

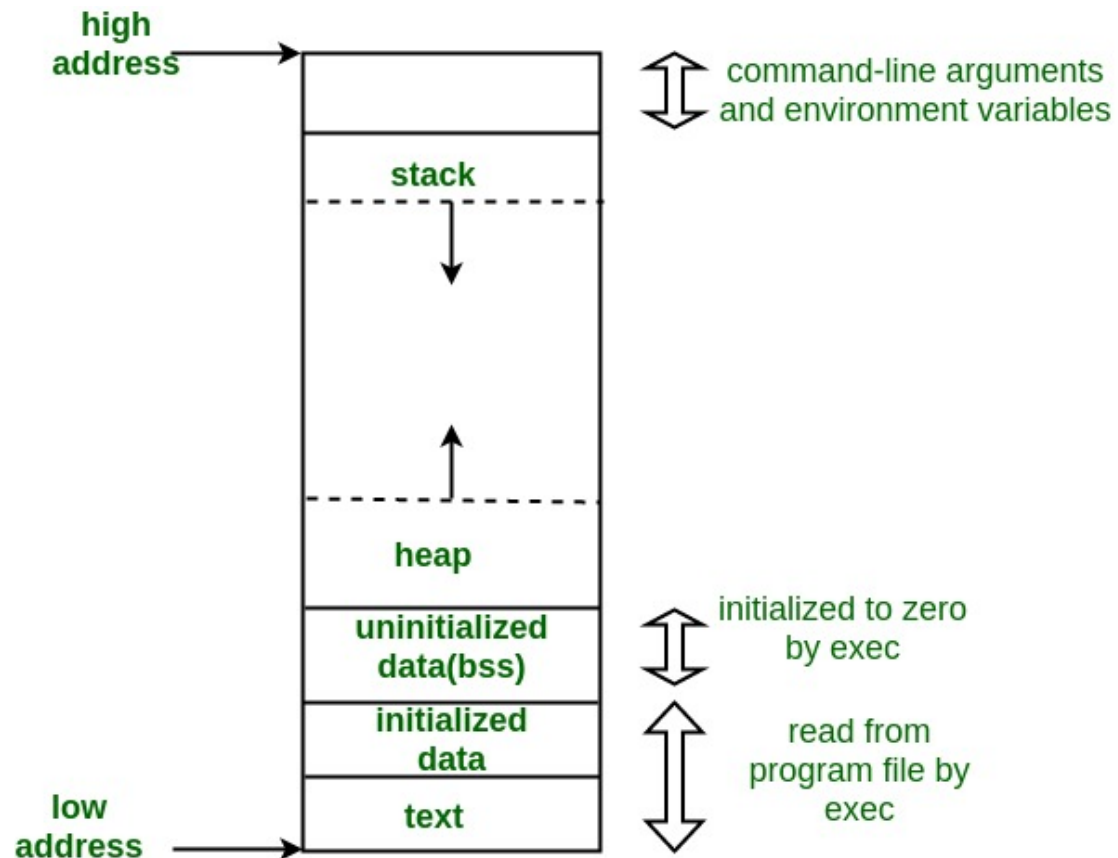


PROCESSES, SHARED MEMORY, PIPES ...

1

- 1) System calls: fork, wait, exit ... (zombies, orphans) & Communication
- 2) SystemV Shmem: shmget, shmat, shmdt, shmctl, ftok functions
- 3) POSIX Shmem: shm_open, ftruncate, mmap, munmap, close, shm_unlink
- 4) Unnamed/Anonymous Pipes (related processes), named Pipes (FIFOs), Msg Queues

PROCESS - MEMORY LAYOUT OF A C PROGRAM



FIELDS IN PROCESS CONTROL BLOCK

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

PROCESS ID & FUNCTIONS

- Every process has a unique (non negative) id between 0 and 65535.
 - IDs are reused when processes terminate
- Linux known processes (implementation-dependent), e.g. in pre-systemd distros
 - PID 0 : scheduler process
 - PID 1 : init process,
 - PID 2 : is the pagedaemon
- A process ID is represented by the `pid_t` data type

A process can call these functions (see page 228):

```
#include <unistd.h>
```

```
pid_t getpid(void);
```

Returns: PID of calling process

```
pid_t getppid(void);
```

Returns: parent PID of calling process

FORK

- A process (called parent) can create new processes by calling `fork()`. Each such process is called a child process, since its PCB is a copy of its parent (few exceptions).

```
#include <unistd.h>
pid_t fork(void);
```

- The `fork()` function is called once, but returns twice.
 - Once in child, returning 0
 - Second in parent, returning the PID of the new child
 - -1 on error
- Both the child and the parent continue executing their code with the instruction that follows the call to `fork()`

CHILD PROPERTIES DURING FORK (MAN FORK)

PCB contents of the parent are inherited by the child:

- Real user ID, real group ID, effective user ID, and effective group ID
- The set-user-ID and set-group-ID flags
- Supplementary group IDs
- Environment
- Controlling terminal
- Session ID
- Current working directory
- Root directory
- File mode creation mask
- Open file descriptors and flags
- Signal mask and dispositions
- Resource limits
- Memory mappings
- Attached shared memory segments

FORK

Differences between the parent and child are:

- The return values from fork are different.
- The PIDs are different.
- The two processes have different parent process IDs:
 - the parent PID of the child is the parent
 - the parent PID of the parent doesn't change
- The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are reset to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child
- The set of pending signals for the child is set to the empty set

PROCESS TERMINATION

There are eight ways for a process to terminate.

Normal termination occurs in five ways:

1. Return from main
2. Calling exit with cleanup (ISO C)
3. Calling _exit (ISO C) or _Exit (POSIX)
4. Return of the last thread from its start routine (see book Section 11.5)
5. Calling pthread_exit (Section 11.5) from the last thread

Abnormal termination occurs in three ways:

1. Calling abort (Section 10.17)
2. Receipt of a signal (Section 10.2)
3. Response of last thread to a cancellation request (Sections 11.5, 12.7)

EXIT FUNCTION

To terminate a process **normally** use one of the exit functions:

<pre>#include <stdlib.h> void exit(int status); void _Exit(int status); #include <unistd.h> void _exit(int status);</pre>	<p>int status : 0 or EXIT_SUCCESS indicates successful, while EXIT_FAILURE indicates unsuccessful termination.</p>
--	--

In **abnormal termination**, the parent of the process can obtain a termination status generated by the kernel using the `wait` or the `waitpid` function (see next). If the child terminated normally, the parent can obtain the exit status of the child.

