Wykład #7

SELECT(), POLL(), EPOLL

Zwielokrotnienie wejścia-wyjścia

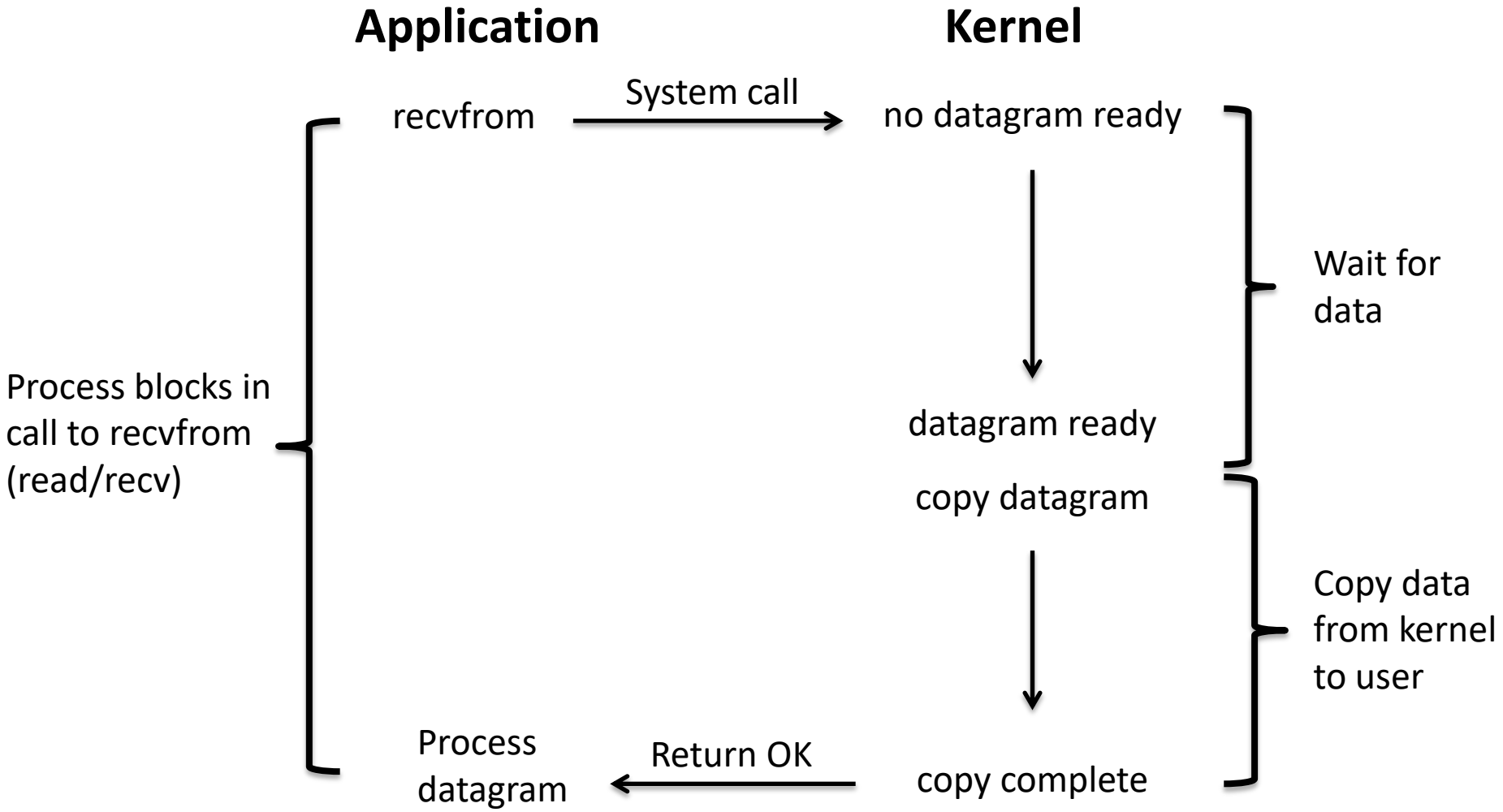
Zwielokrotnianie wejścia-wyjścia funkcje select(), poll(), EPOLL – przypadki użycia

- Proces klienta obsługuje więcej niż jeden deskryptor
- Serwer - ten sam proces obsługujący gniazdo nasłuchujące i gniazda połączone
- Obsługa jednocześnie protokołów TCP, UDP i SCTP
- Serwer obsługujący więcej niż jedną usługę, być może używając przy tym kilku protokołów
- Używane nie tylko w programach sieciowych

