Agenda

• Under-the-hood look at system calls
  – Data buffering and blocking

• Processes, threads, and concurrency models

• Event-based, non-blocking I/O
Recall Interprocess Communication

• Processes must communicate to cooperate

• Must have two mechanisms:
  – Data transfer
  – Synchronization

• Shared memory not an option across network
Message Passing (local)

- Operating system mechanism for IPC
  - send (destination, message_buffer)
  - receive (source, message_buffer)
- Data transfer: in to and out of kernel message buffers
- Synchronization: ?
Where is the synchronization in message passing IPC?

A. The OS adds synchronization.

B. Synchronization is determined by the order of sends and receives.

C. The communicating processes exchange synchronization messages (lock/unlock).

D. There is no synchronization mechanism.
Message Passing (network)

- Same synchronization
- Data transfer
  - Copy to/from OS socket buffer
  - Extra step across network: hidden from applications
socket()

- OS stores a table, per process, of descriptors
socket()

- socket() returns a socket descriptor
- Indexes into table

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
    7
```
socket()

- OS stores details of the socket, connection, and pointers to buffers

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

- Family: AF_INET, Type: SOCK_STREAM
- Local address: NULL, Local port: NULL
- Send buffer, Receive buffer

Kernel

Process
int sock = socket(AF_INET, SOCK_STREAM, 0);  

• OS stores details of the socket, connection, and pointers to buffers
recv()
What should we do if the receive socket buffer is empty? If it has 11 bytes?

Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
(assume we connect()ed here...)
int recv_val = recv(sock, r_buf, 22, 0);
```

Socket buffer (receive)

- Empty
- 11 bytes

Kernel
What should we do if the receive socket buffer is empty? If it has 11 bytes?

Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
(assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 22, 0);
```

---

<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
<th>11 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td>B</td>
<td>Block</td>
<td>Copy 11 bytes</td>
</tr>
<tr>
<td>C</td>
<td>Copy 0 bytes</td>
<td>Block</td>
</tr>
<tr>
<td>D</td>
<td>Copy 0 bytes</td>
<td>Copy 11 bytes</td>
</tr>
<tr>
<td>E</td>
<td>Something else</td>
<td></td>
</tr>
</tbody>
</table>

---

Socket buffer (receive)

Kernel

r_buf (size 22)
What should we do if the send socket buffer is full? If it has 11 bytes?

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
(assume we connect()ed here...)
int send_val = send(sock, s_buf, 22, 0);
```

