Today

- fork
- threads
- select

concurrent servers

when iterative takes too long

- fork()
- threads
- select()
- nonblocking IO (poll/EWOULDBLOCK)
- signal-driven IO
- asynchronous IO

blocking calls: read, recvfrom, accept, write (buffs full)

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++++

threads

lightweight process (but support multiple processeors)

- fork expensive: memory copies, fd's duplicated
- fork: interprocess communication hard
- threads share
  - global variables
  - fd's signals
- threads have their own
  - thread ID
  - registers, stack
  - errno and signal mask

thread issues

- not supported on some OS's
- synchronize access to shared variables
- thread-safe libraries
- local static
- various APIs

thread APIs

- Posix *pthread.h -lpthread* *pthread_create()*
- Solaris *thread.h -lthread* *thr_create()*
- Win32 *process.h -MT* *_beginthread()*

thread speed

```
#include <pthread.h>
main()   /* link with -lpthread */
{
     pthread_t t;
     int j,i;
     int work();
     double s,seconds();

     s=seconds();
     for(i=0;i<N;i++){
             pthread_create(&t, NULL,work,(void *)i);
             pthread_join(t,NULL);
     }
     s=seconds()-s;
     printf("%f secs %f us\n",s,1.e6*s/N);
}

work()  /* do nothin' */
{
}
```

fork str_cli 15.10

```
void
str_cli(FILE *fp, int sockfd)
{
   pid_t   pid;
   char    sendline[MAXLINE], recvline[MAXLINE];

   if ( (pid = Fork()) == 0) {   /* child: server -> stdout */
      while (Readline(sockfd, recvline, MAXLINE) > 0)
              Fputs(recvline, stdout);

      kill(getppid(), SIGTERM);  /* in case parent still running */
      exit(0);
   }

           /* parent: stdin -> server */
   while (Fgets(sendline, MAXLINE, fp) != NULL)
           Writen(sockfd, sendline, strlen(sendline));

   Shutdown(sockfd, SHUT_WR);    /* EOF on stdin, send FIN */
}
```


shutdown

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

int shutdown(int s, int how);
```

- permit closing half of duplex connnection
- SHUT_RD (0) no more reads, discard buffs
- SHUT_WR (1) no more writes after flush

str_cli 23.2

```
void    *copyto(void *);
static int  sockfd; /* global for both threads to access */
static FILE *fp;
{
   char            recvline[MAXLINE];
   pthread_t       tid;

   sockfd = sockfd_arg;    /* copy arguments to externals */
   fp = fp_arg;

   Pthread_create(&tid, NULL, copyto, NULL);

   while (Readline(sockfd, recvline, MAXLINE) > 0)
           Fputs(recvline, stdout);
}
```



```
void *
copyto(void *arg)
{
    char    sendline[MAXLINE];

    while (Fgets(sendline, MAXLINE, fp) != NULL)
            Writen(sockfd, sendline, strlen(sendline));

    Shutdown(sockfd, SHUT_WR);      /* EOF on stdin, send FIN */

    return(NULL);
    /* return (i.e., thread terminates) when end-of-file on stdin */
}
```

select()

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

int select (width, readfds, writefds, exceptfds, timeout)
int width;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET (fd, &fdset)
FD_CLR (fd, &fdset)
FD_ISSET (fd, &fdset)
FD_ZERO (&fdset)
int fd;
fd_set fdset;
```

- select() returns count of ready fd's
- width is max number of fd's, which can be largest fd open plus one
- if **timeout** is NULL, block til ready or signal (EINTR)
- if **timeout** is 0, poll
- if **timeout** is non-zero, then if timer expires, select() returns 0

the fd masks

- bit masks indicating which fd's to test
- masks are modified after call (value-result args)
- NULL may be supplied, if not interested
- **readfds** descriptors ready for reading, or ready for **accept()** if **listen()** issued
- **writefds** descriptors ready for writing
- **exceptfds** arrival of urgent data

doin' it

- FD_ZERO the master mask
- FD_SET the relevant fd's in the relevant mask (usually readfds)
- copy master to working mask
- issue select with working mask and hi-fd count
- check return value (it's interruptible, EINTR)
- find out which fd's are set (FD_ISSET) and do appropriate I/O operation (read/write, recvfrom, etc.)
- if an error or EOF occurs on an fd operation (read/write), need to FD_CLR fd in master mask (assuming your watching more than one fd)
- go back to copy master step ...

