



EDDI

Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv1 과정

제 4기

2022. 10. 02

진동민

학습목표 & 4회차 날짜

학습목표

- while문이 for문과 구조적으로 같음
- 생성된 스택이 확장될 수 있음
- Calling Convention 이해하기
- 스택 완충 공간 간단히 이해하기
- 지저분한 switch문 대신 함수포인터 응용 기법 간단히 살펴보기
- if문 대신 switch문을 사용하자
- goto문을 사용하자
- 'AND NOT' 기법 원리 이해만 하기

수업 날짜

2022-09-17 (토) 오후 6시~9시

- 1) while.c 어셈블리어 간단 분석 (*while.c*)
- 2) 스택은 16바이트만 생성하는가? (*stack_size.c*)
 - 2-2) Calling Convention (*param.c*)
 - 2-3) 스택 완충공간 (참고)
- 3) switch문 (*switch.c*)
 - 3-2) 가독성 측면의 함수포인터 응용 기법 (*GitHub* 참고)
- 4) if문의 약점은 순서 (*if_vs_switch.c*)
- 5) 중첩 반복문에서 효율적인 goto문 (*why_goto.c, super_goto.c*)
- 6) 'AND NOT 기법' 간단하게 살펴보기 (*and_not.c*)

while.c 어셈블리어 간단 분석

1) while.c

```
1 #include <stdio.h>
2
3 #define MAX 10
4
5 int main(void)
6 {
7     int i = 0;
8
9     while (i < MAX)
10    {
11        printf("i = %d\n", i++);
12    }
13
14    return 0;
15 }
```

while.c 어셈블리어 간단 분석

2) while.c의 어셈블리어

```
(gdb) disas
Dump of assembler code for function main:
=> 0x000055555555149 <+0>:      endbr64
    0x00005555555514d <+4>:      push   %rbp
    0x00005555555514e <+5>:      mov    %rsp,%rbp
    0x000055555555151 <+8>:      sub    $0x10,%rsp
    0x000055555555155 <+12>:     movl   $0x0,-0x4(%rbp)
    0x00005555555515c <+19>:     jmp    0x5555555517a <main+49>
    0x00005555555515e <+21>:     mov    -0x4(%rbp),%eax
    0x000055555555161 <+24>:     lea    0x1(%rax),%edx
    0x000055555555164 <+27>:     mov    %edx,-0x4(%rbp)
    0x000055555555167 <+30>:     mov    %eax,%esi
    0x000055555555169 <+32>:     lea    0xe94(%rip),%rdi      # 0x555555556004
    0x000055555555170 <+39>:     mov    $0x0,%eax
    0x000055555555175 <+44>:     callq  0x55555555050 <printf@plt>
    0x00005555555517a <+49>:     cmpl   $0x9,-0x4(%rbp)
    0x00005555555517e <+53>:     jle    0x5555555515e <main+21>
    0x000055555555180 <+55>:     mov    $0x0,%eax
    0x000055555555185 <+60>:     leaveq
    0x000055555555186 <+61>:     retq
End of assembler dump.
```

while.c 어셈블리어 간단 분석

3) 정리

앞의 슬라이드에서 while문의 어셈블리어를 간단하게 살펴본 결과, 3회차 숙제에서 분석했던 for문의 어셈블리어와 구조적으로 다르지 않음을 알 수 있다.

스택은 16바이트만 생성하는가?

정확히 잘 기억나지 않지만 아마 기억상으론 수업중에 while.c 실습을 하다가 어느 수강생이 어셈블리어를 보고 “스택은 16바이트만 생성되나요?”라고 질문 했었던 것 같다.

```
(gdb) disas
Dump of assembler code for function main:
=> 0x000055555555149 <+0>:      endbr64
    0x00005555555514d <+4>:      push   %rbp
    0x00005555555514e <+5>:      mov    %rsp,%rbp
    0x000055555555151 <+8>:      sub    $0x10,%rsp    ← 16바이트 스택 생성
    0x000055555555155 <+12>:     movl   $0x0,-0x4(%rbp)
    0x00005555555515c <+19>:     jmp    0x5555555517a <main+49>
```

while.c의 어셈블리어 일부

그래서 강사님이 while.c 파일에 변수를 더 선언하고 컴파일하여 gdb로 다시 어셈블리어를 보여주셨다. 여기서는 수정한 while.c 파일을 stack_size.c로 변경하겠다.

스택은 16바이트만 생성하는가?

1) 변수를 추가로 선언한 stack_size.c

```
1 #include <stdio.h>
2
3 #define MAX 10
4
5 int main(void)
6 {
7     int i = 0;
8     int a = 2;
9     float b = 3.2f;
10    double c = 8.8, d = 2.2;
11
12    while (i < MAX)
13    {
14        printf("i = %d\n", i++);
15    }
16
17    return 0;
18 }
```


스택은 16바이트만 생성하는가?

2) stack_size.c의 어셈블리어 일부

```
(gdb) disas
Deleted breakpoint 1 Dump of assembler code for function main:
=> 0x000055555555149 <+0>:      endbr64
   0x00005555555514d <+4>:      push   %rbp
   0x00005555555514e <+5>:      mov    %rsp,%rbp
   0x000055555555151 <+8>:      sub    $0x20,%rsp      ← 32바이트 스택 생성
   0x000055555555155 <+12>:     movl   $0x0,-0x1c(%rbp)
   0x00005555555515c <+19>:     movl   $0x2,-0x18(%rbp)
   0x000055555555163 <+26>:     movss  0xea5(%rip),%xmm0      # 0x555555556010
   0x00005555555516b <+34>:     movss  %xmm0,-0x14(%rbp)
   0x000055555555170 <+39>:     movsd  0xea0(%rip),%xmm0      # 0x555555556018
   0x000055555555178 <+47>:     movsd  %xmm0,-0x10(%rbp)
   0x00005555555517d <+52>:     movsd  0xe9b(%rip),%xmm0      # 0x555555556020
   0x000055555555185 <+60>:     movsd  %xmm0,-0x8(%rbp)
   0x00005555555518a <+65>:     jmp    0x555555551a8 <main+95>
```

설명

int형 변수 2개, float형 변수 1개, double형 변수 2개이므로 총 28바이트이다.
실제로는 32바이트 크기의 스택을 생성하였고, 그 스택 중 28바이트는 변수를 저장하였다.

