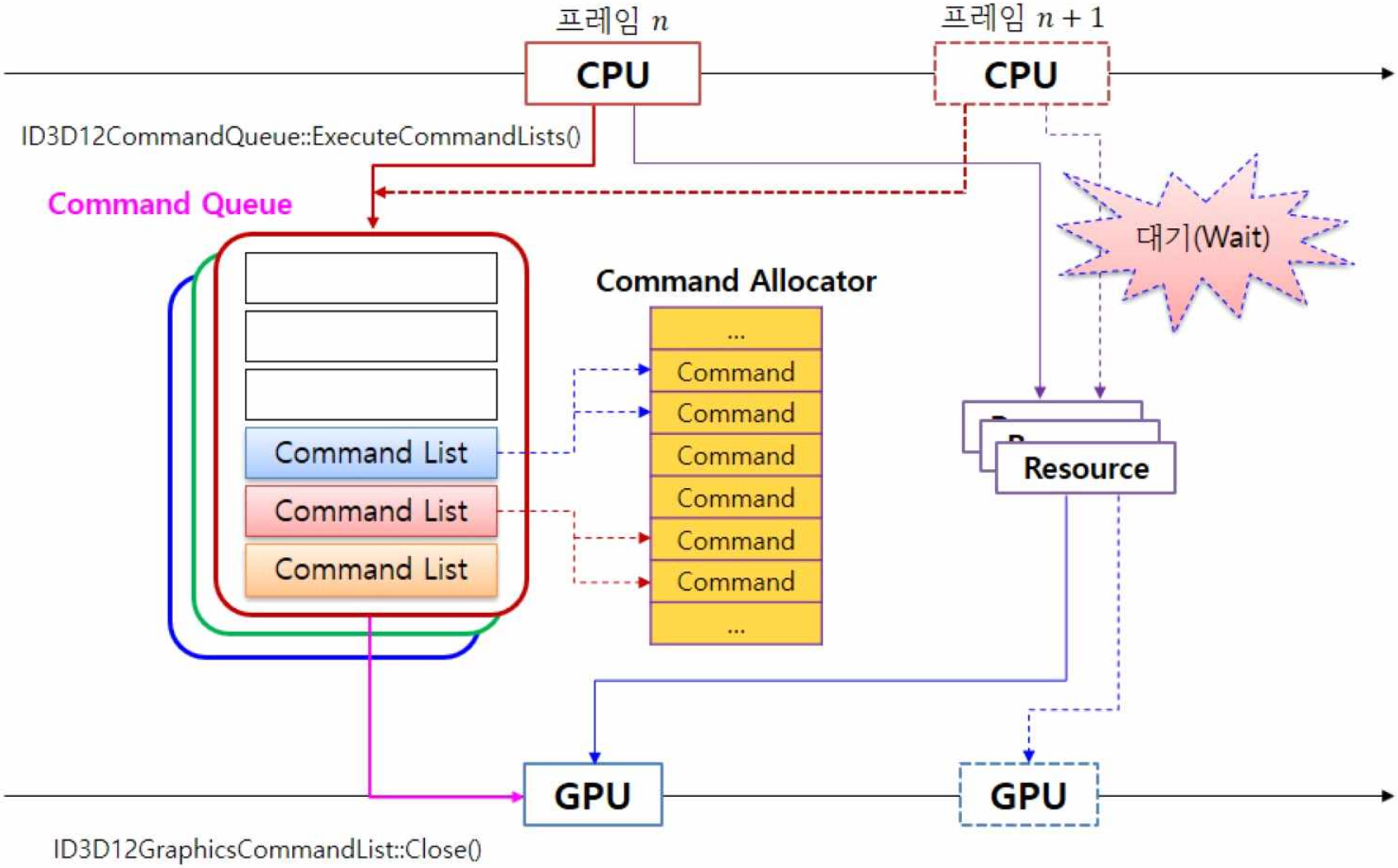
Game Programming with DirectX

Direct3D Multithreading

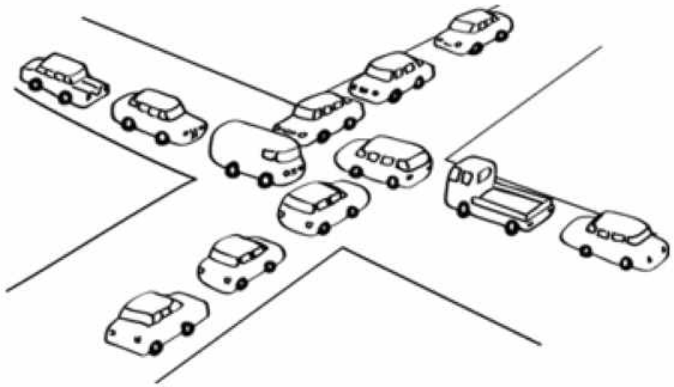
Direct3D Multithreading

- **렌더링 과정**
  - CPU가 리소스를 생성(Write)하고 GPU는 리소스를 소비(Read)함



Direct3D Multithreading

- **동기화 객체(Synchronizing Objects)**
  - 병목 현상(Deadlock, Race Condition)
    병목 현상을 없애려면?
  - 공유 자원(Shared Resource)들에 대한 동기화(Synchronization)
    공유되는 자원에 대한 접근(Read, Write)이 여러 쓰레드에서 동시에 발생하면?
    하나의 쓰레드가 쓰고 있을 때 다른 쓰레드가 읽기를 한다면?
    두 개의 쓰레드가 동시에 공유 메모리에 쓰기(Write)를 하지 못하게 하려면?
    어떤 쓰레드가 쓰기를 하고 있을 때 읽기를 하려는 쓰레드들은 기다려야 함
  - 동기화 객체
    - **Event, Mutex, Semaphore, Waitable Timer Object**
      CreateEvent(), CreateMutex(), CreateSemaphore(), CreateWaitableTimer()
    - Semaphore, Critical Section
    - 상태(State): 활성화(Signaled), 비활성화(Nonsignaled)
  - 대기 함수(Wait Function)
    비활성화 상태의 동기화 객체가 활성화될 때까지 쓰레드의 실행을 막음(Blocking)
    - WaitForSingleObject()
    - WaitForMultipleObjects()



