- Experiment with open and close.
- The code carefully handles errors and interruption by signals.

SELECT

The handling of I/O from multiple sources is an important problem that arises in many different forms.

Ex: Program expects input from two different sources, but it doesn't know which input will be available first.

Solution: Need to block until input from either source becomes available.

Blocking until at least one member of a set of conditions becomes true is called *OR synchronization.*

One method of monitoring multiple file descriptors is to use a separate process for each one

SELECT

Q: Write a program that accept two command-line arguments, i.e. names of two files.

The parent process opens both files before creating the child process. The parent monitors the first file descriptor, and the child monitors the second. Each process echoes the contents of its file to standard output.

SELECT

```
#define BLKSIZE 10
int display(int x,int y, char *buf)
{int r;
 r=read(x, buf, BLKSIZE);
 return r;
}
int main(int argc, char *argv[])
{ char buf[BLKSIZE];
int bytesread,fd,fd1,fd2;
pid_t childpid;
fd1=open(argv[1],O_RDONLY);
fd2=open(argv[2],O_RDONLY);
if(fd1==-1){
   fprintf(stderr,"Failed to open
%s:%s",argv[1],strerror(errno));
   return 1;
   }
if(fd2==-1){
   fprintf(stderr,"Failed to open
%s:%s",argv[1],strerror(errno));
   return 1;
   }
```

```
childpid=fork();
if(childpid > 0)
  fd=fd1;
else
  fd=fd2;
bytesread=display(fd,STDOUT_FILENO,buf);
fprintf(stderr,"\n%d bytes read
%s",bytesread,buf);
return 0;
```

SELECT

// reading of two files given at command line using fork

```
int main(int argc,char **argv){
  int fd,fd1,fd2;
  char buf[20];
  pid_t rid;
  ssize_t read_val;
  fd1=open(argv[1],O_RDONLY);
  printf("1st FD is=%d\n",fd1);
  fd2=open(argv[2],O_RDONLY);
  printf("2nd FD is=%d\n",fd2);
  if(fd1==-1 || fd2==-1){
     perror("Error in opening");
     return -1;
  }
rid=fork();
if(rid > 0)
  fd=fd1;
else
  fd=fd2;
read_val = read(fd,buf,10);
write(1,buf,read_val);
close(fd1);
close(fd2);
return 0;
}
```

SELECT

select call provides a method of monitoring file descriptors from a single process.

It can monitor for three possible conditions—

a read can be done without blocking,

a write can be done without blocking, or

a file descriptor has error conditions pending.

Older versions of UNIX defined the select function in sys/time.h, but the POSIX standard now uses sys/select.h.

SYNOPSIS

```
#include <sys/select.h>

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

Four macros FD_SET, FD_CLR, FD_ISSET and FD_ZERO are used to manipulate the descriptor sets in an implementation-independent way

SELECT

nfds : gives the range of file descriptors to be monitored. Its value must be at least one greater than the largest file descriptor to be checked.

readfds : specifies the set of descriptors to be monitored for reading.

writefds : specifies the set of descriptors to be monitored for writing,

errorfds : specifies the file descriptors to be monitored for error conditions.

timeout value : it forces return from select() after a certain period of time has elapsed, even if no descriptors are ready.

When timeout is NULL, select may block indefinitely.

Any of these parameters may be NULL, in which case select does not monitor the descriptor for the corresponding event.

fd_set : type of descriptor .

select: The First Argument

The first argument to select specifies the number of descriptors to be tested.

- Its value is the maximum descriptor to be tested plus one.

- The descriptors 0, 1, 2, up through and including last one are to be tested.

For example, If indicators for descriptors 1, 4, and 5 are turn on the , then first argument value of select will be value 6.

- The reason it is 6 and not 5 is that we are specifying the number of descriptors, not the largest value, and descriptors start at 0.

so, select(6, ---,---,--, ---);

three middle arguments

The three middle arguments, readfds, writefds, and errorfds, specify the descriptors that we want the kernel to test for reading, writing, and exceptions.

- Let readfds to be tested;

```
fd_set readfds;
select(---, &readfds, ---, ---, ---);
```

- Let writefds to be tested;

```
fd_set writefds;
select(---, ---, &writefds, ---, ---);
```

- Let both readfds and writefds to be tested;

```
fd_set writefds;
select(---, &readfds, &writefds, ---, ---);
```

- Let errorfds not to be tested;

