

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=%2Frzab6%2Frzab6cmultiplex.htm

* Fill up a fd\_set structure with the file descriptors you want to know when data comes in on.

* Fill up a fd\_set structure with the file descriptors you want to know when you can write on.

* Call select() and block until something happens.

* Once select() returns, check to see if any of your file descriptors was the reason you woke up. If so, 'service' that file descriptor in whatever particular way your server needs to (i.e. read in a request for a Web page).

* Repeat this process forever.

**Host processor byte order:**

|  |  |
| --- | --- |
| **Host Processor** | **Endianness** |
| Intel x86 processor family | Little endian |
| Power PC processor family | Big endian |
| SUN SPARC | Big endian |
| Mips | Big endian (IRIX) |
| Mips | Little endian (NT) |

Note that it is the processor architecture which determines the endianness and NOT the OS. The exception is for processors which support both big and little endian byte ordering, such as the MIPS processor.

Also note: Java processes and stores data in big endian byte ordering on any platform.

Character data is not subject to this problem as each character is one byte in length but integer is.

Integer data can be converted from/to host or network byte order with the following routines:

|  |  |
| --- | --- |
| **Function** | **Description** |
| ntohl() | Network to host byte order conversion for long integer data (uint32\_t) |
| ntohs() | Network to host byte order conversion for short integer data (uint16\_t) |
| htonl() | Host to network byte order conversion for long integer data (uint32\_t) |
| htons() | Host to network byte order conversion for short integer data (uint16\_t) |

Requires #include <arpa/inet.h>

#include <[stdio.h](http://linux.die.net/include/stdio.h)>

#include <[stdlib.h](http://linux.die.net/include/stdlib.h)>

#include <[sys/time.h](http://linux.die.net/include/sys/time.h)>

#include <[sys/types.h](http://linux.die.net/include/sys/types.h)>

#include <[unistd.h](http://linux.die.net/include/unistd.h)>

int

main(void)

{

fd\_set rfds;

struct timeval tv;

int retval;

/\* Watch stdin (fd 0) to see when it has input. \*/

FD\_ZERO(&rfds);

FD\_SET(0, &rfds);

/\* Wait up to five seconds. \*/

tv.tv\_sec = 5;

tv.tv\_usec = 0;

retval = select(1, &rfds, NULL, NULL, &tv);

/\* Don't rely on the value of tv now! \*/

if (retval == -1)

perror("select()");

else if (retval)

printf("Data is available now.\n");

/\* FD\_ISSET(0, &rfds) will be true. \*/

else

printf("No data within five seconds.\n");

exit(EXIT\_SUCCESS);

}

**FD\_CLR(fd, &fdset)**

removes the socket descriptor **fd** from the socket descriptor set **fdset** .

 **FD\_ISSET(fd, &fdset)**

returns nonzero if socket descriptor **fd** is a member of **fdset** . Otherwise, it returns a **0** .

 **FD\_SET(fd, &fdset)**

adds socket descriptor **fd** to **fdset** .

 **FD\_SETSIZE**

is defined in **<sys/types.h>** as the number of socket descriptors that a process can have open. The default is 64. This value can be increased before you include **<sys/types.h>** . If **FD\_SETSIZE** is not increased, the **fd\_set** structure will not be defined to be large enough to accommodate socket numbers larger than 63. The default value for FD\_SETSIZE is 256.

 **FD\_ZERO(& fdset)**

initializes **fdset** to **0** , representing the empty set.

In this example **select** waits for data to read from a socket.

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <stdio.h>

/\* This function calls select to wait for data to read from \*/

/\* one of the sockets passed as a parameter. \*/

/\* If more than 3 seconds elapses, it returns. \*/

/\* Return value flags. These indicate the readiness of \*/

/\* each socket for read. \*/

#define S1READY 0x01

#define S2READY 0X02

waittoread(int s1,int s2)

{

fd\_set fds;

struct timeval timeout;

int rc, result;

/\* Set time limit. \*/

timeout.tv\_sec = 3;

timeout.tv\_usec = 0;

/\* Create a descriptor set containing our two sockets. \*/

FD\_ZERO(&fds);

FD\_SET(s1, &fds);

