Chapter 13

Concurrent Single Process Server

Motivation: If little processing is required per request: concurrent servers often behave in a sequential manner So multiple processes (timesharing) doesn't help.
Solution: Single-process, concurrent server.
Server watches all connections and handles those with requests.
Less context switching $\Rightarrow$ less CPU

Idea: Use asynchronous I/O to provide apparent concurrency among clients.

Implementation: Have a single server process keep TCP connections open to multiple clients.

ALGORITHM 8.5

1. Get the master socket.
2. Use select to wait for input on all open sockets.
3. Arrival on master socket, open a slave socket.
4. Arrival on slave socket, read/write (provide service)
5. Repeat 2–4

Select Utilities

`fd` is an integer (`int`). It is an index into the descriptor table. This integer array of pointers to file information structures.

`fd_set` is a set of integers.
implemented by a bit vector (array of Booleans).

`FD_SET(fd, &fdset)`: adds a file descriptor (`fd`) to the set of file descriptors `fdset`.

`FD_CLR(fd, &fdset)`: removed file descriptor from the set of file descriptors.

`FD_ISSET(fd, &fdset)`: returns TRUE if file descriptor is in the set of file descriptors.

`FD_ZERO(&fdset)`: the set of file descriptors is set to the empty set

`select(nfds,rfds,wfds,efds,time)`: Consider only the descriptors `0..nfds-1`. For each descriptor in the set `*rfds`; if there is input on that descriptor (something to read), leave the descriptor in the set, otherwise, remove it.
`rfds` before: we want to know if any of these have input waiting; after: those that have input waiting.

Select Example

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/file.h>
main (){
  int fd1, fd2; /* two file descriptors */
  fd_set file_set;
  int count;
  fd1 = open("p1.c", 0, O_RDONLY);
  fd2 = open("p2.c", 0, O_RDONLY);
  FD_ZERO(&file_set);
  FD_SET(fd1, &file_set);
  FD_SET(fd2, &file_set);
  FD_SET(0, &file_set); /* stdin */
  count = select(5, &file_set, NULL, NULL, NULL);
  printf("%d, %d\n", fd1, fd2);
  printf("%d\n",count);
  close(fd1);
  close(fd2);
}
3 4
2
```

Examines 0,3,4, finds that 3 and 4 can be read

If something is typed: reports "3" instead of 2

Note: `file_set` is modified by `select`

Select

```
ret=select(nfds, rfds, wfds, efds, time);
```

`nfds` number of file descriptors

`rfds` check this set for input.
`wfds` check this set to see if output is allowed.
`efds` check this set for errors.
The above are modified: only those with input or output remain set (others become cleared).

`time` if there is no input (/output), wait this amount of time for some to occur before returning 0.

Warning: use a copy of the "`fd_set`" when calling `select` (because it gets changed).

Both the textbook entry for select (Appendix 1) and "man select" need to be read carefully.

Detailed analysis

see Section 13.5 for code.

```
int main(int argc, char* argv[]){
  char *service="echo";
  struct sockaddr_in fsin;
  int msock;
  fd_set rfds;
  fd_set afds;
  unsigned int alen;
  int fd,nfds;
  switch (argc) {
   ...
  }
```

`rfds`: The set of descriptors that have input pending
`afds`: The set of all open (active) descriptors
`nfds`: The number of file descriptors Unix allows this process

All other code here, including the switch is identical to previous servers.

```
msock=passiveTCP(service,QLEN);
nfds=getdtablesize();
FD_ZERO(&afds);
FD_SET(msock,&afds);
while(1) {
```

`getdtablesize`: this Unix system call returns the maximum number of descriptors in the descriptor table. They will be numbered 0..nfds-1. The loops generally go through all possible descriptors. The loops would be more efficient if we kept track of the highest number descriptor in use, but the code would be more complex.

`FD_ZERO`: Initialized the set of active file descriptors to be empty,

`FD_SET`: Initially the master socket (only) is open, so we put it into the descriptor set.

The service loop is standard `while(1)`

Inside service loop 1

```
memcpy(&rfds,&afds,sizeof(rfds));
if (select(nfds,&rfds,NULL,NULL,NULL)<0)
  errexit(...);
```

`memcpy`: what is in `afds` is copied into `rfds`. Because `select` modifies the sets it is passed we need to pass a copy and not the original to it.

`select`: The program waits here until there is input on at least one descriptor in the `rfds` set.

The last parameter is `NULL` indicating that this wait will never time out.

The number of discriptors with input waiting (i.e., what select returns) is ignored unless it indicates an error.

The set `rfds` will contain those active descriptors with input waiting.

Inside service loop 2

