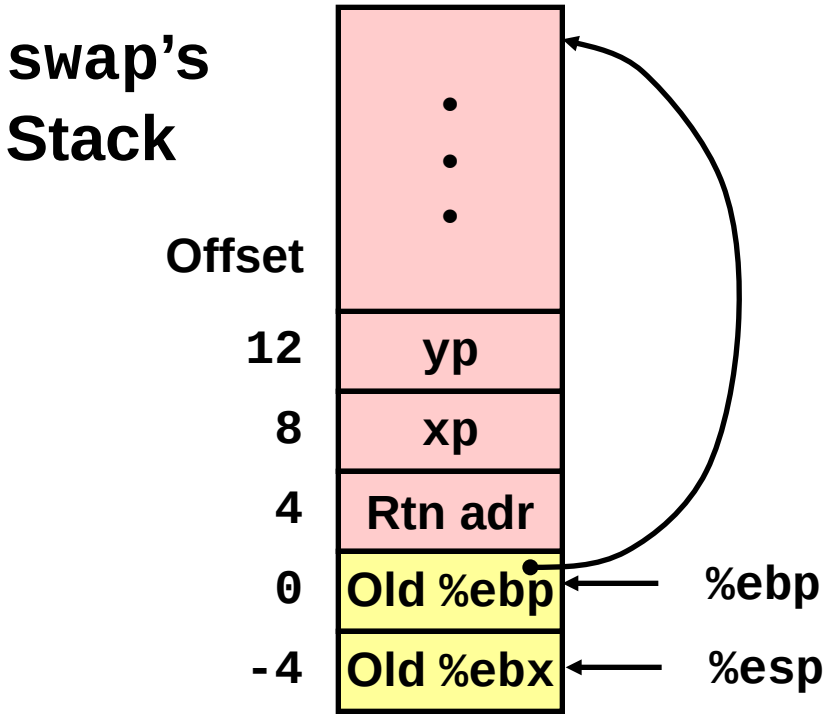
swap Finish #1

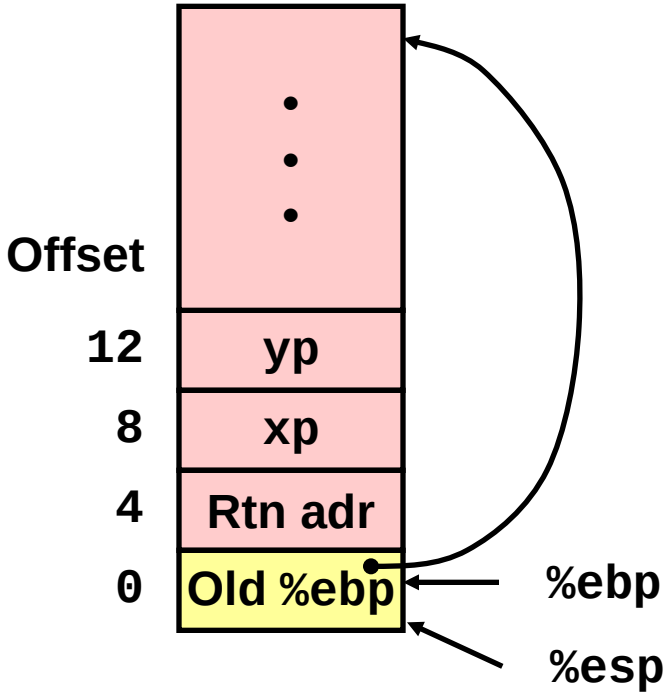


- 관찰 Observation
  - Saved & restored register %ebx

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

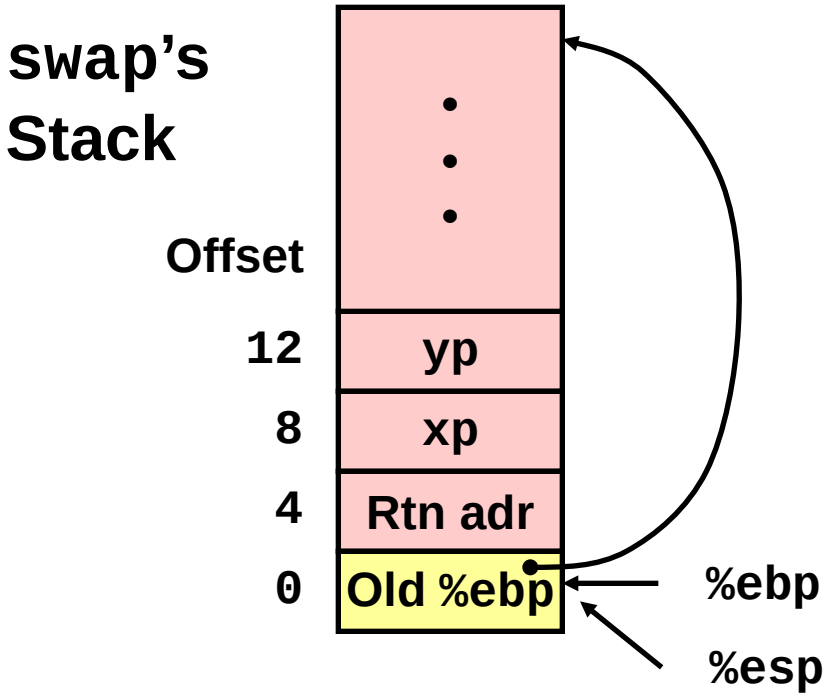
swap Finish #2

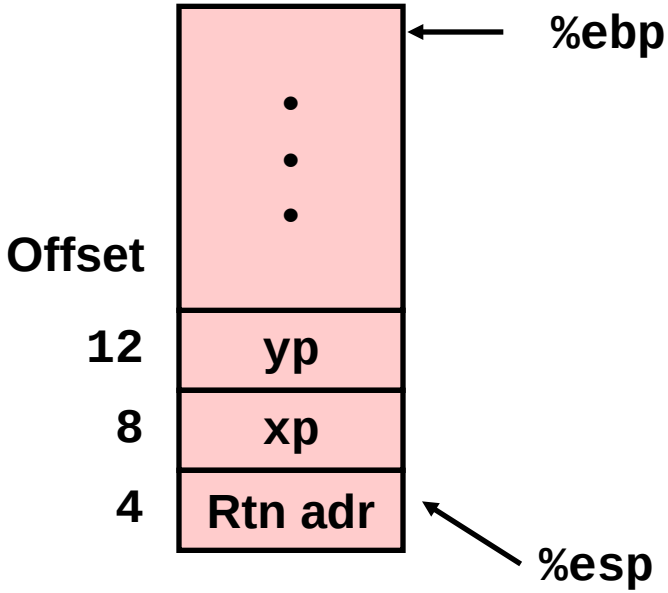





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

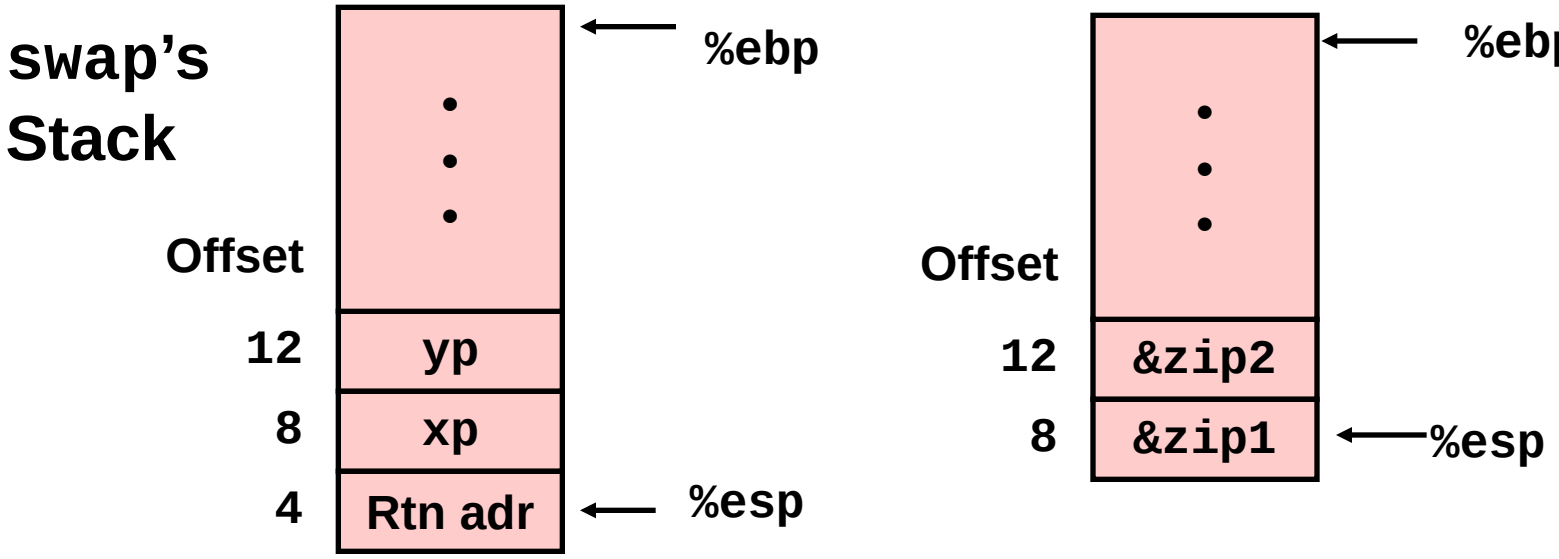
swap Finish #3