스택은 16바이트만 생성하는가?

3) 결론

변수를 추가하고 어셈블리어를 살펴본 결과 스택은 16바이트 보다 더 큰 크기로 생성될 수 있는 것을 확인했다. 그러므로 “스택은 16바이트만 생성하는가”의 답은 “No”이다.

4) 정리

스택 크기는 포인터 크기인 8(바이트)의 배수로 할당된다. **(이 부분은 개인적으로 추가 학습이 필요하다)**

Calling Convention

1) 스택 생성이 언급된 김에 Calling Convention(함수 호출 규약)을 배웠다. 아래 코드를 param.c로 저장한다.

```
1 #include <stdio.h>
2
3 // 매개변수 4개
4 int func(int num1, int num2, int num3, int num4)
5 {
6     return num1 + num2 + num3 + num4;
7 }
8
9 // 매개변수 9개
10 int func2(int num1, int num2, int num3, int num4, int num5, int num6, int num7, int num8, int num9)
11 {
12     return num1 + num2 + num3 + num4 + num5 + num6 + num7 + num8 + num9;
13 }
14
15 int main(void)
16 {
17     // 총 int형 변수 10개
18     int num = 2, num2 = 3, num3 = 4, num4 = 5, num5 = 6;
19     int num6 = 7, num7 = 8, num8 = 9, num9 = 10, res;
20
21     res = func(num, num2, num3, num4);
22     printf("res = %d\n", res); // 14
23
24     res = func2(num, num2, num3, num4, num5, num6, num7, num8, num9);
25     printf("res = %d\n", res); // 54
26
27     return 0;
28 }
```

Calling Convention

2) Calling Convention 설명

- 리눅스 운영체제에 요청하는 시스템콜 (system call)
 - system call 번호를 저장하는 레지스터
 - x86 또는 x64 Architecture: ax 레지스터
 - ARM Architecture: r7 레지스터
- 일반함수를 호출하는 call의 경우 사용하는 레지스터 개수
 - 32bit의 경우 레지스터 4개
 - 64bit의 경우 레지스터 6개 (r9, r8, rcx, rdx, rsi, rax)
 - 그 이상으로는 전부 stack에 배치

(참고)

- ARM Architecture에서 프로그래밍 할 때 성능을 높이하고자 한다면 되도록 인자를 4개로 맞춰 레지스터를 사용하는 것이 좋다.
- 또한, Calling Convention이 CPU Architecture에 의존적이라는 부분을 주의해야 한다.

출처-1: <https://blog.naver.com/eddi2021/222468893530>

출처-2: <https://cafe.naver.com/eddicorp/971>

∴ 최적화가 필요할 때는 **Calling Convention**을 **생각하며** 매개변수 선언에 주의하자!

Calling Convention

3) param.c의 어셈블리어

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555551ba <+0>:      endbr64
0x0000555555551be <+4>:      push    %rbp
0x0000555555551bf <+5>:      mov     %rsp,%rbp
0x0000555555551c2 <+8>:      sub     $0x30,%rsp
0x0000555555551c6 <+12>:     movl    $0x2,-0x28(%rbp)
0x0000555555551cd <+19>:     movl    $0x3,-0x24(%rbp)
0x0000555555551d4 <+26>:     movl    $0x4,-0x20(%rbp)
0x0000555555551db <+33>:     movl    $0x5,-0x1c(%rbp)
0x0000555555551e2 <+40>:     movl    $0x6,-0x18(%rbp)
0x0000555555551e9 <+47>:     movl    $0x7,-0x14(%rbp)
0x0000555555551f0 <+54>:     movl    $0x8,-0x10(%rbp)
0x0000555555551f7 <+61>:     movl    $0x9,-0xc(%rbp)
0x0000555555551fe <+68>:     movl    $0xa,-0x8(%rbp)
0x000055555555205 <+75>:     mov     -0x1c(%rbp),%ecx
0x000055555555208 <+78>:     mov     -0x20(%rbp),%edx
0x00005555555520b <+81>:     mov     -0x24(%rbp),%esi
0x00005555555520e <+84>:     mov     -0x28(%rbp),%eax
0x000055555555211 <+87>:     mov     %eax,%edi
0x000055555555213 <+89>:     callq  0x55555555149 <func>
0x000055555555218 <+94>:     mov     %eax,-0x4(%rbp)
0x00005555555521b <+97>:     mov     -0x4(%rbp),%eax
0x00005555555521e <+100>:    mov     %eax,%esi
0x000055555555220 <+102>:    lea     0xdd(%rip),%rdi      # 0x55555556004
0x000055555555227 <+109>:    mov     $0x0,%eax
0x00005555555522c <+114>:    callq  0x55555555050 <printf@plt>
0x000055555555231 <+119>:    mov     -0x14(%rbp),%r9d
0x000055555555235 <+123>:    mov     -0x18(%rbp),%r8d
0x000055555555239 <+127>:    mov     -0x1c(%rbp),%ecx
0x00005555555523c <+130>:    mov     -0x20(%rbp),%edx
0x00005555555523f <+133>:    mov     -0x24(%rbp),%esi
0x000055555555242 <+136>:    mov     -0x28(%rbp),%eax
0x000055555555245 <+139>:    sub     $0x8,%rsp
```

