Gamasutra.com

병행성 디버깅

Debugging Concurrency

Phillippe Paquet

2005년 6월 6일

http://www.gamasutra.com/features/20050606/paquet_01.shtml

1. 머리말

2000 년 7 월에 애플(Apple)사는 듀얼 프로세싱을 자사의 파워맥(PowerMac) 제품 라인에 도입하였으며, 대칭형 멀티 프로세싱(SMP)이 공식적으로 주류가 되었습니다. 이로부터 4 년도 안 되는 시기에, 인텔(Intel)은 "하이퍼스레딩(Hyperthreading)"이라고 하는 기능을 자사의 IA-32 프로세서 라인에 도입함으로써, 동시성 멀티스레딩(SMT) 역시 주류가 되었습니다. 올 해 말이 지나기 전에, 우리들은 인텔(Intel), AMD 및 마이크로소프트(Microsoft)가 멀티코어(multi-core) 프로세서를 일반인들에게 소개하는 것을 볼 수 있을 것입니다. 멀티스레드 방식의 애플리케이션들은 이제 일상적인 것이 되었으며, 아마 성능의 극대화를 위한 유일한 방안일 것입니다. 불행하게도, 멀티스레드 방식의 애플리케이션에서도 새로운 등급과 유형의 버그(bug)들이 발견되었으며, 이는 심지어 경험이 풍부한 프로그래머들에게도 쉽지 않은 골치거리이기도 합니다.

단순화를 위해, 저는 단일 시스템 내의 스레드(thread)에 국한하여 이야기를 할 것이며, 그 중심 주제 역시 메모리에 대한 자원으로 제한하고자 합니다. 아래에 설명할 조건들은 모두 분산 시스템(distributed system)에서 일어날 수 있는 것으로, 반드시 이에 국한되는 것은

아니나 GPU 및 네트워크 연결과 같은 하드웨어의 모든 부분을 포함할 수도 있다는 사실을 염두에 두시길 바랍니다.

2. 레이스 조건 및 레이스 해저드

경우에 따라, 레이스 해저드(race hazard)라고도 불리는 레이스 조건은 서로 다른 두 개 이상의 스레드(thread)가 동시에 데이터에 접속할 때 발생하는 것으로서, 접속에 대한 시간적 순서화(temporal ordering)를 통제할 수 있는 메커니즘은 없습니다. 어떠한 스레드(thread)가 공유 파일에 먼저 접속을 하느냐에 따라, 프로그램의 행위(behavior)도 변합니다.

아래 예로 든 프로그램의 경우, g_iResult 의 인쇄 값은 어떤 스레드가 자신의 지정 사항(assignment)을 먼저 실행한 것인가에 달린 것입니다:

- 스레드 2(ThreadTwo)가 g_iResult 에 150 을 지정(할당)할 수 있게 되기 전에, 스레드 1(ThreadOne)이 g_iResult 에 50 을 할당(지정)한 경우, g_iResult 는 인쇄를 할 때 150 을 포함하게 될 것입니다.

- 스레드 1(ThreadOne)이 g_iResult 에 50 을 지정(할당)할 수 있게 되기 전에, 스레드 2(ThreadTwo)가 g_iResult 에 150 을 할당(지정)한 경우, g_iResult 는 인쇄를 할 때 50 을 포함하게 될 것입니다.

스레드 1(ThreadOne)과 스레드 2(ThreadTwo)가 개별적으로 실행이 될 경우에는, 항상 같은 결과가 나올 것입니다. 예측 불능(unpredictability)은 두 개의 스레드 사이에 있는 레이스(race)에 의해 일어 납니다.

```
//
// Race.cpp
//
// Example of a race condition
//
//

#include <windows.h>
#include <stdio.h>
#include <process.h>
```

```
int g_iResult = 100;
bool g_bThreadOneFinished = false;
bool g_bThreadTwoFinished = false;

void ThreadOne(void*)
{
  // Wait some random amount of time
  Sleep(rand());

  // Set the result
  g_iResult = 50;

  // Finished
  g_bThread One Finished = true ;
   _end thread();
}

void Thread Two(void*)
{
   // Wait some random amount of time
  Sleep(rand());

  // Set the result
  g_iResult = 150;

  // Finished
  g_bThreadTwoFinished = true ;
  _end thread();
}

int main()
{
  // Start the threads
  _beginthread(Thread One, 0, NULL);
  _beginthread(Thread Two, 0, NULL);

  // Wait for the threads to finish
  while (( false == g_bThreadOneFinished)
  || ( false == g_bThread Two Finished))
  {
    Sleep(1);
  }

  // Print the result
  printf("Result: %i\n", g_iResult);
}
```