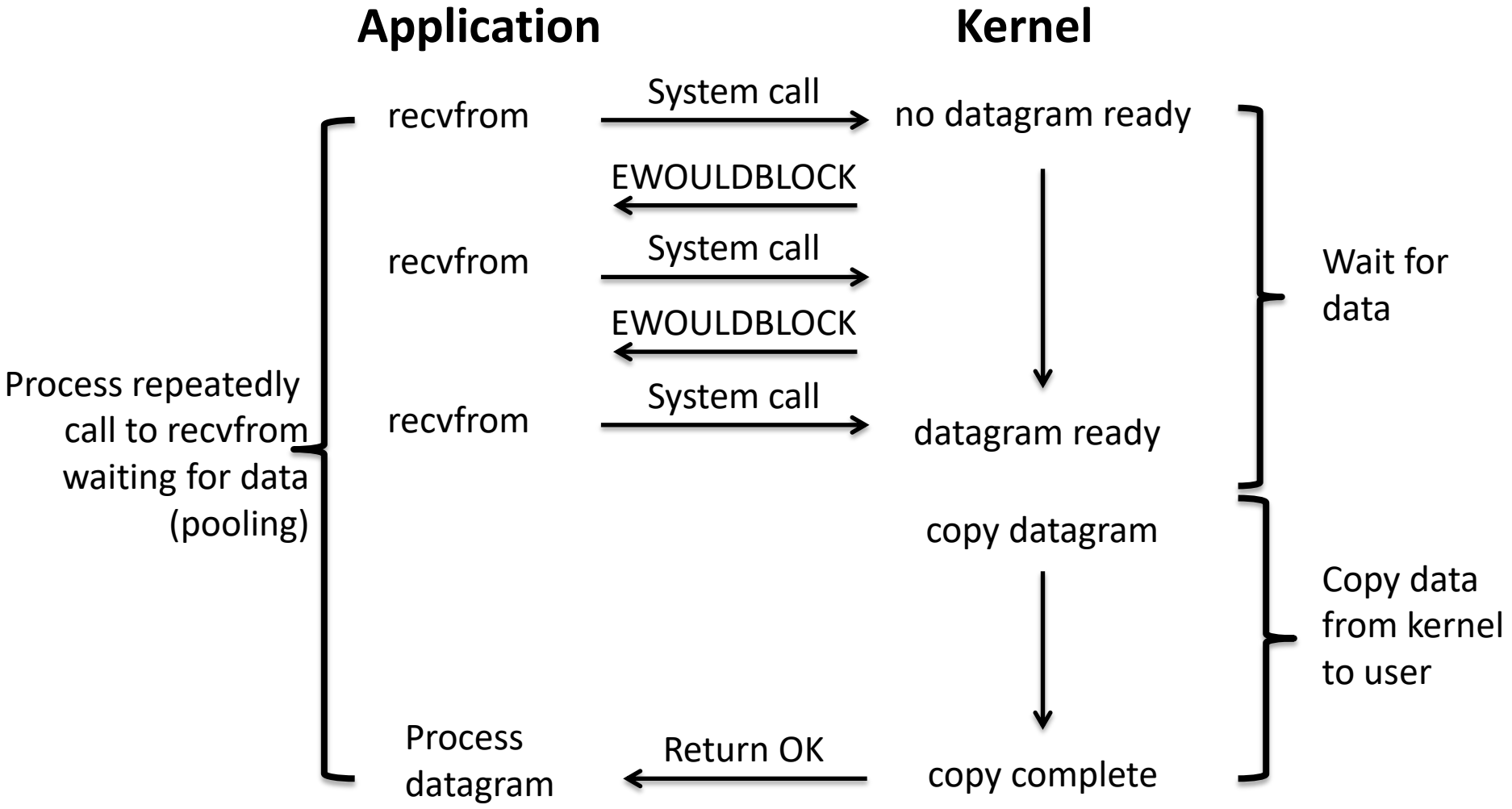
Modele wejścia-wyjścia

- Wejścia-wyjścia blokującego
- Wejścia-wyjścia nieblokującego
- Wejścia-wyjścia zwielokrotnionego
- Wejścia-wyjścia sterowanego sygnałami
- Wejścia-wyjścia asynchronicznego

