CSE 124: OPTIONS, SIGNALS, TIMEOUTS, AND CONCURRENCY

George Porter
Oct 13, 2017
• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
ANNOUNCEMENTS

HW 2 out

Project 1 deadline will be pushed back a bit (TBD)
   But do start on it!
Outline

1. Socket internals
2. Socket options
3. Signals and timeouts
4. Concurrency
rv = connect(s,...);

rv = send(s,buffer0,1000,0);

rv = send(s,buffer1,2000,0);

rv = send(s,buffer2,5000,0);

close(s);
AFTER 3 SEND() CALLS

1. First send call (1000 bytes)
2. Second send call (2000 bytes)
3. Third send call (5000 bytes)
AFTER FIRST RECV()
AFTER ANOTHER RECV()
WHEN DOES BLOCKING OCCUR?

- SendQ size: SQS
- RecvQ size: RQS

```c
send(s, buffer, n, 0);
```

- \( n > SQS \): blocks until \( (n - SQS) \) bytes xferred to RecvQ
- If \( n > (SQS + RQS) \), blocks until receiver calls `recv()` enough to read in \( n - (SQS + RQS) \) bytes

- How does this lead to deadlock?
  - Trivial cause: both sides call `recv()` w/o sending data
MORE SUBTLE REASON FOR DEADLOCK

- SendQ size = 500; RecvQ size = 500
Outline

1. Socket internals
2. Socket options
3. Signals and timeouts
4. Concurrency
Basic “out of the box” socket functionality fine for most purposes
  • But what if you need to tweak the behavior?

Can set/get ‘options’ on sockets

These options apply to different layers of the network stack:
  • IP
  • TCP
  • Socket
TCP BUFFER SIZE OPTIONS

- Send and receive buffer sizes
  - What is the default?

```
[gmporter@seed-f60-100 ~]$ cat /proc/sys/net/ipv4/tcp_rmem
4096  87380  6291456  # Minimum  Maximum
[gmporter@seed-f60-100 ~]$ cat /proc/sys/net/ipv4/tcp_wmem
4096  16384  4194304  # Minimum  Maximum
```

- Can we change that value?
  - Yes!
SETTING/GETTING SOCKET OPTIONS

GETSOCKOPT(2) Linux Programmer's Manual GETSOCKOPT(2)

NAME
getsockopt, setsockopt — get and set options on sockets

SYNOPSIS
#include <sys/types.h>    /* See NOTES */
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname,
                void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname,
                const void *optval, socklen_t optlen);

<table>
<thead>
<tr>
<th>Level</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOL_SOCKET</td>
<td>SO_SNDBUF</td>
<td>Send buffer size</td>
</tr>
<tr>
<td></td>
<td>SO_REUSEADDR</td>
<td>Allow TCP port to be reused instantly</td>
</tr>
<tr>
<td></td>
<td>SO_RCVTIMEO</td>
<td>Set a recv() timeout</td>
</tr>
<tr>
<td></td>
<td>SO_SNDTIMEO</td>
<td>Set a send() timeout</td>
</tr>
</tbody>
</table>
int optval = 1;

/* enable sockets to be immediately reused */
if (setsockopt(serv_sock, SOL_SOCKET,
                  SO_REUSEADDR, &optval, sizeof(optval)) != 0)
{
    die_system("setsockopt() failed");
}
sendfile - transfer data between file descriptors

**SYNOPSIS**

```c
#include <sys/sendfile.h>

ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```
Outline

1. Socket internals
2. Socket options
3. Signals and timeouts
4. Concurrency
OS SIGNALS

- OS mechanism to asynchronously interrupt a program

- Why is this useful?
  - Kill a runaway/hung process
  - Notify program that there is activity on the keyboard
  - Disk read operation has completed
  - The dreaded SIGSEGV
Signals and Networking

- Signals can be used to implement timeouts
- Examples:
  - Close connection after 3 minutes of inactivity
  - HTTP server: is the client going to send another request? Set timeout for e.g., 5 seconds
- Useful any time you need to stop blocking
  - recv()
  - send()
  - ...
SIGNALS: ALARMS

• SIGALRM
  • Issued after a set period of time goes by
  • Like an alarm clock for your program
• Others in D&C Chapter 6.2
SETTING UP EVENT HANDLERS

SIGACTION(2) Linux Programmer's Manual SIGACTION(2)

NAME

sigaction - examine and change a signal action

SYNOPSIS

#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
               struct sigaction *oldact);

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};

Function to handle event

How to handle other events during the handling of this event

Rest of fields can be set to NULL/0
HOW TO USE SIGNALS

• **Define** the event handling function
  
  • `void myfun(int signal);`

• **Associate** that function with the signal you want to handle
  
  • `sigaction()` call
WHAT DOES ‘MASK’ MEAN?

• Signals arrive unpredictably and asynchronously

• Get a SIGINT or SIGTERM for example

• What happens if, if your handler for SIGINT, another SIGINT comes?

• Can simplify our handler by masking signals during our event handler

• Helper functions provided (e.g., sigfillset(…))
What happens to a blocking call when an event comes in?

- Control transferred to event handler
- When control returned, the blocking stops, and an error code is returned

Recv()

- Might return fewer bytes than requested, or EINTR return code if no bytes received

Send()

- Might send fewer bytes than requested or EINTR if no bytes sent
Outline

1. Socket internals
2. Socket options
3. Signals and timeouts
4. Concurrency
Outline

1. Socket internals
2. Socket options
3. Signals and timeouts
4. Concurrency
CONCURRENCY VS PARALLELISM

• Both deal with doing a lot at once, but aren’t the same thing
  • Given set of tasks \( \{T_1, T_2, \ldots, T_n\} \)

• Concurrency:
  • Progress of multiple elements of the set overlap in time

• Parallelism:
  • Progress on elements of the set occur at the same time
CONCURRENCY

- Might be parallel, might not be parallel

- A single thread of execution can **time slice** a set of tasks to make **partial progress over time**
  - Time 0: Work on first 25% of Task 0
  - Time 1: Work on first 25% of Task 1
  - Time 2: Work on first 25% of Task 2
  - Time 3: Work on first 25% of Task 3
  - Time 4: Work on second 25% of Task 0
  - Time 5: Work on second 25% of Task 1
  - ...
Multiple execution units enable progress to be made simultaneously

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time 0: 1\textsuperscript{st} 25% of Task1</td>
<td>Time 0: 1\textsuperscript{st} 25% of Task2</td>
</tr>
<tr>
<td>Time 1: 2\textsuperscript{nd} 25% of Task1</td>
<td>Time 1: 2\textsuperscript{nd} 25% of Task2</td>
</tr>
<tr>
<td>Time 2: 3\textsuperscript{rd} 25% of Task1</td>
<td>Time 2: 3\textsuperscript{rd} 25% of Task2</td>
</tr>
<tr>
<td>Time 3: 4\textsuperscript{th} 25% of Task1</td>
<td>Time 3: 4\textsuperscript{th} 25% of Task2</td>
</tr>
<tr>
<td>Time 4: 1\textsuperscript{st} 25% of Task3</td>
<td>Time 4: 1\textsuperscript{st} 25% of Task4</td>
</tr>
</tbody>
</table>
FLASH TRAFFIC

- USGS Pasadena, CA office Earthquake site
- Oct 16, 1999 earthquake
Too much parallelism causes thrashing, excessive switching, lower performance.
Follow along at:

https://github.com/gmporter/cse124-lec-concurrency
UC San Diego