시스템 프로그래밍 최적화 실습

2분반 32181827 박종기

1. 소개

재귀함수는 반복적인 처리를, 짧은 코드와 효과적인 코드로 편의를 제공한다. 하지만 함수를 계속해서 부르는 구조를 가진 재귀호출에서 caller는 callee함수를 위해 argument와 return address를, callee함수는 해당 stack frame의 기준 역할을 하는 ebp, 지역변수 및 함수 내부 처리를 위한 요소들을 위해 스택을 구성한다. 이런 구조를 가진 재귀호출 함수는 함수 특성상 호출 횟수가 많아지게 되면 stack frame이 많아질 것이고 이때문에 프로그램 실행시간도 만드는 stack frame이 많아질수록 길어지게 될 것이다. 즉 함수를 위한 비용이 많이 들게 된다. 재귀함수의 대표적인 예로 팩토리얼 함수를 들 수 있지만 계산 결과가 기하급수적으로 커지기 때문에 1부터 n까지의 정수의 합을 구하는 구조를 예로 들어 실습을 진행하였다.

재귀의 사용이 비효율적이라는 이유때문에 사용을 하지 않을 수는 없다. 물론 재귀함수의 경우 반복문으로 어느정도 대체를 할 수도 있겠지만 퀵 정렬 알고리즘을 사용하는 경우처럼 비효율성이 문제가 되지 않는 경우도 있고 프로그램의 유지보수 및 가독성이 매우 뛰어나기 때문에 사용을 기피하는 방법보다는 좀 더 실용적으로 만들어보는 방안을 찾고자 하였다.

실습에서는 1부터 n까지의 합을 구하는 3개의 함수인 sum1, sum2, sum3 을 사용하여 실습을 진행한다. sum1은 일반적인 재귀호출함수의 형태를, sum2는 재귀호출을 사용하지 않고 일반 함수로 합을 구하는 과정을 진행했을 때, sum3는 재귀호출의 과정에서 성능을 향상시킬 수 있는 방법의 후보로 생각한 함수를 구현하였다. 각각의 함수를 실행하면서 n을 늘려나갈 때 각각의 수행시간들을 직접 측정해보고 어셈블리 코드를 분석하며 성능향상의 원인을 찾아보고 더 성능을 향상시킬 수 있는 요소들을 찾아보고자 한다.

2. 본문

1) 프로그램 설계 개요

1부터 n까지의 합을 구하는 함수를 작성하는 방법은 매우 다양하여 셀 수 없이 많다. 그 중 1부터 n까지의 합을 구하는 방법으로 재귀 함수를 사용하고 사용된 재귀 함수의 수행시간을 단축시킬 방법을 찾아보고자 하였다. 실습과정은 다음과 같다. 먼저 일반적인 재귀함수 sum1을 작성한다. 재귀함수를 사용하지 않았을 때와 비교하기 위해 재귀함수를 사용하지 않고 합을 구하는 프로그램인 sum2의 수행시간과 비교하고 재귀함수의 수행시간을 단축시킬 수 있는 방법으로 sum3를 제시한다. 이 3개의 프로그램들을 n을 늘려가며 수행해보고 gettimeofday() 함수를 이용하여 수행시간들을 측정하는 순서로 실습과정을 진행하였다.

2) 프로그램 설명

먼저 일반적인 형태의 재귀함수인 sum1 이다. 재귀는 각 호출을 위한 인자와 변수를 스택에 쌓아두어 관리한다. sum1함수는 재귀호출이 끝나는 조건으로 인자 x의 값이 1보다 작아지도록 하였고 이 조건을 충족하기 전까지는 sum1함수를 계속 호출하게 하며 이전 호출의리턴 값으로 현재 호출의 계산이 이루어지므로 함수는 처음 호출된 함수는 마지막 조건을 만나기 전까지는 끝날 수 없다. 마지막 조건을 만나고나서야 비로소 실행되고 있는 함수들이 리턴값을 반환하며 나중에 수행된 함수부터 하나씩 종료되는 구조를 갖는다.

```
int sum1(int x){
        if (x<1) return 0;
        else
                return (x+sum1(x-1));
}
```

