Aricent

Linux IPC Mechanisms

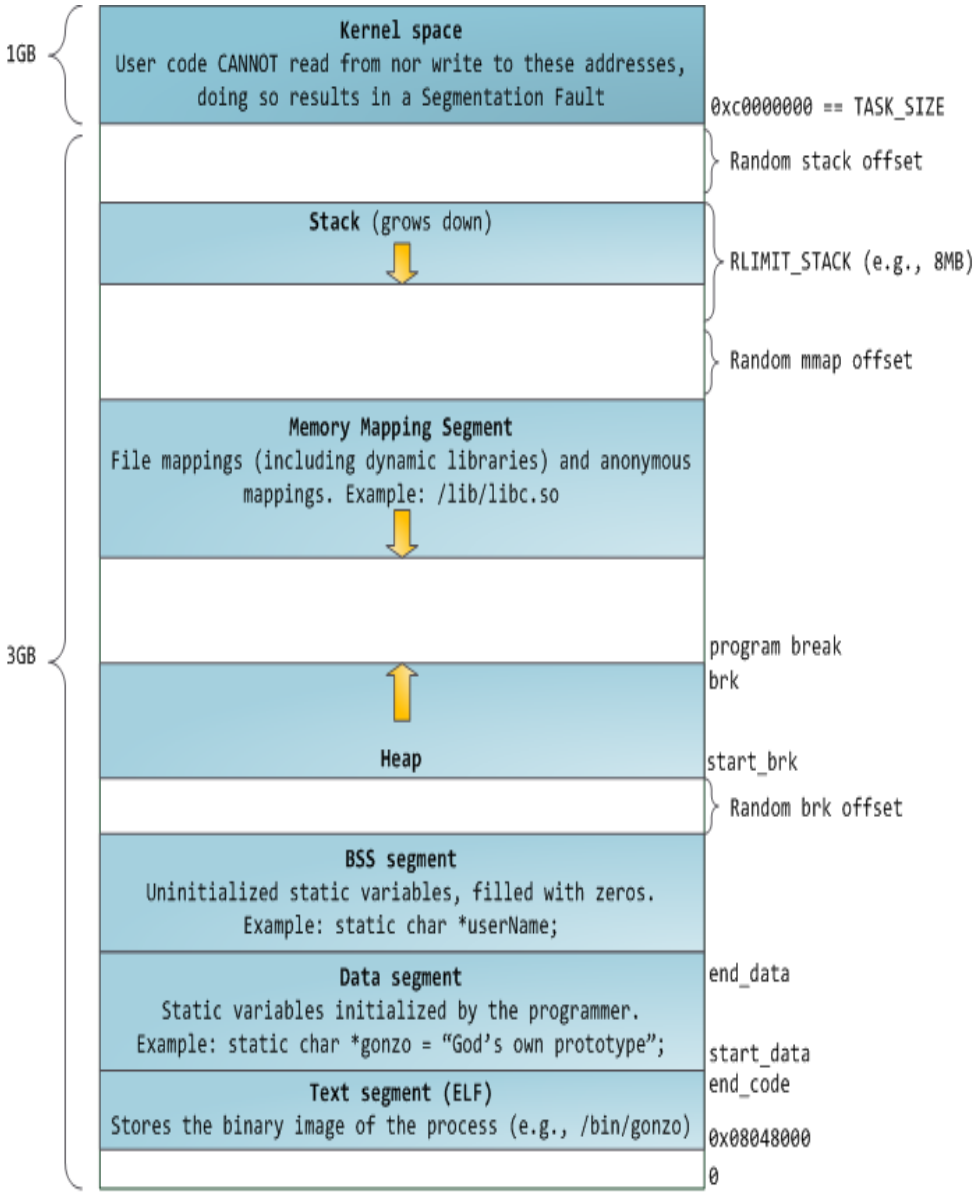
Agenda

- Introduction
- Basics of Process Creation
- IPC Mechanisms
- Pipes and FIFOs
- Introduction to SystemV and POSIX
- SystemV
  - Common features
  - Message Queues
  - Shared Memory
- POSIX
  - Common features
  - Memory mapped I/O
  - Message Queues
  - Shared Memory
  - Semaphore
- Command-line IPC control

Address Space of A Process (Linux)

- The address space of a process consists of all linear addresses that the process is allowed to use/access.
- On 32-bit system, the linear addresses i.e. virtual addresses ranges from 0B to 4GB for a single process
- These virtual addresses are mapped to physical RAM pages, and hence each process has its own virtual addresses in 4GB flat space, but operating system maps different physical pages for every process corresponding to these virtual addresses. That is the reason there is no address space sharing or data sharing between two process by default
- The linear address space of a process comprise of Code, Data, Stack and Heap segements laid out on 3GB virtual memory space out of 4GB space as 1GB is reserved for Kernel
- Page tables are used to maintain the mapping of linear or virtual address (which might seems common across multiple process) from physical address for every process by kernel in process control block
- Every segment of a process has different permissions so even writing at some segments of a process is not allowed for example code segment is read only hence any attempt to write on it will result in general protection fault
- Process can ask for dynamic mapping of memory regions in its address space using system calls such as ***mmap***(), or by dynamically loading shared object using ***dlopen***() system calls etc.
- A process can execute the kernel code but only using the system call interface

Address Space of A Process (Linux)



fork primer

- A task or process is an independent flow of instructions associated with a memory space and a private context.
- Immediately after the fork, the parent and the child are identical for both the code, the data and the context.
- They can modify everything in their environment independently because their contexts are separate.
- Since the code segment is identical, they will also run the same instructions just after the fork.
- Looking at the return value of the forking function is the only way to distinguish who's who.
  - The parent is returned the Process ID of the child, whereas the child always gets the value "0".