```
if (FD_ISSET(msock,&rfds)) {
  int ssock;
  alen=sizeof(fsin);
  ssock=accept(msock,
    (struct sockaddr *)&fsin,&alen);
  if (ssock<0)
    errexit(...);
  FD_SET(ssock,&afds);
}
```

The first check is to see if the master socket is set.

Activity on the `msock` descriptor means someone has requested a connect.
Activity on other descriptors means someone already connected has sent a message.

`if (FD_ISSET(msock...`: if a connect request is waiting, do the standard accept and
use (`FD_SET`) to add the new socket descriptor to the list of active descriptors.

Because there may be more than one descriptor active, we need to continue with the tests. We are not yet ready to go back to the accept.

Inside service loop 3

```
for (fd=0;fd<nfds;++fd)
  if (fd!=msock && FD_ISSET(fd,&rfds))
    if (echo(fd)==0) {
      (void) close(fd);
      FD_CLR(fd,&afds);
    }
```

If there is activity on any of the slave sockets, it is an echo request. In this case we do the echo.

The `for` goes through all possible descriptors
`if`:
`fd!=msock`: do not consider the master socket, that was already handled. `FD_SET(fd,&rfds)`: Does the (slave) socket have input waiting.

`echo(fd)`: do an echo on each slave descriptor with input waiting.

`if (echo...` the recall will return 0 if it got EOF. An EOF means the client has exited; so we won't be talking to that client again.
So we `close` the descriptor and remove (`FD_CLR`) the descriptor from the active set (Notice the `afds` here!)

Detailed descripton of the `echo` procedure:

```
int echo(int fd){
  char buf[BUFSIZ];
  int cc;
  cc=read(fd,buf,sizeof buf);
  if (cc<0) errexit(...);
  if (cc && write(fd,buf,cc)<0)
    errexit(...);
  return cc;
}
```

`buf`: a real buffer to read into.

`BUFSIZ`: he meant `BUFSIZE`; but it happens a system variable (without the E) has the value `4096`.

`cc=read`: we know input is waiting, read it. Three cases.

`cc < 0`: A read error has occurred. The error exit is bad here. One bad read from a crashed client brings down the whole server. A `return 0;` would be better, that would cause the socket to be closed and the entry for the crashed client to be removed from the `afds` list.

`cc == 0`: means a normal eof, that is, the client did a normal shutdown. The code will cause us to return this value. A return of 0, makes the main program close the socket and remove this clients entry from the `afds`.

`cc > 0`: In this case this is the number of bytes read we want do do an echo of what we have just read.

`if (cc && write...`: if `cc` is `> 0` then do the write if `write` fails (`< 0`)), crash the server. This is bad, a `return 0;` would be better here.

`return cc`: In the case of eof `cc` is 0, `cc` is the number bytes read (which is a positive number)

Fairness/Behavior

If input is waiting from several clients:

The `for` loop will process one request from each client,

No process can be blocked by other processes.

No matter how many requests a client has pending, the server will only process one request per iteration of the while;

It will also process one request from every other client that has requests waiting.

Sending lots of requests will not slow down other clients.

If several connect requests are pending, it will take several rounds of the while loop.

Sockets and errors

Old style: return `-1`

New style: for some errors, return `-1`
for reads/writes to a broken/closed socket (or pipe)
raise `SIGPIPE`.

Error handling choices.

1) Write a handler for `SIGPIPE`

```
void handler(int x){...}
main(){
  signal(SIGPIPE,handler);
  ...
}
```

2) ignore the signal

```
signal(SIGPIPE,SIG_IGN);
```

3) Use `send`/`recv` in a way that `SIGPIPE` isn't raised.

send and recv

send–similar to write
recv–similar to read

Both have one extra parameter for flags:
The possible flags are:

`MSG_OOB`: process out-of-band data

`MSG_DONTROUTE`: send direct, ignore routing

`MSG_DONTWAIT`: send normally copies the contents of the message to the system send buffer before proceeding. If the contents need time to send, the process blocks.
Don't block, return the special value `EAGAIN` (`errno`)
A future send will overwrite the buffer losing data.
useful with `select`'s option for "ready to send"

`MSG_NOSIGNAL`: don't raise SIGPIPE

Example:

```
write(sock,buf,sizeof(buf)};
/* With send and flags could become */
send(sock,buf,sizeof(buf),MSG_NOSIGNAL);
```