Linux's Guide to the Galaxy

Terminating Processes

void _exit(int status);

LIBRARIES: unistd

DESCRIPTION:

_exit() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, init, and the process's parent is sent a SIGCHLD signal.

*int atexit(void (*function)(void));*

LIBRARIES: stdlib

DESCRIPTION:

atexit() function registers the given function to be called at nor- mal process termination, either via exit(3) or via return from the pro- gram's main(). Functions so registered are called in the reverse order of their registration; no arguments are passed.

Process Creation

pid_t fork(void);

LIBRARIES: unistd

DESCRIPTION:

fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent

RETURNS: -1 (error), 0(If it is the child), > 0 (child's pid)

*pid_t wait(int *status);*
*pid_t waitpid(pid_t pid, int *status, int options);*

LIBRARIES: sys/types, sys/wait

DESCRIPTION:

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child ter- minated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

MACROS for status:

- **WIFEXITED(stat_val)** Nonzero if the child is terminated normally
- **WEXITSTATUS(stat_val)** If WIFEXITED is nonzero, this returns child exit code
- **WIFSIGNALED(stat_val)** Nonzero if the child is terminated on an uncaught signal
- **WTERMSIG(stat_val)** If WIFSIGNALED is nonzero, this returns a signal number
- **WIFSTOPPED(stat_val)** Nonzero if the child has stopped
- **WSTOPSIG(stat_val)** If WIFSTOPPED is nonzero, this returns a signal number

PARAMETERS:

int *status, int *stat_loc: Returned status from child

int options:

- **WNOHANG** return immediately if no child has exited
- **WUNTRACED** return if a child has stopped (but not traced via ptrace()). Status for traced children which have stopped is provided even if this option is not specified
- **WCONTINUED** also return if a stopped child has been resumed by delivery of **SIGCONT**.

Shared Memory

int shmget(key_t key, size_t size, int shmflg);
*int shmctl(int shmid, int cmd, struct shmid_ds *buf);*

LIBRARIES: sys/ipc, sys/shm

DESCRIPTION:

shmget() returns the identifier of the System V 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 if key has the value IPC_PRIVATE or key isn't IPC_PRIVATE, no shared memory segment corresponding to key exists, and IPC_CREAT is specified in shmflgpid_t wait(int *status);

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

PARAMETERS:

key_t key: Random integer or IPC_PRIVATE
size_t size: Size to allocate, in byteç
int shmflg:

- **NULL** or
- **IPC_CREAT** Create a new segment. If this flag is not used, then **shmget**() will find the segment associated with *key* and check to see if the user has permission to access the segment.
- **IPC_EXCL** This flag is used with **IPC_CREAT** to ensure that this call creates the segment. If the segment already exists, the call fails.
- **S_IRUSR** Allow the owner of the shared memory segment to attach to it in read mode.
- **S_IWUSR** Allow the owner of the shared memory segment to attach to it in write mode.
- **S_IRGRP** Allow the group of the shared memory segment to attach to it in read mode.
- **S_IWGRP** Allow the group of the shared memory segment to attach to it in write mode.
- **S_IROTH** Allow others to attach to the shared memory segment in read mode.
- **S_IWOTH** Allow others to attach to the shared memory segment in write mode.

int cmd:

- **IPC_STAT** Copy information from the kernel data structure associated with *shmid* into the *shmid_ds* structure pointed to by *buf*. The caller must have read permission on the shared memory segment.
- **IPC_SET** Write the values of some members of the *shmid_ds* structure pointed to by *buf* to the kernel data structure associated with this shared memory segment, updating also its *shm_ctime* member. The following fields can be changed: *shm_perm.uid*, *shm_perm.gid*, and (the least significant 9 bits of) *shm_perm.mode*. The effective UID of the calling process must match the owner (*shm_perm.uid*) or creator (*shm_perm.cuid*) of the shared memory segment, or the caller must be privileged.
- **IPC_RMID** Mark the segment to be destroyed. The segment will actually be destroyed only after the last process detaches it. The caller must be the owner or creator of the segment, or be privileged. The *buf* argument is ignored.

