Programowanie sieciowe
Architektury programów sieciowych

Michał Kalewski

Instytut Informatyki
Politechnika Poznańska
ul. Piotrowo 2, 60–965 Poznań

mkalewski@cs.put.poznan.pl
http://www.cs.put.poznan.pl/mkalewski

Poznań · 4 czerwca 2019 r.

Plan wykładu

Multipleksacja wejścia/wyjścia

Funkcja systemowa `select(2)`

```
int select(int nfds, fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

Parametry:

`nfds` największa wartość deskryptora w sprawdzanych zbiorach powiększona o jeden.

`readfds, writefds, errorfds` zbiory deskryptorów sprawdzanych do gotowości: odczytu, zapisu, odebrania wyjątku.

`timeout` własny limit czasu blokowania funkcji.

Wartość zwracana: liczba deskryptorów w sprawdzanych zbiorach, `0` przypadku upływu limitu czasu, `-1` w przypadku błędu.

Makra: `FD_SET(fd, &fdset)`; `FD_CLR(fd, &fdset)`; `FD_ISSET(fd, &fdset)`; `FD_COPY(&fdset_orig, &fdset_copy)`; `FD_ZERO(&fdset)`;

Funkcja systemowa poll(2) (1/2)

```
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

Parametry:

`fds` wskaźnik na tablicę struktur `pollfd`.

`nfds` rozmiar tablicy `fds`.

`timeout` własny limit czasu blokowania funkcji (w milisekundach).

Wartość zwracana: liczba deskryptorów gotowych do operacji wejścia/wyjścia, `0` przypadku upływu limitu czasu, `-1` w przypadku błędu.

Funkcja systemowa poll(2) (2/2)

```
1 struct pollfd {
2   int    fd;       /* file descriptor */
3   short  events;   /* events to look for */
4   short  revents;  /* events returned */
5 };
```

Wybrane stałe określające stany/zdarzenia deskryptorów (*maski bitowe*):

- `POLLIN` — dane mogą zostać odczytane w sposób nieblokujący;
- `POLLOUT` — dane mogą zostać zapisane w sposób nieblokujący;
- `POLLERR` — wyjątek deskryptora;
- `POLLHUP` — rozłączenie gniazda sieciowego.

Mechanizm systemowy epoll(7) (1/5)

W systemach operacyjnych GNU/Linux dostępny jest mechanizm o nazwie `epoll(7)` o zwiększonej efektywności obsługi dużej liczy deskryptorów, który dodatkowo wspiera dwa podejścia: *level-triggered* oraz *edge-triggered*.

Podejście *edge-triggered* monitoruje tylko *zmianę* stanu deskryptora; podejście *level-triggered* (domyślne) monitoruje zmianę stanu deskryptora oraz jego gotowość do wykonania operacji odczytu lub zapisu.

Mechanizm systemowy `epoll(7)` składa się z trzech funkcji systemowych:

- `epoll_create1(2)`: otwierającej *deskryptor* mechanizmu;
- `epoll_ctl(2)`: wykonującej operacje *kontrolne* na deskryptorze mechanizmu;
- `epoll_wait(2)`: oczekujące na *zdarzenia wejścia/wyjścia* na deskryptorze mechanizmu.

Mechanizm systemowy epoll(7) (2/5)

```
int epoll_create1(int flags);
```

Parametry:

`flags` flagi mechanizmu; jedyną dostępną flagą jest `EPOLL_CLOEXEC`.

Wartość zwracana: deskryptor mechanizmu lub `-1` w przypadku błędu.

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

Parametry:

`epfd` deskryptor mechanizmu `epoll(7)`.

`op` numer (rodzaj) operacji do wykonania na deskryptorze `fd`.

`fd` docelowy deskryptor pliku.

`event` opis deskryptora `fd` na potrzeby mechanizmu `epoll(7)`.

Wartość zwracana: `0` lub `-1` w przypadku błędu.

Mechanizm systemowy epoll(7) (3/5)

Numery (rodzaje) operacji do wykonania na deskryptorze `fd`:

- `EPOLL_CTL_ADD`: rejestracja nowego deskryptora w mechanizmie;
- `EPOLL_CTL_MOD`: zmiana opisu `event` dla zarejestrowanego już deskryptora;
- `EPOLL_CTL_DEL`: usunięcie rejestracji deskryptora z mechanizmu.

Wybrane stałe określające stany/zdarzenia deskryptorów (pole `events` w strukturze `epoll_event`, *maski bitowe*):

- `EPOLLIN` — deskryptor jest gotowy do wykonania operacji zapisu;
- `EPOLLOUT` — deskryptor jest gotowy do wykonania operacji odczytu;
- `EPOLLERR` — wyjątek deskryptora (stan ten jest zawsze obsługiwany przez funkcję systemową `epoll_wait(2)`);
- `EPOLLHUP` — rozłączenie gniazda sieciowego (stan ten jest zawsze obsługiwany przez funkcję systemową `epoll_wait(2)`).

