

# Practical Multi-core Game Programming

김성익(noerror@hitel.net)  
2009.9.13

Spirit of Flame  
3D RealTime Graphics Programming Study  
*Kasa*

# 개요

- 멀티 코어의 시대
  - 듀얼 코어, 쿼드 코어의 일반화
  - 새로운 패러다임
  - 필수
  - 동기화등이 복잡하지 않은 모델 개발 필요
    - 모든 Architect의 NEW JOB
- 멀티 코어 시대 관련 이슈들 소개
- 적용중인 아이디어 일부 제안 *아쉽게도 시도 중이라 검증 단계인 아이디어도 존재. 개인적으로 문의하시면 좀 더 진행된 세부적인 정보를 드릴 수 있습니다*

# 게임 프로그래밍

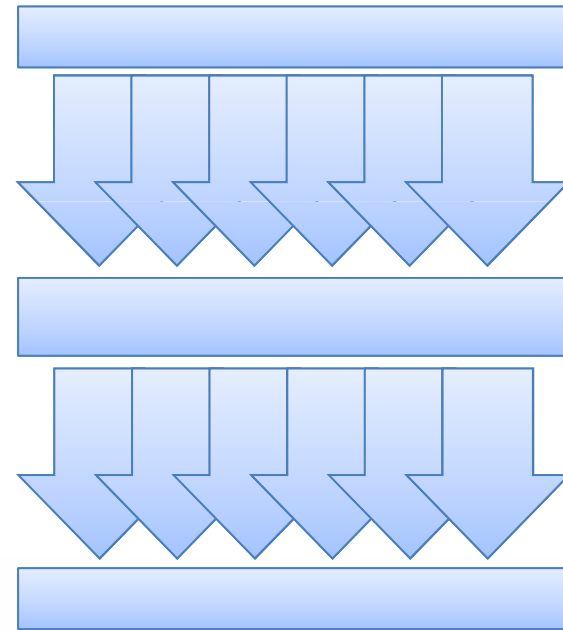
- 게임에서의 멀티 쓰레딩
  - IO 처리를 위한 멀티 쓰레딩 (병렬성, 빠른 반응)
    - 주로 서버 모델
    - 클라이언트의 IO 관련 로딩 모델
  - CPU활용을 극대화 하기 위한 멀티 쓰레딩 (많은 연산)
    - 클라이언트 프레임워크
    - (3가지 실행모델)

# 실행 모델

- 실행 3 모델 (from Getting More from Multicore / Ian Lewis)
  - BSP (Bulk Synchronous Processing)
  - CSP (Communicating Sequential Processing)
  - TASK POOL
- IO를 고려하지 않은 모델
  - 목표는 CPU 활용도 100%
  - 게임 프레임 워크에서 고려

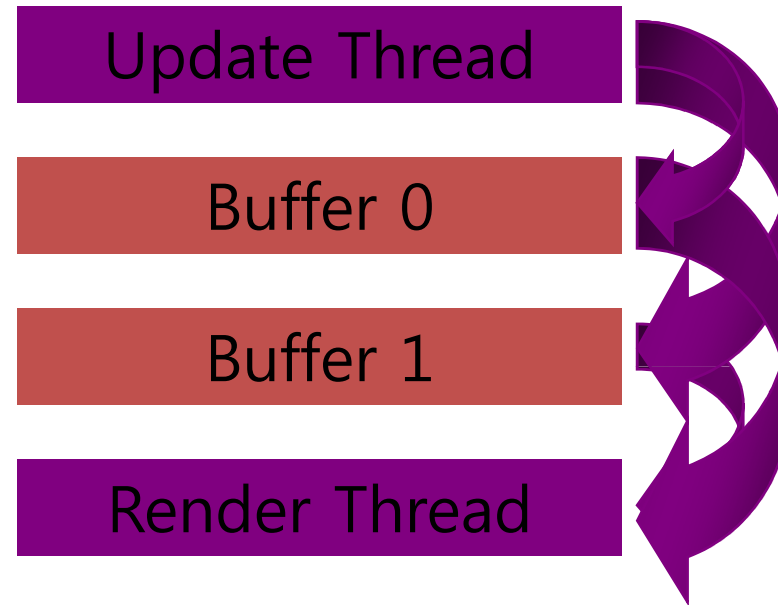
# BSP(1)

