UTSA The University of Texas at San Antonio™

EE3233 Systems Programming for Engrs

Reference: M. Kerrisk, The Linux Programming Interface

Lecture 11

Process Creation/Termination

Electrical & Computer Engineering

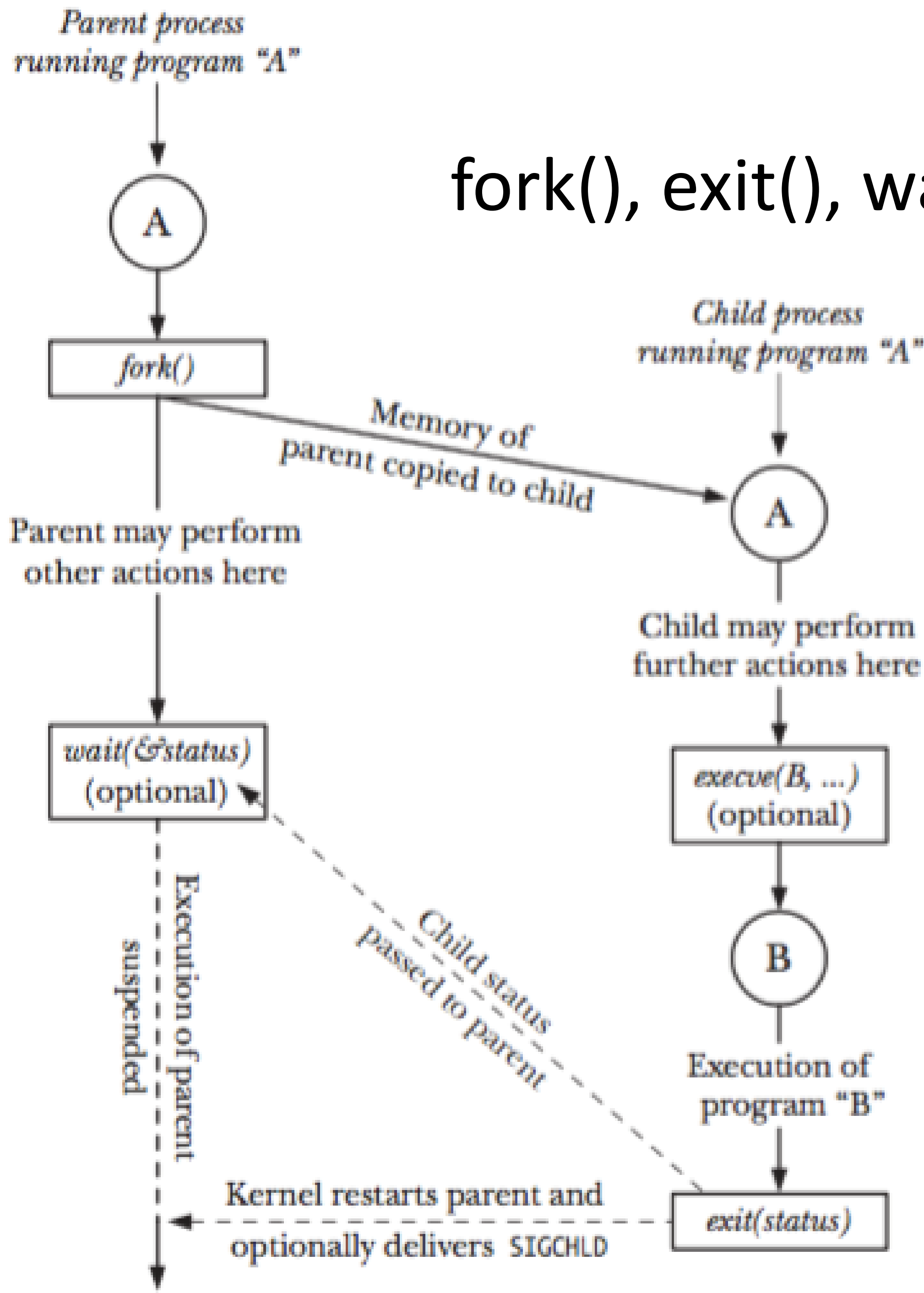
Overview of fork(), exit(), wait(), execve()