```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4



관찰 Observation

- Saved & restored register %ebx
- %eax, %ecx, %edx 는 저장하지 않았음

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

레지스터 저장하기

프로시져 yoo() 가 프로시져 who()를 호출할 때

- yoo() => caller, who() => callee

레지스터를 임시 저장장소로 이용할 수 있을까?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

Q. 위의 프로시져 호출 관계에서 문제를 찾는다면 ?

레지스터 저장하기

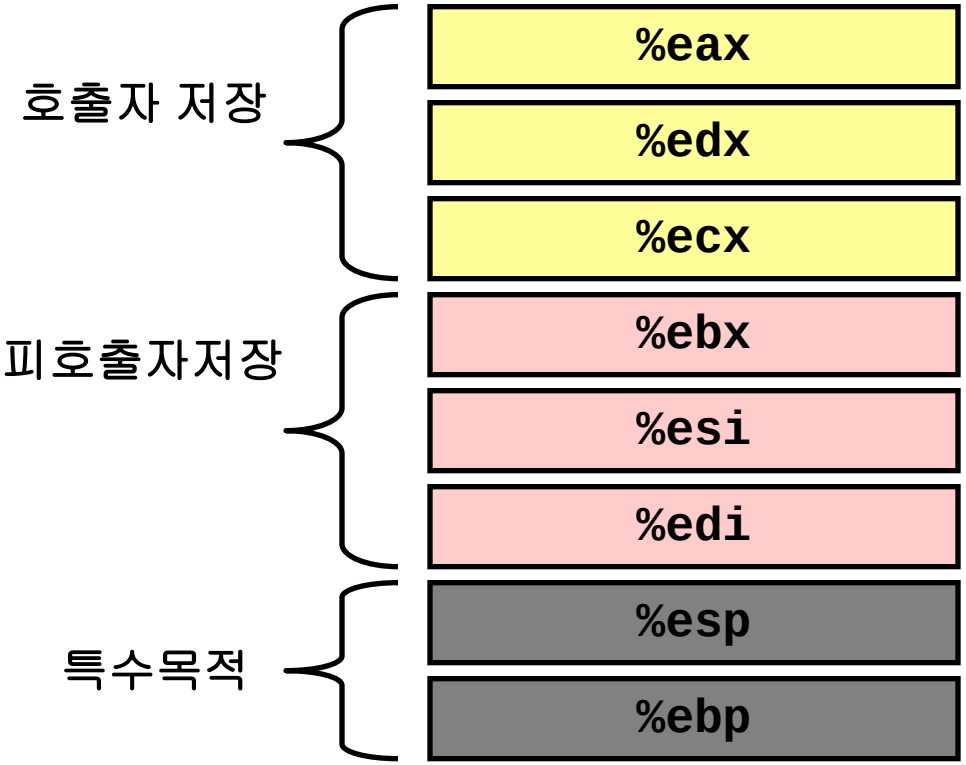
일반적인 규칙

- "Caller Save"
  - 호출하는 프로시져가 임시 레지스터 값들을 자신의 프레임에 프로시져 호출 전에 미리 저장
- "Callee Save"
  - 피호출 프로시져가 임시 레지스터 값들을 사용 전에 저장

IA32/Linux에서의 레지스터 저장

정수 레지스터

- 특수 목적
  - %ebp, %esp
- 피호출자 저장 callee-save
  - %ebx, %esi, %edi
    - 레지스터 사용전에 이전 값은 스택에 저장해 놓는다
- 호출자 저장 caller-save
  - %eax, %edx, %ecx
    - 피호출 함수에서 맘대로 이용하라. 그러나, 다른 피호출 함수도 그렇게 한다는 것을 명심하라
- %eax : 리턴값의 저장



연습문제 2. call 명령과 리턴주소

다음의 코드는 라이브러리 루틴의 컴파일 시에 종종 발견된다.