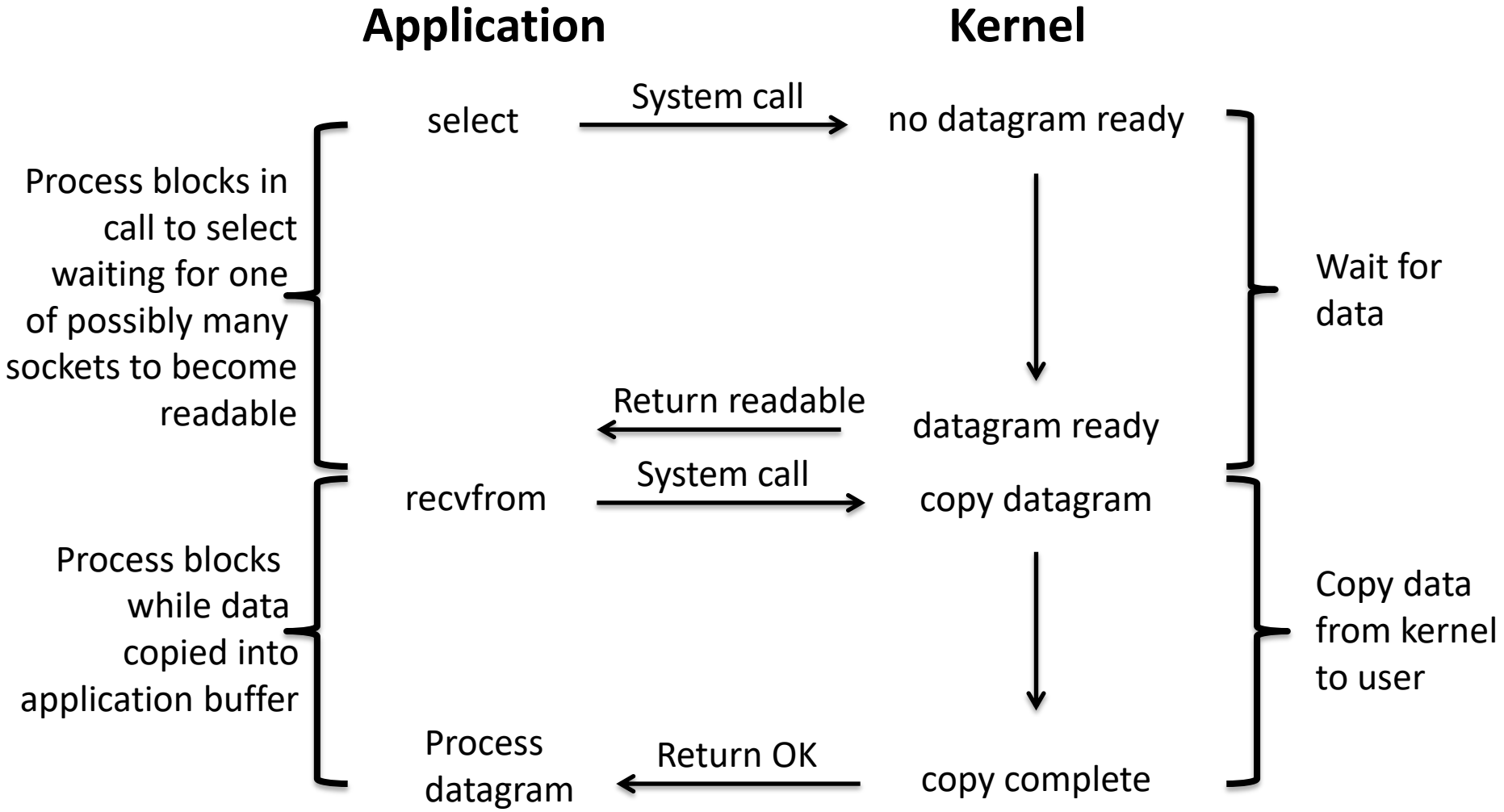
Model wejścia-wyjścia blokującego



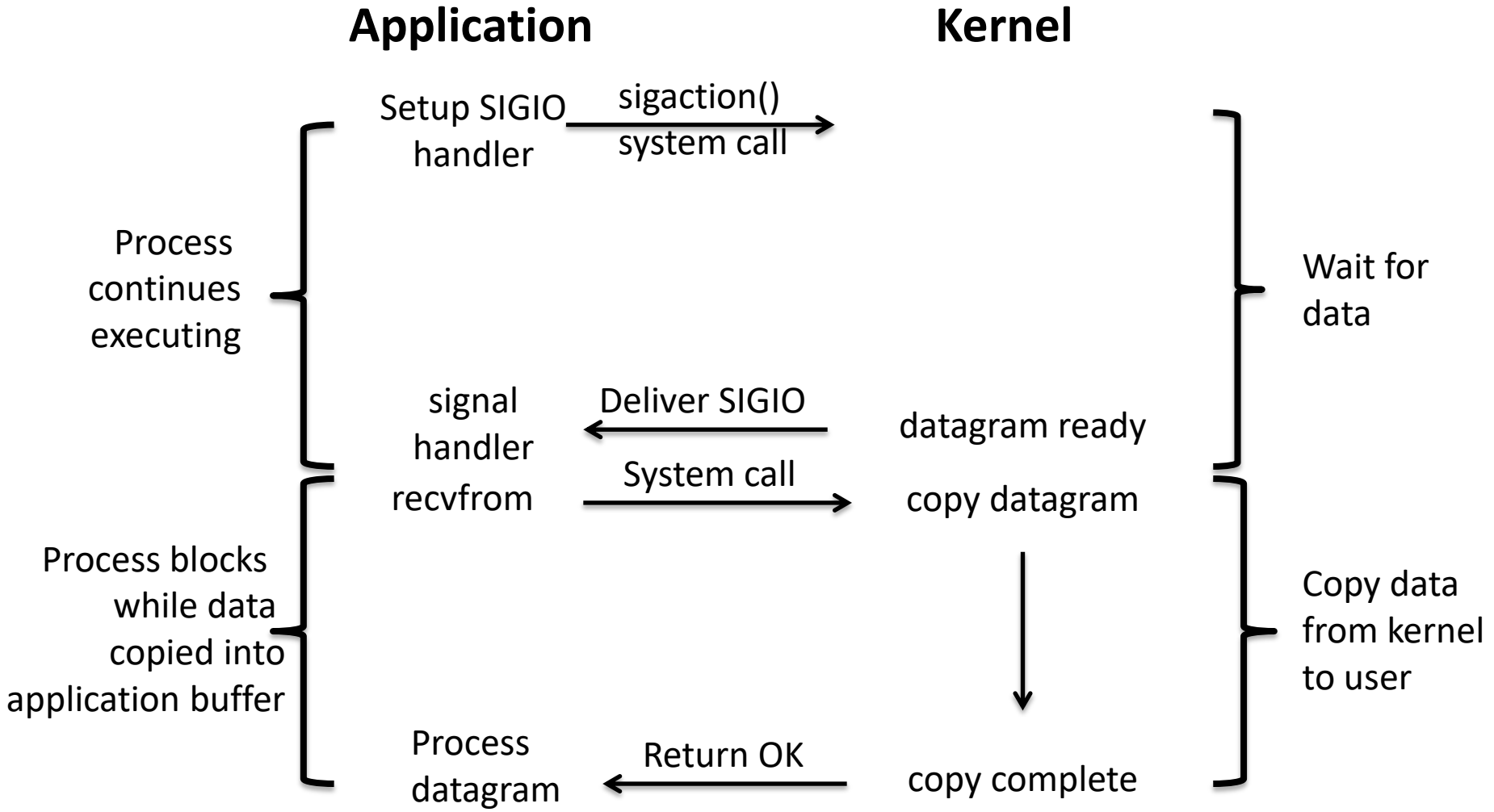
Model wejścia-wyjścia nieblokującego



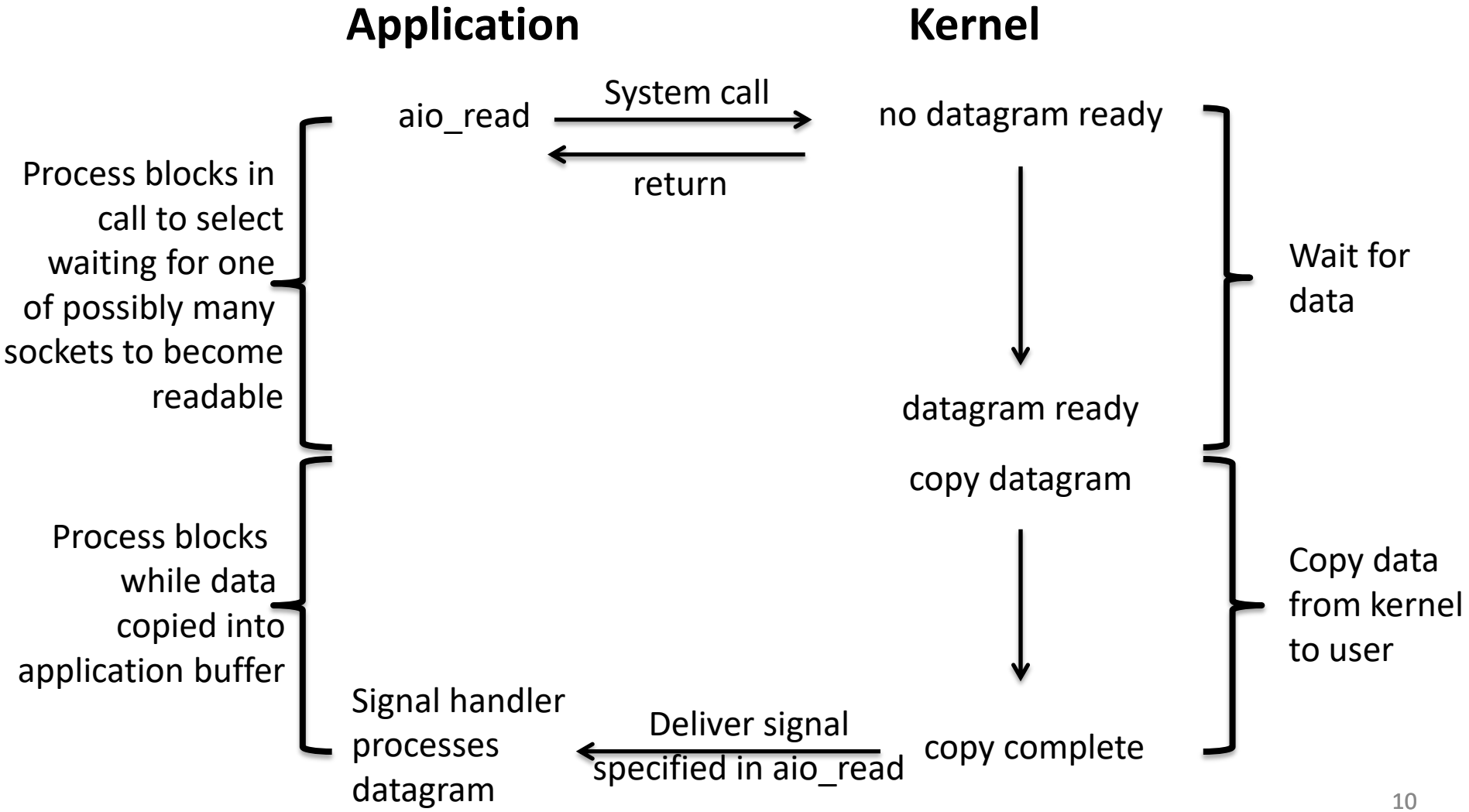
Model wejścia-wyjścia **zwielokrotnionego** (*multiplexing mode*)



