CMPUT 379 Lab
Lab 6 - select() and fork()

Benyamin Noori
(bnoori@ualberta.ca)
Based on slides by Muhammad Waqar
(mwaqar@ualberta.ca)

Revision

Server

socket()

bind()

listen()

accept()

send() / recv()

shutdown() (optional)

close()

Client

socket()

connect()

send() / recv()

shutdown() (optional)

close()

send()

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Preferred to write() for sockets

recv()

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Preferred to read() for sockets

Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until a connection is established
  - recv: until a packet is received
  - send: until data is pushed into socket's buffer
    - Not until it is received at the destination! Why?

Dealing with blocking calls (contd)

- For simple programs, blocking is convenient
- What about more complex programs?
  - Multiple connections
  - Simultaneous sends and receives
  - Simultaneously doing non-networking processing

Dealing with blocking calls (contd)

- Options?
  - Create multi-process or multi-threaded code
  - Turn off the blocking feature (fcntl())
  - **select() function**
- What does select() do?
  - Can be permanent blocking, time-limited blocking or non-blocking
  - Input: a set of file descriptors
  - Output: Info on the file descriptors' status

    i.e., can identify sockets that are “ready for use”: calls involving that socket will return immediately

In other words ...

- select()
  - Block until something happens
  - “Something” can be
    - Incoming connection: accept()
    - Clients sending data: recv()
    - Pending data to send: send()
    - Timeout

select()

```
/* According to POSIX.1-2001 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

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

select()

- status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);
  - status: # of ready objects, -1 if error
  - nfds: largest file descriptor to check + 1
  - readfds: list of descriptors to check if read-ready
  - writefds: list of descriptors to check if write-ready
  - exceptfds: list of descriptors to check if an exception is registered
  - timeout: time after which select returns, even if nothing ready – can be zero or infinity

    (point timeout parameter to NULL for infinity)

To be used with select()

- Recall select() uses a structure, **struct fd_set**
  - It is just a bit-vector
  - If bit *i* is set in [readfds, writefds, exceptfds], select() will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]
- Before calling select:
  - FD_ZERO(&fdvar): clears the structure
  - FD_SET(i, &fdvar): to set file desc. *i*
  - FD_CLR(i, &fdvar): to clear file desc. *i*
- After calling select:
  - FD_ISSET(i, &fdvar): returns TRUE iff *i* is ready

Demo 1

Demo 2

Creating new processes

```
#include <unistd.h>

pid_t fork(void);
```

- Creates new process (child) that is identical to the calling process (parent)
- Returns 0 to child process
- Returns child's *pid* to parent process
- It is interesting (and often confusing) because it is called once but returns **twice**

Key points

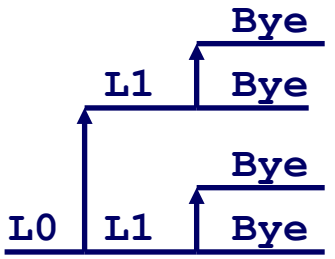
- Parent and child both run the same code
  - Distinguish parent from child by fork() return value
- Start with same state, but each has private copy
  - Including shared input and output file descriptors

Demo

Example #2

- Both parent and child can continue forking

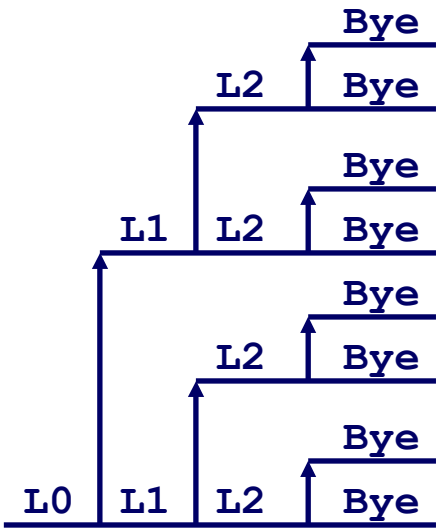
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Example #3

- Both parent and child can continue forking

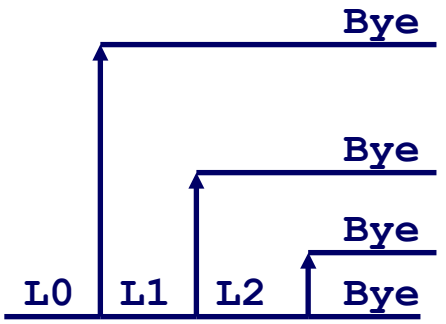
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Example #4

- Both parent and child can continue forking

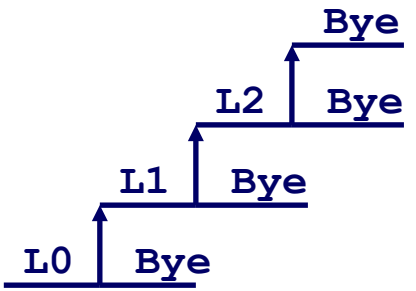
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Example #5

- Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



get_pid() and wait()

- get_pid() is used to get the process ID of a process.
- wait() is used when you want the parent process to wait for its children to finish execution.
  - What if the parent doesn’t wait?
    - Orphan processes
  - What if the child exits before parent waits?
    - Zombie processes
  - wait(NULL); //waits for all children to exit.

Exercise

Write a program that takes N + 1 command line arguments. The first N arguments are english words. The last argument is a number. The program writes each of those words to separate files the number of times that is specified by the last argument.