```
0x000055555555249 <+143>:    mov     -0x8(%rbp),%edi
0x00005555555524c <+146>:    push    %rdi
0x00005555555524d <+147>:    mov     -0xc(%rbp),%edi
0x000055555555250 <+150>:    push    %rdi
0x000055555555251 <+151>:    mov     -0x10(%rbp),%edi
0x000055555555254 <+154>:    push    %rdi
0x000055555555255 <+155>:    mov     %eax,%edi
0x000055555555257 <+157>:    callq  0x55555555171 <func2>
0x00005555555525c <+162>:    add     $0x20,%rsp
0x000055555555260 <+166>:    mov     %eax,-0x4(%rbp)
0x000055555555263 <+169>:    mov     -0x4(%rbp),%eax
0x000055555555266 <+172>:    mov     %eax,%esi
0x000055555555268 <+174>:    lea     0xd95(%rip),%rdi      # 0x55555556004
0x00005555555526f <+181>:    mov     $0x0,%eax
0x000055555555274 <+186>:    callq  0x55555555050 <printf@plt>
0x000055555555279 <+191>:    mov     $0x0,%eax
0x00005555555527e <+196>:    leaveq  %eax
0x00005555555527f <+197>:    retq
End of assembler dump.
```

(참고) 어셈블리어를 분석하는 환경

- CPU: **AMD** 라이젠5 PRO 4650G
- 메인보드: ASUS PRIME B550M-A
- 메모리: 삼성전자 DDR4-3200 (8GB) * 2개

```
try@try-desktop:~$ uname -m
x86_64
```

64비트라는 것이 더 중요

Calling Convention

4) param.c의 어셈블리어 간단하게 분석-1

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000555555551ba <+0>:      endbr64
   0x0000555555551be <+4>:      push    %rbp
   0x0000555555551bf <+5>:      mov     %rsp,%rbp
   0x0000555555551c2 <+8>:      sub     $0x30,%rsp ← 48바이트 스택 생성
   0x0000555555551c6 <+12>:     movl    $0x2,-0x28(%rbp)
   0x0000555555551cd <+19>:     movl    $0x3,-0x24(%rbp) ← 변수 할당
   0x0000555555551d4 <+26>:     movl    $0x4,-0x20(%rbp)
   0x0000555555551db <+33>:     movl    $0x5,-0x1c(%rbp)
   0x0000555555551e2 <+40>:     movl    $0x6,-0x18(%rbp)
   0x0000555555551e9 <+47>:     movl    $0x7,-0x14(%rbp)
   0x0000555555551f0 <+54>:     movl    $0x8,-0x10(%rbp)
   0x0000555555551f7 <+61>:     movl    $0x9,-0xc(%rbp)
   0x0000555555551fe <+68>:     movl    $0xa,-0x8(%rbp)
   0x000055555555205 <+75>:     mov     -0x1c(%rbp),%ecx
   0x000055555555208 <+78>:     mov     -0x20(%rbp),%edx ← func 함수 호출 직전에 매개변수를 레지스터로 복사
   0x00005555555520b <+81>:     mov     -0x24(%rbp),%esi
   0x00005555555520e <+84>:     mov     -0x28(%rbp),%eax
   0x000055555555211 <+87>:     mov     %eax,%edi
   0x000055555555213 <+89>:     callq  0x55555555149 <func> ← func 함수 호출
   0x000055555555218 <+94>:     mov     %eax,-0x4(%rbp)
   0x00005555555521b <+97>:     mov     -0x4(%rbp),%eax
   0x00005555555521e <+100>:    mov     %eax,%esi
   0x000055555555220 <+102>:    lea     0xdd(%rip),%rdi      # 0x555555556004
   0x000055555555227 <+109>:    mov     $0x0,%eax
   0x00005555555522c <+114>:    callq  0x55555555050 <printf@plt>
```

Calling Convention

4) param.c의 어셈블리어 간단하게 분석-2

```
0x000055555555231 <+119>: mov    -0x14(%rbp),%r9d
0x000055555555235 <+123>: mov    -0x18(%rbp),%r8d
0x000055555555239 <+127>: mov    -0x1c(%rbp),%ecx
0x00005555555523c <+130>: mov    -0x20(%rbp),%edx
0x00005555555523f <+133>: mov    -0x24(%rbp),%esi
0x000055555555242 <+136>: mov    -0x28(%rbp),%eax
0x000055555555245 <+139>: sub    $0x8,%rsp
0x000055555555249 <+143>: mov    -0x8(%rbp),%edi
0x00005555555524c <+146>: push   %rdi
0x00005555555524d <+147>: mov    -0xc(%rbp),%edi
0x000055555555250 <+150>: push   %rdi
0x000055555555251 <+151>: mov    -0x10(%rbp),%edi
0x000055555555254 <+154>: push   %rdi
0x000055555555255 <+155>: mov    %eax,%edi
0x000055555555257 <+157>: callq  0x55555555171 <func2>
0x00005555555525c <+162>: add    $0x20,%rsp
0x000055555555260 <+166>: mov    %eax,-0x4(%rbp)
0x000055555555263 <+169>: mov    -0x4(%rbp),%eax
0x000055555555266 <+172>: mov    %eax,%esi
0x000055555555268 <+174>: lea    0xd95(%rip),%rdi    # 0x555555556004
0x00005555555526f <+181>: mov    $0x0,%eax
0x000055555555274 <+186>: callq  0x55555555050 <printf@plt>
0x000055555555279 <+191>: mov    $0x0,%eax
0x00005555555527e <+196>: leaveq
0x00005555555527f <+197>: retq
End of assembler dump.
```

← func2 함수 호출 직전에 매개변수 9개 중 6개를 레지스터로 복사

← 나머지 매개변수는 스택에 배치 ★

← func2 함수 호출

이를 처음부터 그림으로 한 번 보자

Calling Convention

main 함수의 지역변수(num~num9) 할당까지만 그림으로 보여줌

메모리

bp

10 (num9)
9 (num8)
8 (num7)
7 (num6)
6 (num5)
5 (num4)
4 (num3)
3 (num2)
2 (num)

bp - 0x8

bp - 0x10

bp - 0x18

bp - 0x20

bp - 0x28

bp - 0x30: sp (여기까지가 main 함수 스택 끝 부분)

