Chapter 15
Multiple Service Servers

Outline:

Motivation

Iterative, Connectionless Server.

Concurrent, Connection-Oriented Server.

Single-Process Server

Flexibility of Invoking Separate Programs

Multiservice, Multiprotocol Designs

Example Multiservice Server

BSD super server: inetd

Motivation:

Consolidation of servers reduces:
–the number of executing processes
–the total code required

Principle: use a separate socket (and port) for each service, but have one program handle them all.

Otherwise similar to the corresponding single service Server.

Iterative, Connectionless, Multiservice Server

1 Opens a set of UDP sockets.

2 Binds each one to a well-known port for one of the services being offered.

3 A table maps services to sockets (for each descriptor, the table records the address of a procedure to call for service).

4 Call select to wait for a datagram to arrive.

5 Examine the descriptor and call the appropriate service procedure.

6 Return to the select call.

Iterative is good for handling short, non-interactive services.

Iterative, Connection-Oriented, Multiservice Server

1 Opens one socket for each service and binds each one to the appropriate well-known port.

2 A table maps services to sockets (for each descriptor, the table records the address of a procedure

3 Calls select to wait for an incoming connection request.

4 When a request arrives, calls accept to create a new socket for the incoming connection.

5 Uses the new socket to interact with the client, then closes the socket.

6 Return to the select call.

Concurrent, Connection-Oriented, Single-Process Multiservice Server

1 Opens one master socket for each service and binds each one to the appropriate well-known port.

2 A table maps services to sockets (for each descriptor, the table records the address of a procedure

3 Calls select to wait for incoming connection requests master sockets or incoming data to read on slave sockets.

4a For connect requests, calls `accept` to create a new socket for the incoming connection.

4b For incoming data, calls the appropriate procedure to provide service.

5 Return to the select call.

Concurrent, Connection-Oriented, Multi-Process Multiservice Server

1 Opens one master socket for each service and binds each one to the appropriate well-known port.

2 A table maps services to sockets (for each descriptor, the table records the address of a procedure

3 Calls select to wait for incoming connection requests.

4 Creates a slave processes for each new connection.

5 Each slave handles service for client.

Software Design

Break code into independent components using separately compliled programs.
Compile those modules which change and relink.

Multi-process Servers: `fork`/`exec` allow call of a separately compiled (and maintained program).

Example Multiservice Server

see Section 15.9 for code

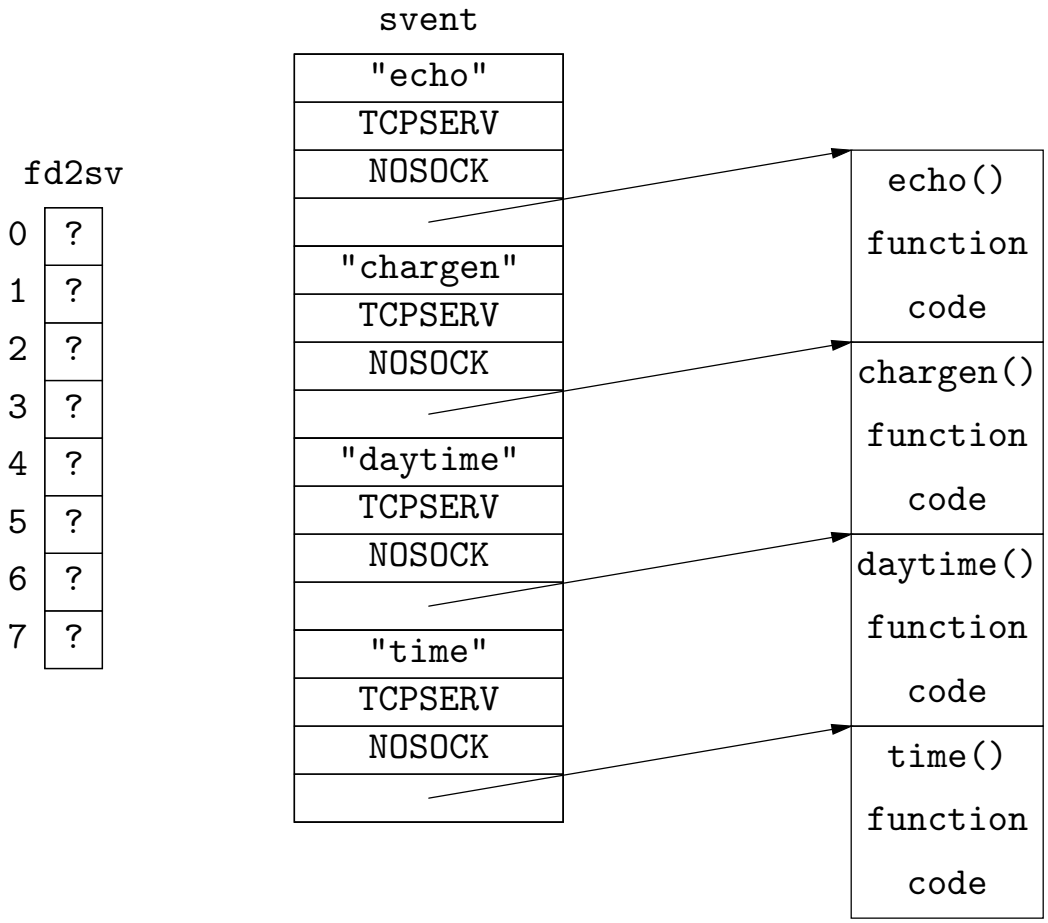
Four services: echo, chargen, daytime, and time

```
struct service{
  char *sv_name;      /* ASCII name of service */
  char sv_useTCP;     /* 0=UDP 1=TCP           */
  int sv_sock;        /* socket descriptor     */
  int (*sv_func)(int);/* function to call      */
}
```