- BSP (Bulk Synchronized Process)
  - 사이클
    - 스레드 별로 작업할당
    - 병렬로 실행
    - 타이밍 동기화
    - 반복
  - 이상적인 형태
  - 게임에는 적용하기 힘든 모델
    - 프로세스간의 연관성
      - AI, 물리 내용이 렌더링에 영향을 줌



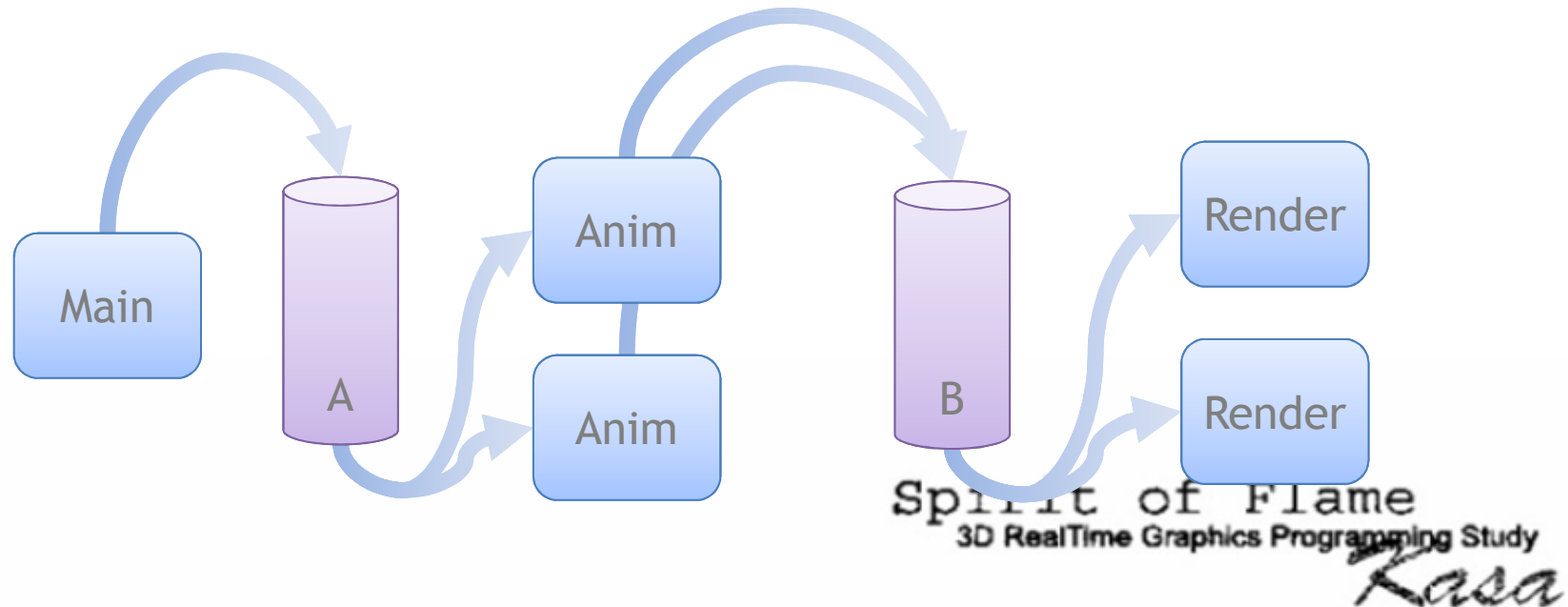
# BSP(2)

- 구조를 변경하면 일부 가능성도 있음
- 참고 : Coding for Multiple Cores / Bruce Dawson & Chuck Walbourn