```
    call next
next:
    popl %eax
```

A. 레지스터 %eax는 어떤 값을 갖게 되는가?
B. 이 call 인스트럭션과 같은 동작을 ret 명령으로 구현할 수 없는 이유를 설명하시오
C. 이 코드는 어떤 유용한 목적으로 사용될 수 있는가?

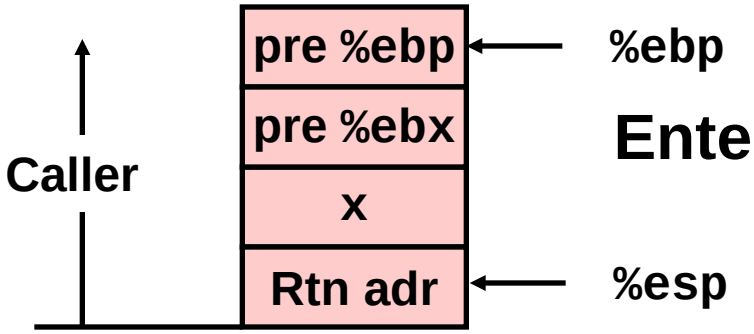
! 의 재귀적 구현

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

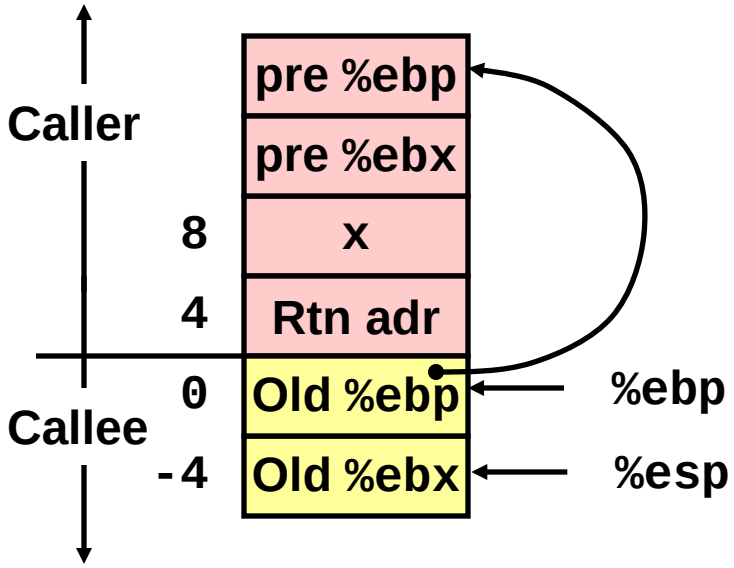
- 레지스터의 사용
  - %eax 는 저장 없이 바로 이용
  - %ebx 는 미리 저장후 리턴전에 복구

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Rfact의 스택 설정




