CpSc 360: Distributed and Network Programming

I/O Multiplexing

James Wang

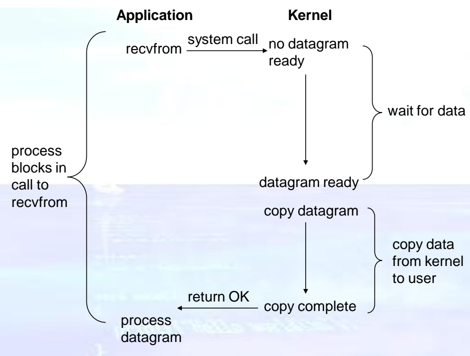
Textbook Chapter 6

Motivation

- **We often need to be able to monitor multiple descriptors:**
  - a generic TCP client (like telnet)
  - A server that handles both TCP and UDP
  - Client that can make multiple concurrent requests (browser?).
  - A TCP server handles both a listening socket and its connected sockets.
  - A server handles multiple services and perhaps multiple protocols.

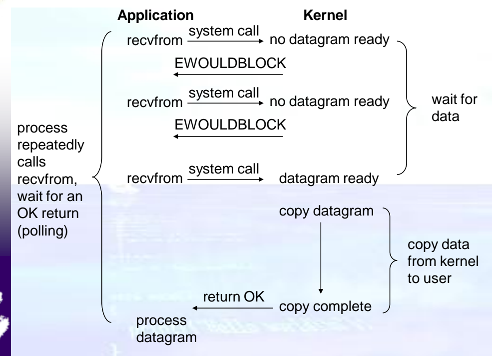
2

Blocking I/O Model



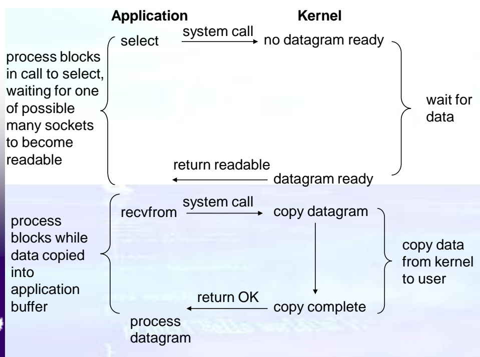
3

Nonblocking I/O Model



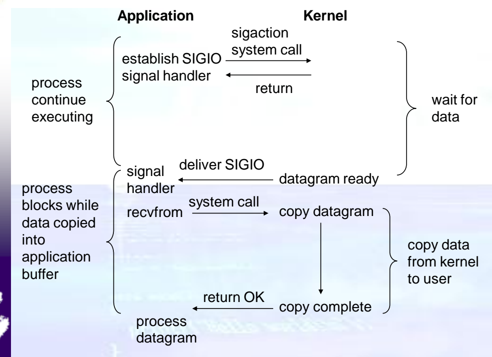
4

I/O Multiplexing Model



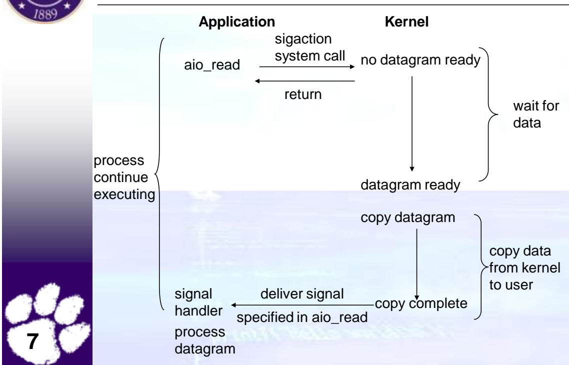
5

Signal-Driven I/O Model

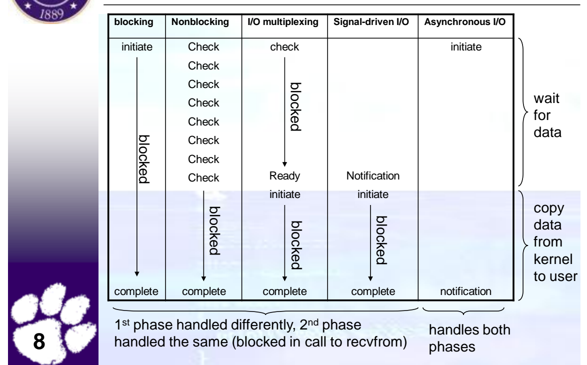


6

Asynchronous I/O Model

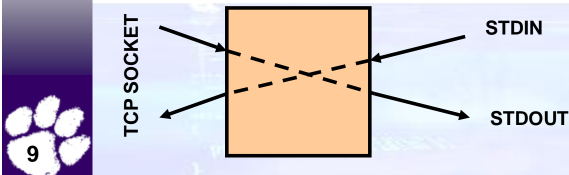


Comparison of the five I/O models



Example - generic TCP client

- **Input from standard input should be sent to a TCP socket.**
- **Input from a TCP socket should be sent to standard output.**
- **How do we know when to check for input from each source?**



