CS61 Section Notes - Process Management and Sockets

We'll be using the section repo today. If you don't have it handy run "git clone git@code.seas.harvard.edu:cs61/cs61-section.git" or otherwise just git pull to get the newest code (it'll be in the s11/ directory). Type "make" and then to run some file foo.c referenced in the notes, just do "./foo".

0. kill()

int kill(pid_t pid, int sig)
The system call kill() takes two arguments, the process ID you want to send a signal to, and the second is the signal you want to send. If successful, kill() returns 0, otherwise the return value is negative.

1. Building on process management

Let's look at how we can use process management system calls to build other interesting system functions.

First, we'll investigate **mutual exclusion locks**. In the last unit of the class, we'll learn how to write **multithreaded** programs. A multithreaded program allows a single process to contain two or more logical processors, which we call **threads** or "threads of control". Threads aren't isolated: all threads in a process share the same memory space. However, each thread has a separate stack. Threads are cool because (for example) on a multicore machine, many processors can work simultaneously on the same data.

A **mutual exclusion lock** has two operations, acquire and release. Acquire and release are paired, kind of like open and close. Acquire() succeeds immediately, *unless* another thread has called acquire() but not release(). That is, at most one thread can have the lock acquired at a time. If another thread calls acquire(), it will block until the lock is release()d.

How can we implement mutual exclusion locks? You might think this would work:

```
typedef struct {
  int islocked;
} mutexlock;

void mutexlock_init(mutexlock* m) {
  m->islocked = 0;
}
```

```
void acquire(mutexlock* m) {
    while (m->islocked)
        /* do nothing */;
    m->islocked = 1;
}

void release(mutexlock* m) {
   m->islocked = 0;
}
```

But that doesn't work—at all! In acquire(), the check of m->islocked and the assignment m->islocked = 1 *do not happen atomically*. If two threads were waiting for the lock, they might *simultaneously* see that m->islocked == 0, and then *simultaneously* set m->islocked = 1. Then two threads would think they had acquired the lock, which is illegal.

There are good ways to solve this problem; we'll see them soon. But we already know one way: the kernel implements most system calls atomically! In particular, it implements fork(), waitpid(), read(), write(), and kill() atomically.

Can you **implement a mutual exclusion lock based on helper processes**? You may assume that no external code will call waitpid() on your helper process(es), and that process IDs are not reused.

Start with this:

```
typedef struct {
    volatile pid_t p;
   // maybe more stuff here
} mutexlock;

void mutexlock_init(mutexlock* m) {
   // your code here
}

void acquire(mutexlock* m) {
   // your code here
}

void release(mutexlock* m) {
   // your code here
}
```

Any mutual exclusion lock requires some operation that waits. For processes, we know that waitpid *can block when waiting for a child to exit. But also read() waits for data to become available.*

Mutual exclusion also requires that at most one thread can have a lock in the acquired state at any instant. We therefore need a system call that gives at-most-once behavior. read *from a pipe gives at-most-once behavior because it reads characters in first-in, first-out order without duplicates or omissions: if one character is in a pipe, and two processes or threads call* read *at the same time, only one of them will read the character.*

These insights lead to the following lock. The lock is in the unlocked state iff there is a character waiting in the `acquirepipe`. The first thread to read that character will get the lock. To release the lock, the thread writes a character to the `releasepipe`, which tells the helper process to go back to the unlocked state.

Here's a version of the solution that uses pipes.

```
typedef struct {
    int acquirepipe[2];
   int releasepipe[2];
} mutexlock;

void mutexlock_init(mutexlock* m) {
   pipe(m->acquirepipe);
   pipe(m->releasepipe);
   if (fork() == 0) {
       close(m->acquirepipe[0]); close(m->releasepipe[1]); // just for general cleanliness
       char ch = '!';
       while (1) {
           write(m->acquirepipe[1], &ch, 1);   // for full correctness, would need a loop
           ssize_t r = read(m->releasepipe[0], &ch, 1);
           if (r == 0) // parent died
               exit(0);
       }
   }
   close(m->acquirepipe[1]);   // just for general cleanliness
   close(m->releasepipe[0]);
}

void acquire(mutexlock* m) {
   char ch;
   read(m->acquirepipe[0], &ch, 1);
}

void release(mutexlock* m) {
```

```
  char ch = '!';
  write(m->releasepipe[1], &ch, 1);
}
```

But there's a simpler choice. waitpid also gives at-most-once behavior, because when a process dies, its status is collected at most once (the zombie is destroyed once the status is collected, so future waitpid attempts on that process ID will fail).