한 칸이
8바이트

설명

main 함수는 초반에 48바이트 크기의 스택을 생성함

(참고) 지역변수를 할당할 때는 스택의 아래부터 위로 채움

Calling Convention

4) param.c의 어셈블리어 간단하게 분석-2

```
0x000055555555231 <+119>: mov    -0x14(%rbp),%r9d
0x000055555555235 <+123>: mov    -0x18(%rbp),%r8d
0x000055555555239 <+127>: mov    -0x1c(%rbp),%ecx
0x00005555555523c <+130>: mov    -0x20(%rbp),%edx
0x00005555555523f <+133>: mov    -0x24(%rbp),%esi
0x000055555555242 <+136>: mov    -0x28(%rbp),%eax
0x000055555555245 <+139>: sub    $0x8,%rsp
0x000055555555249 <+143>: mov    -0x8(%rbp),%edi
0x00005555555524c <+146>: push   %rdi
0x00005555555524d <+147>: mov    -0xc(%rbp),%edi
0x000055555555250 <+150>: push   %rdi
0x000055555555251 <+151>: mov    -0x10(%rbp),%edi
0x000055555555254 <+154>: push   %rdi
0x000055555555255 <+155>: mov    %eax,%edi
0x000055555555257 <+157>: callq  0x55555555171 <func2>
0x00005555555525c <+162>: add    $0x20,%rsp
0x000055555555260 <+166>: mov    %eax,-0x4(%rbp)
0x000055555555263 <+169>: mov    -0x4(%rbp),%eax
0x000055555555266 <+172>: mov    %eax,%esi
0x000055555555268 <+174>: lea    0xd95(%rip),%rdi    # 0x555555556004
0x00005555555526f <+181>: mov    $0x0,%eax
0x000055555555274 <+186>: callq  0x55555555050 <printf@plt>
0x000055555555279 <+191>: mov    $0x0,%eax
0x00005555555527e <+196>: leaveq
0x00005555555527f <+197>: retq
End of assembler dump.
```

← 레지스터로 복사

← 스택 확장 후 배치 ★

설명

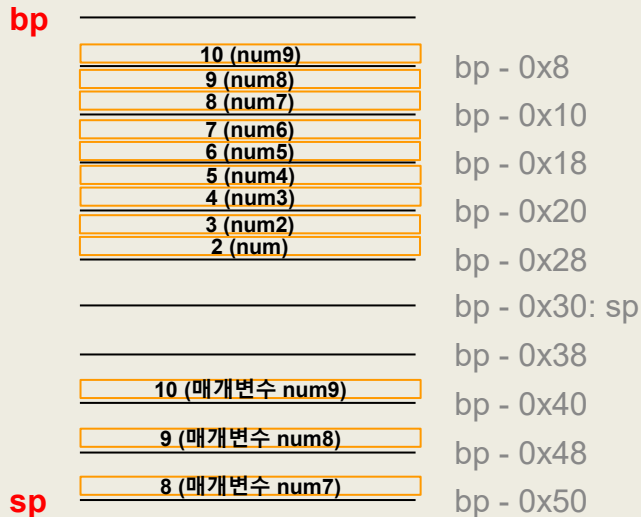
func2 함수 호출 직전에 매개변수를 9개 중에서 6개만 레지스터에 담았기 때문에, 나머지 3개의 매개변수는 **main 함수 스택을 확장한 후 배치**

앞의 슬라이드에 이어 그려보자

Calling Convention

앞의 슬라이드에서 func2 함수 호출 직전에 매개변수를 복사한 상황을 그림으로 보여줌

메모리



설명

- `sub $0x8, rsp`
 - 스택을 8바이트 추가 확장
- `mov`와 `push` 명령어
 - 나머지 매개변수는 main 함수의 스택을 확장하여 쌓음

r9	7 (매개변수 num6)
r8	6 (매개변수 num5)
rcx	5 (매개변수 num4)
rdx	4 (매개변수 num3)
rsi	3 (매개변수 num2)
rax	2 (매개변수 num1)

Calling Convention

(참고) 나머지 매개변수를 스택에 쌓을 때의 면밀한 분석 설명

```
0x000055555555231 <+119>: mov    -0x14(%rbp),%r9d
0x000055555555235 <+123>: mov    -0x18(%rbp),%r8d
0x000055555555239 <+127>: mov    -0x1c(%rbp),%ecx
0x00005555555523c <+130>: mov    -0x20(%rbp),%edx
0x00005555555523f <+133>: mov    -0x24(%rbp),%esi
0x000055555555242 <+136>: mov    -0x28(%rbp),%eax
0x000055555555245 <+139>: sub    $0x8,%rsp
0x000055555555249 <+143>: mov    -0x8(%rbp),%edi
0x00005555555524c <+146>: push   %rdi
0x00005555555524d <+147>: mov    -0xc(%rbp),%edi
0x000055555555250 <+150>: push   %rdi
0x000055555555251 <+151>: mov    -0x10(%rbp),%edi
0x000055555555254 <+154>: push   %rdi
0x000055555555255 <+155>: mov    %eax,%edi
0x000055555555257 <+157>: callq 0x55555555171 <func2>
```

- 나머지 매개변수를 push 명령어로 값을 스택에 쌓는 상황이다.
- 매개변수의 크기는 int형 크기의 4바이트이므로 그 크기만큼 스택에 저장할 줄 알았으나, 자세히 분석한 결과 8바이트 단위로 저장하고 있음을 알 수 있다.
- 그러므로 다음과 같이 나타내는 것이 정확하다.

