I/O Multiplexing and Posix Threads

Possible Mechanisms for Creating Concurrent Service

1. Processes
   - Kernel automatically interleaves multiple logical flows.
   - Each flow has its own private address space.
2. I/O multiplexing with **`select()`**
   - User manually interleaves multiple logical flows.
   - Each flow shares the same address space.
   - Popular for high-performance server designs.
3. Threads
   - Kernel automatically interleaves multiple logical flows.
   - Each flow shares the same address space.

Process, Thread, Context Switch

- A process has its own memory address space
- Threads share the heap of their parent process
- Context switch
  - Save all the process/thread states and/or registers

Fork()

```
#include <unistd.h>
pid_t fork(void)
```

- Fork(): Clones calling process
  - Returns: 0 in child, process ID of child in parent, -1 on error
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list

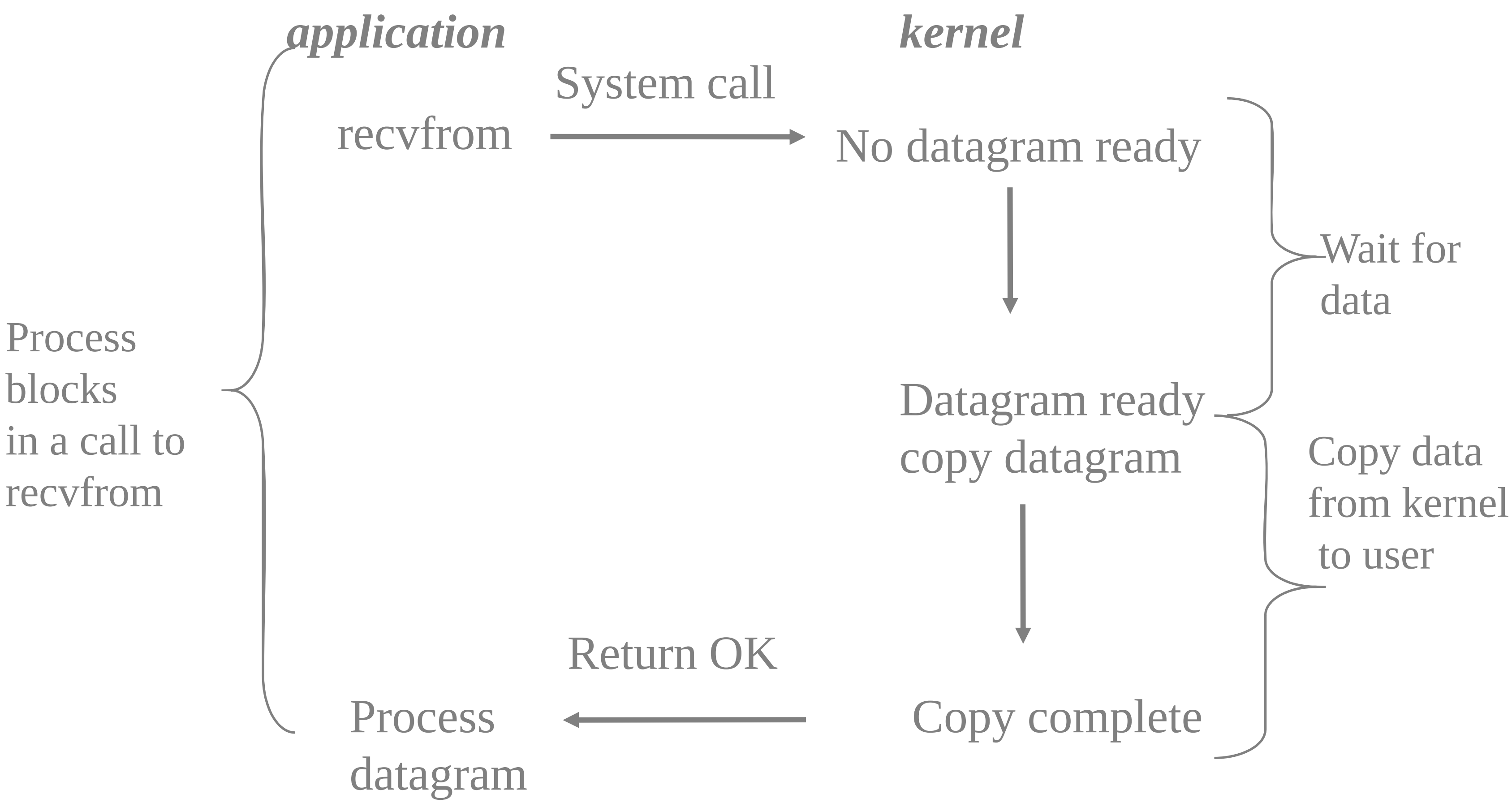
All descriptors open in the parent are shared with the child