WAIT/WAITPID FUNCTION

After a child is created using `fork()`, the parent must call `wait` to monitor child termination (to cleanup PCB, e.g. memory maps, close files, etc)

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: PID if OK, 0 (see later), or -1 on error

- The `wait` function can block the caller until a child process terminates, whereas `waitpid` has an option that prevents it from blocking.
 - **`statloc`** provides the exit code of the child
- The `waitpid` function doesn't wait for the child that terminates first; it can control which process it waits for via the `pid` argument (see next)

WAITPID PARAMETERS

- Can wait for specific processes
 - `pid > 0`, normal (specific process)
 - `pid == 0`, wait for any child whose process group ID is equal to that of the calling process ID
 - `pid < -1`, wait for any child whose process group ID is equal to `|pid|`
 - `pid == -1`, wait for any child process (as wait)
- Extended functions (for detailed resource utilization statistics)
- `pid_t wait3(int *status, int options, struct rusage *rusage);`
- `pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);`

WAIT FUNCTIONS

Function	<i>pid</i>	<i>options</i>	<i>rusage</i>	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
wait				•	•	•	•	•
waitid	•	•		XSI				•
waitpid	•	•		•	•	•	•	•
wait3		•	•		•	•	•	•
wait4	•	•	•		•	•	•	•

Figure 8.11 Arguments supported by wait functions on various systems

WAIT/WAITPID FUNCTION

From `statloc`, we can determine how a child exited using macros:

- `WIFEXITED(status)` is true if the child terminated normally. Then, use `WEXITSTATUS(status)` to obtain the exit status.
- `WIFSIGNALED(status)` is true if child terminated abnormally (by receiving a signal it didn't catch). Then, use `WTERMSIG(status)` to retrieve the signal number
- `WCOREDUMP(status)` to see if the child left a core image
- `WIFSTOPPED(status)` is true if the child is currently stopped. Then, use `WSTOPSIG(status)` to determine the signal that caused this.

Finally, with `WNOHANG` **option**, if the requested PID has not terminated, `waitpid` returns immediately instead of blocking.

FORK – EXAMPLE 1

```
#include <unistd.h>    //fork()
#include <sys/wait.h>   //wait()
#include <stdio.h>
#include <stdlib.h>     //exit()

int main(){
    pid_t pid ; //process data type (pid_t)

    pid = fork(); //create new proc
    if (pid < 0){
        printf ("error");
    }else if (pid==0){ //check if child proc
        printf("\n Hello I am the child process ");
        printf("\n%d - %d - %d", pid, getpid(), getppid());
        exit(0);
    }else{ // check if parent proc
        wait(0);
        printf("\n Hello I am the parent process ");
        printf("\n%d - %d - %d \n", pid, getpid(), getppid());
    }
}
```

WAITPID EXAMPLE

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

ORPHAN PROCESSES

What happens if parent terminates before the child?

- The child becomes an orphan and the init process (systemd) becomes the (adopted) parent process
- The init process will now use `wait` to monitor termination information consists of the PID, termination status, and amount of CPU time taken by the process

ZOMBIES VS ORPHANS

- Zombie is a process that has terminated, but whose parent has not yet called the `wait/waitpid` function
- Orphan is a process whose parent has terminated and is inherited by the `init` process

INTER-PROCESS COMMUNICATION (IPC)

Processes on the same system

- Shared memory with Synchronization Objects (Locks, Semaphores, ...)
- Unnamed (anonymous) Pipes
- FIFO queues
- Message queues

Processes on different systems (LAN, WAN, ...)

- Sockets
- STREAMS



SHARED MEMORY

Shared memory allows two or more processes to communicate by sharing a persistent physical memory segment to their virtual address space

- Processes share the memory region for transferring data
 - Writes are immediately visible to other processes (fastest IPC)
 - However, processes must synchronize accesses to the region
 - Use IPC lock, barrier, semaphore, condition variable, ...