```
select(---, ---,---, NULL, ---);
```

- Let readfds to be tested and right to others not

```
fd_set readfds; int maxfdp1;
select(maxfdp1, &readfds, NULL, NULL, NULL);
```

- Simillarly set the descriptor sets on requirement.

Macros

Returns :
If successful : the number of file descriptors that are ready.
If unsuccessful : –1 and sets errno

The descriptor sets are now usually represented by bit fields in arrays of integers. To manipulate the descriptor sets it use macros

- FD_SET : sets the bit in *fdset corresponding to the fd file descriptor :
  void FD_SET(int fd, fd_set *fdset);
- FD_CLR : clears the corresponding bit.
  void FD_CLR(int fd, fd_set *fdset);
- FD_ZERO : clears all the bits in *fdset (use before select)
  void FD_ZERO(fd_set *fdset);
- FD_ISSET : test for the file descriptor fd is set (use after select)
  int FD_ISSET(int fd, fd_set *fdset);

fd_set data type

The file descriptor sets for the select function are specified as fd_set objects.

- The fd_set data type represents file descriptor sets for the select function. It is actually a bit array.

-So, allocate a descriptor set of the fd_set data type.

Example: To define a variable of type fd_set is fd_set rset.

- The four macros are used to set and test the bits in the set.

- Like ordinary variable assignment, we can also assign it to another descriptor set across an equals sign (=).

Example : Define a variable of type fd_set and turn on the bits for descriptors 1, 4, 5

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

Last argument

The timeout argument tells the kernel how long to wait for one of the specified descriptors to become ready.

- A timeval structure defined in <time.h> specifies the number of seconds and microseconds.

-The predefined timeval structure:

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

There are three possibilities:

- Wait forever: set the timeout argument as NULL.
- Wait up to a fixed amount of time:

```
struct timeval tv
tv.tv_sec=5; /* seconds */
v. tv_usec=0; /* microseconds */
select(----, ----, ----, ----, &tv);
```

- Do not wait at all: set tv_sec and tv_usec to Zero.

```
select(---, ---, ---, ---, &tv);
```

SELECT

Let us consider that select() to monitor readfds, and writefds file descriptors.

call to select:

```
fd_set readfds;  /* reading fds et*/
fd_set writefds;  /* writing fd set */
int nfds;
...
...
select(nfds, &readfds, &writefds, NULL, NULL);
```

Program Using Select

// Use of select to get input from standard input within 5 sec

```
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<errno.h>
#include<sys/select.h>
int main(void)
{
fd_set rfds;
struct timeval tv;
int retval;
FD_ZERO(&rfds);
/* Watch stdin to see when it has input. */
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");
else
  printf("No data within five seconds.\n");
return 0;
}
```

Write whichready() function that blocks until one of two file descriptor is ready

```
int whichready(int fd1,int fd2)
{
fd_set readset;
int maxfd,nfds;
maxfd=((fd1>fd2)? fd1 : fd2);
FD_ZERO(&readset);
FD_SET(fd1,&readset);
FD_SET(fd2,&readset);
nfds = select(maxfd+1, &readset, NULL, NULL,
NULL);
//printf("nfds=%d\n",nfds);
if (nfds == -1)
        return -1;
if (FD_ISSET(fd1, &readset))
        return fd1;
if (FD_ISSET(fd2, &readset))
        return fd2;
//errno = EINVAL;
return -1;
}
```

```
int main(void)
{
int fd1,fd2,fd;
ssize_t read_val;
char buf[20];
fd1=open("doc1.txt",O_RDONLY);
fd2=open("info.txt",O_RDONLY);
//printf("FD1=%d\tFd2=%d\n",fd1,fd2);

fd=whichready(fd1,fd2);
//printf("FD Retun=%d",fd);
read_val=read(fd,buf,1);
while(read_val!=0){
  write(1,buf,read_val);
  read_val = read(fd,buf,1);
        }
close(fd1);
close(fd2);
}
```

SELECT

whichready() function is problematic

because it always chooses fd1 if both fd1 and fd2 are ready

function that uses select to do two concurrent file copies.

