Chapter 13. 코딩 최적화

**캐싱**

- 데이터 캐싱과 명령 캐싱

- 캐싱은 빈번하고 부하가 많이 드는 연산의 결과를 기억하여 이러한 연산을 반복하여 수행하지 않도록 만드는 목적을 가짐

- 캐싱 기회는 코딩 분야에서 검출하고 구현하기에 쉽다. 반복문 내에서 상수 표현을 연산하는 것은 잘 알려진 성능 비 효율성 이다.

|  |
| --- |
| for ( … ; != done ; … )  done = patternMatch(pat1, pat2, isCaseSensitive()); |

- patternMatch() 함수는 일치 시킬 두 개의 문자열 패턴을 받아들이고, 대 소 문자 구별 여부에 관한 세번째 인자를 받음

- isCaseSensitive() 함수를 호출해 대, 소 문자 여부를 결정

- 이 결정은 고정되어 있어 반복문과 무관하며, 하나의 반복문에서 다음으로 넘어갈 때 바뀌지 않으므로 반복문 밖에서 호출되어야 한다.

|  |
| --- |
| int isSensitive = isCaseSensitive();  for ( … ; != done ; … )  done = patternMatch(pat1, pat2, isSensitive); |

- 이제, 대 소문자 구별은 한번 결정하고 이 결과를 지역 변수에 캐시하고 재사용한다.

**미리 계산**

- 결과를 캐시하면 성능이 중요한 경로에서 한 번만 연산하는 비용을 부담한다.

- 미리 연산하면 처음의 부담도 없어진다. 성능이 중요한 경우의 외부(ex 초기화) 에서 미리 연산을 수행하고, 성능이 중요한 경로 내부에서 수행하는 연산의 비용을 절대 부담할 필요가 없다.

ex) 문자열 작업을 처리하기 이전에 모두 대문자로 치환하는 이슈가 생길 때, 매 작업마다 toupper 같안 함수를 호출하는 것이 아닌 미리 치환해 놓은 uppertable[] 자료 구조등을 만들어 놓고 원하는 문자의 대문자를 치환하는 식으로 동작

**유연성 감소**

- 성능을 증가시키기 위해 상황을 단순화 시키는 가정을 만들 수 있다.

ex) 초기에 IP 주소를 위해 IP 문자열의 size 만큼 힙 영역에 new(), delete()를 수행하여 보관하였다. 하지만 매번 new(), delete()를 수행하는 것은 성능에 좋은 영향을 끼치지 않으며, 메모리 풀을 사용한다고 해도, 전혀 사용하지 않는 것에 비해 성능이 떨어질 수 밖에 없다.

이럴 때는 IP 문자열의 길이의 최대인 16바이트만큼의 char 배열을 스택에 선언하는 방법이 성능에 오히려 좋은 영향을 줄 수 있다.

**80-20 법칙 : 공통 경로의 속도를 높여라**

- 80-20의 법칙은 많은 응용을 가지고 있다. 실행 시나리오의 80%는 소스 코드의 오직 20%만을 통과하고, 수행 시간의 80%는 실행 경로에서 실행되는 20%의 함수에서 소요된다.

**지연 연산**

- 필요하지 않은 연산 때문에 인과 성능 저하가 발생하는 것은 나쁜 생각.

- 이러한 상황은 복잡한 코드에서 매우 빈번하다. **부하를 주는 연산은 오직 필요한 경우에만 수행해야 한다.**

|  |
| --- |
| int route(Message\* msg)  {  ExpensiveClass upstream(msg); // 부하가 많이 드는 객체  if (/\*...\*/1)  {  // 부하를 많이 주는 객체에 대한 처리  }  // upstream 객체는 여기서 사용되지 않음  return true;  } |

- upstream 객체는 매우 부하를 많이 주고 있으며, route 함수 최상단에 위치 하고 있다.

- 하지만 upstream 객체는 if 문 안에서만 사용되는 객체이다.