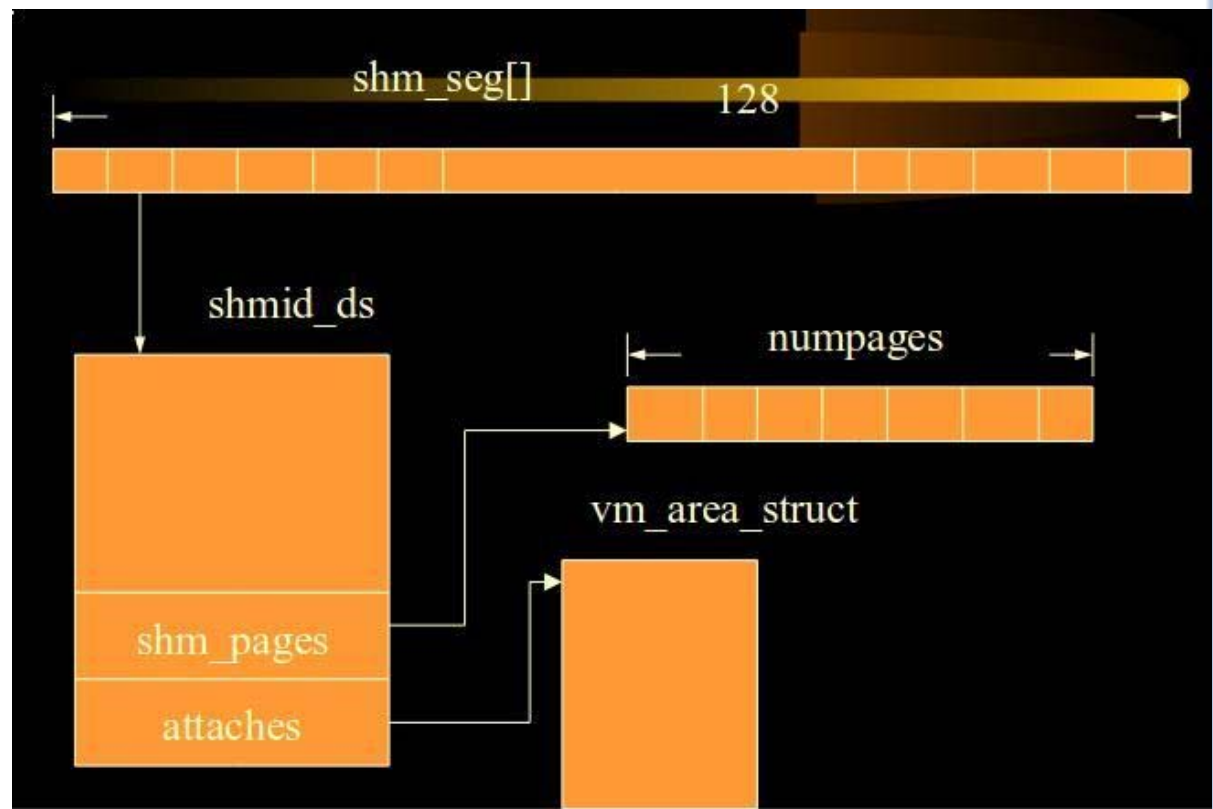
KERNEL SUPPORT FOR SHARED MEMORY

- The `shm_ids` variable stores IPC shared memory resources
- It includes pointers to `shmid_kernel` data structures

User API

`ipcs` - lists all IPC objects owned by the user

`ipcrm` - removes specific IPC object



<https://www.halolinux.us/kernel-reference/ipc-shared-memory.html>

XSI (SYSTEM V) SHARED MEMORY

The kernel maintains a data structure for each shared memory segment:

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* operation perms */
    int shm_segsz;               /* size of segment (bytes) */
    time_t shm_atime;            /* last attach time */
    time_t shm_dtime;            /* last detach time */
    time_t shm_ctime;            /* last change time */
    unsigned short shm_cpid;     /* pid of creator */
    unsigned short shm_lpid;     /* pid of last operator */
    short shm_nattch;            /* no. of current attaches */
    /* the following are private */
    unsigned short shm_npages;   /* size of segment (pages) */
    unsigned long *shm_pages;    /* array of ptrs to frames -> SHMMAX */
    struct vm_area_struct *attaches; /* descriptors for attaches */
};
```

XSI (SYSTEM V) SHARED MEMORY

```
struct ipc_perm {  
    key_t key;  
    ushort uid; // owner uid, gid, euid, egid  
    ushort gid;  
    ushort cuid;  
    ushort cgid;  
    ushort mode; // access modes  
    ushort seq; //sequence number  
};
```

API: CREATING & ATTACHING PHYSICAL SEGMENTS

`int shmget(key_t key, int size, int msgflg)`

Shared memory identifier → `key`

Name of the shared memory → `key`

#of bytes → `size`

Flag (IPC_CREAT, IPC_EXCL read, write permission) → `msgflg`

`void *shmat(int shmid, const void *shmaddr, int shmflg);`

Virtual address of the shared memory segment → `shmid`

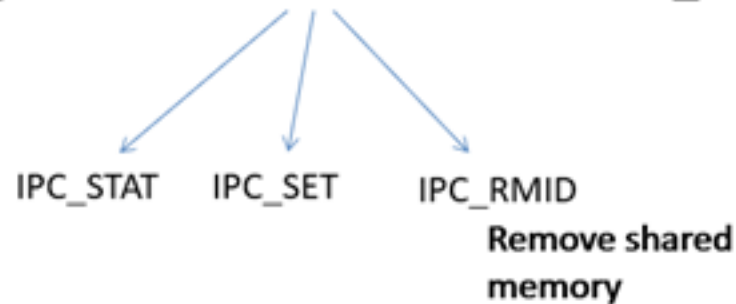
attach occurs at the address `shmaddr` → `shmaddr`

`SHM_RDONLY, SHM_RND` → `shmflg`

API: DETACHING & DELETING PHYSICAL SEGMENTS

```
int shmdt(const void *shmaddr);
```

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```



SHMGET FUNCTION

```
#include <sys/shm.h>
```

```
int shmget(key_t key, size_t size, int flag);
```

Returns: shared memory ID if OK, -1 on error

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

- `key_t key`: If `key` is **IPC_PRIVATE** a new segment is created. Otherwise an old value previously obtained using `ftok` function can be used to access a previously defined shared memory segment
- `size_t size`: size of the shared memory segment in bytes
- `int flag`: bitwise OR combination of 9-bit security r/w flag with
 - **IPC_CREAT**: Create a new segment.
 - **IPC_EXCL**: Used with **IPC_CREAT** to ensure that this call creates the segment. If the segment already exists, the call fails.

KEY GENERATION – FTOK()

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

The `ftok()` function returns a key based on `path` and the eight least significant bits of `id`. The key is usable in subsequent calls to `msgget()`, `semget()`, and `shmget()`.

The `path` argument must correspond to an existing file that the process is able to `stat()`.

SHMGET FUNCTION

When a new segment is created, several fields in `shmid_ds` are initialized.

- `ipc_perm` structure: The `mode` member of this structure is set to the corresponding permission bits of *flag*
- `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are all set to 0
- `shm_ctime` is set to the current time
- `shm_segsz` is set to the requested *size* in bytes

SHMAT FUNCTION

```
#include <sys/shm.h>
```

```
void *shmat(int shmid, const void *addr, int flag);
```

Returns: pointer to SM segment if OK, -1 on error

- `const void *addr`: The address at which the segment is attached depends on *addr* and whether the SHM_RND bit is specified in *flag*.
 - If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the **recommended** technique.
 - If *addr* is nonzero and SHM_RND is not specified, the segment is attached at the address given by *addr*
 - If *addr* is nonzero and SHM_RND is specified, the segment is attached at the address given by $(addr - (addr \bmod SHMLBA))$

SHMAT FUNCTION

- `int flag`: refers to following fields
 - `SHM_EXEC` (Linux-specific; since Linux 2.6.9): Allow the contents of the segment to be executed. The caller must have execute permission
 - `SHM_RDONLY`: Attach the segment for read-only access. Otherwise, the segment is attached as read–write
 - `SHM_REMAP`: (Linux-specific) This flag specifies that the mapping of the segment should replace any existing mapping in the range starting at *shmaddr* and continuing for the size of the segment. In this case, *shmaddr* must not be `NULL`.

SHMDT FUNCTION