Mechanizm systemowy epoll(7) (4/5)

```
1  typedef union epoll_data {
2    void        *ptr;
3    int          fd;
4    uint32_t     u32;
5    uint64_t     u64;
6  } epoll_data_t;
7
8  struct epoll_event {
9    uint32_t     events;      /* Epoll events */
10   epoll_data_t data;        /* User data variable */
11 };
```


Mechanizm systemowy epoll(7) (5/5)

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

Parametry:

`epfd` deskryptor mechanizmu `epoll(7)`.

`events` tablica zdarzeń na zarejestrowanych w mechanizmie `epoll(7)` deskryptorach plików.

`maxevents` maksymalny rozmiar tablicy `events`.

`timeout` własny limit czasu blokowania funkcji (w milisekundach).

Wartość zwracana: rozmiar tablicy `events`, `0` w przypadku upływu limitu czasu, `-1` w przypadku błędu.

</> Mechanizm systemowy epoll(7) — przykład

```
1  for (;;) {
2    nfds = epoll_wait(efd, events, maxevents, -1);
3    for (n = 0; n < nfds; ++n) {
4      if (events[n].data.fd == sfd) {
5        cfd = accept(sfd, (struct sockaddr*)&addr, &addrlen);
6        setnonblocking(cfd);
7        ev.events = EPOLLIN;
8        ev.data.fd = cfd;
9        epoll_ctl(efd, EPOLL_CTL_ADD, cfd, &ev);
10     } else {
11       do_use_fd(events[n].data.fd);
12     }
13   }
14 }
```

Pozostałe rozwiązania multipleksacji wejścia/wyjścia

W systemach operacyjnych UNIX dostępna jest funkcja systemowa `kqueue(2)` służąca do informowania procesów użytkownika o zdarzeniach w systemie.
W https://en.wikipedia.org/wiki/Kqueue

W systemach operacyjnych Windows dostępne są funkcje `select()` oraz `WSAPoll()` (w ramach biblioteki `winsock2.h`).
https://docs.microsoft.com/en-us/windows/desktop/api/winsock2/

> Stosując multipleksację wejścia/wyjścia operacje sieciowe powinny działać w trybie nieblokującym.

Architektury klientów sieciowych

Architektury klientów sieciowych

Podstawowe architektury implementacji klientów sieciowych:

- pojedynczy proces, blokujące operacje wejścia/wyjścia;
- pojedynczy proces, multipleksacja wejścia/wyjścia;
- dwa (dodatkowe) procesy: proces odbierający dane i proces transmitujący dane;
- dwa (dodatkowe) wątki: wątek odbierający dane i wątek transmitujący dane.

Architektury serwerów sieciowych

Serwer iteracyjny i współbieżny

Serwer sieciowy *iteracyjny* obsługuje połączenie (komunikację) tylko z pojedynczym klientem.

Serwer sieciowy *współbieżny* obsługuje połączenia (komunikację) z wieloma klientami.

Serwer sieciowy współbieżny może być realizowany przez jeden lub wiele (pod)procesów lub wątków.

W serwerach sieciowych współbieżnych jedno-procesowych wykorzystuje się funkcje multipleksacji wejścia/wyjścia.

Pozostałe architektury serwerów sieciowych współbieżnych wymagają (dodatkowo) stosowania procesów potomnych lub wątków.

Podstawowe architektury współbieżnych serwerów sieciowych

1. Osobny proces potomny dla każdego podłączonego klienta.
   - Proces macierzysty jedynie akceptuje połączenia klientów i tworzy procesy potomne.
   - Procesy potomne zamykają gniazdo główne serwera.
   - Proces macierzysty zamyka gniazdo klienta.
   - Proces macierzysty odczytuje statusy zakończenia procesów potomnych poprzez obsługę sygnału `SIGCHLD` (lub wymusza się ignowanie statusów zakończenia procesów potomnych).
2. Osobny wątek dla każdego podłączonego klienta.
   - Proces macierzysty jedynie akceptuje połączenia klientów i tworzy wątki.
   - Deskryptor gniazda klienta przekazywany jest jako argument funkcji wątku.
   - Wątek może zostać odłączony — funkcja `pthread_detach(3)`.

Optymalizacja: utworzenie (przygotowanie) procesu potomnego/wątku przed akceptacja połączenia.

Problem C10k

Problem **C10k** dotyczy konstrukcji serwerów sieciowych współbieżnych obsługujących jednocześnie dziesiątki tysięcy (*10k*) połączeń klientów (*C*).

Termin ten został wprowadzony w 1999 roku przez Dana Kegela w artykule pt. „The C10K problem”.

Artykuł ten dostępny jest w Internecie pod następującym adresem: http://www.kegel.com/c10k.html.

Rozwiązania podstawowych architektur współbieżnych serwerów sieciowych nie są adekwatne dla takiego typu obciążeń.