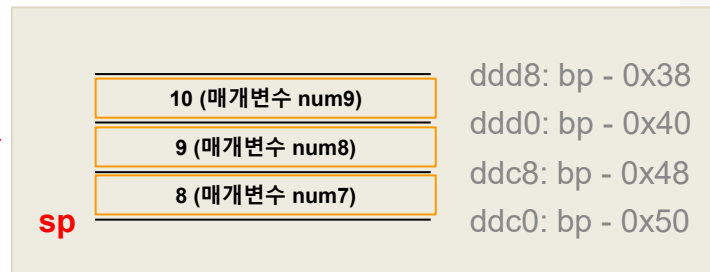
```
(gdb) x/gd $rsp-8
0x7fffffffddd0: 140737488346640
(gdb) si
0x00005555555524d      24
(gdb) x/gd $rsp
0x7fffffffddd0: 10
```

매개변수 num9를 스택에 쌓는 상황

← push %rdi 실행

이걸 그리면

메모리



스택 완충 공간 (참고)

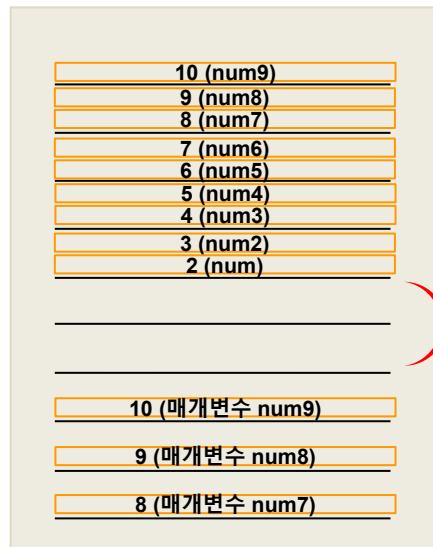
Calling Convention 실습을 하면서 스택 중간에 16바이트의 빈 공간이 있었다. 이유를 알아보자.

이유

- func 함수의 복귀 주소를 건드리지 못하게 하기 위해 스택공간을 확장한 것이다.

(이 부분도 추가 학습이 필요함)

메모리



← 이를 스택 완충 공간이라 한다.

1) switch.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define MIN    5
6 #define MAX    10
7
8 #define CAMERA    5
9 #define LIDAR     6
10 #define RADAR     7
11 #define STEERING  8
12 #define BLDC      9
13 #define ULTRASONIC 10
14
15 int main(void)
16 {
17     int rand_num, i;
18
19     srand(time(NULL));
20
21     for (i = 0; i < 10; i++)
22     {
23         rand_num = rand() % (MAX - MIN + 1) + MIN;
24
25         switch (rand_num)
26         {
27             case CAMERA:
28                 printf("Camera 처리 함수 동작!\n");
29                 break;
30
```

```
31             case LIDAR:
32                 printf("Lidar 처리 함수 동작!\n");
33                 break;
34
35             case RADAR:
36                 printf("Radar 처리 함수 동작!\n");
37                 break;
38
39             case STEERING:
40                 printf("Steering 처리 함수 동작!\n");
41                 break;
42
43             case BLDC:
44                 printf("BLDC 처리 함수 동작!\n");
45                 break;
46
47             case ULTRASONIC:
48                 printf("UltraSonic 처리 함수 동작!\n");
49                 break;
50
51             default:
52                 printf("그런거 없어요!\n");
53         }
54     }
55
56     return 0;
57 }
```

2) switch.c의 일부 코드 해석

```

15 int main(void)
16 {
17     int rand_num, i;
18
19     srand(time(NULL));
20
21     for (i = 0; i < 10; i++)
22     {
23         rand_num = rand() % (MAX - MIN + 1) + MIN;
24
25         switch (rand_num)           수식 해석
26         {
27             case CAMERA:
28                 printf("Camera 처리 함수 동작!\n");
29                 break;

```

(참고)

수학에서 수의 범위를 표현하는 방법

- 이상, 이하 - [,]
 - 시작과 끝의 숫자를 포함
- 초과, 미만 - (,)
 - 시작과 끝의 숫자를 제외

변수 rand_num에 대입하는 값이 무엇인지 알아보자

rand()

먼저 rand()는 [0, RAND_NUM] 중에서 숫자 하나를 반환하는 데, 이 실습 환경을 기준으로 RAND_NUM은 2147483647이다.

(MAX - MIN + 1)

% 연산자 오른쪽에 있는 (MAX - MIN + 1)은 [MIN, MAX]의 개수를 구하는 수식이다.

여기서는 MAX가 10, MIN이 5이므로 수식에 대입하면 6이 나온다.

rand() % (MAX - MIN + 1)

rand()가 반환한 값을 6으로 나머지 연산을 하면 결과로 [0, 5] 중 숫자 하나가 나온다.

+ MIN

위에서 나머지 연산한 결과에다가 MIN을 더하는 이유는 [MIN, MAX]의 범위로 변환하기 위해서이다.

∴ 변수 rand_num에 대입하기 위한 계산한 수식은 **[MIN, MAX] 중 랜덤숫자 하나를 구하기 위한 것이다.**

3) 실행 결과

```
try@try-desktop:~/Desktop/eddi/4_week$ ./switch
Camera 처리 함수 동작!
UltraSonic 처리 함수 동작!
Radar 처리 함수 동작!
BLDC 처리 함수 동작!
Lidar 처리 함수 동작!
Radar 처리 함수 동작!
Steering 처리 함수 동작!
Camera 처리 함수 동작!
UltraSonic 처리 함수 동작!
Radar 처리 함수 동작!
try@try-desktop:~/Desktop/eddi/4_week$
```

switch문

4) switch문의 단점: 프로토콜이 많아지면 가독성이 떨어진다

예를 들어 우리가 TV 리모컨을 개발한다고 하자

리모컨의 버튼이 눌렸을 때 그에 맞는 작동을 실행시키고자 switch문을 사용하여 오른쪽과 같이 코드를 작성했다고 한다면...(사실 앞서 보여준 switch.c 파일을 생각해도 된다)

명령어 개수만큼 switch문의 case 또한 늘어나 가독성 측면에 영향을 준다.

위의 예시를 통해 솔루션으로 “가독성 측면의 함수포인터 응용 기법”을 간단히 소개한다



프로토콜이 많아질수록 →
case가 더 추가됨

```
1 #include <stdio.h>
2
3 #define POWER 0
4 #define VOLUME_UP 1
5 #define VOLUME_DOWN 2
6 #define MUTE 3
7
8 int main(void)
9 {
10     int btn;
11
12     switch (btn)
13     {
14         case POWER:
15             break;
16
17         case VOLUME_UP:
18             break;
19
20         case VOLUME_DOWN:
21             break;
22
23         case MUTE:
24             break;
25
26         default:
27             printf("Error!\n");
28             break;
29     }
30
31     return 0;
32 }
```


가독성 측면의 함수포인터 응용 기법

4) 간단하게 살펴보기-1

임베디드 마스터 Lv2에서 배울 내용이기 때문에, 수업 시간에는 간단히 살펴보았다.



The screenshot shows a code editor window for a file named `protocol_call_table_map.h` in the `Smart-City / smart_city_socket_server / protocol` directory. The file contains C preprocessor directives and function pointer definitions. A red arrow points to line 8, which is a `#define` macro used for array initialization.