**Socket buffer (send)**

- Full
- 11 bytes

**s_buf (size 22)**
What should we do if the send socket buffer is full? If it has 11 bytes?

**Process**

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
(assume we connect()ed here...)
int send_val = send(sock, s_buf, 22, 0);
```

---

<table>
<thead>
<tr>
<th></th>
<th>Full</th>
<th>11 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Return 0</td>
<td>Copy 11 bytes</td>
</tr>
<tr>
<td>B</td>
<td>Block</td>
<td>Copy 11 bytes</td>
</tr>
<tr>
<td>C</td>
<td>Return 0</td>
<td>Block</td>
</tr>
<tr>
<td>D</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td>E</td>
<td>Something else</td>
<td></td>
</tr>
</tbody>
</table>

---

**Socket buffer (send)**

- **Full**: 11 bytes filled.
- **11 bytes**: 11 bytes available for sending.

---

**Kernel**
Blocking Summary

send()
• Blocks when socket buffer for sending is full
• Returns less than requested size when buffer cannot hold full size

recv()
• Blocks when socket buffer for receiving is empty
• Returns less than requested size when buffer has less than full size

Always check the return value!
Concurrency

• Think you’re the only one talking to that server?
Without Concurrency

• Think you’re the only one talking to that server?
Without Concurrency

- Think you’re the only one talking to that server?

Client taking its time...

If only we could handle these connections separately...

Web Server

recv() request

Server Process Blocked!

Ready to send, but server still blocked on first client.
Multiple Processes

Web Server

Child process recv()s

Server fork()s

Web Server

Web Server

Services the new client request

Client

Client
# Processes/Threads vs. Parent

(More details in an OS class...)

<table>
<thead>
<tr>
<th>Spawned Process</th>
<th>Spawned Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherits descriptor table</td>
<td>Shares descriptor table</td>
</tr>
<tr>
<td>Does not share memory</td>
<td>Shares memory</td>
</tr>
<tr>
<td>– New memory address space</td>
<td>– Uses parent’s address space</td>
</tr>
<tr>
<td>Scheduled independently</td>
<td>Scheduled independently</td>
</tr>
<tr>
<td>– Separate execution context</td>
<td>– Separate execution context</td>
</tr>
<tr>
<td>– Can block independently</td>
<td>– Can block independently</td>
</tr>
</tbody>
</table>
## Processes/Threads vs. Parent

(More details in an OS class...)

<table>
<thead>
<tr>
<th>Spawned Process</th>
<th>Spawned Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherits descriptor table</td>
<td>Shares descriptor table</td>
</tr>
<tr>
<td>Does not share memory</td>
<td>Shares memory</td>
</tr>
<tr>
<td>- New memory address space</td>
<td>- Uses parent’s address space</td>
</tr>
<tr>
<td>Scheduled independently</td>
<td>Scheduled independently</td>
</tr>
<tr>
<td>- Separate execution context</td>
<td>- Separate execution context</td>
</tr>
<tr>
<td>- Can block independently</td>
<td>- Can block independently</td>
</tr>
</tbody>
</table>

Often, we don’t need the extra isolation of a separate address space. Faster to skip creating it and share with parent – threading.
Threads & Sharing

• Global variables and static objects are shared
  – Stored in the static data segment, accessible by any thread

• Dynamic objects and other heap objects are shared
  – Allocated from heap with malloc/free or new/delete

• Local variables are not shared
  – Refer to data on the stack
  – Each thread has its own stack
  – Never pass/share/store a pointer to a local variable on another thread’s stack
Whether processes or threads...

• Several benefits
  – Modularizes code:
    • one piece accepts connections, another services them
  – Each can be scheduled on a separate CPU
  – Blocking I/O can be overlapped
Which benefit is the most critical?

A. Modular code/separation of concerns.

B. Multiple CPU/core parallelism.

C. I/O overlapping.

D. Some other benefit.
Whether processes or threads...

• Several benefits
  – Modularizes code:
    • one piece accepts connections, another services them
  – Each can be scheduled on a separate CPU
  – Blocking I/O can be overlapped

• Still not maximum efficiency...
  – Creating/destroying threads still takes time
  – Requires memory to store thread execution state
  – Lots of context switching overhead
Non-blocking I/O

- One operation: add a flag to send/recv
- Permanently, for socket: fcntl() – “file control”
  - Allows setting options on file/socket descriptors

```c
int sock, result, flags = 0;
sock = socket(AF_INET, SOCK_STREAM, 0);
result = fcntl(sock, F_SETFL, flags | O_NONBLOCK)

check result – 0 on success
```
Non-blocking I/O

• With O_NONBLOCK set on a socket
  – No operations will block!

• On recv(), if socket buffer is empty:
  – returns -1, errno is EAGAIN or EWOULDBLOCK

• On send(), if socket buffer is full:
  – returns -1, errno is EAGAIN or EWOULDBLOCK
How about...

server_socket = socket(), bind(), listen()
connections = []

while (1)
    new_connection = accept(server_socket)
    if new_connection != -1, add it to connections
    for connection in connections:
        recv(connection, ...)  // Try to receive
        send(connection, ...) // Try to send, if needed
    }
Will this work?

server_socket = socket(), bind(), listen()
connections = []

while (1)
    new_connection = accept(server_socket)
    if new_connection != -1, add it to connections
    for connection in connections:
        recv(connection, ...) // Try to receive
        send(connection, ...) // Try to send, if needed

A. Yes, this will work.        C. No, this will use too many resources.
B. No, this will execute too slowly. D. No, this will still block.
Event-based Concurrency

• Rather than checking over and over, let the OS tell us when data can be read/written

• Create set of FDs we want to read and write

• Tell system to block until at least one of those is ready for us to use. The OS worry about selecting which one.

   select()
int main(void) {
    fd_set rfds;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input. */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

    /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
    /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");
}

- More interesting example in the select_tut man page.
- Beej’s guide also has a good example.
- You’ll use it in a future lab!
Event-based Concurrency

• Rather than checking over and over, let the OS tell us when data can be read/written

• Tell system to block until at least one of those is ready for us to use. The OS worry about selecting which one.

• Only one process/thread (or one per core)
  – No time wasted on context switching
  – No memory overhead for many processes/threads
Reading

• Next class: Naming and DNS
  – Section 2.5

• Lab 2: Multi-threaded Web server
  – Due Thursday, September 24