Sieci Komputerowe

Wykład 5 — Programowanie serwerów sieciowych

Szymon Acedański

Instytut Informatyki
Uniwersytet Warszawski

5 kwietnia 2017

Hasła z labów

`socket(2)`, `SOCK_STREAM`, `SOCK_DGRAM`, `AF_INET`, `AF_INET6`, `AF_UNSPEC`, `struct sockaddr_in`, `socklen_t`, `bind`, `listen`, `accept`, `read`, `write`, `recvfrom`, `sendto`

Uwaga z labów — SOCK_STREAM a read

Spróbujmy stworzyć funkcję, która odbierze nagłówek
z ustanowionego wcześniej połączenia TCP.

```
int recv_header(int fd, struct header* header) {
    if (read(fd, header, sizeof(*header)) <= 0) {
        return ERROR;
    }
    return OK;
}
```

Uwaga z labów — SOCK_STREAM a read

Spróbujmy stworzyć funkcję, która odbierze nagłówek
z ustanowionego wcześniej połączenia TCP.

```
int recv_header(int fd, struct header* header) {
    if (read(fd, header, sizeof(*header)) <= 0) {
        return ERROR;
    }
    return OK;
}
```

Uwaga z labów — SOCK_STREAM a read

Spróbujmy stworzyć funkcję, która odbierze nagłówek
z ustanowionego wcześniej połączenia TCP.

```
int recv_header(int fd, struct header* header) {
    ssize_t size = sizeof(*header)
    if (read(fd, header, size) != size) {
        return ERROR;
    }
    return OK;
}
```

Uwaga z labów — SOCK_STREAM a read

Spróbujmy stworzyć funkcję, która odbierze nagłówek
z ustanowionego wcześniej połączenia TCP.

```
int recv_header(int fd, struct header* header) {
    ssize_t size = sizeof(*header)
    if (read(fd, header, size) != size) {
        return ERROR;
    }
    return OK;
}
```

A poza tym też patrz tu:

http://www.gnu.org/software/libc/manual/html_node/Interrupted-Primitives.html

Uwaga z labów — SOCK_STREAM a read

```
int recv_header(int fd, struct header* header) {
    size_t to_read = sizeof(*header);
    char* buf = (char*)header;
    while (to_read > 0) {
        ssize_t bytes = read(fd, buf, to_read);
        if (bytes < 0) {
            if (errno == EINTR) continue;
            return ERROR;
        } else if (bytes == 0) {
            // zamknięto połączenie
            return ERROR;
        }
        to_read -= bytes;
        buf += bytes;
    }
    return OK;
}
```

Wiele połączeń

```
for (;;) {
    int conn = accept(listening_fd, NULL, NULL);
    sleep(10);
    write(conn, "Hello world!", 12);
    close(conn);
}
```

Co się dzieje z innymi połączeniami, które przychodzą podczas czekania?

Jak to łatwo sprawdzić?

Wiele połączeń — procesy

W tym modelu na każde połączenie tworzymy nowy proces (np. przy użyciu `fork`).

Zalety:

- łatwość obsługi połączenia

Wady:

- ograniczona liczba równoległych połączeń
- względnie duże zużycie pamięci na proces, nieopłacalne przy prostych protokołach, w których chcemy obsługiwać wiele równoległych połączeń

Pod Linuxem tworzenie procesu jest bardzo szybkie.

Wiele połączeń — wątki

W tym modelu na każde połączenie tworzymy nowy wątek (np. przy użyciu `pthread_create`).

Zalety:

- łatwość obsługi połączenia
- niewielkie zużycie pamięci

Wady:

- konieczność zapewnienia żywotności, użycia blokad itp.
- ograniczona liczba równoległych połączeń

Pod Linuxem wątki niespecjalnie różnią się (wydajnościowo) od procesów.

Wiele połączeń — komunikacja sterowana zdarzeniami

- Program jest jednowątkowy.
- W pętli oczekuje na przychodzące *zdarzenia*, którymi może być:
  - przyjście nowego połączenia na nasłuchującym gnieździe,
  - odebranie przez system operacyjny danych na którymś z otwartych gniazd,
  - możliwość wysłania kolejnych danych na którymś z otwartych gniazd (po wcześniejszym całkowitym wypełnieniu bufora nadawczego).
- W czasie obsługi zdarzenia program nie wykonuje operacji blokujących, oczekujących na zewnętrzne zdarzenia.

Gniazda nieblokujące

Domyślnie operacje na gniazdach są blokujące. Np. `read` oczekuje na to, aż jakieś dane będą dostępne lub połączenie zostanie zamknięte.

Należy zawsze pamiętać, że `read` nie musi czekać, aż wszystkie żądane dane (drugi parametr) będą dostępne; może zakończyć się wcześniej, zwracając choćby 1 bajt.

Gniazda nieblokujące

Włączenie trybu nieblokującego:

```
int on = 1;
ioctl(socket_fd, FIONBIO, &on);
```

Wystarczy to wykonać na gnieździe nasłuchującym, ponieważ gniazda połączeń dziedziczą powyższe ustawienie.