```
1 #ifndef SMART_CITY_SOCKET_SERVER_PROTOCOL_CALL_TABLE_MAP_H
2 #define SMART_CITY_SOCKET_SERVER_PROTOCOL_CALL_TABLE_MAP_H
3
4 #define __PROTOCOL_CALL_TABLE(nr, sym) [nr] = sym,
5
6 #include "protocol_handler.h"
7
8 __PROTOCOL_CALL_TABLE(0, protocol_dummy)
9 __PROTOCOL_CALL_TABLE(1, vehicle_handler)
10 __PROTOCOL_CALL_TABLE(2, electric_plant_handler)
11 __PROTOCOL_CALL_TABLE(3, shooting_range_handler)
12 __PROTOCOL_CALL_TABLE(4, gas_sensor_handler)
13 __PROTOCOL_CALL_TABLE(5, traffic_control_handler)
14 __PROTOCOL_CALL_TABLE(6, central_socket_server_handler)
15 __PROTOCOL_CALL_TABLE(7, crime_prevention_cctv_handler)
16 __PROTOCOL_CALL_TABLE(8, traffic_monitor_cctv_handler)
17 __PROTOCOL_CALL_TABLE(9, edge_device_handler)
18 __PROTOCOL_CALL_TABLE(10, central_web_server_handler)
19
20 #endif //SMART_CITY_SOCKET_SERVER_PROTOCOL_CALL_TABLE_MAP_H
```

← ① #define 매크로를 사용하여 배열에 함수포인터를 저장

가독성 측면의 함수포인터 응용 기법

4) 간단하게 살펴보기-2

이런 식으로 switch문 대신 함수포인터를 사용하면 한 줄로 가독성의 향상을 얻을 수 있다.

```
45 void processing_protocol (work_queue *prot_queue)
46 {
47     queue_node *node = prot_queue->head;
48     prot_analysis_metadata *data = node->data;
49
50     // TODO: 처리 방식 변경시 대응 방법에 대한 의존성 문제로 리팩토링 필요
51     // protocol_call_table[((protocol_pkt *)pkt)->target_command](pkt);
52     protocol_call_table[data->target](data); ← ② 거대한 switch문 대신 한 줄만 작성
53 }
```

if문의 약점

1) if_vs_switch.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define MIN 1
6 #define MAX 10
7
8 int main(void)
9 {
10     int rand_num, i;
11
12     srand(time(NULL));
13
14     for (i = 0; i < 10; i++)
15     {
16         rand_num = rand() % (MAX - MIN + 1) + MIN;
17
18         if (rand_num > 5)
19         {
20             printf("5보다 크다: %d\n", rand_num);
21         }
22         else if (rand_num > 9)
23         {
24             printf("9보다 크다: %d\n", rand_num);
25         }
26         else if (rand_num > 3)
27         {
28             printf("3보다 크다: %d\n", rand_num);
29         }
30     }
31
32     return 0;
33 }
```

if문의 약점

2) 실행 결과

```
try@try-desktop:~/Desktop/eddi/4_week$ ./if_vs_switch
5보다 크다: 10
3보다 크다: 4
5보다 크다: 9
5보다 크다: 8
3보다 크다: 5
5보다 크다: 8
try@try-desktop:~/Desktop/eddi/4_week$ ./if_vs_switch
5보다 크다: 6
5보다 크다: 8
5보다 크다: 7
3보다 크다: 5
3보다 크다: 4
5보다 크다: 9
3보다 크다: 5
5보다 크다: 9
try@try-desktop:~/Desktop/eddi/4_week$
```

(참고)

else문을 추가하지 않았기
때문에 무조건 10번 출력하지
않는다.



if문의 약점

3) 순서가 약점인 이유


rand_num에 저장된 값이 10일 경우에 9보다 크므로 “9보다 크다: 10”이라는 문자열을 출력하고자 한다.

앞서 본 코드를 기준으로 실행시킨다면, if문의 맨 앞의 조건문(rand_num > 5)을 만족하기 때문에 “5보다 크다: 10”이라는 문자열을 출력한다. 이를 의도한 대로 바꾸고자 한다면 **조건문의 순서를 고려한 오른쪽 그림과 같이 코드를 수정**해야 한다.

```
1 if (rand_num > 5)
2 {
3     printf("5보다 크다: %d\n", rand_num);
4 }
5 else if (rand_num > 9)
6 {
7     printf("9보다 크다: %d\n", rand_num);
8 }
9 else if (rand_num > 3)
10 {
11     printf("3보다 크다: %d\n", rand_num);
12 }
```



```
1 if (rand_num > 9)
2 {
3     printf("9보다 크다: %d\n", rand_num);
4 }
5 else if (rand_num > 5)
6 {
7     printf("5보다 크다: %d\n", rand_num);
8 }
9 else if (rand_num > 3)
10 {
11     printf("3보다 크다: %d\n", rand_num);
12 }
```



결론

if문의 약점은 순서이며, 만약 if문을 써야한다면 차라리 switch가 낫다.

