Today

- inetd
- finger/fingerd

UNIX startup

- boot bootstrap code
- load **/vmunix**
- mount root file system and /usr
- start **init**
- **/etc/rc.boot** sets host name, **ifconfig** network interfaces (associate host IP address with interface), and check disks (**fsck**)
- **rc.local** starts portmapper, yp, nfs, mail, syslogd, mountd, and xntpd
- **rc** starts cron, **inetd**, and lpd
- Sys V, AIX, OSF, linux use rc.d/ files

inetd

the mother of all network daemons

- opens ports on selected services (/etc/inetd.conf)
- fork/exec's servers when requests arrive
- handles some simple services internally
- reduce process load (memory, swap space)
- simplifies servers (inetd daemonizes)
- run-time reconfiguration (SIGHUP)
- security extensions tcpwrappers/xinetd
- disadvantages

  overhead of fork/exec
  system mgr. must install new servers

- illustrates **fork()/exec(), select(), signal(), setuid()**,iterative and concurrent servers, critical region, resiliency

function summary

main() daemonize, init, config, and loop on select() (fork()/exec() servers)

config() read **inetd.conf** and open/close server ports using **setup()**; also initiated by SIGHUP

***conf*()** routines for parsing config file entries

setup() create socket **socket(), bind(), listen()** for a config file entry

builtin servers

inetd.conf

```
#service type proto     wait    uid     program   args
ftp     stream  tcp     nowait  root    /usr/etc/in.ftpd
   in.ftpd
talk    dgram   udp     wait    root    /usr/etc/in.talkd
   in.talkd
tftp    dgram   udp     wait    root    /usr/etc/in.tftpd
   in.tftpd -s /tftpboot
daytime stream  tcp     nowait  root    internal
daytime dgram   udp     wait    root    internal
sprayd/1        dgram   rpc/udp wait root /usr/etc/rpc.sprayd
   rpc.sprayd
```

nowait except for single-threaded datagram servers

wait/nowait

nowait (for TCP servers)

- inetd does not wait for server to terminate before accepting another connection request
- server gets a new sockfd from the accept

wait (UDP)

- inetd must wait for child to terminate
- saves pid for use by *waitpid()*
- disables the socket from future selects (avoid race condition with server)
- child terminates, SIGCHLD checks pid, reenables select socket

server table

```
struct   servtab {
   char  *se_service;      /* name of service */
   int   se_socktype;      /* type of socket to use */
   char  *se_proto;     /* protocol used */
   short se_wait;    /* single threaded server */
   short se_checked;    /* looked at during merge */
   char  *se_user;      /* user name to run as */
   struct   builtin *se_bi;    /* if built-in, description */
   char  *se_server;    /* server program */
#define MAXARGV 5
   char  *se_argv[MAXARGV+1]; /* program arguments */
   int   se_issunrpc;      /* sunrpc entry */
   long  se_prognum;    /* sunrpc program number */
   long  se_lowvers;    /* lowest version supported */
   long  se_highvers;      /* highest version supported */
   int   se_fd;         /* open descriptor */
   struct   sockaddr_in se_ctrladdr;/* bound address */
   int   se_count;      /* number started since se_time */
   struct   timeval se_time;  /* start of se_count */
   struct   servtab *se_next;
} *servtab;
```


builtins table

```
struct biltin {
   char  *bi_service;      /* internally provided service name */
   int   bi_socktype;      /* type of socket supported */
   short bi_fork;    /* 1 if should fork before call */
   short bi_wait;    /* 1 if should wait for child */
   int   (*bi_fn)();    /* function which performs it */
} biltins[] = {
   /* Echo received data */
   "echo",     SOCK_STREAM,   1, 0, echo_stream,
   "echo",     SOCK_DGRAM, 0, 0, echo_dg,

   /* Return human-readable time */
   "daytime",  SOCK_STREAM,   0, 0, daytime_stream,
   "daytime",  SOCK_DGRAM, 0, 0, daytime_dg,
   0
};
```

daemonize

```
if (fork())
   exit(0);
{ int s;
for (s = 0; s < 10; s++)
   (void) close(s);
}
(void) open("/", O_RDONLY);
(void) dup2(0, 1);
(void) dup2(0, 2);
{ int tt = open("/dev/tty", O_RDWR);
  if (tt > 0) {
   ioctl(tt, TIOCNOTTY, (char *)0);
   close(tt);
  } else

  } else
   setpgrp(0, 0);
}
```