Odpowiedni fragment z `man 2 ioctl`:

```
FIONBIO int
  Set non-blocking I/O mode if the argument is
  non-zero. In non-blocking mode, read(2) or
  write(2) calls return -1 and set errno to EAGAIN
  immediately when no data is available.
```

Gniazda nieblokujące

```
for (;;) {
  ssize_t bytes = read(fd, buf, buf_size);
  if (bytes < 0) {
    if (errno == EAGAIN) {
      printf("Chwilowo nie ma więcej danych\n");
      break;
    } else {
      printf("Błąd: %s\n", strerror(errno));
      break;
    }
  } else if (bytes == 0) {
    printf("Zakończono połączenie\n");
    break;
  }

  printf("Odebrano %zd bajtów: %*s\n", bytes, bytes, buf);
}
```

Oczekiwanie na zdarzenia

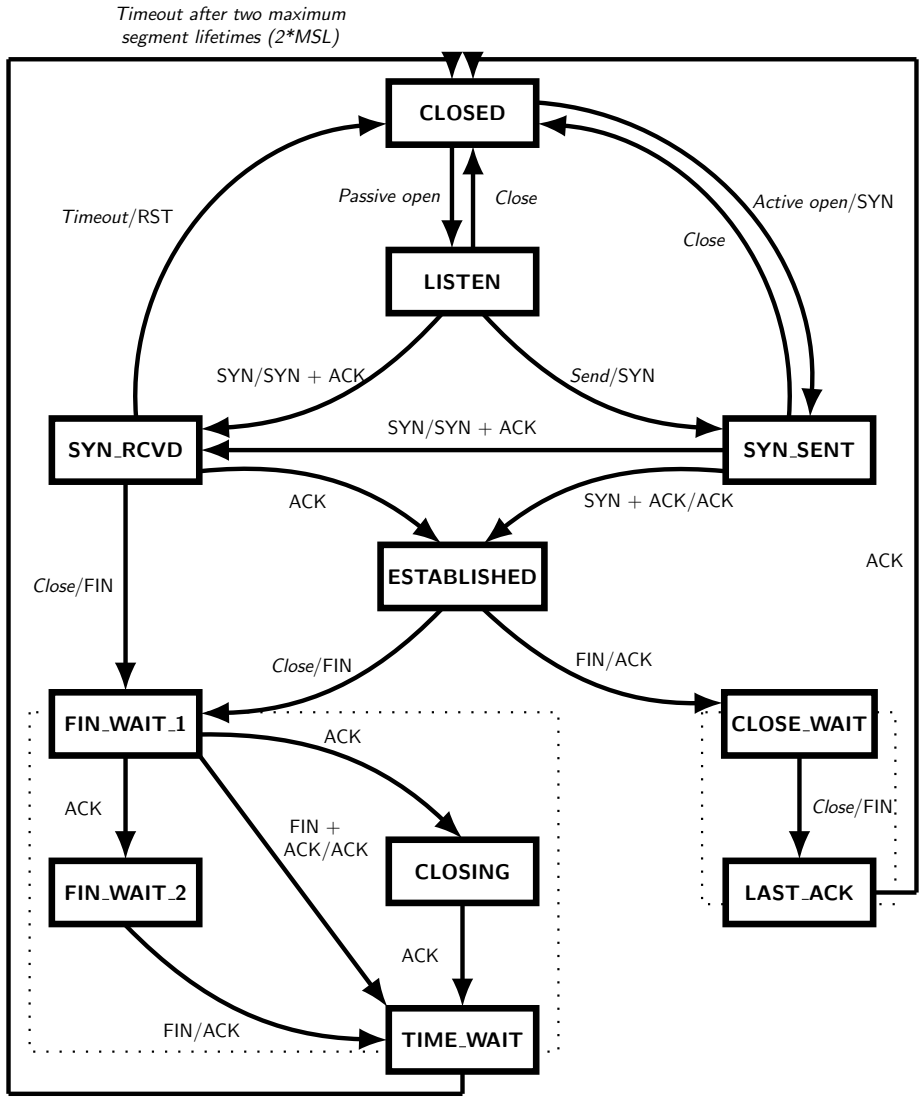
Demo. Przykład: `poll(2)`

O tym będzie też na labach.

Dla nieobecnych:
`http://publib.boulder.ibm.com/infocenter/iseries/v6r1m0/topic/rzab6/xnonblock.htm`,
`http://publib.boulder.ibm.com/infocenter/iseries/v6r1m0/topic/rzab6/poll.htm`

Projektowanie serwerów asynchronicznych

Dawno temu wiadomo było, że automat stanowy to dobry wzorzec.