Architektury współbieżnych serwerów sieciowych (1/3)

1. Pula procesów potomnych bez blokowania funkcji systemowej `accept(2)`.
   - Proces macierzysty jedynie tworzy główne gniazdo sieciowe oraz N procesów potomnych.
   - Każdy proces potomny wykonuje funkcje systemową `accept(2)` niezależnie na tym samym deskryptorze głównego gniazda sieciowego (działa jak serwer sieciowy iteracyjny).
   - Proces macierzysty może zarządzać liczą procesów potomnych w puli.

Wykonanie funkcji systemowej `connect(2)` powoduje „zbudzenie” (zmiana stanu procesu) wszystkich procesów oczekujących na funkcji systemowej `accept(2)`, lecz tylko jeden z nich akceptuje połączenie (funkcja systemowa `accept(2)` zwraca sterowanie).

Architektury współbieżnych serwerów sieciowych (2/3)

2. Pula procesów potomnych z blokowaniem funkcji systemowej `accept(2)`.
   - Procesy potomne synchronizują pomiędzy sobą dostęp do wywołania funkcji systemowej `accept(2)`: `lock(); accept(...); unlock();`
3. Pula procesów potomnych z przekazywaniem deskryptora gniazda sieciowego klienta.
   - Jedynie proces macierzysty akceptuje połączenia klientów, a następnie przekazuje deskryptor nowego połączenia do jednego z procesów potomnych.

To rozwiązanie wymaga mechanizmu komunikacji pomiędzy procesem macierzystym a procesami potomnymi; proces macierzysty musi także posiadać informacje o tym, które procesy potomne są zajęte, a które są wolne.

Rozwinięciem tego podejścia jest *wzorzec reaktora*.

Architektury współbieżnych serwerów sieciowych (3/3)

4. Pula wątków bez blokowania funkcji systemowej `accept(2)`.
5. Pula wątków z blokowaniem funkcji systemowej `accept(2)`.
   - Synchronizacja pomiędzy wątkami z użyciem funkcji `pthread_mutex_lock(3)` i `pthread_mutex_unlock(3)`.
6. Pula wątków z przekazywaniem deskryptora gniazda sieciowego klienta.

   (Komunikacja w ramach pojedynczego procesu).

Wzorzec reaktora

Schemat konstrukcji serwera zgodnie ze **wzorcem reaktora**:

- **Demultiplekser** (ang. *synchronous event demultiplexer*): sprawdza stan deskryptorów dla operacji wejścia/wyjścia; jeżeli któryś z tych deskryptorów jest gotowy do wykonania operacji odczytu/wzapisu, to jest on przekazywany do *dyspozytora*.
- **Dyspozytor** (ang. *dispatcher*): sprawdza stan *wykonawców* i przekazuje otrzymany od *demultipleksera* deskryptor do wolnego/właściwego *wykonawcy*.
- **Wykonawca** (ang. *request handler*): realizuje operację wejścia/wyjścia tylko w zakresie aktualnie możliwym do wykonania zgodnie z bieżącą informacją z maszyny stanowej dla obsługiwanego deskryptora.

Wykonawcy utrzymują **maszynę stanową** dla wszystkich połączeń/deskryptorów z informacją o aktualnym stanie realizacji operacji wejścia/wyjścia i przejściach do kolejnych stanów zgodnie z obsługiwanym protokołem.

Podsumowanie

Podsumowanie

- Multipleksacja wejścia/wyjścia: funkcje systemowe `select(2)`, `poll(2)` oraz mechanizm `epoll(7)`.
- Architektury klientów sieciowych.
- Architektury serwerów sieciowych:
  - osobny proces potomny dla każdego podłączonego klienta;
  - osobny wątek dla każdego podłączonego klienta;
  - pula procesów potomnych bez blokowania funkcji systemowej `accept(2)`;
  - pula procesów potomnych z blokowaniem funkcji systemowej `accept(2)`;
  - pula procesów potomnych z przekazywaniem deskryptora gniazda sieciowego klienta;
  - pula wątków bez blokowania funkcji systemowej `accept(2)`;
  - pula wątków z blokowaniem funkcji systemowej `accept(2)`;
  - pula wątków z przekazywaniem deskryptora gniazda sieciowego klienta;
  - wzorzec reaktora.

Pytania i zadania

Pytania i zadania — praca własna

1. Sprawdź jaka jest różnica pomiędzy funkcjami systemowymi `select(2)` i `pselect(2)` oraz `epoll_wait(2)` i `epoll_pwait(2)`.
2. Dowiedz się na czym polega mechanizm `SO_REUSEPORT` i jak może on zostać wykorzystany do realizacji współbieżnego serwera sieciowego.
3. Przeczytaj dokument pt. „The C10K problem”, który dostępny jest w Internecie pod następującym adresem: http://www.kegel.com/c10k.html

Dziękuję za uwagę!