*void *shmat(int shm_id, const void *shm_addr, int shmflg)*
*int shmdt(const void *shmaddr);*

LIBRARIES: sys/types.h, sys/shm

DESCRIPTION:

shmat() attaches the System V shared memory segment identified by shmid to the address space of the calling process.

shmdt() detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process. The to-be-detached segment must be currently attached with shmaddr equal to

PARAMETERS:

int shm_id: id taken from **shmget()**
const void *shm_addr: If *shmaddr* is NULL, the system chooses a suitable (unused) pagealigned address to attach the segment.
int shmflg:

- **NULL** or
- **SHM_EXEC** Allow the contents of the segment to be executed. The caller must have execute permission on the segment.
- **SHM_RDONLY** Attach the segment for read-only access. The process must have read permission for the segment. If this flag is not specified, the segment is attached for read and write access, and the process must have read and write permission for the segment. There is no notion of a write-only shared memory segment.
- **SHM_REMAP** 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. (Normally, an **EINVAL** error would result if a mapping already exists in this address range.) In this case, *shmaddr* must not be NULL.

const void *shmaddr (of shmdt): Address to detach (taken from **shmat()**'s result)

Anonymous Pipes

*FILE *popen(const char *command, const char *type);*
*int pclose(FILE *stream);*

LIBRARIES: stdio

DESCRIPTION:

popen() function opens a process by creating a pipe, forking, and invoking the shell. Since a pipe is by definition unidirectional, the type argument may specify only reading or writing, not both; the resulting stream is correspondingly read-only or write-only.

pclose() function waits for the associated process to terminate and returns the exit status of the command as returned by wait4(2).

PARAMETERS:

const char *type: If *mode* is ***"r" ("w")*** the *stream* is the stream pointer returned by *popen*() shall be the **readable (writable)** end of the pipe.

int pipe(int pipefd[2]);

LIBRARIES: unistd.h

DESCRIPTION:

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

RETURNS: 0 if successful, error code otherwise

Named Pipes/FIFOs

*int mkfifo(const char *pathname, mode_t mode);*
*int mknod(const char *pathname, mode_t mode, dev_t dev);*

LIBRARIES: sys/types, sys/stat

DESCRIPTION:

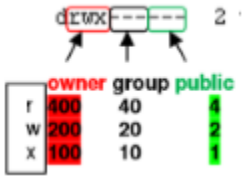
mkfifo() makes a FIFO special file with name pathname. mode specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

mknod() creates a file system node (file, device special file or named pipe) named pathname, with attributes specified by mode and dev.

PARAMETERS:

pathname: String containing the path to the file. Example: "/tmp/my_fifo"

mode: Permissions given to file. Example: 0777



int pipe(int pipefd[2]);

LIBRARIES: unistd.h

DESCRIPTION:

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

RETURNS: 0 if successful, error code otherwise

Threads

*int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);*
*int pthread_join(pthread_t thread, void **retval);*
*void pthread_exit(void *retval);*

LIBRARIES: pthread

DESCRIPTION:

pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

thread_exit() function terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls pthread_join().

Exec Command Family

extern char **environ;
int execl(const char *path, const char *arg, …);
int execlp(const char *file, const char *arg, …);
int execle(const char *path, const char *arg, …, char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);

LIBRARIES: unistd

DESCRIPTION:

The **exec()** family of functions replaces the current process image with a new process image. The initial argument for these functions is the name of a file that is to be executed. The const char *arg and subsequent ellipses in the execl(), execlp(), and execle() functions can be thought of as arg0, arg1, ..., argn. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments must be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast (char *) NULL.

The **execv()**, **execvp()**, and **execvpe()** functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers must be terminated by a NULL pointer. The **execle()** and **execvpe()** functions allow the caller to specify the environment of the executed program via the argument envp. The envpargument is an array of pointers to null-terminated strings and must be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable environ in the calling process.

Other Useful Commands

ssize_t read/write(int fildes, void *buf, size_t nbyte); - System read/write functions using buffer of nbytes

void *memset(void *str, int c, size_t n) - copies the character c (an unsigned char) to the first n characters of str

long int strtol(const char *str, char **endptr, int base) - converts the initial part of the string in str to a long int value according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.

char *strncpy(char *dest, const char *src, size_t n) - copies up to n characters from the string pointed to, by src to dest.

shmctl (segment_id, IPC_RMID, 0) - Deallocate a shared memory segment

void perror(const char *s); - Print system error. Example: perror("shmget");

int open(const char *path, int oflags) - Used to open a new file and obtain its file descriptor. (Flags: O_RDONLY, O_WRONLY, O_NONBLOCK)

Example Codes

Pipes

```
int read_example()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) printf("Output was:-\n%s\n", buffer);
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

