epoll

Pętla zdarzeń

Programista przygotowuje kod (funkcje) obsługi możliwych zdarzeń, następnie w pętli czeka na zdarzenie i wywołuje kod powiązany ze zdarzeniem. Aplikacja może być jedno- lub wielowątkowa.

Przykład:

```
while() {
	concrete_event = wait_for_any_event(observed_elements);
	handle_event(concrete_event); //możliwe że w innym wątku
}
```

Koncepcja asynchronicznego serwera

- Tworzymy gniazdo serwera
- Dodajemy gniazda podłączanych klientów do zbioru obserwowanych
- W pętli oczekujemy na zdarzenia skojarzone z dowolnym elementem ze zbioru obserwowanych
  - dla gniazd klientów
    - można wysłać dane (~ jest miejsce w buforze nadawczym)
    - odebrać dane (~ są dane do odbioru)
  - dla gniazda serwera (jeśli je również obsługujemy przez epoll)
    - jest połączenie do odebrania (można wykonać accept). Po odebraniu połączenia, przestawiamy w tryb nieblokujący i dodajemy do zbioru obserwowanych.
- Przy zakończeniu połączenia z klientem, usuwamy go ze zbioru obserwowanych

epoll - schemat użycia

epoll_create1() //utworzenie deskryptora epoll

epoll_ctl(..., EPOLL_CTL_ADD, …) //rejestracja deskryptora

epoll_wait() //oczekiwanie na zdarzenia

(*) epoll_ctl(..., EPOLL_CTL_MOD, …) //modyfikacja parametrów deskryptora

(*) epoll_ctl(..., EPOLL_CTL_DEL, …) //wyrejestrowanie deskryptora

(*) epoll_ctl(..., EPOLL_CTL_ADD, …) //ponowna rejestracja deskryptora

Sposoby wykorzystania epoll

- level-triggered (domyślny) - zwraca kontrolę za każdym razem gdy urządzenie jest gotowe (są dane do odebrania / jest miejsce w buforze nadawczym)
- edge-triggered (flaga EPOLLET) - zwraca kontrolę gdy przyjdą nowe dane.
- EPOLLONESHOT - po zwróceniu kontroli wyrejestrowuje deskryptor (można dodać go ponownie przez epoll_ctl)
- EPOLLEXCLUSIVE - jeśli jest wiele wątków, każdy ma ustawioną tę flagę i ma w swoich (różnych) instancjach epoll ten sam deskryptor, wybudzony zostanie tylko 1 wątek. Domyślnie wybudzone byłyby wszystkie.

Przekazywanie danych

Przy dodawaniu deskryptora do zbioru obserwowanych:

epoll_ctl(epoll_fd, EPOLL_CTL_ADD, socket_fd, &event);

Podajemy argument event (struct epoll_event*), w którym możemy określić na jakie zdarzenia (np. EPOLLIN) chcemy nasłuchiwać (event.events).

Możemy też przekazać dodatkowe dane (event.data). Do wyboru: albo który deskryptor wyemituje zdarzenie (event.data.fd), albo uint32 (event.data.u32), albo uint64 (event.data.u64), albo wskaźnik na dowolną rzecz (event.data.ptr). W szczególności możemy przekazać wskaźnik na strukturę lub funkcję (callback). Uwaga: wybrać można tylko jedną opcję (event.data jest unią).

Nieblokujące I/O

Przestawienie trybu:

- po utworzeniu gniazda
  - fcntl(socket_fd, F_SETFL, O_NONBLOCK, 1);
- przy odbiorze/wysyłaniu, bez wcześniejszego fcntl()
  - send(..., MSG_DONTWAIT) / recv(..., MSG_DONTWAIT)
- w trakcie tworzenia gniazda
  - socket(..., SOCK_STREAM | SOCK_NONBLOCK, 0);
  - accept4(..., SOCK_NONBLOCK);

Jeśli wywołanie read()/write()/accept() wymagałoby czekania funkcja zwraca -1 i ustawia errno na EAGAIN lub EWOULDBLOCK.

Przy connect, operacja zostaje rozpoczęta w tle, zwraca -1 i ustawia errno na EINPROGRESS.

