opiframe
Knowledge equals power

Opiframe Oy

Inter Process Communication

IPC - Pipes

- Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process.

- In Linux, a pipe is implemented using two file data structures which both point at the same temporary VFS inode which itself points at a physical page within memory

IPC - Pipes

- Each file data structure contains pointers to different file operation routine vectors.
- One is for writing to the pipe
- and other for is reading from the pipe.

IPC - Pipes

- As the writing process writes to the pipe, bytes are copied into the shared data page.
- When the reading process reads from the pipe, bytes are copied from the shared data page.
- Linux must synchronize access to the pipe.

IPC - Pipes

- When the writer wants to write to the pipe it uses the standard write library functions.

- If there is enough room to write all of the bytes into the pipe and, so long as the pipe is not locked by its reader, Linux locks it for the writer and copies the bytes to be written from the process's address space into the shared data page.

IPC - Pipes

- If the pipe is locked by the reader or if there is not enough room for the data then the current process is made to sleep on the pipe inode's wait queue.

- When the data has been written, the pipe's VFS inode is unlocked and any waiting readers sleeping on the inode's wait queue will themselves be woken up.

IPC - Pipes

opiframe
Knowledge equals power

- Pipe is created by system call **pipe()**:
- **int pipe(int** *filedes***[2]);**

- **pipe**() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by *filedes*. *filedes[0]* is for reading, *filedes[1]* is for writing.

- On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

IPC - Pipes

- Linux also supports named pipes, also known as FIFOs because pipes operate on a First In, First Out principle.
- Unlike pipes, FIFOs are not temporary objects, they are entities in the file system.
- Processes are free to use a FIFO so long as they have appropriate access rights to it.

IPC - Pipes

opiframe
Knowledge equals power

- A pipe is created in one go whereas a FIFO already exists and is opened and closed by its users.

- Linux must handle readers opening the FIFO before writers open it as well as readers reading before any writers have written to it.

IPC - Pipes

opiframe
Knowledge equals power

- FIFOs are created with **mkfifo()** system call:
- **int mkfifo(const char *pathname, mode_t mode);**

- **mkfifo**() makes a FIFO special file with name *pathname*. *mode* specifies the FIFO's permissions. It is modified by the process's **umask** in the usual way.
- Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

IPC - Sockets

- Sockets provide point-to-point two-way communication between two processes.
- Sockets are very versatile and are basic component of interprocess and intersystem communication.
- A socket is an endpoint of communication to which a name can be bound.

IPC - Sockets

- Sockets exist in communication domains.
- A socket domain is an abstraction that provides an addressing structure and a set of protocols.
- Sockets connect only with sockets in the same domain.

IPC - Sockets

- Most interprocess communication uses the client server model.
- One of the two processes, the client, connects to the other process, the server, typically to make a request for information.
- A good analogy is a person who makes a phone call to another person.

IPC - Sockets

- The system calls for establishing a connection are somewhat different for the client and the server.
- Both involve the basic construction of a socket.
- The two processes each establish their own socket.

IPC - Sockets

- There are two widely used address domains:
- The Unix domain, in which two processes which share a common file system communicate.
- The Internet domain, in which two processes running on any two hosts on the Internet communicate.

IPC - Sockets

- The address of a socket in the Unix domain is a character string which is basically an entry in the file system. In short, a file.

- The address of a socket in the Internet domain consists of the Internet address of the host machine (IP) and the port.

IPC - Sockets

opiframe
Knowledge equals power

- There are two popular socket types used:
- **SOCK_STREAM**, a streaming socket: It provides bidirectional, sequenced and unduplicated flow of data without boundaries. Except for the bidirectionality of data flow this is analogious to pipes.
- **SOCK_DGRAM**, a datagram socket: it supports bidirectional flow of data in the datagram model network level protocol.

IPC - Sockets

- Sockets are created with socket() system call:
- **int socket(int** *address_family***, int** *socket_type***, int** *protocol***);**

- Argument *address_family* states the connection domain in which the socket exists (There are 23 connection domains):

- **AF_INET** (or **AF_INET6**) for internet sockets

- **AF_UNIX** for unix (or local) sockets.

IPC - Sockets

opiframe
Knowledge equals power

- Argument *socket_type* states the type of the socket:
- **SOCK_STREAM**, for streaming sockets
- **SOCK_DGRAM**, for datagram sockets
- **SOCK_RAW**, for raw access
- Argument *protocol* states the protocol chosen for address family and socket type. Give value 0 to let the kernel decide.