- fork()
  - allows one process (parent) to create a new process (child)
  - exact duplication of the parent: the child obtains copies of the parent's stack, data, heap, and text segments
- exit(status)
  - terminates a process
  - makes all resources (memory, open file descriptor) used by the process available for subsequent reallocation by the kernel
  - *status* is a termination status for the process

Overview of fork(), exit(), wait(), execve()

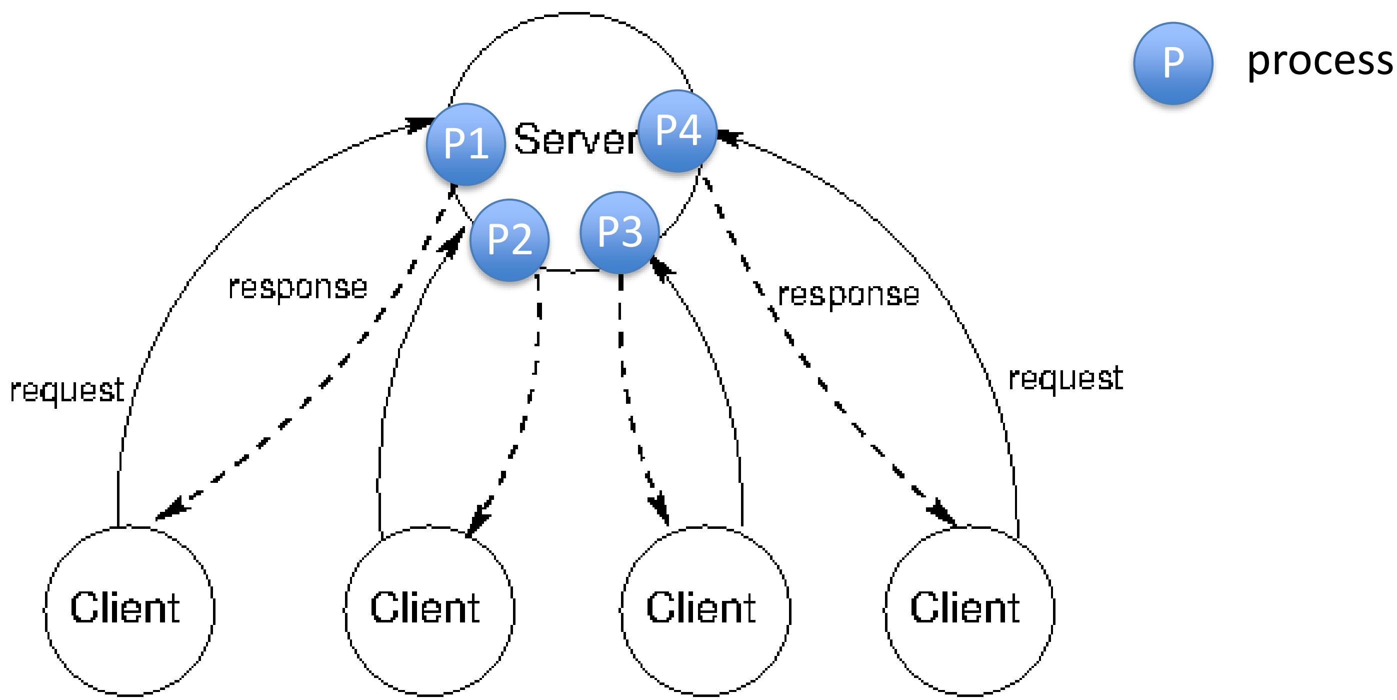
- wait(&status)
  - If a child of this process has not yet terminated by calling *exit()*, then *wait()* suspends execution of the process until one of its children has terminated
- execve(pathname, argv, envp)
  - loads a new program (*pathname* with argument list *argv*, and environment list *envp*) into a process's memory
  - The existing program text is discarded, and the stack, data, and heap segments are freshly created for the new program

Overview of fork(), exit(), wait(), execve()



Creating a New Process: fork( )