I/O Models

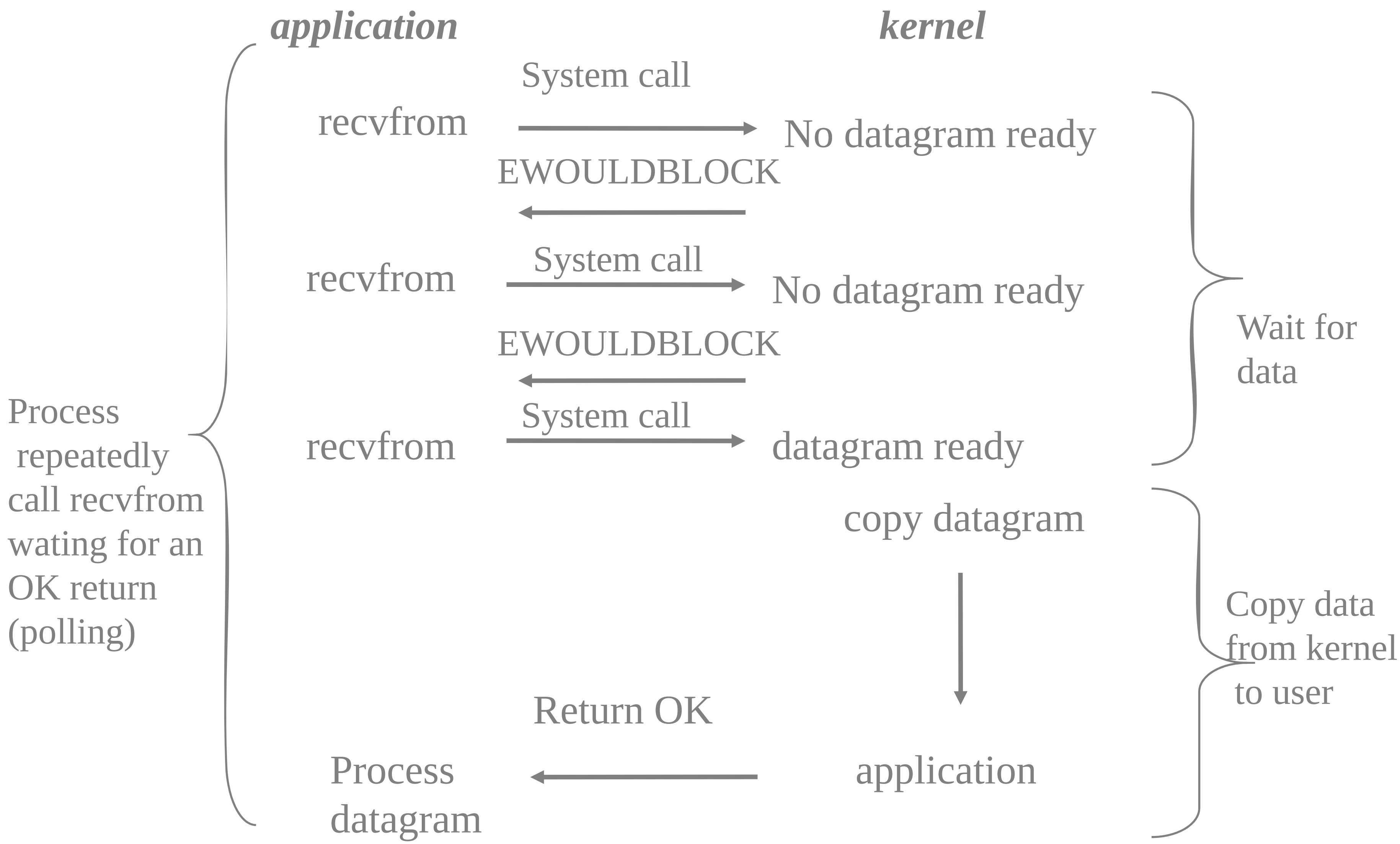
- Blocking I/O
- Non-blocking I/O
- I/O multiplexing
- Signal driven I/O
- Asynchronous I/O