IPC - Sockets

- A socket is created without a name.
- To be used, it must be given a name so that processes can reference it and messages can be received on it.
- Names can be bound explicitly by a process or implicitly given by the system during certain calls.

IPC - Sockets

- Communication processes are bound by an association.
- In Unix domain an association (a name) is composed of local and foreign pathnames.
- Foreign pathname here means name created by another process.

IPC - Sockets

- Unix domain socket names, like file pathnames, may be either absolute like **/dev/socket** or relative like **../socket**.

- Because these names are used to allow processes to establish connections, relative pathnames can pose difficulties and should be used with care.

IPC - Sockets

- In Internet domain names consist of the IP address of the machine and a port number.
- Incoming port numbers (like server listening to a given port) are explicitly bound by a system call.
- Outgoing ports are implicitly bound by the system.

IPC - Sockets

- Sockets can be explicitly bound to name with system call bind():
- **int bind(int** *sockfd***, const struct sockaddr ****my_addr***, socklen_t** *addrlen***);**

- The argument sockfd is the socket currently being bound.

- struct sockaddr is an unifying socket address struct for all types of socket addresses. In reality you'll be using specific structs for each address type.

IPC - Sockets

- **struct sockaddr_un** is for Unix domain sockets and is defined by:
- **struct sockaddr_un {**
  short sun_family; /*AF_UNIX*/
  char sun_PATH[108]; /*path name */
  };
- **struct sockaddr_in** is for Internet domain sockets:
- **struct sockaddr_in {**
  short *sin_family*; // e.g. AF_INET, AF_INET6
  unsigned short *sin_port*; // e.g. htons(3490)
  struct in_addr *sin_addr*; // see struct in_addr, next page
  char *sin_zero*[8]; // zero this if you want to

};

IPC - Sockets

- **struct in_addr {**
  unsigned long *s_addr*; // address is copied here
};
- The sockaddr structs are different because the naming conventions and the communication domains are so different.

- Thus we need to have the **socklen_t** argument in the **bind()** call. This is used to reserve just enough memory for each type of address.

IPC - Sockets

- Note that binding to name explicitly is usually done in the server side, especially when dealing with Internet domain.

- Client side, the active process, is usually implicitly bound when connection is established.

IPC - Sockets

- The difference between active (client) and passive (server) process comes apparent at this stage.

- Servers usually bind explicitly where as clients do not. First let’s go through the system calls for server side and then client side.

IPC - Sockets

opiframe
Knowledge equals power

- After the **bind()** server side is ready to listen to the socket. This is done with two system calls.
- **int listen(int** *sockfd***, int** *backlog***);**
- To accept connections, a socket is first created with **socket()**, a willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**(), and then the connections are accepted with **accept()**. The **listen**() call applies only to sockets of type **SOCK_STREAM.**
- The *backlog* parameter defines the maximum length the queue of pending connections may grow to.

IPC - Sockets

- **int accept(int** *sockfd***, struct sockaddr ***addr***, socklen_t ***addrlen***);**
- The **accept**() system call is used with connection-based socket types (**SOCK_STREAM).**
- It extracts the first connection request on the queue of pending connections, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket *sockfd* is unaffected by this call.
- Note that **accept()** returns a new file descriptor which is then used for the actual connection!

IPC - Sockets

- The client side needs to compile the address for the server.
- Again the analogy with phone call. Caller needs to know the phone number to call.
- Compiling the address is easy for Unix sockets, just a filename, but can be difficult for Internet sockets.

IPC - Sockets

- For Internet sockets there are several system calls available.
- First system calls are for getting the address
- And others are for translating the port number into network byte order which the system can understand. This is also used by the server side when creating the address for listening.

IPC – Sockets, creating address

- For address creation:
- #include <netdb.h>

struct hostent ***gethostbyname**(const char **name*);

struct hostent ***gethostbyaddr**(const char **addr*, int *len*, int *type*);

These functions map back and forth between host names and IP addresses. For instance, if you have "www.example.com", you can use **gethostbyname()** to get its IP address

IPC – Sockets, creating address

- struct **hostent** is defined as
- struct **hostent**
- {

  char *__h_name__; /* official name of host */

  char **__h_aliases__; /* alias list */

  int __h_addrtype__; /* host address type */

  int __h_length__; /* length of address */

  char **__h_addr_list__; /* list of addresses from name server */

  #define **h_addr h_addr_list[0]** /* address, for backward compatiblity */
- }

IPC – Sockets, creating address

opiframe
Knowledge equals power

- These two are going the way of dodo and new system calls are replacing these:

- int **getaddrinfo**(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);
- int **getnameinfo**(const struct sockaddr *sa, socklen_t salen, char *host, size_t hostlen, char *serv, size_t servlen, int flags);

- **getaddrinfo()** converts human-readable text strings representing hostnames or IP addresses into a dynamically allocated linked list of struct addrinfo structures.