높은 수준에서, 가장 일반적인 레이스 조건에 대한 징후는 프로그램 행위에 대한 비결정(확정)성입니다. 보다 낮은 수준의 경우에는, 다수의 스레드들 간에 공유되어 있는 변수들에 저장된 값에 대한 예측 불능을 생각할 수 있습니다.

레이스 조건은 전혀 일어날 것 같지 않은 상황에서 종종 발생하기 때문에 결함을 수정하거나 시험을 하기 대단히 어렵습니다. 징후(Symptom)는 경우에 따라 결함 수정자(디버거)를 부가하거나 추적 메시지를 추가하면 사라지기도 합니다. 이와 같은 현상을 (양자 물리학에서 사용되는 하이젠버그(Heisenberg)의 불확실성 원리에서 그 명칭을 차용하여) 하이젠버그(HeisenBug)라고 합니다.

소스-레벨 디버거는 일반적으로 그다지 도움이 되지 않기 때문에 (그러한 것들은 프로그램 타이밍에 직접적으로 영향을 미칩니다), 최선책은 메시지 추적을 활용하는 것입니다. 중요한 자원 활용을 함에 있어 주변에 트레이스(추적) 메시지를 추가하여, 모든 사항을 기록하고 이벤트의 순서가 예측했던 것과 맞는가를 확인합니다.

구조적으로, 일반적으로 활용되는 해결법은 “공유가 될 가능성이 있는” 자원에 대하여 단 하나의 유일한 스레드 접속을 부여하고, 나머지 다른 스레드들은 “공유가 될 가능성이 있는” 자원에 스레드 간의 커뮤니케이션(inter-thread communication)을 통해 접속을 하도록 하는 것입니다.

3. 데드락(교착 상태)과 라이브락

데드락이란 두 개 이상의 스레드가 순서대로 돌아가며 고정되는(locked) 자원에 대한 순서가 오기를 기다릴 경우에 발생하는 조건입니다. 멀티스레드 프로그램의 경우, 락(lock) 메커니즘은 병렬 프로그래밍의 기본적이고 근원적인 부분이기 때문에 흔히 발생합니다.

라이브락(livelock)이란 데드락과 매우 유사한 것입니다. 둘 간의 차이점은, 데드락의 경우에는, 두 개 이상의 스레드가 서로 대기하며 고정되어 있는 상태로, 라이브락

상태에서는 고정이나 정지가 되지 않으며, 끊임없이 다양한 자원들을 획득하고자 하며 역동적으로 서로 고정을 시킨다는 점입니다.

때때로, 데드락은 동일한 스레드 내에서 발생할 수도 있습니다. 그러한 상태를 자체 데드락(self deadlock)이라고 합니다. 자체 데드락이 데드락과 동일한 징후를 보인다 할지라도, 진정한 의미에서의 데드락은 아닙니다.

아래 예는 전형적인 데드락의 경우에 대한 것입니다. 두 개의 스레드에서 일어나는 실행 순서를 살펴 보시기 바랍니다:

- ThreadOne acquires g_hMutexOne
- ThreadTwo acquires g_hMutexTwo
- ThreadOne blocks attempting to acquire g_hMutexTwo
- ThreadTwo blocks attempting to acquire g_hMutexOne)

