Part 03. 리눅스 소켓 프로그래밍

Chapter 07. 다중 입출력(2)

진행 순서

Chapter 07_01	poll
Chapter 07_02	epoll
Chapter 07_03	epoll 실습

Chapter 07_01 poll

select()의 단점

select()를 이용해 멀티플렉서를 구현한 서버가 있다고 가정해보겠습니다.
이때 접속한 클라이언트가 2개인데 accept된 소켓의 파일 디스크립터가 각각 3번과 900번 입니다.
이벤트 감지를 위하여
FD_SET(3, &readfds);
FD_SET(900, &readfds);
와 같이 세트에 추가하였습니다.
그리고 select() 시스템 콜을 아래와 같이 호출하게 되면
select(901, &readfds, NULL, NULL, NULL);
내부적으로는 0부터 900까지 루프를 돌면서 입력된 fd_set 구조체 readfds의 비트 마스크 값을 검사하게 됩니다.
실제 이벤트 감지가 필요한 소켓은 두 개이지만 불필요하게 루프를 많이 돌아야 합니다.
소켓에서 이벤트 발생이 초당 수십, 수백, 수천 번인 최악의 경우(worst case)라면 아주 많이 비효율적입니다.

이와 같은 select 함수의 단점을 해결하기 위해 poll, epoll을 사용합니다.
poll은 감시할 파일 디스크립터를 pollfd 구조체에 넣어서 관리하며,
시스템 콜은 내부적으로 pollfd 구조체의 개수만큼만 루프를 돌게 되어 있습니다.

Chapter 07_01 poll

#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

pollfd 구조체 fds는 감시할 파일 디스크립터와 이벤트의 정보를 담고 있는 배열 포인터 입니다.
호출자는 fds 배열의 항목 수를 nfds로 지정해야 합니다.
timeout 인수는 밀리 초를 의미합니다. 음수 값을 지정하면 무한을 의미하며, 0으로 지정할 경우 즉시 리턴됩니다.

성공 시 이벤트 또는 오류가 보고된 디스크립터의 수를 리턴합니다.
timeout 시간이 만료된 경우 0을 리턴합니다.
오류가 발생하면 -1이 리턴되고 errno가 적절히 설정됩니다.

감시할 파일 디스크립터 세트는 다음 형식의 구조체 배열 인 fds 인수에 지정됩니다.

```
struct pollfd {
   int fd;              /* file descriptor */
   short events;        /* requested events */
   short revents;       /* returned events */
};
```

Chapter 07_01 poll

```
struct pollfd {
   int fd;          /* file descriptor */
   short events;    /* requested events */
   short revents;   /* returned events */
};
```

fd는 열린 파일에 대한 파일 디스크립터를 포함합니다. 이 필드가 음수이면 해당 이벤트 필드는 무시되고 revents 필드는 0을 반환합니다.

events는 파일 디스크립터 fd에 관심이 있는 이벤트를 지정하는 비트 마스크 입력 매개 변수입니다. 이 필드가 0으로 지정되면 fd에 대한 모든 이벤트가 무시되고 revents는 0을 리턴합니다.

revents는 커널이 채워주는 실제로 발생한 이벤트 출력 매개 변수입니다. revents에 리턴된 비트는 이벤트에 지정된 비트 또는 POLLERR, POLLHUP 또는 POLLNVAL 값 중 하나를 포함 할 수 있습니다. (이 세 비트는 events 필드에서 의미가 없으며 해당 조건이 참일 때마다 revents 필드에서 설정됩니다.)

이벤트 및 revent에서 설정/반환 될 수 있는 비트는 <poll.h>에 정의되어 있습니다.

POLLIN	읽을 데이터가 있습니다.
POLLPRI	읽을 긴급한 데이터가 있습니다. (ex TCP OOB data)
POLLOUT	지금 쓰는 것은 차단되지 않습니다.
POLLERR	에러 발생 (출력만 해당)
POLLHUP	닫힌 연결에 쓰기 시도 (출력만 해당)
POLLNVAL	잘못된 요청: fd가 열리지 않습니다. (출력만 해당)

Chapter 07_01 poll

poll() 사용 예시

```
struct pollfd pollfds[10];
int nfds, pollret;

pollfds[0].fd = 4;
pollfds[0].events = POLLIN; /* 읽기 이벤트 감지 */
pollfds[1].fd = 100;
pollfds[1].events = POLLOUT; /* 쓰기 이벤트 감지 */

nfds = 2;
pollret = poll(pollfds, nfds, -1);
if (pollret == -1) {
   /* error */
} else if (pollret > 0) {
   for (i = 0; i < nfds; i++) {
      if (pollfds[i].revents & POLLIN) {
         /* 읽기 이벤트 감지 */
      } else if (pollfds[i].revents & POLLOUT) {
         /* 쓰기 이벤트 감지 */
      } else if (…) {
      }
   }
}
```