# CSP

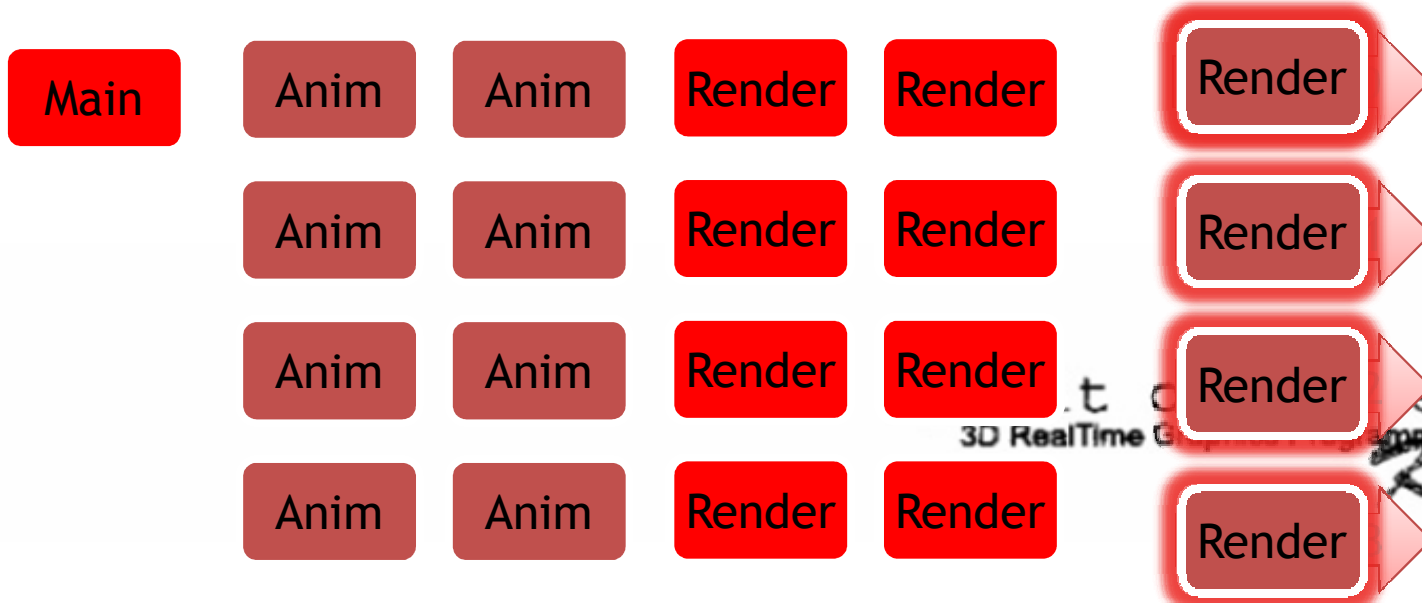
- CSP 모델 (Communicating Sequential processing)
  - 동기화 객체를 이용해 연쇄적으로 실행되도록 프로그래밍
  - 각 기능별 스레드 존재
    - 각기 통신을 통해 실행 권을 전달
    - 대기 중에는 블록 상태 (Sleep 상태)



# Task Pool

- TASK POOL

- 실행 가능한 스레드 풀 구성
- 각 스레드에서 Task(JOB)을 가져와서 실행
- CPU 활용도가 가장 높다
- CSP에 비해 많은 스레드를 할당하지 않아도 됨
  - Seirialize 할 때 동기화 객체에 덜 의존적
- Context Switching 이 많이 발생하지 않음
- 연관되는 Task의 실행 순서를 맞추기 위한 약간의 복잡한 매니징이 필요



# (recall)

- 쓰레드 풀
  - n개의 쓰레드에서 각각 필요한 작업을 할당 받아서 실행하는 방식
    - 주로 각 쓰레드는 특화된 기능이 아닌 일반적인 기능들을 실행하는 게 일반적
  - 서버 IOCP 활용하는 모델에서 주로 사용
- Context Switching
  - 현재 실행중인 쓰레드에서 다른 쓰레드로 실행권을 넘기 과정으로 statck 등 프로그램 실행에 필요한 정보(context)를 복사가 이루어짐
  - 잦은 Context Switching은 속도 저하

