I/O Models

Kartik Gopalan
Types of Concurrency

- True Concurrency (multiple processes or threads)
  - Multi-processor machines
    - Child processes/threads execute in parallel.
  - Multi-process (forking) servers
    - If one child server blocks, another executes.
  - Multi-threaded servers
    - If one thread blocks, another executes.
    - Only if threads supported by kernel.

- Apparent Concurrency
  - Single process does multiplexing among multiple clients.
    - E.g. I/O multiplexing with select().
  - Multi-threaded Server (again!)
    - If threads implemented at user level.
True Concurrency: Forking Concurrent Servers

Listening Server

Child Server 1 ← Client 1

Child Server 2 ← Client 2
Apparent Concurrency
Non-Forking Concurrent Server

Concurrent Server

select()

Sockets ➔ fd1 ➔ C1
fd2 ➔ C2
fd3 ➔ C3
fd4 ➔ C4
fd5 ➔ C5

Remote Clients ➔

listenfd
I/O Models

- Blocking I/O
- Non-blocking I/O
- I/O multiplexing – select()
- Signal driven I/O
- Asynchronous I/O
Two steps in data reception

1. Data Arrival (network)
2. Copy

Overheads: Context switching, Data copying
Blocking I/O

Application

read( ) / recv( )

System call

Operating system

No data ready

Wait for data

data ready

Copy data

Copy data to user

Copy complete

Return ok

Read data

Process blocks

Time
Non-Blocking I/O (polling)

**Application**

read()/recv() → System call → No data ready

read()/recv() → System call → No data ready

read()/recv() → System call → data ready

Copy data

Process blocks

Read data

**Operating system**

Return ok

Copy complete

Copy data to user
I/O Multiplexing

Concurrent Server

select()

listen

socket descriptor

Sockets or Files descriptors
Or Pipe descriptors

fd1 fd2 fd3 fd4 fd5

Remote Clients
Or files
Or pipes

C1 C2 F3 F4 P5
I/O Multiplexing

Application

select() on multiple fds

System call

No data ready

Operating system

Wait for data on any fd

read()/recv()

System call

data ready

Copy data

Copy data to user

Time

Process blocks

Return readable

Process blocks

Return ok

Read data

Copy complete

Return ok
Signal driven I/O

Application

Establish **SIGIO**
Signal handler

**Time**

Process continues

**Signal Handler**

read()/recv()  

Process blocks

Read data

Operating system

System call

**No data ready**

Wait for data

data ready

Copy data

System call

Deliver SIGIO

**Return ok**

Copy complete

Copy data to user
Asynchronous I/O

**Application**

*aio_read()*

**System call**

*return*

**Operating system**

*No data ready*

*Wait for data*

**Data ready**

*Copy data*

*Copy data to user*

*Copy complete*

**Process continues**

**Signal handler**

*Read data*

**Deliver signal**

*specified in* *aio_read()*

**Time**
I/O Multiplexing

Example of Event-oriented programming
What is I/O multiplexing?

• When an application needs to handle multiple I/O descriptors at the same time
  • E.g. file and socket descriptors, multiple socket descriptors

• When I/O on any one descriptor can result in blocking
Non-forking concurrent server

Concurrent Server

select()

listen socket

Files/ Sockets

\[ \text{fd1} \rightarrow C1 \]
\[ \text{fd2} \rightarrow C2 \]
\[ \text{fd3} \rightarrow C3 \]
\[ \text{fd4} \rightarrow C4 \]
\[ \text{fd5} \rightarrow C5 \]

Clients
select() call

- Allows a process to wait for an event to occur on any one of its descriptors.

- Types of event
  - ready for read
  - ready for write
  - Exception condition
select() call

```c
int select(
    int maxfdp1, /* max. fd + 1 */
    fd_set *readfds, /* read ready? */
    fd_set *writefds, /* write ready? */
    fd_set *exceptfds, /* exceptions? */
    struct timeval *timeout);
```

```c
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
}
```
struct fd_set

• Set of descriptors that we want to wait on for events.

• Typically holds 256 descriptor states.

• Manipulation macros
  • void FD_ZERO(fd_set *fds)
  • void FD_SET (int fd, fd_set *fds)
  • void FD_CLR (int fd, fd_set *fds)
  • int FD_ISSET(int fd, fd_set *fds)
Non-forking Concurrent Server

```c
fdset rdset, wrset;
int listenfd, connfd1, connfd2;
int maxfdp1;
```
```
/* initialize */
FD_ZERO(&rdset);
FD_ZERO(&wrset);
```
for( ;; ) {
    FD_SET(connfd1, &rdset);
    FD_SET(connfd2, &wrset);
    FD_SET(listenfd, &rdset);

    maxfdp1 = max(connfd1, connfd2, listenfd) + 1;

    /* wait for some event */
    Select(maxfdp1, &rdset, &wrset, NULL, NULL);

    if( FD_ISSET(connfd1, &rdset) ) {  
        Read data from connfd1...
    }

    if( FD_ISSET(connfd2, &wrset) ) {  
        Write data to connfd2...
    }

    if( FD_ISSET(listenfd, &rdset) ) {  
        Process a new connection...
    }
}