Chapter 07_02 epoll (event poll)

epoll은 1983년에 개발된 select와 1992년에 개발된 poll을 대체할 수 있는 고성능 I/O 멀티플렉서입니다.

epoll의 특징

1. epoll은 select/poll과 달리 호출할 때마다 파일 디스크립터 정보를 입력할 필요가 없습니다.
2. 파일 디스크립터를 추가/수정/삭제 하는 함수와 감시하는 함수가 분리되었습니다.
3. 엣지 트리거(edge trigger) 방식을 사용할 수 있습니다. (기본은 select/poll과 동일한 레벨 트리거 level trigger)

epoll 주요 함수

1. epoll_create()는 epoll 인스턴스를 작성하고 해당 인스턴스를 참조하는 파일 디스크립터를 리턴합니다.
2. epoll_ctl()을 통해 특정 파일 디스크립터에 대한 관심을 등록하거나 해제합니다.
3. epoll_wait()는 현재 사용 가능한 이벤트가 없는 경우 호출 스레드를 차단하여 I/O 이벤트를 기다립니다.

Chapter 07_02 epoll (event poll)

레벨 트리거(Level Trigger) vs 엣지 트리거(Edge Trigger)

개념적으로 레벨 트리거는 일정 기준치 이상에 도달했는지를 감시하고, 엣지 트리거는 이전 상태에서 변화가 생겼는지를 감시합니다.

레벨 트리거를 사용하는 poll을 이용해서 소켓 수신 버퍼를 감시하는 경우,
이벤트가 발생하여 poll이 값을 리턴하고 5번 소켓에 POLLIN 이벤트가 있었다고 생각해 보겠습니다.
5번 소켓의 버퍼를 recv하지 않고 다시 poll을 호출하면 poll은 다시 5번 소켓의 수신 버퍼에 데이터가 있으므로
또 POLLIN 이벤트를 감지하게 됩니다.

엣지 트리거 방식을 이용하여 epoll을 사용하는 경우 위와 같은 상황에서 recv 하지 않고 다시 epoll을 호출해도
이전 상태에서 추가로 수신된 이벤트가 없다면 감지하지 않습니다. 게다가 수신 버퍼에서 일부만을 읽었다고 해도
수신 버퍼에 새로운 추가가 있지 않는 한 엣지 트리거는 감지하지 않습니다.

결론적으로 엣지 트리거가 마지막에 감지한 이전 상태에서 변화가 없으므로 새로운 데이터가 도착하여 상태에 변화가 생길 때까지 엣지 트리거는 감지하지 않습니다.

소켓을 통해 데이터를 수신 받을 때 읽어서 분석해야 할 데이터가 100 바이트라고 가정하면,
데이터가 조각되어 50 바이트 씩 수신된다면 50 바이트를 읽어 들이고 저장한 후 나머지 50바이트를 다시 읽어 들여 저장한 데이터와 합친 후에 데이터를 분석해야 하는데,
엣지 트리거를 이용한다면 50 바이트만 수신 버퍼에 있을 경우 recv 하지 않고 다시 감시하여 수신된 버퍼가 100 바이트가 될 경우 처리하면 됩니다.

Chapter 07_02 epoll (event poll)

```
#include <sys/epoll.h>

int epoll_create(int size);
```

epoll_create()는 epoll 인스턴스를 만듭니다. Linux 2.6.8부터 size 인수는 무시되지만 0보다 커야 합니다.
(최신의 커널은 필요한 데이터 구조의 size를 동적으로 조정합니다.)

epoll_create()는 새 epoll 인스턴스를 참조하는 파일 디스크립터를 리턴합니다. 이 파일 디스크립터는 epoll 인터페이스에 대한 모든 후속 호출에 사용됩니다. 더 이상 필요하지 않은 경우, epoll_create()에 의해 리턴 된 파일 디스크립터는 close()를 사용하여 닫아야 합니다. epoll 인스턴스를 참조하는 모든 파일 디스크립터가 닫히면 커널은 인스턴스를 삭제하고 재사용을 위해 관련 자원을 해제합니다.

성공하면 이러한 시스템 호출은 음이 아닌 파일 디스크립터를 리턴합니다. 오류가 발생하면 -1이 반환되고 오류를 나타내도록 errno가 설정됩니다.

Chapter 07_02 epoll (event poll)

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

이 시스템 호출은 파일 디스크립터 epfd가 참조하는 epoll 인스턴스에서 제어 조작을 수행합니다. 대상 파일 디스크립터 fd에 대해 조작 op를 수행하도록 요청합니다.

op의 종류

EPOLL_CTL_ADD
: 파일 디스크립터 epfd가 참조하는 epoll 인스턴스에 대상 파일 디스크립터 fd를 등록하고 event 이벤트를 fd에 링크 된 내부 파일과 연관 시킵니다.