이렇게 프로그램을 작성했을 때 n이 매우 커져서 많은 함수가 재귀적으로 호출되면 각 함수에서 사용할 스택을 할당받아야 하고 분기와 관련된 레지스터 값의 로딩 및 저장을 포함한 context switch, 분기에 의한 파이프라인을 효과적으로 사용하지 못하는 등 재귀함수때문에 발생하는 손실 비용이 크다.

sum1함수의 어셈블리어를 살펴보자.

```
sys32181827@embedded:~/optimization$ objdump -d ./branchtest
```

어셈블리어는 objdump -d 를 이용하여 branchtest.c의 object file인 branchtest 를 disassemble 한 코드를 가져왔다.

```
080483c2 <sum1>:
 80483c2:       55                      push   %ebp
 80483c3:       89 e5                   mov    %esp,%ebp
 80483c5:       83 ec 08                sub    $0x8,%esp
 80483c8:       83 7d 08 00             cmpl   $0x0,0x8(%ebp)
 80483cc:       7f 09                   jg     80483d7 <sum1+0x15>
 80483ce:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
 80483d5:       eb 12                   jmp    80483e9 <sum1+0x27>
 80483d7:       8b 45 08                mov    0x8(%ebp),%eax
 80483da:       48                      dec    %eax
 80483db:       89 04 24                mov    %eax,(%esp)
 80483de:       e8 df ff ff ff          call   80483c2 <sum1>
 80483e3:       03 45 08                add    0x8(%ebp),%eax
 80483e6:       89 45 fc                mov    %eax,-0x4(%ebp)
 80483e9:       8b 45 fc                mov    -0x4(%ebp),%eax
 80483ec:       c9                      leave
 80483ed:       c3                      ret
```

어셈블리어를 보면 1~3번째 줄의 push, mov, sub명령어를 통해 stack frame을 구성하였다. 처음에 인자 cmpl으로 메모리에서 읽어온 함수의 인자와 0을 비교하여 인자로 들어온 값이 0보다 크면 새로운 함수 호출을 위한 argument, return address 를 준비하는 과정에 해당하는 주소 0x80483d7 부터의 mov, dec, mov, call 에 해당하는 return (x+sum1(x-1)) 를 수행한다. 인자가 0이 되는 경우에는 0x80483ce 의 movl, jmp 명령어를 수행하고 stack frame을 정리한다. 나중에 호출된 함수가 리턴되면 리턴값이 저장된 eax레지스터와 argument에 해당하는 값의 합을 구해 다시 eax를 통해 이전에 호출된 함수로 전달된다.

sum1(2)를 수행하는 과정을 보자.

	stack frame for main
	2 argument
	return address
sum1(2)	ebp
	1 %eax
	return address
sum1(1)	ebp
	0 %eax
	return address
sum1(0)	ebp
	0 %eax

sum1함수가 재귀호출 되는 동안의 stack frame 을 위의 그림과 같이 그릴 수 있다. 노란색 공간은 아래의 재귀호출된 함수들이 return 한 이후 eax레지스터를 통해 정리되는 공간이다. sum1 함수의 어셈블리어에서 아래에서부터 5번째 명령어인 add를 보면 sum(0) 함수에서 eax레지스터를 통해 return 된 값인 0과 sum1(1)함수의 argument 인 1과의 덧셈연산을 하여 eax레지스터에 저장한 이후 sum1(2)함수로 전달하는 과정을 확인할 수 있다. 즉 eax레지스터에 있는 값의 이동을 통해 함수는 순차적으로 종료되어야만 한다.

sum2 함수를 sum1 과 후에 나올 sum3함수와의 수행시간 비교를 위해 sum1에서의 재귀함수를, 반복문을 사용하여 특별한 최적화 기법들을 사용하지 않고 인자의 값에 상관없이 함수를 한번만 호출하도록 다음과같이 작성하였다.

```
int sum2(int x){
        int i=1,result=0;
        if (x==0) return 0;
        for (i=1;i<=x;i++)
                result+=i;
        return result;
}
```