IPC – Sockets, creating address

- After you have gained the address needed with any of those system calls you need to translate the port number into network byte order. A host of functions exists for that.

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

- Those starting with ”h” are host to network byte and those with ”n” are network byte to host.

IPC – Sockets

- Now that you have a working address (hopefully) you can finally **connect()** the client to the server:

- **int connect(int** *sockfd*, **const struct sockaddr ***serv_addr*, **socklen_t** *addrlen*);

- The **connect**() system call connects the socket referred to by the file descriptor *sockfd* to the address specified by *serv_addr*. The *addrlen* argument specifies the size of *serv_addr*.

- Note that **connect()** does not return a new file descriptor!

IPC – Sockets

- Finally there are used just like any ordinary file descriptor.
- You can **read()** from a socket. Again if read() is in blocking mode your program will block until input is available.
- And you can **write()** to a socket. If write buffer is full and you are in blocking mode your program will block.

System V IPC

- Linux supports three types of interprocess communication mechanisms that first appeared in Unix System V (1983).
- These are message queues, semaphores and shared memory.
- These System V IPC mechanisms all share common authentication methods.

System V IPC

- Processes may access these resources only by passing a unique reference identifier to the kernel via system calls.

- Access to these System V IPC objects is checked using access permissions, much like accesses to files are checked.

System V IPC

- The access rights to the System V IPC object is set by the creator of the object via system calls.

- The object's reference identifier is used by each mechanism as an index into a table of resources.

System V IPC

- All Linux data structures representing System V IPC objects in the system include an **ipc_perm** structure which contains:

- The owner and creator process's user and group identifiers.

- The access mode for this object (owner, group and other) and the IPC object's key.

System V IPC

- The key is used as a way of locating the System V IPC object's reference identifier.
- Two sets of keys are supported:
- Public: Any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object.
- Private: Only the creator process and its children can access the object.

System V IPC

- System V IPC consists of message queues, semaphores and shared memory areas.
- All three have very similar syntax despite their different purposes.
- Generally, do following:

System V IPC

- Use **ftok()** to generate a proper IPC key
- Get the IPC specific identifier associated with the IPC key using **msgget()**, **semget()** or **shmget()** for message queues, semaphores or shared memory areas respectively.
- Modify the IPC attributes with **msgctl()**, **semctl()** or **shmctl()**.
- Use the IPC instance for your purpose.
- Remove the IPC instance using **msgctl()**, **semctl()** or **shmctl()** and IPC_RMID flag.

System V IPC

- To generate a key:
- **key_t ftok(const char **pathname*, int *proj_id*);**

- The **ftok**() function uses the identity of the file named by the given *pathname* (which must refer to an existing, accessible file) and the least significant 8 bits of *proj_id* (which must be non-zero) to generate a **key_t** type System V IPC key, suitable for use with ***msgget()***, ***semget()***, or ***shmget()***.

- On success the generated **key_t** value is returned. On failure -1 is returned, with *errno* indicating the error.

System V IPC

- Once a set of processes have independently come up with their IPC keys, they must then get a specific identifier associated with the particular IPC instance using one of the get system calls.

- The get calls all require the IPC key and a set of flags and perhaps additional information.

System V IPC

- Because UNIX is a multiuser system, the flags include file permissions in the familiar octal format (0666 for everybody read/write).

- If the **IPC_CREAT** flag is also set, the IPC instance will be created if it does not exist.

System V IPC – Message Queues

opiframe
Knowledge equals power

- Message queues allow one or more processes to write messages, which will be read by one or more reading processes.
- Linux maintains a list of message queues, the msgque vector.
- When message queues are created a new msqid_ds data structure is allocated from system memory and inserted into the vector.

System V IPC – Message Queues

opiframe
Knowledge equals power

- Each **msqid_ds** data structure contains an **ipc_perm** data structure and pointers to the messages entered onto this queue.

- Linux keeps queue modification times such as the last time that this queue was written to and so on.

System V IPC – Message Queues

- The **msqid_ds** also contains two wait queues:
- one for the writers to the message queue
- one for the readers of the message queue.