```
int write_example()
{
    FILE *write_fp;
    char buffer[BUFSIZ + 1];
    sprintf(buffer, "Once upon a time, there was...\n");
    write_fp = popen("od -c", "w");
    if (write_fp != NULL) {
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        pclose(write_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

```
#define MSGLEN 64
int walkie-talkie()
{
    int fd[2];
    pid_t pid;
    char message[MSGLEN];
    if(pipe(fd) == 0){
        pid = fork();
        if(pid < 0) exit(EXIT_FAILURE);
        if(pid == 0){
            while(1){
                memset(message, '\0', sizeof(message));
                printf("(Child) Type: ");
                scanf("%s", message);
                write(fd[1], message, strlen(message));
                sleep(1);
            }
        }else{
            while(1){
                memset(message, '\0', sizeof(message));
                read(fd[0], message, sizeof(message));
                printf("(Parent) Received: %s\n", message);
            }
        }
    }
    return 0;
}
```

Shared Memory

```
int cook-waitor(int argc, char **argv)
{
    int key = 1234;
    int shmid;
    int *pizza;
    if((shmid = shmget(key, sizeof(int), IPC_CREAT | 0666)) < 0)
exit(1)
    if((pizza = shmat(shmid, NULL, 0)) < 0) exit(1);
    int pizza_in_stock = 10;
    while(pizza_in_stock > 0){
        (*pizza)++;
        printf("Cook added a pizza! (%d)(%d)\n", *pizza,
pizza_in_stock);
        sleep(1);
        pizza_in_stock--;
    }
    printf("Cook: I am out of pizza! Bye\n");
    //while(1){
        //(*pizza)--;
        //printf("Waitor delivered a pizza! (%d)\n", *pizza);
        //sleep(2);
        //if(pizza < 0) break;
    //}
    //printf("Mamma mia! We are out of pizza!\n");
    shmctl(shmid, IPC_RMID, (struct shmid_ds*) 0);
    return 0;
}
```

Threads

```
void *thread_function(void *arg);
char message[] = "Hello World";

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);

    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);

    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined, it returned %s\n", (char *)thread_result);
```

```
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    printf("thread_function is running. Argument was %s\n", (char
*)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for the CPU time");
}
```

```
#define NUM_OF_THREADS 4
int matrix[4][10];
void *calc_sum(void *thread_id){
    int j, tid = *((int *)thread_id);
    int *sum = malloc(sizeof(int));
    *sum = 0;
    for(j=0;j<10;j++)
        *sum += matrix[tid][j];
    printf("Line %d , sum = %d\n", tid, *sum);
    free(thread_id);
    pthread_exit(sum);
}
void print_table(){
    int i,j;
    for(i=0;i<4;i++){
        for(j=0;j<10;j++)
            printf("%d ", matrix[i][j]);
        printf("\n");
    }
}
int main(int argc, char **argv)
{
    pthread_t pthreads[NUM_OF_THREADS];
    int i, j;
    srand(time(NULL));
    for(i=0;i<4;i++)
        for(j=0;j<10;j++)
            matrix[i][j] = rand()%20;
    print_table();
    int t, sum, result = 0;
    void *thread_result;
    for(t=0; t<NUM_OF_THREADS; t++){
        int *arg = malloc(sizeof(int));
        *arg = t;
        if(pthread_create(&pthreads[t], NULL, calc_sum, arg))
            free(arg);
    }
    for(t=0; t<NUM_OF_THREADS; t++){
        pthread_join(pthreads[t], &thread_result);
        sum += *((int *)thread_result);
    }
    printf("Final sum: %d", sum);
    return 0;
}
```

Exec Command Family

```
/* Example of an argument list! */
/* Note that we need a program name for argv[0] */
char *const ps_argv[] = {“ps”, “ax”, 0};
/* Example enviroment*/
char *const pos_envp[]={“PATH=/bin:/usr/bin”, “TERM=console”, 0};
/* Possible calls to exec functions*/
execl(“/bin/ps”, “ps”, “ax”, 0); /* assumes ps is in /bin*/
execlp(“ps”, “ps”, “ax”, 0); /*assumes /bin is in PATH */
execle(“/bin/ps”, “ps”, “ax”, 0, ps_envp);/*passes own enviroment*/

execv(“/bin/ps”, ps_argv);
execvp(“ps”, ps_argv);
execve(“/bin/ps”, ps_argv, ps_envp);
```