- Creating multiple processes can be a useful way of dividing up a task
- A network server process may listen for incoming client requests and create a new process to handle each request
  - meanwhile, the server process continues to listen for further client connections (great concurrency)



Creating a New Process: fork( )

```
#include <unistd.h>

pid_t  fork(void);
```

- Two processes are executing the same program text
  - but they have separate copies of the stack, data, and heap segments
- Each process can modify the variables in its stack, data, and heap without affecting the other process
- For parent, returns **PID of child** on success, or -1 on error
- For the child, *fork()* returns **0**
  - child can obtain its own process ID using *getpid()*,
  - and get its parent process ID using *getppid()*

Creating a New Process: fork( )

General format to use fork( )

```
pid_t childPid;                /* Used in parent after successful fork()
                                  to record PID of child */
switch (childPid = fork()) {
case -1:                       /* fork() failed */
    /* Handle error */

case 0:                        /* Child of successful fork() comes here */
    /* Perform actions specific to child */

default:                       /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```

procexec/t_fork.c

```c
#include "tlpi_hdr.h"

static int idata = 111;             /* Allocated in data segment */

int
main(int argc, char *argv[])
{
    int istack = 222;               /* Allocated in stack segment */
    pid_t childPid;

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:
        idata *= 3;
        istack *= 3;
        break;

    default:
        sleep(3);                   /* Give child a chance to execute */
        break;
    }

    /* Both parent and child come here */

    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);

    exit(EXIT_SUCCESS);
}
```

Creating a New Process: fork( )

- After a *fork()*, it is indeterminate which of the two processes is next scheduled to use the CPU
  - Poorly written programs can lead errors known as race condition
- *sleep()* in the program permits the child to be scheduled for the CPU before the parent, so that the child can complete its work and terminate before the parent continues execution
- Result