```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```



Rfact Body

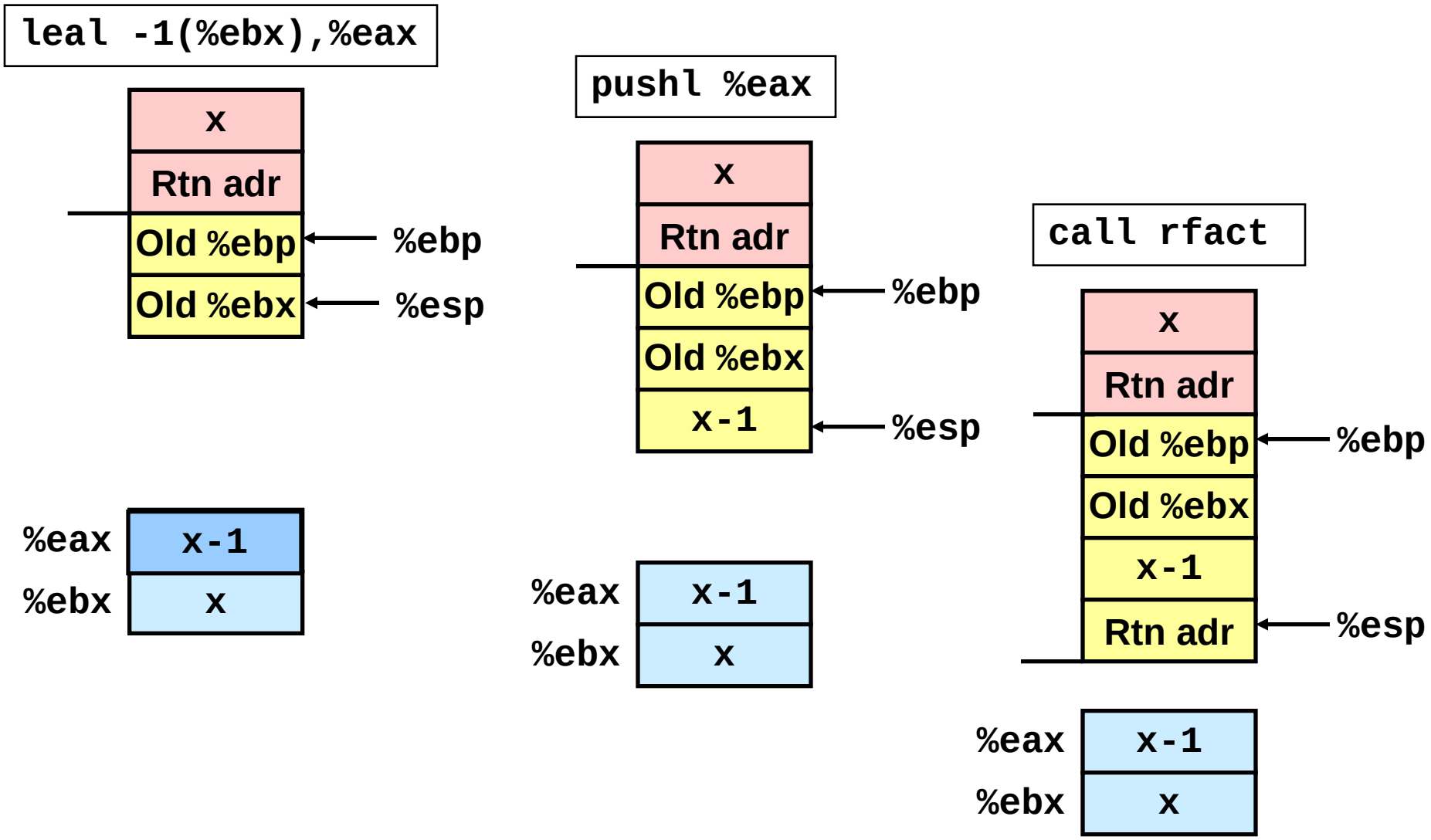
Recursion

```
  movl 8(%ebp),%ebx   # ebx = x
  cmpl $1,%ebx        # Compare x : 1
  jle .L78            # If <= goto Term
  leal -1(%ebx),%eax  # eax = x-1
  pushl %eax          # Push x-1
  call rfact          # rfact(x-1)
  imull %ebx,%eax     # rval * x
  jmp .L79            # Goto done
.L78:               # Term:
  movl $1,%eax        # return val = 1