```
#include <sys/shm.h>
void *shmdt(const void *addr);
```

Returns: 0 if OK, -1 on error

- `addr` is the value that was returned by a previous call to `shmat`

If successful, `shmdt` detaches the segment (decrements `shm_nattach` counter), but does not remove it from the system. The segment can be later attached to another process via `shmat`, or later removed by calling `shmctl` with a command of `IPC_RMID`.

SHMCTL FUNCTION

```
#include <sys/shm.h>
```

```
int shmctl(int shmid, int cmd, struct shm_id_ds *buf );
```

Returns: 0 if OK, -1 on error

- *cmd*: specifies command to be performed on the *shmid* segment, e.g.
 - IPC_STAT: Fetch the *shm_id_ds* structure for this segment, storing it in the structure pointed to by *buf*.
 - IPC_SET: Set the following three fields from *buf* to the *shm_id_ds* structure associated with *shmid* segment: *shm_perm.uid*, *shm_perm.gid*, and *shm_perm.mode* (requires privileges)
 - IPC_RMID: Remove the *shmid* segment (requires privileges). The segment is not removed until the last process using the segment terminates or detaches it (see *shm_nattach*)
- *buf*: is a pointer to the *shm_id_ds* structure (parameters, statistics)

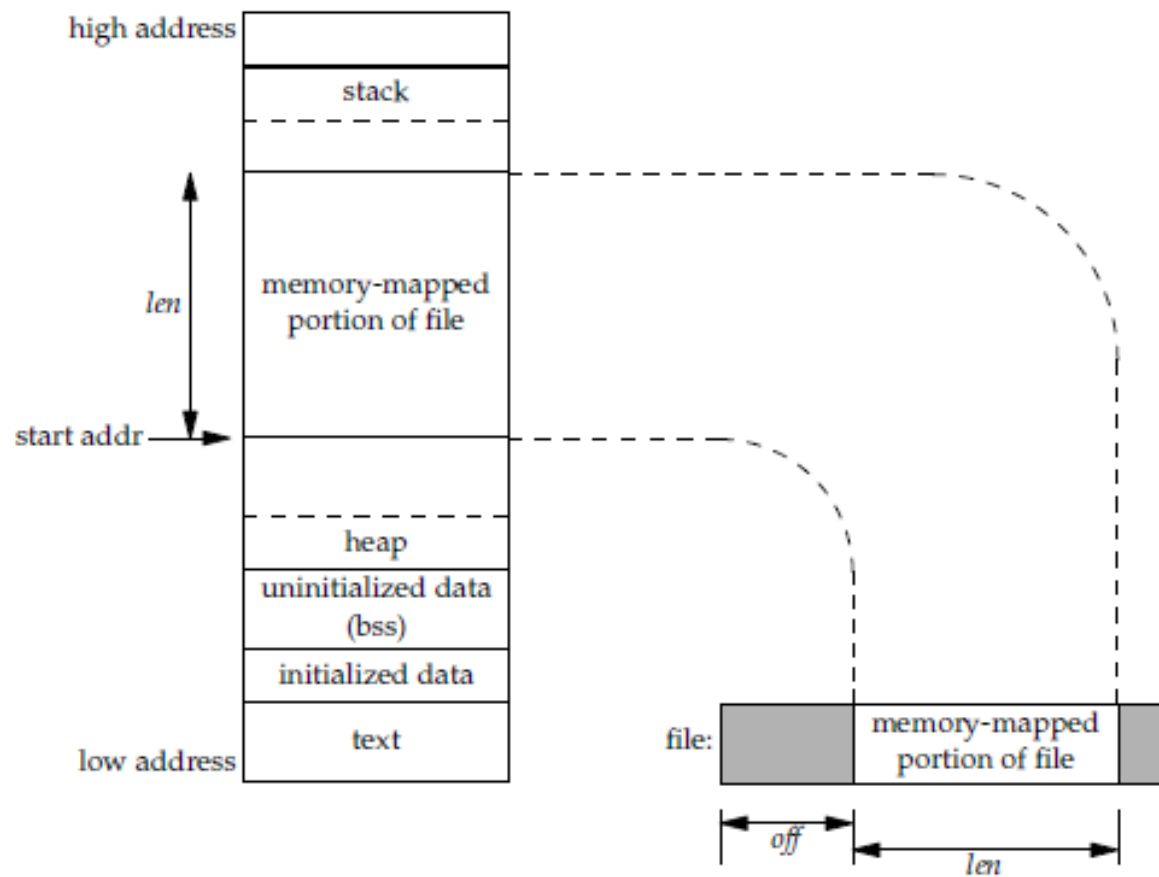
POSIX SHARED MEMORY

POSIX shared memory allows two or more processes to communicate by attaching a persistent physical memory segment to their virtual address space

- Processes share the memory object for transferring data
 - Writes are immediately visible to other processes (efficient)
 - However, processes must synchronize accesses to the region
 - Use IPC: lock, barrier, semaphore, condition variable, ...

Many types of storage data could be shared in the memory, e.g. data files, disks, remote files, etc

POSIX SHARED MEMORY LAYOUT



POSIX SHARED MEMORY