System V IPC – Message Queues

opiframe
Knowledge equals power

- Each time a process attempts to write a message to the write queue its effective user and group identifiers are compared with the mode in this queue's **ipc_perm** data structure.

- Linux restricts the number and length of messages that can be written.

System V IPC – Message Queues

- There may be no room for the message.
- In this case the process will be added to this message queue's write wait queue.
- It will be woken up when one or more messages have been read from this message queue.

System V IPC – Message Queues

opiframe
Knowledge equals power

- Reading from the queue is a similar process. Again, the processes access rights to the write queue are checked.

- A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types.

System V IPC – Message Queues

opiframe
Knowledge equals power

- If no messages match this criteria the reading process will be added to the message queue's read wait queue

- When a new message is written to the queue this process will be woken up and run again.

System V IPC – Message Queues

opiframe
Knowledge equals power

- Creating (or getting) an instance of message queue:
- **int msgget(key_t** *key***, int** *msgflg***);**

- The **msgget**() system call returns the message queue identifier associated with the value of the *key* argument. A new message queue is created if *key* has the value **IPC_PRIVATE** or *key* isn't **IPC_PRIVATE**, no message queue with the given key *key* exists, and **IPC_CREAT** is specified in *msgflg*.

System V IPC – Message Queues

opiframe
Knowledge equals power

- Changing the setup of a specific queue:
- **int msgctl(int** *msqid***, int** *cmd***, struct msqid_ds ***buf***);**

- **msgctl**() performs the control operation specified by *cmd* on the message queue with identifier *msqid*.

- **IPC_STAT**, Copy information from the kernel data structure associated with *msqid* into the *msqid_ds* structure pointed to by *buf*. The caller must have read permission on the message queue.

System V IPC – Message Queues

opiframe
Knowledge equals power

- **IPC_SET,** Write the values of some members of the *msqid_ds* structure pointed to by *buf* to the kernel data structure associated with this message queue.

- **IPC_RMID,** Immediately remove the message queue, awakening all waiting reader and writer processes.

- **IPC_INFO,** returns information about system-wide message queue limits and parameters in the structure pointed to by *buf.*

System V IPC – Message Queues

opiframe
Knowledge equals power

- Writing to the queue:
- **int msgsnd(int** *msqid***, const void ****msgp***, size_t** *msgsz***, int** *msgflg***);**

- The second argument must be of type:
- struct **msgbuf** {

  long **mtype**; /* message type, must be > 0 */
  char **mtext[X]**;   /* message data */
  };

- The third argument must be sizeof(msgbuf.**mtext**);

System V IPC – Message Queues

opiframe
Knowledge equals power

- Reading from the queue:
- **ssize_t msgrcv(int** *msqid*, **void ***msgp*, **size_t** *msgsz*, **long** *msgtyp*, **int** *msgflg*);**

- The system call **msgrcv**() removes a message from the queue specified by *msqid* and places it in the buffer pointed to *msgp*. Again the argument must be a **struct** with first member being a **long**.

- The argument *msgsz* specifies the maximum size in bytes for the member *mtext* of the structure pointed to by the *msgp* argument.

System V IPC – Message Queues

- The fourth parameter, *msgtyp*, is the type parameter, and it allows you to be selective about which messages you get:
- If the type is 0, the first message in the queue is returned
- If the type is a positive integer, the first message in the queue with the same type is returned.
- If the type is a negative integer, the first message in the queue with the lowest value that is less than or equal to the absolute value of the specified type is returned.

System V IPC - Semaphores

opiframe
Knowledge equals power

- Semaphores are used to protect critical pieces of code from simultaneous access that would cause errors.
- Semaphores usually have two states, locked or unlocked.
- Only the process which locked the semaphore can unlock it.

System V IPC - Semaphores

- System V IPC semaphores are slightly more complex than that.
- They can have any number of values and as such can display more complex behavior than basic versions.

System V IPC - Semaphores

opiframe
Knowledge equals power

- System V IPC semaphore objects each describe a semaphore array and Linux uses the semid_ds data structure to represent this.

- There are **sem_n** semaphores in each semaphore array.

System V IPC - Semaphores

opiframe
Knowledge equals power

- Creating (or accessing) a semaphore array:
- **int semget(key_t** *key***, int** *nsems***, int** *semflg***);**

- The **semget**() system call returns the semaphore set identifier associated with the argument *key*. A new set of *nsems* semaphores is created.

- If successful, the return value will be the semaphore set identifier (a nonnegative integer), otherwise -1 is returned, with *errno* indicating the error.