FD\_SET(s2, &fds);

rc = select(sizeof(fds)\*8, &fds, NULL, NULL, &timeout);

if (rc==-1) {

perror("select failed");

return -1;

}

result = 0;

if (rc > 0)

{

if (FD\_ISSET(s1, &fds)) result |= S1READY;

if (FD\_ISSET(s2, &fds)) result |= S2READY;

}

return result;

}

=======================

http://beej.us/guide/bgnet/output/html/multipage/advanced.html

/\*

\*\* selectserver.c -- a cheezy multiperson chat server

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#define PORT "9034" // port we're listening on

// get sockaddr, IPv4 or IPv6:

void \*get\_in\_addr(struct sockaddr \*sa)

{

if (sa->sa\_family == AF\_INET) {

return &(((struct sockaddr\_in\*)sa)->sin\_addr);

}

return &(((struct sockaddr\_in6\*)sa)->sin6\_addr);

}

int main(void)

{

fd\_set master; // master file descriptor list

fd\_set read\_fds; // temp file descriptor list for select()

int fdmax; // maximum file descriptor number

int listener; // listening socket descriptor

int newfd; // newly accept()ed socket descriptor

struct sockaddr\_storage remoteaddr; // client address

socklen\_t addrlen;

char buf[256]; // buffer for client data

int nbytes;

char remoteIP[INET6\_ADDRSTRLEN];

int yes=1; // for setsockopt() SO\_REUSEADDR, below

int i, j, rv;

struct addrinfo hints, \*ai, \*p;

FD\_ZERO(&master); // clear the master and temp sets

FD\_ZERO(&read\_fds);

// get us a socket and bind it

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_flags = AI\_PASSIVE;

if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {

fprintf(stderr, "selectserver: %s\n", gai\_strerror(rv));

exit(1);

}

for(p = ai; p != NULL; p = p->ai\_next) {

listener = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol);

if (listener < 0) {

continue;

}

// lose the pesky "address already in use" error message

setsockopt(listener, SOL\_SOCKET, SO\_REUSEADDR, &yes, sizeof(int));

if (bind(listener, p->ai\_addr, p->ai\_addrlen) < 0) {

close(listener);

continue;

}

break;

}

// if we got here, it means we didn't get bound

if (p == NULL) {

fprintf(stderr, "selectserver: failed to bind\n");

exit(2);

}

freeaddrinfo(ai); // all done with this

// listen

if (listen(listener, 10) == -1) {

perror("listen");

exit(3);

}

// add the listener to the master set

FD\_SET(listener, &master);

// keep track of the biggest file descriptor

fdmax = listener; // so far, it's this one

// main loop

for(;;) {

read\_fds = master; // copy it

if (select(fdmax+1, &read\_fds, NULL, NULL, NULL) == -1) {

perror("select");

exit(4);

}

// run through the existing connections looking for data to read

for(i = 0; i <= fdmax; i++) {

if (FD\_ISSET(i, &read\_fds)) { // we got one!!

if (i == listener) {

// handle new connections

addrlen = sizeof remoteaddr;

newfd = accept(listener,

(struct sockaddr \*)&remoteaddr,

&addrlen);

if (newfd == -1) {

perror("accept");

} else {

FD\_SET(newfd, &master); // add to master set

if (newfd > fdmax) { // keep track of the max

fdmax = newfd;

}

printf("selectserver: new connection from %s on "

"socket %d\n",

inet\_ntop(remoteaddr.ss\_family,

get\_in\_addr((struct sockaddr\*)&remoteaddr),

remoteIP, INET6\_ADDRSTRLEN),

newfd);

}

} else {

// handle data from a client

if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {

// got error or connection closed by client

if (nbytes == 0) {

// connection closed

printf("selectserver: socket %d hung up\n", i);

} else {

perror("recv");

}

close(i); // bye!

FD\_CLR(i, &master); // remove from master set

} else {

// we got some data from a client

for(j = 0; j <= fdmax; j++) {

// send to everyone!

if (FD\_ISSET(j, &master)) {

// except the listener and ourselves

if (j != listener && j != i) {

if (send(j, buf, nbytes, 0) == -1) {

perror("send");

}

}

}

}

}

} // END handle data from client

} // END got new incoming connection

} // END looping through file descriptors

} // END for(;;)--and you thought it would never end!

return 0;

}