`svent` is an array of 4 of these
Each entry is used to store the information about one of the services.
C note: `sv_func` stores the address of a function.
function calls are by address
`svent[3]->sv_func(4)` will call the function whose address is found in the 4th record (`TCPtimed`), and ask it to use the socket found in descriptor 4.
`svent[3]->sv_sock` is the master socket for the service

Array `fd2sv` to map from the descriptor back to an entry in svent (reverse map).

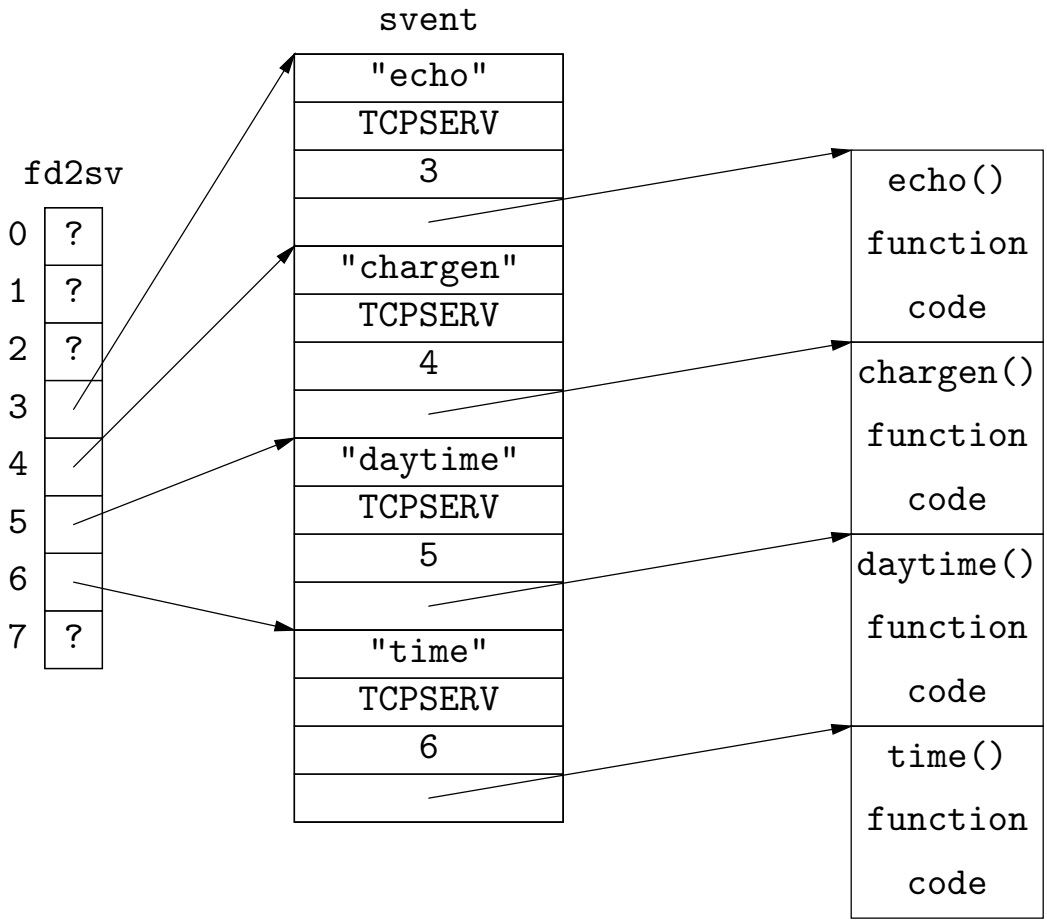
Structure Before socket allocation



Strings are for human interface purposes and for use with the `passiveUDP` and `passiveTCP` functions.