```
struct stat {  
    dev_t st_dev; // ID of device containing file  
    ino_t st_ino; // inode number  
    mode_t st_mode; // protection  
    nlink_t st_nlink; // number of hard links  
    uid_t st_uid; // user ID of owner  
    gid_t st_gid; // group ID of owner  
    dev_t st_rdev; // device ID (if special file)  
    off_t st_size; // total size, in bytes  
    blksize_t st_blksize; // blocksize for file system I/O  
    blkcnt_t st_blocks; // number of 512B blocks allocated  
    time_t st_atime; // time of last access  
    time_t st_mtime; // time of last modification  
    time_t st_ctime; // time of last status change  
};
```

API: CREATING PHYSICAL SEGMENTS

❖ `int shm_open(const char *name, int oflag, mode_t mode);`

file descriptor shared memory object name `O_RDONLY,`
`O_CREATE,`
`O_EXCL,`
`O_TRUNC` e.g. `S_IRWXU |`
`S_IRWXG` Programs using
POSIX shared
memory are
linked with real-
time library
(`gcc -lrt`)

❖ `int ftruncate(int fd, off_t length);`

file descriptor size (bytes)


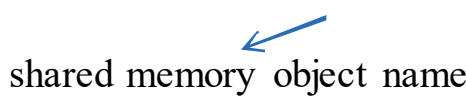
❖ `void *mmap(void *addr, size_t length, int prot,
int flags, int fd, off_t offset);`

`MAP_SHARED`
`MAP_PRIVATE`

file descriptor

`PROT_READ|PROT_WRITE`

API: DETACHING & DELETING A SHARED OBJECT

- ❖ `int munmap(void *addr, size_t length);`

address of unmap size of mapped region
- ❖ `int close(int fd);`
- ❖ `int shm_unlink(const char *name);`

shared memory object name
- ❖ `int fstat(int fd, struct stat *buf);`
- ❖ `int fchown(int fd, uid_t owner, gid_t group);`
- ❖ `int fchmod(int fd, mode_t mode);`

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>
```

MANAGING POSIX SHARED MEMORY - SUMMARY

A POSIX shared memory segment API.

- Create and/or open a shared memory object with `shm_open()`. A file descriptor is returned if `shm_open()` runs successfully
- Set the shared memory object size using `ftruncate()`
- Map the shared memory object into the current virtual address space of the calling process using `mmap()`
- Access the shared memory (`read/write/modify`)
- Unmap the shared memory from virtual address space of the calling process using `munmap()`
- Close the file descriptor of the shared memory object with `close()`
- Delete shared memory object using `shm_unlink()`

SHMOPEN FUNCTION

Create and/or open new/existing a POSIX shared memory object. A file descriptor is returned if successful

```
#include <sys/mman.h>
#include <sys/stat.h> // For mode constants
#include <fcntl.h> // For O_* constants */
int shm_open (const char *name, int oflag, mode_t mode);
Returns: a nonnegative file descriptor if OK, -1 on error
```

- **const char *name:** defines a specific path name, usually /somename
- **int oflag:** bit mask OR combination
 - **O_RDONLY:** Open the object for read
 - **O_RDWR:** Open the object for read/write access
 - **O_CREAT:** Create shared memory object if it does not exist, use `uid/egid` & mode bits except `umask`)
 - **O_EXCL:** if **O_CREAT** was specified, return an error.
 - **O_TRUNC:** initialize the shared memory object to zero bytes
- **mode_t mode:** defines file access mode bits. Symbolic macro definitions from `<sys/stat.h>`

SHMOPEN –MACROS FOR ACCESS PERMISSION

- S_ISUID: set-user-ID on execution
 - S_ISGID: set-group-ID on execution
 - S_ISVTX: on directories, restricted deletion flag
 - S_IRUSR: read permission, owner
 - S_IWUSR: write permission, owner
 - S_IXUSR: execute/search permission, owner
 - S_IRGRP: read permission, group
 - S_IWGRP: write permission, group
 - S_IXGRP: execute/search permission, group
 - S_IROTH: read permission, others
 - S_IWOTH: write permission, others
 - S_IXOTH: execute/search permission, others
-
- S_IRWXU: Read, write, execute/search by owner
 - S_IRWXG: read, write, execute/search by group
 - S_IRWXO: read, write, execute/search by others

FTRUNCATE FUNCTION

Define shared memory of length bytes

```
#include <unistd.h>
int ftruncate(int fd, off_t length);
    Returns: 0 if OK, -1 on error
```

- `int fd`: file descriptor mapped to shared memory
- `off_t length`: number of bytes

MMAP FUNCTION

Map the shared memory object into the virtual address space of the calling process.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flag, int fd,
           off_t off);
```

Returns: starting address of mapped region if OK, -1 on error

- `void *addr`: specify start address of mapped region. If 0, kernel chooses this. The return value of this function is the starting address of the mapped area
- `size_t len`: set the number of bytes to map
- `int prot`: specify protection of the mapped region

<i>prot</i>	Description
<code>PROT_READ</code>	Region can be read.
<code>PROT_WRITE</code>	Region can be written.
<code>PROT_EXEC</code>	Region can be executed.
<code>PROT_NONE</code>	Region cannot be accessed.

MMAP FUNCTION

- `int flag`: affects attributes of the mapped region
 - `MAP_SHARED`: This flag describes a shared mapping, where updates are carried through to the underlying file and are visible to other processes mapping the same region
 - `MAP_PRIVATE`: store operations into the mapped region write to a private copy of the mapped file (copy-on-write)
 - ...
- `int fd`: file descriptor of mapped file
- `off_t off`: starting offset of the file to map

MUNMAP FUNCTION

The memory-mapped region is automatically unmapped when the process terminates or we can unmap a region directly by calling `munmap`

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
           Returns: 0 if OK, -1 on error
```

- `void *addr`: specify start address of shared region to be unmapped.
- `size_t len`: number of bytes to unmap

CLOSE FUNCTION

Closes a file descriptor, so that it no longer refers to any file and may be reused

```
#include <unistd.h>  
int close(int fd);
```

Returns: 0 if OK, -1 on error

SHM_UNLINK FUNCTION

Remove the shared memory object that was created with `shm_open`

```
#include <sys/mman.h>
#include <sys/stat.h> // For mode constants
#include <fcntl.h> // For O_* constants
int shm_unlink(const char *name);
                Returns: 0 if OK, -1 on error
```

EXTRA FUNCTIONS - POSIX SHARED MEMORY

- Obtain a stat structure that describes the shared memory object using `fstat()`. For example, this returns the size (`st_size`), permissions (`st_mode`), owner (`st_uid`), and group info (`st_gid`). See also Linux `stat` command
- Change owner of a shared memory with `fchown()`
- Change permissions of a shared memory using `fchmod()`