EPOLL_CTL_MOD
: 대상 파일 디스크립터 fd와 연관된 이벤트 이벤트를 변경합니다.

EPOLL_CTL_DEL
: epfd가 참조하는 epoll 인스턴스에서 대상 파일 디스크립터 fd를 제거합니다.

Chapter 07_02 epoll (event poll)

event 인수는 파일 디스크립터 fd에 링크 된 오브젝트를 설명합니다. epoll_event 구조체는 다음과 같이 정의됩니다 :

```
typedef union epoll_data {
   void       *ptr;
   int        fd;
   uint32_t    u32;
   uint64_t    u64;
} epoll_data_t;

struct epoll_event {
   uint32_t    events;     /* Epoll events */
   epoll_data_t data;       /* User data variable */
};
```

Chapter 07_02 epoll (event poll)

events 멤버는 다음 이벤트 유형을 사용하여 구성된 비트 세트입니다.

EPOLLIN 연관된 파일 디스크립터는 read 조작에 사용 가능합니다.

EPOLLOUT 연관된 파일 디스크립터는 write 조작에 사용 가능합니다.

EPOLLRDHUP (since Linux 2.6.17)스트림 소켓 피어 닫힌 연결 또는 연결 절반 쓰기를 종료합니다.

EPOLLPRI read 조작에 사용할 수 있는 긴급 데이터가 있습니다. (ex. TCP OOB Data)

EPOLLERR 연관된 파일 디스크립터에서 오류 조건이 발생했습니다.
epoll_wait는 항상이 이벤트를 기다립니다. 이벤트에서 설정할 필요는 없습니다.

EPOLLHUP 관련 파일 디스크립터에서 끊기가 발생했습니다.
epoll_wait는 항상이 이벤트를 기다립니다. 이벤트에서 설정할 필요는 없습니다.

EPOLLET 연관된 파일 디스크립터에 대해 엣지 트리거 된 동작을 설정합니다.
epoll의 기본 동작은 레벨 트리거입니다.

EPOLLONESHOT (since Linux 2.6.2) 이벤트 감시를 일회용으로 사용합니다.
한번 감지된 후에는 해당 파일 디스크립터의 이벤트 마스크를 비활성화 시키므로
EPOLL_CTL_MOD를 이용해 이벤트 마스크를 재설정할 때까지 이벤트를 감지하지 않습니다.

Chapter 07_02 epoll (event poll)

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

epoll_wait 시스템 콜은 파일 디스크립터 epfd가 참조하는 epoll 인스턴스의 이벤트를 기다립니다. events가 가리키는 메모리 영역에는 호출자가 사용할 수 있는 이벤트가 포함됩니다. epoll_wait()에서 최대 maxevents를 리턴합니다. maxevents 인수는 0보다 커야 합니다.

timeout 인수는 epoll_wait()가 대기할 최소 밀리 초 수를 지정합니다. 값을 -1로 지정하면 epoll_wait()가 무한대로 차단되고 0 일 경우 사용 가능한 이벤트가 없더라도 epoll_wait()가 즉시 리턴되도록 합니다.

성공하면 epoll_wait()는 요청 된 I/O에 대해 준비된 파일 디스크립터 수를 리턴합니다.
요청 된 timeout (밀리 초) 동안 파일 디스크립터가 준비되지 않은 경우 0을 리턴합니다.
오류가 발생하면 -1을 리턴하고 errno가 적절하게 설정됩니다.

Chapter 07_02 epoll (event poll)

man epoll
example for suggested usage

```
#define MAX_EVENTS 10
struct epoll_event ev, events[MAX_EVENTS];
int listen_sock, conn_sock, nfds, epollfd;

/* Set up listening socket, 'listen_sock' (socket(),
bind(), listen()) */

epollfd = epoll_create(10);
if (epollfd == -1) {
   perror("epoll_create");
   exit(EXIT_FAILURE);
}

ev.events = EPOLLIN;
ev.data.fd = listen_sock;
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1) {
   perror("epoll_ctl: listen_sock");
   exit(EXIT_FAILURE);
}
```

```
for (;;) {
   nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
   if (nfds == -1) {
      perror("epoll_pwait");
      exit(EXIT_FAILURE);
   }

   for (n = 0; n < nfds; ++n) {
      if (events[n].data.fd == listen_sock) {
         conn_sock = accept(listen_sock,
                  (struct sockaddr *) &local, &addrlen);
         if (conn_sock == -1) {
            perror("accept");
            exit(EXIT_FAILURE);
         }
         setnonblocking(conn_sock);
         ev.events = EPOLLIN | EPOLLET;
         ev.data.fd = conn_sock;
         if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock, &ev) == -1) {
            perror("epoll_ctl: conn_sock");
            exit(EXIT_FAILURE);
         }
      } else {
         do_use_fd(events[n].data.fd);
      }
   }
}
```