Introduction to Communication - Sockets

David E. Culler
CS162 – Operating Systems and Systems Programming
Lecture 8
Sept 24, 2019

HW 2 out, Due 10/12
Proj 1
Producer/Consumer Problem

• With multiple threads, each waits for the other to make process
• Scheduling constraints:
  – Consumer waits for producer if buffer is empty
  – Producer waits for consumer if buffer is full
• Mutual Exclusion: Only one thread manipulates the buffer data structure at a time
Condition Variables

• Collection of threads waiting *inside* a critical section

• Operations:
  
  – *wait*(&lock): Atomically release lock and go to sleep. *Re-acquire* the lock before returning.

  – *signal()*: Wake up one waiting thread (if there is one)

  – *broadcast()*: Wake up all waiting threads

• Rule: Hold lock when using a condition variable
Producer/Consumer Buffer – 2nd cut

mutex buf_lock = <initially unlocked>
Condvar buf_signal = <initially nobody>

Producer(item) {
    lock buffer
    while (buffer full) { cond_wait(buf_signal, buf_lock) }
    Enqueue(item);
    cond_broadcast(buf_signal);
    unlock buffer
}

Consumer() {
    lock buffer
    while (buffer empty) { cond_wait(buf_signal, buf_lock) }
    item = queue();
    cond_broadcast(buf_signal, buf_lock);
    unlock buffer
    return item
}

Release lock; signal others to run; reacquire on resume
n.b. OS must do the reacquire
Why User must recheck?
Why the while Loop?

• When a thread is woken up by `signal()` or `broadcast()`, it is simply put on the ready queue
  
  • It may or may not reacquire the lock immediately!
    
    – Another thread could be scheduled first and "sneak in" to empty the queue
    
    – Need a loop to re-check condition on wakeup
Recall: Semaphore Solution

Semaphore fullSlots = 0; // Queue empty to start
Semaphore emptySlots = bufSize; // All slots empty
Semaphore mutex = 1; // No one in critical sect.

Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
Recall: UNIX System Structure

<table>
<thead>
<tr>
<th>User Mode</th>
<th>Kernel Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
<td>Kernel</td>
</tr>
<tr>
<td>(the users)</td>
<td>System-call interface to the kernel</td>
</tr>
<tr>
<td>Standard Libs</td>
<td>Kernel interface to the hardware</td>
</tr>
<tr>
<td>shells and commands</td>
<td>CPU scheduling</td>
</tr>
<tr>
<td>compilers and interpreters</td>
<td>page replacement</td>
</tr>
<tr>
<td>system libraries</td>
<td>demand paging</td>
</tr>
</tbody>
</table>

- **Kernel Mode**
  - signals terminal handling
  - character I/O system
  - terminal drivers
  - file system
  - swapping block I/O system
  - disk and tape drivers
  - CPU scheduling
  - page replacement
  - demand paging
  - virtual memory

- **Hardware**
  - terminal controllers
  - terminals
  - device controllers
  - disks and tapes
  - memory controllers
  - physical memory
A Kind of Narrow Waist

User

System

Portable OS Library

System Call Interface

Portable OS Kernel

Platform support, Device Drivers

Application / Service

Compilers

Word Processing

Email

Web Servers

Databases

Web Browsers

Email

Word Processing

Compilers

Portable OS Library

System Call Interface

Portable OS Kernel

Platform support, Device Drivers

Application / Service

User

System

Hardware

Software

x86

PowerPC

ARM

Ethernet (1Gbs/10Gbs)

802.11 a/g/n/ac

SCSI

Graphics

Thunderbolt

9/22/2019

CS162 ©UCB Fa19

Lec 8.8
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Server

Kernel

Hardware

Request

Reply
Recall: Key Unix I/O Design Concepts

- Uniformity
  - file operations, device I/O, and interprocess communication through open, read/write, close
  - Allows simple composition of programs
    » find | grep | wc ...

- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures

- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes

- Kernel buffered reads
  - Streaming and block devices looks the same
  - read blocks process, yielding processor to other task

- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue

- Explicit close
How can a process communicate with another?

write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);

- Producer and Consumer of a file may be distinct processes
  - Also separated in time (one writes and then one later reads)
- However, what if data written once and consumed once?
  - Don't we want something more like a queue?
  - Can still look like File I/O!
Communication between processes

• Can we view files as communication channels?

write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);

• We have seen one example – pipes
• Routinely used with the shell
  >>> grep list src/*/*/*.c | more
Communication Across the world looks like file IO

write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);

• Connected queues over the Internet
  – But what’s the analog of open?
  – What is the namespace?
  – How are they connected in time?
What Is A Protocol?