- 그러므로, if문 안에서 upstream을 지연 연산하도록 변경해야 한다.

|  |
| --- |
| int route(Message\* msg)  {  if (/\*...\*/1)  {  ExpensiveClass upstream(msg); // 부하가 많이 드는 객체  // 부하를 많이 주는 객체에 대한 처리  }  // upstream 객체는 여기서 사용되지 않음  return true;  } |

**불필요한 계산**

- 지연 연산은 실행 흐름에 따라 지금 당장은 필요하지 않은 계산에 관한 것

- 불필요한 계산은 절대 필요하지 않은 계산에 관한 것이다. 실행 흐름과 상관 없이 결과가 절대 사용되지 않는 완전히 불필요한 계산이 있다.

|  |
| --- |
| class Student  {  public:  Student(char\* nm)  {  nm\_ = nm;  }  private:  std::string nm\_;  }; |

- 이때 컴파일러는 string 기본 생성자에 대한 호출을 삽입한다.

- 우리는 컴파일러가 생성한 string 기본 생성자 호출이 만든 결과를 절대 사용할지 않을 것이다.(임시)

- Student 생성자 초기화 리스트에 string 생성자를 명시함으로 써 이 불필요한 연산을 없앨 수 있다.

|  |
| --- |
| class Student  {  public:  Student(char\* nm) : nm\_(nm)  {  }  private:  std::string nm\_;  }; |

**메모리 관리**

- 힙 메모리를 할당하고 해지 하는 것은 부하를 많이 준다.

- 성능 관점에서 보았을 때, 명시적 관리가 필요 없는 메모리를 사용하여 작업을 하는 것이 훨씬 가볍다.

- 지역 변수로 정의된 객체는 스택에 저장된다. 이 객체가 차지하는 스택 메모리는 객체가 정의된 범위 내부인 함수를 위해 설정된 스택 메모리의 일부분이다. 지역 객체 대신 new()와 delete()를 사용하여 힙 메모리를 할당하고 해지 할 수 있다.

**키 포인트**

- 코딩 최적화는 국지적 범위를 가지고 있으며, 전체 적인 프로그램 디자인의 이해를 필요로 하지 않는다. 여러분이 아직 디자인을 이해하지 못한 진행 중인 프로젝트에 합류한 경우, 코딩 최적화부터 시작 하면 좋다.

가장 빠른 코드는 절대 실행하지 않는 코드이다. 부하를 많이 주는 연산을 밖으로 빼내기 위해 다음 사항을 점검해 보자.

**1. 결과를 사용할 것인가?** 당연하다는 듯이 들리지만 실제로 일어난다. 종종 연산을 수행해 놓고 결과를 절대 사용하지 않는다.

**2. 결과가 지금 필요한가?** 결과가 실제로 필요한 지점 까지 연산을 연기한다. 미숙한 연산이 어떤 실행 경로에서 절대 사용되지 않을 수 있다.

**3. 결과를 이미 알고 있는가?** 부하를 많이 주는 연산의 결과가 두 줄 위에 나와 있음에도 불구하고 이 연산을 다시 수행하는 것을 보았다. 만약 실행 흐름의 앞부분에서 미리 계산하였다면, 결과를 재사용할 수 있도록 만든다.

가끔 연산을 밖으로 빼낼 수 없으며 수행해야만 하는 경우가 있다. 이제 속도를 증가 시키기 위해 도전

**1. 연산이 전체적으로 일반적인가?** 문에 영역이 필요로 하는 만큼만 유연하면 되고 더 이상은 필요하지 않다. 단순한 가정을 이용하자. 유연함을 감소시키면 속도는 증가한다.

**2. 어떤 유연성은 라이브러리 호출에 숨겨져 있다.** 자주 호출되는 특정 라이브러리 호출을 자체 제작하여 속도를 증가시킬 수 있다. 이 라이브러리들은 여러분의 노력을 정당화 시켜 주기에 충분히 자주 호출되어야 한다. 여러분이 사용하는 이 라이브러리와 시스템 호출의 숨겨진 부하를 대해서 잘 알아야 한다.

**3. 메모리 관리를 최소화 한다.** 대부분의 컴파일러에서 메모리 관리는 상대적으로 부하를 많이 준다.

4. 모든 입력 데이터의 집합을 고려한다면 이것의 20%가 시간의 80%를 차지한다. 다른 시나리오를 희생하고 전형적인 입력을 처리하는데 드는 속도를 개선 시킨다.

5. 캐시, RAM, 디스크 액세스 간의 속도 차이는 상당히 크다. 캐시 적중이 많이 일어날 만한 코드를 작성한다.