startup

```
#define SIGBLOCK (sigmask(SIGCHLD)|sigmask(SIGHUP)|sigmask(SIGALRM))
  openlog("inetd", LOG_PID | LOG_NOWAIT, LOG_DAEMON);
  bzero((char *)&sv, sizeof(sv));
  sv.sv_mask = SIGBLOCK;
  sv.sv_handler = retry;
  sigvec(SIGALRM, &sv, (struct sigvec *)0);
  config();
  sv.sv_handler = config;
  sigvec(SIGHUP, &sv, (struct sigvec *)0);
  sv.sv_handler = reapchild;
  sigvec(SIGCHLD, &sv, (struct sigvec *)0);
```


config & setup

```
config() reads in /etc/inetd.conf,
  fills socket struct and does setup()
      sp = getservbyname(sep->se_service, sep->se_proto);
            sep->se_ctrladdr.sin_port = sp->s_port;

setup(sep)
   register struct servtab *sep;
{
   if ((sep->se_fd = socket(AF_INET, sep->se_socktype, 0)) < 0) {
      syslog(LOG_ERR, "%s/%s: socket: %m",
          sep->se_service, sep->se_proto);
      return;
   }
   if (bind(sep->se_fd, &sep->se_ctrladdr,
       sizeof (sep->se_ctrladdr)) < 0) {
      syslog(LOG_ERR, "%s/%s: bind: %m",
          sep->se_service, sep->se_proto);
      (void) close(sep->se_fd);
      sep->se_fd = -1;
  ...  rpc servers ...
   if (sep->se_socktype == SOCK_STREAM)
      listen(sep->se_fd, 10);
   FD_SET(sep->se_fd, &allsock);
   nsock++;
   if (sep->se_fd > maxsock)
      maxsock = sep->se_fd;
}
```


the loop

```
for (;;) {
    int s, ctrl, n;
    fd_set readable;
    while (nsock == 0)
       sigpause(0);
    readable = allsock;
    if ((n = select(maxsock + 1, &readable, (fd_set *)0,
   (fd_set *)0, (struct timeval *)0)) <= 0) {
       if (n < 0 && errno != EINTR)
         syslog(LOG_WARNING, "select: %m\n");
       sleep(1);
       continue;
    }
    for (sep = servtab; n && sep; sep = sep->se_next)
    if (FD_ISSET(sep->se_fd, &readable)) {
      n--;
      if (!sep->se_wait && sep->se_socktype == SOCK_STREAM) {
      ctrl = accept(sep->se_fd, (struct sockaddr *)0,
          (int *)0);
      if (ctrl < 0) {
         if (errno == EINTR)
            continue;
         syslog(LOG_WARNING, "accept: %m");
         continue;
      }
   } else ctrl = sep->se_fd;
```

```
      (void) sigblock(SIGBLOCK);
      pid = 0;
      dofork = (sep->se_bi == 0 || sep->se_bi->bi_fork);
      if (dofork) {
...
         pid = fork();
      }
      if (pid < 0) {
         if (!sep->se_wait && sep->se_socktype == SOCK_STREAM)
            close(ctrl);
         sigsetmask(0);
         sleep(1);
         continue;
      }
       if (pid && sep->se_wait) {
         sep->se_wait = pid;
         FD_CLR(sep->se_fd, &allsock);
         nsock--;
      }
      sigsetmask(0);
```



```
      if (pid == 0) {
         if (dofork)
            for (i = getdtablesize(); --i > 2; )
               if (i != ctrl)
                  close(i);
         if (sep->se_bi)
            (*sep->se_bi->bi_fn)(ctrl, sep);
         else {
            dup2(ctrl, 0);
            close(ctrl);
            dup2(0, 1);
            dup2(0, 2);
            if ((pwd = getpwnam(sep->se_user)) == NULL) {
               syslog(LOG_ERR,
                  "getpwnam: %s: No such user",
                  sep->se_user);
               if (sep->se_socktype != SOCK_STREAM)
                  recv(0, buf, sizeof (buf), 0); /*consume*/
               _exit(1);
            }
```



```
            if (pwd->pw_uid) {
               (void) setgid((gid_t)pwd->pw_gid);
               initgroups(pwd->pw_name, pwd->pw_gid);
               (void) setuid((uid_t)pwd->pw_uid);
            }
            if (debug)
               fprintf(stderr, "%d execl %s\n",
                   getpid(), sep->se_server);
            execv(sep->se_server, sep->se_argv);
            if (sep->se_socktype != SOCK_STREAM)
               recv(0, buf, sizeof (buf), 0);
            syslog(LOG_ERR, "execv %s: %m", sep->se_server);
            _exit(1);
         }
      }
      if (!sep->se_wait && sep->se_socktype == SOCK_STREAM)
         close(ctrl);
       }
   }
}
```



```
reapchild()
{
   union wait status;
   int pid;
   register struct servtab *sep;

   for (;;) {
      pid = wait3(&status, WNOHANG, (struct rusage *)0);
      if (pid <= 0)
         break;
      for (sep = servtab; sep; sep = sep->se_next)
         if (sep->se_wait == pid) {
            if (status.w_status)
               syslog(LOG_WARNING,
                   "%s: exit status 0x%x",
                   sep->se_server, status);
            FD_SET(sep->se_fd, &allsock);
            nsock++;
            sep->se_wait = 1;
         }
   }
}
```

