Today

- signals
- fork
- concurrent servers

make

- multi-file project build tool
- dependency structure
- build rules
- modification times
- look at booksrc/


```
# makefile for server and client
#  tabs are important at start of command lines
CFLAGS = -O
OBJS = client.o server.o
LIBOBJS = darth.o bart.o misc.o
LIBS =

all: server client libtom.a

$(OBJS):        stuff.h

server: server.o
        cc -o $@  $@.o $(LIBS)

client: client.o
        cc -o $@  $@.o $(LIBS)

$(LIBOBJS):     tom.h

libtom.a:       $(LIBOBJS)
        ar cr libtom.a
        ranlib libtom.a
clean:
        rm client server libtom.a $(OBJS) $(LIBOBJS)
```


UDP

- datagrams, unreliable
- lightweight
- sendto(), recvfrom()
- need timeout (alarm(secs))

What if?

things that go bump in the net

1. UDP sendto, and no server process
2. UDP sendto with connect(), and no server process
3. active UDP session, ctrl-c server
4. inactive UDP session, server computer crashes and reboots
5. server tries to bind to port $<$ 1024
6. start 2nd copy of server
7. A sends faster than B can receive (UDP)
8. A sends faster than its interface (ENOBUFS)
9. A sends 10000 byte datagram
10. A sends 500-byte datagram, B recvfrom len is 100
11. A sending to B, can C send to B?
12. fd's/host (TCP vs UDP)

netstat -s may show overruns

increase SO_RCVBUFF may help

UDP timeouts

client will worry about lost packets

```
...
void ding(int signo) { timedout=1;}
...
   signal(SIGALRM, ding);
   siginterrupt(SIGALRM, 1);   /* don't restart syscalls */
....
   sendto(sockfd,buff,lths[i],0,NULL,0);
   timedout=0;
   alarm(SECS);
   if (recvfrom(sockfd,buff,lths[i],0,&from,&fromlen) <0){
     if (timedout) lost++;
     else err_sys("recvfrom");
   }
   alarm(0);
...
```


Signals

- software interrupts (event notification)
- synchronous

  addressing error
  illegal instruction
  floating pt. exception
- asynchronous

  I/O-network events
  timer interrupts
  process termination
  ctrl-* or kill -SIG
- often non-portable
- used to be unreliable

portability

Beware!

- BSD, Posix, Sys V, Linux
- installed once ?
- blocking calls resumed?
- signal queueing
- based on sigaction()
- watch out for race conditions

Posix semantics

- installed once
- during handling, same signal is blocked
- if multiple signal, only one delivered
- slow system calls (accept/recvfrom/select/read) may not be restarted, *errno == EINTR*
- *connect()* cannot be restarted manually, need *select()*

book's wrapper

```
#include        "unp.h"

Sigfunc *
signal(int signo, Sigfunc *func)
{
        struct sigaction        act, oact;

        act.sa_handler = func;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
            act.sa_flags |= SA_INTERRUPT;   /* SunOS 4.x */
#endif
        } else {
#ifdef  SA_RESTART
            act.sa_flags |= SA_RESTART; /* SVR4, 44BSD */
#endif
        }
        if (sigaction(signo, &act, &oact) < 0)
                return(SIG_ERR);
        return(oact.sa_handler);
}
```


Generating signals

- by process itself
- by kernel
- by another process (kill())
- by user (kill, ctrl-*)

Signal calls

void (*signal (int sig, void (*disp)(int)))(int)

- SIG_IGN to ignore, or
- SIG_DFL to revert to default, or
- handler to call when signal occurs
- returns previous setting
- one handler can handle multiple signals
- handler remains installed after signal
- signal held during handling

int sigpause(int sigmask)
int sigblock(int mask)
int sigsetmask(int mask)
int sigmask(int sig)
int kill(int pid, int sig)

```c
/* cleanup.c */
#include <signal.h>
void bang(int signo)
{
        printf("done\n");
        exit(0);
}

main()
{
        int i;

        signal(SIGINT,bang);
        signal(SIGTERM,bang);

        for(;;)i++;
}
```



```c
/* sigenv.c */
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#define SECS 3
#define MAXLTH 256
int timedout;
...
```



```c
void ding(int signo){ timedout = 1; }
main()
{
   int status,ding();
   char *getenv(), var[MAXLTH],*val;

   signal(SIGALRM,ding);
   siginterrupt(SIGALRM,1);  /* don't restart */
   for(;;){
      printf("environ variable name: ");
      alarm(SECS);
      status = scanf("%s",var);
      if (status  < 0 ) {
         if (errno == EINTR && timedout){
            printf("\ntimed out\n");
            exit(0);
         } else {
            perror("scanf failed");
            exit(1);
         }
      }
      alarm(0);  /* turn off timer */
      val = getenv(var);
      if (val)printf("%s = %s\n",var,val);
   }
}
```