FSTAT FUNCTION

Stat info on file descriptor

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int fstat(int fd, struct stat *statbuf);
                        Returns: 0 if OK, -1 on error
```

<https://pubs.opengroup.org/onlinepubs/007908799/xsh/sysstat.h.html>

FCHOWN FUNCTION

Change file permissions

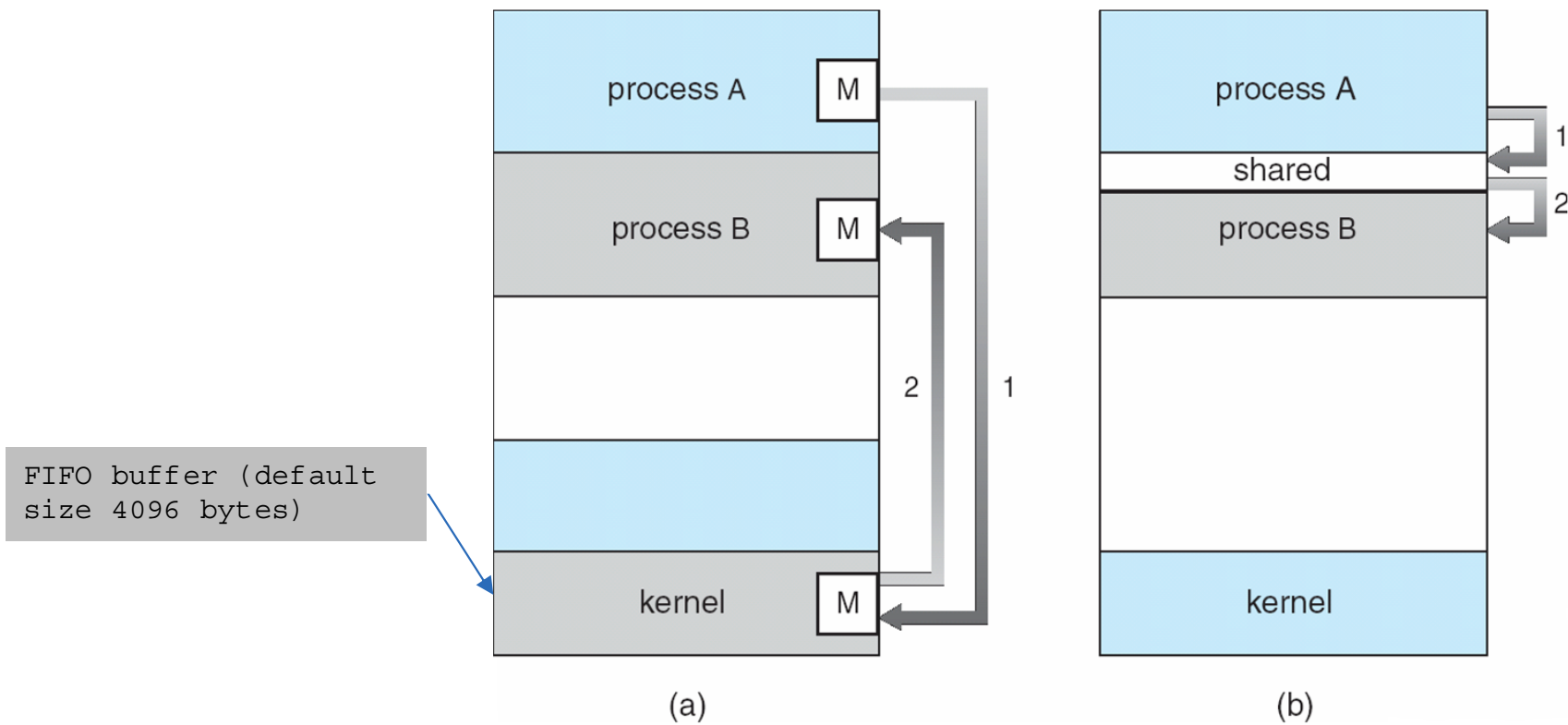
```
#include <unistd.h>
int fchown(const char *pathname, uid_t owner, gid_t group);
Returns: 0 if OK, -1 on error
```


FCHMOD FUNCTION

Change file ownership

```
#include <sys/stat.h>
int fchmod(int fd, mode_t mode);
           Returns: 0 if OK, -1 on error
```

PIPE VS SHARED MEMORY



FILE VS PIPE

- Both can be used to share information among processes
- Both share the file API
- Semantic difference
 - A pipe is a FIFO queue
 - Pipe data can only be read once (data is lost after read)
 - A storage in a file is persistent
 - File data can be used many times.
- Performance difference
 - A pipe is usually realized in memory (r/w), a file on disk (I/O)



UNNAMED VS NAMED PIPES (FIFOS)

○ Unnamed pipes

- An unnamed pipe does not associate with any physical file
- It can only be shared by related processes (descendants of a process that creates the unnamed pipe)
- Created using system call `pipe()`

○ Named pipes

- treat them as files (`mknod`, `mkfifo`, `open`, `read/write`)
- can be shared by any process
- discussed in detail later

UNNAMED PIPES

- Oldest form of IPC in UNIX
- Historically they are half duplex (uni-directional)
- Pipes used between processes that have a common ancestor
 - Typically, a process forks after creating an unnamed pipe
 - pipe can be shared between parent & child.
- Default size 65K bytes (16 pages, limit defined by POSIX)
 - Cannot write more than 65K bytes (`PIPE_BUF`)

Extra functionality (see manual)

- nonatomic writes for large sizes
- nonblocking operations via `fcntl`

UNNAMED PIPES

A pipe is created by calling the `pipe` function

```
#include <unistd.h>
int pipe(int fd[2]);
```

Returns: 0 if OK, -1 on error

Two file descriptors are returned through the `fd` argument:

- `fd[0]` is open for reading, and
- `fd[1]` is open for writing
- output of `fd[1]` is the input for `fd[0]` (since, it's a pipe)

An open file descriptor is closed by calling `close`