fork primer

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     fork();
     pid = getpid();
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
          write(1, buf, strlen(buf));
     }
}
```

- Both processes start their execution right after the system call **fork()**.
- Since both processes have identical but separate address spaces, those variables initialized **before** the **fork()** call have the same values in both address spaces.
- Since every process has its own address space, any modifications will be independent of the others. This can be understood by the fact that for child process, new physical pages will be copied from parent pages, along with a new page table will be allocated by the OS.
- In other words, if the parent changes the value of its variable in exclusive pages of parent process, the modification will only affect the variable in the parent process's address space as child has its own physical pages different than parent
- Other address spaces created by **fork()** calls will not be affected even though they have identical variable names.

fork primer

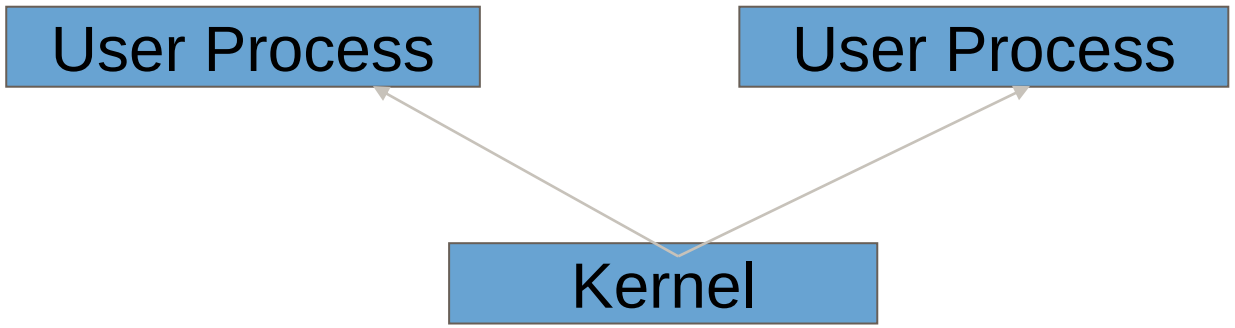
- Being processes, children can terminate and return exit values.
  - The wait() function lets the parent fetch this value.
  - It blocks the calling process until the first child terminates; then, it returns the pid of the child and stores its exit status.
  - When a child terminates, the OS signals its parent with a SIGCHLD.
  - When the parent produced several children and one particular is expected to terminate, the waitpid() function is to be used.
  - Waitpid can also be instructed not to be blocking, thereby it is used frequently in place of wait() too.
  - The SIGCHLD signal can also be bound to a "purging" wait() function for skipping zombies accumulation.
  - If a process terminates before any of its children, the process with PID 1 (init) receives the paternity of them.

signal primer

- A signal is a very primitive kind of message.
- Process A can send process B a signal. If B associated an handler to the given signal, that code will be executed when the signal is received, interrupting the normal execution.
- Processes send signals with the kill() function. A process may associate an handler function to a specific signal type with the signal() function.
- Two standard handlers have been provided with the standard C library: SIG_DFL causing process termination, and SIG_IGN causing the signal to get ignored.
- When an handler fires for a signal, it won't be interrupted by other handlers if more signals arrive.

What is Inter Process Communication (IPC)?

- As discussed earlier, the two process run in their own address space having mutually exclusive physical pages, they do not share any shared memory pages
- IPC is a mechanism which allows multiple process to communicate with each other
- There are different mechanisms to enable data sharing between processes
  - Via Linux OS which takes copy of the information from one process and copy it to other processes address space. As content pass via kernel, it will be a slower communication
    - **Message queue, Socket etc**
  - Allocate common physical pages on request from user process which can be mapped to linear or virtual address space of cooperating processes and any write to these pages will be immediately visible to other process. As the pages are common and does not involve kernel, it is a fast paced mechanism
    - **Shared memory**



IPC Mechanisms

- Linux IPC has various mechanisms:
  - Pipe
  - FIFO
  - Message Queues
  - Shared Memory
  - Semaphore
  - Sockets

Pipes

A pipe is used for one-way communication of a stream of bytes.

A pipe has no permanent or external name ,a program can access it through its two descriptors.

Pipes can only be used between processes that have a parent process in common i.e. related processes.

+ It is worth noting that any two arbitrary processes cannot communicate using a pipe. The pipe has to be set by the parent process and the children can just use it

Pipes are temporary that they disappear when no process has them open.

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

- Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
- **Pipes can be used only between processes that have a common ancestor**. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

We'll see that FIFOs get around the second limitation, and that **UNIX domain sockets get around both limitations**.

Pipes

The *pipe* function creates a pair of file descriptors pointing to a pipe inode and places them into an array pointed to by filedes.

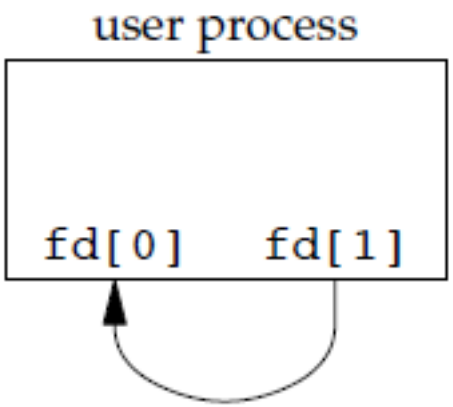
filesdes[0] is for reading and filedes[1] is for reading.

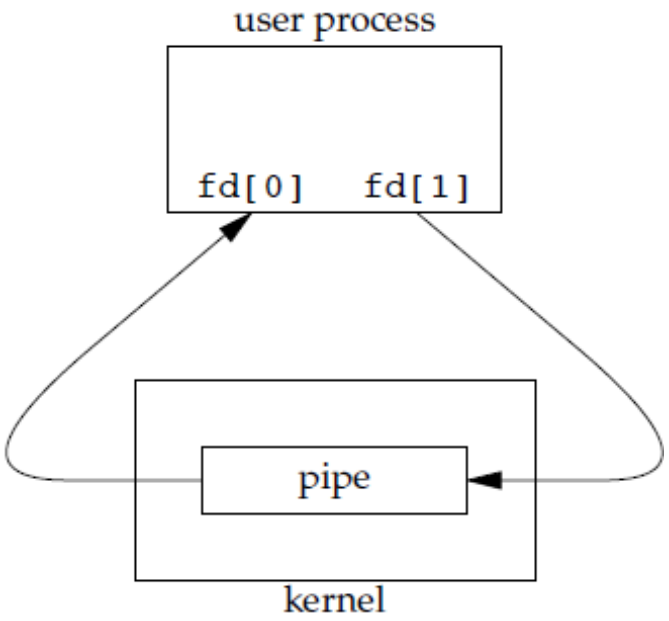
#include< unistd.h >

int pfd[2];

int pipe(pfd);

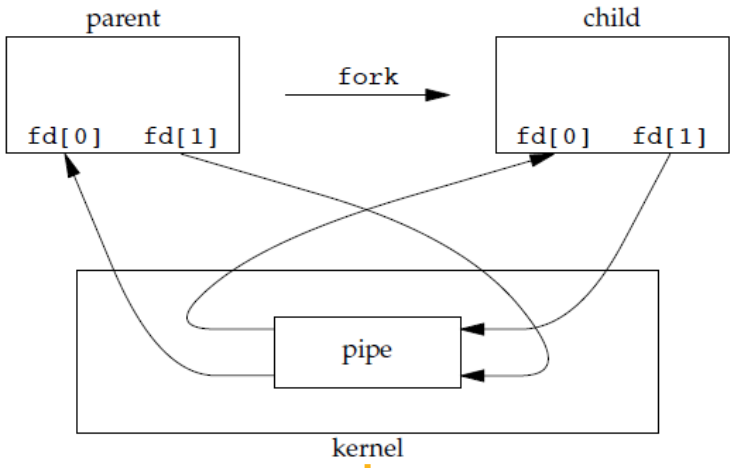
Two file descriptors are returned through the pfd argument: pfd[0] is open for reading and pfd[1] is open for writing. The output of *fd[1]* is the input for *fd[0]*.



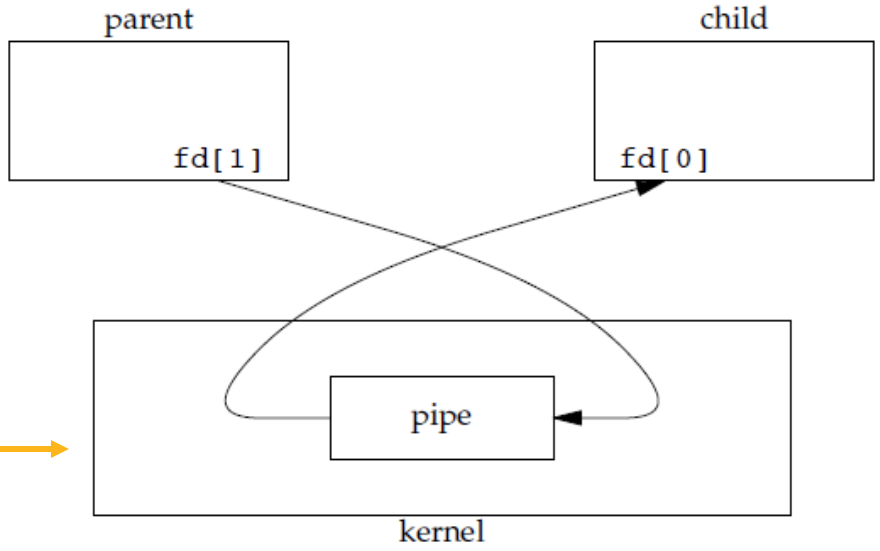


Pipes

- A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa.
- Figure below shows this scenario.

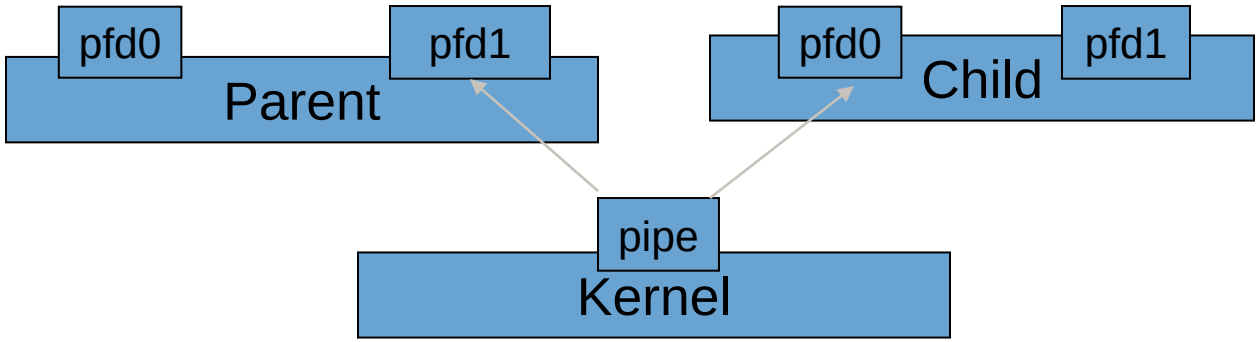


What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Next figure shows the resulting arrangement of descriptors.



Pipes

- A pipe opened before the fork becomes shared between the two processes.



- In a process, pipe() system call is called first which returns two file descriptors, and then fork() is called where file descriptors are copied to both process, parent and child process.

Pipes

- read() will return 0 (end of file) when the write end of the pipe is closed. if write end of the is still open and there is no data, read() will sleep until input become available.
- if a read() tries to get more data than is currently in pipe, read() will only contain the number of bytes actually read. Subsequent reads will sleep until more data is available.
- The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.
- If read end of pipe is closes, a write() will fail and process will be sent SIGPIPE signal. Default SIGPIPE handler terminates
- Either process can write into the pipe, and either can read from it. Which process will get what is not known.

- For predictable behaviour, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.

- Suppose the parent process is a writer process and child process is a reader process. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end.

- When the processes have ceased communication, the parent closes its write end. This means that the child gets eof on its next read, and it can close its read end.

Pipes

- #include <stdio.h>
  #define READ 0
  /* The index of the “read” end of the pipe */
  #define WRITE 1

  /* The index of the “write” end of the pipe */
  char * phrase = “Stuff this in your pipe and smoke it”;

  main () {
    int fd[2], bytesRead;
    char message [100]; /* Parent process’s message buffer */

    pipe (fd); /*Create an unnamed pipe*/

- if (fork ( ) == 0) { /* Child Writer */
    close (fd[READ]); /* Close unused end*/
    write (fd[WRITE], phrase, strlen (phrase) +1); /* include NULL*/
    close (fd[WRITE]); /* Close used end*/
  }
  else { /* Parent Reader */
    close (fd[WRITE]); /* Close unused end*/ bytesRead = read (fd[READ], message, 100);
    printf (“Read %d bytes: %s\n”, bytesRead, message);
    close (fd[READ]); /* Close used end */
  }
}

Pipes

- The parent process creates a pipe. Now, we must remember that a process's system data comprising of open file descriptors and other items like the current directory, the accumulated CPU time, etc. is inherited by the child process and is preserved across the exec system calls.
- So, when a parent makes a pipe and forks a child and, then, *exec*s the child program, the child gets the pipe file descriptors.
- Actually, the parent duplicates the pipe file descriptor to be used by the child from the standard input or output file descriptor and closes the pipe file descriptors.
- The child reads from the standard input or writes to the standard output as per its program, but actually, courtesy parent, it is reading from or writing to the pipe.

When one end of a pipe is closed, two rules apply.
1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe. When we get to FIFOs in the next section, we'll see that often there are multiple writers for a single FIFO.)

2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.

Pipes

- FILE ***popen(const** char *command, const char *mode)
  - The popen() function shall execute the command specified by the string command. **It shall create a pipe between the calling program and the executed command**, and shall return a pointer to a stream that can be used to either read from or write to the pipe.