중첩 반복문에서 효율적인 goto문

1) why_goto.c

중첩 반복문에서 빠져나와야 할 때, if문을 사용하면 다음과 같이 구성할 수 있다.

```
1 // 강사님 파일은 why_goto2.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5 #include <stdbool.h>
6
7 #define ERROR      7
8 #define LOOP_END   50
9
10 int main(void)
11 {
12     int i, j, k;
13     int data;
14     bool error_flag = false;
15
16     srand(time(NULL));
17
18     for (i = 0; i < LOOP_END; i++)
19     {
20         for (j = 0; j < LOOP_END; j++)
21         {
22             for (k = 0; k < LOOP_END; k++)
23             {
24                 data = rand() % (LOOP_END + 1); // 범위는 [0, LOOP_END]
```

```
25                 printf("data = %d\n", data);
26
27                 if (data == ERROR)
28                 {
29                     printf("Error 발생!\n");
30                     error_flag = true;
31                     break;
32                 }
33             }
34
35             if (error_flag)
36             {
37                 break;
38             }
39         }
40
41         if (error_flag)
42         {
43             break;
44         }
45     }
46
47     return 0;
48 }
```

중첩 반복문에서 효율적인 goto문

2) super_goto.c

만약 앞의 코드에서 if문 대신 goto문을 사용하면 다음과 같이 구성할 수 있다.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <stdbool.h>
5
6 #define ERROR      7
7 #define LOOP_END   50
8
9 #define DATA_RECEIVE_ERROR -3
10
11 int main(void)
12 {
13     int i, j, k;
14     int data;
15
16     srand(time(NULL));
17
18     for (i = 0; i < LOOP_END; i++)
19     {
20         for (j = 0; j < LOOP_END; j++)
21         {
```

```
22             for (k = 0; k < LOOP_END; k++)
23             {
24                 data = rand() % (LOOP_END + 1); // 범위는 [0, LOOP_END]
25                 printf("data = %d\n", data);
26
27                 if (data == ERROR)
28                 {
29                     printf("Error 발생!\n");
30                     goto error_handler;
31                 }
32             }
33         }
34     }
35
36     return 0;
37
38 error_handler:
39     printf("에러 핸들링 시작!\n");
40     return DATA_RECEIVE_ERROR;
41 }
```

중첩 반복문에서 효율적인 goto문

3) goto문을 사용해야 하는 이유-1

앞서 중첩 반복문에서 빠져나오는 방법으로 if문을 사용한 경우와 goto문을 사용한 경우를 비교해보자

```
27     if (data == ERROR)
28     {
29         printf("Error 발생!\n");
30         error_flag = true;
31         break;
32     }
33
34
35     if (error_flag)
36     {
37         break;
38     }
39
40
41     if (error_flag)
42     {
43         break;
44     }
45 }
46
47 return 0;
48 }
```

①

②

carbon
carbon.now.sh

why_goto.c의 일부

첫 번째로 if문을 사용한 경우를 살펴보자

왼쪽 그림의 ①

data가 에러인지 확인하는 조건문인데, 이는 에러가 발생했는지 무조건 확인해야 하는 **필수 조건문**이다.

추후에 if문을 사용하여 중첩 반복문을 나가고 있기 때문에 error_flag라는 플래그 변수에 true 값을 대입하고 있다.

break문을 만나 가까운 반복문 하나를 탈출한다.

②

나머지 반복문을 빠져나가기 위해 플래그 변수 error_flag가 true인지 확인하는 조건문이다.

여기서 이 ② 조건문의 단점은 **에러가 발생하지 않아도 수시로 if문으로 플래그 변수를 확인해야**

중첩 반복문에서 효율적인 goto문

3) goto문을 사용해야 하는 이유-2

```
27     if (data == ERROR)
28     {
29         printf("Error 발생!\n");
30         goto error_handler;
31     }
32 }
33 }
34 }
35
36 return 0;
37
38 error_handler:
39     printf("에러 핸들링 시작!\n");
40     return DATA_RECEIVE_ERROR;
41 }
```

← ①

carbon
carbon.now.sh

super_goto.c의 일부

이번에는 goto문을 사용한 경우를 보자.

왼쪽 그림의 ①

앞의 슬라이드와 마찬가지로 에러가 발생했는지 확인하는 필수 조건문이다.

if문과 다른 점은 goto문을 사용하여 중첩 반복문을 한 번에 빠져나가기 때문에 **나머지 반복문의 탈출 조건으로 플래그 변수를 선언해서 사용할 필요가 없어졌다.**

그리고 **나머지 반복문의 탈출 조건을 확인할 조건문 또한 필요 없어졌다.**

중첩 반복문에서 효율적인 goto문

4) 이유 정리

중첩 반복문에서 빠져나오는 방법으로 if문 대신 goto문을 사용할 경우

- 플래그 변수를 사용할 필요가 없다 → 불필요한 메모리 절약
- 나머지 반복문들의 탈출 조건(if문)을 확인 할 필요가 없다
 - if문을 사용하지 않아도 된다 → 가독성 향상
 - 에러가 아니더라도 수시로 if문을 확인할 필요가 없어졌다 → 성능 개선

(참고)

어셈 레벨에서 보면 goto문은 그냥 jmp 이지만, if문은 mov, cmp, jmp 이다.

또한, if문을 돌 때마다 파이프라인이 깨지게 되어있다.

중첩 반복문에서 효율적인 goto문

5) 리눅스 커널로 goto문 종결 & 결론

카페 링크: <https://cafe.naver.com/eddicorp/936>


오른쪽 그림은 세계 최고의 석학들이 개발하고 수정하여 배포 중인 리눅스 커널 코드의 일부인데, goto문이 떡칠 된 것을 볼 수 있다.

나중에는 goto의 확장형인 setjmp, longjmp는 리눅스에서 배울 예정인 듯하다.

참고: <https://blog.naver.com/eddi2021/222880815981>

결론

에러 컨트롤은 무조건 goto문이다.