```
$ ./t_fork
PID=28557 (child)    idata=333 istack=666
PID=28556 (parent) idata=111 istack=222
```

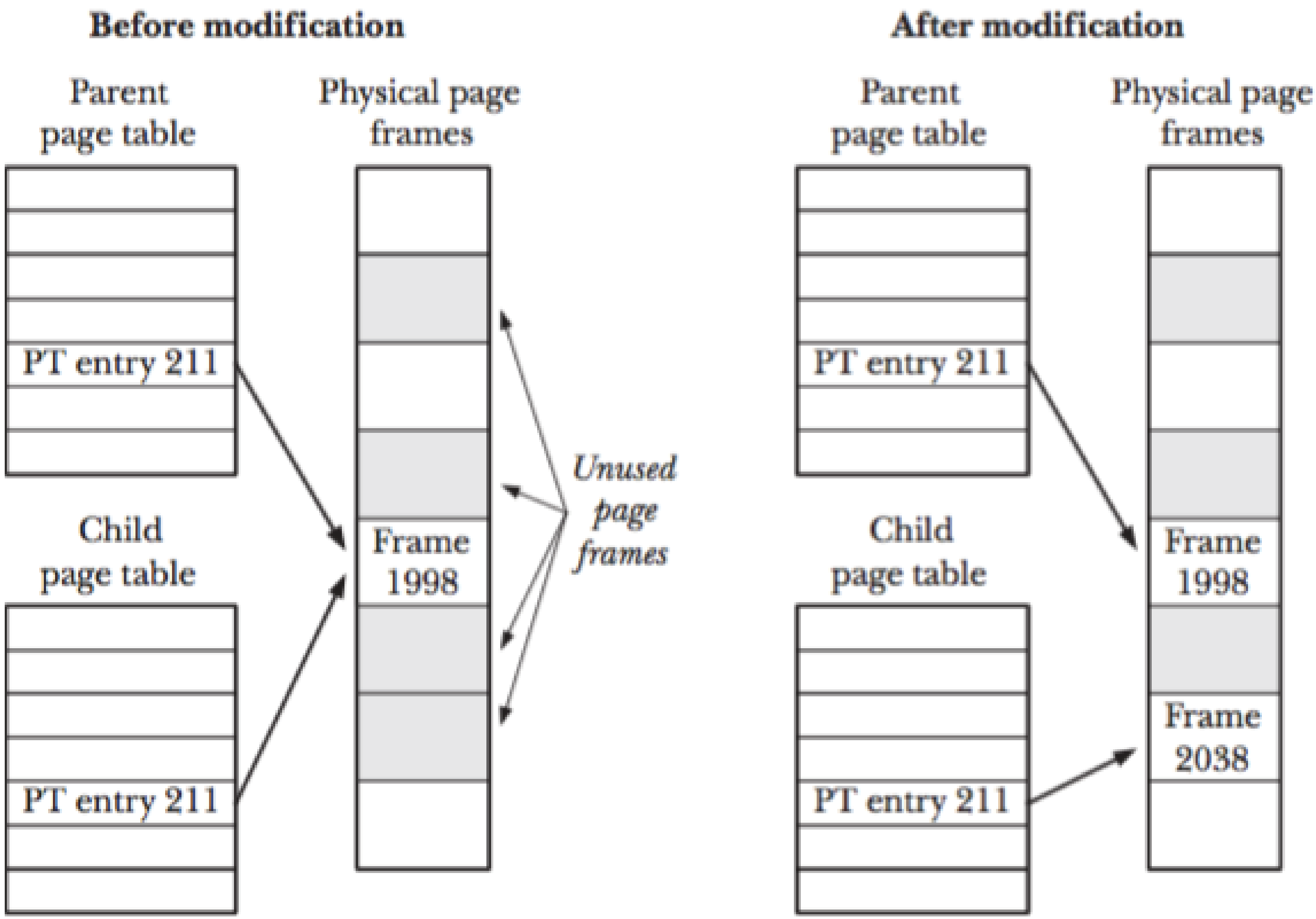
Memory Semantics of *fork()*

- Copy of parent's virtual memory pages into the new child process would be wasteful
  - *fork()* is often followed by an immediate *exec()*, which replaces the process's text with a new program and reinitializes the process's data, heap, and stack segment
- Modern UNIX implementations including Linux use two techniques to avoid such wasteful copying

Two Techniques to Avoid Wasteful Copying

- Kernel marks text segment of each process as *read-only*, so that a process can’t modify
  - Parent and child can share the same text segment
  - *fork()* creates page-table entries that refer to the same physical page frames used by parent
- *Copy-on-write*
  - Initially, page table entries refer to the same physical memory pages as the corresponding page-table entries in the parent, and the pages themselves are marked read-only
  - After fork(), kernel traps any attempts by either parent or child to modify one of pages, and makes a duplicate copy of the about-to-modified page

Two Techniques to Avoid Wasteful Copying



Race Conditions After *fork()*

```
#include <sys/wait.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numChildren, j;
    pid_t childPid;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [num-children]\n", argv[0]);

    numChildren = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-children") : 1;

    setbuf(stdout, NULL);              /* Make stdout unbuffered */

    for (j = 0; j < numChildren; j++) {
        switch (childPid = fork()) {
        case -1:
            errExit("fork");

        case 0:
            printf("%d child\n", j);
            _exit(EXIT_SUCCESS);

        default:
            printf("%d parent\n", j);
            wait(NULL);                /* Wait for child to terminate */
            break;
        }
    }

    exit(EXIT_SUCCESS);
}
```

```
$ procexec/fork_whos_on_first 2
0 parent
0 child
1 parent
1 child
```

Race Conditions After *fork()*

- This program loops, using *fork()* to create multiple children
  - After each *fork()*, both parent and child print a message containing the loop counter value and a string indicating parent of child
- When using this program to create 1 million children on a Linux/x86-32 2.2.19 system showed that the parent printed its message first in all but 332 cases (99.97%)
  - The reason that the child occasionally printed its message first was that the parent's <u>CPU time slice</u> ran out before it had time to print its message

Waiting for a Signal using a Mask

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

- replaces the process signal mask by the signal set pointed to by *mask*, and then suspends execution of the process until a signal is caught and its handler returns
- Calling *sigsuspend()* is equivalent to atomically performing these operations
  - sigprocmask(SIG_SETMASK, &mask, &prevMask);
  - pause();
  - sigprocmask(SIG_SETMASK, &prevMask, NULL);

procexec/fork_sig_sync.c