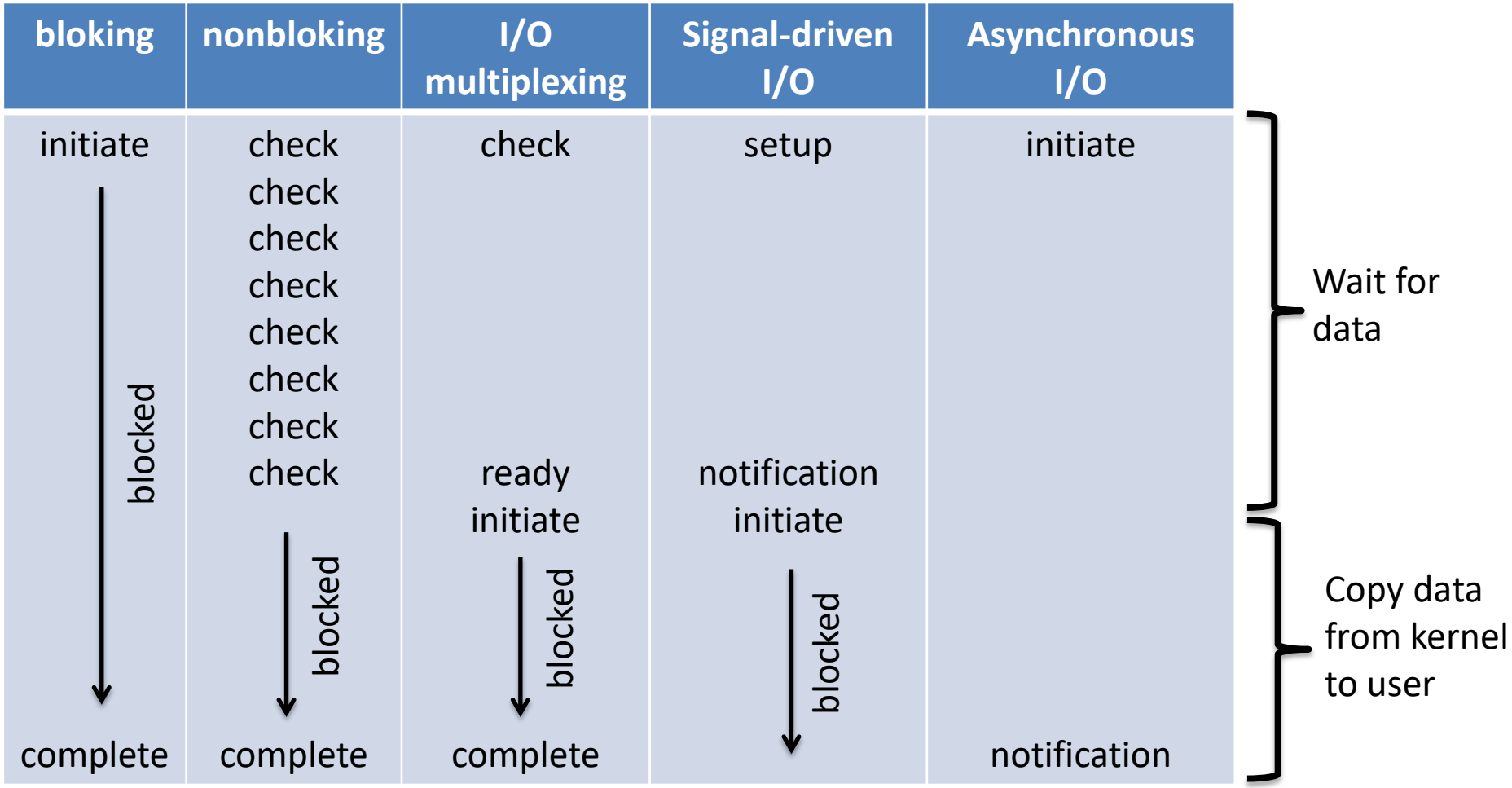
Model wejścia-wyjścia sterowanego sygnałami



Model wejścia-wyjścia asynchronicznego



Porównanie modeli wejścia-wyjścia



Model wejścia-wyjścia zwielokrotnionego - funkcja **SELECT()**

- Zlecenie dla jądra systemu na oczekiwanie na którekolwiek z pewnej liczby zdarzeń, np.:
  - Dowolny deskryptor ze zbioru jest gotowy do czytania
  - Dowolny deskryptor ze zbioru jest gotowy do pisania
  - Dla dowolnego deskryptora z pewnego zbioru istnieje nieobsłużona sytuacja wyjątkowa
  - Upłynęło więcej czasu niż ustalony limit
- Jądro budzi proces jeśli wystąpiło jedno z powyższych zdarzeń

Funkcja SELECT()

int select(int **nfds**, fd_set ***readfds,** fd_set ***writefds,** fd_set ***exceptfds**, struct timeval *timeout);

- readfds, writefds, exceptfds – określają zbiory deskryptorów, dla których jądro ma sprawdzać odpowiednio gotowość do czytania, pisania oraz występowania sytuacji wyjątkowych
- nfds – największy numer deskryptora użytego w zbiorach plus 1. (dla przykładu poniżej 6)
- Operacje na zbiorach deskryptorów:

```
fd_set   rset;
FD_ZERO(&rset); /* initialize the set: all bits off */
FD_SET(1, &rset); /* turn on bit for fd 1 */
FD_CLR(4, &rset); /* turn off bit for fd 4 */
FD_ISSET(5, &rset); /* check bit for fd 5 */
```

Funkcja SELECT()

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

- Gniazdo jest gotowe do czytania jeśli:
  - Liczba bajtów w buforze odbiorczym gniazda jest większa lub równa 1 – funkcja read() zwróci liczbę przeczytanychbajtów
  - ~~Liczba bajtów w buforze odbiorczym gniazda jest większa lub równa **SO_RCVLOWAT** (domyślnie 1) – funkcja read() zwróci liczbę bajtów skopiowanych~~
  - Została zamknięta część czytająca połączenia (funkcja read() zwróci 0)
  - Operacja dotyczy gniazda nasłuchującego, a liczba nawiązanych połączeń jest większa od zera (można wywołać funkcję accept() dla gniazda)
  - Istnieje nieobsłużony błąd dotyczący gniazda (funkcja read() zwróci -1) - – błąd można obsłużyć funkcją getsockopt() z opcją **SO_ERROR**

Funkcja SELECT()

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

- Gniazdo jest gotowe do pisania jeśli:
  - Bufor nadawczy gniazda nie jest pełny – funkcja write() zwróci liczbę zapisanych bajtów
  - ~~Liczba bajtów w buforze nadawczym jest mniejsza lub równa **SO_SNDLOWAT** (domyślnie 2048) – funkcja write() zwróci liczbę zapisanych bajtów~~
  - Została zamknięta część pisząca połączenia (funkcja write() wygeneruje sygnał SIGPIPE)
  - Istnieje nieobsłużony błąd dotyczący gniazda (funkcja write() zwróci -1) – błąd można obsłużyć funkcją getsockopt() z opcją SO_ERROR