builtins

```
echo_stream(s, sep)     /* Echo service -- echo data back */
   int s;
   struct servtab *sep;
{
   char buffer[BUFSIZ];
   int i;
   setproctitle("echo", s);
   while ((i = read(s, buffer, sizeof(buffer))) > 0 &&
       write(s, buffer, i) > 0)
      ;
   exit(0);
}

echo_dg(s, sep)         /* Echo service -- echo data back */
   int s;
   struct servtab *sep;
{
   char buffer[BUFSIZ];
   int i, size;
   struct sockaddr sa;
   size = sizeof(sa);
   if ((i = recvfrom(s, buffer, sizeof(buffer), 0, &sa, &size)) < 0)
      return;
   (void) sendto(s, buffer, i, 0, &sa, sizeof(sa));
}
```



```
daytime_stream(s, sep)     /* Return human-readable time of day */
   int s;
   struct servtab *sep;
{
   char buffer[256];
   time_t time(), clock;
   char *ctime();

   clock = time((time_t *) 0);

   sprintf(buffer, "%.24s\r\n", ctime(&clock));
   (void) write(s, buffer, strlen(buffer));
}
```



```
daytime_dg(s, sep)      /* Return human-readable time of day */
   int s;
   struct servtab *sep;
{
   char buffer[256];
   time_t time(), clock;
   struct sockaddr sa;
   int size;
   char *ctime();

   clock = time((time_t *) 0);

   size = sizeof(sa);
   if (recvfrom(s, buffer, sizeof(buffer), 0, &sa, &size) < 0)
      return;
   sprintf(buffer, "%.24s\r\n", ctime(&clock));
   (void) sendto(s, buffer, strlen(buffer), 0, &sa, sizeof(sa));
}
```



```
machtime_dg(s, sep)
   int s;
   struct servtab *sep;
{
   long result;
   struct sockaddr sa;
   int size;

   size = sizeof(sa);
   if (recvfrom(s, (char *)&result, sizeof(result), 0, &sa, &size) <
      return;
   result = machtime();
   (void) sendto(s, (char *) &result, sizeof(result), 0, &sa, sizeof
}
long
machtime()
{
   struct timeval tv;

   if (gettimeofday(&tv, (struct timezone *)0) < 0) {
      fprintf(stderr, "Unable to get time of day\n");
      return (0L);
   }
   return (htonl((long)tv.tv_sec + 2208988800));
}
```

inetd/daytimetcpsrv3.c

fds 0,1,2 are the socket

```
main(int argc, char **argv)
{
    socklen_t              len;
    struct sockaddr *cliaddr;
    char                   buff[MAXLINE];
    time_t                 ticks;

    cliaddr = Malloc(MAXSOCKADDR);
    len = MAXSOCKADDR;
    Getpeername(0, cliaddr, &len);
    err_msg("connection from %s", Sock_ntop(cliaddr, len));

    ticks = time(NULL);
    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
    Write(0, buff, strlen(buff));
}
```


finger/fingerd

- example of inetd-run server
- simple ASCII/TCP protocol
- remote server invokes finger

```
finger stream  tcp     nowait  nobody  /usr/etc/in.fingerd     in.fi
```

```
finger bob
finger bob@remote.com
finger @remote.com
```


finger

- gets name info from /etc/passwd
- gets who's logged in from /etc/utmp
- gets lastlogin info from /usr/adm/lastlog
- checks for mail in /usr/spool/mail/
- displays user .plan and .project files