These insights lead to the following lock. The locked state is represented by a living helper process, the unlocked state by a zombie helper process. To release the lock we kill the helper process. To acquire the lock, we use waitpid to try to collect the helper process's state. If the helper is a zombie (the lock is unlocked), waitpid will succeed; we create a new helper process to represent the locked state. If multiple threads try to acquire a lock at once, only one of them will succeed (waitpid will return m->p); the rest of them will return an error since the zombie has been destroyed. The error indicates that some other thread grabbed the lock and we should retry. Finally, to initialize the lock, we create a thread that immediately exits, since the lock should start out in the unlocked state.

```
typedef struct {
    volatile pid_t p;
} mutexlock;

void mutexlock_init(mutexlock* m) {
  m->p = fork();
  if (m->p == 0)
      exit(0);
}

void acquire(mutexlock* m) {
  int s;
   while (waitpid(m->p, &s, 0) == -1)
      /* */;
   m->p = fork();
   if (m->p == 0) {
      while (1)
         sleep(1000000);
  }
}

void release(mutexlock* m) {
  kill(m->p, SIGKILL);
}
```

2. IPC across machines: RPC!

Now that you are all pipe experts, IPC is a breeze and life is good. But what do you do if your processes are on separate machines? **Sockets** to the rescue. How are sockets different from pipes?

The primary difference is where they can be used. Pipes are only usable when both processes are on the same logical machine; while sockets can be used across machines or within a single machine. They also support the additional protocols needed to connect across separate physical machines. This is almost always transmission control protocol (TCP) or user datagram protocol (UDP) over internet protocol (IP). Also, sockets offer bidirectional communication. Because they can connect across machines the two ends behave differently from pipes. One side is a listener (frequently called the "server") and the other side connects to the first (frequently called the "client").

Let's start by making sockets in the shell. First, let's take a look at a super useful tool **netcat** (note that netcat comes in a few flavors. on some systems it's *nc* on others it's *ncat* the versions are slightly different but just as powerful as far as I can tell).

Netcat is self described as the "TCP/IP swiss army knife". It will allow you to make arbitrary socket connections from the shell. For example we can make an instant chat system. Server:

Demo
1) run ifconfig to expose your ip address
2) run netcat in listen mode on port 6667:
    nc -l -p 6667
3) have someone in class run netcat to connect to you (first one who gets there gets to connect):
    nc <ip from step 1> 6667
4) have them say hello…
But it gets better: you can send binary data across the connection too:
5) restart the netcat connection this time redirect to a file:
    nc -l -p 6667 > their_ls
6) have someone in class send you their ls program (note, some versions of nc don't seem to respect the EOF character. That is, when input ends, the connection is held open. you can make the connection close by using the `-q *n*' flag on the client side, which is supposed to wait n seconds after seeing EOF and then close the connection. In my experience the q flag causes a close but it is immediate rather than after the n seconds.):
    nc -q 1 <ip from step 1> 6667 < /bin/ls
7) show that it works (should have the same results):
    chmod +x their_ls
    ./their_ls

ls

Great, so how could we build this tool. Obviously we are going to need the program to build sockets. For this, we need to learn a few more system calls. First, we'll look at an example. Check out s11/readclient.c. This code connects to a socket on a named host and port, reads from that socket, and writes what comes back to the standard output. Try it like this:

```
make
./readclient cs61.seas.harvard.edu 6162
```

What's different between readsocket and netcat?