Options

- **Use nonblocking I/O.**
  - use fcntl() to set O_NONBLOCK
- **Use alarm and signal handler to interrupt slow system calls.**
- **Use multiple processes/threads.**
- **Use functions that support checking of multiple input sources at the same time.**

Nonblocking I/O

- **use** `fcntl()` **to set** `O_NONBLOCK`**:**

```
int flags;
flags = fcntl(sock,F_GETFL,0);
fcntl(sock,F_SETFL,flags | O_NONBLOCK);
```

- **Now calls to** `read()` **(and other system calls) will return an error and set** `errno` **to** `EWOULDBLOCK`**.**

Non blocking I/O

```
while (! done) {
  if ( (n=read(STDIN_FILENO,…)<0))
      if (errno != EWOULDBLOCK)
          /* ERROR */
  else write(tcpsock,…)

  if ( (n=read(tcpsock,…)<0))
      if (errno != EWOULDBLOCK)
          /* ERROR */
  else write(STDOUT_FILENO,…)
}
```

The problem with nonblocking I/O

- **Using blocking I/O allows the Operating System to put your process to sleep when nothing is happening (no input). Once input arrives, the OS will wake up your process and read() (or whatever) will return.**
- **With nonblocking I/O, the process will chew up all available processor time!!!**

Using alarms

```
signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(STDIN_FILENO,…);
...

signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(tcpsock,…);
...
```

A function you write

- **What will happen to the response time ?**
- **What is the 'right' value for MAX_TIME?**

select()

- **This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more these events occurs or when a specified time has passed.**
- **The select() system call allows us to use blocking I/O on a *set* of descriptors (file, socket, …).**
- **For example, we can ask select to notify us when data is available for reading on either STDIN or a TCP socket.**

select()

```
int select( int maxfd,
            fd_set *readset,
            fd_set *writeset,
            fd_set *excepset,
          const struct timeval *timeout);
```

`maxfd`: **highest number assigned to a descriptor+1.**
`readset`: **set of descriptors we want to read from.**
`writeset`: **set of descriptors we want to write to.**
`excepset`: **set of descriptors to watch for exceptions.**
`timeout`: **maximum time select should wait**

struct timeval

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

- **Wait forever – return only when one of the specified descriptors is ready for I/O.**
  set null pointer.
- **Wait up to fixed amount of time - return only when one of the specified descriptors is ready for I/O, but don't wait beyond timeval.**
  `struct timeval max = {1,0};/* 1 second */`
- **Do not wait at all – return immediately after checking the descriptor. This is called polling.**
  `struct timeval max = {0,0};`

When is a Descriptor ready?

Condition	Readable?	Writable?	Exception?
Data to read	x		
Read half of the connection closed	x		
New connection ready for listening socket	x		
Space available for writing		x	
Write half of the connection closed.		x	
Pending error	x	x	
TCP out-of-band data			x

fd_set

- **Implementation is not important**
- **Operations you can use with an `fd_set`:**

```
void FD_ZERO( fd_set *fdset);
void FD_SET( int fd, fd_set *fdset);
void FD_CLR( int fd, fd_set *fdset);
int FD_ISSET( int fd, fd_set *fdset);
```

```
fd_set rset;
FD_ZERO(&rset); /* initialize the set, all bits off */
FD_SET(1, &rset); /* turn on bit for fd 1 */
```

19

Using select()

- **Create `fd_set`**
- **Clear the whole thing with `FD_ZERO`**
- **Add each descriptor you want to watch using `FD_SET`.**
- **Call `select`**
- **when `select` returns, use `FD_ISSET` to see if I/O is possible on each descriptor.**
- **Sample code:**
  http://www.sbin.org/doc/unix-faq/examples.tar.gz

20

Example

```
struct sockaddr_in addr;
fd_set readset, writeset;
int sock1, sock2, newsoc, max, n;
char buf[1000];
sock1 = socket(…);  sock2 = socket(…);
FD_ZERO(&readset);
FD_SET(sock1, &readset); if(sock1 > max) max = sock1;
FD_SET(sock2, &readset); if(sock2 > max) max = sock2;
res = select(max+1, &readset, NULL, NULL, NULL);
if(FD_ISSET(sock1, &readset)) { // sock1 is ready
    len = sizeof(addr);
    n= recvfrom(sock1, buff, 1000, (struct sockaddr *) &addr, &len);
}
if(FD_ISSET(sock2, &readset)) { // sock2 is ready
    len = sizeof(addr);
    n= recvfrom(sock2, buff, 1000, (struct sockaddr *) &addr, &len);
}
```

21

22