# Tutorial 4: COMP4621

## Concurrent Client Server Using select and poll

Spring 2024

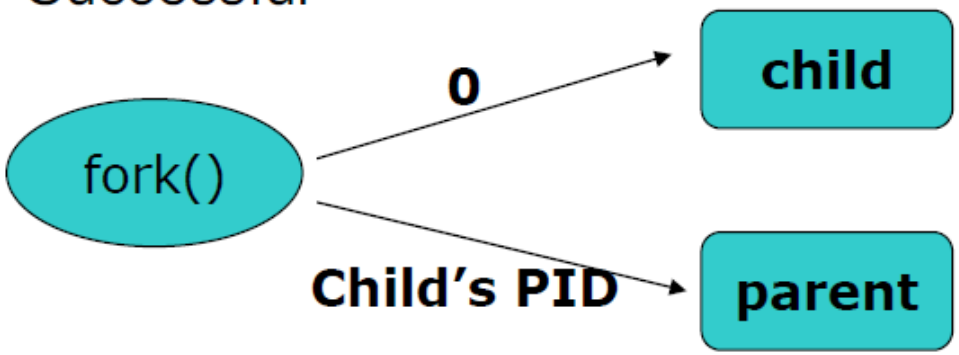
TA: Xinyu YANG
E-mail: xinyu.yang@connect.ust.hk

# Review

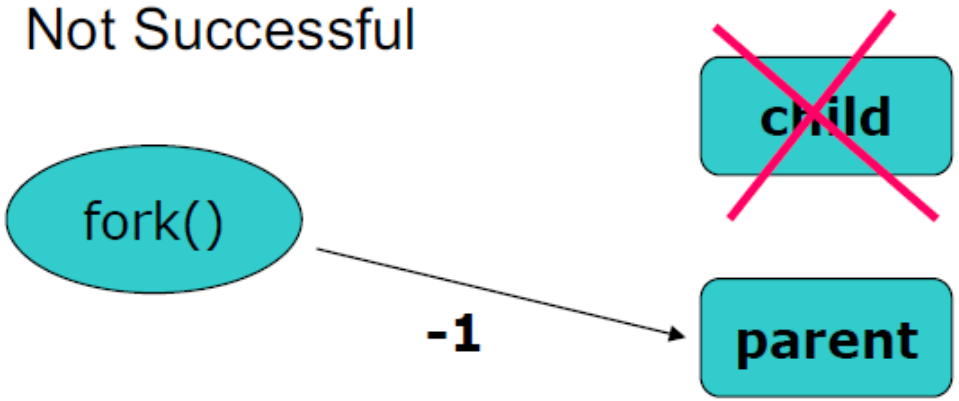
- Concurrent Client Server Using fork
- Multi-client server example
- Non-blocking sockets with timed I/O: setsockopt()

# Review of fork()

- Successful




- Not Successful




```
// Fork a new server process to service thi
if ((PID = fork()) == -1) {
    // Failed to fork: Give up
    close(client_socket);
    continue;
} else if (PID > 0) {
    // Parent process must continue to acce
    close(client_socket);
    continue;
}
// Child process, we do not need the server
printf("Start communicating with client, ")
```

**errno is set to indicate error**

# The problem

Problem: The server is still blocked after one child process terminates; how can we solve the problem to implement real concurrency?

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

```
static void
sigchld_handler(int signo){
    pid_t PID;
    int status;

    do {
        PID = waitpid(-1, &status, WNOHANG);
    } while (PID != -1);

    /* Re-instate handler*/
    signal(SIGCHLD, sigchld_handler);
}
```

The condition in while() misses the case the PID=0, which means there are no state change in any child processes.

Replace “PID != -1” to “PID != -1 && PID != 0” or “PID > 0”.

# Non-blocking sockets with timed I/O

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

- All the socket information is in the man page for socket in section 7. (Type: “man 7 socket” to get all the information)

- Arguments:
  - s is the socket in question
  - level should be set to SOL_SOCKET
  - optname is the name of the option you’re interested in changing
  - optval is usually a pointer to an int indicating the value in question. For booleans, zero is false, and non-zero is true
  - optlen should be set to the length of optval, probably sizeof(int), but varies depending on the option.