```
#include <stdio.h>
int main(void) {
    FILE *in;
    extern FILE *popen();
    char buff[512];
    if(!(in = popen("ls -sail", "r"))){
        exit(1);
    }
    while(fgets(buff, sizeof(buff), in)!=NULL){
        printf("%s", buff);
    }
    pclose(in);
}
```

Pipes -Assignment

- Time for an assignment

  - Implement the following using pipes. The program receives two filenames as command line parameters. The parent process writes the contents of first file to the pipe and the child process reads from the pipe and writes into the second file. The contents which are copied into second file should be reversed. (For example: " I am funny" as "funny am I"). After receiving the contents from parent process ,the child process sends a signal "SIGCHLD" to the parent process .The parent process after receiving the signal closes the pipe and exits.

FIFOs

- A FIFO is a pipe that has a name in the file system.

- FIFOs or named pipes are special files that persist even after all processes have closed them.
- They exist until they are removed with rm or unlink()
- A FIFO has a name and permissions like an ordinary file and appears in the directory listing given by *"ls".*

- FIFO could be used for related process as well as unrelated process, since creation of a FIFO results in named pipe.

  - #include <sys/ types.h>
  - #include <sys/stat.h>
  - int mkfifo(const char *pathname, mode_t mode);

FIFOs

- FIFO 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.
- When a process tries to write to a FIFO that is not opened for read on the other side, the process is sent a SIGPIPE signal.

FIFO

- Writer

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";
    /* create the FIFO (named pipe) */
    mkfifo(myfifo, 0666);
    /* write "Hi" to the FIFO */
    fd = open(myfifo, O_WRONLY);
    write(fd, "Hi", sizeof("Hi"));
    close(fd);
    /* remove the FIFO */
    unlink(myfifo);

    return 0;
}
```

Reader

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#define MAX_BUF 1024
int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";
    char buf[MAX_BUF] = {0x00};

    /* open, read, and display the message from the FIFO */
    fd = open(myfifo, O_RDONLY);
    read(fd, buf, MAX_BUF);
    printf("Received: %s\n", buf);
    close(fd);

    return 0;
}
```

SYSTEM V

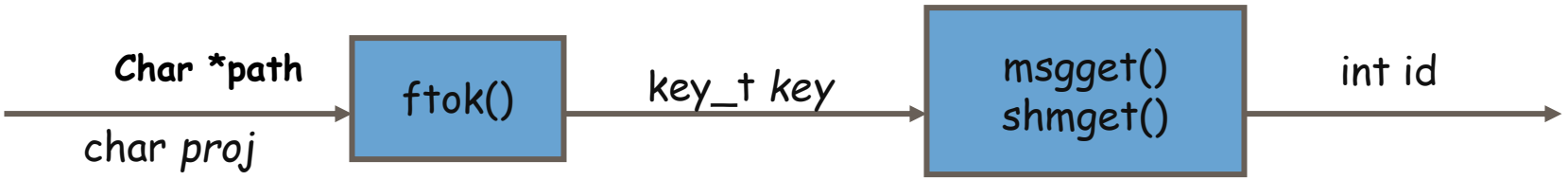
System V and POSIX

- System V
  - UNIX ***System V*** is one of the versions of UNIX operating system. It was originally developed by AT&T. It added support for inter-process communication using messages, semaphores, and shared memory.
- POSIX
  - ***Portable Operating System Interface for UNIX*** is a family of standards developed through the IEEE and ISO/IEC. It describes the operating system service interface, based on UNIX existing practice and experience, and exists to permit application programs to be written that are source code portable across multiple diverse operating systems.

Common features(Keys)

Kernel maintains a structure of information for each of the IPC channels (Message Queues ,Semaphores and Shared Memory) currently open in the system.

Keys are used to identify an IPC object on a UNIX system.



Common features(Keys)

- A key is an integer value of type key_t;
- A key should not be used as int or long, since the length of a key is system dependent.
- There are three different ways of using keys, namely:
  - Do it yourself.
  - Use a function ftok.
  - Ask the system to provide a private key.

Common Features (Keys)

- Keys are global entities. If other processes know the key value ,they can access or attach to same IPC segments created by one.
- Do it yourself
  - Key_t key;
  - Key = 1234;
- Use ftok() function
  - key_t ftok(const char *path, int proj_id);
    - Key = ftok(“./x”, ‘b’);
    - Path is a pathname(e.g. “./x”)
    - Proj_id is an integer value(e.g. ‘b’)
- System provides a private key using a system defined macro “IPC_PRIVATE”.

Message Queue

- A message queue is a linked list of messages held in the kernel, which processes can access.
- A queue has a unique key, fixed size, ownership, and access modes.
- Message queues have kernel persistence.
- It must be explicitly deleted.
- A message consists of type (a positive long integer),length of the data portion of the message (can be zero) and data (length > 0).
- It provides bidirectional communication.
- /etc/sysctl.conf contains the OS hard limits on message queue size and numbers

kernel.msgmni=128 #Max # of msg queue identifiers

kernel.msgmnb=163840 #Size of message queue

kernel.msgmax=40960 #Max size of a message

Message Queue APIs

- msgget()
  - It returns the message queue identifier .
  - It provides access to a message queue.
- msgsnd
  - It is used to send a message to a message queue.
- msgrcv
  - It removes the message form the message queue.
- msgctl
  - It is used to deallocate or change permissions for the message queue

To Create a message queue

- int msgget (key_t key, int flag);
  - key –created through any of the 3 methods discussed previously.
  - msgflag
    - IPC_CREAT|0666

- For Example:

```
key_t key;
int msgqid;
key = ftok(“./x” ,’b’);
msgqid-=msgget( key, IPC_CREAT | 0666 );
if (-1 == msgqid)
 {
    perror("msgget");
    exit(1);
}
```

- $ ipcs -q
  - Provides information about the message queue IPC Object

The Message Structure - struct msgbuf

```
struct msgbuf {
        long mtype;     /* message type, must be > 0 */
        char mtext[1];  /* message data */
    };
```

- The mtext field is an array (or other structure) whose size is specified by msgsz, a non-negative integer value.
- Messages of zero length (i.e., no mtext field) are permitted.
- The mtype field must have a strictly positive integer value that can be used by the receiving process for message selection

Writing messages onto a queue

- int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
  - int msqid - id of message queue, as returned from the msgget() call.
  - struct msgbuf* msg - a pointer to a properly initializes message structure, such as the one we prepared in the previous section.
  - int msgsz - the size of the data part (mtext) of the message, in bytes.
  - int msgflg - flags specifying how to send the message. may be a logical "or" of the following:
    - To set no flags, use the value '0'.
- For example:

```
int rc = msgsnd (queue_id, msg, MSGSZ , 0);
if (rc == -1) {
  perror ("msgsnd");
 exit(1);
 }
```

Reading A Message From The Queue

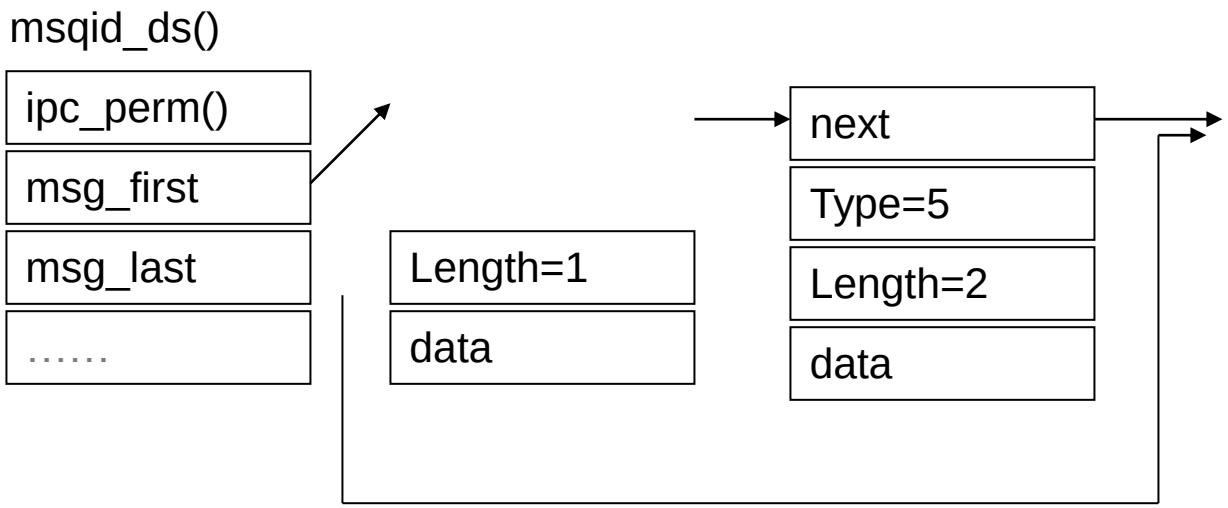
- ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);
  - int msgsz - size of largest message text we wish to receive. Must not be larger than the amount of space we allocated for the message text in 'msg'.
  - int msgtyp - Type of message we wish to read. may be one of:
    - 0 - The first message on the queue will be returned.
    - a positive integer - the first message on the queue whose type (mtype) equals this integer
    - a negative integer - the first message on the queue with the lowest type less than or equal to the absolute value of mtype will be read.
  - int msgflg - flags specifying how to send the message. may be a logical "or" of the following:
    - To set no flags, use the value '0'.

Reading A Message From The Queue

```
#include <sys/types.h> /* standard system data types.       */
#include <sys/ipc.h>   /* common system V IPC structures.   */
#include <sys/msg.h>   /* message-queue specific functions. */

/* prepare a message structure large enough to read our "hello world". */
struct msgbuf* recv_msg =
     (struct msgbuf*) malloc(sizeof(struct msgbuf)+strlen("hello world"));
/* use msgrcv() to read the message. We agree to get any type, and thus */