.L79:               # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

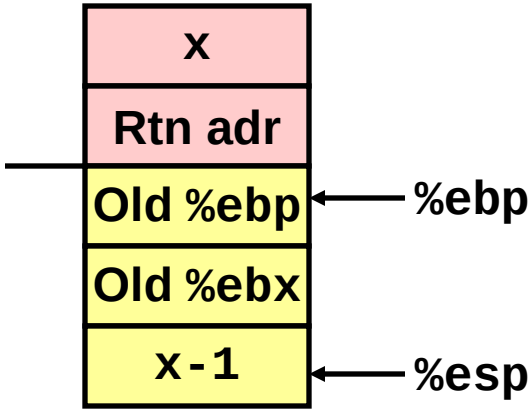
- 레지스터의 이용
  - %ebx 는 x를 저장
  - %eax
    - x-1 의 임시저장
    - rfact(x-1)의 리턴값
    - 이번 호출에서의 리턴값

Rfact Recursion

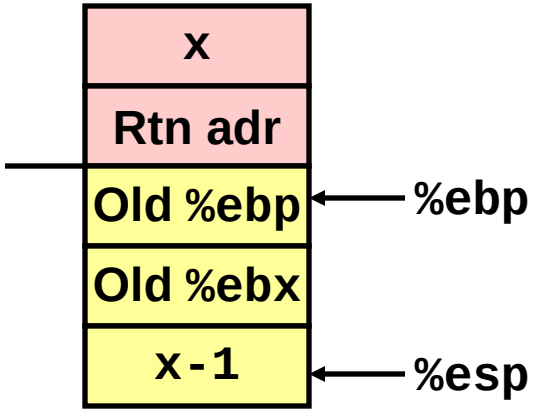


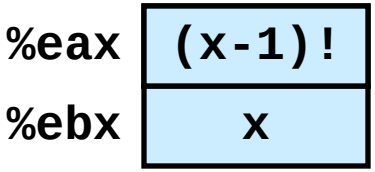
Rfact의 계산

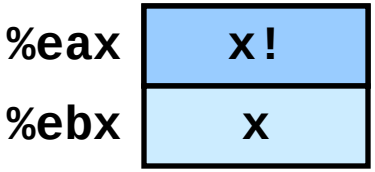
Return from Call



`imull %ebx,%eax`



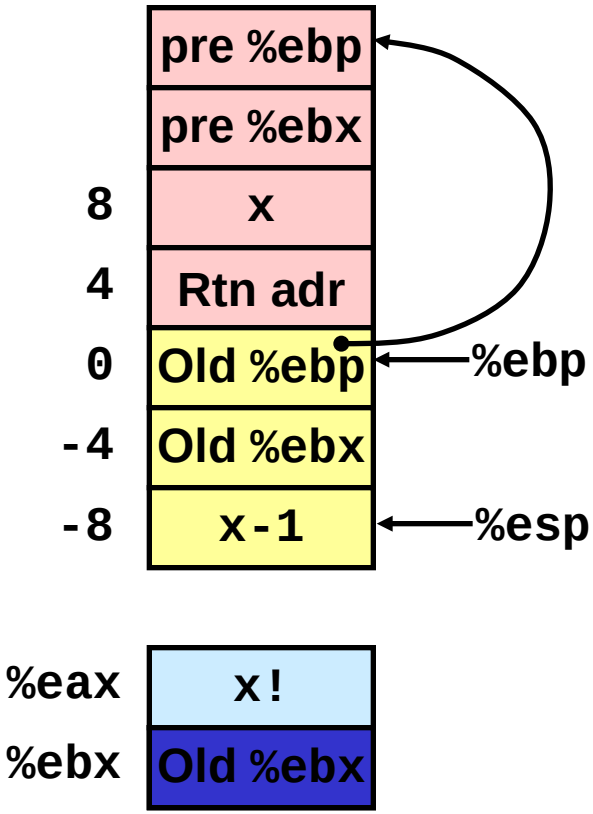