System V IPC - Semaphores

opiframe
Knowledge equals power

- Controlling the semaphore array:
- **int semctl(int** *semid***, int** *semnum***, int** *cmd***, ...);**

- **semctl**() performs the control operation specified by *cmd* on the semaphore set identified by *semid*, or on the *semnum*-th semaphore of that set

- This function has three or four arguments, depending on *cmd*. When there are four, the fourth has the type *union semun*.

- Argument *cmd* has same options as with **msgctl()**.

System V IPC - Semaphores

opiframe
Knowledge equals power

- Finally operating on the semaphore in a set:
- **int semop(int** *semid*, **struct sembuf ***sops*, **unsigned** *nsops*);

- **semop**() performs operations on selected semaphores in the set indicated by *semid*. Each of the *nsops* elements in the array pointed to by *sops* specifies an operation to be performed on a single semaphore. The elements of this structure are of type *struct sembuf* containing the following members:
- unsigned short **sem_num**; /* semaphore number, starts at 0 */
- short **sem_op**; /* semaphore operation */
- short **sem_flg**; /* operation flags */

System V IPC - Semaphores

opiframe
Knowledge equals power

- If any of the semaphore operations would fail Linux may suspend the process but only if the operation flags have not requested that the system call is non-blocking.

- If the process is to be suspended then Linux must save the state of the semaphore operations to be performed and put the current process onto a wait queue.
- If all of the semaphore operations would have succeeded and the current process does not need to be suspended, Linux goes ahead and applies the operations to the appropriate members of the semaphore array.

System V IPC - Semaphores

opiframe
Knowledge equals power

- The **sem_op** member of the struct **sembuf** has following possible values:

- 0, **sem_num** is tested to see if it is 0. If **sem_num** is 0, the next test runs. If **sem_num** is not 0, either the operation blocks until the semaphore becomes 0.

- If **sem_op** is a positive integer, the value of **sem_op** is added to the value of the semaphore.

System V IPC - Semaphores

opiframe
Knowledge equals power

- The **sem_flags** argument can have two values even both at the same time:

- **IPC_NOWAIT**, do not wait on any of the operations specified by **semop()**.

- **SEM_UNDO**, undo all locks incase of process terminating when it has acquired a lock. This prevents semaphores remaining locked after the termination of a process.

System V IPC – Shared Memory

opiframe
Knowledge equals power

- Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces.

- The pages of the virtual memory is referenced by page table entries in each of the sharing processes' page tables.

System V IPC – Shared Memory

opiframe
Knowledge equals power

- Shared memory does not have to be at the same address in all of the processes' virtual memory.
- Once the memory is being shared, there are no checks on how the processes are using it.
- They must rely on other mechanisms, for example semaphores, to synchronize access to the memory.

System V IPC – Shared Memory

opiframe
Knowledge equals power

- Each newly created shared memory area is represented by a **shmid_ds** data structure.

- The **shmid_ds** data structure decribes how big the area of shared memory is, how many processes are using it and information about how that shared memory is mapped into their address spaces.

System V IPC – Shared Memory

- Each process that wishes to share the memory must attach to that virtual memory via a system call:
- **int shmget(key_t** *key***, size_t** *size***, int** *shmflg***);**

- **shmget**() returns the identifier of the shared memory segment associated with the value of the argument *key*. A new shared memory segment, with size equal to the value of *size* rounded up to a multiple of **PAGE_SIZE**, is created.

System V IPC – Shared Memory

- When processes no longer wish to share the virtual memory, they detach from it.
- So long as other processes are still using the memory the detach only affects the current process.
- When the last process sharing the memory detaches from it, the pages of the shared memory current in physical memory are freed

System V IPC – Shared Memory

- Accessing the actual memory area when we have the identified:
- **void *shmat(int** *shmid*, **const void ***shmaddr*, **int** *shmflg*);**
- **shmat**() attaches the shared memory segment identified by *shmid* to the address space of the calling process. The attaching address is specified by *shmaddr.*

- Specify *shmaddr* as **NULL** if you wish for the system to decide on the address.

- The pointer returned by this call is the pointer to the shared memory area. Use it as you would any other pointer.

System V IPC – Shared Memory

opiframe
Knowledge equals power

- For controlling the shared memory area:
- **int shmctl(int** *shmid***, int** *cmd***, struct shmid_ds ***buf***);**

- **shmctl**() performs the control operation specified by *cmd* on the shared memory segment whose identifier is given in *shmid*.

- Again the command flags are similar to **msgctl()**.