# 멀티 스레딩 사례

- IOCP를 사용한 네트워크 서버 모델
- 클라이언트 P2P 모듈
- 사운드 스트리밍
- 우아한 종료
- PhysX 사례
- 텍스처 지연 로딩
- 멀티 쓰레드 렌더링
- 기타

# 일반적인 IOCP 서버 모델

- 보통의 IOCP 활용한 서버 쓰레드 풀 모델
  - IOCP
    - 사이클
      - 쓰레드 풀에서 GQCS 함수로 블록 상태에서 대기
        - » 등록된 소켓에서 이벤트가 발생하면 블록 상태에서 벗어남
      - 이벤트 발생하면 GQCS로 이벤트 발생됨
        - » Context Switching 되면서 실행 권을 얻음
      - 메시지 처리하고 소켓 이벤트 등록
        - » 메시지 처리 중 SQL 등으로 블록 되고 다른 쓰레드 풀로 Context Switching됨
      - 반복

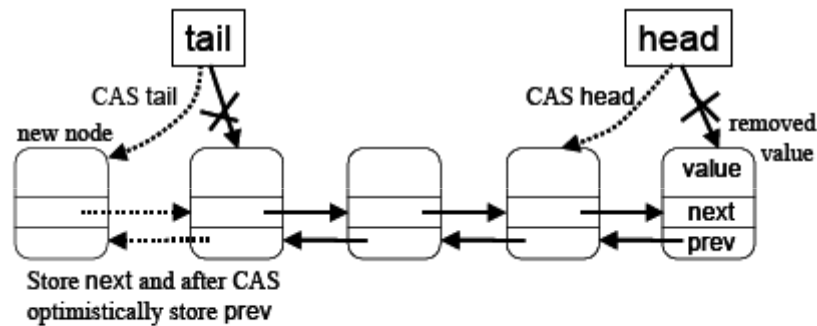
# 일반적인 P2P 쓰레드

- P2P 처리를 위해, 빠른 response가 필요한 경우
  - mainloop 에서 패킷 처리하는 것보다는 별도의 쓰레드로 운영한다면 패킷이 도착한 시간을 좀 더 정확하게 얻을 수 있음
    - 패킷을 받는 쓰레드를 운영하고, 패킷을 메시지 큐에 넣어두고, mainloop 안에서 메시지 큐의 패킷을 가져와 처리
    - ping등의 일부 패킷은 쓰레드 내에서 처리 (mainloop에서 처리하는 것보다 좀 더 정확한 ping 타임을 얻을 수 있음)
  - 패킷을 넣고, 빼는 부분은 동기화 필요
    - 동기화 객체 사용하지 않고 Lock-Free FIFO QUEUE로 구현 가능

# (recall)

- Lock-Free Queue

- 동기화 객체를 사용하지 않고 리스트 처리



- 동기화 객체의 사용 => 동기화 객체 사용으로 인한 부하 + 컨텍스트 스위칭될 빌미~ (동기화 객체 사용을 줄일 수 있다면 줄인다.)
- (참고 : Saints Row Scheduler / Randall Turner)

- STL 동기화 정책

- 읽는 연산은 동기화를 하지 않아도 됨
- 쓰는 연산 시에는 쓰는 스레드에서 독점적으로 사용해야 함 (동기화 필요)

# 스트리밍 사운드

- 스트리밍 사운드 출력을 위한 로딩 스레드
  - 앞으로 출력될 버퍼에 내용을 미리 채워둠
  - 아래 pseudo 코드처럼 구성가능

```
while(stopsound == false)
{
    if (loadedtick - currentplaytick < BUFFERTICKSIZE)
    {
        loadedtick += read(loadedtick, currentplaytick + BUFFERTICKSIZE - loadedtick);
    }

    Sleep(100);
}
```

- 만약 검사하고 읽는 부분은 mainloop에 둔다면 로딩하는 IO작업으로 프레임 드롭이 생길 수 있음 => 스레드로 분리가 필요
- sleep등을 통해서 스레드의 부하를 줄여줌
  - 블록 되도록 하지 않으면 불필요하게 CPU를 많이 사용하게됨