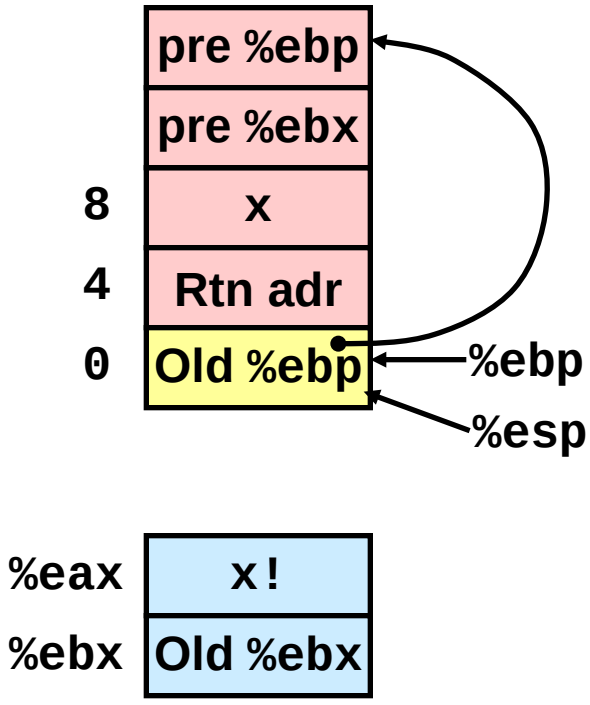


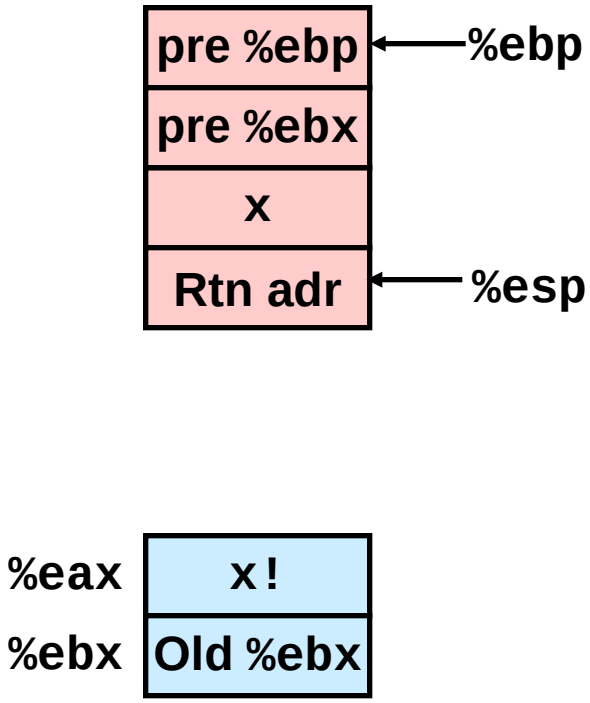
`rfact(x-1)`는 `(x-1)!` 를 레지스터 `%eax`에 리턴한다

Rfact Finish

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```







포인터를 이용한 구현

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

Top-Level Call

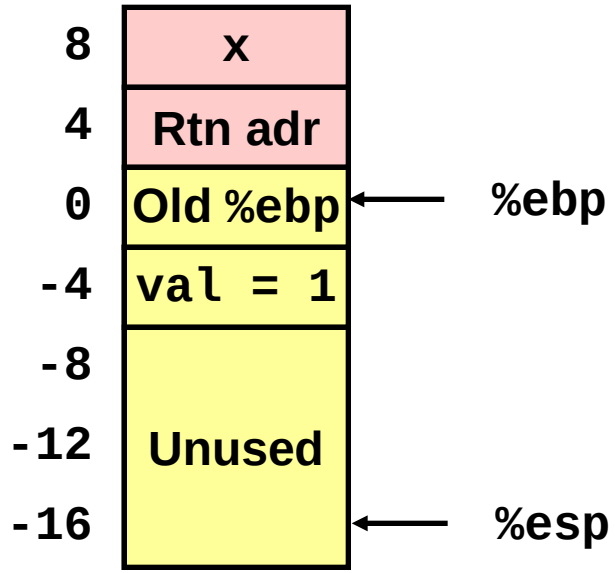
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

결과저장을 위해 포인터를 넘겨준다

포인터의 생성 및 초기화

Initial part of `sfact`

