

Other Topics

register, errno, fork, signals,
shared memory, message queues

John Dempsey

COMP-232 – Programming Languages
California State University, Channel Islands
March 23, 2022

What to cover

- Register
- errno
- Signal
- Fork
- Shared Memory
- Message Queues
- Theads
- Semaphores / Mutex
- Database Integration
- Client/Server Networking

register

To speed up your program, you can use:

```
void main() {  
    register int i;  
    register int j;  
  
    for (i=0; i<MAX; i++) {  
        .....
```

fopen – Return Values

NAME

fopen - open a stream

SYNOPSIS

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

```
FILE *fopen(const char *filename, const char *mode);
```

DESCRIPTION

The `fopen()` function opens the file whose pathname is the string pointed to by `filename`, and associates a stream with it.

RETURN VALUES

Upon successful completion, `fopen()` returns a pointer to the object controlling the stream. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

#include <errno.h>

ERRORS

The fopen() function will fail if:

- EACCES** Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
- EINTR** A signal was caught during the execution of fopen().
- EISDIR** The named file is a directory and mode requires write access.
- ELOOP** Too many symbolic links were encountered in resolving path.
- EMFILE** There are {OPEN_MAX} file descriptors currently open in the calling process.
- ENAMETOOLONG** The length of the filename exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
- ENFILE** The maximum allowable number of files is currently open in the system.
- ENOENT** A component of filename does not name an existing file or filename is an empty string.

Checking errno Value

```
if ((fd=fopen(myfile,"a+")) == NULL) {
    printf("----> ERROR: Cannot open file %s.\n", myfile);
    switch (errno) {
        case EACCES:      fprintf(stderr, "----> ERROR: errno returns EACCES.\n"); break;
        case EINTR:      fprintf(stderr, "----> ERROR: errno returns EINTR.\n"); break;
        case EISDIR:     fprintf(stderr, "----> ERROR: errno returns EISDIR.\n"); break;
        case ELOOP:      fprintf(stderr, "----> ERROR: errno returns ELOOP.\n"); break;
        case EMFILE:     fprintf(stderr, "----> ERROR: errno returns EMFILE.\n"); break;
        case ENAMETOOLONG: fprintf(stderr, "----> ERROR: errno returns ENAMETOOLONG.\n"); break;
        case ENFILE:     fprintf(stderr, "----> ERROR: errno returns ENFILE.\n"); break;
        case ENOENT:     fprintf(stderr, "----> ERROR: errno returns ENOENT.\n"); break;
        case ENOSPC:     fprintf(stderr, "----> ERROR: errno returns ENOSPC.\n"); break;
        default:         fprintf(stderr, "----> ERROR: errno returns undefined value.\n"); break;
    }
    printf("Program will now exit.\n");
    exit(1);
}
```

Signals

```
signal( SIGALRM, timeout);
```

← If alarm goes off, call timeout().

```
alarm (10);
```

← Alarm will go off in 10 seconds.

```
// Do something.
```

```
alarm(0);
```

← Turn off alarm.

```
void timeout()
```

```
{
```

```
    printf("ERROR: Timeout occurred.\n");
```

```
}
```

Signals

The signals currently defined by <signal.h> are as follows:

| Name | Value | Default | Event |
|----------|-------|---------|--------------------------|
| SIGHUP | 1 | Exit | Hangup (see termio(4I)) |
| SIGINT | 2 | Exit | Interrupt (^C) |
| SIGQUIT | 3 | Core | Quit (see termio(4I)) |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGTRAP | 5 | Core | Trace or Breakpoint Trap |
| SIGABRT | 6 | Core | Abort |
| SIGEMT | 7 | Core | Emulation Trap |
| SIGFPE | 8 | Core | Arithmetic Exception |
| SIGKILL | 9 | Exit | Killed |
| SIGBUS | 10 | Core | Bus Error |
| SIGSEGV | 11 | Core | Segmentation Fault |
| SIGSYS | 12 | Core | Bad System Call |
| SIGPIPE | 13 | Exit | Broken Pipe |
| SIGALRM | 14 | Exit | Alarm Clock |
| SIGTERM | 15 | Exit | Terminated |
| SIGUSR1 | 16 | Exit | User Signal 1 |
| SIGUSR2 | 17 | Exit | User Signal 2 |
| SIGCHLD | 18 | Ignore | Child Status Changed |
| SIGPWR | 19 | Ignore | Power Fail or Restart |
| SIGWINCH | 20 | Ignore | Window Size Change |

There are more signals defined too...