# 우아한 종료(1)

- 쓰레드 내에서 변수들을 통해서 리턴으로 종료
  - TerminateThread 등 절대 사용 X
    - 동기화 객체 사용 중에 외부에서 terminate 시킨다면 ???

```
void loop()
{
    while(terminate == false)
    {
        Sleep(100);

        thread job
    }
}

void start()
{
    terminate = false;
    handle = beginthread(loop..);
}

void stop()
{
    terminate = true;
    waitforsingleobject(handle, INFINITE);
}
```

# 우아한 종료(2)

- 이벤트 객체를 사용하면 블록 중 일 때 바로 종료 시킬 수 있다
  - 이벤트 객체는 외부에서 signal 제어를 할 수 있다

```
void loop()
{
    while(terminate == false)
    {
        if (waitforsingleobject(closesignal, 100) == timeout)
        {
            thread job
        }
    }
}

void start()
{
    terminate = false;
    closesignal = CreateEvent(NULL, TRUE, FALSE, NULL);
    handle = beginthread(loop..);
}

void stop()
{
    terminate = true;
    SetEvent(closesignal);
    waitforsingleobject(handle);
}
```

# (recall)

- 동기화 객체

- Semaphore

- 동기화 된 영역에 n개만 통과할 수 있도록 함

```
wait();
```

```
critical section
```

```
exit();
```

- Mutex

- Semaphore 와 같지만 1개만 통과할 수 있음

- Critical Section (win32)

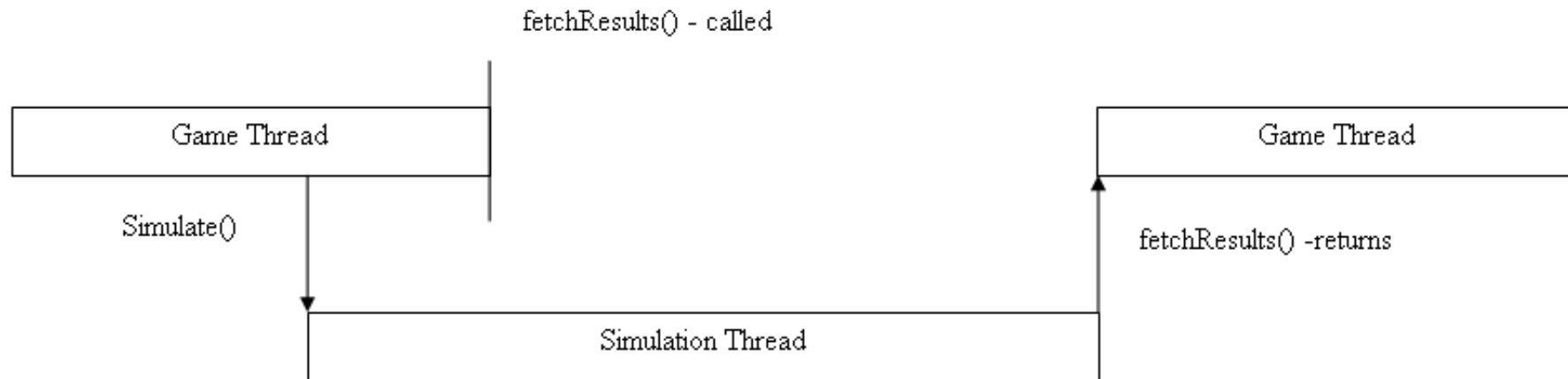
- Mutex 와 같지만 가능한 경우 좀 더 가볍게 처리. Spinlock 이나 같은 스레드에서 중복 호출 시 처리 추가적인 기능

- Event (win32)

- Mutex 와 유사하지만 외부에서 시그널 처리를 할 수도 있음

# PhysX 사례

- 독자적인 스레드 운영
  - 선택적으로 제어도 가능
- 특화된 기능이라면 별도의 스레드 운영도 좋은 모델
  - 스레드 로딩, 사운드 스트리밍, 파티클 계산 등등



