



Linux Programming

Topics

- 1) How can we do **multitasking**?
- 2) How can our **multiple tasks communicate**?
- 3) How can we **communicate** over the **network**?



Concurrency: Processes & Threads

Processes: fork() / exec__()

- Each process has a separate..
- `fork():...`
- `exec__()`: replaces current process with an executable file.

```
pid_t child_pid = fork();
if (child_pid != 0)
    printf ("Parent process: id %d\n", (int) getpid());
else {
    printf ("Child process: id %d\n", (int) getpid());

    // Exchange child for executing /bin/lS
    char *args[] = {"/bin/lS", "-l", "/dev/tty", (char *) 0};
    execv("/bin/lS", args);

    printf("Won't see this!\n"); ...
}
```

Threads

- All threads of a process..

- Thread function:

```
void *myThreadFn(void *args)
{
    // Do stuff
    return NULL;
}
```

Direct access to
shared (global)
variables.

- Call:

```
pthread_t id;
pthread_create(&id, NULL, &myThreadFn, NULL);
```

Thread
attributes

void*
Arguments

- Wait till thread finishes (and cleans up some memory):

```
pthread_join(id, NULL);
```

- #include <pthread.h>

Can be void** to
hold return value
from thread function

Race Case

- Race case
if a memory location (a global variable) is..
 - What is the value of **count** after executed as two threads?

```
#define MAX 1000000
long long count = 0;

void* foo(void* args)
{
    for(long long i = 0; i < MAX; i++) {
        count++;
    }
    return NULL;
}
```

```
COUNT is:    1107469
Off by:      892531
```

- What helps? **volatile**? **static**?

__Atomic

- ..
- Add __Atomic to a type to make updates atomic (including ++)

```
#define MAX 1000000
__Atomic long long count = 0;

void* foo(void* args)
{
    for(long long i = 0; i < MAX; i++) {
        count++;
    }
    return NULL;
}
```

COUNT is: 2000000

Works On

Primitives

```
__Atomic int count;
__Atomic unsigned char ch;
```

Pointers

```
__Atomic long long *pValue;
```

Does Not Work On

Structs / Unions / Arrays

```
typedef struct {
    long count;
} sData;
__Atomic sData bad;
...
bad.count++;
```



But Does Work On

Structs Fields

```
typedef struct {
    __Atomic long count;
} sData;
sData data;
...
data.count++;
```

<stdatomic.h>

- <stdatomic.h> defines some useful types
 - Nothing special, just for convenience

```
typedef _Atomic _Bool          atomic_bool;
typedef _Atomic char           atomic_char;
typedef _Atomic int            atomic_int;
typedef _Atomic unsigned int   atomic_uint;
typedef _Atomic long           atomic_long;
typedef _Atomic unsigned long  atomic_ulong;
typedef _Atomic long long      atomic_llong;
typedef _Atomic unsigned long long atomic_ullong;
typedef _Atomic __CHAR16_TYPE__ atomic_char16_t;
typedef _Atomic __CHAR32_TYPE__ atomic_char32_t;
typedef _Atomic __INTMAX_TYPE__ atomic_intmax_t;
typedef _Atomic __UINTMAX_TYPE__ atomic_uintmax_t;
...
```

Thread Synchronization

- **Mutex:**
 - Control access to critical sections.
 -

- **Create:**

```
pthread_mutex_t myMutex =  
    PTHREAD_MUTEX_INITIALIZER;
```

- **Critical Section:**

```
pthread_mutex_lock(&myMutex);  
{  
    // Do critical stuff here!  
}  
pthread_mutex_unlock(&myMutex);
```

```
static int data[SIZE];  
void foo()  
{  
    int sum = 0;  
    pthread_mutex_lock(&dataMutex);  
    {  
        for (int i = 0; i < SIZE; i++) {  
            sum += data[i];  
        }  
    }  
    pthread_mutex_unlock(&dataMutex);  
    printf("Sum of elements: %d\n", sum);  
}
```

IO outside of
critical section.

Thread considerations

- **Tips for Critical Sections:**
 - Keep critical sections short: avoid blocking other threads.
 - Calculate values with temporary variables; then update shared variables in critical section.
 - Use extra {...} to highlight the critical section.
 - Always unlock!
- **Compiling (linking)**
`aarch64-linux-gnu-gcc -Wall -g demo_thread.c \`
`-o demo_thread -pthread`

Communicating Between Threads

- Code in different threads can interact in many ways
 - ..
Use to signal events between threads.
 - ..
Accessible between threads
(but may need to be atomic or protected by critical sections).
 - .. (next)
Can push data between threads or processes.

Pipes

Basics

- Pipe:
 -
 - Good for inter-thread and inter-process communication.
- Needed Functions:
 - `pipe()` to create file descriptors for read and write ends of pipe.
 - `fdopen()` to open the pipe (from descriptor)
 - `fprintf()` to write (or other functions)
 - `fgets()` to read [blocking] (or other functions)
 - `close()` to close the file descriptor.

Pipe Code

```
// Writer: Convert the write file descriptor  
// to a FILE object  
FILE* streamW = fdopen (fds[1], "w");  
fprintf (streamW, "Hello World!\n");  
fflush (streamW);  
close (fds[1]);
```

```
// File descriptors for pipe ends  
int fds[2];  
// Create a pipe.  
pipe (fds);
```



Likely fork() or
pthread_create()

```
// Reader: Convert read file descriptor to a FILE object.  
FILE* streamR = fdopen (fds[0], "r");  
// Read until end of the stream.  
char buffer[1024];  
while (!feof (streamR) && !ferror (streamR)  
      && fgets (buffer, sizeof (buffer), streamR) != NULL) {  
    printf("%s", buffer);  
}  
close (fds[0]);
```

popen() = Fork & pipe

- Execute a shell command using a pipe for output [or input].