스레드 1(ThreadOne)과 스레드 2(ThreadTwo)는 다른 스레드가 보유하고 있는 것이 확실한 뮤텍스(mutex) 상에서 무한정으로 기다리기만 합니다. 이와 같은 데드락을 어떻게 하든 풀지 않고는, 프로그램은 작동하지 않을 것입니다.

```
//
// Deadlock.cpp
//
// Example of a deadlock
//
//

#include <windows.h>
#include <stdio.h>
#include <process.h>

HANDLE g_hMutex One;
HANDLE g_hMutex Two;
Bool g_bThread One Finished = false;
bool g_bThread Two Finished = false;

void ThreadOne( void *)
{
  // Get first mutex
  print f("ThreadOne ask for g_hMutexOne\n");
  Wait For Singl eObject(g_hMutexOne, INFINITE);
  print f("ThreadOne gets g_hMutexOne\n");

  // Wait some time, so the second thread can get the second mutex
  Sleep(100);

  // Try to get the second mutex. We will wait indefinetly here as
  // the second mutex is already owned by ThreadTwo
  printf("ThreadOne ask for g_hMutexTwo\n");
  Wait For Single Object(g_hMutexTwo, INFINITE);
  printf("ThreadOne gets g_hMutexTwo\n");

  // Release the two mutex
  Release Mutex(g_hMutexTwo);
  Release Mutex(g_hMutexOne);

  // Finished
  g_bThreadOneFinished = true ;
  _end thread();
}

void Thread Two(void*)
{
  // Get the second mutex
  printf("ThreadTwo ask for g_hMutexTwo\n");
  Wait For Single Object(g_hMutexTwo, INFINITE);
  printf("ThreadTwo gets g_hMutexTwo\n");

  // Wait some time, so the first thread can get the first mutex
  Sleep(100);

  // Try to get the first mutex. We will wait indefinetly here as
  // the first mutex is already owned by ThreadOne
  printf("ThreadTwo ask for g_hMutexOne\n");
  Wait For Single Object(g_hMutexOne, INFINITE);
  printf("ThreadTwo gets g_hMutexOne\n");
  // Release the two mutex
  ReleaseMutex(g_hMutexOne);
  ReleaseMutex(g_hMutexTwo);

  // Finished
  g_bThreadTwoFinished = true;
  _endthread();
}

int main()
{
  // Create the two mutex
  g_hMutexOne = CreateMutex(NULL, FALSE, NULL);
  g_hMutexTwo = CreateMutex(NULL, FALSE, NULL);

  // Start the threads
  _beginthread(ThreadOne, 0, NULL);
  _beginthread(ThreadTwo, 0, NULL);

  // Wait for the threads to finish
  while (( false == g_bThreadOneFinished)
  || ( false == g_bThreadTwoFinished))
  {
    Sleep(1);
  }

  // Free the two mutex
  CloseHandle(g_hMutexTwo);
  CloseHandle(g_hMutexOne);
}
```

데드락에 대한 가장 일반적인 징후는 정지 또는 충돌(crash)입니다.

다행스럽게도, 데드락 소스를 찾는 것은 일반적으로 어렵지 않습니다. 데드락이 일어나기 위해서는, 적어도 하나의 스레드가 락(lock)을 보유하고 있고, 적어도 하나의 스레드가 락(lock)을 획득하고자 하는 시도를 하여야만 합니다.

소스-레벨 디버거(결함 수정자)를 부가하고, 프로그램을 중지시키면 어떠한 락(lock)과 스레드가 관련이 있는 것인지 즉시 알 수 있게 될 것입니다. 어떠한 락(lock)과 스레드가 관련이 있는가를 아는 것만으로는 충분하지 않으며, 병렬 프로그램을 하나씩 살펴 보는 것도 굉장히 시간이 많이 소요되는 더딘 과정입니다.

여기서 다시, 추적 메시지에 의존을 할 수 밖에 없게 됩니다. 추적(trace) 메시지를 각각의 락(lock) 메커니즘 내부 또는 주변에 추가하여 모든 사항을 기록합니다. 그런 후에, 잠금(locking) 순서가 예상한 것과 같은가를 확인합니다.

구조적으로 볼 때, 데드락에 대한 해결책은 아래와 같습니다:

- 스레드가 한 번에 하나의 락(lock)만을 보유할 수 있도록 하는 방법.
- 스레드가 작동을 하기 위해 필요로 하는 모든 리소스(자원)을 고정(lock)시키는 방법

위와 같은 해결 방법들은 과중한 제약을 부과하는 것이기에 매 번 사용하는 것은 불가능할 수도 있지만, 항상 염두에 두는 것이 좋습니다.