Direct3D Multithreading

- **멀티 쓰레딩(Multithreading)**
  - **프로세스(Process)**
    실행되는 프로그램(Program)
    각 프로세스는 프로그램을 실행하기 위하여 필요한 리소스를 제공
    가상 주소 공간, 코드, 시스템 객체들에 대한 핸들, ...
    각 프로세스는 단일 쓰레드(주 쓰레드: Primary Thread)로 실행됨
    프로세스는 추가적인 쓰레드를 생성할 수 있음(CreateThread)
  - **쓰레드(Thread)**
    실행을 스케쥴링할 수 있는 프로세스 내의 단위(함수)
    프로세스의 모든 쓰레드들은 프로세스의 가상 주소 공간과 시스템 리소스를 공유함
    각 쓰레드는 독립적인 스택 공간을 가짐
    각 쓰레드는 쓰레드 로컬 메모리(Thread Local Storage)를 가짐
    ```
    DWORD TlsAlloc();
    BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);
    LPVOID TlsGetValue(DWORD dwTlsIndex);
    BOOL TlsFree(DWORD dwTlsIndex);
    ```
  - Windows는 Preemtive Multitasking을 지원
  - 멀티 쓰레딩
    예제 프로그램(BounceMultiThread)
  - Direct3D 12 멀티 쓰레딩
    예제 프로그램(D3DMultiThreading)

Direct3D Multithreading

- **다중 쓰레드(Multiple Threads)**
  - 쓰레드의 생성(Create Thread)

```
#include <processthreadsapi.h>
```

```
HANDLE CreateThread( //함수를 호출하는 프로세스의 가상 주소 공간에서 실행할 쓰레드를 생성
  LPSECURITY_ATTRIBUTES lpThreadAttributes, //NULL: 기본 Security Descriptor를 가짐(상속 안됨)
  SIZE_T dwStackSize, //스택의 초기 크기(바이트), 0: 기본 크기(1MB)
  LPTHREAD_START_ROUTINE lpStartAddress, //쓰레드 함수
  __drv_aliasesMem LPVOID lpParameter, //쓰레드 함수의 실인자(Argument)
  DWORD dwCreationFlags, //CREATE_SUSPENDED: 실행 대기
  LPDWORD lpThreadId //쓰레드 ID
);
```

```
DWORD WINAPI ThreadProc(LPVOID lpParameter);
```

```
void ExitThread(DWORD dwExitCode);
```

```
BOOL CloseHandle(HANDLE hObject);
```

```
DWORD SuspendThread(HANDLE hThread); //쓰레드를 대기 상태로 만듬, 대기 카운트를 1 증가(디버깅 목적)
DWORD ResumeThread(HANDLE hThread); //쓰레드 대기 카운트를 1 감소, 카운트가 0이 되면 쓰레드가 실행됨
```

```
typedef struct _SECURITY_ATTRIBUTES {
  DWORD nLength; //sizeof(SECURITY_ATTRIBUTES)
  LPVOID lpSecurityDescriptor; //SECURITY_DESCRIPTOR
  BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

```
typedef struct _SECURITY_DESCRIPTOR {
  BYTE Revision;
  BYTE Sbz1;
  SECURITY_DESCRIPTOR_CONTROL Control;
  PSID Owner;
  PSID Group;
  PACL Sacl;
  PACL Dacl;
} SECURITY_DESCRIPTOR;
```

```
HANDLE OpenThread( //쓰레드 ID에 해당하는 쓰레드 핸들을 반환
  DWORD dwDesiredAccess, //SYNCHRONIZE, THREAD_ALL_ACCESS
  BOOL bInheritHandle, //TRUE이면 이 프로세서가 생성한 프로세서들이 핸들을 상속함
  DWORD dwThreadId //쓰레드 ID
);
```

Direct3D Multithreading

- 다중 쓰레드(Multiple Threads)

```
uintptr_t _beginthread( //생성된 쓰레드에 대한 핸들을 반환
   void (__cdecl *start_address)(void *), //void (__clrcall *start_address)(void *)
   unsigned stack_size, //스택 크기, 0: 기본 크기
   void *arglist //쓰레드 실행시 전달될 실인자(Argument) 리스트
);

uintptr_t _beginthreadex( //생성된 쓰레드에 대한 핸들을 반환
   void *security, //SECURITY_ATTRIBUTES, 쓰레드 핸들이 자식 프로세스들에게 상속되는 가를 결정
   unsigned stack_size, //스택 크기, 0: 기본 크기
   unsigned (__stdcall *start_address)(void *), //unsigned (__clrcall *start_address)(void *)
   void *arglist, //쓰레드 실행시 전달될 실인자(Argument) 리스트
   unsigned initflag, //쓰레드의 초기 상태, 0: 즉시 실행, CREATE_SUSPENDED: ResumeThread()
   unsigned *thrdaddr //쓰레드 ID를 저장할 32-비트
);
```

#include <process.h>

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength; //sizeof(SECURITY_ATTRIBUTES)
   LPVOID lpSecurityDescriptor;
   BOOL bInheritHandle;
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES, *LPSECURITY_ATTRIBUTES;
```

```
void start_address(void *arglist); //함수가 종료(반환)하면 쓰레드는 자동으로 종료됨
```

```
void _endthread(void); //쓰레드에 대한 핸들을 자동적으로 닫음
void _endthreadex(unsigned retval); //생성된 쓰레드에 대한 핸들을 CloasHandle() 함수로 닫아야 함
```

Direct3D Multithreading

- **동기화 객체(Synchronizing Objects)**
  - **이벤트(Event)**
    어떤 일(사건)이 발생했음을 대기하는 쓰레드에게 통지할 때
    일반적으로 어떤 작업이 종료되었음을 통지할 때(TRUE, FALSE)

#include <synchapi.h>

```
HANDLE CreateEventA(
  LPSECURITY_ATTRIBUTES lpEventAttributes, //NULL: 자식 프로세스에게 상속되지 않음
  BOOL bManualReset, //FALSE: 자동으로 리셋(Reset: 비활성화), TRUE: ResetEvent()로 리셋해야 함
  BOOL bInitialState, //초기 상태(TRUE: 활성화)
  LPCSTR lpName //이벤트의 이름
);
```

```
BOOL SetEvent(HANDLE hEvent); //활성화 상태로 설정
BOOL ResetEvent(HANDLE hEvent); //비활성화 상태로 설정
```

```
HANDLE CreateEventExA(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  LPCSTR lpName, //이벤트의 이름
  DWORD dwFlags, //초기 상태와 수동 리셋 설정
  DWORD dwDesiredAccess //접근 매스크
);
```

```
CREATE_EVENT_INITIAL_SET
CREATE_EVENT_MANUAL_RESET
```

```
HANDLE OpenEventA( //존재하는 이벤트 객체의 핸들을 열고 반환
  DWORD dwDesiredAccess, //접근할 유형, 이벤트를 생성할 때의 SECURITY_ATTRIBUTES에 따라 접근 허용됨
  BOOL bInheritHandle, //TRUE이면 이 프로세서가 생성한 프로세서들이 핸들을 상속함
  LPCSTR lpName //이벤트의 이름
);
```