# 텍스처 지연 로딩(1)

- 게임 중 텍스처, 메시 등으로 인한 프레임 드롭을 없애기 위해 Proxy 패턴 활용

```
class CTexture
{
public :
    virtual IDirect3DTexture * GetTexture() = 0;
};

CTexture * LoadTexture();
```

```
CTexture * check = LoadTexture("checkbox.dds");
```

- 위가 원래 루틴이라면 아래 pseudo 코드처럼 지연로딩로딩 구현 가능 (로딩 쓰레드 혹은 태스크 풀이 있다고 가정 - Threadmain 이 호출됨)

```
class CTextureProxy : public CTexture, public CThreadJob
{
public :
    CTextureProxy(const char * fname) { m_Filename = fname; m_Texture = NULL; }
    void ThreadMain() { m_Texture = LoadTexture(m_Filename); }
    bool IsLoadingComplete() { return m_Texture != NULL; }
    IDirect3DTexture * GetTexture() { return m_Texture == NULL ? NULL : m_Texture->GetTexture(); }
private :
    std::string m_Filename;
    CTexture * m_Texture;
};
```

```
CTextureProxy * tex = new CTextureProxy("checkbox.dds");
ThreadPool.Insert(tex);
```

```
CTexture * check = tex;
```

# 텍스처 지연 로딩(2)

- 로딩이 안된 상태에서는 미리 로딩된 저해상도 텍스처를 리턴하도록 응용 가능
- 만약 texture 로딩 큐에 텍스처가 100여장이 로딩을 대기하고 있는데, 프레임 드롭은 생기더라도 화면에 찍힐 녀석은 먼저 읽어야 한다면 아래처럼 바로 로딩하도록 응용 가능

```
class CTextureProxy : public CTexture, public CThreadJob
{
public :
    CTextureProxy(const char * fname) { m_Filename = fname; m_Texture = NULL; }
    void ThreadMain()
    {
        EnterCriticalSection();
        if (m_Texture == NULL)
            m_Texture = LoadTexture(m_Filename);
        LeaveCriticalSection();
    }
    bool IsLoadingComplete() { return m_Texture != NULL; }
    IDirect3DTexture * GetTexture()
    {
        if (m_Texture != NULL)
            ThreadMain();
        return m_Texture->GetTexture();
    }
private :
    std::string m_Filename;
    CTexture * m_Texture;
};
```

# 텍스처 지연 로딩(3)

- 이 구조는 텍스처 뿐 아니라, 메시, 애니메이션에도 공통적으로 적용 가능
- 만약 Task Pool 방식으로 Object 의 Update 와 Draw 를 각기 Job으로 설정해서 queue 에 넣을 경우 별도의 매니징 없이 전 페이지의 pseudo 코드처럼 순서를 조절 가능 (예를 들어 스레드 1에서 Update 가 실행중인데, 스레드 2에서 Draw 가 호출되었다면 ??)

# 멀티 쓰레드 렌더링(1)

- D3D 함수를 멀티 쓰레드로 호출할 수 있다고는 하지만 아래처럼 구성된 Render 함수를 여러 쓰레드에서 동시에 호출한다면 ??

```
SetRenderState  
SetVertexStream  
DrawPrimitive
```

- DrawPrimitive 시점에서 설정된 RenderState를 보장받을 수 없음
- 해결방법

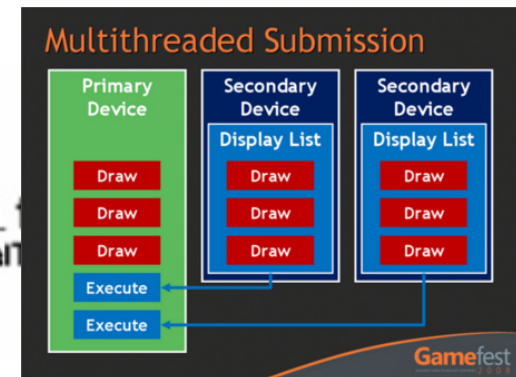
- D3D 관련 함수 호출들을 (근성으로) Command Buffer 같은 메시지 형태로 매핑한 후 인스턴스 단위로 큐에 넣고, 렌더 쓰레드는 대기하다가 큐에 메시지가 쌓이면 해당 메시지의 함수를 호출하는 방식 <- Unreal3 (예상)
- 아래처럼 인스턴스 단위로 동기화 처리를 해서 순서대로 D3D함수가 호출되도록 동기화 처리

```
wait();
```

```
SetRenderState  
SetVertexStream  
DrawPrimitive
```

```
exit();
```