/* use '0' in the message type parameter, and use no flags (0).         */
int rc = msgrcv (queue_id, recv_msg, strlen("hello world")+1, 0, 0);
if (rc == -1) {
    perror ("msgrcv");
    exit(1);
}
```



- *Assume that there are 3 messages in a queue with lengths of 1,2,and 3 bytes. The type of messages is 1,5 and 79 respectively.*
- *If type = 5 remove second message.*
- *If type = -100 then three calls to msgrcv() removes messages of type 1,5,79.*

Message queue control operations

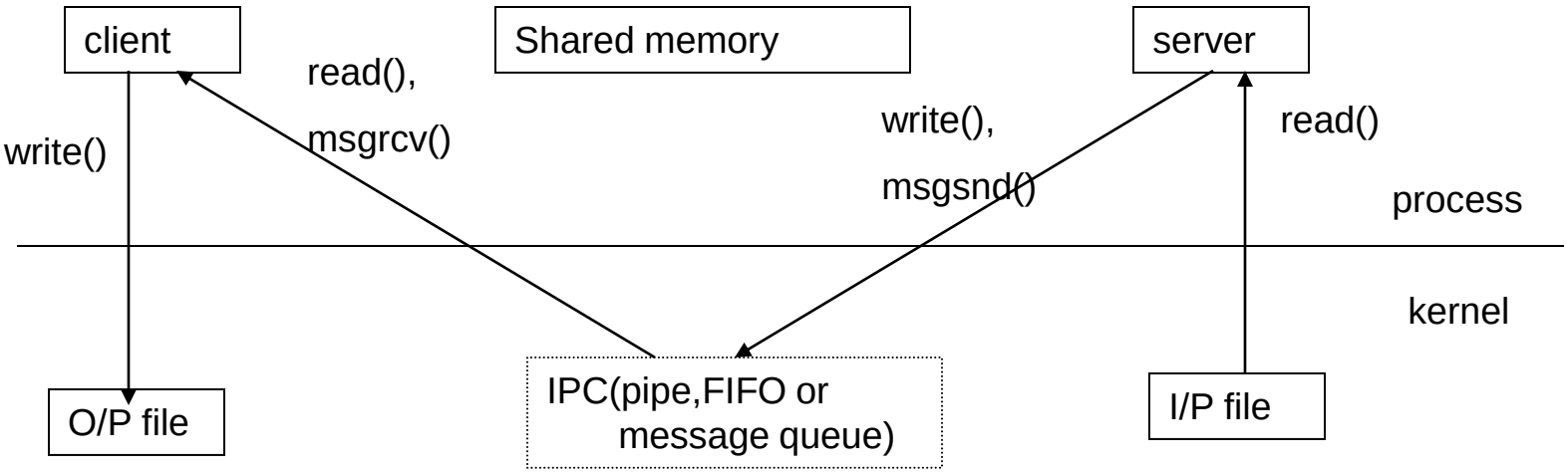
- int msgctl(int msqid, int cmd, struct msqid_ds *buf);
  - This function performs the control operation specified by cmd on the message queue with identifier msqid.
  - One possible value of cmd is IPC_RMID
    - Immediately remove the message queue and its associated data structure, awakening all waiting reader and writer processes.

- For example:
  - msgctl (msgqid, IPC_RMID, NULL);

Shared Memory

- Pipe, FIFO and message queues (?) are sequential in nature.
- What do we do in order to allow processes to share data is random access manner?
  - Shared Memory
- A shared memory segment is a piece of memory that can be allocated and attached to an address space.
- It is the fastest form of IPC mechanism because the data does not need to be copied via kernel from one process to the other
- It allows two or more processes to share a memory.
  - Virtual Address Space of a process
  - Any of the communicating process define via system call interface a given section in the memory as one that will be used simultaneously by several processes. This segment is then mapped to all the other processes in their virtual address which looks as if they have allocated a memory section using malloc() or calloc() call
  - Any process can refer to a specific location in this shared segment using the same offset across any of the communicating processes. One of the example is sharing the offset of the shared memory segment using message queue so that other process can use this offset to read from the right location in its own address space
- Race conditions can occur when several processes try to access the shared memory segment attached at the same time. This problem can be solved by semaphores

Copying file data using shared memory



- In the above figure as shown ,kernel movement for shared memory is less as compared to when pipe,FIFO or message queue is used.
  - *The server reads from the I/P file into the shared memory object. The second argument to the read, the address of the data buffer points into the shared memory object.*
  - *The client writes the data from the shared memory object to the output file*

Shared Memory APIs

- shmget
  - To allocate a shared memory segment
- shmat
  - To attach a shared memory to an address space.
- shmdt
  - To detach a shared memory from an address space.
- shmctl
  - To deallocate a shared memory

Allocating A Shared Memory Segment

- int shmget(key_t key,size_t size,int shmflg);
- RETURNS: shared memory segment identifier on success
  - Key :key to identify a shared memory segment,
  - Size :Memory size
  - Shmflg
    - IPC_CREAT|0666

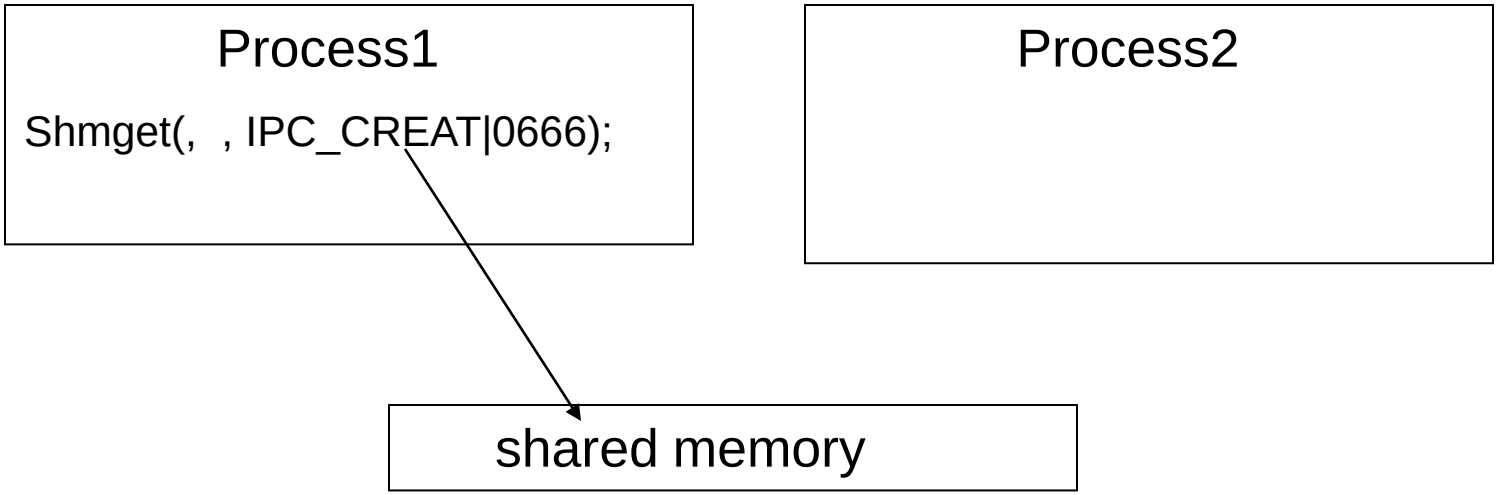
- Example:

```
struct data{int a;char c;};
  struct data *p;
    int     shmid;
    key_t key;
    key = ftok(“./x” ,’b’);

    shmid-=shmget( key, sizeof(struct data), IPC_CREAT | 0666 )
```

Allocating A Shared Memory Segment

- When asking for a shared memory segment,the process that creates uses IPC_CREAT|0666 and the process that accesses a created one uses 0666.
- After the execution of shmget(),shared memory is allocated but is not part of the address space.



Attaching A Shared Memory Segment

- shmat() is used to attach an existing shared memory into address space.
  - void *shmat (int shmid, const void *shmaddr, int shmflg);
  - If successful, It returns a void pointer to the memory.
  - Shmid is the id returned from shmget()
  - If the address argument is NULL then the kernel tries to find out an unmapped region. This is the recommended method
  - Example

```
ptr = (struct data *)shmat(shmid,NULL,0);
If ((int) ptr <0) {perror(“shmat”);exit(-1);}
ptr ->a=1;
ptr ->c=‘a’;
```

Attaching A Shared Memory Segment

- Attach the given shared memory segment, at some free position that will be allocated by the system. *

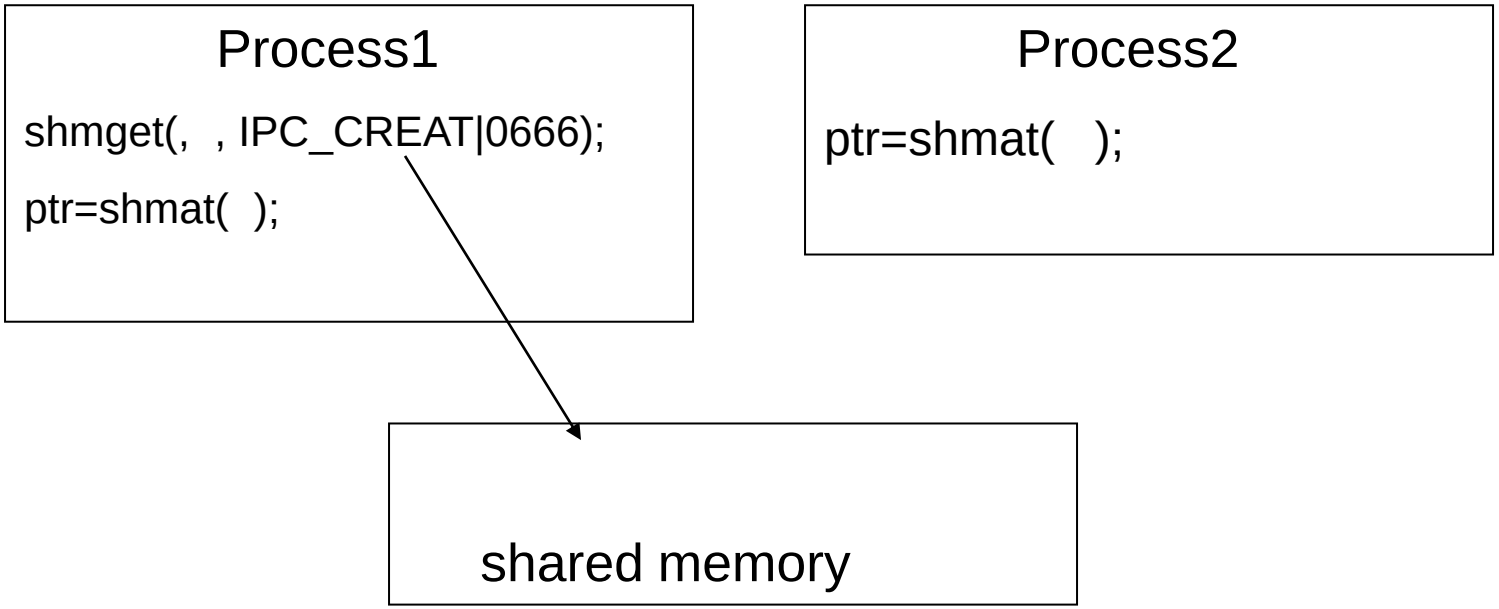
```
shm_addr = shmat(shm_id, NULL, 0);
if (!shm_addr) { /* operation failed. */
    perror("shmat: ");
    exit(1);
}
```

- Attach the same shared memory segment again, this time in read-only mode. Any write operation to this page using this address will cause a segmentation violation (SIGSEGV) signal.