```
/ kernel / fork.c

2209
2210
2211 /* Perform scheduler related setup. Assign this task to a CPU. */
2212 retval = sched_fork(clone_flags, p);
2213 if (retval)
2214     goto bad_fork_cleanup_policy;
2215
2216 retval = perf_event_init_task(p, clone_flags);
2217 if (retval)
2218     goto bad_fork_cleanup_policy;
2219 retval = audit_alloc(p);
2220 if (retval)
2221     goto bad_fork_cleanup_perf;
2222 /* copy all the process information */
2223 shm_init_task(p);
2224 retval = security_task_alloc(p, clone_flags);
2225 if (retval)
2226     goto bad_fork_cleanup_audit;
2227 retval = copy_semundo(clone_flags, p);
2228 if (retval)
2229     goto bad_fork_cleanup_security;
2230 retval = copy_files(clone_flags, p);
2231 if (retval)
2232     goto bad_fork_cleanup_semundo;
2233 retval = copy_fs(clone_flags, p);
2234 if (retval)
2235     goto bad_fork_cleanup_files;
2236 retval = copy_sighand(clone_flags, p);
2237 if (retval)
2238     goto bad_fork_cleanup_fs;
2239 retval = copy_signal(clone_flags, p);
2240 if (retval)
2241     goto bad_fork_cleanup_sighand;
2242 retval = copy_mm(clone_flags, p);
2243 if (retval)
2244     goto bad_fork_cleanup_signal;
2245 retval = copy_namespaces(clone_flags, p);
2246 if (retval)
2247     goto bad_fork_cleanup_mm;
2248 retval = copy_io(clone_flags, p);
2249 if (retval)
2250     goto bad_fork_cleanup_namespaces;
2251 retval = copy_thread(p, args);
2252 if (retval)
2253     goto bad_fork_cleanup_io;
2254
2255 stackleak_task_init(p);
```

‘AND NOT 기법’ 간단하게 살펴보기

1) and_not.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define MAGIC 16
6
7 int main(void)
8 {
9     int i, data;
10
11     srand(time(NULL));
12
13     for (i = 0; i < 10; i++)
14     {
15         data = rand() % 4096 + 4096; // 범위는 [4096, 8191]
16
17         printf("data = %d, data & ~(MAGIC - 1) = %d\n", data, data & ~(MAGIC - 1));
18     }
19
20     return 0;
21 }
```

‘AND NOT 기법’ 간단하게 살펴보기

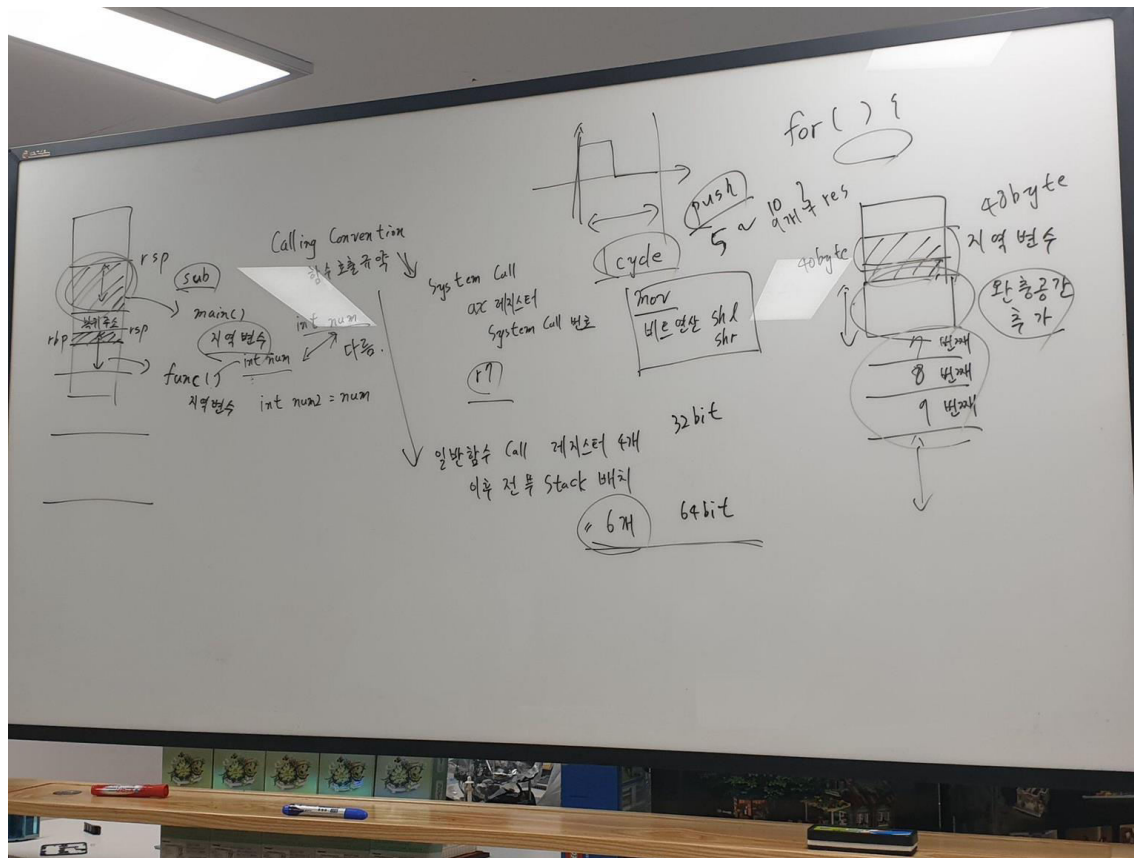
2) 실행 결과

```
try@try-desktop:~/Desktop/eddi/4_week$ ./and_not
data = 6141, data & ~(MAGIC - 1) = 6128
data = 5344, data & ~(MAGIC - 1) = 5344
data = 6653, data & ~(MAGIC - 1) = 6640
data = 7570, data & ~(MAGIC - 1) = 7568
data = 7328, data & ~(MAGIC - 1) = 7328
data = 5212, data & ~(MAGIC - 1) = 5200
data = 7818, data & ~(MAGIC - 1) = 7808
data = 6637, data & ~(MAGIC - 1) = 6624
data = 5593, data & ~(MAGIC - 1) = 5584
data = 5377, data & ~(MAGIC - 1) = 5376
try@try-desktop:~/Desktop/eddi/4_week$
```

간단 설명: 'data & ~(16 - 1)'의 결과는 data 이하의 수 중에서 가장 가까운 16배수의 값이다.

(참고) 이 기법은 물류센터 물류 알고리즘에 사용한다고 한다.

수업내용 사진



수업내용 사진