Program can ignore or handle signals:

```
sigset(SIGINT, SIG_IGN);
sigset(SIGHUP, SIG_IGN);
signal(SIGSYS, error_seen);
signal( SIGTERM, error_seen);
signal( SIGPWR, error_seen);
signal( SIGILL, error_seen);
signal( SIGFPE, error_seen);
signal( SIGBUS, error_seen);
signal( SIGSEGV, error_seen);
signal( SIGUSR1, SIG_IGN);
signal( SIGUSR2, error_seen);
```

```
// Program continues on ...
```


fork

- When a process calls fork, it is deemed the parent process and the newly created process is its child.
- After the fork, the parent and child process don't know if they are the parent or child until the process id returned from the fork call is checked.
- fork() call from the parent process will return the process id of the child.
- fork() call from the child process will return 0.
- After the fork, both processes resume execution starting at the fork call.
- Based on the return value, the parent and child can perform different functions.

fork example

```
switch (pid = fork()) {  
    case -1:  
        printf("----> ERROR: Unable to create child process.\n");  
        exit(0);  
    case 0:  
        go_do_child_stuff();  
        break;  
    default:  
        printf("PARENT started CHILD process id %d\n", pid);  
        break;  
}  
... Parent process continues
```

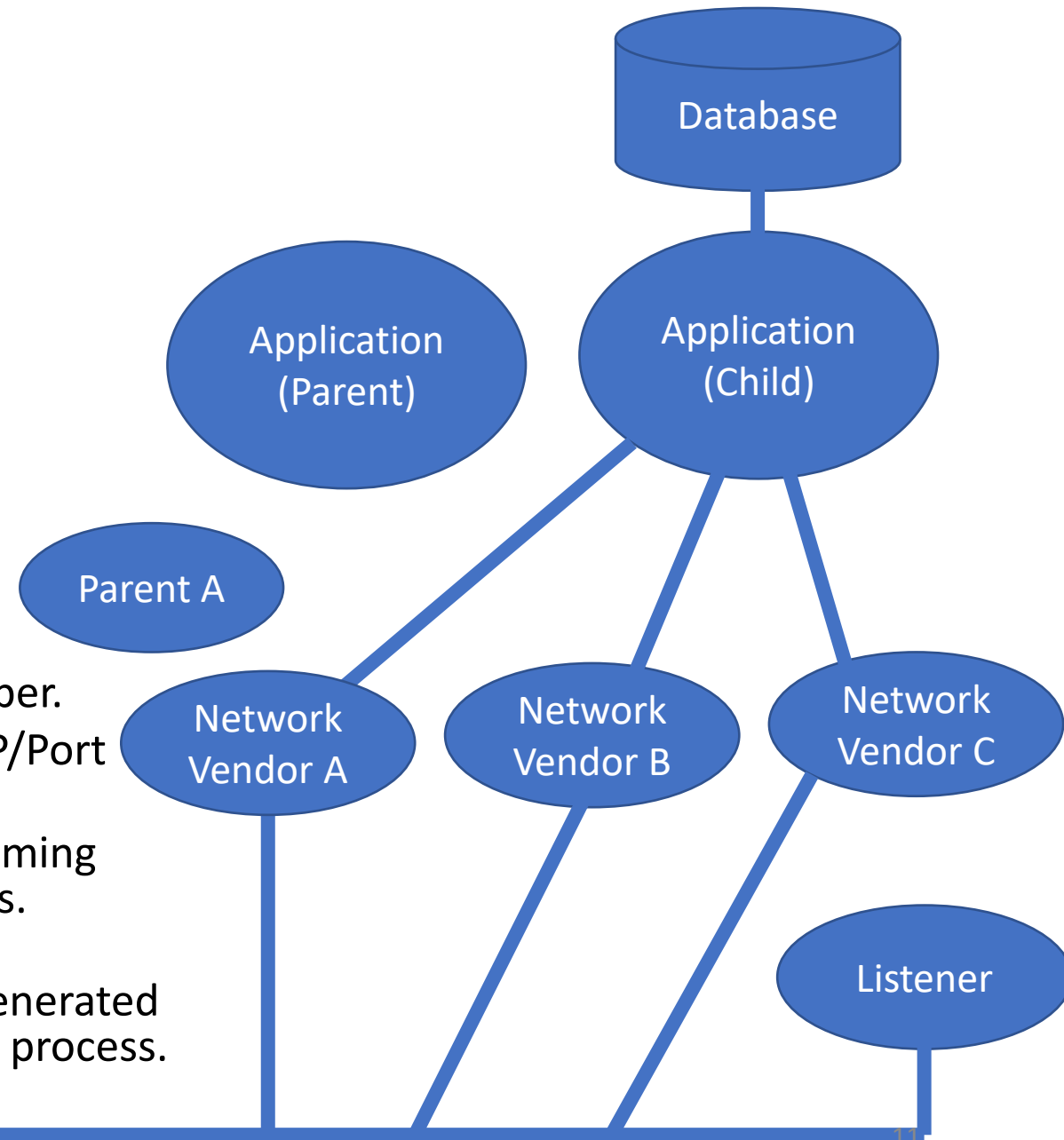
fork Usage

Application Level

- Start application.
- Have parent spawn a child process.
- Parent monitors child.
- If child process dies, parent respawns child process to keep application up and running.

