```c
/*
 * Introducing process creation using fork().
 * What is the output of this program?
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    int pid = fork();
    int i = 10;
    switch (pid)
    {
        case −1:
            perror("cannot fork()");
            break;
        case 0:
            i = 40;
            printf("child: %d", i);
            // child
            break;
        default:
            // parent.
            printf("parent: %d", i);
            pid = fork();
            break;
    }
    printf("\n");
    return 0;
}
```



```c
/*
 * This example shows that you cannot predict
 * the sequence of execution between parent and
 * child.
 */

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>

int main()
{
    pid_t pid = fork();
    int i;
    switch (pid)
    {
        case −1:
            perror("cannot fork()");
            break;
        case 0:
            // child
            for (i = 0; i < 10; i++)
                printf("\tchild: %d\n", i);
            break;
        default:
            // parent
            for (i = 0; i < 10; i++)
                printf("parent: %d\n", i);
            break;
    }
    return 0;
}
```

```c
/*
 * Illustrates exit() and atexit() functions.
 *
 * What happen when exit() is changed to _exit()?
 */

#include <stdio.h>
#include <stdlib.h>

void foo()
{
    printf ("foo\n");
}

void bar()
{
    printf ("bar\n");
}

int main()
{
    atexit(foo);
    atexit(bar);
    exit(0);
}
```



```c
/*
 * Child process should use _exit() instead
 * of exit().
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    int pid;

    printf("before fork ");
    pid = fork();
    switch (pid)
    {
        case −1:
            perror("cannot fork()");
            break;
        case 0:
            // child
            _exit(0);
            break;
        default:
            // parent.
            exit(0);
            break;
    }
    return 0;
}
```

Jun 02, 04 21:00 **eg05−wait.c** Page 1/1

```c
/*
 * You can do simple synchronization between parent
 * and child by using wait() and waitpid()
 */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main()
{
    int status;
    int pid = fork();
    switch (pid) {
        case −1 :
            perror("failed to fork()"); break;
        case 0 :
            // child
            _exit(41);
        default:
            // parent
            pid = waitpid(pid, &status, 0);
            switch (pid) {
                case −1:
                   // wait failed.
                   break;
                case 0 :
                   printf("no child available\n"); break;
                default:
                   printf("%d is done\n", pid);
                   if (WIFEXITED(status))
                       printf("exit status of child is %d\n",
                           WEXITSTATUS(status));
                   break;
            }
    }
    return 0;
}
```

Jun 02, 04 21:00 **eg06−errno.c** Page

```c
/*
 * You can use errno global variable to handle
 * errors from system calls or use perror which
 * prints default error messages.
 */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main()
{
    int status;
    int pid = fork();
    switch (pid) {
        case −1 :
            perror("failed to fork()"); break;
        case 0 :
            // child
            _exit(41);
        default:
            // parent
            pid = waitpid(pid, &status, 4014);
            switch (pid) {
                case −1:
                   switch (errno) {
                       case ECHILD:
                            fprintf(stderr, "invalid child\n"); bre
                       case EINVAL:
                            fprintf(stderr, "invalid options\n"); b
k;
                   }
                   perror("wait failed");
                   break;
                case 0 :
                   printf("no child available\n"); break;
                default:
                   printf("%d is done\n", pid);
                   if (WIFEXITED(status))
                       printf("exit status of child is %d\n",
                            WEXITSTATUS(status));
                   break;
            }
    }
    return 0;
}
```

Jun 02, 04 21:00 **eg07−wait.c** Page 1/1

```c
/*
 * Option WNOHANG of waitpid is useful if you do not
 * want to block the parent process when waiting for
 * child process to finish.
 */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main()
{
    int status;
    int pid = fork();
    switch (pid) {
        case −1 :
            perror("failed to fork()"); break;
        case 0 :
            // child
            sleep(10);
            _exit(41);
        default:
            // parent
            while ((pid = waitpid(pid, &status, WNOHANG)) == 0)
            {
                printf ("make myself useful while waiting ..\n");
                sleep(1);
            }
            switch (pid) {
                case −1:
                   // wait failed.
                   break;
                case 0 :
                   printf("no child available\n"); break;
                default:
                   printf("%d is done\n", pid);
                   if (WIFEXITED(status))
                       printf("exit status of child is %d\n",
                           WEXITSTATUS(status));
                   break;
            }
    }
    return 0;
}
```