Direct3D Multithreading

- **동기화 객체(Synchronizing Objects)**
  - **Mutex**
    다른 쓰레드가 뮤텍스를 소유하지 않을 때 활성화 상태(Signaled State)로 설정
    한 순간에 하나의 쓰레드가 하나의 뮤텍스를 소유할 수 있음(Mutually Exclusive)
    뮤텍스를 소유한 쓰레드가 공유 자원에 대하여 접근하고 접근이 끝나면 소유권을 반환

```
HANDLE CreateMutexA(
   LPSECURITY_ATTRIBUTES lpMutexAttributes, //NULL: 자식 프로세스에게 상속되지 않음
   BOOL bInitialOwner, //TRUE이면 호출한 쓰레드가 뮤텍스를 소유
   LPCSTR lpName //뮤텍스의 이름
);
HANDLE CreateMutexW(
   LPSECURITY_ATTRIBUTES lpMutexAttributes,
   BOOL bInitialOwner, //TRUE이면 호출한 쓰레드가 뮤텍스를 소유
   LPCWSTR lpName //뮤텍스의 이름
);
HANDLE CreateMutexExA(
   LPSECURITY_ATTRIBUTES lpMutexAttributes,
   LPCSTR lpName, //뮤텍스의 이름
   DWORD dwFlags,
   DWORD dwDesiredAccess //접근할 유형(SYNCHRONIZE, MUTEX_ALL_ACCESS)
);
```

```
#include <synchapi.h>
```

```
BOOL ReleaseMutex(HANDLE hMutex);
```

```
CREATE_MUTEX_INITIAL_OWNER
```

```
HANDLE OpenMutexW( //존재하는 뮤텍스 객체의 핸들을 열고 반환
   DWORD dwDesiredAccess, //접근할 유형(SYNCHRONIZE, MUTEX_ALL_ACCESS)
   BOOL bInheritHandle, //TRUE이면 이 프로세서가 생성한 프로세서들이 핸들을 상속함
   LPCWSTR lpName //뮤텍스의 이름
);
```

Direct3D Multithreading

- **동기화 객체(Synchronizing Objects)**
  - **대기 함수(Wait Function)**
    호출 쓰레드는 실행을 멈추고 동기화 객체가 활성화될 때까지 대기함
    호출 쓰레드는 대기 상태(Wait State)로 들어감
    동기화 객체가 활성화되거나 대기 시간이 경과할 때까지 대기함
    대기 시간이 경과하면 WAIT_TIMEOUT을 반환
    대기 시간이 0이면 대기하지 않고 즉시 반환됨
    이벤트, 뮤텍스, 쓰레드 등을 기다릴 수 있음

```
#include <synchapi.h>
```

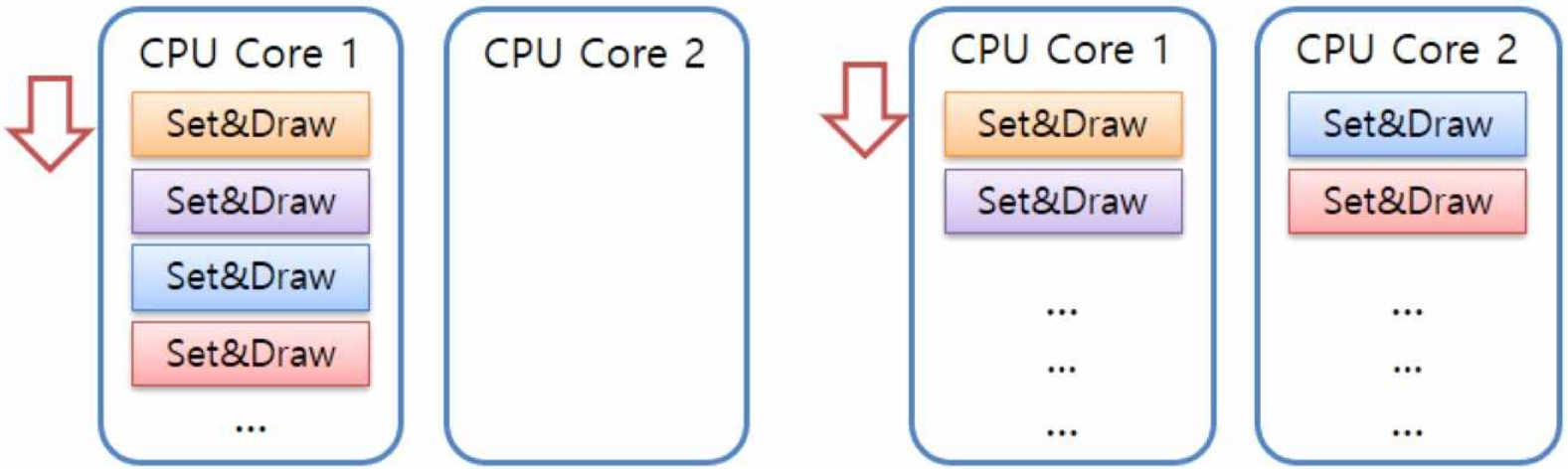
```
DWORD WaitForSingleObject( //WAIT_OBJECT_0, WAIT_TIMEOUT
  HANDLE hHandle, //동기화 객체의 핸들(SYNCHRONIZE 접근 권한을 가져야 함)
  DWORD dwMilliseconds //동기화 객체가 활성화 될 때까지 기다리는 시간(0.001초 단위), INFINITE: 무한 대기
);
```

```
DWORD WaitForMultipleObjects(//WAIT_OBJECT_0 ~ (WAIT_OBJECT_0 + nCount – 1), WAIT_TIMEOUT
  DWORD nCount, //동기화 객체의 개수
  HANDLE *lpHandles, //동기화 객체의 핸들들의 배열
  BOOL bWaitAll, //모든 동기화 객체들이 활성화 될 때까지 기다리는가, FALSE: 하나만 활성화되어도 대기 종료
  DWORD dwMilliseconds //동기화 객체들이 활성화 될 때까지 기다리는 시간(0.001초 단위), INFINITE
);
bWaitAll가 TRUE일 때, 반환값은 WAIT_OBJECT_0 ~ (WAIT_OBJECT_0 + nCount – 1) 사이의 값임
bWaitAll가 FALSE일 때, (반환값 - WAIT_OBJECT_0)는 활성화된 동기화 객체들의 가장 작은 인덱스
  동기화 객체의 핸들들의 배열을 순서대로 검사
```

