Professor Nelson Fonseca

http://www.lrc.ic.unicamp.br/mc833/

Roteiro

- **Objetivo: explicar formas de tratar vários descritores de arquivos de forma concorrente em um programa baseado em sockets (Capítulo 6 do livro texto)**

- Multiplexação de E/S
- Tipos de E/S no Unix
- `select`
- `poll`
- Atividade prática

Problemas no cliente/servidor atual

- Se o servidor for “morto” (CTRL+C)?
  - FIN enviado para o cliente
  - Cliente está aguardando a entrada do usuário (`fgets`)
  - Cliente só percebe que o cliente “morreu” tarde demais
- Se o servidor pudesse enviar dados a qualquer momento?
  - Cliente não sabe quando é esse momento
- Se os vários clientes pudessem comunicar-se entre si?
  - Cliente também não sabe quando esperar os dados de cada conexão (assíncrono)

Multiplexação de E/S

- Capacidade de avisar o kernel que se deseja ser notificado quando condições de E/S estejam válidas

  Ex: dados para leitura estão disponíveis

- `select` e `poll`

- Não só para sockets

Multiplexação de E/S – Quando usar?

- Quando cliente está manipulando vários descritores. Ex: descritor do socket é de entrada interativa
- Quando o cliente manipula vários sockets ao mesmo tempo
- Quando TCP manipula listening sockets e outros sockets conectados. Ex: servidor concorrente sem `fork`
- Quando o servidor lida com TCP e com UDP simultaneamente
- Quando o servidor manipula vários protocolos e serviços simultaneamente. Ex: `inetd`

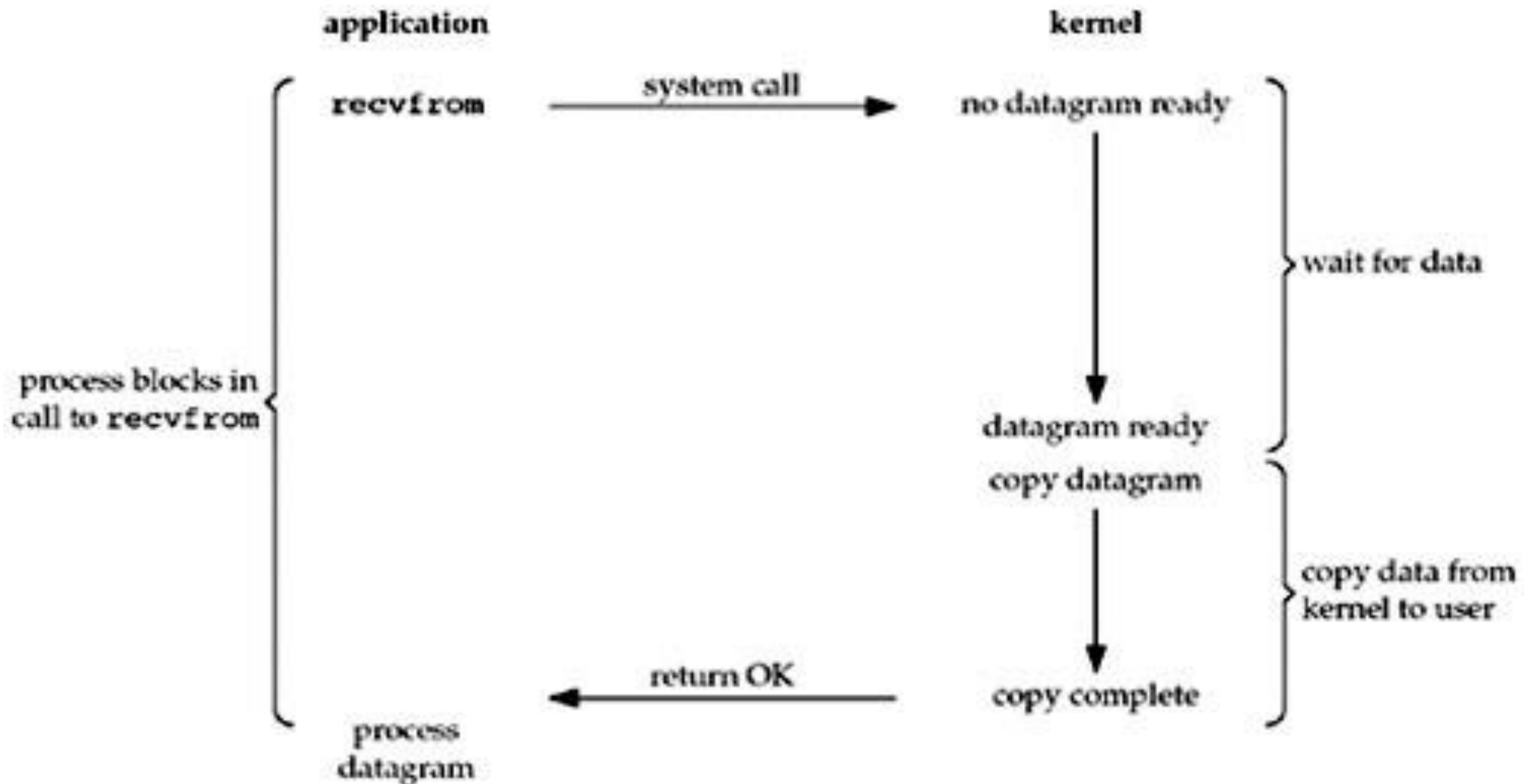
Tipos de E/S

- E/S bloqueante
- E/S **não** bloqueante
- Multiplexação de E/S
- E/S orientada a sinal
- E/S assíncrona

Operações de E/S

- Duas fases:
  1. Espera até dados estarem disponíveis
  2. Cópia dos dados do Kernel para processo