```
int copy2files(int fd1,int fd2,int fd3,int fd4)
{
fd_set readset;
int maxfd,nfds,byteread,num; int totalbyte=0;
maxfd=fd1;
if(fd3>maxfd)
  maxfd=fd3;
for(;;){      FD_ZERO(&readset);
              FD_SET(fd1,&readset);
              FD_SET(fd3,&readset);
              if((num=select(maxfd+1,&readset,NULL,NULL,NULL)) == -1 )
                continue;
              if(num==-1)
                return totalbyte;
              if(FD_ISSET(fd1,&readset)){
                byteread=readwrite(fd1,fd2);
                if (byteread<=0)
                break;
              totalbyte+=byteread;
              }
              if(FD_ISSET(fd3,&readset)){
                byteread=readwrite(fd3,fd4);
              if (byteread<=0)
                 break;
              totalbyte+=byteread;
              }
}
return totalbyte;
}
```

function that uses select to do two concurrent file copies.

```
int readwrite(int fromfd, int tofd) {
char *bp; char buf[BLKSIZE];
int bytesread, byteswritten; int totalbytes = 0;
for ( ; ; ) {
   while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) && (errno == EINTR)) ;
    if (bytesread <= 0) /* real error or end-of-file on fromfd */
              break;
   bp = buf;
   while (bytesread > 0) {
     while(((byteswritten = write(tofd, bp, bytesread)) == -1 ) && (errno == EINTR)) ;
      if (byteswritten <= 0) /* real error on tofd */
        break;
      totalbytes += byteswritten;
      bytesread -= byteswritten;
      bp += byteswritten;
   }
   if (byteswritten == -1) /* real error on tofd */
     break;
}
return totalbytes;
}
```

```
int main(void)
{
int f1,t1,f2,t2,rt;
int tt;
ssize_t read_val;
char buf[20];
mode_t mode=S_IRWXU;
f1=open("doc1.txt",O_RDONLY);
f2=open("doc2.txt",O_RDONLY);
t1=open("select11.txt", O_CREAT | O_RDWR,mode);
t2=open("select22.txt", O_CREAT | O_RDWR,mode );
printf("FR1=%d\tTO1=%d\tFR2=%d\tTO2=%d\n",f1,t1,f2,t2);
tt=copy2files(f1,t1,f2,t2);
printf("Total = %d\n",tt);
close(f1);
close(f2);
close(t1);
close(t2);
}
```

SELECT

It can be generalized to monitor multiple file descriptors for input.

Such a problem might be encountered by a command processor that was monitoring requests from different terminals.

The program cannot predict which source will produce the next input, so it must use a method such as select.

In addition, the set of monitored descriptors is dynamic—the program must remove a source from the monitoring set if an error condition arises on that source's descriptor.

//Function to Monitor array of file descriptor

```
void monitorselect(int fd[], int numfds) {
        char buf[BUFSIZE];
        int bytesread, i,maxfd,numnow, numready;
        fd_set readset;
        maxfd = 0; /* set up the range of descriptors to monitor */
        for (i = 0; i < numfds; i++) {
                if ((fd[i] < 0) || (fd[i] >= FD_SETSIZE))
                        return;
                if (fd[i] >= maxfd)
                        maxfd = fd[i] + 1;
        }
        numnow = numfds;
```

SELECT

```
while (numnow > 0)
        { /* continue monitoring until all are done */
                FD_ZERO(&readset); /* set up the file descriptor mask */
                for (i = 0; i < numfds; i++)
                  if (fd[i] >= 0)
                        FD_SET(fd[i], &readset);
                numready = select(maxfd, &readset, NULL, NULL, NULL); /* which ready? */
                if ((numready == -1) && (errno == EINTR)) /* interrupted by signal */
                        continue;
                else if (numready == -1) /* real select error */
                        break;
                for (i = 0; (i < numfds) && (numready > 0); i++)
                 { /* read and process */
```

SELECT

```
if (fd[i] == -1) /* this descriptor is done */
            continue;
             if (FD_ISSET(fd[i], &readset))
                 { /* this descriptor is ready */
                        bytesread = read(fd[i], buf, BUFSIZE);
                        numready--;
                        if (bytesread > 0)
                                    docommand(buf, bytesread);
                        else { /* error occurred on this descriptor, close it */
                                    close(fd[i]);
                                    fd[i] = -1;
                                    numnow--;
                                    }
                        }
            }
}
for (i = 0; i < numfds; i++)
    if (fd[i] >= 0)
                close(fd[i]);
}
```

```
void docommand(char *buf, int by){
printf("Buffer = %s",buf);
printf("Bytes = %d",by);
}
```