Netcat also reads from the standard INPUT and writes to the socket.

Cool, let's fix that. Check out bidiclient.c. For example:

```
./bidiclient cs61.seas.harvard.edu 6163
(type some stuff at standard input)
```

And for comparison:

```
nc cs61.seas.harvard.edu 6163
(type some stuff at standard input)
```

For your information, the program running on cs61's port 6163 is cs61echo.c.

What's different between bidiclient and netcat? What different output do you see? Why in the code is this happening?

Bidiclient always reads from the standard input first. So if the remote socket writes something first, it won't show up right away. Also, bidiclient alternates between one read system call per reading file descriptor (standard input and the socket). So if the remote socket has a lot to say (more than one read is necessary), bidiclient will get behind.

What we need is logically to run two procedures in parallel: one that reads from standard input and writes to the socket, and one that reads from the socket and writes to standard output. How can we do this?

There are several ways to handle logically parallel connections. One, called **event-driven programming**, involves system calls that can handle many file descriptors at once. The key event-driven system call is select(), which waits until at least one of its input file descriptors has data ready to read (or space ready to write into). The second, called **multi-threading**, involves

splitting the process into multiple logical threads of control, which run in parallel. These strategies are implemented by bidiclient2.c and bidiclient3.c, respectively.

Important System Calls and System Structures:

Overview:

An Internet Protocol Address or IP address is a user's numerical identifier on the internet that is typically a 32 bit number. A port is a logical connection portal on your computer, represented by a number from 0 to 65535. When your computer connects to a host server computer, the connection is made between an assigned outgoing port on the client's computer and a specific incoming port on the host computer. For example, when you access http://www.google.com, you are actually accessing http://www.google.com:80, as this is the default port for web servers. Ports are numbered for consistency and programming with multiple different purposes. A port is associated with an IP address of the host and this is important as it allows applications or processes running on a single computer to be uniquely identifiable such that they can share a single physical connection.

Struct:

addrinfo

The *addrinfo* structure contains the following fields:

```
struct addrinfo {
        int             ai_flags;
        int             ai_family;
        int             ai_socktype;
        int             ai_protocol;
        socklen_t       ai_addrlen;
        struct sockaddr*    ai_addr;
        char*           ai_canonname;
        struct addrinfo*    ai_next;
};
```

This *ai_flag* field specifies additional options (i.e. AI_PASSIVE, etc.) and multiple flags are OR-ed together. The *ai_family* field specifies the desired address family for the returned addresses (i.e. AF_INET, etc) and the *ai_socktype* field specifies the preferred socket type such as SOCK_STREAM or SOCK_DGRAM. The *ai_protocol* field specifies the protocol for the returned socket addresses and the *ai_addrlen* contains the length of the socket address. The *ai_addr* field contains the socket address. The *ai_canonname* field points to the official name of the host. The *ai_next* field points to the linked list of addrinfo structures, one for each network address that matches node and service.

Calls:

getaddrinfo()

```
int getaddrinfo(const char* node, const char* service, const struct addrinfo* hints, struct addrinfo** res)
```

This function returns one or more *addrinfo* structures and the first two arguments are the above mentioned node and service, which identify an internet host and a service. The hints argument points to an addrinfo structure that contains criteria for selecting the socket address structures.

If hints is not NULL it points to a structure whose *ai_family*, *ai_socktype*, and *ai_protocol* limit the set of socket addresses that can be returned by the function. The *res* argument points to the linked list that will be returned by the function.

socket()

int socket (int namespace, int style, int protocol)

This function creates a socket and specifies the communication style (per the *style* argument), which will be one of the defined socket styles. The return value from *socket* is the file descriptor for the new socket, or -1 in case of error. The potential *errno* error conditions are the following:
EPROTONOSUPPORT: The *protocol* or *style* is not supported by the *namespace* specified.
EMFILE: The process already has too many file descriptors open.
ENFILE: The system already has too many file descriptors open.
EACCES: The process does not have the privilege to create a socket of the specified *style* or *protocol*.
ENOBUFS: The system ran out of internal buffer space.
The file descriptor returned by the *socket* function supports both read and write operations, but they do not support file positioning operations.