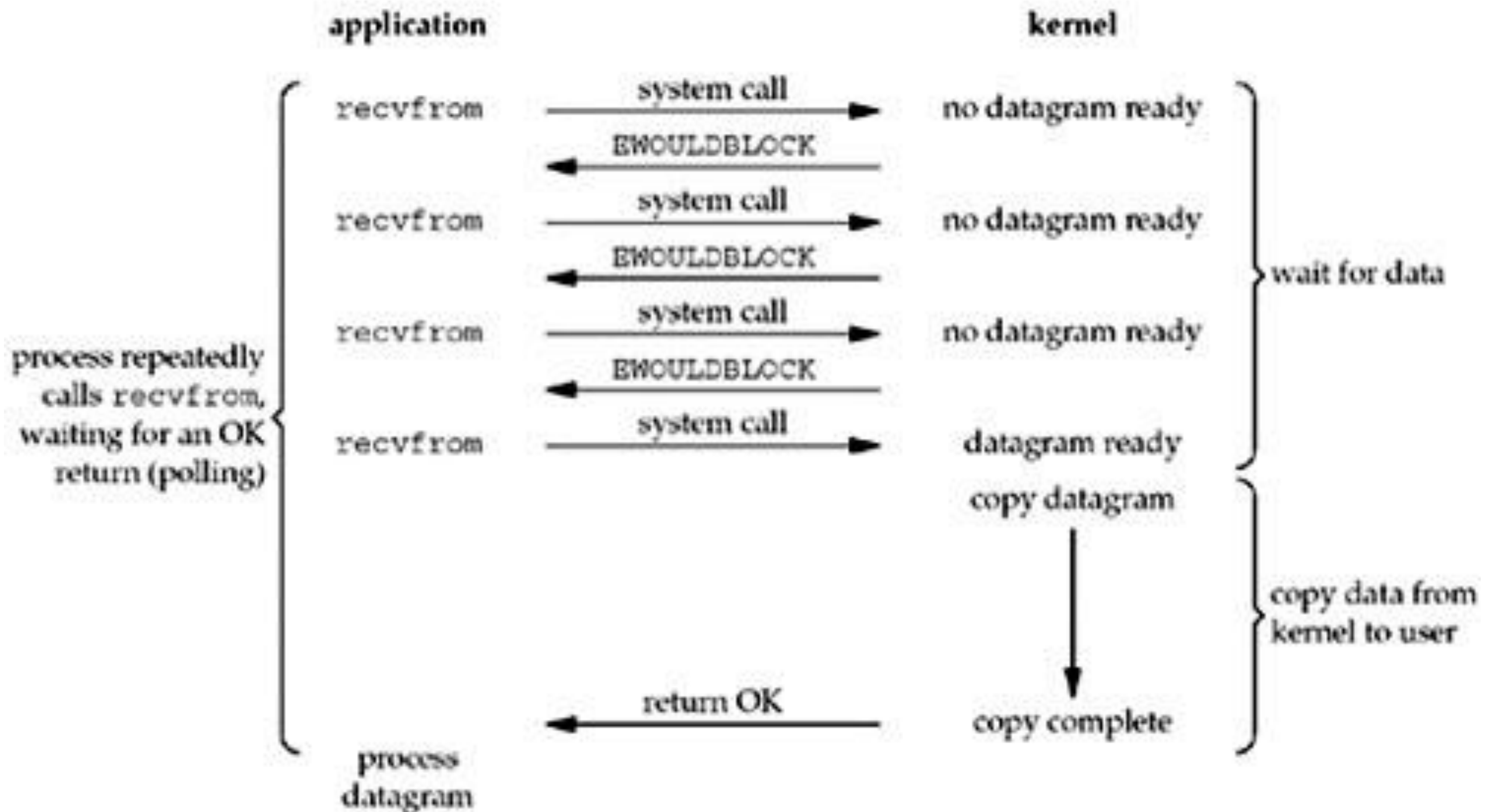
E/S bloqueante



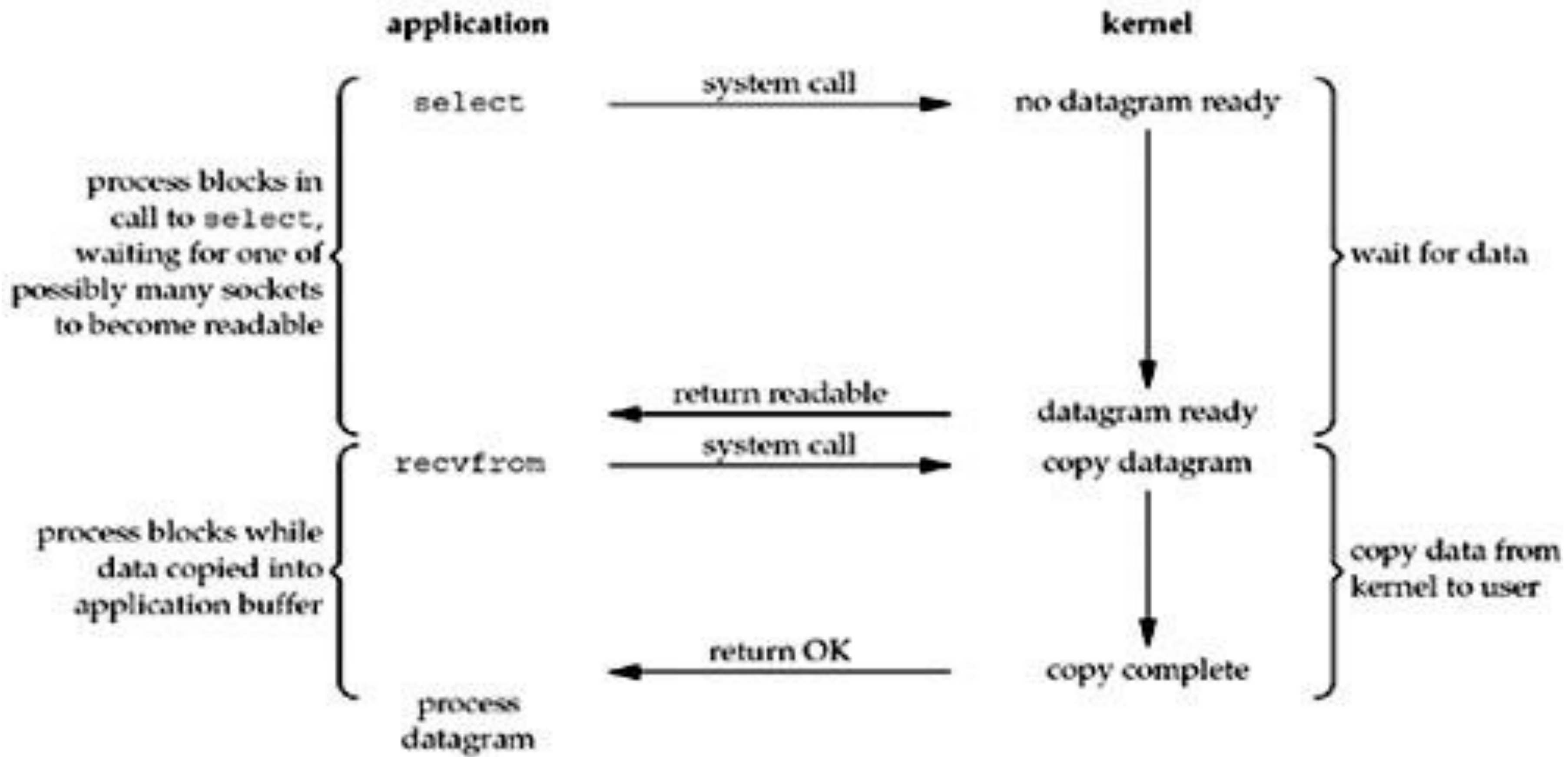
E/S não bloqueante

- Não coloca processo em estado “sleep”
- Retorna código de erro: `EWOULDBLOCK`
- Polling