- A protocol is an **agreement on how to communicate**
- **Includes**
  - **Syntax**: how a communication is specified & structured
    - Format, order messages are sent and received
  - **Semantics**: what a communication means
    - Actions taken when transmitting, receiving, or when a timer expires
- Described formally by a state machine
  - Often represented as a message transaction diagram
Examples of Protocols in Human Interactions

• Telephone
  1. (Pick up / open up the phone)
  2. Listen for a dial tone / see that you have service
  3. Dial
  4. Should hear ringing …
  5. Callee: “Hello?”
  6. Caller: “Hi, it’s John….”
     Or: “Hi, it’s me” (← what’s that about?)
  7. Caller: “Hey, do you think … blah blah blah …” pause
  8. Callee: “Yeah, blah blah blah …” pause
  9. Caller: Bye
  10. Callee: Bye
  11. Hang up
Examples of Protocols in Human Interactions

Asking a question

1. Raise your hand
2. Wait to be called on
3. Or: wait for speaker to pause and vocalize
Clients and Servers

• Client program
  – Running on end host
  – Requests service
  – E.g., Web browser

GET /index.html
Clients and Servers

- **Client program**
  - Running on end host
  - Requests service
  - E.g., Web browser

- **Server program**
  - Running on end host
  - Provides service
  - E.g., Web server

```
GET /index.html
```

"Site under construction"
Request Response Protocol

Client (issues requests)
write(rqfd, rqbuf, buflen);

wait

responses

Server (performs operations)
requests

n = read(rfd, rbuf, rmax);
service request
write(wfd, respbuf, len);

n = read(resfd, resbuf, resmax);
Request Response Protocol

Client (issues requests)

write(rqfd, rqbuf, buflen);

Server (performs operations)

n = read(rfd, rbuf, rmax);

service request

write(wfd, resbuf, len);

n = read(resfd, resbuf, resmax);

wait

requests

responses
Client-Server Models

- File servers, web, FTP, Databases, …
- Many clients accessing a common server
Client-Server Communication

- **Client “sometimes on”**
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn’t communicate directly with other clients
  - Needs to know the server’s address

- **Server is “always on”**
  - Services requests from many client hosts
  - E.g., Web server for the www.cnn.com Web site
  - Doesn’t initiate contact with the clients
  - Needs a fixed, well-known address
Sockets

• **Socket:** an abstraction of a network I/O queue
  – Mechanism for inter-process communication
  – Embodies one side of a communication channel
    » Same interface regardless of location of other end
    » Could be local machine (called “UNIX socket”) or remote machine (called “network socket”)
  – First introduced in 4.2 BSD UNIX: big innovation at time
    » Now most operating systems provide some notion of socket

• Data transfer like files
  – Read / Write against a descriptor

• Over ANY kind of network
  – Local to a machine
  – Over the internet (TCP/IP, UDP/IP)
  – OSI, Appletalk, SNA, IPX, SIP, NS, …
Silly Echo Server – running example

Client (issues requests)  Server (performs operations)

fgets(buf,BUF_SIZE,stdin);

write(fd, buf,BUF_SIZE);

requests

n = read(fd,buf,);

responses

write(fd, buf,);

print

wait

fgets(buf,BUF_SIZE,stdin);

n = read(fd,buf, BUF_SIZE);

print

print

n = read(fd,buf,)

write(fd, buf,);
Echo client-server example

**client**

```c
char buf[BUF_SIZE];
fgets(buf, BUF_SIZE, stdin); // prompt
write(sockfd, buf, strlen(sndbuf)); // send request
memset(buf, 0, BUF_SIZE); // clear
read(sockfd, buf, BUF_SIZE-1); // receive response
printf("%s\n", buf); // echo
```

**server**

```c
char buf[BUF_SIZE];
memset(buf, 0, BUF_SIZE);
read(consockfd, reqbuf, MAXREQ-1); // receive response
printf("%s\n", buf); // echo
write(consockfd, buf, strlen(reqbuf)); // send response
```
What assumptions are we making?

- Reliable
  - Write to a file => Read it back. Nothing is lost.
  - Write to a (TCP) socket => Read from the other side, same.
  - Like pipes
- In order (sequential stream)
  - Write X then write Y => read gets X then read gets Y
- When ready?
  - File read gets whatever is there at the time. Assumes writing already took place.
  - Like pipes!
Socket creation and connection

- File systems provide a collection of permanent objects in structured name space
  - Processes open, read/write/close them
  - Files exist independent of the processes
- Sockets provide a means for processes to communicate (transfer data) to other processes.
- Creation and connection is more complex
- Form 2-way pipes between processes
  - Possibly worlds away
- How do we name them?
- How do these completely independent programs know that the other wants to “talk” to them?
Namespaces for communication over IP

- **Hostname**
  - www.eecs.berkeley.edu

- **IP address**
  - 128.32.244.172 (IPv4, 32-bit Integer)
  - 2607:f140:0:81::f (IPv6, 128-bit Integer)

- **Port Number**
  - 0-1023 are “well known” or “system” ports
    » Superuser privileges to bind to one
  - 1024 – 49151 are “registered” ports
    » Assigned by IANA for specific services
  - 49152–65535 ($2^{15} + 2^{14}$ to $2^{16} - 1$) are “dynamic” or “private”
    » Automatically allocated as “ephemeral Ports”
How do they “shake hands”? 