4. 통신(커뮤니케이션) 불일치

통신(커뮤니케이션)의 불일치는 대응되는 수신과 발신이 서로 일치하지 않거나 또는 수신과 대응하는 발신이 일치하지 않을 경우에 일어납니다. 이는 적절하게 계획된 것이 아니거나 또는 적절하게 계획이 되었다 할지라도 메시지의 순서가 예상한 것과는 다를 경우에 발생합니다. 통신(커뮤니케이션)의 불일치는 흔히 데드락(deadlock)으로 확대되곤 합니다.

통신(커뮤니케이션)의 불일치는 스레드, 프로세스, CPU, 네트워크에서 서로 상이한 수준으로 일어납니다. 이는 API 또는 통신(커뮤니케이션) 프로토콜이 무엇이건 간에, 서로 다른 소프트웨어 요소들 간에 통신(커뮤니케이션)이 발생하는 곳에서는 문제가 됩니다.

아래 예에서, 우리는 스레드 간에 메시지를 통과하도록 하기 위하여 매우 원시적인 메커니즘을 사용하였습니다. 메인 스레드 외에, 스레드 1 과 스레드 2 라는 두 개의 스레드가 있습니다. 메인 스레드(main thread)는 “스레드 1:OK(ThreadOne: OK)”라는 메시지가 뜰 경우 스레드 1 을 통과시킵니다. 이와 같은 메시지를 기다리던 스레드 1 은 이를 취하여 스레드 2 로 “스레드 2:OK(ThreadTwo: OK)”라는 메시지를 보냅니다. 불행하게도, 스레드 2 는 “스레드 2:OK(ThreadTwo: OK)” 가 아닌 “스레드 2:NOTOK(ThreadTwo: NOTOK)”라는 메시지를 기다리고 있었기에, 스레드 2(ThreadTwo)는 메시지를 받아 들이지 않고, 원하는 메시지를 무한정 기다리게 될 것입니다.

```
//
// Mistmatched.cpp
//
// Show mismatched communication
//
//

#include <windows.h>
#include <stdio.h>
#include <process.h>

HANDLE   g_hMutex;
char     g_achMessage[64];
bool     g_bThreadOneFinished = false ;
bool     g_bThreadTwoFinished = false ;

void ThreadOne( void *)
{
  do
  {
    // Wait some time
    Sleep(1);

    // Get access to the message
    WaitForSingleObject(g_hMutex, INFINITE);

    // If we get an OK message, send an OK message to ThreadTwo
    if (0 == strcmp(g_achMessage, "ThreadOne: OK"))
    {
      printf("ThreadOne received a message\n");
      printf("ThreadOne send a message to ThreadTwo\n");
      strcpy(g_achMessage, "ThreadTwo: OK");
      g_bThreadOneFinished = true ;
    }

    // Free access to the message
    ReleaseMutex(g_hMutex);
  }
  while ( false == g_bThreadOneFinished);

  // Clean up
  _endthread();
}

void ThreadTwo(void*)
{
  do
  {
    // Wait some time
    Sleep(1);

    // Get access to the message
    WaitForSingleObject(g_hMutex, INFINITE);

    // If we get an OK message, finish the thread.
    // Unfortunatly, the message we are waiting for
    // is not the right one
    if (0 == strcmp(g_achMessage, "ThreadTwo: NOTOK"))
    {
      printf("ThreadTwo received a message\n");
      g_bThreadTwoFinished = true ;
    }

    // Free access to the message
    ReleaseMutex(g_hMutex);
  }
  while ( false == g_bThreadTwoFinished);

  // Clean up
  _endthread();
}

int main()
{
  // Initialize the message
  strcpy(g_achMessage, "");

  // Create the mutex
  g_hMutex = CreateMutex(NULL, FALSE, NULL);

  // Start the threads
  _beginthread(ThreadOne, 0, NULL);
  _beginthread(ThreadTwo, 0, NULL);

  // Send a message to ThreadOne
  printf("Main send a message to ThreadOne\n");
  WaitForSingleObject(g_hMutex, INFINITE);
  strcpy(g_achMessage, "ThreadOne: OK");
  ReleaseMutex(g_hMutex);

  // Wait for the threads to finish
  while (( false == g_bThreadOneFinished)
  || ( false == g_bThreadTwoFinished))
  {
  Sleep(1);
  }

  // Free the mutex
  CloseHandle(g_hMutex);
}
```

통신(커뮤니케이션) 불일치의 가장 일반적인 징후는 정지(freeze)입니다.