- D3D 11이후 버전인 경우 Display List 를 활용 ^^



# 멀티 쓰레드 렌더링(2)

- 동기화 함수를 이용해서 멀티 쓰레딩을 구현하는 경우
  - 함수들 호출하는 곳들이 분산되어 있어서 Draw 함수 전체에 걸쳐 동기화를 해야 하는 경우

```
void SetVertexBuffer(IDirect3DVertexBuffer * buff)
{
    m_pD3D->SetVertexBuffer(buff);
}

void DrawPrimitive(...)
{
    m_pD3D->DrawPrimitive(..);
}
```

- 위의 함수를 아래처럼 구성한다면 동기화를 최종 DrawPrimitive 하는 순간으로 축소할 수도 있다 (TLS 를 적절히 활용할 경우 굉장히 단순화시킬 수 있다)

```
__declspec(thread) IDirect3DVertexBuffer * s_buff;

void SetVertexBuffer(IDirect3DVertexBuffer * buff)
{
    s_buff = buff;
}

void DrawPrimitive(...)
{
    wait();
    m_pD3D->SetVertexBuffer(s_buff);
    m_pD3D->DrawPrimitive(..);
    exit();
}
```

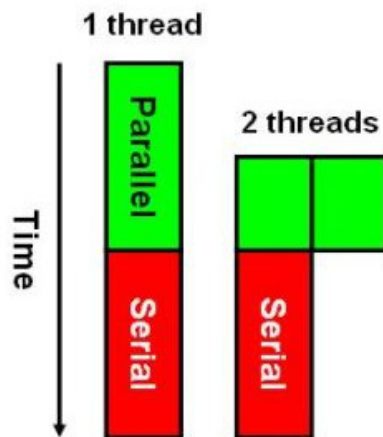
# 기타

- 테스크 풀 정책 아이디어
  - Priority
  - Serialize
- 최적의 스레드 개수 이슈
  - IO 위주 모델
  - Task 위주 모델
- 코어 간의 메모리 공유에 따른 최적화 이슈
- 동기화 복잡도와 작업 속도와의 관계

# (recall)

- 암달의 법칙

- 싱글 스레드의 프로그램에서 병렬화 처리를 한 %와 스레드의 개수를 알면 얼마나 개선되는 지 계산이 가능
- 40% 정도를 병렬처리 되도록 수정했을 경우 2CPU에서 1.25 향상, 즉 25%의 향상 예상. 30프레임 => 37.5 프레임으로 향상 (너무 실망 말자, 코어가 4개가 되면 43프레임까지 향상) 그리고 GPU idle을 줄이는 역할을 한다면 개선 폭은 비약적으로 증가



### Amdahl's Law

$$T_p = \left( \%S + \frac{1 - \%S}{N} \right) * T_s$$

$$Speedup = \frac{T_s}{T_p}$$

$T_p$	Parallel runtime
$T_s$	Serial runtime
$\%S$	Percentage of time spent in serial code
$N$	Number of processors

# 질문/답

# 참고

- Coding for Multiple Cores, Bruce Dawson & Chuck Walbourn
- Multicore Strategies for Games, Aaron Lanterman
- Getting More from Multicore, Ian Lewis
- Threading 3D Game Engine Basics, Henry Gabb and Adam Lake
- Saints Row Scheduler, Randall Turner
- Lockless Programming Considerations for Xbox 360 and Microsoft Windows, Bruce Dawson
- An Optimistic Approach to Lock-Free FIFO Queues, Edya Ladan-Mozes and Nir Shavit
- Developing Gears of War in Unreal Engine 3, Michael Capps
- Multi-Threaded Rendering for Games, Matt Lees