```
shm_addr_ro = shmat(shm_id, NULL, SHM_RDONLY);
if (!shm_addr_ro) { /* operation failed. */
    perror("shmat: ");
    exit(1);
}
```

Attaching A Shared Memory Segment

- Process1 and Process 2 can access the shared memory using ptr.



Placing Data In Shared Memory

- Placing data in a shared memory segment is done by using the pointer returned by the shmat() system call

- Any kind of data may be placed in a shared segment, except for pointers
  - Pointers contain virtual addresses.
  - Since the same segment might be attached in a different virtual address in each process, a pointer referring to one memory area in one process might refer to a different memory area in another process

Detaching/Removing Shared Memory Segment

- To detach a shared memory
  - shmdt (ptr);
  - ptr is the pointer returned from shmat
- After a shared memory is detached ,it is still there. You can re-attach and reuse it again.
- To remove a shared memory
  - shmctl(shmid ,IPC_RMID, NULL);
- After a shared memory is removed, it will not be reused.

Detaching/Removing Shared Memory Segment

```
#include <sys/shm.h>
#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600 /* user read/write */
char array[ARRAY_SIZE]; /* uninitialized data = bss */
int
main(void) {
int shmid;
char *ptr, *shmptr;
printf("array[] from %p to %p\n", (void *)&array[0],
(void *)&array[ARRAY_SIZE]);
printf("stack around %p\n", (void *)&shmid);
if ((ptr = malloc(MALLOC_SIZE)) == NULL)   err_sys("malloc error");
printf("malloced from %p to %p\n", (void *)ptr, (void *)ptr+MALLOC_SIZE);
if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)    err_sys("shmget error");
if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)   err_sys("shmat error");
printf("shared memory attached from %p to %p\n", (void *)shmptr, (void *)shmptr+SHM_SIZE);
if (shmctl(shmid, IPC_RMID, 0) < 0)  err_sys("shmctl error");
exit(0);
}
```

Detaching/Removing Shared Memory Segment

Running this program on a 64-bit Intel-based Linux system gives us the following output:

$ **./a.out**

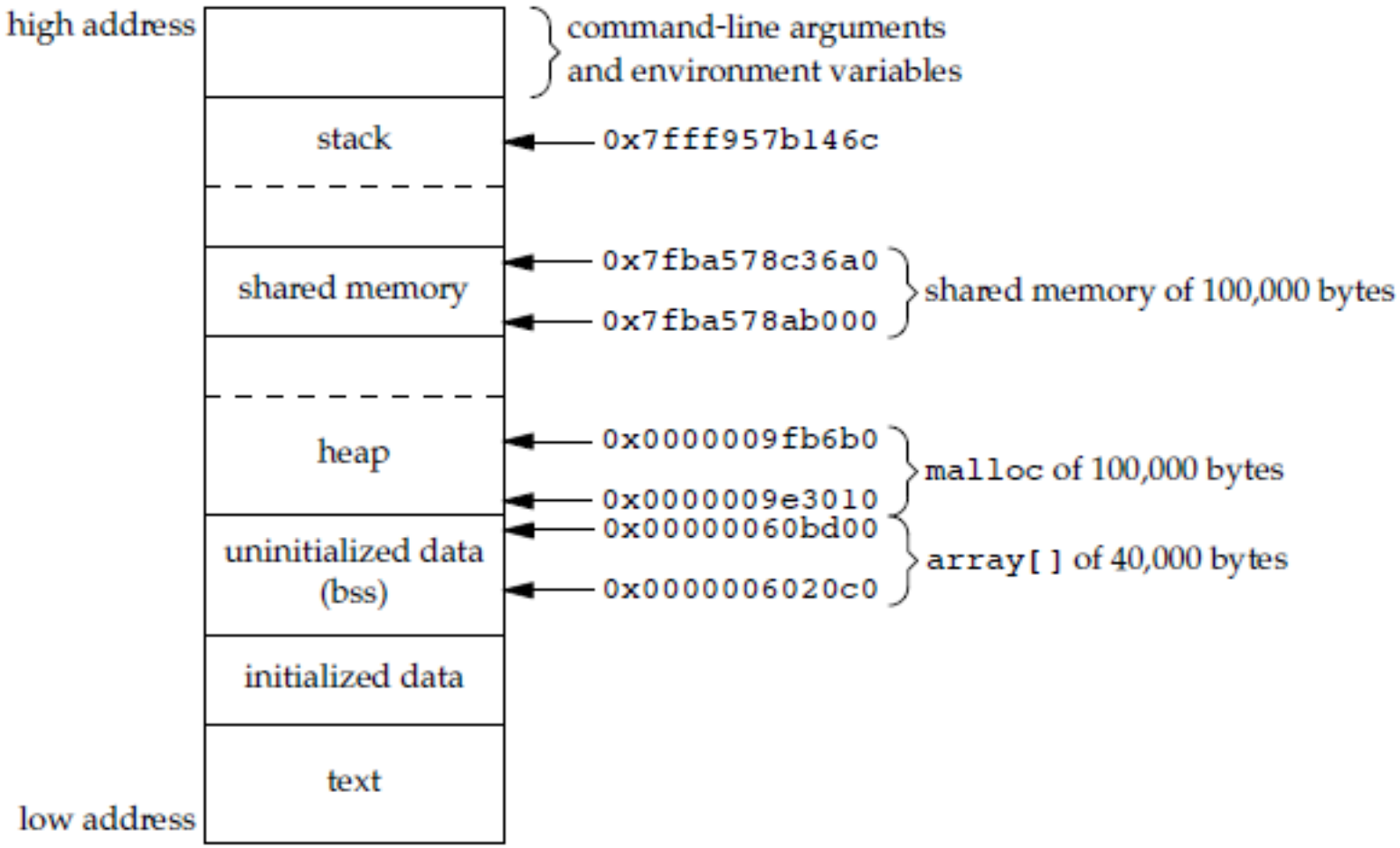
array[] from 0x6020c0 to 0x60bd00

stack around 0x7fff957b146c

malloced from 0x9e3010 to 0x9fb6b0

shared memory attached from 0x7fba578ab000 to 0x7fba578c36a0

Detaching/Removing Shared Memory Segment



POSIX

Mapped Memory

- Mapped Memory permits different processes to communicate via a shared file.
- It forms an association between a file and a process's memory.
- The process can read the file's contents with ordinary memory access.

Memory Mapped I/O

- Maps a file on disk into a buffer in Memory
  - A program maps the file into memory and scan it using memory reads.
  - Fetching bytes from the buffer results in the corresponding bytes of the file to be read .
  - Storing data in buffer results in bytes automatically written to the file.
  - Results in performing I/O without read / write .

Memory Mapped I/O

- Need to tell the kernel to map a given file to a region in memory .
  - void * mmap(void * addr , size_t len , int prot , int flag , int fd , off_t off);
    - addr :specifies the starting address where the file is mapped into the process address space.
      - (NULL) specifies to choose any available starting address
    - len :number of bytes to map
    - prot :protection on the mapped address range (Read / Write / Execute)
    - flag :various attributes of the mapped region (Shared/Private)
    - filedes :file descriptor opened to the file to be mapped.
    - off :offset from the beginning of the file from which to start the map

Message Queues

- Difference between System V and POSIX Message Queues:
  - POSIX Message Queues are reference counted.
  - Each system V message has an integer type and messages can be selected in a variety of ways using msgrcv() function.
  - Each POSIX message has an associated priority, and messages are always strictly queued in priority order.

Message Queues

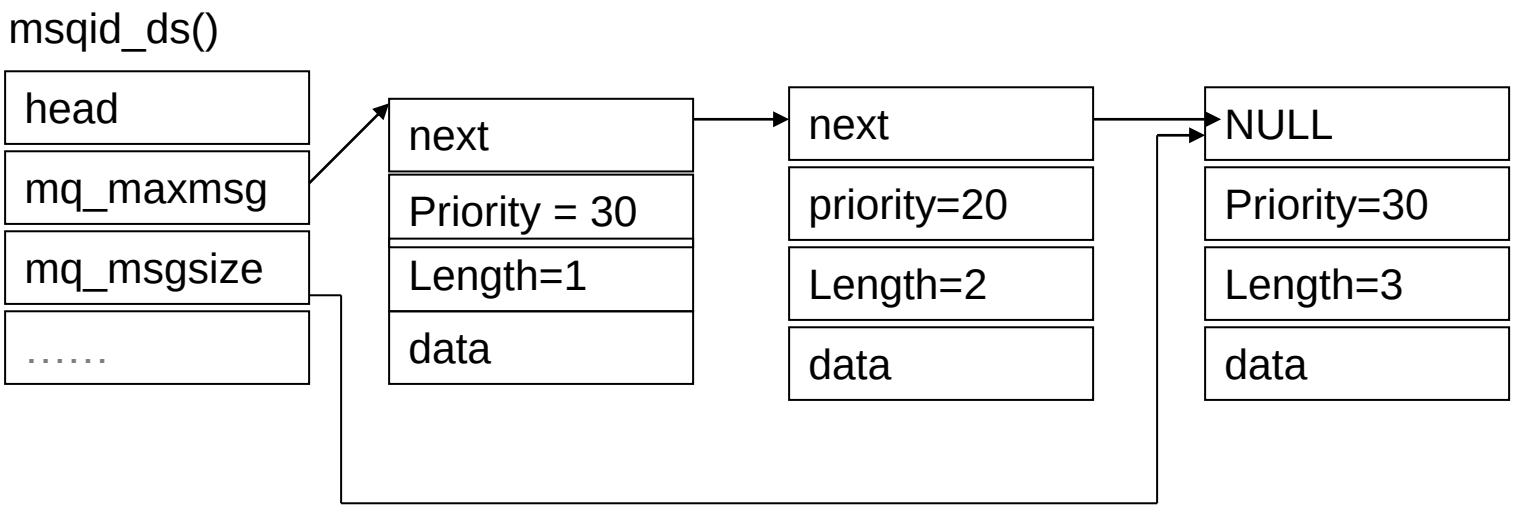
- A read on a POSIX message queue returns the oldest message of the highest priority.

- Every message in a queue has the following attributes:
  - An unsigned integer priority
  - The length of the data portion of the message
  - The data itself

Message queue APIs

- mq_open()
  - It creates a new message queue or opens an existing queue, returning a message queue descriptor for use in later calls.
- mq_send()
  - It writes a message to a queue.
- mq_receive()
  - It reads a message from a queue.
- mq_close()
  - It closes a message queue that the process previously opened.
- mq_unlink()
  - function removes a message queue name and marks the queues for deletion when all processes have closed it.

Message queue structure in kernel



- *The head of the linked list contains the two attributes of the queue*
  - *»mq_maxmsg : the maximum number of messages allowed on the queue.*
  - *»mq_msgsize : the maximum size of a message.*

Structure of Message Queues

- Each message Queue has four attributes which is in the following structure:
- *struct mq_attr*