- Return values
  - Returns zero on success, or -1 on error (and errno will be set accordingly)

# Non-blocking sockets with timed I/O

- Socket options
  - SO_RCVTIMEO and SO_SNDTIMEO
    - specify the receiving or sending timeouts until reporting an error.
    - The argument is a struct timeval.
    - If an input or output function blocks for this period of time, and data has been sent or received, the return value of that function will be the amount of data transferred; if no data has been transferred and the timeout has been reached, then -1 is returned with errno set to EAGAIN or EWOULDBLOCK, or EINPROGRESS just as if the socket was specified to be nonblocking. If the timeout is set to zero (the default), then the operation will never timeout.

```
struct timeval {
    int tv_sec;     // seconds
    int tv_usec;    // microseconds
};
```

# Non-blocking sockets with timed I/O

- Example

```
tv.tv_sec = 5;  /* 5 Secs Timeout */
tv.tv_usec = 0;
ret = setsockopt(client_socket, SOL_SOCKET, SO_RCVTIMEO, (struct timeval *)&tv, sizeof(struct timeval));

if (ret == SO_ERROR)
{
printf("setsockopt() failed...\n");
return -1;
}
else
printf("setsockopt() is OK!\n");

// read the message from client and copy it in buffer
int numBytes = recv(client_socket, buffer, sizeof(buffer), 0);
if (numBytes <= 0)
{
// nothing received from client in last 5 seconds
close(client_socket);
printf("nothing received in the last %ld seconds, close client socket\n", tv.tv_sec);
break;
}
```

# Outline of this lab

- Concurrent client-server using **poll()**
- Concurrent client-server using **select()**

# poll()--Synchronous I/O Multiplexing

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

- Description:
  - poll() will keep an array of struct pollfds with information about which socket descriptors we want to monitor, and what kind of events we want to monitor for. The OS will block on the poll() call until one of those events occurs (e.g. “socket ready to read!”) or until a user-specified timeout occurs.

- Arguments:
  - fds is our array of information (which sockets to monitor for what)
  - nfds is the count of elements in the array
  - timeout is a timeout in milliseconds.

- Return value: the number of elements in the array that have had an event occur.

# poll()--Synchronous I/O Multiplexing

```
struct pollfd {
    int fd;         // the socket descriptor
    short events;   // bitmap of events we're interested in
    short revents;  // when poll() returns, bitmap of events that occurred
};
```

- The events field is the bitwise-OR of the following:

| Macro | Description |
|---|---|
| POLLIN | Alert me when data is ready to recv() on this socket. |
| POLLOUT | Alert me when I can send() data to this socket without blocking. |

# Skeleton of concurrent client server using poll()

```
// Start off with room for 5 connections
// (We'll realloc as necessary)
int fd_count = 0;
int fd_size = 5;
struct pollfd *pfds = malloc(sizeof *pfds * fd_size);
```

/* create a listener socket and bind it*/

```
// Add the listener to set
pfds[0].fd = listener;
pfds[0].events = POLLIN; // Report ready to read on incoming connection

fd_count = 1; // For the listener
```

# Skeleton of concurrent client server using poll()

```
// Main loop
for(;;){
        int poll_count = poll(pfds, fd_count, -1);

        if (poll_count == -1) {
                perror("poll");
                exit(1);
        }

        // Run through the existing connections looking for data to read
        for(int i = 0; i < fd_count; i++) {

        // Check if someone's ready to read
        if (pfds[i].revents & POLLIN) { // We got one!!

                if (pfds[i].fd == listener) {
                // If listener is ready to read, handle new connection
                addr_size = sizeof(client_addr);
                client_socket = accept(listener, (struct sockaddr*)&client_addr, &addr_size);

                if (client_socket == -1) {
                perror("accept");
                } else {
                add_to_pfds(&pfds, client_socket, &fd_count, &fd_size);
                printf("pollserver: new connection from %s on socket %d\n",inet_ntoa(client_addr.sin_addr),client_socket);
                }
```

# Skeleton of concurrent client server using poll()

```
} else {
// If not the listener, we're just a regular client
bzero(buffer, MAX);

int nbytes = recv(pfds[i].fd, buffer, sizeof buffer, 0);

int sender_fd = pfds[i].fd;

if (nbytes <= 0) {
 // Got error or connection closed by client
 if (nbytes == 0) {
     // Connection closed
     printf("pollserver: socket %d hung up\n", sender_fd);
 } else {
     perror("recv");
 }

 close(pfds[i].fd); // Bye!

 del_from_pfds(pfds, i, &fd_count);
}
```

```
else {
// We got some good data from a client\
// print buffer which contains the client contents
printf("From client: %s", buffer);
// transform the characters into uppercase
for (int ii=0;ii<strlen(buffer);ii++){
buffer[ii] = toupper(buffer[ii]);
}
// if msg contains "Exit" then server exit and chat ended.
if (strncmp("EXIT", buffer, 4) == 0) {
printf("Client Exit...\n");
close(pfds[i].fd);
del_from_pfds(pfds, i, &fd_count);
}
else{
int dest_fd = sender_fd;
// and send that buffer to client
send(dest_fd, buffer, sizeof(buffer), 0);
printf("To client: %s", buffer);
}
```