E/S não bloqueante



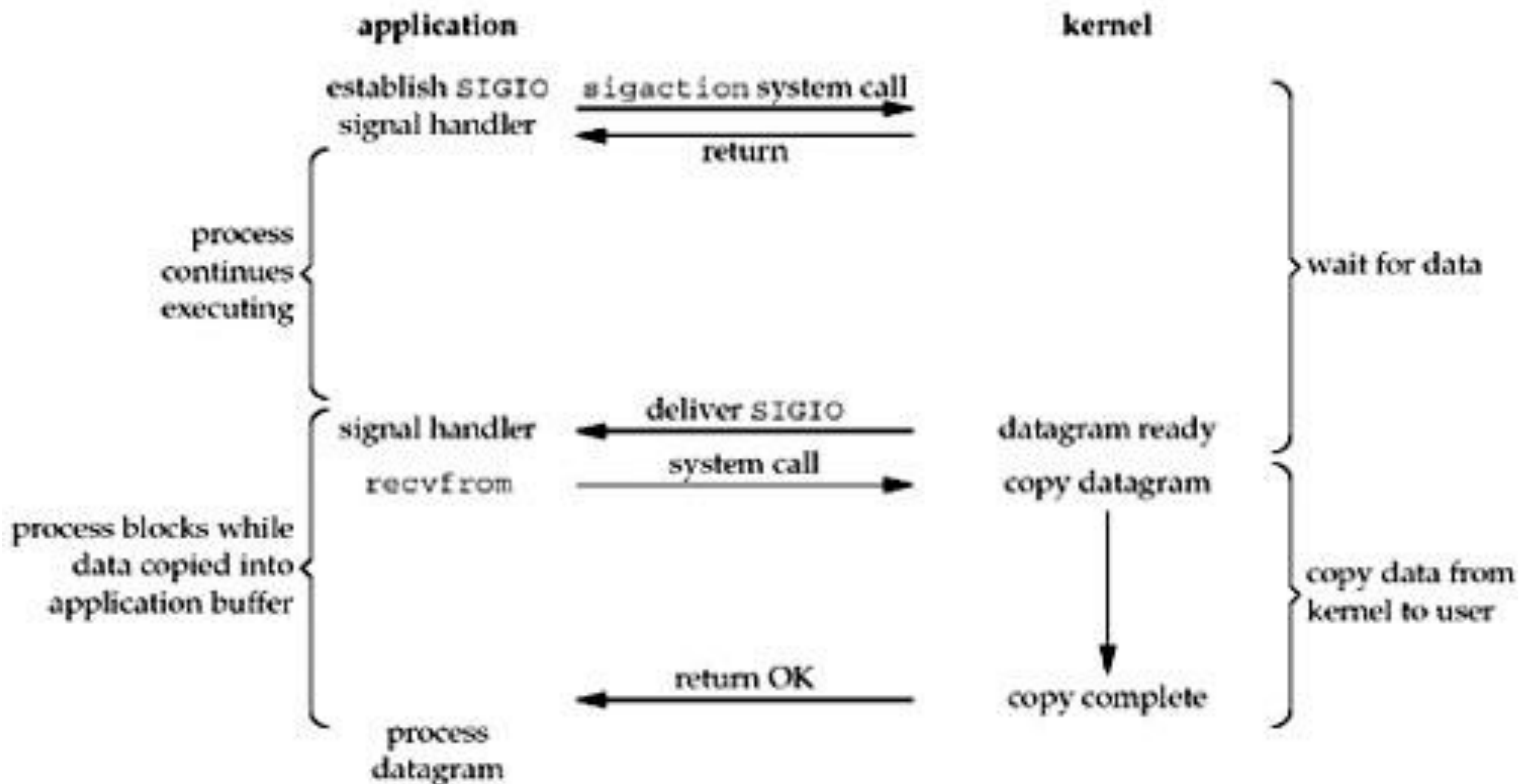
Multiplexação de E/S `(select)`



E/S orientada a sinal

- Solicita ao kernel que notifique através do sinal `SIGIO` quando evento ocorrer
- Não é bloqueante
- Opção de leitura: **`recvfrom`** ou o laço principal

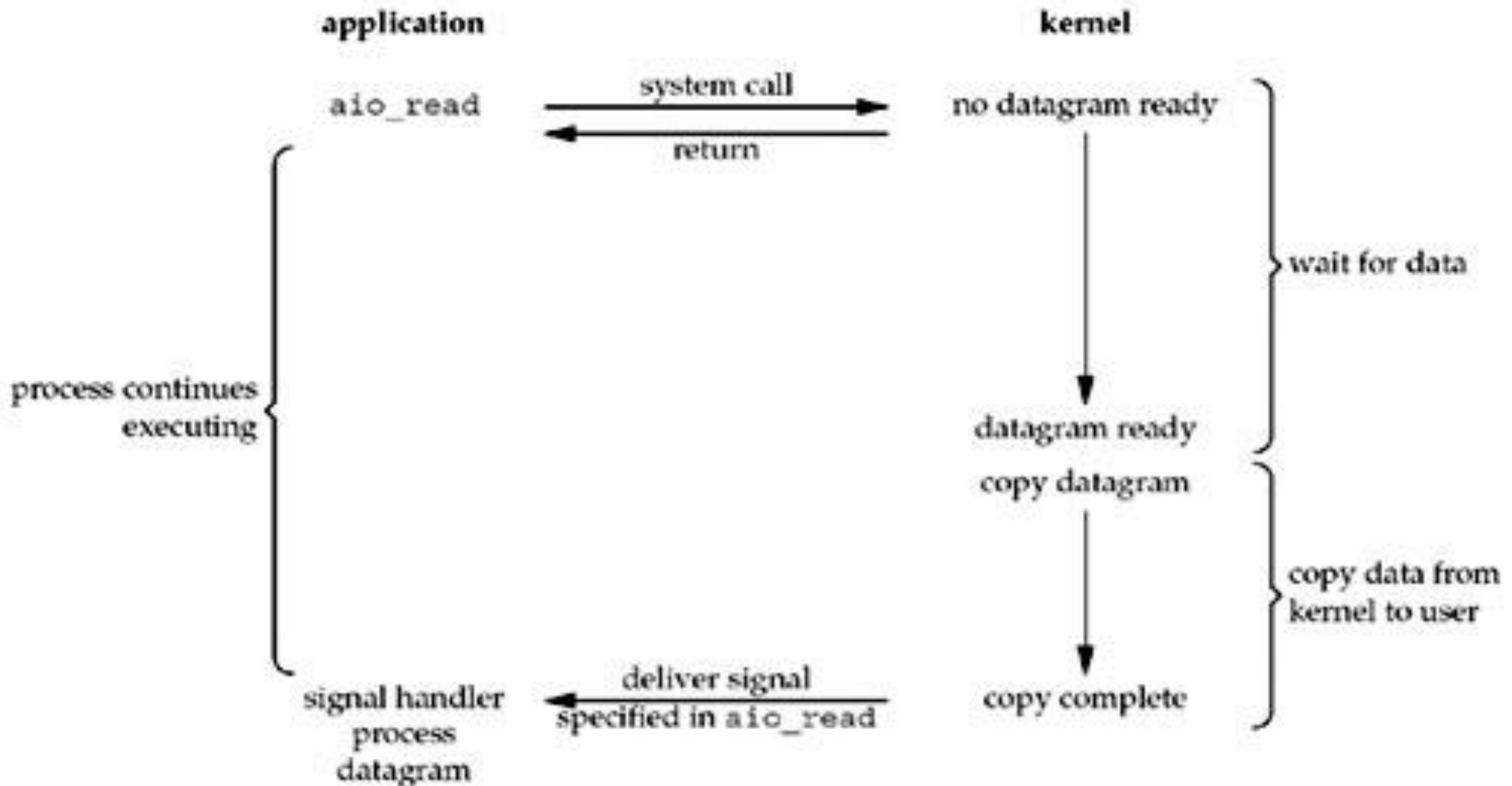
E/S orientada a sinal



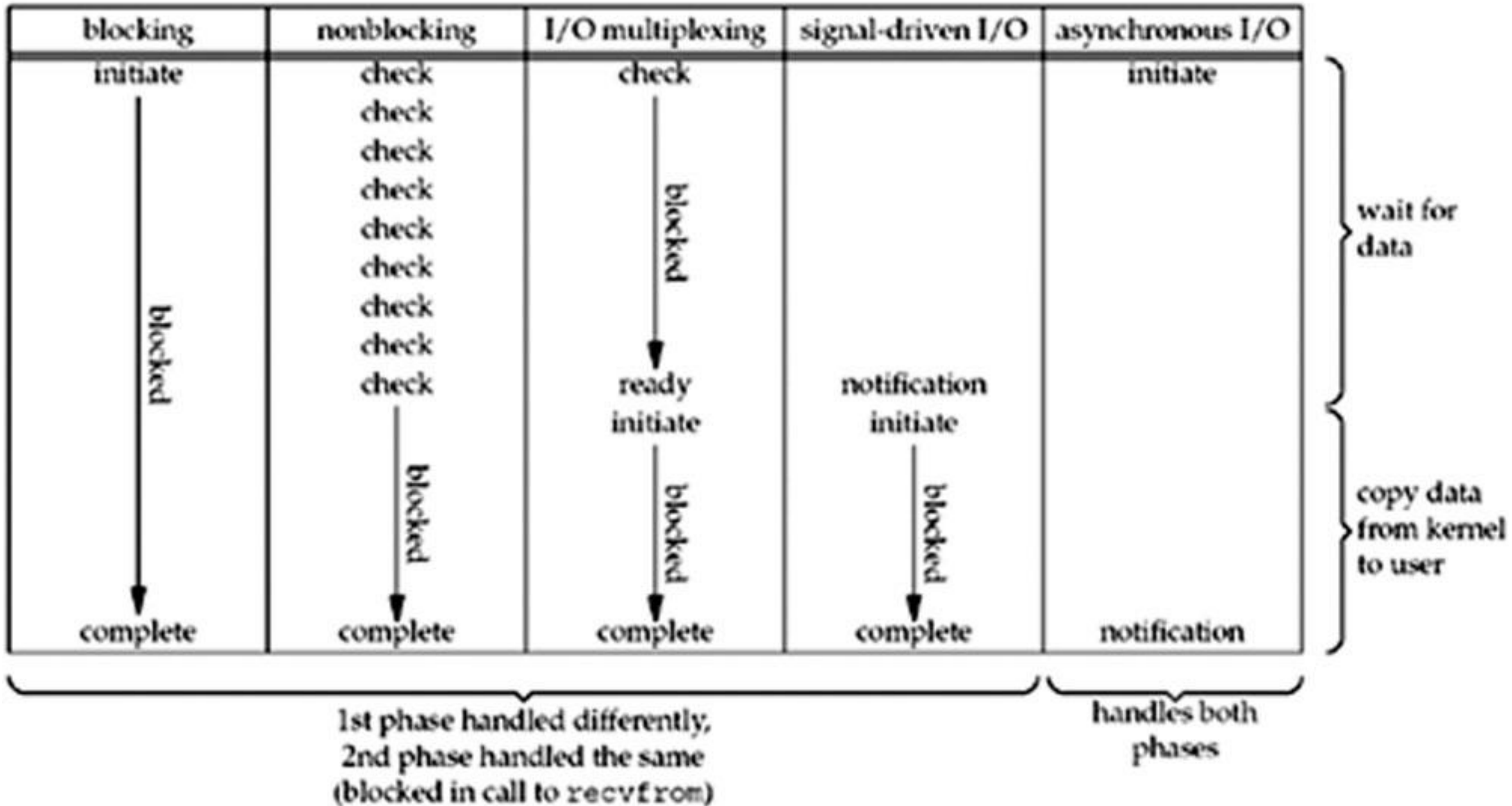
E/S assíncrona

- O processo diz ao kernel quando começar a operação
- O kernel notifica o processo quando toda a operação estiver completa incluindo cópia do dados do kernel para o buffer

E/S assíncrona



Tipos de E/S - Comparação



`select` (multiplexação de E/S)

- Instrui o kernel para “acordar” o processo quando um entre um conjunto de eventos ocorrer ou quando um certo intervalo de tempo tiver ocorrido
- Especifica qual descritor se tem interesse (leitura, escrita ou exceção)

select

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfdp1, fd_set *readset, fd_set *writeset,
            fd_set *exceptset, const struct timeval *timeout);

    Returns:      positive count of ready descriptors,
                  0 on timeout,
                  -1 on error
```

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

`select`

- Três possibilidades de espera (valores do `timeout`):
  - Espera até condição se tornar verdadeira – `NULL`
  - Espera por um valor máximo de intervalo
  - Não espera. Retorna após verificar os descritores (polling)
    - Os valores de temporização em `timeout` devem ser zero

select

- Duas opções de teste de exceção:
  - Chegada de dados fora de faixa (out-of-band)
  - Informações de controle de status em pseudo- terminais

select

- **descriptor set** (vetor de inteiros) em que cada bit de cada inteiro corresponde a um descritor
- Quatro macros:

```
void FD_ZERO(fd_set *fdset);

void FD_SET(int fd, fd_set *fdset);

void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);
```

`select`

- **`select`** modifica os descriptor sets de leitura, escrita, exceção
- Ligar todos os bits e verifica o valor no retorno (o `select` põe 0 onde não teve atividade)
- Importante deixar todos os bits em 1 antes!!! O autor do livro texto passou duas horas depurando um código em que o erro era esse

`select` (até agora)

1-) Definir quais são os descritores de interesse para ler, escrever ou monitorar exceções

2-) Criar um conjunto de descritores onde cada bit de cada inteiro corresponde a um descritor

3-) Ligar todos os bits antes de chamar o `select`

4-) Chamar o `select`

5-) Verificar quais bits nos descritores são iguais a 1

Condições de pronto - leitura

- Quantidade de bytes maior ou igual a um limitante inferior (low-water mark no livro). A opção do socket `SO_RCVLOWAT` permite atribuir o valor do limitante inferior
- Uma direção da conexão está fechada. Leitura não irá bloquear e retornará zero
- O número de conexões estabelecidas em um listening socket é não nulo
- Condição de erro, retorna -1 em **`errno`**. Condições de erro podem ser verificadas e anuladas com a opção **`getsockopt`**.

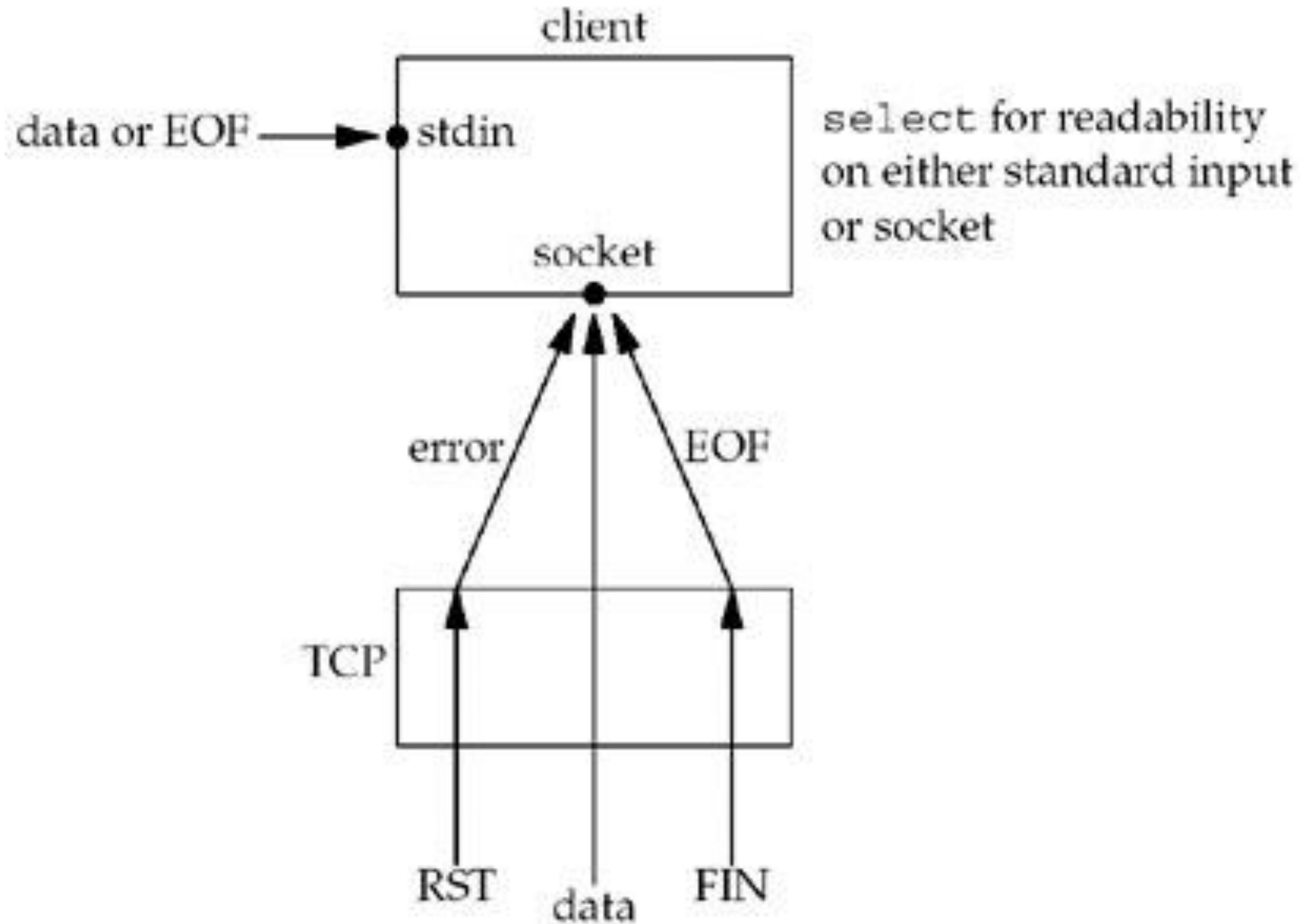
Condições de pronto - escrita

- Quantidade de bytes no buffer de escrita é maior que limite inferior e **i)** socket está conectado ou **ii)** não requer conexão (UDP). O limitante pode ser especificado pela opção `SO_SNDLOWAT`
- A direção na conexão está fechada. Retorna `SIGPIPE`
- Condição de erro existente
- Dados fora de faixa recebidos

cliente original

```
1 #include "unp.h"
2      void
3      str_cli(FILE *fp, int sockfd)
4      {
5              char sendline[MAXLINE], recvline[MAXLINE];
6              while (Fgets(sendline, MAXLINE, fp) != NULL) {
7              Writen(sockfd, sendline, strlen (sendline));
8              if (Readline(sockfd, recvline, MAXLINE) == 0)
9                      err_quit("str_cli:server terminated
prematurely");
10             Fputs(recvline, stdout);
11     }
12 }
```

cliente modificado



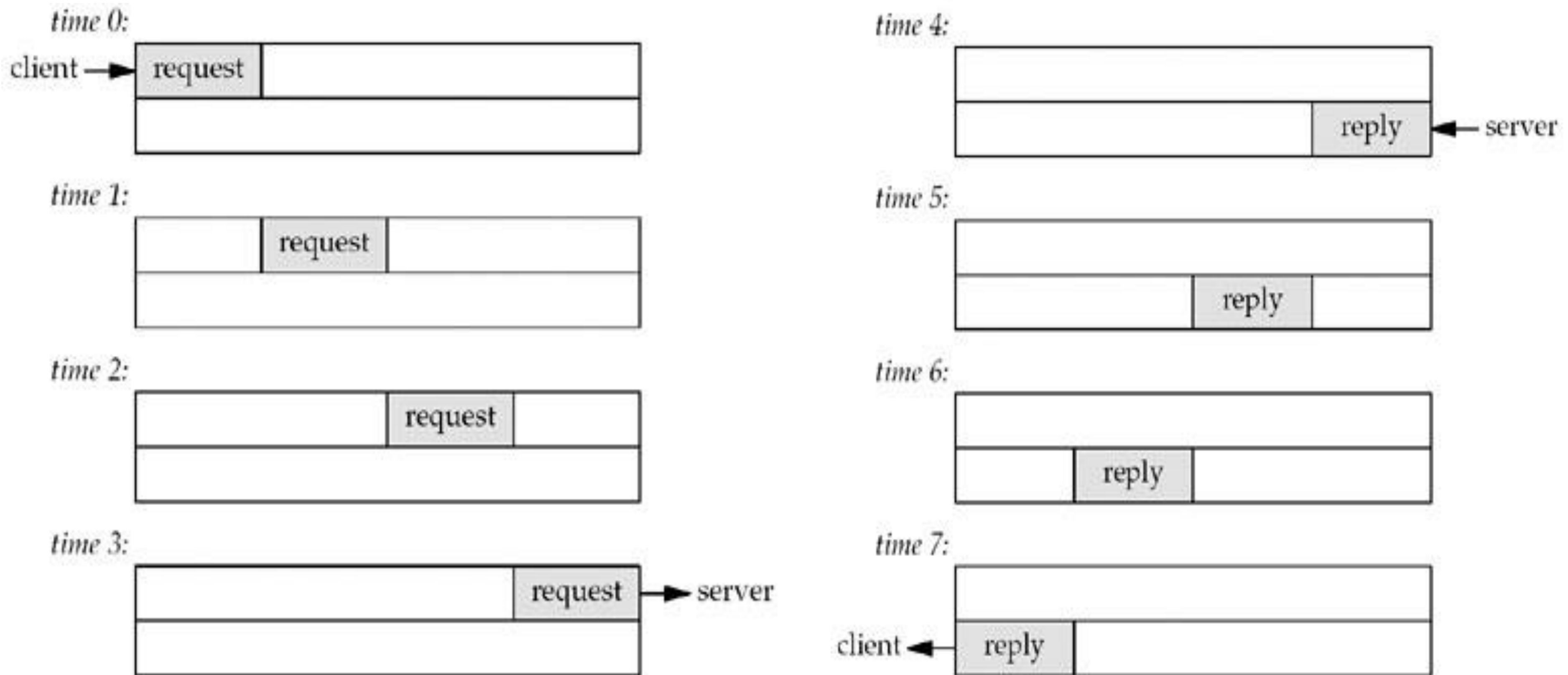
cliente modificado

```
2 void str_cli(FILE *fp, int sockfd)  {
...
8       FD_ZERO(&rset);
9       for ( ; ; ) {
10              FD_SET(fileno(fp), &rset);
11              FD_SET(sockfd, &rset);
12              maxfdp1 = max(fileno(fp), sockfd) + 1;
13              Select(maxfdp1, &rset, NULL, NULL, NULL);
14              if (FD_ISSET(sockfd, &rset)) {  /* atividade no socket */
15                      if (Readline(sockfd, recvline, MAXLINE) == 0)
16                           err_quit("str_cli: server terminated
prematurely");
17                      Fputs(recvline, stdout);
18              }
19              if (FD_ISSET(fileno(fp), &rset)) {      /* atividade na
entrada padrão */
20                      if (Fgets(sendline, MAXLINE, fp) == NULL)
21                           return;
22                      Writen(sockfd, sendline, strlen(sendline));
23              }  } }
```

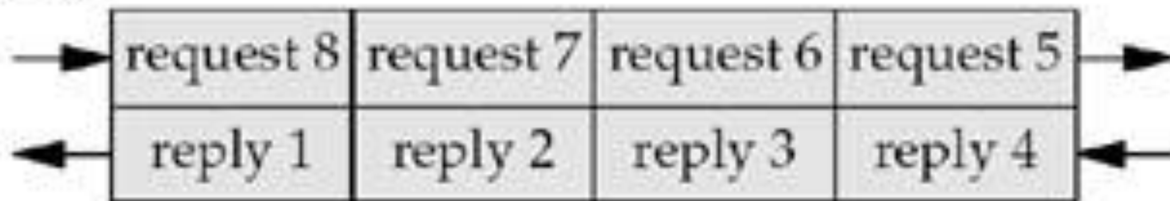
cliente modificado

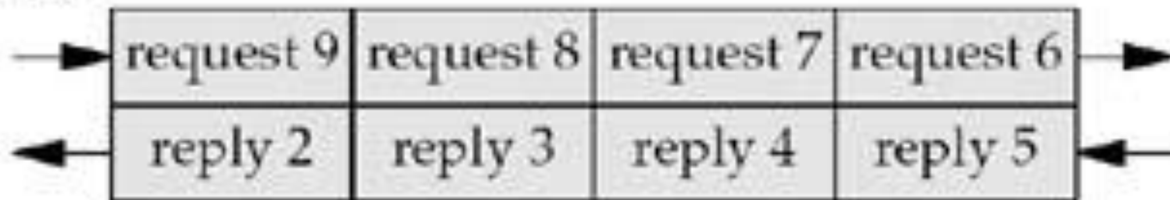
- Função `fileno` converte um ponteiro para arquivo E/S em um ponteiro para descritores (`select` e `poll` trabalham só com descritores)
- Descritores de escrita e exceção são atribuídos com `NULL`
- O fluxo mudou, não é mais ditado por `fget` mas sim por `select`

cliente modificado (Modo pára-espera OK)



cliente modificado (Processamento em bloco traz problemas)



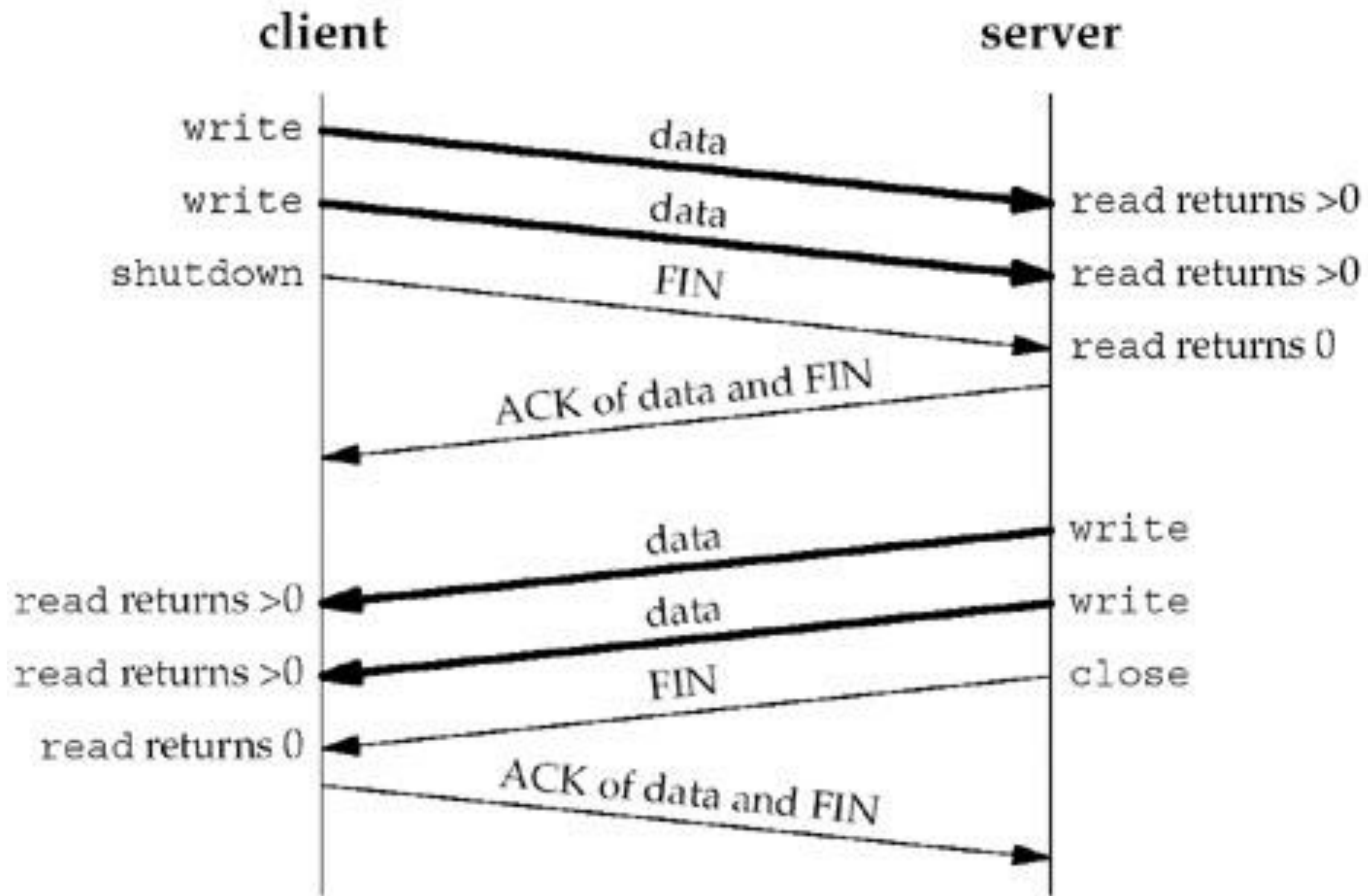


- Se após o "request 9", o arquivo termina, o cliente não pode encerrar pois ainda falta receber dados!

Como fechar a conexão somente do lado do cliente?

- `close` fecha as duas direções da conexão full duplex
- **`shutdown` é a solução** – envia FIN

shutdown



shutdown

```
#include <sys/socket.h>
int shutdown(int sockfd, int howto);
                    Returns: 0 if OK, -1 on error
```

- **howto**:
  - SHUT_RD – fecha para leitura (read-half). Conteúdo do buffer e futuras recepções são descartadas
  - SHUT_WR – qualquer dado no buffer de escrita será enviado. Não pode mais escrever
  - SHUT_RDWR – efeito de fazer SHUT_WR após o SHUT_RD

Correções no cliente

- Ao ter atividade na entrada padrão:
  - Verificar se a leitura da entrada chegou no EOF
    - Se chegou no EOF, fecha a conexão só para envio de dados (`SHUT_WR`) e...
    - ...pára de tratar a entrada padrão no conjunto de descritores
- Ao ter atividade no socket:
  - Verificar se a leitura do socket chegou no EOF
    - Se chegou no EOF do socket e já chegou no EOF da entrada padrão (detectado no passo anterior), o cliente termina com sucesso;
    - Se chegou no EOF do socket mas não chegou no EOF da entrada padrão, ocorreu erro.

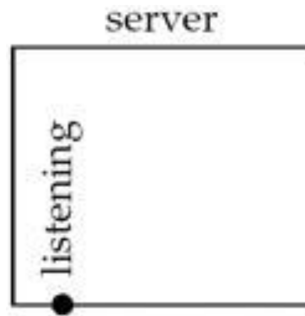
Até agora... O que vem a seguir...

- Cliente foi modificado para utilizar o `select`
- A atividade *prática* já pode ser realizada
  - Só precisa modificar o cliente
- Mas o servidor também pode tirar proveito do select como será visto a seguir

Atividade prática

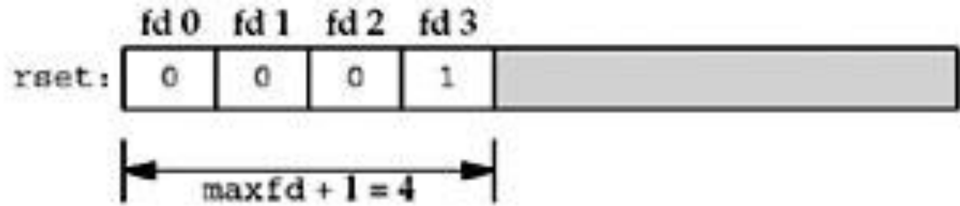
- Melhorar o **cliente** de eco da atividade 2 parte 1 utilizando multiplexação de E/S
- http://www.lrc.ic.unicamp.br/mc833

Servidor modificado

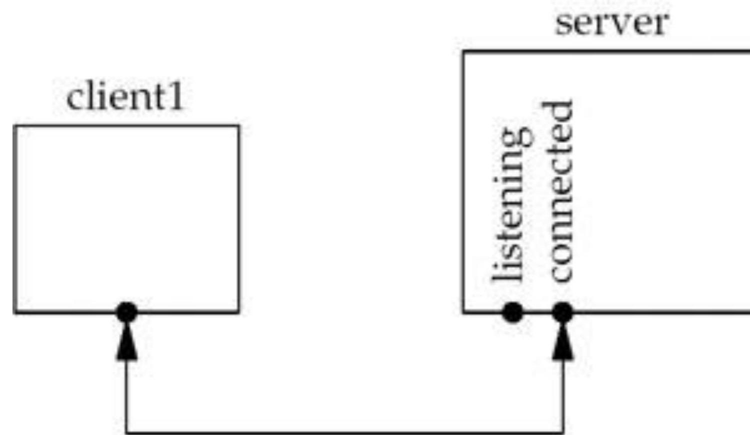


client[]:

[0]	−1
[1]	−1
[2]	−1
[FD_SETSIZE-1]	−1

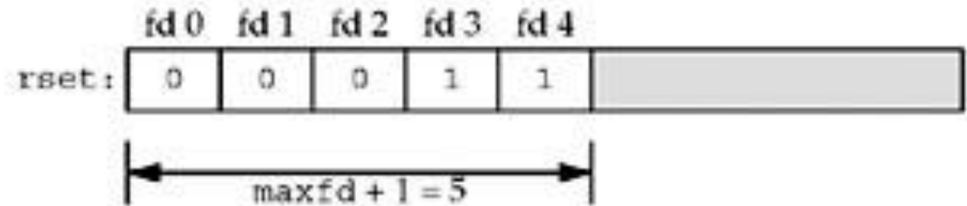


Servidor modificado

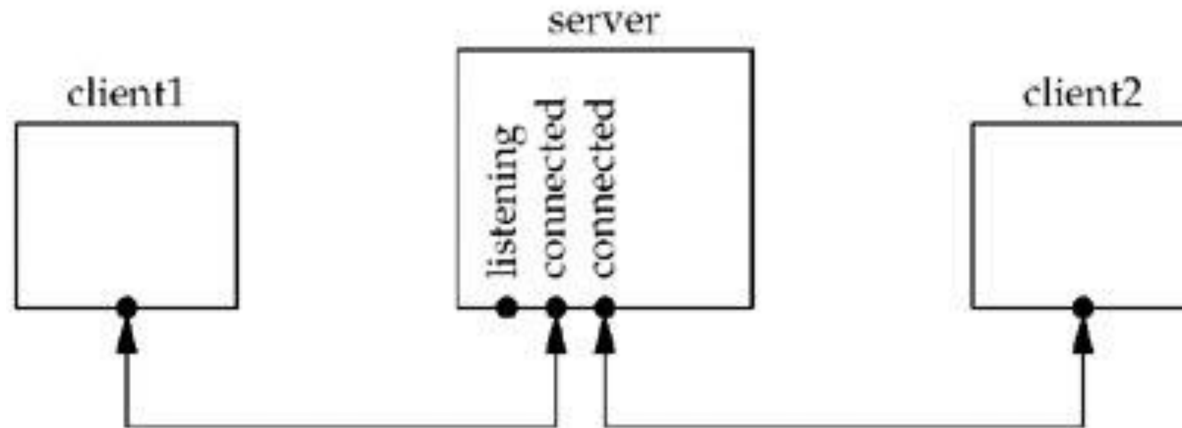


client[]:

[0]	4
[1]	−1
[2]	−1
[FD_SETSIZE-1]	−1



Servidor modificado



client[]:

[0]	4
[1]	5
[2]	-1
[FD_SETSIZE-1]	-1

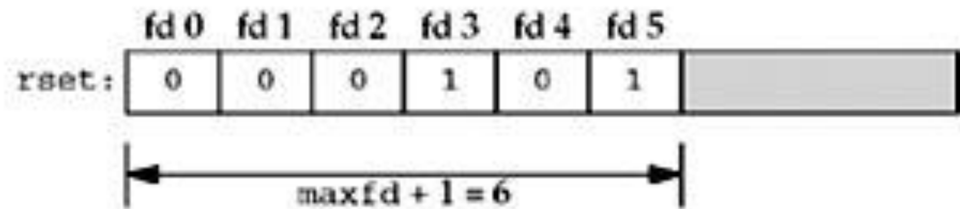
	fd 0	fd 1	fd 2	fd 3	fd 4	fd 5
rset:	0	0	0	1	1	1

maxfd + 1 = 6

Servidor modificado

client[]:

[0]	-1
[1]	5
[2]	-1
[FD_SETSIZE-1]	-1



Servidor modificado

```
1 int  main(int argc, char **argv)  {
...
12      listenfd = Socket(AF_INET, SOCK_STREAM, 0);
13      bzero(&servaddr, sizeof(servaddr));
14      servaddr.sin_family = AF_INET;
15      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
16      servaddr.sin_port = htons(SERV_PORT);
17      Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
18      Listen(listenfd, LISTENQ);
19      maxfd = listenfd; /* initialize */
20      maxi = -1; /* index into client[] array */
21      for (i = 0; i < FD_SETSIZE; i++)
22              client[i] = -1; /* -1 indicates available entry */
23      FD_ZERO(&allset);
24      FD_SET(listenfd, &allset);
…
```

```
25      for ( ; ; ) {
26          rset = allset; /* structure assignment */
27          nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);
28          if (FD_ISSET(listenfd, &rset)) { /* new client conn */
29               clilen = sizeof(cliaddr);
30               connfd = Accept(listenfd, (SA *) &cliaddr,
&clilen);
31               for (i = 0; i < FD_SETSIZE; i++)
32                   if (client[i] < 0) {
33                        client[i] = connfd; /* save descriptor */
34                        break;
35                   }
36               if (i == FD_SETSIZE)
37                   err_quit("too many clients");
38               FD_SET(connfd, &allset);  /* add new descriptor to
set */
39               if (connfd > maxfd)
40                   maxfd = connfd; /* for select */
41               if (i > maxi)
42                   maxi = i; /* max index in client[] array */
43               if (--nready <= 0)
44                   continue; /* no more readable descriptors */
45          }*/ if */
...
```

Servidor modificado

```
46      for (i = 0; i <= maxi; i++) { /* check all clients for
data */
47      if ( (sockfd = client[i]) < 0)
48              continue;
49      if (FD_ISSET(sockfd, &rset)) {
50              if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
51                      /* connection closed by client */
52                      Close(sockfd);
53                      FD_CLR(sockfd, &allset);
54                      client[i] = -1;
55              } else
56                      Writen(sockfd, buf, n);
57              if (--nready <= 0)
58                              break; /* no more readable
descriptors */
59              }
60      }
61    } /* end for */
62 }/* end main */
```

Servidor modificado (Problema de DoS)

- Se o cliente envia 1 byte e entra em estado “sleep”, o servidor bloqueia
- Servidor que aceita pedidos de vários clientes nunca pode se bloquear com um pedido de um cliente individual

- Possíveis soluções:
  1. Uso de E/S não bloqueante
  2. Cada cliente tratado como uma thread de controle (individual)
  3. Colocar temporizador na operação de E/S

poll

```
#include <poll.h>
int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);

      Returns: count of ready descriptors, 0 on timeout, –1 on error
```

```
struct pollfd {
int fd;                  /* descriptor to check */
short events;            /* events of interest on fd */
short revents;           /* events that occurred on fd */
};
```

poll

Constant	Input to *events* ?	Result from *revents* ?	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 occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

poll

- Três tipos de dados:
  - Normal
  - Faixa prioritária (priority band)
  - Alta prioridade
- Dados TCP e UDP são normais
- TCP fora de faixa (out-of-band) é faixa prioritária
- Nova conexão em socket listening pode ser considerada normal ou prioritária

timeout value	Description
INFTIM	Wait forever
0	Return immediately, do not block
> 0	Wait specified number of milliseconds

Servidor com `poll`

```
1 int  main(int argc, char **argv) {
...
13     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
14     bzero(&servaddr, sizeof(servaddr));
15     servaddr.sin_family = AF_INET;
16     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17     servaddr.sin_port = htons(SERV_PORT);
18     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
19     Listen(listenfd, LISTENQ);
20     client[0].fd = listenfd;
21     client[0].events = POLLRDNORM;
22     for (i = 1; i < OPEN_MAX; i++)
23            client[i].fd = -1; /* -1 indicates available entry
*/
24     maxi = 0; /* max index into client[] array */
...
```

Servidor com `poll`

```
...
25     for ( ; ; ) {
26     nready = Poll(client, maxi + 1, INFTIM);
27     if (client[0].revents & POLLRDNORM) { /* new client
conn */
28     clilen = sizeof(cliaddr);
29     connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
30     for (i = 1; i < OPEN_MAX; i++)
31             if (client[i].fd < 0) {
32                client[i].fd = connfd; /* save descriptor */
33                break;
34             }
35     if (i == OPEN_MAX)
36             err_quit("too many clients");
37     client[i].events = POLLRDNORM;
38     if (i > maxi)
39             maxi = i; /* max index in client[] array */
40     if (--nready <= 0)
41             continue; /* no more readable descriptors */
42 }
...
```

Servidor com `poll`

```
43      for (i = 1; i <= maxi; i++) {    /* check all clients for data */
44              if ( (sockfd = client[i].fd) < 0)
45                      continue;
46              if (client[i].revents & (POLLRDNORM | POLLERR)) {
47                      if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
48                          if (errno == ECONNRESET) {
49                              /* connection reset by client */
50                              Close(sockfd);
51                              client[i].fd = -1;
52                          } else
53                              err_sys("read error");
54                      } else if (n == 0) {
55                              /* connection closed by client */
56                              Close(sockfd);
57                              client[i].fd = -1;
58                      } else
59                              Writen(sockfd, buf, n);
60                      if (--nready <= 0)
61                              break;    /* no more readable descriptors
*/
62              } } } }
```