Network Level

- Start listener process at IP Address/Port Number.
- When a network connection is requested at IP/Port Number, parent spawns a child process.
- Child then spawns another child process becoming the parent process that monitors child process. E.g., Parent A monitors Network Vendor A.
- The child process handles networking at IP/Generated Port Number. If it dies, parent respawns child process.



Shared Memory Calls - shmget

NAME

shmget - get shared memory segment identifier

SYNOPSIS

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

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

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

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

RETURN VALUES

Upon successful completion, a non-negative integer representing a shared memory identifier is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

shmflg = 0644 | IPC_CREAT

To create a shared memory id, we can use:

```
if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1)
```

But wait, IPC_CREATE is defined as:

```
#define IPC_CREATE 0x200
```

So there is no change?

We want:

```
if ((shmid = shmget(key, SHM_SIZE, 01644)) == -1)
```

Answer

```
if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1)
```

0644 is in octal.

IPC_CREAT is 0x200, but 0x200 is in hex.

0644 = 000 110 100 100

0x200 = 0010 0000 0000

| | | | | | | | | | | | | |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|
| 0644 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0x200 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 01644 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

So there is a change and

01644 = 0644 | IPC_CREAT

shmat – Shared Memory Attach

NAME

shmop, shmat, shmdt - shared memory operations

SYNOPSIS

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

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

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

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

RETURN VALUES

Upon successful completion, shmat() returns the data segment start address of the attached shared memory segment. Otherwise, SHM_FAILED (-1) is returned, the shared memory segment is not attached, and errno is set to indicate the error.

Create Shared Memory Segment

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

#define SHM_SIZE 1024

int main()
{
    key_t key = 1234;
    int shmid;
    // Create shared memory.
    // Tricky Note: IPC_CREAT = 0x200 hex, 0644 is octal.
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1)
    {
        printf("ERROR: shmget failed.\n");
        exit(1);
    }
    return 0;
}
```


Copy b.c to b1.c, b2.c, b3.c. Update memory.

```
#define SHM_SIZE 1024
// Usage: b "text to write into shared memory"
int main(int argc, char *argv[])
{
    key_t key = 1234;
    int shmid;
    char *shm_ptr;
    int mode;

    // Get shared memory.
    if ((shmid = shmget(key, SHM_SIZE, 0644)) == -1) {
        printf("ERROR: shmget failed.\n"); exit(1);
    }
    // Attach to shared memory. shmat returns pointer.
    shm_ptr = shmat(shmid, (void *)0, 0);
    if (shm_ptr == (char *)(-1)) {
        printf("ERROR: shmat failed.\n"); exit(1);
    }
}
```

```
// Read from shared memory.
printf("Read from shared memory: \"%s\"\n",
shm_ptr);

// Write to shared memory.
printf("Write to shared memory: \"%s\"\n",
argv[1]);
strncpy(shm_ptr, argv[1], SHM_SIZE);

// Read from shared memory.
printf("Read from shared memory: \"%s\"\n",
shm_ptr);

// Detach from shared memory.
if (shmdt(shm_ptr) == -1) {
    printf("ERROR: shmdt failed.\n"); exit(1);
}
return 0;
}
```

Delete Shared Memory

```
#define SHM_SIZE 1024

int main(int argc, char *argv[])
{
    key_t key = 1234;
    int shmid;

    if ((shmid = shmget(key, SHM_SIZE, 0644)) == -1) {        // Get shared memory id.
        printf("ERROR: shmget failed.\n");
        exit(1);
    }
    shmctl(shmid, IPC_RMID, NULL);    // Delete shared memory.
    return 0;
}
```

Sample Run for Shared Memory

```
john@oho:~/SHM$ b1 "Hello World"  
ERROR: shmget failed.
```

```
john@oho:~/SHM$ create_shm
```

```
john@oho:~/SHM$ b1 "Hello World"  
Read from shared memory: ""  
Write to shared memory: "Hello World"  
Read from shared memory: "Hello World"
```

```
john@oho:~/SHM$ b2 "This is a test"  
Read from shared memory: "Hello World"  
Write to shared memory: "This is a test"  
Read from shared memory: "This is a test"
```