Critical region

```c
oldmask = sigblock(sigmask(SIGQUIT)
   | sigmask(SIGINT));
/* ...  critical region .... */
sigsetmask(oldmask);
```

see *sigaction()/sigprocmask()*

concurrent servers

- when iterative takes too long
- fork()
- threads
- select()
- asynchronous IO

Processes

- unit of execution
- hierarchical
- context

  program counter
  memory + swap
  registers
  open files

Process calls

int fork()
create new process (copy)

void exit(int status)
terminate process, return status to parent
_exit(int status) for no-flush

parent or shell may be checking status so
main() should exit(0) or return(0)

int wait(int *status)
int wait3(union wait *s, int opt, struct rusage *r)
pid_t waitpid(pid_t pid, int *status, int options);
blocking/nonblocking wait on child termination

```
       +--------------+---------------+
status |exit argument | signal # or 0 |
       +--------------+---------------+
```


fork

- fork() returns twice
- fork() returns 0 to child
- fork() returns child's pid to parent
- child is <u>*copy*</u> of parent, including:

  memory, pgm counter, open files,
  IDs, umask, signals, directories

- new process/parent ID
- used for concurrent processing
- used for running new image (exec())

```
/* fork0.c  */
main()
{
   int pid;

   pid = fork();
   if (pid == 0){ /* child */
      printf("the new process speaks\n");
   } else {
      printf("the old parent speaks\n");
   }
   printf("both say my pid is %d\n",getpid());
}
```



```
/* forkjoin.c  */
main()
{
   int pid,status;

   pid = fork();
   if (pid == 0){ /* child */
      /*child does work .... */
      exit(0);
   }
   /*parent continues ... */
   if (pid < 0 ) { /* bad news */
      perror("fork failed");
      exit(1);
   }
   /* parent does work ... */
   pid = wait(&status); /* await child */
   exit(0);
}
```


What's printed?

```
/* forkbuf.c */
main()
{
        printf("Budeea ");
        if (fork() == 0) exit(0);
        printf("that's all folks\n");
}
```

Budea Budea that's all folks
_exit() won't flush buffers

How many processes?

```
/* forkn.c */
main()
{
        fork();
        fork();
        fork();
        printf("%d\n",getpid());
}
```

Why fork?

- avoid blocking
- overlap slow operations
- real speedup on parallel processors
- isolation (daemon)
- utilize other executables (exec())

Process termination

What happens when:

child terminates before parent
parent exits before child
parent is stopped (ctrl-Z)
child is killed
parent is killed

zombies

terminated child processes

- wait for child processes to terminate
- termination is asynchronous, use SIGCHLD
- multiple signals could be lost, need waitpid()
- signal could interrupt (EINTR) blocking calls

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);

   /* older */
pid_t wait3(int *status, int options,
     struct rusage *rusage)
```


tcpcliserv/tcpserv04.c

```
...
main(int argc, char **argv)
{
        int   listenfd, connfd;
        pid_t   childpid;
        socklen_t  clilen;
        struct sockaddr_in  cliaddr, servaddr;
        void sig_chld(int);

        listenfd = Socket(AF_INET, SOCK_STREAM, 0);

        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family      = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port        = htons(SERV_PORT);

        Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

        Listen(listenfd, LISTENQ);

        Signal(SIGCHLD, sig_chld);      /* must call waitpid() */
```

```
   for ( ; ; ) {
        clilen = sizeof(cliaddr);
        if ( (connfd = accept(listenfd, (SA *) &cliaddr,
              &clilen)) < 0) {
                if (errno == EINTR)
                        continue;    /* back to for() */
                else
                        err_sys("accept error");
        }

        if ( (childpid = Fork()) == 0) { /* child process */
                Close(listenfd);        /* close listening socket */
                str_echo(connfd);       /* process the request */
                exit(0);
        }
        Close(connfd);     /* parent closes connected socket */
   }
}
```



```
void
sig_chld(int signo)
{
        pid_t   pid;
        int             stat;

        while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
                printf("child %d terminated\n", pid);
        return;
}
```


Relative timings (μs)

```
subroutine call    1
system call       21
thread           463
vfork           1559
rpc             1664
fork           13710++++
```


next time

threads, daemons, select