No sockets are entered (yet)
0,1,2: are `stdin`, `stdout` and `stderr` and will be unused

Structure after socket allocation



Section 15.9 code details

```
#define  UDP_SERV  0
#define  TCP_SERV  1
#define  NOSOCK -1 /*an invalid socket descriptor*/
struct service {
  char  *sv_name;
  char  sv_useTCP;
  int   sv_sock;
  int   (*sv_func)(int);
};
struct service svent[] = {
  {"echo",TCP_SERV,NOSOCK,TCPechod },
  {"chargen",TCP_SERV,NOSOCK,TCPchargend },
  {"daytime",TCP_SERV,NOSOCK,TCPdaytimed },
  {"time",TCP_SERV,NOSOCK,TCPtimed },
  {0, 0, 0, 0 }
};
```

The defines set up values to go into the structure.

The structure is declared and initialized to have everything but the descriptor (socket) number.

```
int main(int argc, char *argv[]) {
  struct service  *psv,    /*service table pointer*/
    *fd2sv[NOFILE];    /*map fd to service pointer*/
  int  fd, nfds;
  fd_set  afds, rfds;
  nfds = getdtablesize();
  FD_ZERO(&afds);
  for (psv = &svent[0]; psv->sv_name; ++psv) {
    if (psv->sv_useTCP)
      psv->sv_sock = passiveTCP(psv->sv_name, QLEN);
    else
      psv->sv_sock = passiveUDP(psv->sv_name);
    fd2sv[psv->sv_sock] = psv;
    FD_SET(psv->sv_sock, &afds);
  }
```

`for`: go through the array of services and fill in the socket numbers.
a 0 for the name indicates the end of the array

Open either a TCP master socket or
a UDP socket for each service.

Also, fill in the reverse lookup array `fd2sv`

```
while (1) {
  memcpy(&rfds, &afds, sizeof(rfds));
  if (select(nfds, &rfds, NULL, NULL, NULL) < 0) {
    if (errno == EINTR) continue;
    errexit(...);
  }
```

`while(1)` the main service loop,

`memcpy...select`: wait for a connect/message on any of the ports we have open.

`EINTR`: Falling out the select because the program has received a `SIGCHILD` signal is handled by going back around the while loop to the `select`. This is necessary since we have `fork` and a `reaper`.

```
  for (fd=0; fd<nfds; ++fd)
    if (FD_ISSET(fd, &rfds)) {
      psv = fd2sv[fd];
      if (psv->sv_useTCP)
        doTCP(psv);
      else
        psv->sv_func(psv->sv_sock);
    }
}
```

`for` process each connect/message.
This is the real service inside the service loop.
Use the reverse map (`fd2sv`) to find the service entry information
If the service entry indicates this is a UDP function, just call the service function passing the socket.

If the service entry indicates this is a TCP function, call `doTCP` because TCP needs additional setup before calling the service procedure

```
void doTCP(struct service *psv) {
  struct sockaddr_in fsin;
  int  alen, ssock;
  alen = sizeof(fsin);
  ssock=accept(psv->sv_sock,
    (struct sockaddr *)&fsin, &alen);
  if (ssock < 0) errexit(...);
  switch (fork()) {
  case 0:  break;
  case -1: errexit(...);
  default: (void) close(ssock);
    return;    /* parent */
  }
  exit(psv->sv_func(ssock)); /* child */
}
```

This is the special TCP setup.

do an `accept` to set up the slave socket

fork a child that calls the service function
when the service function returns, the child exits

`return` takes the parent back to the main program, which for-loops to the next `rfds` needing service.

The service function must be correctly programmed.

For TCP, the service function is passed the slave socket on which it must do any necessary reads and writes.

The `accept` has been done, the `close` is not required because of the `exit`

For UDP, the service function must do a `recvfrom` so has an address to use for the `sendto`.

When done, the service function should return; which will trigger the exit.

inetd

BSD UNIX super server

Supplies many small TCP/IP services

Services can be added to (deleted from) inetd using a configuration file

`/etc/inetd.conf` the config file

Service name–such as `telnet`

Socket type–`dgram`/`stream`

Service type–`udp`/`tcp`

wait/nowait–single process/multiprocess (fork)

user–privilege level (actually user name)

service procedure–name of the executable file, internal indicates a tiny service like time provided directly by inetd.

display name–what to show to `ps`

Only one server is running waiting for many things.

Specific servers are not started unless needed.