다음은 sum3 함수이다. 재귀호출의 종료조건으로 x가 0이 될때 y를 리턴하도록 하였고 조건을 충족하기 전까지는 sum3함수를 계속 재귀호출하도록 하였다.

```
int sum3(int x, int y){
        if(x==0)
                return y;
        else return sum3(x-1,x+y);
}
```

sum3 함수에 대해서도 object file 의 disassemble 을 통해 어셈블리어를 살펴보면 다음과 같다.

```
0804843a <sum3>:
 804843a:       55                      push   %ebp
 804843b:       89 e5                   mov    %esp,%ebp
 804843d:       83 ec 0c                sub    $0xc,%esp
 8048440:       83 7d 08 00             cmpl   $0x0,0x8(%ebp)
 8048444:       75 08                   jne    804844e <sum3+0x14>
 8048446:       8b 45 0c                mov    0xc(%ebp),%eax
 8048449:       89 45 fc                mov    %eax,-0x4(%ebp)
 804844c:       eb 19                   jmp    8048467 <sum3+0x2d>
 804844e:       8b 45 0c                mov    0xc(%ebp),%eax
 8048451:       03 45 08                add    0x8(%ebp),%eax
 8048454:       89 44 24 04             mov    %eax,0x4(%esp)
 8048458:       8b 45 08                mov    0x8(%ebp),%eax
 804845b:       48                      dec    %eax
 804845c:       89 04 24                mov    %eax,(%esp)
 804845f:       e8 d6 ff ff ff          call   804843a <sum3>
 8048464:       89 45 fc                mov    %eax,-0x4(%ebp)
 8048467:       8b 45 fc                mov    -0x4(%ebp),%eax
 804846a:       c9                      leave
 804846b:       c3                      ret
```

어셈블리어를 보면 1~3번쨰 줄의 push, mov, sub명령어를 통해 stack frame을 구성하였다. cmpl 를 통해 첫번째 인자와 비교를 통해 함수를 호출하는 804844e주소로 점프해 함수의 내용을 수행한 이후 다음 함수의 호출을 위한 stack frame을 구성한다. 인자 x의 값이 0이 되어 더이상 함수의 재귀호출이 없을 때에는 순차적으로 stack frame들을 정리하는 과정을 거치게 된다. sum3(2,0)을 예로 들어 stack frame이 생성되는 과정을 그려보면 다음과 같다.

	stack frame for main
	0 argument y
	2 argument x
	return address
sum3(2,0)	ebp
	2 %eax (x+y)
	1 %eax (x-1)
	return address
sum3(1,2)	ebp
	3 %eax
	0 %eax
	return address
sum3(0,3)	ebp
	3 %eax

노란색 공간은 아래의 재귀호출된 함수들이 return 한 이후 eax레지스터를 통해 정리되는 공간이다. sum3(2,0) , sum2(1,2), sum3(0,3)이 차례로 호출되며 각 함수가 호출될때마다 원하는 결과가 argument y 에 저장된다. 따라서 sum3(0,3) 이 호출될 시에는 결과가 y에 이미 저장되어 있는 구조가 된다.

sum1함수와 비교했을 때 함수가 리턴 된 후 sum3와는 다르게 sum1함수는 add 를 추가적으로 수행하는 모습을 확인할 수 있다. sum3의 경우 함수가 하나 실행될 때마다 함수의 결과를 도출해내기 때문에 이후 호출된 함수가 마치는 것과 상관없이 결과를 도출할 수 있다.따라서 재귀호출을 통해 불러졌던 이전의 함수들의 stack frame을 eax레지스터를 통해 정리하는 과정에 있어서 eax레지스터에는 어떠한 값이 있어도 상관이 없게된다.

다시말해 이전 함수에서 sum1에서처럼 add명령어와 같은 연산을 수행하지 않기 때문에 함수가 리턴 된 이후 eax레지스터의 내용을 mov해주는 명령어를 수행하는데 eax레지스터에 무슨 내용이 들어가있던 전혀 상관이 없게 되는 것이다. 따라서 pipeline을 효과적으로 활용할 수 있게 된다.