Direct3D Multithreading

- **다중 쓰레드 렌더링(Multithread Rendering)**
  - 멀티 코어 CPU
  - 단일 쓰레드(Single Thread)
    많은 렌더링 명령을 순차적으로 실행하면 CPU가 병목 현상을 발생시킴
    멀티 코어 CPU에서 다른 CPU들은 놀고 있으며 GPU도 놀고 있음(Execute할 때까지)
  - 다중 쓰레드(Multithreading)
    동기화(Synchronization) 필요
  - 다중 쓰레드를 사용하여 객체의 생성과 렌더링을 수행할 수 있음
    CPU와 GPU의 사용을 극대화
  - 렌더링 명령의 처리(GPU)와 렌더링 명령의 생성(CPU)은 리소스를 공유
  - 다중 쓰레딩
    객체의 생성
    렌더링

무엇을 다중 쓰레드로 처리할 것인가?
모델/텍스쳐 로딩(?), 쉐이더 컴파일/생성(?)



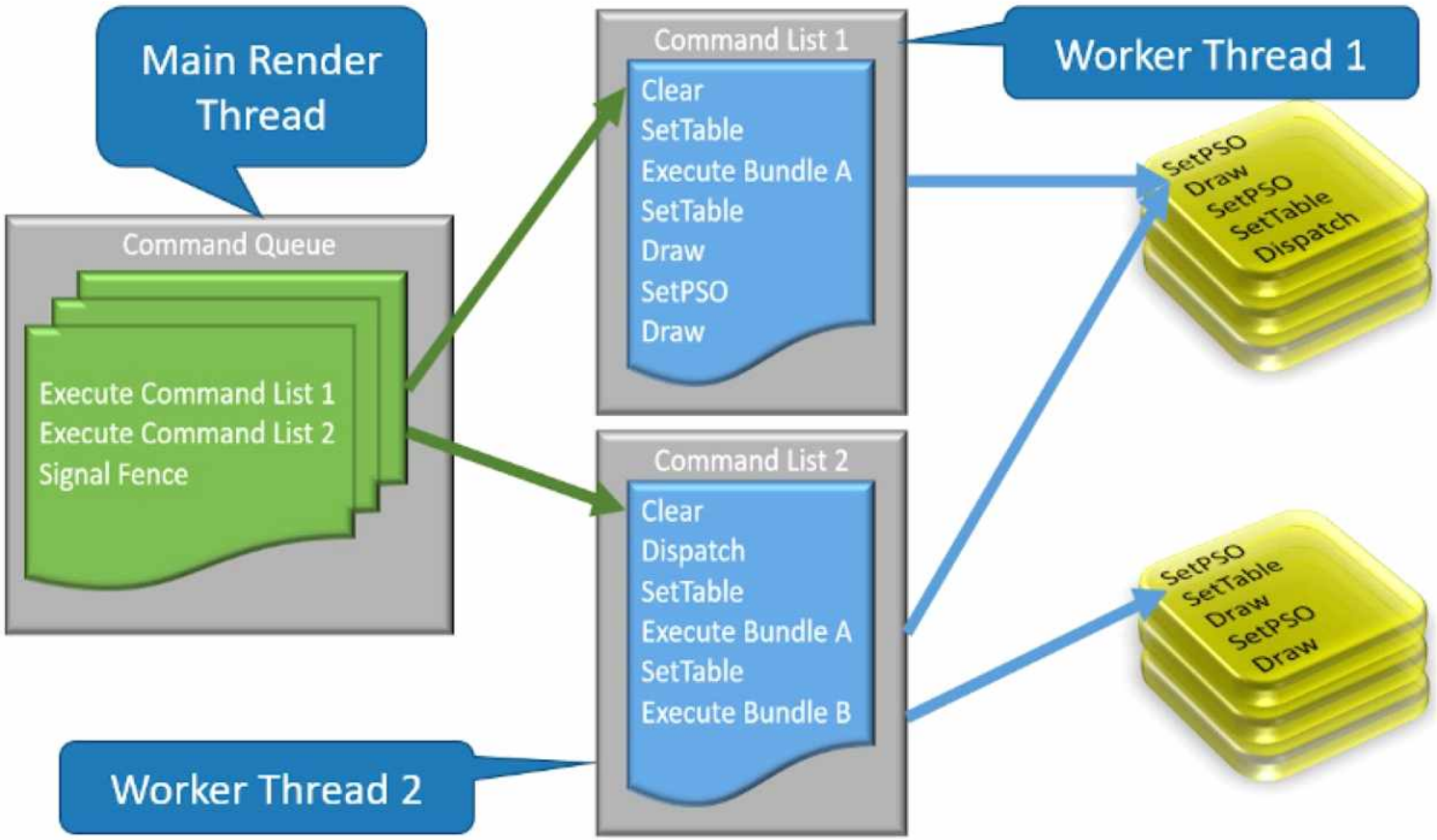
Direct3D Multithreading

- **멀티 쓰레딩(Multithreading)**
  - Direct3D 12는 효율적인 멀티 쓰레딩을 위하여 설계되었음
    하나의 커다란 씬을 하나의 CPU로 처리(그리기)하는 것은 시간이 많이 걸림
    커다란 씬을 여러 개로 분할하여 병렬적으로 처리하는 것이 효율적일 수 있음
    명령 리스트를 병렬적으로 생성하고 실행하는 것이 더 적은 시간이 걸릴 수 있음
  - 명령 리스트(ID3D12CommandList)는 쓰레드 안전하지 않음
    여러 개의 쓰레드가 하나의 명령 리스트를 공유하지 않아야 함
    각 쓰레드는 별도의 명령 리스트를 가져야 함
  - 명령 할당자(ID3D12CommandAllocator)는 쓰레드 안전하지 않음
    한 순간에 각 명령 할당자에서 하나의 명령 리스트만 Reset() 할 수 있음
    여러 개의 쓰레드가 하나의 명령 할당자를 공유하지 않아야 함
    각 쓰레드는 별도의 명령 할당자를 가져야 함
    필요한 명령 할당자의 개수 = 쓰레드의 개수 * 프레임 버퍼의 개수
    펜스의 개수 = 쓰레드의 개수
  - 명령 큐(ID3D12CommandQueue)는 쓰레드 안전함
    여러 개의 쓰레드가 병렬적으로 명령 큐에 명령 리스트를 추가할 수 있음
  - 방법
    패스-기반(Pass-Based): 쓰레드의 개수=패스의 개수
    청크-기반(Chunk-Based): 패스를 작은 크기로 나눔
  - Multithreading12 샘플

Thread Synchronization Overhead
Command List Submission Overhead

Direct3D Multithreading

- 멀티 쓰레딩(Multithreading)



*Reference to "Direct3D 12 API Preview" presented by Max McMullen, Microsoft

Direct3D Multithreading

- **멀티 쓰레딩(Multithreading)**
  - Direct3D 12 멀티 쓰레딩 시나리오

```
class CD3D12Multithreading
{
   ID3D12Resource* m_ppd3dRenderTargets[nFrames];
   CFrameResource* m_pFrameResources[nFrames];

   ID3D12Resource* m_pd3dDepthStencilBuffer;

   ID3D12CommandAllocator* m_pd3dCommandAllocator;
   ID3D12CommandQueue* m_pd3dCommandQueue;

   ID3D12PipelineState* m_pd3dPipelineStateScene;
   ID3D12PipelineState* m_pd3dPipelineStateShadowMap;

   ID3D12Fence* m_pd3dFence;
   UINT64 m_nFenceValue;
   HANDLE m_hFenceEvent;

   HANDLE m_phWorkerBeginRenderFrame[nThreads]; //Event Handles
   HANDLE m_phWorkerFinishShadowPass[nThreads]; //Event Handles
   HANDLE m_phWorkerFinishedRenderFrame[nThreads]; //Event Handles
   HANDLE m_phThreadHandles[nThreads]; //Thread Handles
};
```

nFrames: 후면버퍼의 개수
nThreads: 쓰레드의 개수

씬을 쓰레드의 개수만큼 분할

그림자 맵을 생성할 때 렌더 타겟과 픽셀 쉐이더는 필요하지 않음(그림자 맵을 DSV로 설정)