Funkcja SELECT()

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

- Sytuacja wyjątkowa wystąpi jeśli:
  - Istnieją dane pozapasmowe dla tego gniazda
- Funkcja zwraca liczbę gotowych deskryptorów, 0 jeśli został przekroczony czas oczekiwania, -1 w przypadku błędu
- Podsumowanie:

Condition	Readable?	Writable?	Exeption?
Data to read Read half of the connection closed New connection ready for listening socket	• • •		
Space avaliable for writing Write half of the connection closed		• •	
Pending error	•	•	
TCP out-of-band data			•

Funkcja SELECT()

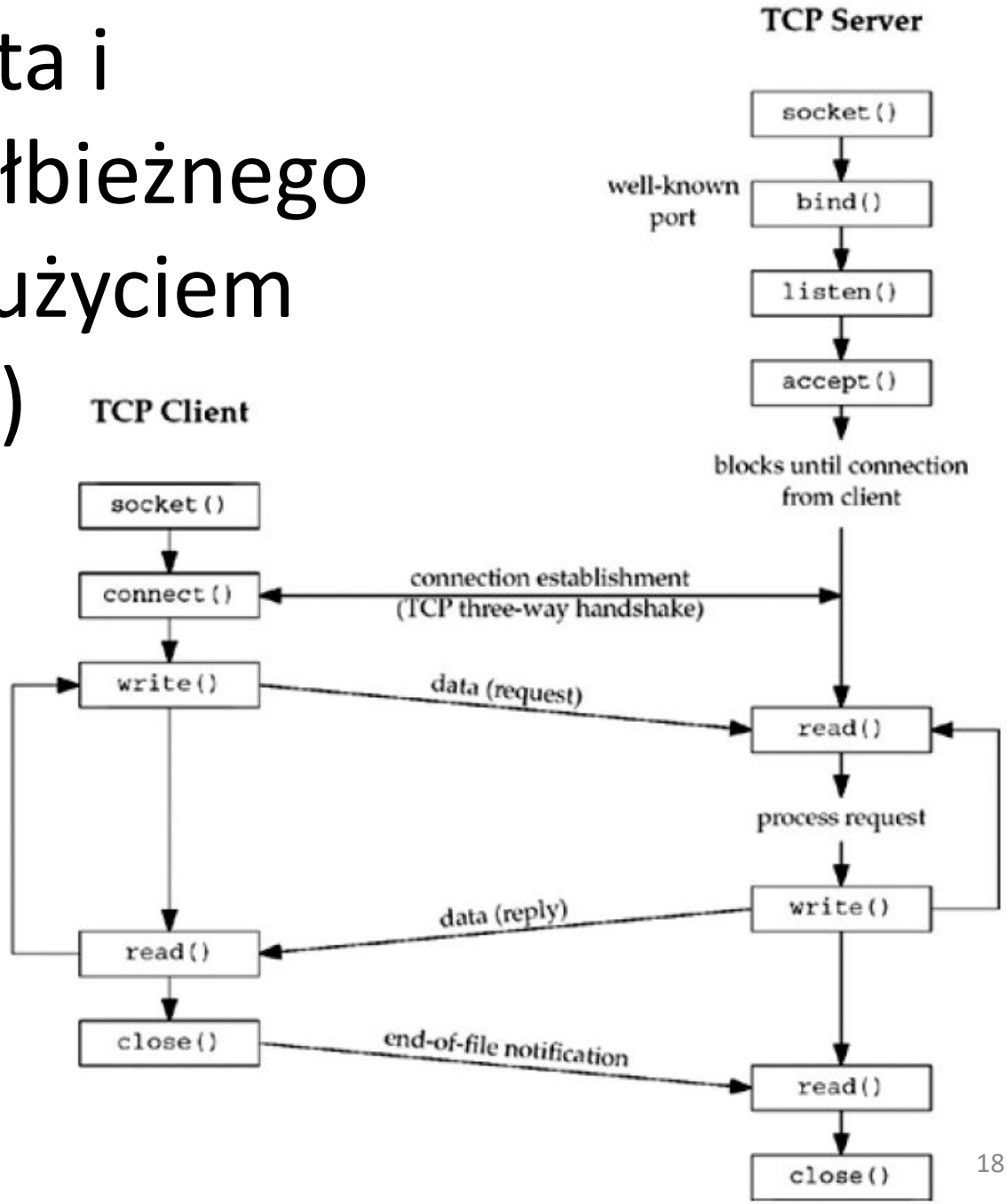
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, **struct timeval *timeout**);

- struct **timeval** *timeout – pusty: czas nieskończony, zerowy: nieblokujący, określony czas; dla systemu Linux struktura przechowuje czas, po którym system przekazał sterowanie

```
<sys/time.h>

struct timeval {
    long    tv_sec;         /* seconds */
    long    tv_usec;        /* microseconds */
};
```

Przykład klienta i serwera współbieżnego usługi **echo** z użyciem funkcji **select**()



Przykład klienta z wykorzystaniem funkcji select() – 1/3

```
1.    ….  socket();, … connect(); ….
2.   void
3.   str_cli(FILE *fp, int sockfd)
4.   {
5.        int                maxfdp1, stdineof;
6.        fd_set             rset;
7.        char               buf[MAXLINE];
8.        int                n;

9.        stdineof = 0;   /* flaga do wykrycia końca pliku */
10.       FD_ZERO(&rset);
11.       for ( ; ; ) {
12.                 if (stdineof == 0)
13.                           FD_SET(fileno(fp), &rset);
14.                 FD_SET(sockfd, &rset);
15.                 maxfdp1 = fmax(fileno(fp), sockfd) + 1;
```

Przykład klienta z wykorzystaniem funkcji select() – 2/3

```
16.     if ( (n = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0){
17.                     perror("select error");
18.                     exit(1);
19.     }
20.     if (FD_ISSET(sockfd, &rset)) {    /* socket is readable */
21.             if ( (n = read(sockfd, buf, MAXLINE)) <= 0 ) {
22.                     if (stdineof == 1)
23.                             return;     /* normal termination */
24.                     else{
25.                             perror("str_cli: server terminated prematurely");
26.                             exit(1);
27.                     }
28.             }
29.             if ( write(fileno(stdout), buf, n) != n){
30.                     perror("write error");
31.                     exit(1);
32.     }}
```

Przykład klienta z wykorzystaniem funkcji select() – 3/3

```
33.                 if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
34.                         if ( (n = read(fileno(fp), buf, MAXLINE)) == 0) {
35.                                 stdineof = 1;
36.                                 if (shutdown(sockfd, SHUT_WR) < 0){
37.                                         perror("shutdown error");
38.                                         exit(1);
                                    }
39.                                 FD_CLR(fileno(fp), &rset);
40.                                 continue;
41.                         }
42.                         Writen(sockfd, buf, n);
43.                 }
44.         }
45. }
```