Process Creation

```
pid_t childpid;
int retval, status;
childpid = fork();
if (childpid >= 0){
    if (childpid == 0){
        printf("CHILD: I am the child process!\n");
        printf("CHILD: Here's my PID: %d\n", getpid());
        printf("CHILD: My parent's PID is: %d\n", getppid());
        printf("CHILD: The value of my copy of childpid is: %d\n",
childpid);
        printf("CHILD: Sleeping for 1 second...\n");
        sleep(1);
        printf("CHILD: Enter an exit value (0 to 255): ");
        scanf(" %d", &retval);
        printf("CHILD: Goodbye!\n");
        exit(retval);
    }else{
        printf("PARENT: I am the parent process!\n");
        printf("PARENT: Here's my PID: %d\n", getpid());
        printf("PARENT: The value of my copy of childpid is %d\n",
childpid);
        printf("PARENT: I will now wait for my child to exit.\n");
        wait(&status); /* wait for child to exit, and store its status */
        printf("PARENT: Child's exit code is: %d\n",
WEXITSTATUS(status));
        printf("PARENT: Goodbye!\n");
        exit(0);
    }
}else{
    perror("fork");
    exit(0);
}
```

```
if (pid != 0) {
    int stat_val;
    pid_t child_pid;
    child_pid = wait(&stat_val);
    printf("Child has finished: PID = %d\n", child_pid);
    if(WIFEXITED(stat_val))
        printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
    else
        printf("Child terminated abnormally\n");  }  exit(exit_code);
```

Named Pipes – FIFO’s

```
    #define FIFO_NAME "/tmp/my_fifo"
    #define BUFFER_SIZE PIPE_BUF
    #define TEN_MEG (1024 * 1024 * 10)
    int main()
```

```
{
    int pipe_fd;
    int res;
    int open_mode = O_WRONLY;
    int bytes_sent = 0;
    char buffer[BUFFER_SIZE + 1];
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n",
FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }
    printf("Process %d opening FIFO O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);
    if (pipe_fd != -1) {
        while(bytes_sent < TEN_MEG) {
            res = write(pipe_fd, buffer, BUFFER_SIZE);
            if (res == -1) {
                fprintf(stderr, "Write error on pipe\n");
                exit(EXIT_FAILURE);
            }
                bytes_sent += res;
        }
        (void)close(pipe_fd);
    }else {
        exit(EXIT_FAILURE);
    }
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

```
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;
    memset(buffer, '\0', sizeof(buffer));
    printf("Process %d opening FIFO O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);
    if (pipe_fd != -1) {
        do {
            res = read(pipe_fd, buffer, BUFFER_SIZE);
            bytes_read += res;
        } while (res > 0);
        (void)close(pipe_fd);
    } else {
        exit(EXIT_FAILURE);
    }
    printf("Process %d finished, %d bytes read\n", getpid(),
bytes_read);
    exit(EXIT_SUCCESS);
}
```

Example Exercise: Addition using Memory & Args (Different files)

```
int main(int argc, char **argv)
{
    pid_t pid;
    int segment_id;
    int *sm;

    segment_id = shmget(IPC_PRIVATE, 10*sizeof(int), S_IRUSR |
S_IWUSR);
    sm = (int *)shmat(segment_id, NULL, 0);
    int i;
    char par1[10], par2[10];
    for(i=0;i<10;i++){
        if(fork() == 0){
            srand(time(NULL) ^ (getpid()<<16));
            if(i%2 == 0){ //Memory Addition (add_mem.c)
                for(i=0;i<10;i++)
                    sm[i] = rand()%10;
                sprintf(par1, "%d", segment_id);
                sprintf(par2, "%d", 10);
                execl("add_mem", "add_mem", par1, par2, NULL);
            }else{ //Arg Addition(add_excl.c)
                sprintf(par1, "%d", rand()%20);
                sprintf(par2, "%d", rand()%20);
                execl("add_execl", "add_execl", par1, par2, NULL);
            }
            shmdt(sm);
            shmctl(segment_id, IPC_RMID, 0);
        }
    }
    return 0;
}
```

```
/* add_mem.c*/
int main(int argc, char **argv)
{
    int seg_id = strtol(argv[1], NULL, 10);
    int nums = strtol(argv[2], NULL, 10);
    int *sm = (int *)shmat(seg_id, NULL, 0);
    int i, sum=0;
    for(i=0;i<nums;i++){
        printf("%d", sm[i]);
        if(i != nums-1) printf(" + ");
        sum += sm[i];
    }
    printf(" = %d (Using mem)\n", sum);
    return 0;
}
```

```
/* add_execl */
int main(int argc, char **argv)
{
    int x1 = strtol(argv[1], NULL, 10);
    int x2 = strtol(argv[2], NULL, 10);
    printf("%d + %d = %d (Using Args)\n", x1, x2, x1+x2);
    return 0;
}
```