```
#include <unistd.h>
int close(int fd);
```

Returns: 0 if OK, -1 on error

UNNAMED PIPES

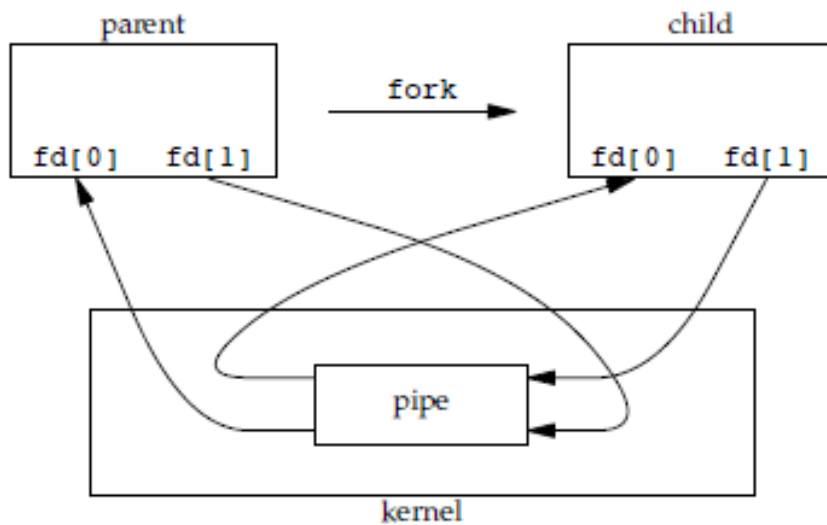
Data is written to the pipe using `write` function (asynchronous)

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes);
Returns: number of bytes written if OK, -1 on error
```

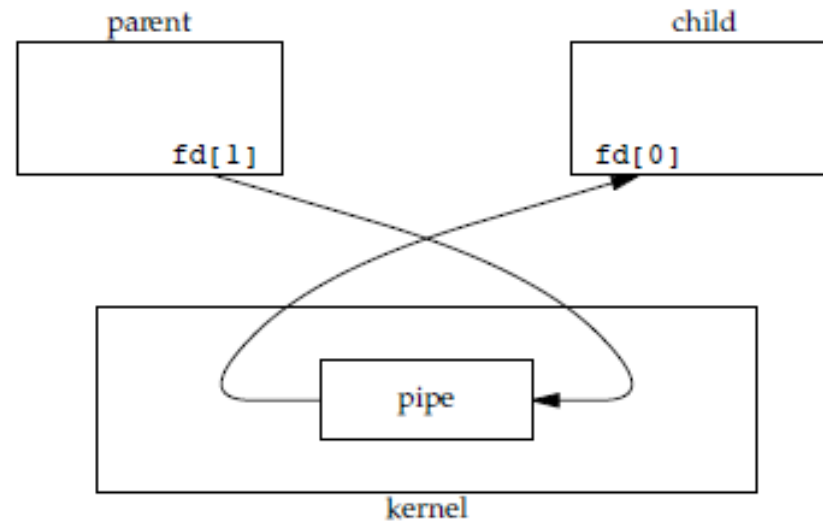
Data is read from the pipe using `read` function (blocking)

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
Returns: number of bytes read, 0 if end of file, -1 on error
```

FORK THE PROCESS?



Pipe after fork



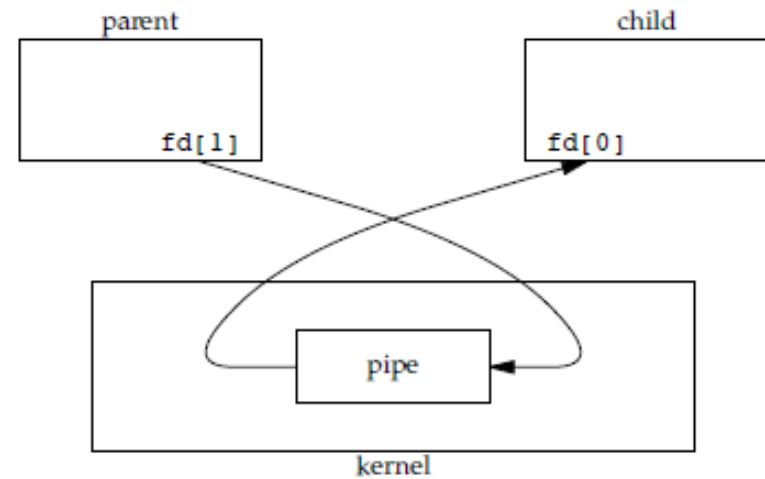
**Pipe from parent to child
(example)**

PIPE AFTER FORK

What we do after fork depends on the direction of data flow. For example (right previous image)

- if the parent process wants to write to the pipe, it must **close its read end of the pipe**, and correspondingly
- if the child process wants to read from the pipe, it must **close its write end of the pipe**

This provides a one-way flow of data between the two processes using pipe messages



Pipe from parent to child

UNNAMED PIPES

- Reading from an empty pipe that has its write end closed returns 0 to indicate EOF
- Writing to a pipe when the read end has been closed generates SIGPIPE signal
 - Ignoring the signal, or returning from the handler causes the corresponding write to return an error with errno set to EPIPE



PIPE QUESTIONS

- How come each end during `fork()` works both ways?
- How is a pipe shared between two processes?
 - When is it declared?
 - How must the processes be related?
 - Can the pipe be used by threads also?
- How can the pipe be used to accomplish one-way communication?
- What happens when we use both I/O mechanisms in the same program?
- How to fix the order in printing?
- How can full-duplex be achieved?
- If multiple processes share one end of a pipe, how can we be sure transmissions aren't interleaved?



POPEN/PCLOSE - LINK COMMAND TO UNNAMED PIPE

```
#include <stdio.h>
```

```
FILE *popen(const char *command, const char *type);
```

Returns: pointer to open stream if OK, NULL on error

- `popen` creates a pipe (or a socket), forks, closes un-needed ends of pipe, execs a shell to run `command` in child, and waits for command to finish
- `type` either `r` to read from stdout, or `w` to write to stdin (of child)
- `FILE*` returned is the created pipe

```
#include <stdio.h>
```

```
int pclose(FILE *stream);
```

Returns: exit status of command if OK, -1 on error

- `pclose` closes standard I/O stream, waits for the `command` to terminate and returns the exit status



EXAMPLE – FROM COMMAND TO PROGRAM VIA POPEN

```
#include <stdio.h>
int main() {
    FILE *fp;
    char buffer[100];
    if ((fp = popen("ls -l", "r")) != NULL) {
        while(fgets(buffer, 100, fp) != NULL) {
            printf("Line from ls:\n");
            printf("  %s\n", buffer);
        }
        pclose(fp);
    }
    return 0;
}
```



HANDS ON - EXECV & OTHER EXEC FUNCTIONS