위에서와 마찬가지로, 소스-레벨 디버거(결함 수정자)를 부가하고, 프로그램을 중지시키면 그에 대한 정보를 알 수 있지만, 이로써 문제점에 대한 진짜 원인을 알고 이해하기에는 충분하지 않습니다. 프로그램을 하나씩 세밀히 살펴 보는 것은 굉장히 시간이 많이 소요되는 더딘 과정일 뿐만 아니라, 통신(커뮤니케이션) 타이밍을 바꾸어 놓을 수도 있습니다.

여기서 다시, 추적 메시지에 의존을 할 수 밖에 없게 됩니다. 디버깅(debugging)을 쉽게 할 수 있는 방법들로는 아래와 같이 몇 가지가 있습니다:

첫 번째 방법은 식별 기능(code facilities)에 스레드, 프로세스 등의 사이를 통과시킨 모든 메시지를 버리도록 설정해 두는 것으로서 ... 모든 메시지를 확인할 수 있게 해 줌으로써 실제로 어떤 일이 일어나고 있는가에 대해 정확하게 파악할 수 있도록 해 줄 것입니다. 즉, 어떠한 메시지가 발신되었고, 어떤 순서로 발신되었는지 그리고 성공적으로 수신이 되었는가를 알 수 있게 됩니다.

두 번째 방법은 메시지 대기열(queue)을 사용하는 것입니다. 메시지 대기열(queue)을 사용함으로써, 시스템의 현황을 아주 쉽게 파악할 수 있습니다. 서로 다른 대기열(queue)을 사용하면 훨씬 도움이 됩니다. API 를 통과한 일반적인 메시지의 경우에 있어서는, 발신 대기(pending sends), 수신 대기(pending receives), 미예측 메시지(unexpected messages)와 같은 세 가지의 대기열(queue)이 적용되게 됩니다.

5. 최선의 관행

가장 중요한 점은 애초부터 디버깅(debugging)에 대한 대비를 해 두는 것입니다. 병행성 버그(concurrency bugs)를 피할 수 있는 것이라고 생각해서는 안 되며, 사용 중인 코드(code)에서 병행성 버그가 발생하는 것을 기다리거나 방관해서도 안 됩니다. 코드를 작성하기에 앞서 병행성에 대한 결함 수정(debug)을 어떻게 할 것인가에 대한 대비를 해 둘 필요가 있습니다.

메시지를 통과시키는 데에 있어서도 단 하나의 방식만을 사용하는 것이 좋습니다. 그렇게 함으로써, 통신(커뮤니케이션) 불일치 상황이나 데드락이 발생했을 경우 한 곳만 조사해 보면 될 것입니다. API 를 통과한 메시지를 활용할 수 있는 경우라면, 다른 방법을 애써 찾지 말고 바로 그러한 메시지를 활용하십시오.

“모든 것을 하나의 스레드를 통해 관리하도록 합시다." 사용하고 있는 애플리케이션(application)을 n 개의 스레드(thread)를 가지고 실행할 수 있도록 계획을 세우십시오. 이와 같은 방법을 통해, 병행성 디버깅(결함 수정)의 단계를 거쳐야 하는 상황에 빠지기 전에 직렬 디버깅(serial debugging)의 단계를 통해 애플리케이션을 사용할 수 있을 것입니다. 직렬성 애플리케이션을 디버그(debug)하는 편이 훨씬 수월하기 때문에, 저항이 가장 적은 경로를 선택하여야 하며, 직렬 애플리케이션으로 사용하는 애플리케이션을 운용하는 경우 모든 버그를 발견하자 마자 즉시 제거하여야 합니다. 당신이 사용하는 애플리케이션에 “일반적인” 버그들에 따른 문제가 없다면, 병행성 버그(concurrency bugs)를 처리하고 대처하기가 훨씬 수월할 것입니다.