cat select

```
/* select.c */
#include <stdio.h>
#include <sys/types.h>
main()
{
   int n,nfds =1;
   fd_set rdfds, rdfds_master;
   char buf[1024];

   FD_ZERO(&rdfds_master);
   FD_SET(0,&rdfds_master);

   for(;;){
      rdfds = rdfds_master; /* reset */
      if ( select(FD_SETSIZE, &rdfds, NULL,NULL,NULL) <0)
       err_sys("select");
      if (FD_ISSET(0,&rdfds)){
         n=read(0,buf,sizeof(buf));
         if (n<0) err_sys("read");
         if (n == 0) break;
         write(1,buf,n);
      }
   }
}
```

select with timeout

```
main()
{
   int n,nfds =1;
   fd_set rdfds, rdfds_master;
   static struct timeval timeout = {3,0};
   char buf[1024];

   FD_ZERO(&rdfds_master);
   FD_SET(0,&rdfds_master);

   for(;;){
      rdfds = rdfds_master; /* reset */
      if ( (n=select(FD_SETSIZE, &rdfds, NULL,NULL,&timeout))<0)
        err_sys("select");
      if (n == 0) {
         printf("\ntimed out! bye\n");
         exit(1);
      }
      if (FD_ISSET(0,&rdfds)){
         n=read(0,buf,sizeof(buf));
         if (n<0) err_sys("read");
         if (n == 0) break;
         write(1,buf,n);
      }
   }
   exit(0);
}
```


delay.c

```
main()
{
   double t,seconds();
   t= seconds();
   delay(3,700000);
   t = seconds()-t;
   printf("%f\n",t);
}
delay(s,us)
{
   struct timeval tv;

   tv.tv_sec = s;
   tv.tv_usec = us;
   (void)select( 0, NULL, NULL, NULL, &tv );
}

double
seconds()
{
   struct timeval ru;
   gettimeofday(&ru, (struct timezone *)0);
   return(ru.tv_sec + ((double)ru.tv_usec)/1000000);
}
```


gethostbyname

```
...
  s = socket(AF_INET, SOCK_DGRAM, 0);
  if (sendto(s, buf, buflen, 0, &_res.nsaddr_list[ns],
   sizeof(struct sockaddr)) != buflen) {
     if (_res.options & RES_DEBUG)
      perror("sendto");
      continue;
   }
   timeout.tv_sec = (_res.retrans << (_res.retry - retry))
     / _res.nscount;
   if (timeout.tv_sec <= 0)timeout.tv_sec = 1;
   timeout.tv_usec = 0;

wait:
   FD_ZERO(&dsmask);
   FD_SET(s, &dsmask);
   n = select(s+1, &dsmask, (fd_set *)NULL,
       (fd_set *)NULL, &timeout);
   if (n == 0) {
       if (_res.options & RES_DEBUG)
          printf("timeout\n");
      continue
   }
   if ((resplen = recv(s, answer, anslen, 0)) <= 0) {
   ...
```


strcliselect01.c fig 6.9

```
{
        int    maxfdp1;
        fd_set  rset;
        char   sendline[MAXLINE], recvline[MAXLINE];

        FD_ZERO(&rset);
        for ( ; ; ) {
                FD_SET(fileno(fp), &rset);
                FD_SET(sockfd, &rset);
                maxfdp1 = max(fileno(fp), sockfd) + 1;
                Select(maxfdp1, &rset, NULL, NULL, NULL);
```

```
    if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
            if (Readline(sockfd, recvline, MAXLINE) == 0)
                    err_quit("str_cli: server terminated prematurely");
            Fputs(recvline, stdout);
    }

    if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
            if (Fgets(sendline, MAXLINE, fp) == NULL)
                    return;         /* all done */
            Writen(sockfd, sendline, strlen(sendline));
        }
 }
}
```


next time

daemons and inetd