```
{
 long mq_flags;       /* message queue flags */
 long mq_maxmsg; /* max no. of msgs allowed on queue */
 long mq_msgsize ; /* max size of a msg(in bytes) */
 long mq_curmsgs;  /*number of messages currently in queue */
}
```

Opening a Message Queue

- mqd_t mq_open(const char *name, int flags, ... [mode_t mode, struct mq_attr *mq_attr])
    - Opens the queue referenced by name for access,
  - Flags
    - O_RDONLY, O_WRONLY, O_RDWR ,O_CREAT, O_EXCL Mode
    - S_IRWXU, S_IRWXG etc
  - attr
    - It is an mq_attr structure that specifies attributes for the new message queue.

- For Example:

```
mqd_t mqd;
Struct mq_attr attr;
struct mq_attr attr = {mq_maxmsg = 20, .mq_msgsize = 256 };
mqd = mq_open ("q1", O_CREAT | O_EXCL | O_RDWR, 0600, &attr);
    if(-1==mqd){perror(“mq_open”);exit(-1);}
```

Message Queue Attributes

- int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
  - returns an mq_attr structure containing information about the message queue description and the message queue associated with the descriptor mqdes.
- int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr);
  - It set attributes associated with the open message queue.

- For Example:
  - To modify the non-blocking attribute of a message queue.

    mq_getattr(mq, &attr);

    attr.mq_flags |= O_NONBLOCK;

    mq_setattr(mq, &attr, NULL);
  - It obtains the number of messages in a queue.

    mq_getattr(mq, &attr);

    nmsg = attr.mq_curmsgs;

Message Queue Operations

- int mq_close(mdq_t mqdes)
  - Closes the queue described by mqdes.
  - Returns 0 for ok, otherwise -1.

- int mq_unlink(const char *name)
  - like unlink(2), but with the posix object referenced by name.
  - Returns 0, or -1 on error.

- For Example:

```
mq_close(mqd);
mq_unlink(mqd);
```

Sending/Recieving

- int mq_send(mqd_t mqdes, const char *msgbuf, size_t len, unsigned int prio)
  - Sends len bytes from msgbuf to the queue mqdes, associating a prio priority.
  - Return 0 on succes, -1 otherwise.
- ssize_t mq_receive(mqd_t mqdes, char *buf, size_t len, unsigned *prio)
  - Takes the oldest message with the highest priority from mqdes into buf.
  - prio, if not NULL, is filled with the priority of the given message.
  - Returns the length of the message received, or -1 on error.

Shared Memory

- There are two ways to share memory between unrelated processes:
  - Memory-mapped files
    - A file is opened by open() function and the resulting descriptor is mapped into the address space of the process by mmap().
  - Shared Memory objects
    - A file is opened by shm_open() function that is then mapped into the address space of the process by mmap().

Shared Memory APIs

- mmap()
  - Map a file or shared memory object into the address space.
- shm_open()
  - Create or gain access to a shared memory object.
- shm_unlink()
  - Destroy a shared memory object when no references to it remain open.

Opening a Shared Memory Object

- int shm_open(const char *name, int flags, mode_t mode)
  - Open (or create) a shared memory object with the given POSIX name.
  - Flags
    - O_RDONLY, O_WRONLY, O_RDWR
    - O_CREAT
  - Mode is only used when the object is to be created, and specifies its access permission .
- mmap() function maps a shared memory object into the process address space.
- For example:

```
shmid=shmopen(“shared”,O_CREAT|O_RDWR,S_IRWXU);
if (-1 == shm_fd)
 { perror(“shm_open”);exit(-1);}
attach = mmap(NULL,size,PROT_READ|PROT_WRITE,MAP_SHARED,shmid,(off_t)0);
If(-1==attach)
{ perror(“mmap”);exit(-1);}
```

Truncating a file to a specified length

- int truncate(const char *path, off_t length);
- int ftruncate(int fd, off_t length);

  - The truncate and ftruncate functions cause the regular file named by path or referenced by fd to be truncated to a size of precisely length bytes.
  - If the file previously was larger than this size, the extra data is lost.
  - If the file previously was shorter, it is extended, and the extended part reads as zero bytes.

Removing a Shared Memory Object

- int shm_unlink(const char *name)
  - Removes a shared memory object specified by name.
  - If some process is still using the shared memory segment associated to name, the segment is not removed until all references to it have been closed

Semaphore

- A semaphore is used to provide synchronization between various processes.
- It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic (all or nothing)
  operation. (https://en.wikipedia.org/wiki/Compare-and-swap)
- E.g. cmpxchg instruction on intel processors,
- There are 3 operations that a process can perform on a semaphore:
  - Create a semaphore
  - Wait for a semaphore
    - This tests the value of the semaphore ,waits(blocks) if the value

      is less than or equal to 0 and then decrements the semaphore value once it is greater than 0.
  - Post to a semaphore
    - This increments the value of the semaphore .If some processes are blocked ,waiting for this semaphore's value to be greater than 0, one of those processes can be unblocked.

Unnamed Semaphore Operations

- int sem_init(sem_t *sem, int pshared, unsigned int value);
  - The pshared argument must be 1 for different processes to share semaphore.
- int sem_wait(sem_t * sem);
  - It tests the value of the specified semaphore and if the value is greater than 0,the value is decremented and the function returns immediately.
  - If the value is 0 the calling process is put to sleep until the semaphore value is greater than 0 at which time it will be decremented and the function then returns.
- int sem_post(sem_t * sem);
  - It increments the value of the semaphore by 1 and wakes up any process that are waiting for the semaphore value to become positive.
- int sem_destroy(sem_t * sem);
  - To remove the semaphore.

Named Semaphore Operations

- sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
  - oflag can be set to O_CREAT or O_CREAT|O_EXCL
- For Example:

```
// the absolute path to the semaphore
const char sem_name[] = "/tmp/sem";

// 0644 is the permission of the semaphore
sem_t *sem = sem_open(sem_name, O_CREAT, 0644, 1);
```

Close/Remove a semaphore

- int sem_close(sem_t *sem);
  - Disassociates the named semaphore pointed by sem from the process (only used for semaphores created by sem_open()).
- int sem_unlink(const char *name);
  - Removes the named name semaphore. This is used after all the processes have closed the semaphore by calling sem_close().
- For Example

```
const char sem_name[] = "/tmp/sem";
sem_t *sem = sem_open(sem_name, O_CREAT, 0644, 1);
...
sem_close(sem);
sem_unlink(sem);
```

Command-line IPC control

- ipcs [-qms]
  - Provides information about the IPC Objects for which the calling process has read access.

- ipcrm [-Q key | -q id | -M key | -m id | -S key | -s id]
  - Removes specific IPC object and associated data structures from the system. The caller must be the superuser, or the creator/ owner of the object.
  - A shared memory object is removed only after all currently attached processes have detached from it.

References

- *Advanced Programming in the UNIX Environment by Stevens and Rago*
- *UNIX Network Programming (Interprocess Communication) Volume 2 by Stevens*
- *http://publib.boulder.ibm.com/*
- *UNIX Systems Programming by Kay A. Robbins & Steven Robbins*
- *Beej's Guide to Unix IPC*

Disclaimer

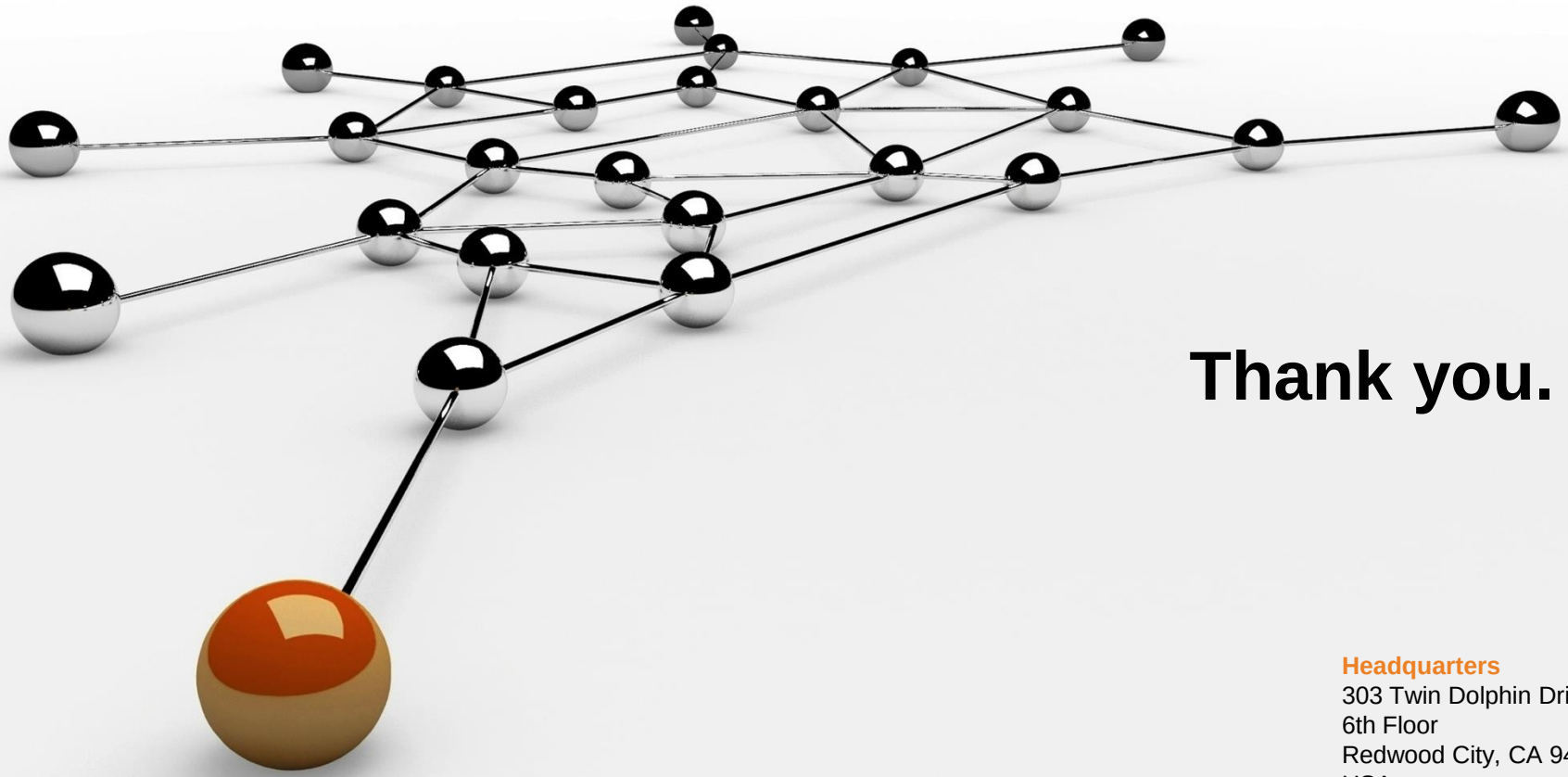
- *Aricent makes no representations or warranties with respect to contents of these slides and the same are being provided "as is". The content/materials in the slides are of a general nature and are not intended to address the specific circumstances of any particular individual or entity. The material may provide links to internet sites (for the convenience of users) over which Aricent has no control and for which Aricent assumes no responsibility for the availability or content of these external sites. While the attempt has been to acknowledge sources of materials wherever traceable to an individual or an institution; any materials not specifically acknowledged is purely unintentional*

Revision History

Revision no.	Date	Description of change	Author	Reviewed & Approved By
0.1	17-Aug-2010	Initial draft	Sibu Cyriac	Shiv Kumar
1.0	2-Feb-2011	Contents aligned	Soma	Shiv Kumar
1.1	26-July-2012	Changed to New Template	Tanmoy	Bhoopendra S
2.0	06-Apr-2016	Added few slides for fork and signal. Converted to New Template	Soma	GP
2.1	29-May-2017	Added figures for Pipe and address space	Deepak	

Aricent®

Engineering excellence. Sourced.



Thank you.

Headquarters
303 Twin Dolphin Drive
6th Floor
Redwood City, CA 94065
USA
Tel: +1 650 632 4310
Fax: +1 650 551 9901

www.aricent.com