or if @xxxx forward arguments on to remote

netfinger in finger.c

user@host.com

```
hp = gethostbyname(host);
if (hp == NULL) {
   defaddr.s_addr = inet_addr(host);
   if (defaddr.s_addr == -1) {
      printf("unknown host: %s\n", host);
      return (1);
   }
   .... build a dummy hostent
 }
 sin.sin_family = hp->h_addrtype;
 bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
 sin.sin_port = htons(IPPORT_FINGER);
 s = socket(hp->h_addrtype, SOCK_STREAM, 0);
 connect(s, (char *)&sin, sizeof (sin));
 write(s, name, strlen(name));
 write(s, "\r\n", 2);
 f = fdopen(s, "r");
 while ((c = getc(f)) != EOF) {
   /* read from socket and display to user */
   putchar(c);
 }
```

fingerd

- fork'd by inetd when connect arrives on port 79
- inetd maps socket to stdin/stdout
- takes ascii data from port and uses as argument to fork/exec of finger
- takes piped output of the finger command and writes it back over the socket

fingerd.c

```
struct sockaddr_in sin;
int i, p[2], pid, status;
FILE *fp;
char *av[4];

i = sizeof (sin);
getpeername(0, &sin, &i);
fgets(line, sizeof(line), stdin);
sp = line;
av[0] = "finger";
i = 1;
while (1) {
  ... create arguments in av[] from line
}
pipe(p);  /* create pipe for finger output*/
if ((pid = fork()) == 0) { /* child */
  close(p[0]);  /* close read side of pipe */
  if (p[1] != 1) {
    dup2(p[1], 1);  /* make pipe stdout */
    close(p[1]);
  }
  execv("/usr/ucb/finger", av);
  _exit(1);
}
```



```
  /* mother process continues */
  close(p[1]);  /* close write side of pipe */
  fp = fdopen(p[0], "r"); /* use buffered i/o */
  while ((i = getc(fp)) != EOF) {
    /* read finger output from pipe, write to socket */
    putchar(i);
  }
  fclose(fp);
  while ((i = wait(&status)) != pid && i != -1);
}
```


pipes

int pipe(int *fdvec)

int status, fds[2];
status = pipe(fds);

- *classic* UNIX IPC
- creates **1-way** communication stream
- returns two file descriptors
- *fds[0]* is opened for reading
- *fds[1]* is opened for writing
- write() of 4096 or less is atomic
- pipe size is 4096 (*sys/param.h*)

```
main()
{
   int pipefd[2],n;
   char buff[100];

   if (pipe(pipefd) <0){
      perror("pipe");
      exit(-1);
   }
   printf(" rd %d wrt %d\n",pipefd[0],pipefd[1]);
   write(pipefd[1],"hello world\n",12);
   n=read(pipefd[0],buff,sizeof(buff));
   write(1,buff,n);
}
```



```
/* pg102.c with fork */
#define DANG(X) {perror("X"); exit(-1);}
main()
{
   int pipefd[2],n,pid;
   char buff[100];

   if (pipe(pipefd) <0) DANG(pipe)
   printf(" rd %d wrt %d\n",pipefd[0],pipefd[1]);
   if ((pid = fork()) == 0){ /* child */
      close(pipefd[0]); /* optional */
      write(pipefd[1],"hello world\n",12);
      _exit(0);
   }
   if (pid < 0 ) DANG(fork)
   close(pipefd[1]); /* optional */
   n=read(pipefd[0],buff,sizeof(buff));
   write(1,buff,n);
}
```



```
/* sortem.c */
main()
{
   int p[2], stat;

   if (pipe(p) < 0) err_sys("pipe");
   if (fork()==0){ /* child */
      close(p[1]); /* so detect EOF */
      dup2(p[0],0); /*stdin from pipe*/
      system("sort");
      exit(0);
   }
   write(p[1],"zebra\n",6);
   write(p[1],"apple\n",6);
   write(p[1],"bart\n",5);
   close(p[1]);  /* EOF to sort */
   wait(&stat); /* not critical */
}
```


2 pipes

If you you need two-way flow of information, you need two pipes.

- create two pipes
- fork
- parent closes read-end of 1 and write-end of 2
- child closes write-end of 1 and read-end of 2

Watch out for **deadlock**!

```
   parent              time     child
...   (pipes are empty)  |     .....
read(p2[0],b,n);         |     read(p1[0],c,m);
  ...                    |     ....

OR

....                     |     .....
write(p1[1],b,5000);     |     write(p2[1],c,5000);
 ....                    |      .....
```

next time

tooltime: tcpdump

IP