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    // Execute the shell command (output into pipe)
    FILE *pipe = popen("ls -l /dev/tty*", "r");

    // Dump contents of pipe to the screen.
    char buffer[1024];
    while (!feof(pipe) && !ferror(pipe)) {
        if (fgets(buffer, sizeof(buffer), pipe) == NULL)
            break;
        printf("--> %s", buffer);
    }

    // Close pipe, check program's exit code
    int exitCode = WEXITSTATUS(pclose(pipe));
    if (exitCode != 0) {
        printf("program failed: %d\n", exitCode);
    }

    return 0;
}
```

I know a great joke about UDP,
but I'm not sure anyone would get it.

Sockets: Bidirectional network communication



Socket Intro

- Socket
 -
 - Used to send data between processes on the same computer, or across the network.
 - Like a pipe, but works across a network too.
- Use
 - Server:..
 - Usually at a known port number.
 - When data received, it knows client IP and port.
 - Client:..
 - May also wait for a reply.

Socket Types

- **Connection (TCP):**
 - in order delivery, automatic retransmission
 - single connection between specific host and server.
 - Better for long term connections with large amount of data (fetch web-page).
- **Datagram (UDP):**
 - no persistent connection (connectionless):
..
 - Better for short, single packet messages.
- See section 5.5 of Advanced Linux Programming for socket examples.

UDP Constants

- FYI: Here are what some of the socket constants mean:
 - `sockaddr_in`: **S**ocket **A**ddress for **I**Nternet (struct)
 - `sin`: **S**ocket **I**Nternet, such as in `sin_family`
 - `AF_INET`: **A**ddress **F**amily, **I**nternet (IP v4)
 - `PF_INET`: **P**rotocol **F**amily, **I**nternet (IP v4)
 - `SOCK_DGRAM`: **S**ocket, user **D**atagram protocol (**U**DP)

(You don't need to memorize these).

UDP Server Programming (1/3 - Init)

- Address Structure

```
#define MAX_LEN 1024
#define PORT 22110
```

```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(PORT);
```

`_in` means internet

Connection may be from network

`htonl` = host to network long;
`htons` = host to network short

- Create and bind to socket

```
int socketDescriptor = socket(PF_INET, SOCK_DGRAM, 0);
bind(socketDescriptor, (struct sockaddr*) &sin, sizeof(sin));
```

C has numerous socket address structures:
`sockaddr` (generic), `sockaddr_in` (internet), ...
`bind()` accepts a generic `sockaddr` and decides what to do based on the family field (shared by all `sockaddr` structs).

UDP Server Programming (2/3 - Read)

- Receive Data

```
struct sockaddr_in sinRemote;  
unsigned int sin_len = sizeof(sinRemote);  
char messageRx[MAX_LEN];  
  
int bytesRx = recvfrom(socketDescriptor,  
messageRx, MAX_LEN - 1, 0,  
(struct sockaddr *) &sinRemote, &sin_len);
```

Client's data written
into `messageRx` string

`sinRemote` is output parameter;
`sin_len` is in/out parameter.

```
// Null terminated (string):  
messageRx[bytesRx] = 0;
```

... What if recvfrom filled the
buffer 100%? Overflow?

```
printf("Message received (%d bytes): '%s'\n",  
bytesRx, messageRx);
```

UDP Socket Programming (3/3 Reply)

- Create Reply

```
// Watch for buffer overflow!  
char messageTx[MAX_LEN];  
snprintf(messageTx, MAX_LEN, "Hello %d\n", 42);
```

- Send Reply

```
sin_len = sizeof(sinRemote);  
sendto( socketDescriptor,  
messageTx, strlen(messageTx),  
0,  
(struct sockaddr *) &sinRemote, sin_len);
```

Have client's IP address
and port from receiving
the message.

- Close socket (when done)

```
close(socketDescriptor);
```

- May take a few seconds for OS to finish closing.

Byte Order

- - 2 bytes of **0xa1cf** transmitted as **0xa1**, **0xcf**
 - **Big-endian** = network byte order...
 - x86 is little-endian; ARM is bi-endian (supports both)
- Never assume your processor is network order:
use **host-to-network** to adjust:

Prototypes

```
uint32_t htonl(uint32_t hostlong);  
uint16_t htons(uint16_t hostshort);  
uint32_t ntohl(uint32_t netlong);  
uint16_t ntohs(uint16_t netshort);
```

Example

```
#include <netdb.h>
```

```
short toTransmit1 = htons(myVal1);  
long toTransmit2 = htonl(myVal2);
```

Summary

- Use **processes** for coarse multitasking:
 - Use **fork()** and **exec__()**.
 - Example: A server and a client with well defined separate roles.
- Use **threads** for fine-grained multitasking.
 - Use **pthread_create()**, **pthread_join**
 - Use **_Atomic** for shared variables
 - Mutex with **pthread_mutex_t**: **pthread_mutex_lock()**, **pthread_mutex_unlock()**.
- **Pipes** for inter process/thread communication.
- **Sockets** for network communication.