connect()

int connect(int socket, struct sockaddr* addr, socklen_t length)

This function initiates a connection from the socket with file descriptor *socket* to the socket whose address is specified by the *addr* and *length* arguments. The function will wait until the server responds to the request before it returns. Upon success, *connect()* will return 0, and if an error occurs, it returns -1. The following *errno* conditions are defined for this function.
EBADF: The socket is not a valid descriptor.
ENOTSOCK: The file descriptor is not a socket.
EADDRNOTAVAIL: The specified address is not available on the remote machine.
EAFNOSUPPORT: The namespace of the *addr* is not supported by this socket.
EISCONN: The socket is already connected
ETIMEDOUT: The attempt to establish the connection timed out.
ECONNREFUSED: The server has refused to establish the connection:
ENETUNREACH: The network of the given *addr* isn't reachable from this host.
EINPROGRESS: The socket is non-blocking and the connection couldn't be established immediately.
EALREADY: The socket is non-blocking and already has a pending connection in progress.

listen()

int listen (int socket, int n)

The listen function enables the socket (*socket)* to accept connections, which makes it a server socket. The argument *n* specifies the length of the queue for pending connections. The function returns 0 on success and -1 on failure. The following *errno* conditions are defined for this function.
EBADF: The argument *socket* is not a valid file descriptor.

ENOTSOCK: The argument *socket* is not a socket.
EOPNOTSUPP: The socket *socket* does not support this operation.

accept()

int accept (int socket, struct sockaddr* addr, socklen_t* length_ptr)

This function is used to accept a connection request on the server socket *socket.* The function waits if there are no connections pending. The *addr* and *length_ptr* arguments are used to return information about the name of the client socket that initiated the connection. Accepting a connection creates a new socket which becomes connected to the *socket* argument. The normal return value is the file descriptor for the new socket. If an error occurs, the function returns -1. The following *errno* conditions are defined for this function:
EBADF: The argument *socket* is not a valid file descriptor.
ENOTSOCK: The descriptor is not a socket.
EOPNOTSUPP: The descriptor does not support this operation.
EWOULDBLOCK: The *socket* has nonblocking mode set.

send()

ssize_t send (int socket, const void* buffer, size_t size, int flags)

This function is similar to write, but with the additional *flags* argument. This function returnsthe number of bytes transmitted or -1 on failure. However, it should be noted that a successful return value indicates that the message has been *sent* without error, but not necessarily that is has been *received* without error. The following *errno* error conditions are defined for this function:
EBADF: The *socket* argument is not a valid file descriptor.
EINTR: The operation was interrupted by a signal before any data was sent.
ENOTSOCK: The descriptor is not a socket.
EMSGSIZE: The socket type requires that the message be sent atomically, but the message is too large for this to be possible.
EWOULDBLOCK: Nonblocking mode has been set on the socket.
ENOBUFFS: There is not enough internal buffer space available.
ENOTCONN: The socket was never connected.
EPIPE: The socket connection is now broken.

recv()

ssize_t recv (int socket, void* buffer, size_t size, int flags)

This function is similar to read, but with the additional argument of *flags*. This function returns the number of bytes received, or -1 on failure. The following *errno* conditions are defined for this function:
EBADF: The *socket* argument is not a valid file descriptor.
ENOTSOCK: The descriptor *socket* is not a socket.
EWOULDBLOCK: Nonblocking mode has been set on the socket, and the ready operation would block.
EINTR: The operation was interrupted by a signal before any data was ready.

ENOTCONN: The socket was never connected.

close()

int close (int filedescriptor)

This function closes the socket represented by the given file descriptor. The normal return value is 0 and -1 is returned in case of failure. The following *errno* conditions are defined for this function:

EBADF: Not a valid file descriptor.

EINTR: The close call was interrupted by a signal.