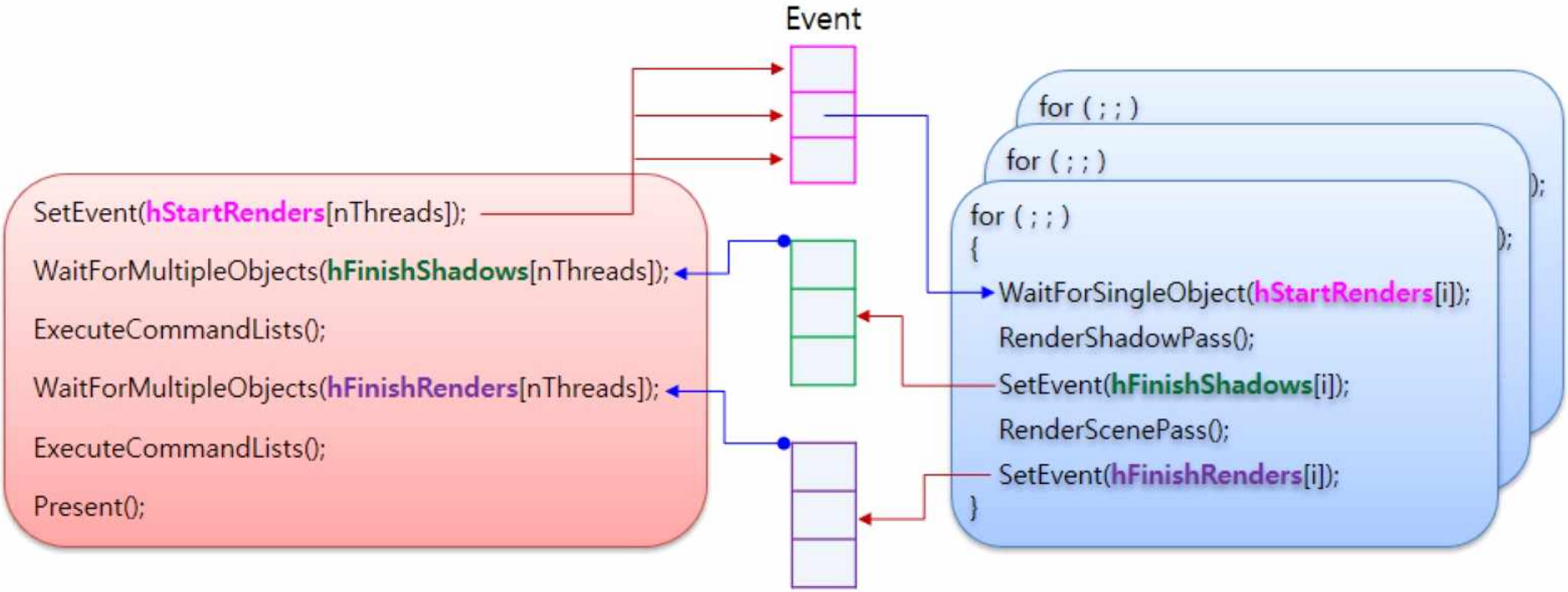
Direct3D Multithreading

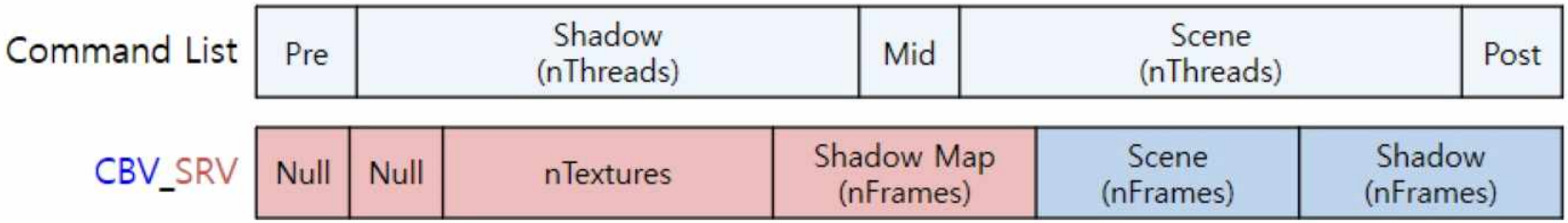
- **멀티 쓰레딩(Multithreading)**
  - Direct3D 12 멀티 쓰레딩 시나리오

```
class CFrameResource
{
   ID3D12CommandList* m_ppd3dBatchSubmits[nThreads * 2 + nCommandLists];

   ID3D12CommandAllocator* m_ppd3dCommandAllocators[nCommandLists];
   ID3D12GraphicsCommandList* m_ppd3dCommandLists[nCommandLists];

   ID3D12CommandAllocator* m_ppd3dShadowCommandAllocators[nThreads];
   ID3D12GraphicsCommandList* m_ppd3dShadowCommandLists[nThreads];

   ID3D12CommandAllocator* m_ppd3dSceneCommandAllocators[nThreads];
   ID3D12GraphicsCommandList* m_ppd3dSceneCommandLists[nThreads];