3) 프로그램 실행

함수의 수행시간 측정은 gettimeofday() 함수를 이용하여 실시하였다. 또한 프로그램의 가독성 및 수정을 유연하게 하기 위해 1부터 몇까지 더하는 시간을 측정할 것인지에 대한 SIZE 상수를 정의하였다.

```
gettimeofday(&stime,NULL);
int result1 = sum1(SIZE);
gettimeofday(&etime1,NULL);

int result2 = sum2(SIZE);
gettimeofday(&etime2,NULL);

int result3 = sum3(SIZE,0);
gettimeofday(&etime3,NULL);
```

SIZE를 10에서부터 시작하여 10배씩 늘려가며 시간측정을 하였고 시간 측정은 프로그램을 두번씩 수행해가며 측정하였다.

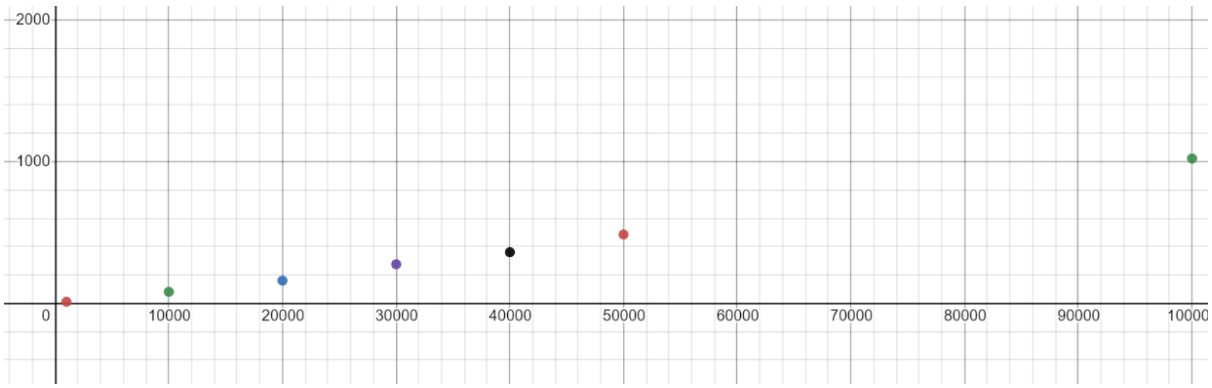
SIZE 10의 경우와 SIZE100의 경우는 다음과 같다. 두가지 경우 모두 3가지 함수의 수행시간 차이가 큰 차이가 없는 것을 확인할 수 있다.

```
sys32181827@embedded:~/optimization$ vi branchtest.c
sys32181827@embedded:~/optimization$ gcc -o branchtest branchtest.c
sys32181827@embedded:~/optimization$ ./branchtest
case sum1 : 55
elapsed time 0sec / 1usec
case sum2 : 55
elapsed time 0sec / 0usec
case sum3 : 55
elapsed time 0sec / 0usec
sys32181827@embedded:~/optimization$ ./branchtest
case sum1 : 55
elapsed time 0sec / 0usec
case sum2 : 55
elapsed time 0sec / 1usec
case sum3 : 55
elapsed time 0sec / 0usec
sys32181827@embedded:~/optimization$ vi branchtest.c
sys32181827@embedded:~/optimization$ gcc -o branchtest branchtest.c
sys32181827@embedded:~/optimization$ ./branchtest
case sum1 : 5050
elapsed time 0sec / 1usec
case sum2 : 5050
elapsed time 0sec / 1usec
case sum3 : 5050
elapsed time 0sec / 1usec
sys32181827@embedded:~/optimization$ ./branchtest
case sum1 : 5050
elapsed time 0sec / 1usec
case sum2 : 5050
elapsed time 0sec / 1usec
case sum3 : 5050
elapsed time 0sec / 1usec
```

