Signals

LECTURE 4

Normal startup

- Run server in the background

  tcpserv &

- Before starting client

  netstat –a

This verifies the state of the server's listening socket

- Now start client on the same terminal

  tcpcli 127.0.0.1

Normal termination (1/2)

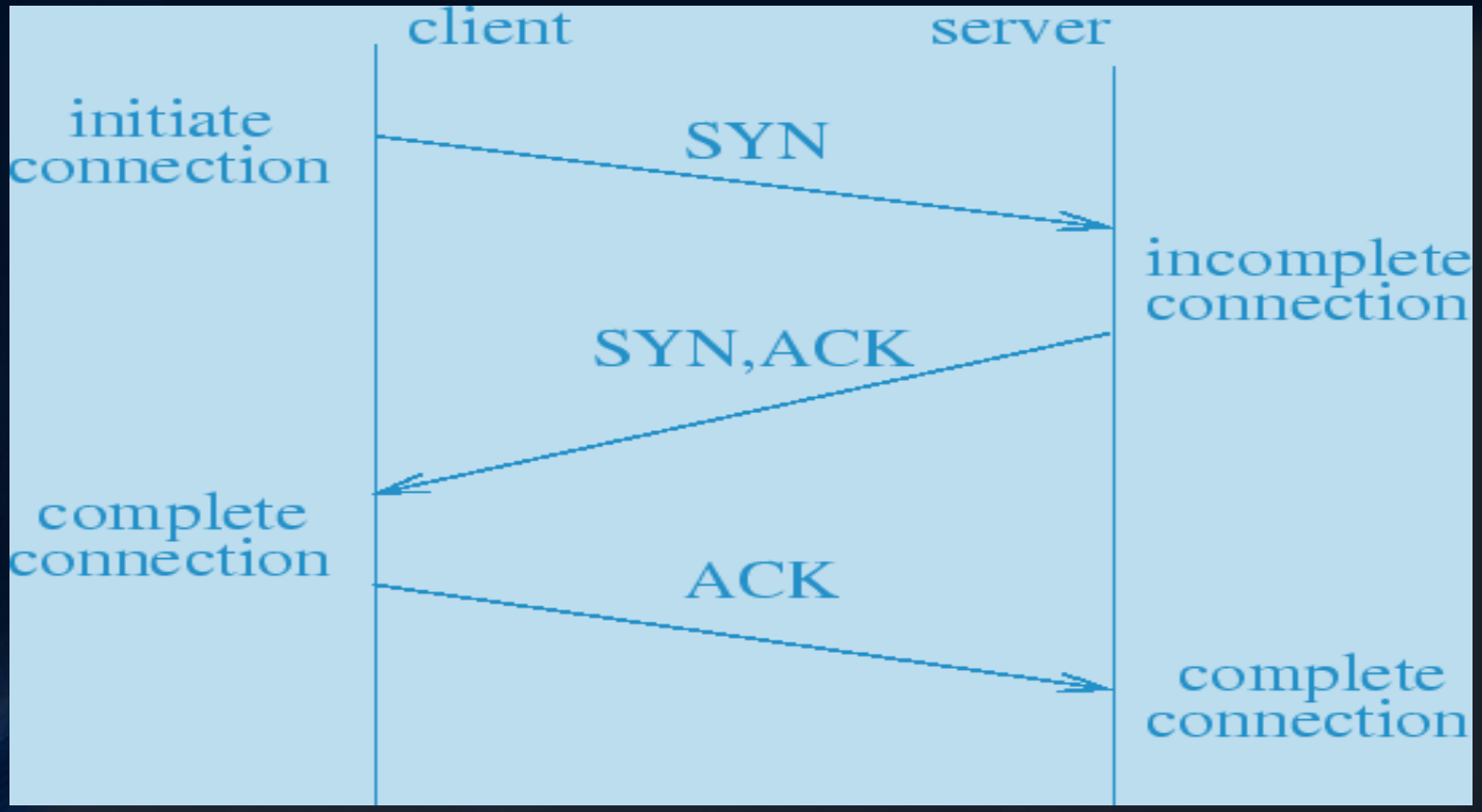
- Suppose, after typing 2 lines, we type ^d (EOF) terminating client then, netstat –a again
- Client side of connection enters the TIME_WAIT state

- Steps
  - Type EOF char (^d), fgets() returns null pointer and echo_client() returns
  - When echo_client() returns, latter terminates (client) by calling exit()

Normal termination (2/2)