```
#include <unistd.h>
```

```
int execv(const char * path, char * argv[])
```

Returns: 0 if OK, -1 on error

The `exec()` family of functions replace the current process PCB image (text, data, heap, stack segments) with a new image from disk (using `vfork`)

```
#include <unistd.h>
```

- o `int execv (const char* pathname, const char** argv, ...);`
- o `int execve (const char* pathname, const char** argv, const char** env);`
- o `int execl (const char* pathname, const char* arg, ...);`
- o `int execle (const char* pathname, const char* arg, ..., const char** env);`
- o `int execvp (const char* filename, const char** argv);`
- o `int execlp (const char* filename, const char* arg, ...);`

REDIRECTING STANDARD I/O - DUP & DUP2

If a process communicates only via pipes, then use `dup`, `dup2` to redirect standard I/O (`stdin`, `stdout`) to the appropriate pipe end

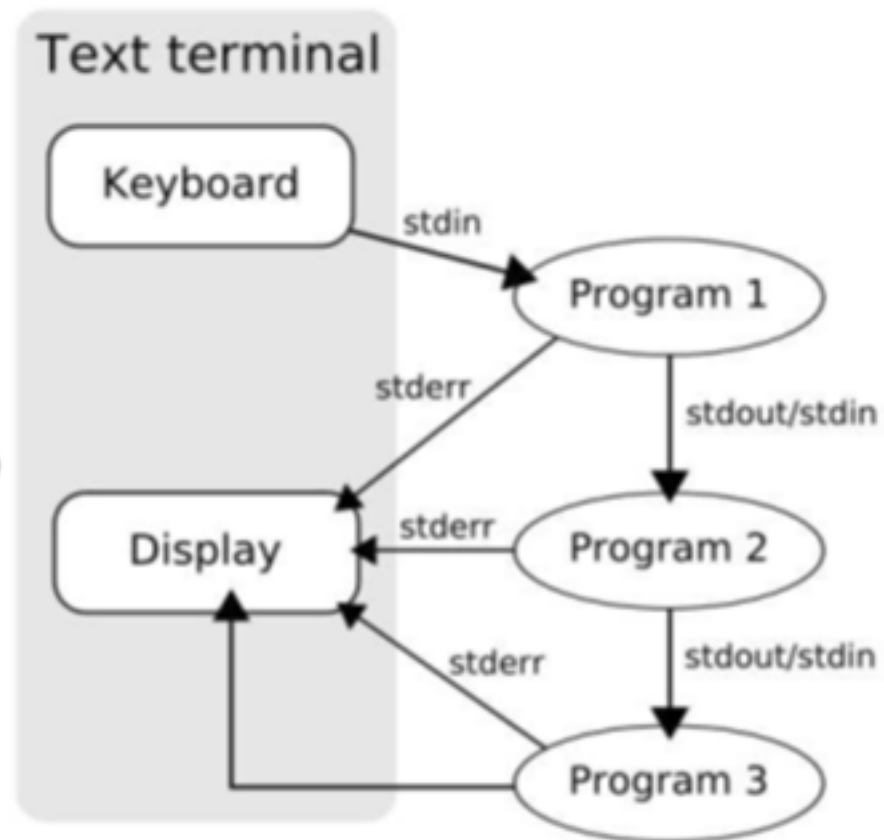
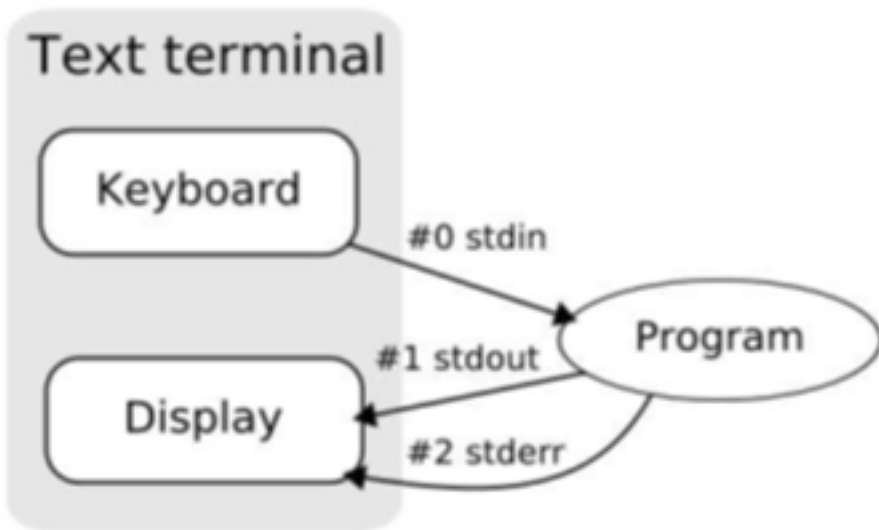
```
#include <unistd.h>
int dup(int fd);
int dup2(int fromFD, int toFD);
```

- `dup` returns a new file descriptor that is a copy of `fd` (first available in file table)
- For example, to `dup` a read pipe end to `stdin` (0)
 - close unused file descriptor (`stdin`), then
 - `dup` the read end of the pipe
- Close unused file descriptors; a process should have only one file descriptor open on a pipe end
- `dup2` duplicates `fromFD` to `toFD`
 - If `toFD` is open, it must be closed first
- If the ends of a pipe are in `fd`, then `dup2(fd[1], 1)` redirects `stdout` to the write end of the pipe
- Must close `fd[1]` (unused read end of pipe)

DUP EXAMPLE

```
if (fork() == 0) { /* Child process */
    close(1) ; dup(fd[1]) ; // Redirect stdout to pipe
    close(fd[0]);
    for (i=0; i < 10; i++) {
        printf("%d\n",n) ; n++;
    }
}
else { /* Parent process */
    close(0) ; dup(fd[0]) ; // Redirect stdin to pipe
    close(fd[1]);
    for (i=0; i < 10; i++) {
        scanf("%d", &n);
        printf("n = %d\n",n) ; sleep(1);
    }
}
```


PIPE EXAMPLES



Q &A