SIZE 1000과 10000의 경우이다. 이때부터 눈에 띄는 실행시간 차이가 발생하기 시작한다.

```
sys32181827@embedded:~/optimization$ vi branchtest.c
sys32181827@embedded:~/optimization$ gcc -o branchtest branchtest.c
sys32181827@embedded:~/optimization$ ./branchtest
case sum1 : 500500
elapsed time 0sec / 18usec
case sum2 : 500500
elapsed time 0sec / 5usec
case sum3 : 500500
elapsed time 0sec / 9usec
sys32181827@embedded:~/optimization$ ./branchtest
case sum1 : 500500
elapsed time 0sec / 19usec
case sum2 : 500500
elapsed time 0sec / 6usec
case sum3 : 500500
elapsed time 0sec / 6usec
sys32181827@embedded:~/optimization$ vi branchtest.c
sys32181827@embedded:~/optimization$ gcc -o branchtest branchtest.c
sys32181827@embedded:~/optimization$ ./branchtest
case sum1 : 50005000
elapsed time 0sec / 191usec
case sum2 : 50005000
elapsed time 0sec / 53usec
case sum3 : 50005000
elapsed time 0sec / 111usec
sys32181827@embedded:~/optimization$ ./branchtest
case sum1 : 50005000
elapsed time 0sec / 194usec
case sum2 : 50005000
elapsed time 0sec / 53usec
case sum3 : 50005000
elapsed time 0sec / 112usec
```

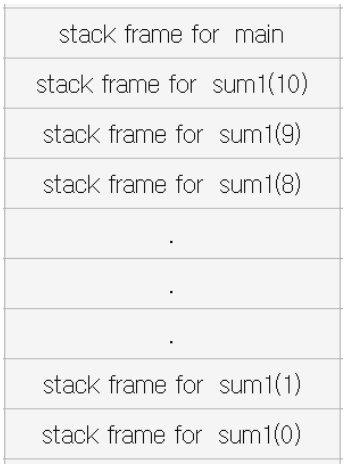
재귀호출을 사용하지 않는 sum2함수의 경우가 SIZE가 1000, 10000인 경우에서 약 5usec , 53usec 의 시간으로 가장 짧았다. sum1과 sum3의 재귀호출 함수를 비교했을 때 SIZE 1000의 경우는 10 usec정도의 시간이 단축되었고 SIZE 10000 의 경우 80 usec 정도 단축되었다. 아래의 그래프를 보면 SIZE를 증가시켜가면서 수행시간을 측정하였을 때 선형적으로 증가하는 것을 확인할 수 있다.



이 3개의 함수들은 시스템이 제공하는 int형 정밀도의 한계를 초과하지 않는 한 올바르게 동작할 것이다. 만약 int 형 범위를 초과하는 경우는 어떻게 되는지 알아보기 위해 100000 까지의 합을 구하게 하면 segmentation fault가 발생하게 된다.

3. 결론

1부터 n까지의 합을 구하는 재귀함수 sum1 을 최적화 하기위한 방안으로 sum3을 제시하였다. n을 늘려가면서 성능측정을 한 결과 실제 성능이 향상된 것을 확인할 수 있었다. sum3이 sum1에 비해 매 재귀호출마다 stack frame의 크기가 하나 더 크지만 n을 10000개씩 늘려가며 성능을 측정한 결과 stack frame 1개당 약 0.009 usec 만큼씩의 성능향상 효과가 있었다.



성능향상의 원인으로 재귀호출 함수의 stack frame을 하나씩 없애갈 때마다 add연산을 하는 것을 원인으로 꼽을 수 있다. add연산이 존재하기 때문에 각각의 stack은 다음 쌓이는 stack과의 data dependency 를 갖게된다. 위의 그림은 sum1(10)을 실행한 경우이다. 이 add연산을 함으로써 호출된 함수들을 역순으로 정리해나가는 과정에 있어 sum1(10)의 stack frame을 정리하기 위해서 sum1(9)에서 eax레지스터를 통해 전달되는 return 값을 기다려야만 한다. 이때문에 stack frame을 정리하는 과정들이 pipeline을 통해 병렬적으로 수행될 수 없게 된다.

반면 sum3은 sum3(10,0) 의 stack frame 을 정리하기 위해서 sum(10,0)의 다음에 호출되는 sum3(9,19) 의 return 값을 기다릴 필요가 없다. 어셈블리어에서도 볼 수 있듯이 재귀함수의 특성상 나중에 호출된 함수부터 stack frame을 정리해나간다. 그러나 stack frame을 정리하는 과정들에 있어서 sum3은 sum1과 다르게 병렬적으로 수행될 수 있다.