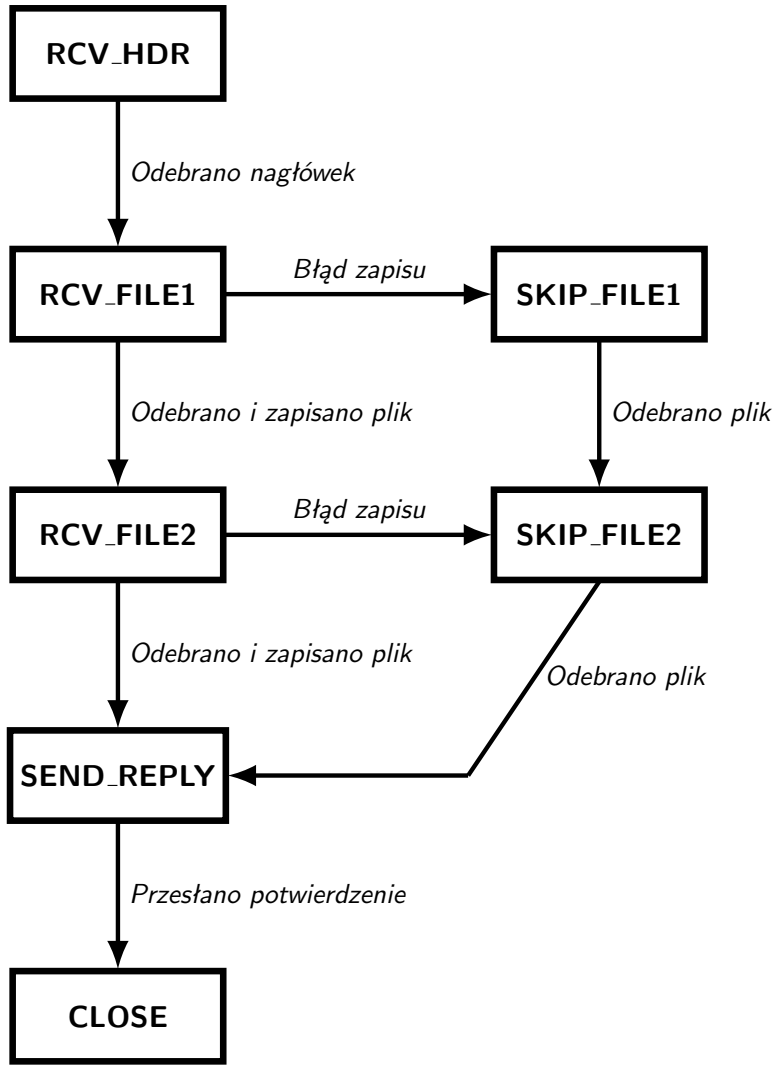


Projektowanie serwerów asynchronicznych

Wyobraźmy sobie protokół, w którym klient przesyła do serwera nagłówek oraz 2 pliki. W nagłówku zawarte są rozmiary tych plików. Serwer zapisuje te pliki na dysk (nie trzyma ich w pamięci), a następnie wysyła potwierdzenie.

Projektowanie serwerów asynchronicznych

Wyobraźmy sobie protokół, w którym klient przesyła do serwera nagłówek oraz 2 pliki. W nagłówku zawarte są rozmiary tych plików. Serwer zapisuje te pliki na dysk (nie trzyma ich w pamięci), a następnie wysyła potwierdzenie.



Projektowanie serwerów asynchronicznych

Tak czy inaczej zazwyczaj tworzy się odpowiednią strukturę przechowującą kontekst połączenia.

```
struct connection_context {
  enum connection_state state;
  int sock;
  size_t file1_bytes_remaining;
  size_t file2_bytes_remaining;
};
```

Komunikacja sterowana zdarzeniami — c.d.

- Bardziej podstawowe operacje też mogą być sterowane zdarzeniami, np. operacje dyskowe. Przecież pod obciążeniem mogą trwać bardzo długo i niepotrzebnie blokować obsługę innych zdarzeń.
- Różne mechanizmy obsługi zdarzeń istnieją w różnych systemach. Nawet pod samym Linuxem mamy wybór pomiędzy `select`, `poll`, `epoll`... Dlatego istnieją biblioteki oferujące ujednolicony interfejs obsługi zdarzeń:
  - libevent w C (będzie omawiana na laboratoriach)
  - `boost::asio` w C++

Komunikacja sterowana zdarzeniami — c.d.

- W aplikacjach interaktywnych obsługa interfejsu użytkownika też jest sterowana zdarzeniami, a odpowiednia biblioteka UI udostępnia ujednolicony interfejs do obsługi zarówno zdarzeń interkacji z użytkownikiem, jak i zdarzeń sieciowych czy innych.
- Do komunikacji pomiędzy wątkami czy procesami często też używa się komunikacji po gniazdach (niekoniecznie sieciowych, ale np. tzw. UNIX sockets), które także mogą być obsługiwane asynchronicznie.

Komunikacja sterowana zdarzeniami — c.d.

Komunikacja sterowana zdarzeniami
nie jest jedynym słusznym rozwiązaniem.