Przykład serwera z wykorzystaniem funkcji select() – 1/5

```
1.  int main(int argc, char **argv){
2.          int       listenfd, connfd, sockfd;
3.          socklen_t         clilen;
4.          struct sockaddr_in6     cliaddr, servaddr;
5.          int       i,   maxi,   maxfd, n;
6.          int       nready,   client[FD_SETSIZE];
7.          fd_set    rset, allset;
8.          char      buf[MAXLINE];
9.          char      addr_buf[INET6_ADDRSTRLEN+1];
10.
11.         ……..
```

Przykład serwera z wykorzystaniem funkcji select() – 2/5

```
1.   ……   socket(), bind(), listen()
2.        maxfd = listenfd;                    /* initialize */
3.        maxi = -1;                           /* index into client[] array */
4.        for (i = 0; i < FD_SETSIZE; i++)
5.                 client[i] = -1;             /* -1 indicates available entry */
6.        FD_ZERO(&allset);
7.        FD_SET(listenfd, &allset);
8.
9.       for ( ; ; ) {
10.                rset = allset;              /* structure assignment */
11.                if ( (nready = select(maxfd+1, &rset, NULL, NULL, NULL)) < 0){
12.                                perror("select error");
13.                                exit(1);
14.                }
```

Przykład serwera z wykorzystaniem funkcji select() (3/5)

```
1.      if (FD_ISSET(listenfd, &rset)) {/* new client connection */
2.              clilen = sizeof(cliaddr);
3.              if ( (connfd = accept(listenfd, (SA *) &cliaddr,
4.                                                      &clilen)) < 0){
5.                      perror("accept error");
6.                      exit(1);
7.              }

8.              printf("new client: %s, port %d\n",
9.                      inet_ntop(AF_INET6,  &cliaddr.sin6_addr,
                        addr_buf, 16),  ntohs(cliaddr.sin6_port));
```

Przykład serwera z wykorzystaniem funkcji select() (4/5)

```
1.          for (i = 0; i < FD_SETSIZE; i++)
2.                  if (client[i] < 0) {
3.                          client[i] = connfd;    /* save descriptor */
4.                          break;
5.                  }
6.          if (i == FD_SETSIZE){
7.                  fprintf(stderr,"too many clients");
8.                  continue;
9.          }
10.         FD_SET(connfd, &allset);        /* add new descriptor to set */
11.         if (connfd > maxfd)
12.                 maxfd = connfd;                 /* for select */
13.         if (i > maxi)
14.                 maxi = i;                       /* max index in client[] array */

15.         if (--nready <= 0)
16.                 continue;                       /* no more readable descriptors */
17.   }   //end if (FD_ISSET(listenfd, &rset))
```

Przykład serwera z wykorzystaniem funkcji select() (5/5)

```
1.          for (i = 0; i <= maxi; i++) {   /* check all clients for data */
2.                  if ( (sockfd = client[i]) < 0)
3.                          continue;
4.                  if (FD_ISSET(sockfd, &rset)) {
5.                          if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
6.                                  /*connection closed by client */
7.                                  close(sockfd);
8.                                  FD_CLR(sockfd, &allset);
9.                                  client[i] = -1;
10.                         } else
11.                                 Writen(sockfd, buf, n);
12.                         if (--nready <= 0)
13.                                 break;    /* no more readable descriptors */
14.                 }
15.  }  }  }
```

Funkcja PSELECT()

int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout, const sigset_t ***sigmask**);

- Korzysta ze struktury timespec (select() korzysta z timeval) - czas można podawać z dokładnością do nanosekund
- Nowy argument **sigmask** – na czas wywołania pselect() możemy zmieniać maskę sygnałów (blokować/odblokowywać lub ustawiać procedury obsługi sygnałów)

Funkcja poll()

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds,
        int timeout);
```

- Oczekuje na zdarzenia na zbiorze deskryptorów
- Podobne zastosowanie do funkcji select(), ale dostarcza dodatkowe informacje związane z obsługą urządzeń strumieniowych (stosuje się nie tylko do gniazd)
- Jako pierwszy argument podaje się wskaźnik do tablicy deskryptorów, które są typu struktury **pollfd**:

```
1. struct pollfd {
2.         int   fd;       /* file descriptor, if -1 ignored*/
3.         short events;    /* requested events */
4.         short revents;   /* returned events */
5.  };
```

Funkcja poll()

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds,
        int timeout);
```

- **ndfs** – liczba deskryptorów plików przekazanych w pierwszym argumencie
- **timeout** - czas oczekiwania w mili sekundach, jeśli ujemny funkcja czeka w nieskończoność na zdarzenia, 0 powraca natychmiast (nieblokująca)
- Funkcja zwraca liczbę deskryptorów dla których wystąpiło zadane zdarzenie

Funkcja poll()

#include <**poll.h**>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

Constant	Input to events	Result from events	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occured
POLLHUP		•	Hangup has occured
POLLNVAL		•	Descriptor is not an open file
POLLRDHUP		•	Stream socket peer closed connection

Funkcja ppoll()

- `int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec *timeout_ts, const sigset_t *sigmask);`

- Funkcja analogiczna do funkcji poll(), ale umożliwia zmianę maski sygnałów w trakcie wywołania funkcji
- Stosuje strukturę timespec, w której czas oczekiwania można podać z dokładnością do nanosekund

Funkcja poll – przykład serwera

```
1.  struct pollfd client[FOPEN_MAX]
2.  socket(), bind(), listen() ….
3.  client[0].fd = listenfd;
4.  client[0].events = POLLRDNORM;
5.  for (i = 1; i < FOPEN_MAX; i++)
6.      client[i].fd = -1;          /* -1 indicates available entry */
7.  maxi = 0;                       /* max index into client[] array */
8.  for ( ; ; ) {
9.      if ( (nready = poll(client, maxi+1, INFTIM)) < 0){
10.                 perror("poll error");
11.                 exit(1);
12.     }
```

Funkcja poll() – przykład serwera

```
1.  if (client[0].revents & POLLRDNORM) {  /* new client connection */
2.          clilen = sizeof(cliaddr);
3.          if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
4.                  perror("accept error");
5.                  exit(1);
6.          }
7.           bzero(addr_buf, sizeof(addr_buf));
8.           inet_ntop(AF_INET6, (struct sockaddr  *) &cliaddr.sin6_addr,
                    addr_buf, sizeof(addr_buf));
9.          printf("new client: %s, port %d\n",          addr_buf,
                    ntohs(cliaddr.sin6_port));
```

Funkcja poll – przykład serwera

```
1.          for (i = 1; i < FOPEN_MAX; i++)
2.                  if (client[i].fd < 0) {
3.                          client[i].fd = connfd;              /* save descriptor */
4.                          break;
5.                   }
6.          if (i == FOPEN_MAX){
7.                          perror("too many clients");
8.                          continue;
9.          }
10.         client[i].events = POLLRDNORM;
11.         if (i > maxi)
12.                         maxi = i;     /* max index in client[] array */

13.         if (--nready <= 0)
14.                         continue; /* no more readable descriptors */
15.   }//end of listenfd service
```

Funkcja poll – przykład serwera

```
1.  for (i = 1; i <= maxi; i++) {        /* check all clients for data */
2.      if ( (sockfd = client[i].fd) < 0)
3.          continue;
4.      if (client[i].revents & (POLLRDNORM | POLLERR)) {
5.          if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
6.              if (errno == ECONNRESET) { /*connection reset by client */
7.                  printf("client[%d] aborted connection\n", i);
8.                  close(sockfd);
9.                  client[i].fd = -1;
10.             } else{
11.                 perror("read error");
12.                 exit(1);
13.             }
```

Funkcja poll – przykład serwera

```
1.          } else if (n == 0) {   /*connection closed by client */
2.                      printf("client[%d] closed connection\n", i);
3.                      close(sockfd);
4.                      client[i].fd = -1;
5.          } else
6.                      Writen(sockfd, buf, n);

7.          if (--nready <= 0)
8.                      break;                  /* no more readable descriptors */
9.        } //for dla klientów
10.     } //for(;;)
11.   }//
```

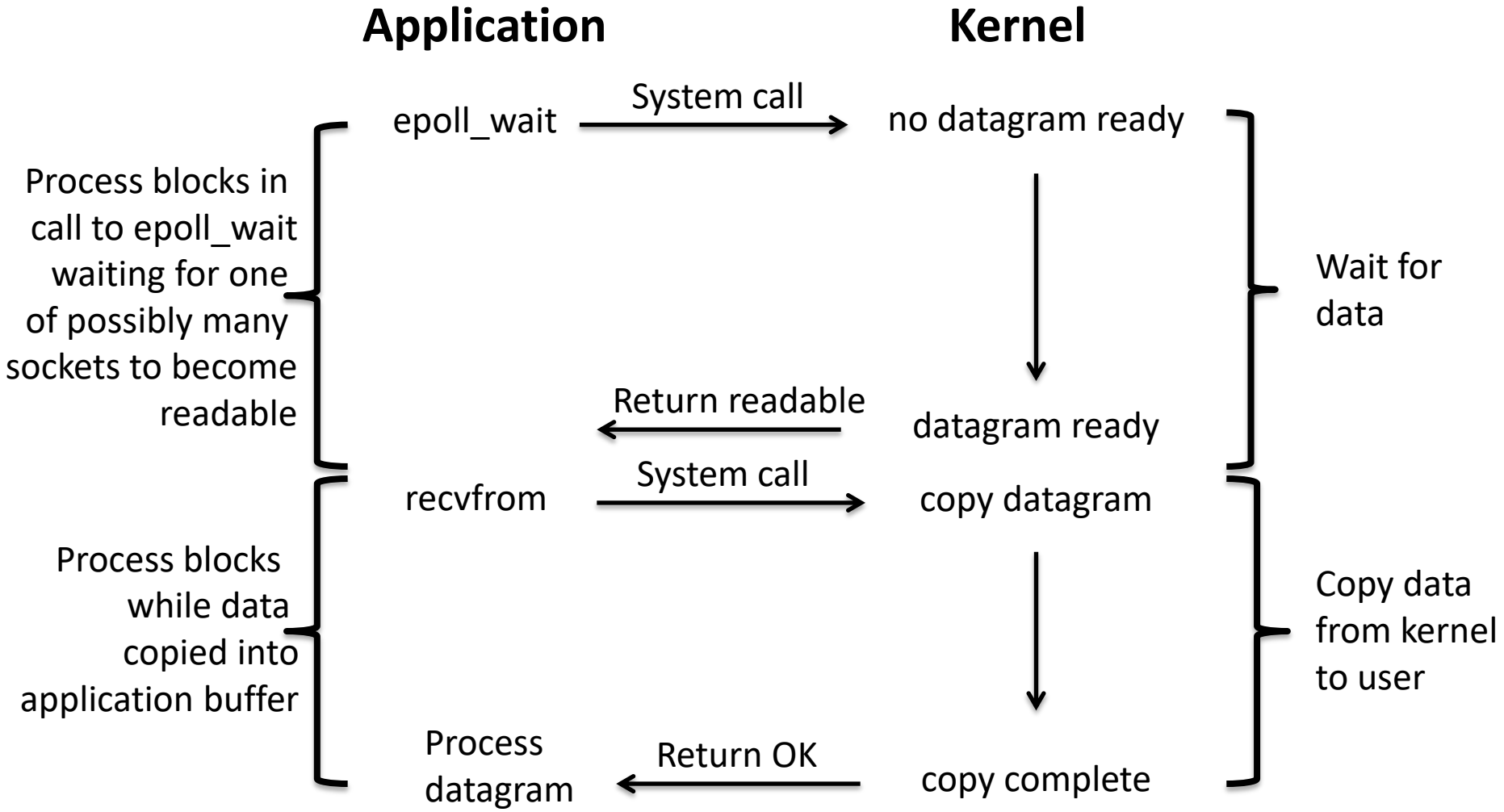
Wejście-wyjście zwielokrotnione – podsumowanie select() i poll()

- W systemach UNIX istnieje pięć różnych modeli wejścia-wyjścia (blokujące, nieblokujące, zwielokrotnione, sterowane sygnałami, asynchroniczne).
- W modelu wejścia-wyjścia zwielokrotnionego najczęściej używało się funkcji select().
- W funkcji select() powiadamiamy jądro o zdarzeniach, na które chcemy oczekiwać, dla interesujących nas deskryptorów (zdarzenia dotyczące czytania, pisania i obsługi sytuacji wyjątkowych), maksymalnym czasie oczekiwania oraz maksymalnej wartości w zbiorze deskryptorów powiększonej o 1.
- W przypadku powodzenia funkcja select() zwraca informację o liczbie deskryptorów i które deskryptory należy obsłużyć.
- Czas oczekiwania określa, czy funkcja jest blokująca, czy nie.
- Funkcja pselect() dodatkowo umożliwia ustawianie maski sygnałów.
- Funkcje poll() i ppoll() spełniają podobną funkcję do select() i pselect(), z tym że przekazują dodatkowe informacje dotyczące urządzeń strumieniowych.

EPOLL – LINUX solution

- Similar to poll() but faster
- Performance optimisation solution – **keeps file descriptors table in the kernel**

EPOLL - model wejścia-wyjścia zwielokrotnionego (*multiplexing mode*)



EPOLL API

- **epoll_create()** – creates an epoll instance and returns a file descriptor referring to that instance. The more recent **epoll_create1** () extends the functionality of epoll_create().
- **epoll_ctl()** – registration of particular file descriptors to be watched. The set of file descriptors currently registered on an epoll instance is sometimes called an epoll set.
- **epoll_wait()** – waits for I/O events, blocking the calling thread if no events are currently available.

EPOLL API – epoll_create()

int epoll_create(int size);

int epoll_create1(int flags);

- size – hint to kernel about size of epoll descriptor set (**currently not used**)
- flags: CLOEXEC
- epoll_create() creates an epoll "instance", requesting the kernel to allocate an event backing store. epoll_create() returns a file descriptor referring to the new epoll instance. This file descriptor is used for all the subsequent calls to the epoll interface. When no longer required, the file descriptor returned by epoll_create() should be closed by using close().

EPOLL API – epoll_ctl()

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

- This system call performs control operations on the epoll instance referred to by the file descriptor epfd. It requests that the operation op be performed for the target file descriptor, fd.
- Valid values for the op argument are :
  - **EPOLL_CTL_ADD** - Register the target file descriptor fd on the epoll instance referred to by the file descriptor epfd and associate the event with the internal file linked to fd.
  - **EPOLL_CTL_MOD** - Change the event associated with the target file descriptor fd.
  - **EPOLL_CTL_DEL** - Remove (deregister) the target file descriptor fd from the epoll instance referred to by epfd.

struct epoll_event

```
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 */
    };
```

EPOLL API – epol_ctl() – Events (1/2)

- **EPOLLIN** - The associated file is available for read() operations.
- **EPOLLOUT** - The associated file is available for write() operations.
- **EPOLLRDHUP** - Stream socket peer closed connection, or shut down writing half of connection. (This flag is especially useful for writing simple code to detect peer shutdown when using Edge Triggered monitoring.)
- **EPOLLPRI** - There is urgent data available for read() operations.
- **EPOLLERR** - Error condition happened on the associated file descriptor. epoll_wait(2) will **always wait for this event**; it is not necessary to set it in events.

EPOLL API – epoll_ctl() – Events (2/2)

- **EPOLLHUP** - Hang up happened on the associated file descriptor. epoll_wait(2) will **always wait for this event**.
- **EPOLLET** - Sets the Edge Triggered behavior for the associated file descriptor. The default behavior for epoll is Level Triggered.
- **EPOLLONESHOT** - Sets the one-shot behavior for the associated file descriptor. This means that after an event is pulled out with epoll_wait() the associated file descriptor is internally disabled and no other events will be reported by the epoll interface. The user must call epoll_ctl() with **EPOLL_CTL_MOD** to rearm the file descriptor with a new event mask.

EPOLL API – epoll_wait()

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

int **epoll_pwait**(int epfd, struct epoll_event *events, int maxevents, int timeout, const sigset_t *sigmask);

- The epoll_wait() system call waits for events on the epoll() instance referred to by the file descriptor epfd. The memory area pointed to by events will contain the events that will be available for the caller.
- Up to maxevents are returned by epoll_wait(). The maxevents argument must be greater than zero.

EPOLL API – epoll_wait()

- The timeout argument specifies the minimum number of milliseconds that **epoll_wait()** will block. Specifying a timeout of -1 causes **epoll_wait()** to block indefinitely, while specifying a timeout equal to zero cause **epoll_wait()** to return immediately, even if no events are available.

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

- The data of each returned structure will contain the same data the user set with an epoll_ctl() (**EPOLL_CTL_ADD**, **EPOLL_CTL_MOD**) while the events member will contain the returned event bit field.

Przykład

- Serwer współbieżny usługi echo z mechanizmem EPOLL

echo_serv6_ws_epool.c

Ograniczenia w tworzeniu gniazd w systemie LINUX

- EPOLL - ograniczenie na maksymalną liczbę obserwowanych deskryptorów
- Limity na maksymalną liczbę plików w systemie i maksymalną liczbę otwartych deskryptorów w pojedynczym procesie
- Limit pamięci operacyjnej
- Limit szybkości tworzenia nowych gniazd

EPOLL /proc interfaces

- The following interfaces can be used to limit the amount of kernel memory consumed by epoll:

  /proc/sys/fs/epoll/max_user_watches (since Linux 2.6.28)

- This specifies a limit on the total number of file descriptors that a user can register across all epoll instances on the system. The limit is per real user ID. Each registered file descriptor costs roughly 90 bytes on a 32-bit kernel, and roughly 160 bytes on a 64-bit kernel. Currently, the default value for max_user_watches is 1/25 (4%) of the available low memory, divided by the registration cost in bytes.

Linux – Limits on open files descriptors number

- ulimit -a (show all limits in system, /etc/security/limits.conf)
- setting max open files for process:

  # ulimit -n 99999
- sysctl max files for system:

  # sysctl -w fs.file-max=100000
- pamięć – dla gniazd nie może być stronicowana
- /proc/sys/net/nf_conntrack_max (find /proc -name '*conntrack_max*‘)
- /proc/net/ipv4/ip_local_port_range

Number of outbound sockets a host can create from a particular IP address

- The ephermal port range defines the maximum number of outbound sockets a host can create from a particular IP address. The fin_timeout defines the minimum time these sockets will stay in TIME_WAIT state (unusable after being used once). Usual system defaults are:
  - net.ipv4.ip_local_port_range = 32768 61000
  - net.ipv4.tcp_fin_timeout = 60
- In above case a system cannot guarantee more than (61000 - 32768) / 60 = 470 new sockets at any given time.
- The above should not be interpreted as the factors impacting system capability for making outbound connections/second. But rather these factors affect system's ability to handle concurrent connections in a sustainable manner for large periods of "activity."

EPOLL - Edge-triggered mode

- The epoll event distribution interface is able to behave both as **edge-triggered** (ET) and as **level-triggered** (LT).
- **Edge-triggered** mode (epoll interface using the **EPOLLET** (edge-triggered) flag) delivers events only when changes occur on the monitored file descriptor.
- An application that employs the **EPOLLET** flag should use nonblocking file descriptors to avoid having a blocking read or write starve a task that is handling multiple file descriptors. The suggested way to use epoll as an edge-triggered (EPOLLET) interface is as follows:
  - with non-blocking file descriptors;
  - by waiting for an event only after read() or write() return EAGAIN.

EPOLL - Level-triggered mode

- By contrast to edge-triggered, when epoll is used as a level-triggered interface (the default, when **EPOLLET** is not specified), **epoll** is simply a faster **poll()**, and can be used wherever the latter is used since it shares the same semantics.
- Since even with edge-triggered epoll, multiple events can be generated upon receipt of multiple chunks of data, the caller has the option to specify the **EPOLLONESHOT** flag, to tell epoll to disable the associated file descriptor after the receipt of an event with **epoll_wait()**. When the **EPOLLONESHOT** flag is specified, it is the caller's responsibility to **rearm** the file descriptor using **epoll_ctl()** with **EPOLL_CTL_MOD**.