이는 stack frame을 정리하는 과정에 있어서 eax레지스터의 내용이 변하는지를 봄으로써 간접적으로 각 stack frame들이 data dependency 가 있는지를 확인할 수 있다. sum1과 sum3의 경우 모두 어셈블리어에서 eax레지스터를 통해 stack 을 정리하기 때문이다. 만약 eax의 값이 변하지 않아 data dependency 가 없다면 stack frame을 정리하는 과정들이 pipeline을 이용해 병렬적으로 수행될 수 있다.

다음은 디버거를 이용해 sum1(4)의 경우를 디버거로 분석한 캡쳐화면들이다.

```
Breakpoint 1, sum1 (x=0) at branchtest.c:11
11              if (x<1) return 0;
(gdb) info reg
eax            0x0      0
```

```
(gdb) n
14      }
(gdb) info reg
eax            0x1      1
```

```
(gdb) n
14      }
(gdb) info reg
eax            0x3      3
```

```
(gdb) n
14      }
(gdb) info reg
eax            0x6      6
```

```
(gdb) n
14      }
(gdb) info reg
eax            0xa      10
```

첫번째 break point은 sum1함수의 시작부분이다. 함수를 수행하면서 gdb 명령어 중 info reg 를 통해서 eax레지스터의 값을 계속 추적하였다. sum1(4)을 호출할 경우 sum1(0)까지 호출되고 eax레지스터는 add연산을 통해 값이 계속 갱신된다. 위의 사진에서 "}" 은 stack frame을 정리해나가는 모습이고 이때마다 eax레지스터의 값이 갱신되는 것을 확인할 수 있다. 이를 통해 각 stack frame을 정리해나가는 데 있어서 data dependency 가 존재하는 것을 확인하였다.

반면 sum3의 경우 sum3(4,0)을 호출하였을 경우 sum3(0,10)까지 호출되고 stack frame을 정리하게된다. 다음은 디버거를 이용해 sum1(4,0)의 경우를 디버거로 분석한 캡쳐화면들이다.

```
(gdb) break 22
Breakpoint 2 at 0x8048440: file branchtest.c, line 22.
```

```
(gdb) n
26      }
(gdb) info reg
eax            0xa      10
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xffffd3bc       0xffffd3bc
ebp            0xffffd3c8       0xffffd3c8
esi            0xf7fb8000       -134512640
edi            0xf7fb8000       -134512640
eip            0x8048467        0x8048467 <sum3+45>
eflags         0x246    [ PF ZF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
(gdb) n
26      }
(gdb) info reg
eax            0xa      10
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xffffd3d0       0xffffd3d0
ebp            0xffffd3dc       0xffffd3dc
esi            0xf7fb8000       -134512640
edi            0xf7fb8000       -134512640
eip            0x8048467        0x8048467 <sum3+45>
eflags         0x246    [ PF ZF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
```

두번째 break point는 sum3 함수의 시작부분이다. sum3의 경우도 sum1 과 마찬가지로 info reg명령어를 통해 eax레지스터의 값을 계속 추적하였다. sum3(4,0)을 호출할 경우 sum(0,10) 까지 호출되고 "}" 로 stack frame들을 정리해나가는 과정에서 stack pointer인 esp 의 값은 증가하였지만 eax레지스터의 값은 갱신되지 않는 모습을 확인할 수 있다. 뒷부분에서도 마찬가지로 stack pointer가 증가(stack frame을 제거)하고 있음에도 eax레지스터의 값은 변함이 없다. 이를 통해 각 stack frame을 정리해나가는데 있어 data dependency 가 존재하지 않는 것을 확인하였다.

sum1의 경우와 다르게 sum3의 경우는 stack frame 간의 data dependency 가 존재하지 않기 때문에 stack frame을 정리하는 과정을 sum3은 pipeline을 이용하여 병렬적으로 처리할 수 있게된다. sum3에서 stack frame을 구성할 때 1개씩의 공간을 늘리는 대신 pipeline을 효과적으로 이용할 수 있도록 하여 최적화에 기여한 것이다. 메모리를 조금 더 사용하는 대신 수행시간을 단축시킨 것이다.