- Close all open descriptors. This sends FIN to server, server responds with ACK. At this point, server socket is in CLOSE_WAIT state and client socket is in FIN_WAIT
- When server receives FIN, server child is blocked in call to read() and read returns o causing to return to server child main
- Server child terminates calling exit()
- All open descriptors in server child are closed. Client socket enters TIME_WAIT state
- **SIGCHLD signal is sent to parent when server child terminates**. Signal is ignored (default). Child enters zombie state

Three Way Handshake Scenario



Handling SIGCHLD and Zombies

- Terminates client – type ^d. Client sends FIN to server, server responds with ACK. Child terminates – exit
- Parent is blocked in its call to accept() when SIGCHLD is delivered.
  - The sig_chld() executes
  - wait() fetches child's PID
  - Termination starts
  - printf() is called from signal handler
  - Signal handler returns

Handling interrupted system calls (1/2)

- Since signal was caught by parent while parent was blocked in a slow system call (accept), kernel causes accept() to return error (EINTR – interrupted system call). Parent does not handle error, it aborts

- When process is blocked in slow system call (accept) and process catches a signal and signal handler returns, the system call can return error (EINTR)

- Some kernel automatically restarts some interrupted system calls

Handling interrupted system calls (2/2)

- When we write a program that catches signals, we must be prepared for slow system calls to return EINTR
- Not all interrupted system calls are automatically restarted
- **Can restart read, write, select, open and accept; but not connect**

Signal handling (1/4)

- A signal is a notification to a process that an event has occurred
- Called sometimes as software interrupts
- Usually occur asynchronously i.e. a process doesn't know ahead of time exactly when a signal will occur
- Signals can be sent
  - By one process to another process (or to itself)
  - By the kernel to a process

Signal handling (2/4)

- SIGCHILD signal is sent by the kernel whenever a process terminates, to the parent of the terminating process

- Every signal has a disposition (action) which is associated with the signal

- Set the disposition of a signal by calling the **sigaction()**

Signal handling (3/4)

- Three choices for disposition
  1. **Handle the action – signal handler**. Catch signal. Two signals SIGKILL and SIGSTOP cannot be caught. Prototype of the handler

     void handler (int signo);

  2. **Ignore a signal** by setting its disposition to SIG_IGN. Again SIGKILL and SIGSTOP cannot be ignored

Signal handling (4/4)

3. Can **set the default disposition for a signal** by setting its disposition to SIG_DFL. Default is normally to terminate a process on receipt of a signals. Few signals whose disposition is to be ignored : SIGCHILD and SIGURG (sent on arrival of out-of-band)

signal()

```
void (* signal (int signo, void (*func) (int))) (int);
```

- The above syntax can be broken into

```
typedef void Sigfunc (int);

Sigfunc * signal (int signo, Sigfunc *func);
```

- First arg is signal name
- Second arg is either pointer to a function or one of the constants SIG_IGN or SIG_DFL

wait() – 1/3

- Called wait func to handle the terminated child

```
#include <sys/wait.h>
pid_t wait (int *statloc)
```

- Return value
  - process ID of the terminated child if OK
  - 0 or -1 on error

wait() – 2/3

- Termination status (integer) of the child is returned thro *statloc* pointer

- Three macros we can call that examine termination status and tell us if child is terminated normally, was killed by a signal, or was just stopped by job control

- Use the WIFEXITED and WEXITSTATUS

wait() – 3/3

- If there are no terminated children for the process calling wait, but the process has one or more children that are still executing, then wait blocks until the first of the existing children terminates

Sample program – wait()

waitpid () – 1/2

- The waitpid() gives more control over which process to wait for and whether or not to block

#include <sys/wait.h>

pid_ waitpid (pid_t *pid*, int **statloc*, int *options*);

- Return value
  - process ID of the terminated child if OK
  - 0 or -1 on error
- First, the *pid* arg lets us specify the process ID that we want to wait for

waitpid () – 2/2

- A value of -1 says to wait for the first of the children to terminate
- The *options* arg specify additional options
- Most common is WNOHANG which tells the kernel not to block if there are no terminated children

Sample program – waitpid()

Difference between wait() and waitpid() (kindly go through yourself)

- Pg 126 – sample program (text book)
- When client terminates, all open descriptors are closed automatically by kernel (do not close, only exit)
- All 5 connections are terminated at about the same time
- Cause all 5 children to terminate same time, which cause 5 SIGCHLD signals to be delivered to the parent

Difference between wait() and waitpid() (kindly go through yourself)

- Run the server in background

  tcpserv &

- We will see that only 1 child is terminated, the rest are all zombies
- Problem is that all 5 signals are generated before signal handler is executed
- Signal handler is executed only one time