Jun 02, 04 20:59 **eg08−exec.c** Page

```c
/*
 * exec* family of functions allow us to _replace_
 * the current process with another process.
 */
#include <stdio.h>
#include <unistd.h>

int main()
{
    if (execlp("echo.sh", "echo.sh", "a.c", "a.out", 0) != −1)
        printf("this should not be printed\n");
    else
        perror("execlp failed");
    return 0;
}
```

```c
/*
 * A combination of fork and exec* allows us to write
 * a simple shell.
 */
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#define MAX_CMD_LEN 1024

int main()
{
    char buf[MAX_CMD_LEN+1];

    buf[0] = 0;

    printf("$ ");
    while (fgets(buf, MAX_CMD_LEN, stdin) != NULL)
    {
        int status;
        int pid = fork();
        switch (pid) {
            case −1:
                perror("cannot fork");
                break;
            case 0:
                buf[strlen(buf)−1] = 0;
                execlp(buf, buf, 0);
                perror("exec: ");
                _exit(3);
                break;
            default:
                pid = waitpid(pid, &status, 0);
                if (WIFEXITED(status))
                    printf("exit status of child is %d\n",
                        WEXITSTATUS(status));
                printf("$ ");
        }
    }
    return 0;
}
```



```c
/*
 * system() is not a system call but is useful
 * if you just want to run some shell command.
 * It forks a shell and execute the command you
 * passed in.
 */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    system("cat *.c | wc −l");
    return 0;
}
```

Jun 02, 04 20:57 **eg11−environ.c** Page 1/1

```c
/*
 * Some exec* function expects you to pass in
 * the environment variables.  You can access
 * the current environment variables using
 * global variable environ.
 */
#include <stdio.h>

extern char **environ;

int main()
{
    int i;
    for (i = 0; environ[i] != NULL; i++)
    {
        printf("%s\n", environ[i]);
    }
    return 0;
}
```

Jun 02, 04 20:58 **eg12−getenv.c** Page

```c
/*
 * You can manipulate the current environment
 * variables from C using getenv() putenv()
 * and from bash using "export".
 */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("%s\n", getenv("LD_LIBRARY_PATH"));
    return 0;
}
```

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }
    return 0;
}
```



```c
/*
 * Introducing read() and write() -- kernel's IO
 * functions.  Note that they do not care whether
 * the input is binary or text.  That will be
 * supported by higher level functions such as
 * fgets().
 *
 * Also note the use of STDOUT_FILENO in write
 * (instead of stdout)
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

#define BUFFER_SIZE 1024
int main(int argc, char *argv[])
{
    int len;
    char buf[BUFFER_SIZE];
    int fd = open(argv[1], O_RDONLY);

    if (fd == -1) {
        perror("open");
        return 1;
    }
    while ((len = read(fd, buf, BUFFER_SIZE)) > 0)
    {
        write(STDOUT_FILENO, buf, len);
    }
    if (len == -1)
        perror("read");
    return 0;
}
```

```c
/*
 * Using dup and dup2 for I/O redirection.
 */

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

int main(int argc, char *argv[])
{
    int oldstdout, fd;
    if (argc < 3)
    {
        fprintf(stderr, "usage: %s command outputfile\n", argv[0]);
        exit(1);
    }

    fd = open(argv[2], O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    if (fd == -1) {
        perror("open");
        exit(2);
    }

    oldstdout = dup(STDOUT_FILENO);
    dup2(fd, STDOUT_FILENO);

    if (execlp(argv[1], argv[1], NULL) == -1)
    {
        perror("execlp");
        exit(1);
    }

    dup2(oldstdout, STDOUT_FILENO);

    return 0;
}
```