I/O Multiplexing

Haviland 7.1.6

The problem

- When reading from multiple sources, blocking on one of the sources could be bad.
  - An example of denial of service.
- One solution: one process for every client. What are the pros and cons of this solution?

Another way to look at the problem

Server
while(1)
    accept a new connection
    for each existing connection
      read
      write

• Which of the system calls might block indefinitely?
  - read and accept
• So what happens if there is only one connection?

Blocking I/O Model

application
read
  system call
  no data ready
  wait for data
  copy data from kernel to user
  process data
  copy complete

kernel
read
  system call
  no data ready
  wait for data
  copy data from kernel to user
  process data
  copy complete

process blocks in a call to read

Data: "gone for coffee"

Client 1
read(c1)
blocked
read(c2)
write

Client 2
read(c2)
**Nonblocking I/O Model**

- Application: `read` system call
- Kernel: `no data ready`
- Process repeatedly calls `read` waiting for an OK (polling)

**Signal Driven I/O Model**

- Application: `read` system call
- Kernel: `no data ready`
- Process continues executing

**Asynchronous I/O Model**

- Application: `aio_read` system call
- Kernel: `no data ready`
- Process continues executing

**I/O Multiplexing Model**

- Application: `select` system call
- Kernel: `no data ready`
- Process blocks waiting for one of many fds
select()  

int select(int maxfdp1,
           fd_set *readset,
           fd_set *writeset,
           fd_set *exceptset,
           const struct timeval *timeout);

• A call to select returns when one of the file descriptors in one of the sets is ready for I/O.
• If timeout is not NULL, then select returns when a descriptor is ready or timeout time has passed.
• If timeout is 0, select returns immediately after checking descriptors.

select timeout

• The timeout specifies how long we're willing to wait for a fd to become ready

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

  – If timeout is NULL, wait forever (or until we catch a signal)
  – If timeout is zero, test and return immediately
  – Otherwise wait up to specified timeout
• select returns when a fd ready or we timeout

Readiness

• Ready to read when
  – there is data in the receive buffer to be read
  – end-of-file state on file descriptor
  – the socket is a listening socket and there is a connection pending
  – a socket error is pending
• Ready to write when
  – there is space available in the write buffer
  – a socket error is pending
• Exception condition pending when
  – TCP out-of-band data
• We are typically interested in when bytes are available to be read, but sometimes we use select on write or exception sets

Descriptor sets

• Typically implemented as an array of integers where each bit corresponds to a descriptor (except in Windows).
• Implementation is hidden in the fd_set data type
• FD_SETSIZE is the number of descriptors in the data type
• maxfdp1 specifies the number of descriptors to test
• 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);
Descriptor sets

client1 server client2

maxfd + 1 = 7

allset

fd0 fd1 fd2 fd3 fd4 fd5 fd6

After select:

rset

fd_set rfds;
struct timeval tv;
int retval;

FD_ZERO(&rfds); /* Watch stdin (fd 0) for input */
FD_SET(STDIN_FILENO, &rfds);
tv.tv_sec = 5; /* Wait up to five seconds. */
tv.tv_usec = 0;
retval = select(1, &rfds, NULL, NULL, &tv);
if (retval == -1)
  perror("select()");
else if (retval > 0)
  printf("Data is available now.
"); /* FD_ISSET(0, &rfds) will be true, can use read() */
else
  printf("No data within five seconds.
");

for( ; ; ) {
  nready = Select(maxfd+1, &rset, NULL, NULL, NULL);
  if(FD_ISSET(listenfd, &rset)) {
    connfd = Accept(listenfd, &caddr, &clen);
    for(i = 0; i < FD_SETSIZE; i++)
      if(client[i] < 0) {
        client[i] = connfd; break;
      }
    FD_SET(connfd, &allset);
    if(connfd > maxfd) maxfd = connfd;
  }
  for(i = 0; i <= maxi; i++) {
    if(sockfd = client[i]) < 0) continue;
    if(FD_ISSET(sockfd, &rset))
      Read(sockfd, line, MAXLINE);
  }
}

for( ; ; ) {
  nready = Select(maxfd+1, &rset, NULL, NULL, NULL);
  if(FD_ISSET(listenfd, &rset)) {
    connfd = Accept(listenfd, &caddr, &clen);
    for(i = 0; i < FD_SETSIZE; i++)
      if(client[i] < 0) {
        client[i] = connfd; break;
      }
    FD_SET(connfd, &allset);
    if(connfd > maxfd) maxfd = connfd;
  }
  for(i = 0; i <= maxi; i++) {
    if(sockfd = client[i]) < 0) continue;
    if(FD_ISSET(sockfd, &rset))
      Read(sockfd, line, MAXLINE);
  }
}
for( ; ; ) {
    rset = allset;
    nready = Select(maxfd+1, &rset,NULL,NULL,NULL);
    if(FD_ISSET(listenfd, &rset)) {
        connfd = Accept(listenfd, &caddr, &clen);
        for(i = 0; i < FD_SETSIZE; i++)
            if(client[i] < 0) {
                client[i] = connfd; break;
            }
        FD_SET(connfd, &allset);
        if(connfd > maxfd) maxfd = connfd;
    }
    for(i = 0; i <= maxi; i++) {
        sockfd = client[i];
        if(sockfd < 0) continue;
        if(FD_ISSET(sockfd, &rset))
            Read(sockfd, line, MAXLINE);
    }
}