   UINT64 m_nFenceValue;

   ID3D12PipelineState* m_pd3dPipelineState;
   ID3D12PipelineState* m_pd3dPipelineStateShadowMap;
   ID3D12Resource* m_pd3dShadowTexture;
   D3D12_CPU_DESCRIPTOR_HANDLE m_hd3dShadowDepthView;
   ID3D12Resource* m_pd3dShadowConstantBuffer;
   ID3D12Resource* m_pd3dSceneConstantBuffer;

   D3D12_GPU_DESCRIPTOR_HANDLE m_hd3dNullSrvHandle;
   D3D12_GPU_DESCRIPTOR_HANDLE m_hd3dShadowDepthHandle;
   D3D12_GPU_DESCRIPTOR_HANDLE m_hd3dShadowCbvHandle;
   D3D12_GPU_DESCRIPTOR_HANDLE m_hd3dSceneCbvHandle;
};
```

Direct3D Multithreading

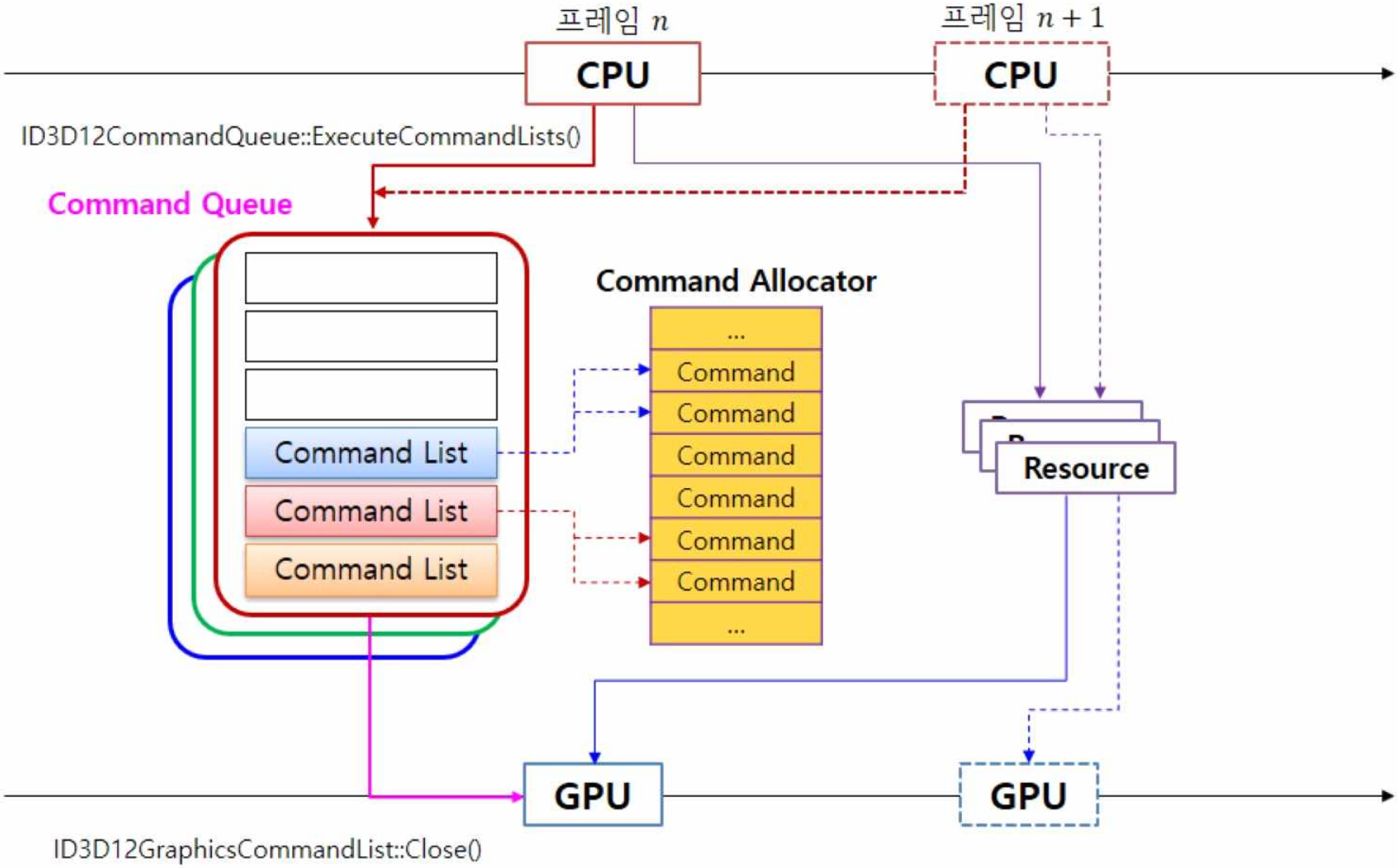
- **멀티 쓰레딩(Multithreading)**
  - Direct3D 12 멀티 쓰레딩 시나리오





Direct3D Multithreading

- **명령 큐(Command Queue), 명령 리스트(Command List)**
  - GPU는 명령 큐들을 가질 수 있으며 명령 큐의 GPU 명령들을 순서대로 실행함



Direct3D Multithreading

- **명령 큐(Command Queue) 인터페이스(ID3D12CommandQueue)**
  - GPU 명령을 실행하기 위하여 명령 큐를 생성해야 함

```
HRESULT ID3D12Device::CreateCommandQueue(
    D3D12_COMMAND_QUEUE_DESC *pDesc,
    REFIID riid, //__uuidof(ID3D12CommandQueue)
    void **ppCommandQueue
);
```

```
typedef struct D3D12_COMMAND_QUEUE_DESC {
    D3D12_COMMAND_LIST_TYPE Type;
    INT Priority; //명령 큐의 우선 순위
    D3D12_COMMAND_QUEUE_FLAGS Flags;
    UINT NodeMask; //단일 GPU: 0
} D3D12_COMMAND_QUEUE_DESC;
```

```
typedef enum D3D12_COMMAND_LIST_TYPE {
    D3D12_COMMAND_LIST_TYPE_DIRECT, //GPU가 직접 실행할 수 있는 명령 버퍼(리스트), 모든 엔진 사용
    D3D12_COMMAND_LIST_TYPE_BUNDLE, //GPU가 직접 실행할 수 없음(Direct 명령 리스트가 필요)
    D3D12_COMMAND_LIST_TYPE_COMPUTE, //계산(Compute Shader)을 위한 명령 버퍼, 계산/복사 엔진 사용
    D3D12_COMMAND_LIST_TYPE_COPY //복사(Copy)를 위한 명령 버퍼, 복사 엔진 사용
} D3D12_COMMAND_LIST_TYPE;
```

```
typedef enum D3D12_COMMAND_QUEUE_PRIORITY {
    D3D12_COMMAND_QUEUE_PRIORITY_NORMAL, //보통 우선 순위
    D3D12_COMMAND_QUEUE_PRIORITY_HIGH //높은 우선 순위
} D3D12_COMMAND_QUEUE_PRIORITY;
```

```
typedef enum D3D12_COMMAND_QUEUE_FLAGS {
    D3D12_COMMAND_QUEUE_FLAG_NONE, //기본 명령 큐
    D3D12_COMMAND_QUEUE_FLAG_DISABLE_GPU_TIMEOUT //명령 큐에 대하여 GPU 타임아웃을 비활성화
} D3D12_COMMAND_QUEUE_FLAGS;
```

```
D3D12_COMMAND_QUEUE_DESC ID3D12CommandQueue::GetDesc();
```

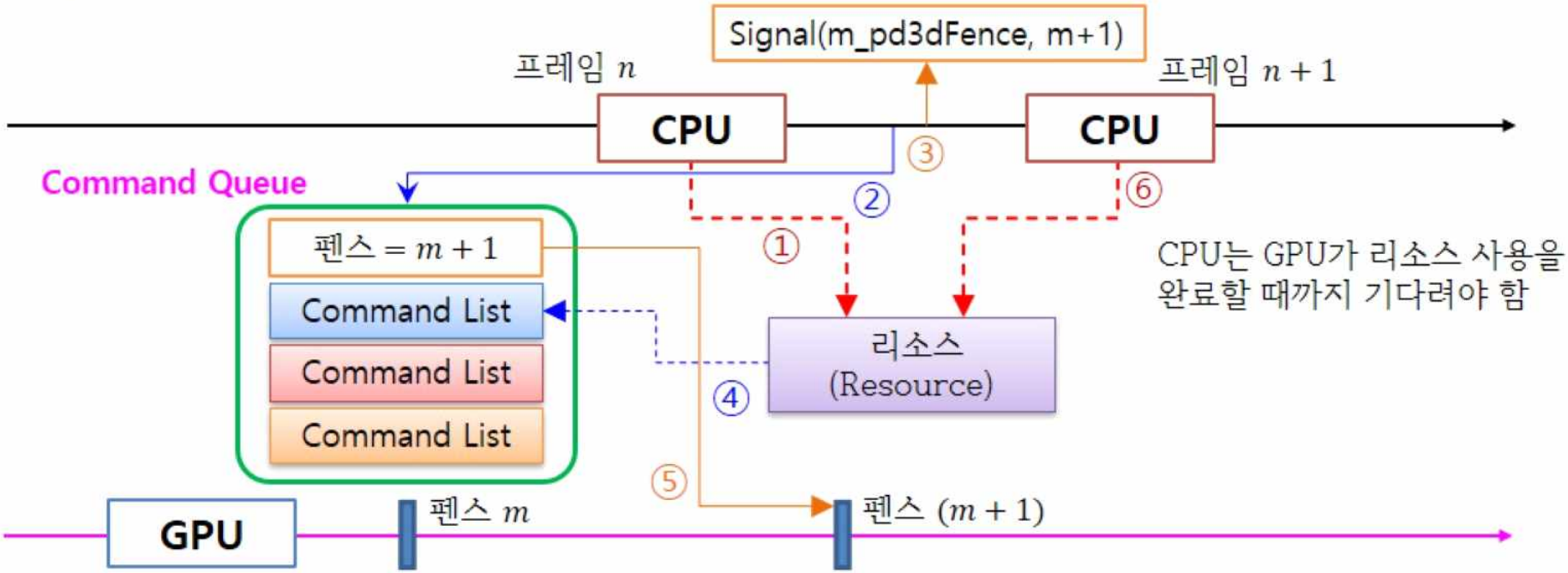
Direct3D Multithreading

- **ID3D12CommandQueue 인터페이스**

```
void ID3D12CommandQueue::ExecuteCommandLists(
    UINT NumCommandLists, //추가할 명령 리스트의 개수(여러 명령 리스트들을 한꺼번에 추가할 수 있음)
    ID3D12CommandList **ppCommandLists //추가할 명령 리스트들의 배열
);
```

```
HRESULT ID3D12CommandQueue::Signal(
    ID3D12Fence *pFence,
    UINT64 Value //GPU가 펜스를 이 값으로 설정하는 명령을 명령 큐에 추가(펜스의 값이 즉시 바뀌지 않음)
);
```

```
HRESULT ID3D12CommandQueue::Wait(
    ID3D12Fence *pFence,
    UINT64 Value //명령 큐(엔진)가 기다리는 펜스 값, 펜스의 현재 값이 이 값보다 작으면 기다림
);
```