Blocking I/O

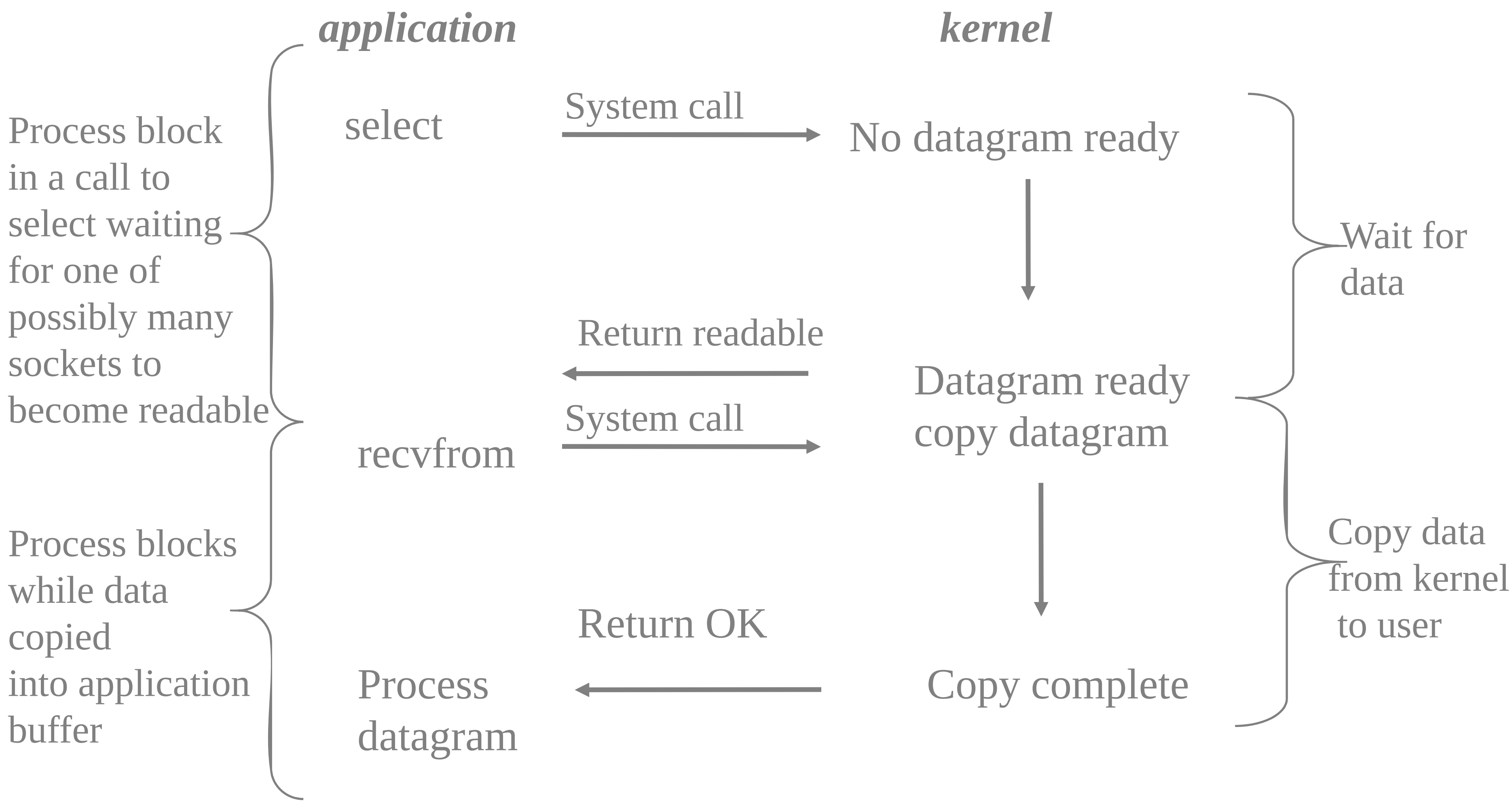
- By default, **`accept()`**, **`recv()`**, etc block until there's input



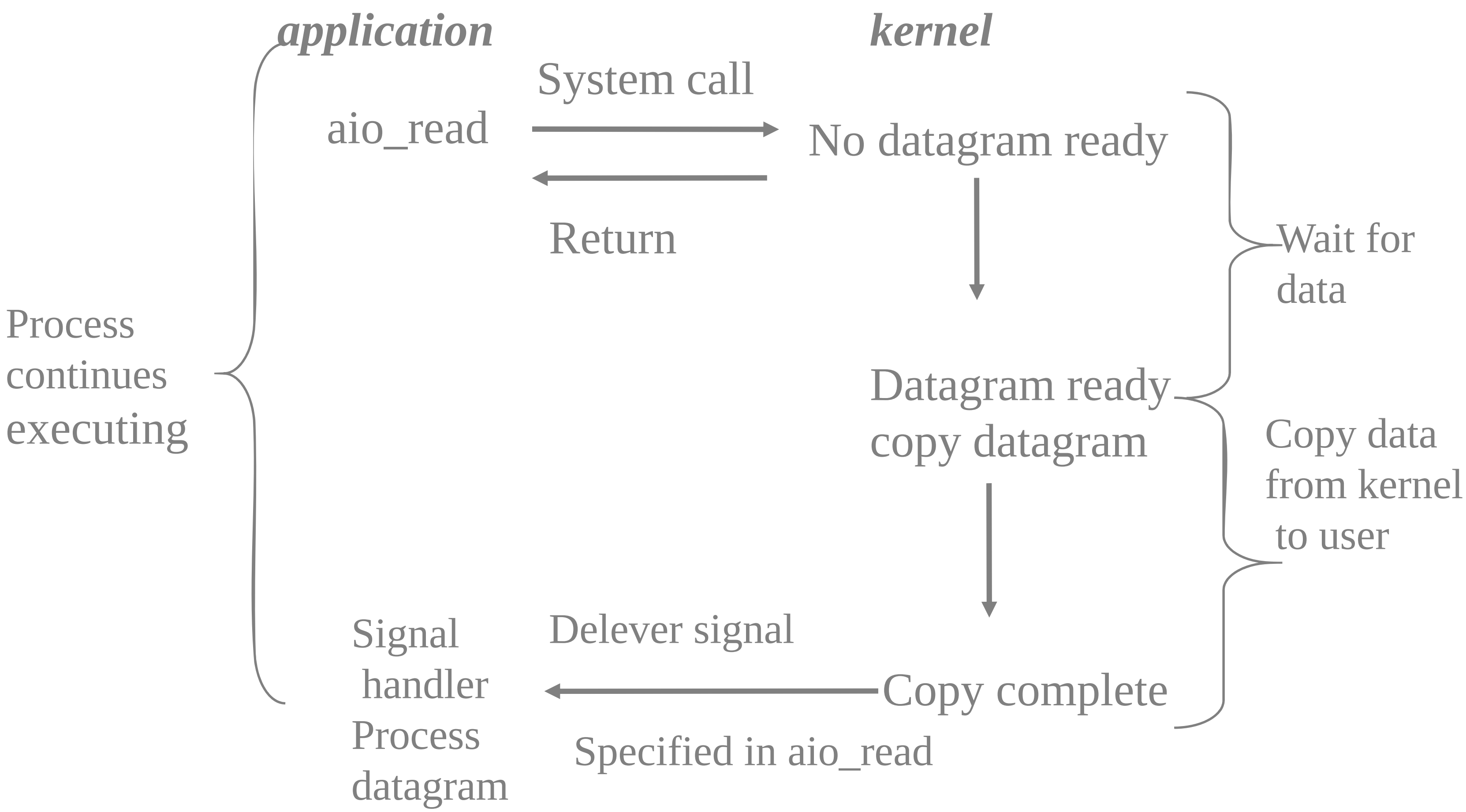
nonblocking I/O



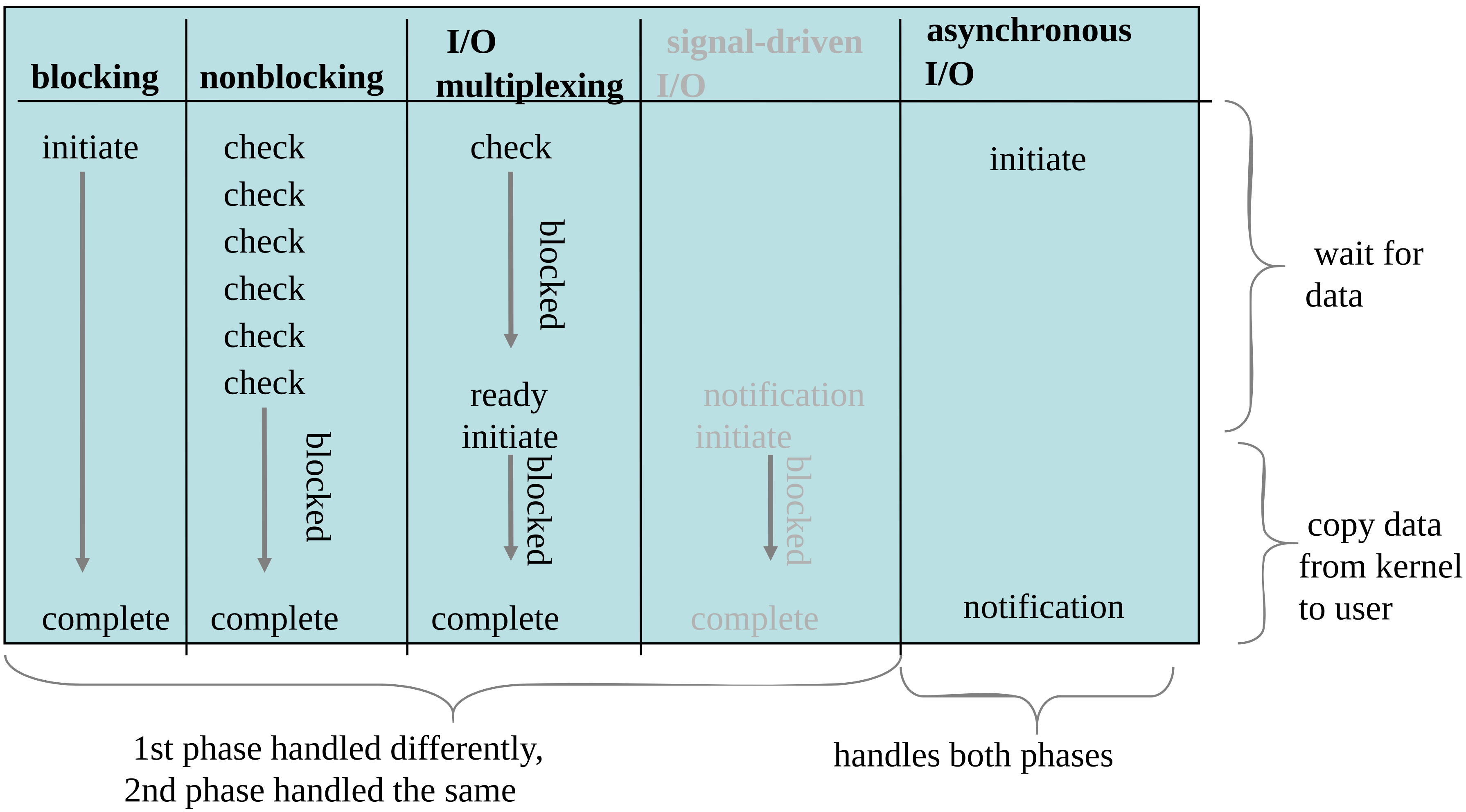
I/O multiplexing(select and poll)



asynchronous I/O



Comparison of the I/O Models



When is I/O Multiplexing Useful?

- A client is handling multiple descriptors (normally interactive input and a network socket)
- A TCP server handles both a listening socket and its connected sockets
- A server handles both TCP and UDP sockets
- A server handles multiple services and multiple protocols

Concurrent Servers Using I/O Multiplexing

- Maintain a pool of connected descriptors.
- Repeat the following forever:
  - Use the Unix select function to block until:
    - (a) New connection request arrives on the listening descriptor.
    - (b) New data arrives on an existing connected descriptor.
  - If (a), add the new connection to the pool of connections.
  - If (b), read any available data from the connection
    - Close connection on EOF and remove it from the pool.

The select Function

- **`select()`** sleeps until one or more file descriptors in the set readset ready for reading or one or more descriptors in writeset ready for writing or in event of an exception condition

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

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

- **`select()`** returns the number of ready descriptors

Select() Arugments

- Value-result arguments

`readset`

- Opaque bit vector (max FD_SETSIZE bits) that indicates membership in a *descriptor set.*
  - On Linux machines, FD_SETSIZE = 1024
- If bit k is 1, then descriptor k is a member of the descriptor set.
- When call **`select`**, should have readset indicate which descriptors to test

`writeset`

- **`writeset`** is similar but refers to descriptors ready for writing

`maxfdp1`

- Maximum descriptor in descriptor set plus 1.
- Tests descriptors 0, 1, 2, ..., maxfdp1 - 1 for set membership.

Macros for Manipulating Set Descriptors

- **`void FD_ZERO(fd_set *fdset);`**
  - Turn off all bits in `fdset`.
- **`void FD_SET(int fd, fd_set *fdset);`**
  - Turn on bit `fd` in `fdset`.
- **`void FD_CLR(int fd, fd_set *fdset);`**
  - Turn off bit `fd` in `fdset`.
- **`int FD_ISSET(int fd, *fdset);`**
  - Is bit `fd` in `fdset` turned on?

Example of Descriptor sets function

```
fd set  rset;

FD_ZERO(&rset);/*all bits off : initiate*/
FD_SET(1, &rset);/*turn on bit fd 1*/
FD_SET(4, &rset); /*turn on bit fd 4*/
FD_SET(5, &rset); /*turn on bit fd 5*/
```

Condition that cause a socket to be ready for *select*

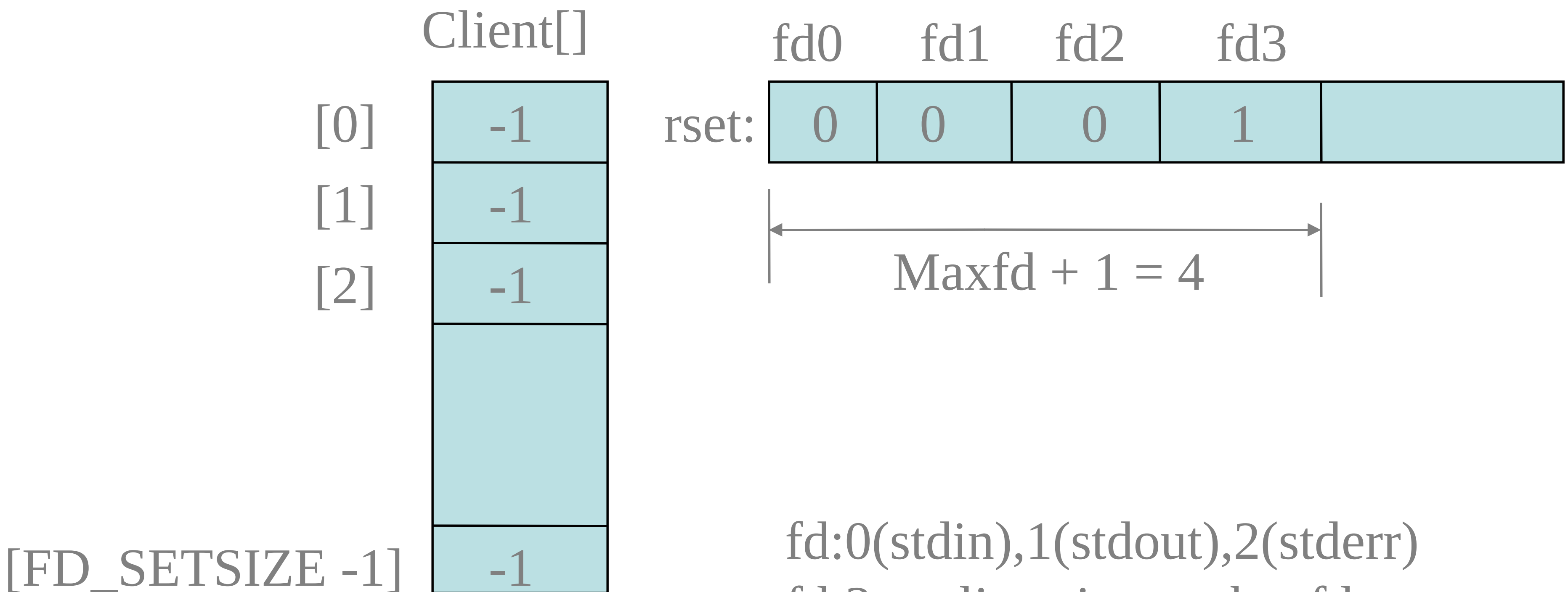
Condition	*Readable?*	*writable?*	*Exception?*
Data to read *read-half of the connection closed* *new connection ready for listening socket*	• • •		
Space available for writing *write-half of the connection closed*		• •	
Pending error	•	•	
TCP out-of-band data			•

TCP echo server

- Rewrite the server as a single process that uses select to handle any number of clients, instead of forking one child per client.

Data structure TCP server(1)

Before first client has established a connection

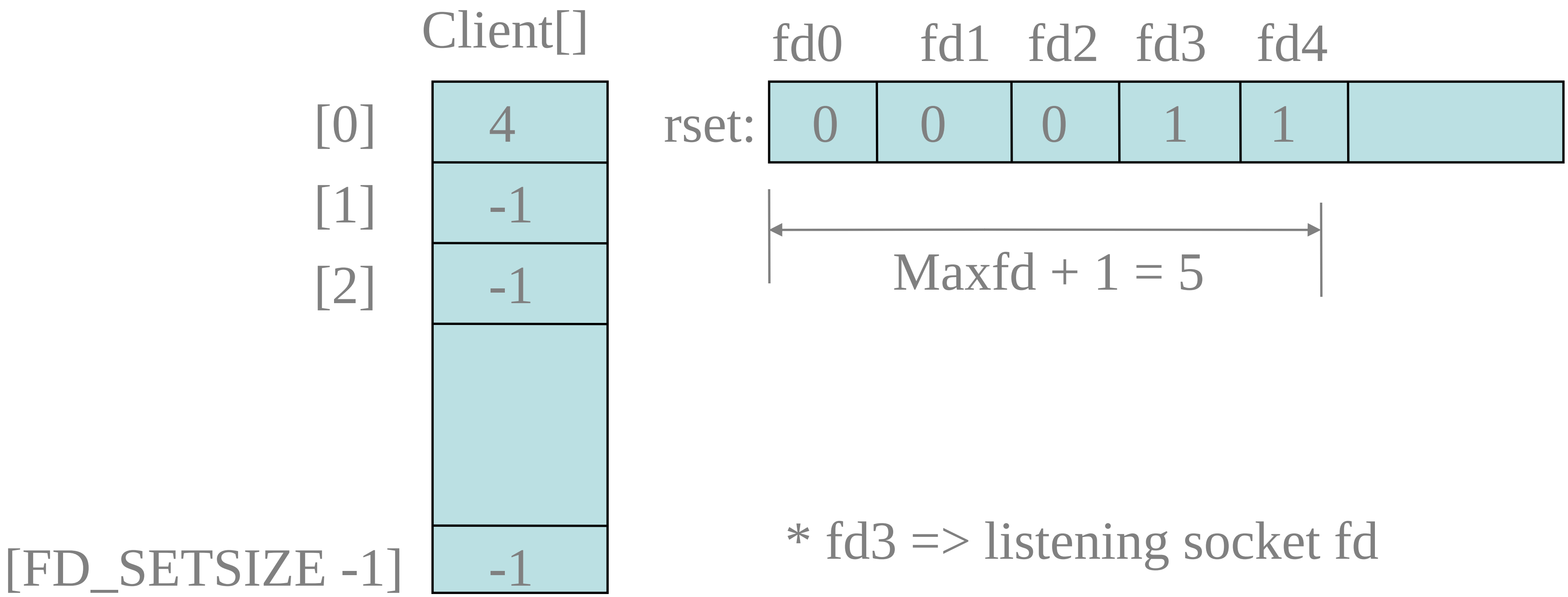


fd:0(stdin),1(stdout),2(stderr)

fd:3 => listening socket fd

Data structure TCP server(2)

After first client connection is established



* fd3 => listening socket fd

*fd4 => client socket fd

TCP echo server using single process

- See Page 2
- Demo

POSIX Thread

- Threads are “lightweight processes”
  - Creation of threads are usually 10-100 faster
  - Threads within a process share the same global memory → sharing of information among threads is easy

- Further readings:

http://www.llnl.gov/computing/tutorials/pthreads/

Comparison

Time to execute 50,000 process/thread creations

Platform	fork()			pthread_create()		
	real	**user**	**sys**	**real**	**user**	**sys**
IBM 375 MHz POWER3	61.94	3.49	53.74	7.46	2.76	6.79
IBM 1.5 GHz POWER4	44.08	2.21	40.27	1.49	0.97	0.97
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

Real time – time between invocation and termination
User time – time spent in the user program
System time – time spent in the kernel as a result of user program

pthread_create function

Value-result parameter: thread Id

Attribute of the thread; NULL if default

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
void *(*func) (void *), void *arg);
```

The function and its argument to execute; multiple arguments are packaged into a structure

0, successful;
Nonzero, error

Thread Termination

```
#include <pthread.h>
void pthread_exit(void *retval);
```

- A thread can be *joinable* (default) or *detached*
  - One can specify which thread to wait for; a joinable thread’s ID and exit status are retained until another thread calls *pthread_join*
  - Detached thread, upon termination, all its resources are release

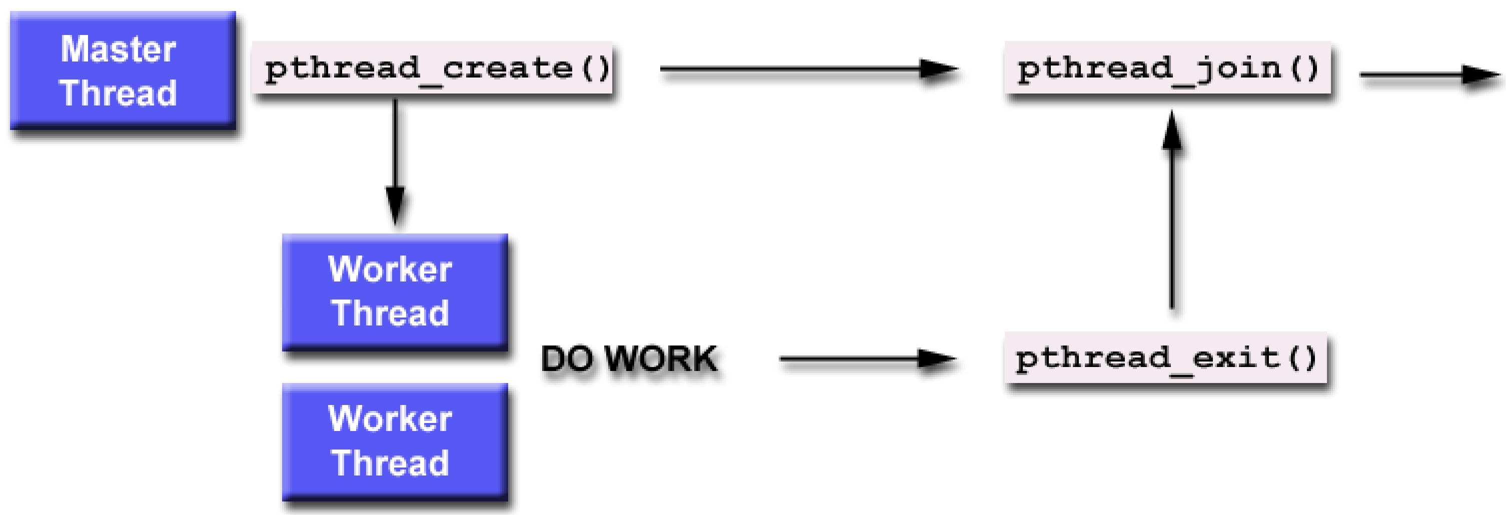
Thread Termination

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **status);
```

Returns 0 if OK, positive EXXX value on error

```
#include <pthread.h>
int pthread_detached(pthread_t tid);
```

pthread_join() subroutine blocks the calling thread until the specified thread terminates


Master Thread
pthread_create()
pthread_join()
Worker Thread
DO WORK
pthread_exit()
Worker Thread

pthread_self function

- Returns thread ID of the calling thread

```
#include <pthread.h>
pthread_t pthread_self(void);
```

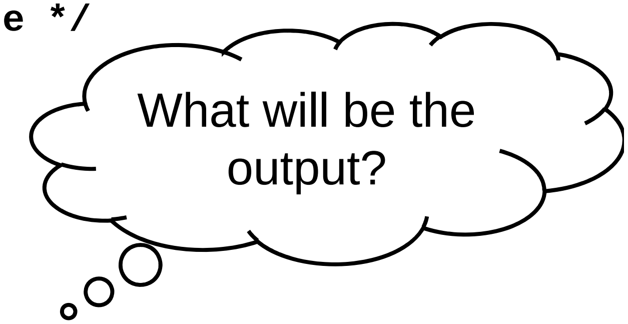
Mutexes

- Threads share global variables
  - Execution of threads are usually non-deterministic
- Demo example01.c

```
#include <pthread.h>
#define NLOOP 5000
int     counter;                /* incremented by threads */
void    *doit(void *);

int
main(int argc, char **argv)
{
        pthread_t       tidA, tidB;
        pthread_create(&tidA, NULL, &doit, NULL);
        pthread_create(&tidB, NULL, &doit, NULL);
                /* 4wait for both threads to terminate */
        pthread_join(tidA, NULL);
        pthread_join(tidB, NULL);
        printf("counter = %d\n", counter);
        exit(0);
}

void *doit(void *vptr)
{
        int             i, val;
        for (i = 0; i < NLOOP; i++) {
                val = counter;
                //printf("%d: %d\n", pthread_self(), val + 1);
                counter = val + 1;
        }
return(NULL);
}
```




```
gcc -lpthread -o example01  example01.c
```

Mutexes (cont'd)

```
#include <pthread.h>
int ptread_mutex_lock(pthread_mutex_t *mptr);
int ptread_mutex_unlock(pthread_mutex_t *mptr);

int pthread_mutex_init (pthread_mutex_t * mutex , pthread_mutexattr_t * attr );
```

- Blocked if trying to lock a mutex locked by some other thread

```
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
…
for (i = 0; i < NLOOP; i++) {
pthread_mutex_lock(&counter_mutex);
counter = counter++;
pthread_mutex_unlock(&counter_mutex);
}
```

Condition Variables

- Check whether a condition is met
- Allow threads to synchronize based upon the actual value of data (as opposed to a binary value)
- Often used in conjunction with mutex

```
#include <pthread.h>
int ptread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t
*mptr);
int ptread_cond_signal(pthread_mutex_t *cptr);

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_condattr_init ((pthread_condattr_t *attr);
```

Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

Main Thread

Join / Continue

Condition Variables

- Consider a web client that downloads multiple objects

```
int ndone; /* number of
terminated threads */
pthread_mutex_t
ndone_mutex =
PTHREAD_MUTEX_INITIALIZER;
```

```
void * do_get_read (void
*vptr) {
    ...
pthread_mutex_lock(&ndone_m
utex);
    ndone++;

pthread_mutex_unlock(&ndone
_mutex);
    return(fptr); /*
terminate thread */
}
```

```
while (nlefttoread > 0) {
 while (nconn < maxnconn && nlefttoconn
> 0)
 { /* find a file to read */
        ...
 } /* See if one of the threads is done
*/
  pthread_mutex_lock(&ndone_mutex);
  if (ndone > 0) {
     for (i = 0; i < nfiles; i++) {
         if (file[i].f_flags & F_DONE) {
            pthread_join(file[i].f_tid,
(void **) &fptr);
/* update file[i] for terminated thread
*/
               ...
         }
     }
  }
pthread_mutex_unlock(&ndone_mutex);
}
```

A Web Client Assignment

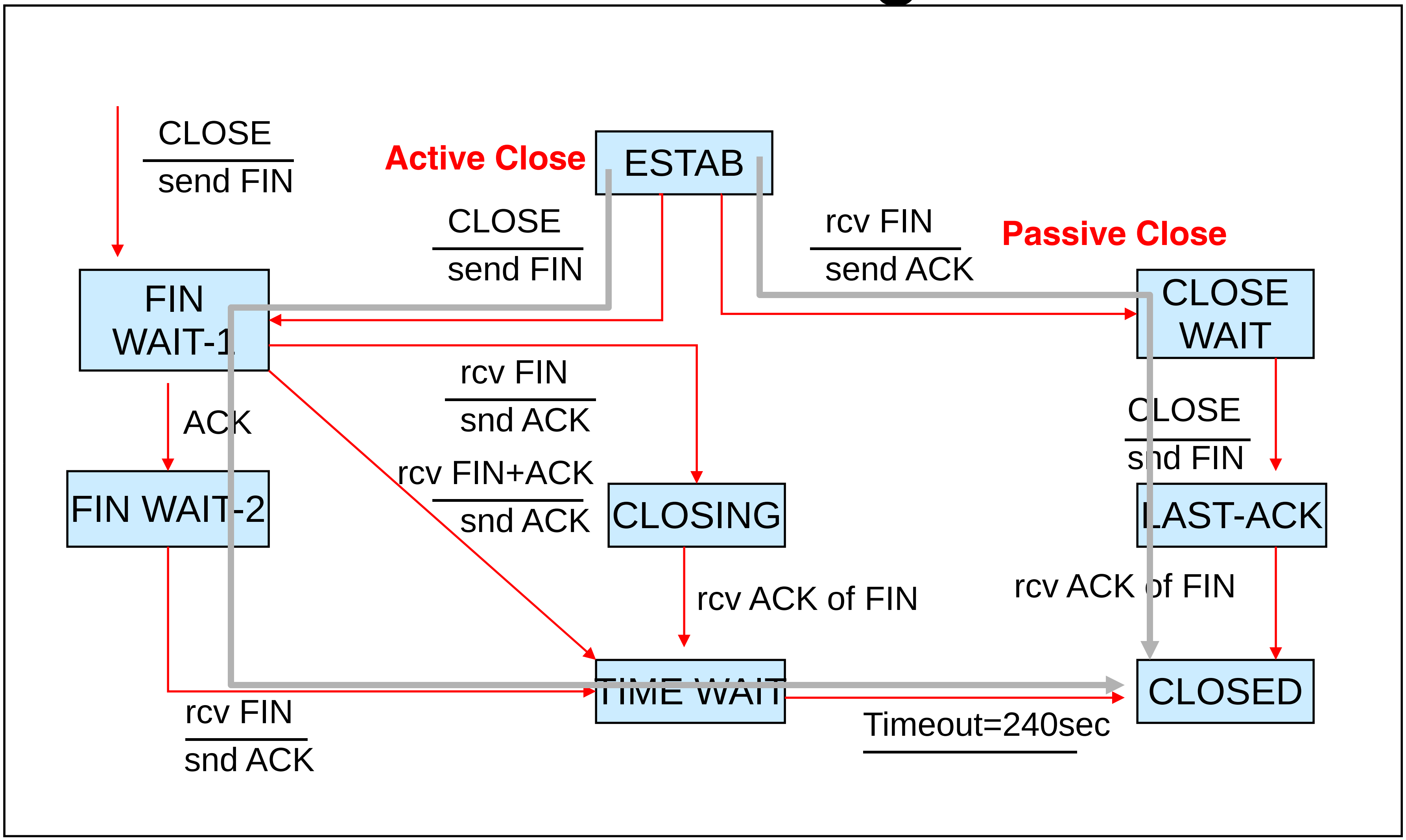
- The problem statement:
  - A web page typically contains multiple clients
  - Concurrent downloading can expedite user's experience

- telnet host port as a testing tool

Roadmap

- I/O multiplexing
- Socket options and why you cant bind to a port immediately

TCP state diagram



Socket Options

- Set and get socket options [see handout]
  - e.g., SO_REUSEADDR allows reuse of address and port
  - e.g., SO_KEEPALIVE allows TCP to automatically send *keep_alive* probe to its peer

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void
*optval, socklen_t *optlen);

int setsockopt(int s, int level, int optname, const void
*optval, socklen_t optlen);
```