```
_sfact:
   pushl %ebp           # Save %ebp
   movl %esp,%ebp       # Set %ebp
   subl $16,%esp        # Add 16 bytes
   movl 8(%ebp),%edx    # edx = x
   movl $1,-4(%ebp)     # val = 1
```



지역변수를 위해 스택을 이용

- 지역변수 val 을 스택에 저장
  - 포인터를 이용해서 사용
- 포인터는 -4(%ebp)로 계산됨
- 두 번째 매개변수로 스택에 저장

```
int sfact(int x)
{
   int val = 1;
   s_helper(x, &val);
   return val;
}
```

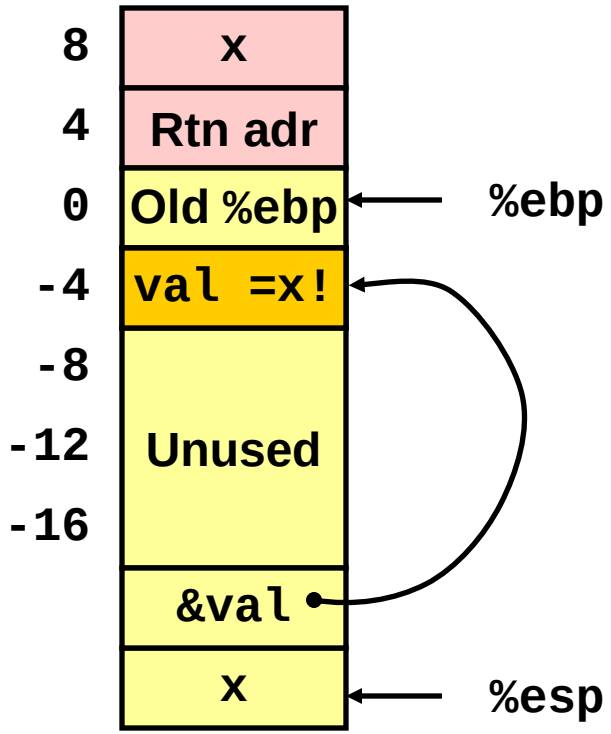
포인터 넘겨주기

Calling s_helper from sfact

```
   leal -4(%ebp),%eax  # Compute &val
   pushl %eax          # Push on stack
   pushl %edx          # Push x
   call s_helper       # call
   movl -4(%ebp),%eax  # Return val
   • • •               # Finish
```

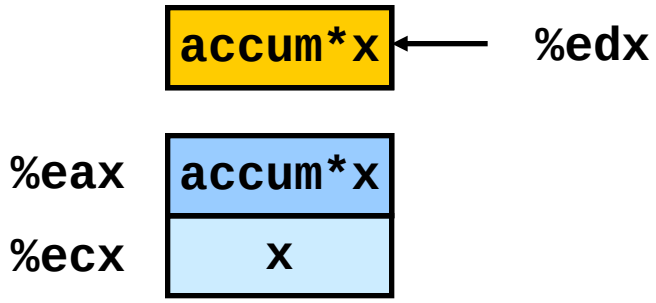
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call



포인터의 사용

```
void s_helper
  (int x, int *accum)
{
  • • •
    int z = *accum * x;
    *accum = z;
  • • •
}
```




```
  • • •
  movl %ecx,%eax     # z = x
  imull (%edx),%eax  # z *= *accum
  movl %eax,(%edx)   # *accum = z
  • • •
```

%ecx는 변수 x를 저장하고 있음

%edx는 accum의 포인터를 저장하고 있음

- (%edx) 를 이용하여 메모리를 액세스 함

연습문제 3. 프로시져호출

C 함수 fun이 다음과 같은 코드 본체와 IA32코드를 갖는다.

```
*p = d;
return x-c;
```

```
1    movsbl   12(%ebp),%edx
2    movl     16(%ebp), %eax
3    movl     %edx, (%eax)
4    movswl   8(%ebp),%eax
5    movl     20(%ebp), %edx
6    subl     %eax, %edx
7    movl     %edx, %eax
```

함수 fun의 프로토타입을 작성하고, 인자 p, d, x, c의 자료형과 순서를 보이시오.

p: d:

x: c: