게임서버프로그래밍

2019년 2학기

한국산업기술대학교
게임공학부

정내훈

3장 소켓 프로그래밍

3. 1 | 블로킹 소켓

- 블로킹

  디바이스에 처리 요청을 걸어 놓고 응답을 대기하는 함수를 호출할 때 스레드에서 발생하는 대기현상.
  소켓뿐만 아니라 파일 핸들에 대한 함수를 호출했을 때도 이러한 대기 현상이 발생하는 것을 모두 블로킹이라고 한다.

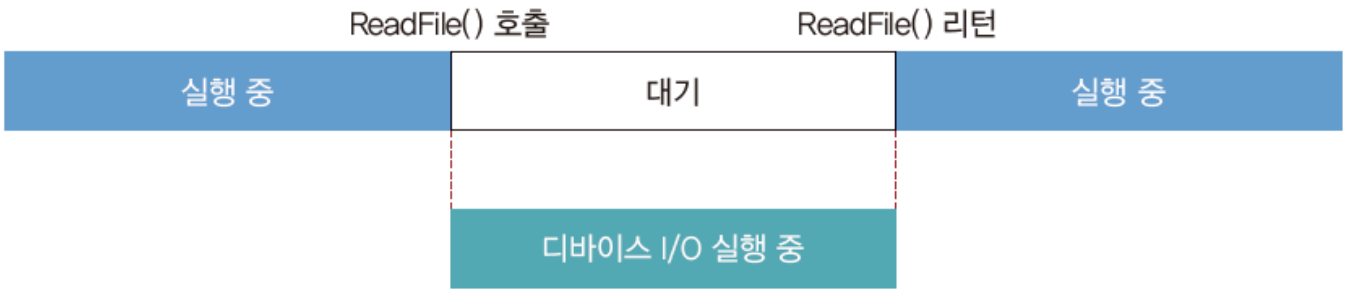


그림 3-1 디바이스에 일을 시키는 함수를 호출한 후 스레드가 한동안 대기 상태가 되는 상황

3. 2 | 네트워크 연결 및 송신

- TCP 소켓을 이용하여 통신하는 프로그램을 살펴보기

TCP는 연결 지향형 프로토콜로 일대일 통신만 허락한다.

따라서 TCP 소켓 1개는 오직 끝점 1개하고만 통신할 수 있다.

```
// 이 프로그램은 주소가 11.22.33.44인 기기에서 실행한다.
main()
{
        s = socket(TCP);                        //1
        s.bind(any_port);                       //2
        s.connect("55.66.77.88:5959");          //3
        s.send("hello");                        //4
        s.clos      e();                        //5
}
```

1 socket(TCP)는 TCP 소켓 핸들 s를 생성한다. 아직 s는 아무것도 할 수 없다.

2 s.bind(any_port)는 자기 컴퓨터(localhost) 안에 있는 포트 6만 5535개 중에서 사용 가능한 포트(빈 포트)를 차지한다.

3 connect(xxx)는 상대방 끝점을 향해 TCP 연결을 시도합니다. 그리고 이 함수는 블로킹이 일어난다.

4 send(xxx)는 상대방 끝점을 향해 데이터를 전송한다.

5 close()는 TCP 소켓을 닫는다. TCP 소켓을 닫으면서 TCP 연결도 해제된다.

3. 3 | 블로킹과 소켓 버퍼

- 송신 버퍼(send buffer)와 수신 버퍼(receive buffer)

  송신 버퍼 : 일련의 바이트 배열(byte array)이라고 보면 됨. 송신 버퍼의 크기는 고정 되어 있으나, 마음대로 크기를 변경할 수 있다.

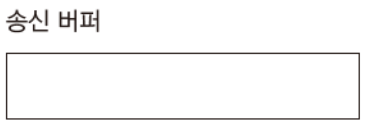


그림 3-2
크기가 5바이트인 송신 버퍼

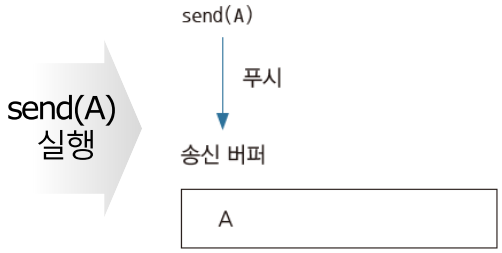


그림 3-3
send(A) 실행 후 송신 버퍼의 모습

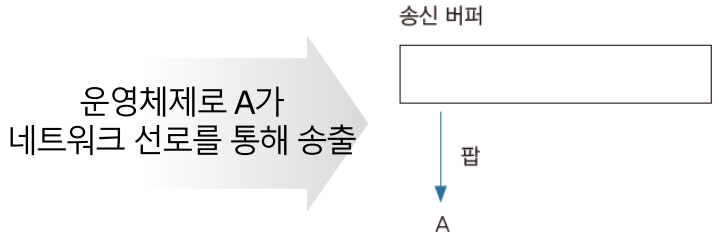
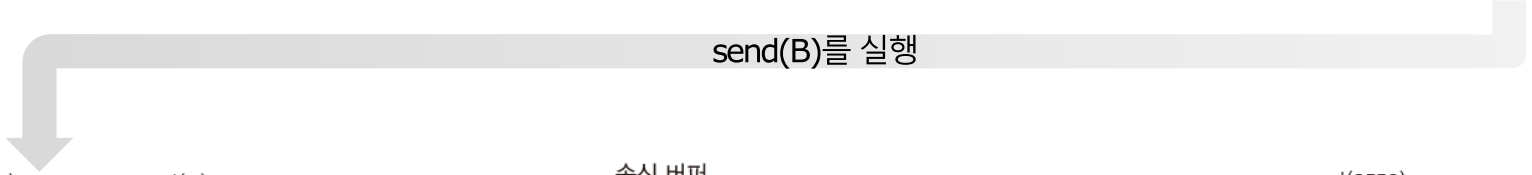


그림 3-4
A가 빠져나간 후 송신 버퍼



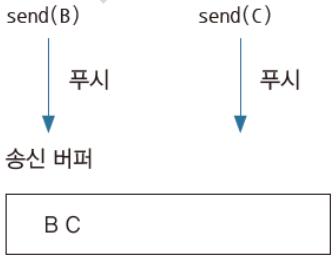


그림 3-5
B와 C로 채워진 송신 버퍼

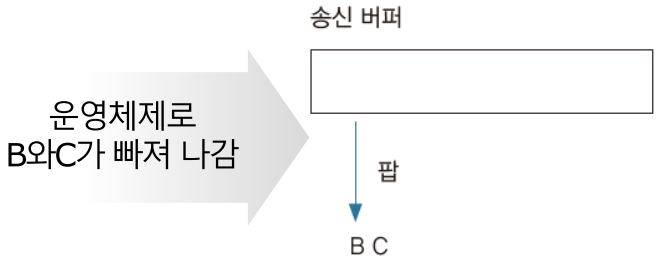


그림 3-6
B와 C가 빠져나간 송신 버퍼

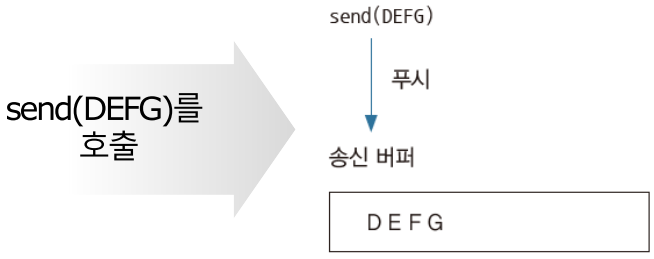


그림 3-7
DEFG가 채워진 송신 버퍼

3. 3 | 블로킹과 소켓 버퍼

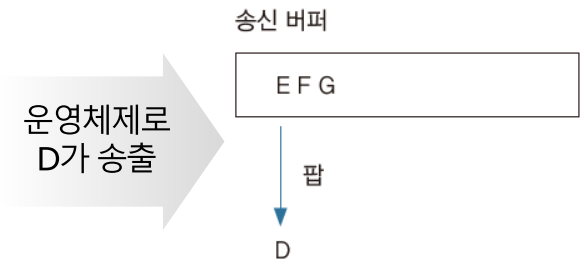


그림 3-8
D가 사라지고 EFG만 남음

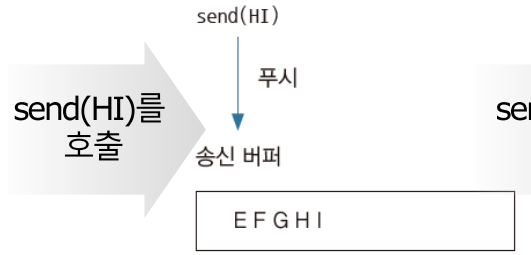


그림 3-9
HI가 추가

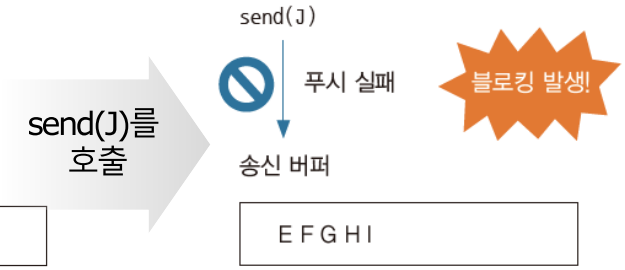


그림 3-10
송신 버퍼가 가득 차서
send( ) 함수의 블로킹 발생

운영체제는 송신버퍼에 뭔가 있는 것을 발견

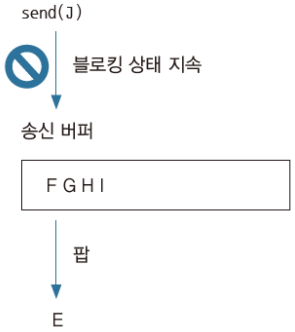


그림 3-11
E를 뺌

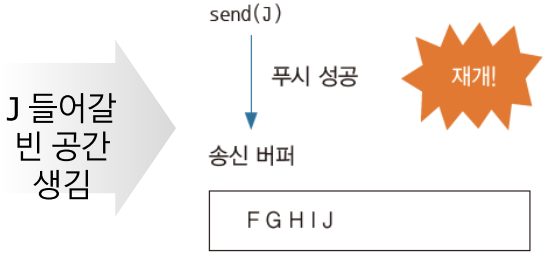


그림 3-12
send( ) 함수의 블로킹 해제

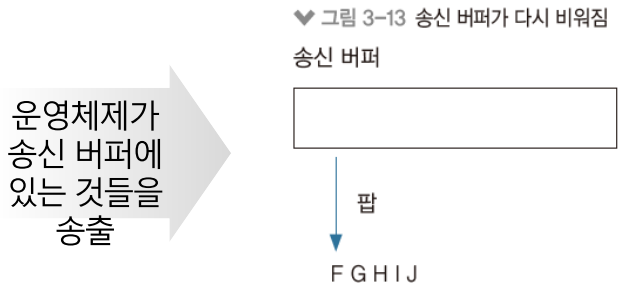


그림 3-13
송신 버퍼가 다시 비워짐

3. 3 | 블로킹과 소켓 버퍼

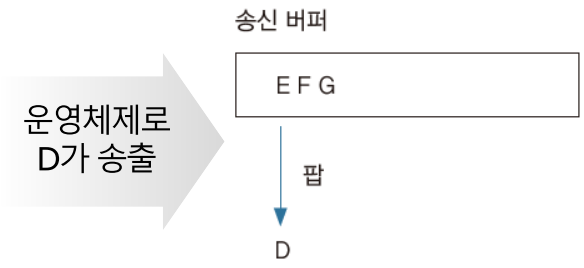


그림 3-8
D가 사라지고 EFG만 남음

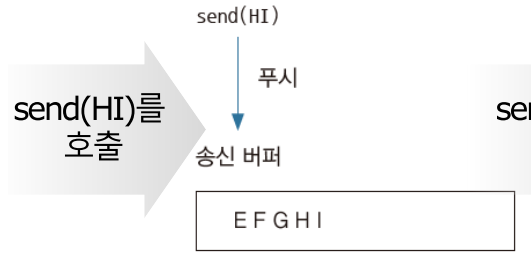


그림 3-9
HI가 추가

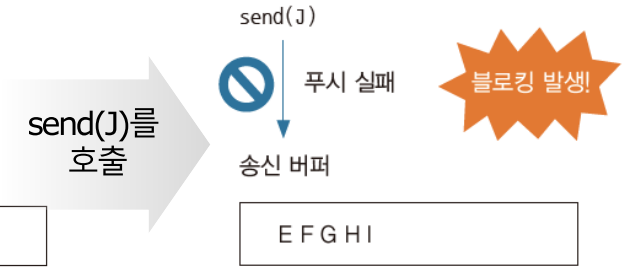


그림 3-10
송신 버퍼가 가득 차서
send( ) 함수의 블로킹 발생

운영체제는 송신버퍼에 뭔가 있는 것을 발견

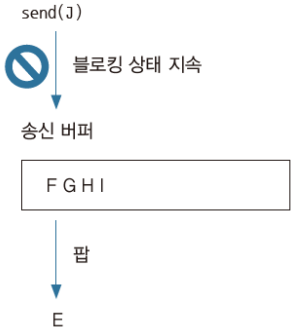


그림 3-11
E를 뺌

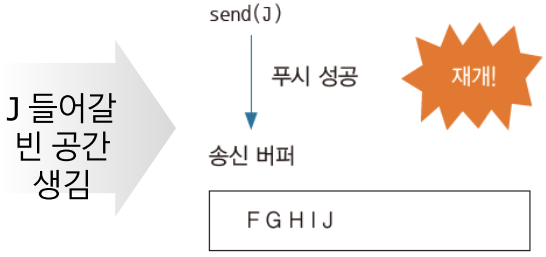


그림 3-12
send( ) 함수의 블로킹 해제

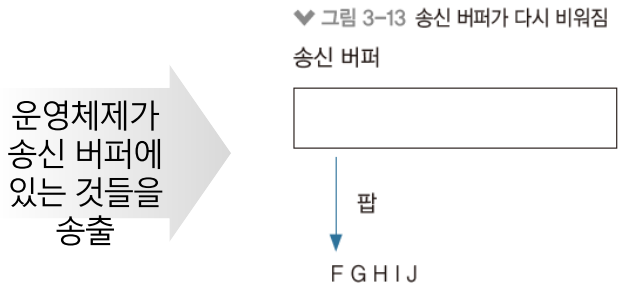


그림 3-13
송신 버퍼가 다시 비워짐

3. 3 | 블로킹과 소켓 버퍼


```
// 이 프로그램은 주소가 11.22.33.44인 기기에서 실행한다.
main()
{
        s = socket(TCP);
        s.bind(any_port);
        s.connect("55.66.77.88:5959");
        s.send("hello"); //1
        s.close();
}
```

1 send()는 즉시 리턴한다. 송신 버퍼는 디폴트로 수천 바이트를 담을 수 있기 때문.

3. 4 | 네트워크 연결받기 및 수신

- 네트워크 연결을 수락하고 데이터를 받는 방법

```
// 이 프로그램은 주소가 55.66.77.88인 기기에서 실행한다.
main()
{
        s = socket(TCP);                        //①
        s.bind(5959);                   //②
        s.listen();                             //③
        s2 = s.accept();                        //④
        print(getpeeraddr(s2));         //⑤
        while (true)
        {
                r = s2.recv();          //⑥
                if (r.length <= 0)              //⑦
                        break;
                print®;
        }
        s2.close();                             //⑧
}
```

① TCP 소켓을 생성한다.

② TCP 포트 5959번을 점유한다.

③ 이 소켓은 TCP 연결을 받는 역할을 시작하여 리스닝 소켓이 되었다.

④ TCP 연결이 들어올 때까지 기다린다.

⑤ accept() 함수에서 받은 새로운 소켓 핸들을 이용해서 저쪽 끝점과 통신한다.

⑥ 새로운 소켓에서 데이터 수신을 한다.

⑦ TCP 소켓에 대해 수신 함수를 호출했을 때 받은 데이터 크기가 0바이트라면 상대방이 tcp 연결을 끝냈음을 의미한다.

⑧ 새로운 소켓의 연결이 끊어짐. 이제 더 이상 사용할 수 없으므로 소켓을 닫는다.

3. 5 | 수신 버퍼가 가득 차면 발생하는 현상

- 수신 함수가 수신 버퍼에서 데이터를 꺼내는 속도가 운영 체제가 수신 버퍼의 데이터를 채우는 속도보다 느리면

TCP 송신 함수로 송신 버퍼에 데이터를 쌓는 속도보다 수신 함수로 수신 버퍼에서 데이터를 꺼내는 속도가 느리다고 해서 TCP 연결은 끊어지지 않는다. (실제 송신 속도가 느린 쪽에 맞추어 작동할 뿐임.)

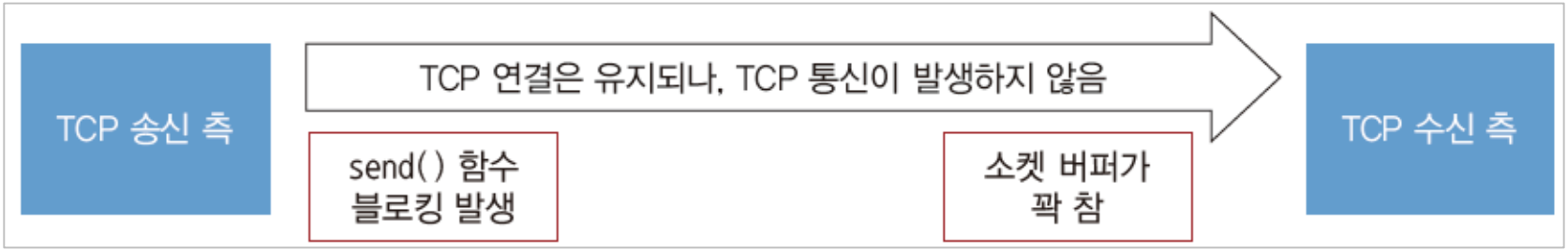


그림 3-14 송신 버퍼에 데이터를 채우는 속도보다 수신 함수가 데이터를 꺼내는 속도가 느릴 때(TCP)

UDP 송신 함수로 송신 버퍼에 데이터를 쌓는 속도보다 수신 함수로 수신 버퍼에서 데이터를 꺼내는 속도가 느리면, 데이터그램 유실이 발생한다.

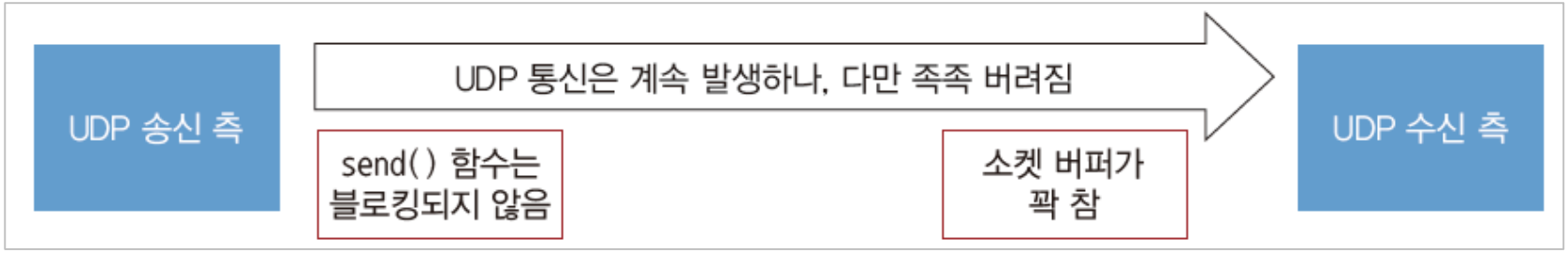


그림 3-16 송신 버퍼에 데이터를 채우는 속도보다 수신 함수가 데이터를 꺼내는 속도가 느릴 때(UDP)

3. 6 | 논블록 소켓

- 네트워킹을 해야 하는 대상이 여럿이라면

```
void BlockSocketOperation()
{
	s = socket(TCP);
	...;
	s.connect(...);
	...;
	while (true)
	{
		s.send(data);
	}
}
```

- 논블록 소켓을 사용하는 방법

1. 소켓을 논블록 소켓으로 모드 전환한다..
2. 논블록 소켓에 대해 평소처럼 송신, 수신, 연결과 관련된 함수를 호출한다.
3. 논블록 소켓은 무조건 이 함수 호출에 대해 즉시 리턴한다.리턴 값은 '성공' 혹은 'wouldblock' 오류 둘 중에 하나이다.

3. 6 | 논블록 소켓

- 논블록 소켓을 사용하는 코드로 바꾸면

```
void NonBlockSocketOperation()
{
	s = socket(TCP);
	...;
	s.connect(...);
	// 논블록 소켓으로 변경
	s.SetNonBlocking(true);
while (true)
{
	//❶
	r = s.send(dest, data);
	if (r = = EWOULDBLOCK)
	{
		// 블로킹 걸릴 상황이었다. 송신을 안 했다.
		continue;
	}

	if (r = = OK)
	{
		// 보내기 성공에 대한 처리
	}
	else
	{
		// 보내기 실패에 대한 처리
	}
	//❷
    }
}
```

❶ 논블록 소켓으로 모드를 바꾼 후 송신 함수 호출은 언제나 즉시 리턴한다.

이 코드는 ❷ 에서 CPU 사용량이 늘어난다는 문제가 있다.

3. 6 | 논블록 소켓

- 논블록 소켓으로 처리해서 블로킹 문제 해결하기

```
List<Socket> sockets;

void NonBlockSocketOperation()
{
	foreach(s in sockets) // 각 소켓에 대해
	{
		// 논블록 수신. 오류 코드와 수신된 데이터를 받는다.
		(result, data) = s.receive();
		if (data.length > 0) // 잘 수신했으면
		{
			print(data); // 출력
		}
		else if (result != EWOULDBLOCK)
		{
			// would block이 아니면 오류가 난 것이므로
			// 필요한 처리를 한다.
			...;
		}
	}
}
```

3. 6 | 논블록 소켓

- 0 바이트 송신

0바이트 송신하는 함수가 '성공'을 리턴하면 TCP 연결되어 있다는 의미.
ENOTCONN(소켓이 연결되어 있지 않음)을 리턴하면 TCP 연결이 진행 중.
기타 오류 코드가 나오면 TCP 연결 시도가 실패한 것.

```
void NonBlockSocketOperation()
{

	result = s.connect();
	if (result = = EWOULDBLOCK)
	{
		while (true)
		{
			byte emptyData[0]; // 길이 0인 배열

			result = s.send(emptyData);
			if (result = = OK)
			{
				// 연결 성공 처리
			}
			else if (result = = ENOTCONN)
			{
				// 연결이 아직 진행 중이다.
			}
			else
			{
				// 연결 실패 처리
			}
		}
	}
}
```

이 예시 코드를 실제로 구동하면
해당 스레드는 소켓이 would block인 상태에서는 계속해서 루프를 돌고,
이 루프는 CPU를 쉬지 못하는 바쁜 상태로 만든다.

3. 6 | 논블록 소켓

- CPU 사용량 폭주 문제

```
List<Socket> sockets;

void NonBlockSocketOperation()
{
        while (true)
        {
                foreach(s in sockets)
                {
                        // 논블록 수신 ❶
                        (result, data) = s.receive();
                        if (data.length > 0)
                        {
                                print(data);
                        }
                        else if (result != EWOULDBLOCK)
                        {
                                // 소켓 오류 처리를 한다.
                        }
                }
                // ❷
        }
}
```

필요한 함수

- 여러 소켓 중 하나라도 would block이었던 상태에 변화가 일어나면, 즉 송신 버퍼에 빈 공간이 생기거나
  수신 버퍼에 뭔가가 들어온다면 그 상황을 알려 주는 함수
- 혹은 그것을 알려 주기 전까지는 블로킹 중이어서 CPU 사용량 폭주를 해결하는 함수

제공하는 기능

1.소켓 리스트 A를 입력.

2. A에 있는 소켓 중 하나라도 I/O 처리를 할 수 있는 것이 생기는 순간까지 블로킹함.

3. 블로킹이 끝나면 어떤 소켓이 I/O 처리를 할 수 있는지 알려 줌.

4. 블로킹은 타임아웃을 지정할 수 있다. '무한대'를 입력하면 I/O 처리를 할 수 있는 소켓이 생길 때까지 영원히 기다린다. 지정한 시간(밀리초 단위)을 입력하면 해당 시간이 될 때까지 기다린다. 0초를 입력하면 블로킹 없이 결과를 리턴한다.

함수

select() 혹은 poll()

3. 6 | 논블록 소켓

- select() 함수를 적용하기

```
List<Socket> sockets;

void NonBlockSocketOperation()
{
        while (true)
        {
                // 100밀리초까지 대기 ❶❶
                // 1개라도 I/O 처리를 할 수 있는 상태가 되면
                // 그 전에라도 리턴
                List<Socket>ready_sockets =
                                select(sockets);

                foreach(s in ready_sockets)
                {
                        // 논블록 수신 ❷
                        (result, data) = s.receive();
                        if (data.length > 0)
                        {
                                print(data);
                        }
                        else if (result != EWOULDBLOCK)
                        {
                                // 소켓 오류 처리를 한다.
                        }
                }
        }
}
```

❶에서 select()를 호출한다. sockets에 I/O 처리가 가능한 소켓들을 즉시 리턴한다.

select() 함수가 리턴한 후에는
sockets의 각 소켓에 대한 논블록 I/O 처리 함수를 호출한다.
성공하는 것도 실패하는 것도 있겠지만,
최소한 하나는 would block이 아닌 다른 결과가 나온다.

3. 6 | 논블록 소켓

- 논블록 처리(accept)

```
void NonBlockSocketOperation()
{
	s = socket(TCP);
	// 논블록 소켓으로 변경
	s.SetNonBlocking(true);
	s.listen(5000);

	while (true)
	{
		(socket, result) = s.accept();

		if (result = = EWOULDBLOCK)
		{
			// 블로킹 걸릴 상황이었다. TCP 연결이 안 들어왔다.
			continue;
		}
		if (result = = OK)
		{
			// TCP 연결을 잘 받았다.
		}
```

3. 6 | 논블록 소켓


```
            else
            {
                // 리스닝 소켓에 무슨 문제가 생겼다.
            }
        }
    }
```

3. 7 | Overlapped I/O 혹은 비동기 I/O

- 논블록 소켓의 장점

  스레드 블로킹이 없으므로 중도 취소 같은 통제가 가능하다.

  스레드 개수가 1개이거나 적어도 소켓을 여러 개 다룰 수 있다.

  스레드 개수가 적거나 1개이므로 연산량이 낭비되지 않는다. 그리고 호출 스택 메모리도낭비되지 않는다.

- 논블록 소켓의 단점

  소켓 I/O 함수가 리턴한 코드가 would block인 경우 재시도 호출 낭비가 발생한다.

  소켓 I/O 함수를 호출할 때 입력하는 데이터 블록에 대한 복사 연산이 발생한다.

  connect() 함수는 재시도 호출을 하지 않지만, send() 함수나 receive() 함수는 재시도 호출해야 하는 API가 일관되지 않는다는 문제가 있다.

3. 7 | Overlapped I/O 혹은 비동기 I/O

- 논블록 소켓의 단점-재시도용 호출의 낭비

```
List<Socket> sockets;

void NonBlockSocketOperation()
{
	while (true)
	{
		// 100밀리초까지 대기
		// 1개라도 I/O 처리를 할 수 있는 상태가 되면
		// 그 전에라도 리턴
		select(sockets, 100ms);

		foreach(s in sockets)
		{
			// 논블록 수신
			(result, length) = s.sendto(dest, data);
			if (length >= 0)
			{
				// 잘 보냈다.
			}
			else if (result != EWOULDBLOCK)
			{
```

3. 7 | Overlapped I/O 혹은 비동기 I/O


```
                    // 소켓 오류 처리를 한다.
                }
                else
                {
                    // 아직 would block이다.
                }
            }
        }
}
```

UDP 소켓의 송신 버퍼에 1바이트라도 비어 있으면 I/O 가능.

보내려는 데이터가 5바이트인데 송신 버퍼 빈 공간이 1바이트라면, 넣을 수 있는 크기를 넘어선다.

TCP와 달리 UDP는 일부만 보낼 수 없으므로 would block이 발생. ('헛발질'이 발생한 셈)

I/O 가능이라 재시도는 했는데, 여전히 would block.

이 상태로라면 UDP send()를 하지 못한 채 계속 헛발질만 반복하고, 이는 결국 CPU 낭비로 이어짐.

3. 7 | Overlapped I/O 혹은 비동기 I/O

- 논블록 소켓의 단점-소켓 함수 내부의 데이터 복사 부하

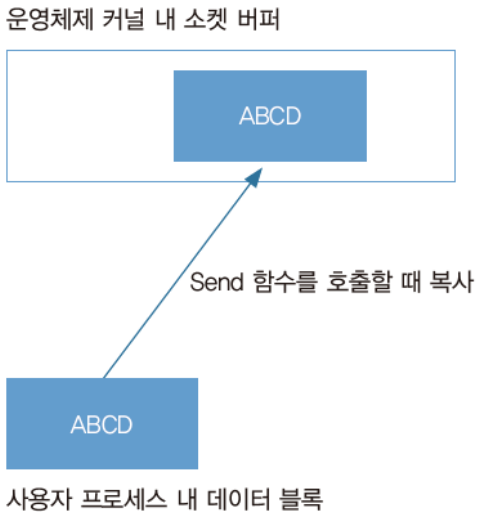


그림 3-16
메모리 복사 연산 발생

캐시에 없는 데이터를 액세스할 때는 메인 메모리 RAM을 액세스하는데, 이 속도는 매우 느림. 따라서 고성능 서버를 개발할 때는 이 복사 연산도 무시할 수 없는 변수인 셈이다.

3. 7 | Overlapped I/O 혹은 비동기 I/O

- Overlapped I/O를 다루는 방법

논블록 소켓이 다루는 데이터	Overlapped I/O 에서는...
1. 소켓이 I/O 가능인 것이 있을 때까지 기다린다. 2. 소켓에 대해 논블록 액세스를 한다. 3. would block이 발생했으면 그대로 두고, 그렇지 않으면 실행 결과 리턴 값을 처리한다	1. 소켓에 대해 Overlapped 액세스를 "건다." 2. Overlapped 액세스가 성공했는지 확인한 후 성공했으면 결괏값을 얻어 와서 나머지를 처리한다.

3. 7 | Overlapped I/O 혹은 비동기 I/O

- Overlapped I/O를 다루는 방법

```
void OverlappedSocketOperation()
{
	// ❶❶
	var overlappedSendStatus;
	// ❷
	(result, length) = s.OverlappedSend(data, overlappedSendStatus);

	if (length > 0)
	{
		// 보내기 성공❸
	}
	else if (result = = WSA_IO_PENDING)
	{
		// Overlapped I/O가 진행 중❹
		while (true)
	{
		(result, length) =
			GetOverlappedResult(s,  overlappedSendStatus); // ❺
```

3. 7 | Overlapped I/O 혹은 비동기 I/O

- Overlapped I/O를 다루는 방법

```
            if (length > 0)
            {
                // 잘 보냈다. ❸
            }
            else
            {
                // 아직 I/O pending이다.
            }
        }
    }
}
```

1. Overlapped I/O를 걸 때 '진행 중인 상태 현황'을 보관하는 구조체를 먼저 준비한다. (❶).
2. 블로킹 소켓을 그대로 사용한다.
3. 소켓에 대한 Overlapped I/O 전용 함수를 호출한다(❷). 전용 함수는 항상 즉시 리턴한다.
4. 즉시 성공했다면 OK가 리턴되고(❸), 그렇지 않으면 완료를 기다리는 중, 즉 I/O pending이 라는 값이 즉시 반환됨(❹).
5. Overlapped I/O 완료 여부는 ❺처럼 확인하는 함수를 호출해 보면 알 수 있다.
6. ❺에서 '완료'라고 결과가 나오면 그냥 나머지 처리를 한다.
7. Overlapped I/O를 수신했다면 data 객체에 수신된 데이터가 자동으로 채워져 있을 것이다. 이를 그냥 액세스한다.

3. 7 | Overlapped I/O 혹은 비동기 I/O

- Overlapped I/O_백그라운드에서 액세스로 인한 주의사항

Overlapped I/O 함수는 즉시 리턴되지만, 운영체제로 해당 I/O 실행이 별도로 동시간대에 진행되는 상태임.

운영체제는 소켓 함수에 인자로 들어갔던 데이터 블록을 백그라운드에서 액세스한다.

호출한 Overlapped I/O 전용 함수가 비동기로 하는 일이 완료(complete)될 때까 지는 소켓 API에 인자로 넘긴 데이터 블록을 제거하거나 내용을 변경해서는 안됨.

Overlapped status 구조체 또한 운영체제에서 백그라운드로 액세스 중이므로 중간에 없애거나 내용을 변경해서도 안된다.

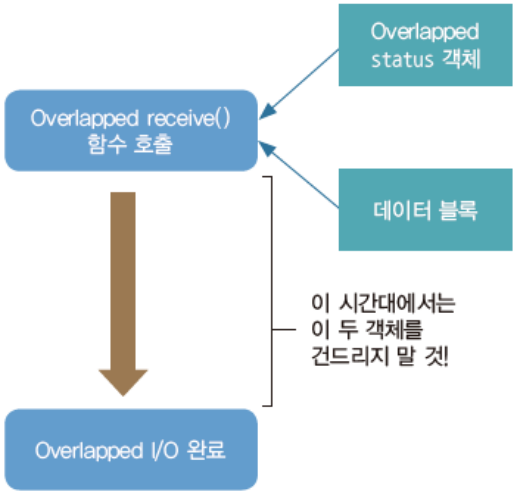


그림 3-17
Overlapped status 구조체와 데이터 블록을 건들지 말아야 하는 구간

일반 버전 함수	Overlapped I/O 전용 함수
send	WSASend
sendto	WSASendTo
recv	WSARecv
recvfrom	WSARecvFrom
connect	ConnectEx
accept	AcceptEx

표 3-1 윈도에서 제공하는 Overlapped I/O 전용 함수

3. 7 | Overlapped I/O 혹은 비동기 I/O

- Overlapped I/O의 장단점

장점	단점
소켓 I/O 함수 호출 후 would block 값인 경우 재시도 호출 낭비가 없다.	완료되기 전까지 Overlapped status 객체가 데이터 블록을 중간에 훼손하지 말아야 한다
소켓 I/O 함수를 호출할 때 입력하는 데이터 블록에 대한 복사 연산을 없앨 수 있다.	윈도 플랫폼에서만 제공하는 기능이다.
send, receive, connect, accept 함수를 한 번 호출하면 이에 대한 완료 신호(3.9절 참고)는 딱 한 번만 오기 때문에 프로그래밍 결과물이 깔끔하다	accept, connect 함수 계열의 초기화가 복잡하다.
뒤에서 설명할 I/O completion port와 조합하면 최고 성능의 서버를 개발할 수 있다.	-

- Overlapped I/O에서 I/O 실행(operation) 상태

Overlapped 송신이 진행 중이고 완료가 아직 안 되었으면 'Overlapped 송신이 아직 완료 대기 중(pending)'이라고 함.

Overlapped 수신이 진행 중이고 완료가 아직 안 되었으면 'Overlapped 수신이 완료 대기 중'이라고 함.

이를 통칭 'I/O 완료 대기 중' 혹은 I/O 실행 중'이라고 함.

3. 7 | Overlapped I/O 혹은 비동기 I/O

- 리액터 패턴과 프로액터 패턴

  논블록 소켓을 '뒤늦게...'라는 의미의 리액터 패턴(reactor pattern)이라고 한다.

  Overlapped I/O는 '미리...'라는 의미의 프로액터 패턴(proactor pattern)이라고 한다.

리액터	프로액터
논블록 소켓	Overlapped I/O
1. I/O를 시도한다(성공할 수도 있고 실패할 수도 있다). 2. 실패할 때는 I/O 가능을 기다린 후 I/O를 재시도한다. 3. 성공할 때는 상황을 종료한다	1. I/O를 시행(무조건 성공)한다. 2. I/O 완료를 기다린다. 3. 상황을 종료한다.
리눅스, FreeBSD, iOS, 안드로이드 등에서 주로 사용되었다. 윈도에서도 사용할 수 있으나 대세는 아니다.	윈도에서는 기본으로 지원한다. 리눅스, FreeBSD, iOS, 안드로이드에서는 Boost/ASIO를 통해 사용한다.

- 소켓 개수가 1만 개인 상황에서...

리액터	프로액터
1. 소켓 1만 개에 대해서 select를 호출. 2. 각 소켓에서 루프를 돌며 I/O를 시도한다 (이 루프를 도는 것은 성능 문제로 이어진다).	1. 개수가 1만 개 이하인 각 Overlapped status 객체에 대해 GetOverlappedStatus를 실행하여I/O 완료 상태인지 확인. 2. I/O 완료 상태면 나머지 처리(받은 데이터에 대한 로직 혹은 다음 보낼 데이터에 대한송신 함수 재호출)를 수행.

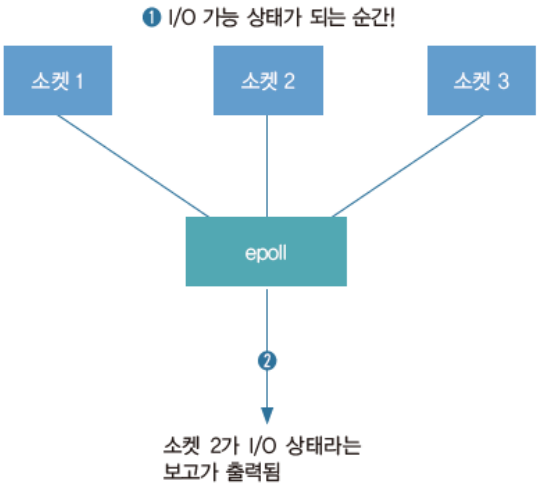
3. 8 | 공지

10월 3일 목요일 정상 수업

3. 8 | epoll

- epoll

  소켓이 I/O 가능 상태가 되면 이를 감지해서 사용자에게 알림을 해 주는 역할을 함, 이때 어떤 소켓이 I/O 가능 상태인지 알려 준다.



소켓 2가 I/O 가능이 되는 순간 epoll은 이 상황을 epoll 안에 내장된 큐에 푸시.

epoll에서 이러한 이벤트(사건) 정보를 팝(pop)할 수 있으며 이렇게 해서 어떤 소켓이 I/O 가능인지 알 수 있다.

따라서 소켓이 1만 개라고 하더라도 이 중에서 I/O 가능이된 것들만 epoll을 이용해서 바로 얻을 수 있다.

그림 3-18 소켓 2에서 I/O 가능 이벤트가 발생

3. 8 | epoll

- epoll의 사용법(코드)

```
epoll = new epoll(); // ❶
foreach(s in sockets)
{
	epoll.add(s, GetUserPtr(s)); // ❷
}

events = epoll.wait(100ms); //❸

foreach(event in events) //❹
{
	s = event.socket; //❺
	// 위 epoll.add에 들어갔던 값을 얻는다.
	userPtr = event.userPtr;
	// 수신? 송신?
	type = event.type;
	if (type == ReceiveEvent)
	{
		(result, data) = s.recv();
		if (data.length > 0)
		{
```

3. 8 | epoll

- epoll의 사용법(코드)

```
                // 수신된 데이터를 처리한다.
                Process(userPtr, s, data);
            }
        }
    }
```

1. epoll 객체를 만든다.
2. 여러 소켓을 epoll에 추가합니다. 추가된 소켓의 I/O 가능 이벤트는 epoll로 감지할 수 있다.
3. 모든 소켓에 대한 select() 대신 epoll에서 이벤트를 꺼내 오는 함수를 호출한다. 이 함수는 사용자가 원하는 시간까지만 블로킹되며, 그 전에 이벤트가 생기는 순간 즉시 리턴한다.
4. 각 이벤트에 대해 루프를 돌며 이벤트가 가리키는 소켓 객체와 사용자 정의 값을 꺼내 온다.
5. 원하는 처리를 한다.

3. 8 | epoll

- 에지 트리거를 쓸 때의 주의 사항

1. I/O 호출을 한 번만 하지 말고 would block이 발생할 때까지 반복해야 한다.
2. 소켓은 논블록으로 미리 설정되어 있어야 한다.

```
...
foreach(event in events) //❹
{
        s = event.socket; //❺
        // 위 epoll.add에 들어갔던 값을 얻는다.
        userPtr = event.userPtr;
        // 수신? 송신?
        type = event.type;
        if (type == ReceiveEvent)
        {
                while (true)
                {
                        (result, data) = s.recv();
                        if (data.length > 0)
                        {
                                // 수신된 데이터를 처리한다.
                                Process(userPtr, s, data);
                        }
                        if (result == EWOULDBLOCK)
                                break;
                        }
                }
        }
}
```

3. 8 | epoll

- connect()와 accept()의 I/O 가능 이벤트

epoll은 connect()와 accept()에 서도 I/O 가능 이벤트를 받을 수 있다.

connect()는 send 이벤트와 동일하게, 그리고 accept() 는 receive 이벤트와 똑같이 취급됨.

리스닝 소켓에 대한 receive 이벤트를 받을 경우, accept()를 호출하면 새 TCP 연결의 소켓을 얻을 수 있다.

```
...
foreach(event in events) //❹
{
        s = event.socket;    //❺
        // 위 epoll.add에 들어갔던 값을 얻는다.
        userPtr = event.userPtr;
        // 수신? 송신?
        type = event.type;
        if (type = = ReceiveEvent)
        {
                if (s가 리스닝 소켓이면)
                {
                        s2 = s.accept();
                        }
                        else
                                {
                                        s.recv();
                                }
        }
}
```

3. 8 | Overlapped I/O 샘플

클라이언트 소스
(non overlapped)

```
#include <iostream>
#include <WS2tcpip.h>
using namespace std;
#pragma comment(lib, "Ws2_32.lib")
#define MAX_BUFFER        1024
#define SERVER_IP        "127.0.0.1"
#define SERVER_PORT        3500

int main()
{
   WSADATA WSAData;
   WSAStartup(MAKEWORD(2, 0), &WSAData);
   SOCKET serverSocket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, 0);
   SOCKADDR_IN serverAddr;
   memset(&serverAddr, 0, sizeof(SOCKADDR_IN));
   serverAddr.sin_family = AF_INET;
   serverAddr.sin_port = htons(SERVER_PORT);
   inet_pton(AF_INET, SERVER_IP, &serverAddr.sin_addr);
   connect(serverSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr));
   while (true) {
      char messageBuffer[MAX_BUFFER];
      cout << "Enter Message :";
      cin >> messageBuffer;
      int bufferLen = static_cast<int>(strlen(messageBuffer));
      int sendBytes = send(serverSocket, messageBuffer, bufferLen, 0);
      cout << "Sent : " << messageBuffer << "(" << sendBytes << " bytes)\n";
      int receiveBytes = recv(serverSocket, messageBuffer, MAX_BUFFER, 0);
      if (receiveBytes == 0) break;
      messageBuffer[receiveBytes] = 0;
      cout << "Received : " << messageBuffer << " (" << receiveBytes << " bytes)\n";
   }
   closesocket(serverSocket);
   WSACleanup();
}
```

3. 8 | Overlapped I/O 샘플

서버 소스
(overlapped I/O)

```
#include <iostream>
#include <map>
using namespace std;
#include <WS2tcpip.h>
#pragma comment(lib, "Ws2_32.lib")
#define MAX_BUFFER        1024
#define SERVER_PORT        3500
struct SOCKETINFO {
   WSAOVERLAPPED overlapped;
   WSABUF dataBuffer;
   SOCKET socket;
   char messageBuffer[MAX_BUFFER];
};
map <SOCKET, SOCKETINFO> clients;

void CALLBACK recv_callback(DWORD Error, DWORD dataBytes, LPWSAOVERLAPPED overlapped, DWORD lnFlags);
void CALLBACK send_callback(DWORD Error, DWORD dataBytes, LPWSAOVERLAPPED overlapped, DWORD lnFlags);

void CALLBACK recv_callback(DWORD Error, DWORD dataBytes, LPWSAOVERLAPPED overlapped, DWORD lnFlags)
{
   SOCKET client_s = reinterpret_cast<int>(overlapped->hEvent);
   if (dataBytes == 0) {
      closesocket(clients[client_s].socket);
      clients.erase(client_s);
      return;
   }  // 클라이언트가 closesocket을 했을 경우
   clients[client_s].messageBuffer[dataBytes] = 0;
   cout << "From client : " << clients[client_s].messageBuffer << " (" << dataBytes << ") bytes)\n";
   clients[client_s].dataBuffer.len = dataBytes;
   memset(&(clients[client_s].overlapped), 0x00, sizeof(WSAOVERLAPPED));
   clients[client_s].overlapped.hEvent = (HANDLE)client_s;
   WSASend(client_s, &(clients[client_s].dataBuffer), 1, &dataBytes, 0,
                              &(clients[client_s].overlapped), send_callback);
}
```

3. 8 | Overlapped I/O 샘플


```
void CALLBACK send_callback(DWORD Error, DWORD dataBytes, LPWSAOVERLAPPED overlapped, DWORD lnFlags)
{
   DWORD receiveBytes = 0;
   DWORD flags = 0;

   SOCKET client_s = reinterpret_cast<int>(overlapped->hEvent);

   if (dataBytes == 0) {
      closesocket(clients[client_s].socket);
      clients.erase(client_s);
      return;
   }  // 클라이언트가 closesocket을 했을 경우

   clients[client_s].dataBuffer.len = MAX_BUFFER;
   clients[client_s].dataBuffer.buf = clients[client_s].messageBuffer;
   cout << "Sent : " << clients[client_s].messageBuffer << " (" << dataBytes << " bytes)\n";
   memset(&(clients[client_s].overlapped), 0x00, sizeof(WSAOVERLAPPED));
   clients[client_s].overlapped.hEvent = (HANDLE) client_s;
   WSARecv(client_s, &clients[client_s].dataBuffer, 1, 0, &flags,
                                  &(clients[client_s].overlapped), recv_callback);
}
```

3. 8 | Overlapped I/O 샘플


```
int main()
{
   WSADATA WSAData;
   WSAStartup(MAKEWORD(2, 2), &WSAData);
   SOCKET listenSocket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
   SOCKADDR_IN serverAddr;
   memset(&serverAddr, 0, sizeof(SOCKADDR_IN));
   serverAddr.sin_family = AF_INET;
   serverAddr.sin_port = htons(SERVER_PORT);
   serverAddr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
   ::bind(listenSocket, (struct sockaddr*)&serverAddr, sizeof(SOCKADDR_IN));
   listen(listenSocket, 5);
   SOCKADDR_IN clientAddr;
   int addrLen = sizeof(SOCKADDR_IN);
   memset(&clientAddr, 0, addrLen);
   SOCKET clientSocket;
   DWORD flags;
   while (true) {
      clientSocket = accept(listenSocket, (struct sockaddr *)&clientAddr, &addrLen);
      clients[clientSocket] = SOCKETINFO{};
      memset(&clients[clientSocket], 0, sizeof(struct SOCKETINFO));
      clients[clientSocket].socket = clientSocket;
      clients[clientSocket].dataBuffer.len = MAX_BUFFER;
      clients[clientSocket].dataBuffer.buf = clients[clientSocket].messageBuffer;
      flags = 0;
      clients[clientSocket].overlapped.hEvent = (HANDLE)clients[clientSocket].socket;
      WSARecv(clients[clientSocket].socket, &clients[clientSocket].dataBuffer, 1, NULL,
               &flags, &(clients[clientSocket].overlapped), recv_callback);
   }
   closesocket(listenSocket);
   WSACleanup();
}
```

3. 8 | Overlapped I/O 샘플

- ## Overlapped 자료 구조

Overlapped I/O를 위해서는 WSAOVERLAPPED라는 자료구조를 만들어서 모든 Send/Recv에 초기화 해서 넣어주어야 한다.

또한, send/recv는 사용할 수 없고 WSASend/WSARecv를 사용해야 한다. 이 때 버퍼는 WSABUF를 사용해야 한다.

```
struct SOCKETINFO {
    WSAOVERLAPPED overlapped;
    WSABUF dataBuffer;
    SOCKET socket;
    char messageBuffer[MAX_BUFFER];
};
map <SOCKET, SOCKETINFO> clients;
```

- ## 다중 접속 처리

여러 개의 소켓이 필요하고 소켓마다 그 소켓에서 사용하는 버퍼 자료 구조가 필요하다.

```
map <SOCKET, SOCKETINFO> clients;
```

- ## Callback 함수에서 소켓

소켓 정보가 따로 오지 않는다. 따라서, WSAOVERLAPPED에 끼워 넣어야 한다.

```
clients[client_s].overlapped.hEvent = (HANDLE) client_s;
```

3.9 | 숙제 #3

게임 서버/클라이언트 프로그램 작성

내용

숙제 (#2)의 프로그램의 다중 사용자 버전
Client/Server 모델, 서버는 반드시 Overlapped I/O callback 을 사용할 것
클라이언트 10개 까지 접속 가능 하게 수정
옆의 클라이언트 에서도 다른 클라이언트의 말의 움직임이 보임

목적

Windows 다중 접속 Network I/O 습득

제약

Windows에서 Visual Studio로 작성 할 것
그래픽의 우수성을 보는 것이 아님

제출

다음 화요일 (10월 9일) 오전 10시
제목에 “2019 게임서버 학번 이름 숙제 3번”
Zip으로 소스를 묶어서 e-mail로 제출

3.9 | IOCP

- IOCP

IOCP는 소켓의 Overlapped I/O가 완료되면 이를 감지해서 사용자에게 알려 주는 역할을 한다.

사용자는 IOCP에서 I/O가 완료되었음을 알려 주는 완료 신호(completion event)를 꺼낼 수 (pop) 있다.

소켓 개수가 1만 개라고 하더라도 이 중에서 I/O가 완료된 것들만 IOCP를 이용해서 바로 얻을 수 있기 때문에, 모든 소켓에서 루프를 돌지 않아도 된다.

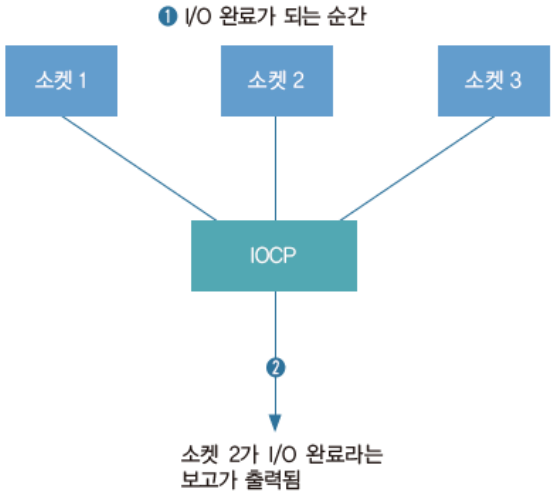


그림 3-20 소켓 2의 I/O가 완료하고 있는 상황

3.9 | IOCP

- IOCP

```
iocp = new iocp();                          //❶
foreach(s in sockets)
{
        iocp.add(s, GetUserPtr(s));                 //❷
        s.OverlappedReceive(data[s],
                receiveOverlapped[s]);          //❻
}

while (true)
{
        event = iocp.wait();                    //❸ ... 교재가 잘못 되었다. event는 하나씩 온다.
                                                //❺
        // 위 iocp.add에 들어갔던 값을 얻는다.
        userPtr = event.userPtr;
        ov = event.overlappedPtr;
        s = GetSocketFromUserPtr(userPtr);

        if (ov == receiveOverlapped[s])
        {
```

3.9 | IOCP


```
        // overlapped receive가 성공했으니,
        // 받은 데이터를 처리한다.
        Process(s, userPtr, data[s]);
        s.OverlappedReceive(data[s],
              receiveOverlapped[s]);   //6

    }
```

1 IOCP 객체를 만든다.

2 여러 소켓과 소켓에 대응하는 원하는 정수 값(epoll의userptr과 동일)을 IOCP에 추가한다.

6 추가된 소켓의 I/O 이벤트는 IOCP를 이용하여 감지할 수 있다. I/O를 하고 싶으면 Overlapped I/O를 건다.

3 모든 Overlapped status 객체에 대한 GetOverlappedResult 대신 IOCP에서 완료 신호를 꺼내 오는 함수를 호출한다. 이 함수는 사용자가 원하는 시간까지만 블로킹되며, 그 전에 이벤트가 생기면 즉시 리턴한다.

5 이벤트가 가리키는 Overlapped status 객체와 대응하는 userptr 값을 꺼내 온 후 원하는 처리를 한다.

6 추가로 I/O를 계속 하고 싶으면 OverlappedI/O를 또 걸면 된다.

3.9 | IOCP

- IOCP_Accept 처리

1. IOCP에 listen socket L을 추가했다면, L에서 TCP 연결을 받을 경우 이에 대한 완료 신호가 IOCP에 추가된다.
2. 단 사전에 이미 AcceptEx로 Overlapped I/O를 건 상태여야 한다.
3. IOCP로 L에 대한 이벤트를 얻어 왔지만, 앞서 Overlapped accep처럼 SO_UPDATE_ACCEPT_CONTEXT와 관련된 처리를 해 주어야 새 TCP 소켓 핸들을 얻어 올 수 있다.

- IOCP_성능상 유리한 기능

IOCP는 스레드 풀을 쉽게 구현할 수 있게 하는 반면, epoll은 그렇지 않다.

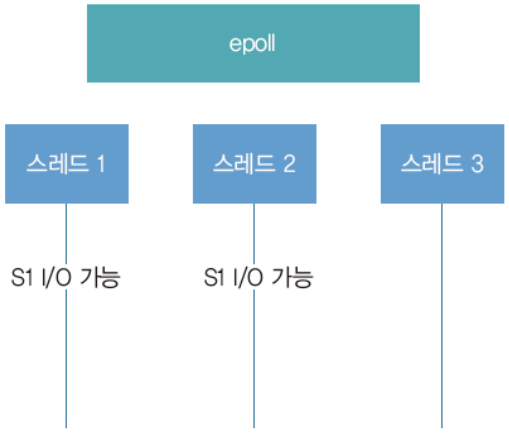


그림 3-21 한 소켓에 쌍인 두 이벤트가 두 스레드에 분배된 상황

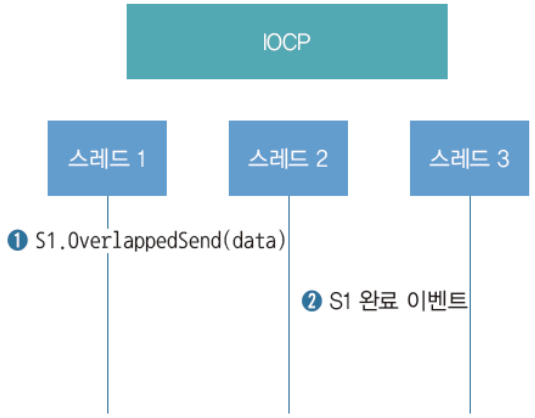


그림 3-22 Overlapped I/O를 걸어야만(❶) 완료 신호(❷)가 어느 스레드에서든지 발생하게 됨

3.9 | IOCP

- epoll에서 스레드 풀링 구현하기

  스레드 개수만큼 epoll 객체를 둔다.

  각 스레드는 자기만의 epoll을 처리한다.

  여러 소켓은 이들 epoll 중 하나에만 배정한다.

그림 3-23 스레드 개수만큼 epoll 객체를 두고 처리

- IOCP와 epoll의 비교와 정리

표 3-4 IOCP와 epoll의 장단점

구분	IOCP	epoll
블로킹을 없애는 수단	Overlapped I/O	논블록 소켓
블로킹 없는 처리 순서	1. Overlapped I/O를 건다. 2. 완료 신호를 꺼낸다. 3. 완료 신호에 대한 나머지 처리를 한다. 4. 끝나고 나서 다시 Overlapped I/O를 건다.	1. I/O 이벤트를 꺼낸다. 2. 꺼낸 이벤트에 대응하는 소켓에 대한 논블록 I/O를 실행한다.
지원 플랫폼	윈도에서만 사용 가능하다.	리눅스(=안드로이드)에서 사용 가능하다

3.9 | IOCP

- IOCP_성능상 유리한 기능

IOCP를 쓰는 윈도 서버에서는 연결 받기와 수신을 소켓 함수 호출 한 번으로 끝낼 수 있다.
부가적으로 연결된 소켓의 끝점 정보를 얻는 것도 같이 끝내 버릴 수 있어 프로그램 최적화에 유리함.

윈도 서버	리눅스 서버
AcceptEx GetAcceptExSockaddrs	accept recv getsockname getpeername

표 3-5 윈도 서버와 리눅스 서버의 커널 함수 호출 비교

3.10 | 더 읽을 거리

- 소켓 프로그래밍에서 고려해야 할 점
  - IP 주소 값이 아니라 도메인 이름으로 접속하려면 도메인 이름에서 IP 주소를 변환하는 일(2.4절 참고)을 해야 한다. 이를 위한 함수로 getaddrinfo()를 사용함.
  - iOS 등 FreeBSD 계열 운영체제에서 소켓 프로그래밍을 할 때는 e poll과 비슷한 것을 사용할 수 있는데, 그 이름은 kqueue이다.
  - 소켓 함수를 직접 이용하는 것이 불편하다면 boost 라이브러리의 a sio를 사용하는 것도 좋은 방법.
  - C/C++에서 소켓 함수는 매우 까다롭다. 하지만 C#, 자바 같은 고수준 언어에서는 소켓 클래스가 매우 정갈하게 정리되어 있어
    사용하기가 훨씬 편리하다