# Skeleton of concurrent client server using poll()

```c
// Add a new file descriptor to the set
void add_to_pfds(struct pollfd *pfds[], int newfd, int *fd_count, int *fd_size)
{
    // If we don't have room, add more space in the pfds array
    if (*fd_count == *fd_size) {
        *fd_size *= 2; // Double it

        *pfds = realloc(*pfds, sizeof(**pfds) * (*fd_size));
    }

    (*pfds)[*fd_count].fd = newfd;
    (*pfds)[*fd_count].events = POLLIN; // Check ready-to-read

    (*fd_count)++;
}

// Remove an index from the set
void del_from_pfds(struct pollfd pfds[], int i, int *fd_count)
{
    // Copy the one from the end over this one
    pfds[i] = pfds[*fd_count-1];

    (*fd_count)--;
}
```

# select()—Synchronous I/O Multiplexing

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

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

- Description:
  - select() gives you the power to monitor several sockets at the same time. It'll tell you which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions.
- Arguments:
  - numfds should be set to the values of the highest file descriptor plus one.
  - readfds, writefds, exceptfds are the sets of file descriptors to be monitored
  - The pointer (timeout) to the timeout requirement, which is to be applied to this function call.
- Return value:
  - -1 indicates that an error in the function call has occurred. The value of errno should be consulted for the nature of the error.
  - zero indicates that a timeout has occurred without anything interesting happening.
  - A return value greater than zero indicates the number of file descriptors where something of interest has occurred.

# select()—Synchronous I/O Multiplexing

- How to manipulate the sets monitored

| Function | Description |
| --- | --- |
| FD_SET(int fd, fd_set *set); | Add fd to the set. |
| FD_CLR(int fd, fd_set *set); | Remove fd from the set. |
| FD_ISSET(int fd, fd_set *set); | Return true if fd is in the set. |
| FD_ZERO(fd_set *set); | Clear all entries from the set. |

# Skeleton of concurrent client server problem using select()

```
int main(){
        fd_set master;    // master file descriptor list
        fd_set read_fds;  // temp file descriptor list for select()
        int fdmax;        // maximum file descriptor number

        FD_ZERO(&master);    // clear the master and temp sets
        FD_ZERO(&read_fds);
```

/* create a listener socket and bind it*/

```
// add the listener to the master set
FD_SET(listener, &master);

// keep track of the biggest file descriptor
fdmax = listener; // so far, it's this one
```

# Skeleton of concurrent client server problem using select()

```
// main loop
for(;;) {
        read_fds = master; // copy it
        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(4);
        }
        // run through the existing connections looking for data to read
        for(i = 0; i <= fdmax; i++) {
          if (FD_ISSET(i, &read_fds)) { // we got one!!
            if (i == listener) {
                // handle new connections
                addr_size = sizeof(client_addr);
                newfd = accept(listener, (struct sockaddr*)&client_addr, &addr_size);
                if (newfd == -1){
                  perror("accept");
                } else {
                    FD_SET(newfd, &master); // add to master set
                    if (newfd > fdmax) { // keep track of the max
                      fdmax = newfd;
                    }
                    printf("selectserver: new connection from %s on "
                      "socket %d\n",inet_ntoa(client_addr.sin_addr),newfd);
                }
```

# Skeleton of concurrent client server problem using select()

```
} else {
    // handle data from a client
    if ((nbytes = recv(i, buffer, sizeof buffer, 0)) <= 0) {
      // got error or connection closed by client
      if (nbytes == 0) {
        // connection closed
        printf("selectserver: socket %d hung up\n", i);
      } else {
        perror("recv");
      }
      close(i); // bye!
      FD_CLR(i, &master); // remove from master set
```

```
} else {
    // we got some data from a client
    // print buffer which contains the client contents
    printf("From client: %s", buffer);
    // transform the characters into uppercase
    for (int ii=0;ii<strlen(buffer);ii++){
      buffer[ii] = toupper(buffer[ii]);
    }
    // if msg contains "Exit" then server exit and chat ended.
    if (strncmp("EXIT", buffer, 4) == 0) {
      printf("Client Exit...\n");
      close(i); // bye!
      FD_CLR(i, &master);  // remove from master set
    }
    else{
      // and send that buffer to client
      if (send(i, buffer, sizeof(buffer), 0) == -1) {
        perror("send");
      } else {
        printf("To client: %s", buffer);
      }
    }
```

# Summary & Tasks

**Summary**

We have learned how to use select() and poll() to enable concurrent client server and write a simple application.

**Task**

The code for concurrent server are available in Canvas. Run the code and answer the following questions:

1. Describe the components of the structure “pollfd”. What’s the difference between “events” and “revents”?
2. In the select() and poll(), how can we know which socket is ready for reading?
3. To enable the concurrency, what is the main difference between fork() and select()/poll()?
4. Practical Exercise: Implement RDT 3.0 on top of UDP client server. (Hint: this will be needed in the programming project later)

***Submit a zip file including a document and code files on Canvas before 11:59pm April 5!***