Direct3D Multithreading

- **명령 리스트(Command List)**
  - GPU가 실행할 명령들의 순서화된 집합을 나타냄(명령들은 순서대로 실행)
    명령 리스트는 생성되면 열린(Open: 명령을 추가할 수 있는) 상태임
    명령 리스트를 닫으면(Close) 더 이상 명령을 추가할 수 없음
    ID3D12CommandList: 명령 리스트에 명령을 추가하기 위한 멤버 함수를 포함
    멤버 함수를 호출하는 것은 명령 리스트에 명령을 추가하는 것임
  - 명령 할당자의 유형과 명령 리스트의 유형은 일치해야 함

```
HRESULT ID3D12Device::CreateCommandList(
    UINT nodeMask,
    D3D12_COMMAND_LIST_TYPE type,
    ID3D12CommandAllocator *pCommandAllocator,
    ID3D12PipelineState *pInitialState,
    REFIID riid, //__uuidof(ID3D12CommandList)
    void **ppCommandList
);
```

```
typedef enum D3D12_COMMAND_LIST_TYPE {
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    D3D12_COMMAND_LIST_TYPE_BUNDLE,
    D3D12_COMMAND_LIST_TYPE_COMPUTE,
    D3D12_COMMAND_LIST_TYPE_COPY
} D3D12_COMMAND_LIST_TYPE;
```

```
UINT ID3D12Device::GetNodeCount();
```

nodeMask	명령어 리스트를 생성할 대상 어댑터(Node), 하나의 GPU일 때 0 여러 개의 GPU가 있으면 대상 GPU의 비트를 설정(1-비트만 설정)
type	명령어 리스트의 유형
pCommandAllocator	명령어 리스트를 생성할 명령어 할당자(ID3D12CommandAllocator)
pInitialState	명령어 리스트를 위한 초기 파이프라인 상태(ID3D12PipelineState) NULL: 라이브러리가 초기 파이프라인 상태를 설정
riid	__uuidof(ID3D12GraphicsCommandList)

```
ID3D12GraphicsCommandList::ClearState(); //명령 리스트를 생성할 때 설정한 파이프라인 상태로 설정
```

Direct3D Multithreading

- **펜스 객체(ID3D12Fence 인터페이스)**
  - CPU와 GPU의 동기화 그리고 GPU 엔진들을 동기화 하기 위하여 사용
    다중 엔진에서 모든 노드들은 어떠한 펜스라도 접근할 수 있음

```
UINT64 ID3D12Fence::GetCompletedValue(); //펜스의 현재 값(UINT64)을 반환
HRESULT ID3D12Fence::Signal(UINT64 Value); //지정한 값으로 펜스 값을 설정(CPU, 즉시 설정됨)
HRESULT ID3D12Fence::SetEventOnCompletion(UINT64 Value, HANDLE hEvent); //값에 도달할 때 이벤트 발생
```

- **펜스 객체의 생성**

```
HRESULT ID3D12Device::CreateFence(
    UINT64 InitialValue, //초기값
    D3D12_FENCE_FLAGS Flags, //bitwise OR
    REFIID riid, //__uuidof(ID3D12Fence)
    void **ppFence
);
```

```
typedef enum D3D12_FENCE_FLAGS {
    D3D12_FENCE_FLAG_NONE, //기본 펜스
    D3D12_FENCE_FLAG_SHARED, //공유 펜스
    D3D12_FENCE_FLAG_SHARED_CROSS_ADAPTER //GPU 공유
} D3D12_FENCE_FLAGS;
```

```
m_pd3dDevice->CreateFence(0, D3D12_FENCE_FLAG_NONE, __uuidof(ID3D12Fence), (void **)&m_pd3dFence);
m_nFenceValue = 0;
m_hFenceEvent = ::CreateEvent(NULL, FALSE, FALSE, NULL);
```

```
ID3D12Fence *m_pd3dFence = NULL;
HANDLE m_hFenceEvent;
UINT64 m_nFenceValue = 0;
```

```
m_nFenceValue++;
m_pd3dCommandQueue->Signal(m_pd3dFence, m_nFenceValue);
if (m_pd3dFence->GetCompletedValue() < m_nFenceValue)
{
    m_pd3dFence->SetEventOnCompletion(m_nFenceValue, m_hFenceEvent);
    ::WaitForSingleObject(m_hFenceEvent, INFINITE);
}
```

Direct3D Multithreading

- **인터락 변수 접근(Interlocked Variable Access)**
  - 여러 쓰레드가 공유하는 변수에 대한 접근을 동기화해야 함
    공유하는 변수에 대한 연산이 부분적으로 이루어지지 말아야 함(Atomically)
  - 64-비트 운영체제
    32(64)-비트 변수에 대한 읽기와 쓰기는 모든 비트에 대하여 완전하게 이루어짐
    읽기와 쓰기 연산에 대한 접근 순서는 동기화되도록 보장되지 않음
    두 개의 쓰레드가 같은 변수에 대하여 읽기와 쓰기를 하는 순서는 보장할 수 없음
  - 인터락 함수(Interlocked Function)
    여러 쓰레드가 공유하는 변수에 대한 접근을 동기화하는 아토믹 함수를 제공

```
LONG InterlockedIncrement(LONG *Addend);
SHORT InterlockedIncrement16(SHORT *Addend);
LONG64 InterlockedIncrement64(LONG64 *Addend);
LONG InterlockedIncrementAcquire(LONG *Addend);
LONG InterlockedIncrementRelease(LONG *Addend);
LONG InterlockedDecrement(LONG *Addend);
LONG InterlockedAdd(LONG *Addend, LONG Value);
LONG InterlockedExchange(LONG *Target, LONG Value); //이전 값을 반환
PVOID InterlockedExchangePointer(PVOID *Target, PVOID Value);
LONG InterlockedCompareExchange(LONG *Destination, LONG ExChange, LONG Comperand);
LONG InterlockedExchangeAdd(LONG *Addend, LONG Value);
unsigned InterlockedExchangeSubtract(unsigned *Addend, unsigned Value);
LONG InterlockedAnd(LONG *Destination, LONG Value);
LONG InterlockedOr(LONG *Destination, LONG Value);
LONG InterlockedXor(LONG *Destination, LONG Value);
```