• How does the server know that a client wants to make a request of them?
• How does a client know that the server is accepting requests?
Socket Setup over TCP/IP

- Special kind of socket: server socket
  - Has file descriptor
  - Can't read or write

- Two operations:
  1. `listen()`: Start allowing clients to connect
  2. `accept()`: Create a new socket for a particular client connection
Socket Setup over TCP/IP

- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)

- Often, Client Port “randomly” assigned
  - Done by OS during client socket setup

- Server Port often “well known”
  - 80 (web), 443 (secure web), 25 (sendmail), etc
  - Well-known ports from 0—1023
Sockets in Schematic

**Client**
- Create Client Socket
- Connect it to server (host:port)
- Connection Socket
- write request
- read response
- Close Client Socket

**Server**
- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
- Connection Socket
- read request
- write response
- Close Connection Socket
- Close Server Socket
char *host_name, port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype, server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

/* Clean up on termination */
close(sock_fd);
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
server->ai_socktype, server->ai_protocol);

// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
// Accept a new client connection, obtaining a new socket
int conn_socket = accept(server_socket, NULL, NULL);
serve_client(conn_socket);
close(conn_socket);
}
close(server_socket);
How does the server protect itself?

- Isolate the handling of each connection
- By forking it off as another process
Sockets With Protection

**Client**

1. Create Client Socket
2. Connect it to server (host:port)
3. Connection Socket
4. write request
5. read response
6. Close Client Socket

**Server**

1. Create Server Socket
2. Bind it to an Address (host:port)
3. Listen for Connection
4. Accept syscall()
5. Connection Socket
6. Close Listen Socket
7. read request
8. write response
9. Close Connection Socket
10. Close Connection Socket
11. Close Server Socket
12. Wait for child
13. Close Listen Socket
14. read request
15. write response
16. Close Connection Socket
17. Close Listen Socket
18. Close Server Socket
19. Parent
20. Child
Server Protocol (v2)

// Socket setup code elided...

while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    pid_t pid = fork();
    if (pid == 0) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else {
        close(conn_socket);
        wait(NULL);
    }
}

close(server_socket);
Concurrent Server

- Listen will queue requests
- Buffering present elsewhere
- But server waits for each connection to terminate before initiating the next
Sockets With Protection and Parallelism

Client

Create Client Socket

Connect it to server (host:port)

Connection Socket

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Child

write request

read response

Close Connection Socket

Close Listen Socket

write response

Close Connection Socket

Parent

Close Connection Socket

Close Server Socket
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {
    close(conn_socket);
    //wait(NULL);
  }
}

close(server_socket);
Concurrent Server without Protection

- Spawn a new thread to handle each connection
- Main thread initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  - More efficient to create new threads
  - More efficient to switch between threads
Sockets With Parallelism, Without Protection

**Client**

- Create Client Socket
- Connect it to server (host:port)
- Connection Socket
- write request
- read response
- Close Client Socket

**Server**

- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
- Spawned Thread
- Connection Socket
- read request
- write response
- Close Connection Socket
- Close Server Socket

Sparse Thread

Main Thread

pthread_create
Server Address - itself

```c
struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    getaddrinfo(NULL, port, &hints, &server);
    return server;
}
```

- Simple form
- Internet Protocol, TCP
- Accepting any connections on the specified port
Client: getting the server address

```c
struct addrinfo *lookup_host(char *host_name, char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    int rv = getaddrinfo(host_name, port_name, &hints, &server);
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}
```
Conclusion (I)

• System Call Interface is “narrow waist” between user programs and kernel

• Streaming I/O: modeled as a stream of bytes
  – Most streaming I/O functions start with “f” (like “fread”)
  – Data buffered automatically by C-library functions

• Low-level I/O:
  – File descriptors are integers
  – Low-level I/O supported directly at system call level

• `STDIN / STDOUT` enable composition in Unix
  – Use of pipe symbols connects `STDOUT` and `STDIN`

  » find | grep | wc ...
Conclusion (II)

• Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  – Supports a standard, internal interface
  – Same kernel I/O system can interact easily with different device drivers

• File abstraction works for inter-processes communication (local or Internet)

• Socket: an abstraction of a network I/O queue
  – Mechanism for inter-process communication