Announcement (1)

- Due date for PA3 is changed (~ next week)

- PA4 will also be started in the next class

<table>
<thead>
<tr>
<th>학번</th>
<th>Correctness</th>
<th>Runtime</th>
<th>Warning</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2753</td>
<td></td>
<td>10.000</td>
<td></td>
<td>10.000</td>
</tr>
<tr>
<td>3065</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0717</td>
<td>89.700</td>
<td>10.000</td>
<td></td>
<td>99.700</td>
</tr>
<tr>
<td>3814</td>
<td>84.000</td>
<td>10.000</td>
<td></td>
<td>94.000</td>
</tr>
<tr>
<td>2243</td>
<td></td>
<td>10.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4275</td>
<td></td>
<td>10.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0197</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4601</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0963</td>
<td>85.700</td>
<td>10.000</td>
<td></td>
<td>95.700</td>
</tr>
<tr>
<td>0399</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1956</td>
<td>90.000</td>
<td>10.000</td>
<td>0</td>
<td>100.000</td>
</tr>
<tr>
<td>2759</td>
<td>90.000</td>
<td>10.000</td>
<td>0</td>
<td>100.000</td>
</tr>
<tr>
<td>2852</td>
<td>83.000</td>
<td>10.000</td>
<td>0</td>
<td>93.000</td>
</tr>
<tr>
<td>2341</td>
<td>84.400</td>
<td>10.000</td>
<td>0</td>
<td>94.400</td>
</tr>
</tbody>
</table>

Not submitted
Not scored
Concurrent Programming

Prof. Jin-Soo Kim(jinsookim@skku.edu)
TA – Sanghoon Han(sanghoon.han@cs1.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Echo Server Revisited

```c
int main (int argc, char *argv[]) {
    ...
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero((char *)&saddr, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);
    saddr.sin_port = htons(port);
    bind(listenfd, (struct sockaddr *)&saddr, sizeof(saddr));

    listen(listenfd, 5);
    while (1) {
        connfd = accept(listenfd, (struct sockaddr *)&caddr, &clen);
        while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf("got %d bytes from client.\n", n);
            write(connfd, buf, n);
        }
        close(connfd);
    }
}
```
Iterative Servers (1)

- One request at a time

client 1

- call connect
- ret_connect
- call read
- ret_read
- close

server

- call accept
- ret_accept
- write
- close

client 2

- call connect
- ret_connect
- call read
- ret_read
- close
Iterative Servers (2)

- **Fundamental flaw**

  ![Diagram showing the fundamental flaw in iterative servers](image)

- **Solution: use concurrent servers instead**
  
  - Use multiple concurrent flows to serve multiple clients at the same time.
Creating Concurrent Flows

- **Processes**
  - Kernel automatically interleaves multiple logical flows.
  - Each flow has its own private address space.

- **Threads**
  - Kernel automatically interleaves multiple logical flows.
  - Each flow shares the same address space.
  - Hybrid of processes and I/O multiplexing

- **I/O multiplexing with select()**
  - User manually interleaves multiple logical flows
  - Each flow shares the same address space
  - Popular for high-performance server designs.
Exercise #1

- With your own code, make echo server
  - At server side, print the number of characters that received from client
  - At client side, print the string that client typed
Concurrent Programming

Process-based
Process-based Servers

Client 1
- call connect
- ret connect
- call fgets

User goes out to lunch

Client 1 blocks waiting for user to type in data

Server
- call accept
- ret accept
- call fgets

Child 1
- call read

Child 2
- fork
- call accept
- ret accept
- call fgets

Child 2
- fork
- call read
- write

Client 2
- call connect
- ret connect
- call fgets

- write
- call read

- end read
- close
- close
Implementation Issues

- Servers should restart `accept()` if it is interrupted by a transfer of control to the `SIGCHLD` handler
  - Not necessary for systems with POSIX signal handling.
  - Required for portability on some older Unix systems.
- Server must reap zombie children
  - to avoid fatal memory leak
- Server must close its copy of `connfd`.
  - Kernel keeps reference for each socket.
  - After `fork()`, `refcnt(connfd) = 2`
  - Connection will not be closed until `refcnt(connfd) = 0`
Exercise #2

- With your own code, make process-based echo server
  - At the same time, multiple client can be served by echo server
- There should be no memory leakage
  - There should be some codes that handle zombie process
  - How about closing files?
### Process-based Designs

**Pros**
- Handles multiple connections concurrently.
- Clean sharing model.
  - Descriptors (no), file tables (yes), global variables (no)
- Simple and straightforward.

**Cons**
- Additional overhead for process control.
  - Process creation and termination
  - Process switching
- Nontrivial to share data between processes.
  - Requires IPC (InterProcess Communication) mechanisms: FIFO’s, System V shared memory and semaphores
Echo Server

- Iterative version

```c
int main (int argc, char *argv[]) {
    ...

    while (1) {
        connfd = accept (listenfd, (struct sockaddr *)&caddr,
                         &caddrlen);

        while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf("got %d bytes from client.\n", n);
            write(connfd, buf, n);
        }

        close(connfd);
    }
}
```
Echo Server: Process-based

```c
int main (int argc, char *argv[]) {
   . . .
   signal (SIGCHLD, handler);

   while (1) {
      connfd = accept (listenfd, (struct sockaddr *)&caddr,
                       &caddrlen));
      if (fork() == 0) {
         close(listenfd);
         while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf("got %d bytes from client.\n", n);
            write(connfd, buf, n);
         }
      }
      close(connfd);
      exit(0);
   }
   close(connfd);
}
```

```c
void handler(int sig) {
   pid_t pid;
   int stat;
   while ((pid = waitpid(-1, &stat,
                          WNOHANG)) > 0);
   return;
}
```