MSG_DONTWAIT

Niektóre funkcje sieciowe (np. recv, send) pozwalają na ustawienie w polu flag wartości MSG_DONTWAIT, która wykonuje żądaną operację w trybie nieblokującym niezależnie od tego w jakim trybie pracuje gniazdo

recv(fileDescriptor, buff, buffSize, MSG_DONTWAIT);

Zadania

Zadanie 1 podstawowy epoll

Napisz serwer korzystając z epoll i gniazd nieblokujących. Przykładowy kod w “man 7 epoll”. Flagi i definicje struktur w “man epoll_ctl”.

a) Dla każdego połączenia nasłuchuj na dane (EPOLLIN), po odebraniu wypisz je na standardowe wyjście serwera. Podłącz kilka klientów (np. nc) i przetestuj działanie.
b) Dodatkowo oczekuj na EPOLLOUT, kiedy będzie zgłoszony wypisz “deskryptor gotowy do wysyłania” na standardowe wyjście. Podłącz 1 klienta, obserwuj konsolę serwera.

Zadanie 2 EPOLLET, EPOLLONESHOT

a) Zmodyfikuj serwer z poprzedniego zadania dodając do gniazd flagę EPOLLET.
b) Zmodyfikuj serwer z poprzedniego zadania dodając do gniazd flagę EPOLLONESHOT. W razie potrzeby ponownie dodaj deskryptor do zbioru obserwowanych.

Zadanie 3 dołączanie danych

Korzystając z epoll stwórz serwer, który będzie odbierał dane od klienta do napotkania nowej linii, po czym odeśle je zwrotnie do klienta. Dla każdego gniazda stwórz bufor, w którym będzie kumulowany komunikat i przekaż go przez event.data.ptr (pamiętaj też o przekazaniu samego deskryptora).

Dodatki

write() i SIGPIPE

Próba wysłania danych na zamknięte połączenie spowoduje wygenerowanie sygnału SIGPIPE (“write to pipe with no readers”). Taka sytuacja wystąpi np. gdy druga strona połączenia wywołała na nim close(). Domyślna akcja dla sygnału SIGPIPE to term[ination], czyli zakończenie **programu**.

man 7 signal -> spis sygnałów, łącznie z domyślnymi akcjami

Obsługa sygnałów

Ignorowanie sygnału w programie:

```
struct sigaction act;
memset(&act, '\0', sizeof(act));
act.sa_handler = SIG_IGN;
if (sigaction(SIGPIPE, &act, NULL)) {
    perror("sigaction");
    exit(1);
}
```

Reagowanie na zdarzenie:

```
int w = write(...)
if ( w < 0 && w == EPIPE ) {
        //obsługa złamanego ‘pipe’
}
```

Obsługa sygnałów - selektywne ignorowanie

Kiedy blokowanie SIGPIPE w programie nie jest najlepsze? Np. wtedy gdy wywołania innych funkcji niż write/send sieciowy może spowodować SIGPIPE. Blokowanie w całym programie przeszkadza wtedy w ich obsłudze.

send(..., MSG_NOSIGNAL) //flaga zapobiegająca SIGPIPE w tym wywołaniu

Ustawienie opcji SO_NOSIGPIPE na gnieździe (niedostępne w sys. Linux).

Obsługa sygnałów - przechwytywanie

Wywołanie sigaction analogicznie jak poprzednio, poza polem sa_handler, w którym podajemy naszą funkcję obsługi sygnału. Alternatywnie można zastąpić przypisanie do pola sa_handler przypisaniem do pola sa_sigaction i ustawić flagę SA_SIGINFO, wtedy nasza funkcja obsługi dostanie więcej informacji.

Uwaga, w wielowątkowej aplikacji sygnał obsłuży dowolny z wątków, który go nie zablokował. Wątki dziedziczą sigmask po rodzicu. Można przestawić sigmask wybranego wątku przez pthread_sigmask(). Dzięki temu można np. wybrać jeden wątek, który będzie obsługiwał sygnał (reszta blokuje ten sygnał w swoim sigmask).

Inne opcje gniazd godne uwagi

Oprócz SO_REUSEADDR, SO_REUSEPORT:

- SO_KEEPALIVE
- SO_LINGER
- SO_RCVBUF
- SO_SNDBUF
- TCP_NODELAY

Dodatki: zadanie 1

Napisz klienta, który wysyła w pętli, co sekundę, komunikaty do serwera. Jako serwer uruchom program netcat (z przełącznikiem -l). Uruchom klienta przez “strace ./klient”. Po odebraniu przez serwer kilku komunikatów wyłącz serwer przez ctrl-c. Zobacz ostatnie komunikaty wypisane przez strace.

Porównaj z uruchomieniem klienta bez strace.

Dodatki: zadanie 2

Stwórz serwer, który akceptuje połączenia i je ignoruje. Stwórz klienta TCP który: łączy się pod podany adres, ustawia tryb nieblokujący w pętli: wysyła dane, wypisuje kolejny numer i ilość wysłanych danych, jeśli wysłał mniej danych niż chciał, kończy się.

Porównać połączenie na localhost i na inną maszynę.