```
john@oho:~/SHM$ b3 "Shared Memory Works!"  
Read from shared memory: "This is a test"  
Write to shared memory: "Shared Memory Works!"  
Read from shared memory: "Shared Memory Works!"
```

```
john@oho:~/SHM$ delete_shm
```

```
john@oho:~/SHM$ b1 "Hello again"  
ERROR: shmget failed.
```

```
john@oho:~/SHM$ ls  
b.c b1 b1.c b2 b2.c b3 b3.c create_shm create_shm.c  
delete_shm delete_shm.c
```

msgget

NAME

msgget - get message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

DESCRIPTION

The msgget() argument returns the message queue identifier associated with key.

% more c.c

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

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

```
int main()
```

```
{
```

```
    int mq;
```

```
    mq = msgget(1000, 0666|IPC_CREAT);
```

← Creates the message queue 1000.

```
    printf("message queue 1000 created.\n");
```

```
}
```

msgsnd / msgrcv

NAME

msgsnd - message send operation

SYNOPSIS

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

DESCRIPTION

The msgsnd() function is used to send a message to the queue associated with the message queue.

NAME

msgrcv - message receive operation

SYNOPSIS

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

DESCRIPTION

The msgrcv() function reads a message from the queue associated with the message queue identifier specified by msqid and places it in the user-defined buffer pointed to by msgp.

msgsnd

```
% cat s.c
#include <stdio.h>
#include <sys/msg.h>

struct my_message {
    long  mtype;
    char  mtext[100];
} my_msg;

int main()
{
    int  mq;
    int  return_val = 0;

    mq = msgget(1000, 0666);

    my_msg.mtype = 100;
    strcpy(my_msg.mtext, "Hello!");
```

```
    if ((return_val = msgsnd(mq, &my_msg,
                             sizeof(struct my_message), 0666)) < 0) {
        printf("msgsnd failed.\n");
    }
    if ((return_val = msgsnd(mq, &my_msg,
                             sizeof(struct my_message), 0666)) < 0) {
        printf("msgsnd failed.\n");
    }

    my_msg.mtype = 200;
    strcpy(my_msg.mtext, "Bye!");

    if ((return_val = msgsnd(mq, &my_msg,
                             sizeof(struct my_message), 0666)) < 0) {
        printf("msgsnd failed.\n");
    }

    printf("msgsnd worked.\n");
}
```

msgrcv – mtype = 100

```
% cat r100.c
```

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

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

```
struct my_message {  
    long  mtype;  
    char  mtext[100];  
} my_msg;
```

```
int main()  
{  
    int  mq;  
    int  return_val = 0;  
  
    mq = msgget(1000, 0666);  
  
    if ((return_val = msgrcv(mq, &my_msg, sizeof(struct my_message), 100, 0666)) < 0) {           // Return message with type = 100  
        printf("msgrcv failed.\n");  
    }  
  
    printf("msgrcv: my_msg.mtype = %d, my_msg.mtext = %s\n", my_msg.mtype, my_msg.mtext);  
    printf("msgrcv worked.\n");  
}
```

msgrcv – mtype = 200

```
% cat r200.c
```

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

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

```
struct my_message {  
    long  mtype;  
    char  mtext[100];  
} my_msg;
```

```
int main()
```

```
{
```

```
    int  mq;
```

```
    int  return_val = 0;
```

```
    mq = msgget(1000, 0666);
```

```
    if ((return_val = msgrcv(mq, &my_msg, sizeof(struct my_message), 200, 0666)) < 0) {    // mtype = 200
```

```
        printf("msgrcv failed.\n");
```

```
    }
```

```
    printf("msgrcv: my_msg.mtype = %d, my_msg.mtext = %s\n", my_msg.mtype, my_msg.mtext);
```

```
    printf("msgrcv worked.\n");
```

```
}
```


msgget, msgsnd, msgget Example

% c

message queue 1000 created.

% s ← Puts {100, "Hello!"}, {100, "Hello!"}, {200, "Bye!"} on queue.

msgsnd worked.

%r100

msgrcv: my_msg.mtype = 100, my_msg.mtext = Hello!

msgrcv worked.

%r200

msgrcv: my_msg.mtype = 200, my_msg.mtext = Bye!

msgrcv worked.

%r100

msgrcv: my_msg.mtype = 100, my_msg.mtext = Hello!

msgrcv worked.

%r100

← No more 100 message types on queue. r100 blocks/waits for next 100 type.

^C

%r200

← No more 200 message types on queue. r200 blocks/waits for next 200 type.

^C

Message Queues

Message queues are used to send incoming transactions from the Network Vendor process to the Application Process.

Separate message queues are used to send responses from the Application to the Network Vendor process.