```
#include <signal.h>
#include "curr_time.h"                  /* Declaration of currTime() */
#include "tlpi_hdr.h"

#define SYNC_SIG SIGUSR1                /* Synchronization signal */

static void             /* Signal handler - does nothing but return */
handler(int sig)
{
}

int
main(int argc, char *argv[])
{
    pid_t childPid;
    sigset_t blockMask, origMask, emptyMask;
    struct sigaction sa;

    setbuf(stdout, NULL);               /* Disable buffering of stdout */

    sigemptyset(&blockMask);
    sigaddset(&blockMask, SYNC_SIG);    /* Block signal */
    if (sigprocmask(SIG_BLOCK, &blockMask, &origMask) == -1)
        errExit("sigprocmask");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = handler;
    if (sigaction(SYNC_SIG, &sa, NULL) == -1)
        errExit("sigaction");
```

```
switch (childPid = fork()) {
case -1:
    errExit("fork");

case 0: /* Child */

    /* Child does some required action here... */

printf("[%s %ld] Child started - doing some work\n",
        currTime("%T"), (long) getpid());
sleep(2);              /* Simulate time spent doing some work */

/* And then signals parent that it's done */

printf("[%s %ld] Child about to signal parent\n",
        currTime("%T"), (long) getpid());
if (kill(getppid(), SYNC_SIG) == -1)
    errExit("kill");

/* Now child can do other things... */

_exit(EXIT_SUCCESS);
```

```
    default: /* Parent */

        /* Parent may do some work here, and then waits for child to
           complete the required action */

        printf("[%s %ld] Parent about to wait for signal\n",
                currTime("%T"), (long) getpid());
        sigemptyset(&emptyMask);
        if (sigsuspend(&emptyMask) == -1 && errno != EINTR)
            errExit("sigsuspend");
        printf("[%s %ld] Parent got signal\n", currTime("%T"), (long) getpid());

        /* If required, return signal mask to its original state */

        if (sigprocmask(SIG_SETMASK, &origMask, NULL) == -1)
            errExit("sigprocmask");

        /* Parent carries on to do other things... */

        exit(EXIT_SUCCESS);
    }
}
```

```
$ ./fork_sig_sync
[17:59:02 5173] Child started - doing some work
[17:59:02 5172] Parent about to wait for signal
[17:59:04 5173] Child about to signal parent
[17:59:04 5172] Parent got signal
```

Terminating a Process: _exit() and exit()

- A process terminate in two ways
  - abnormal
  - normal
- Abnormal termination
  - caused by the delivery of a signal whose default action is to terminate the process
- Normal termination
  - terminated using *_exit()* system call

_exit()

```
#include <unistd.h>

void _exit (int status);
```

- status
  - ***0*** indicates that a process completed successfully
  - ***nonzero*** indicates unsuccessful termination

exit()

```
#include <stdlib.h>

void exit (int status);
```

- Following actions are performed by exit()
  - Exit handlers are called (We will cover)
  - *stdio* stream buffers are flushed
  - *_exit()* system call is invoked using *status*
- Returning from *main()*, either explicitly, or implicitly, by falling off the end of *main()* terminates a process
  - Performing an explicit return *n* is generally equivalent to calling *exit(n)*, since run-time function that invokes *main()* uses that value in a call to *exit()*

Details of Process Termination

- During both normal and abnormal termination of a process, following actions occur:
  - open file descriptors, directory stream (a structure of type DIR when open a directory) are closed
  - file locks (we will cover) held by this process are released
  - If this is the controlling process for a controlling terminal, then the SIGHUP signal is sent to each process in the terminal's foreground process group

Exit Handlers

- Some operations performed automatically on process termination
- programmer-supplied function that is registered at some point during the life of the process and is called during normal process termination via *exit()*
- Not called if a program calls *_exit()* directly or if the process is terminated abnormally by a signal

Registering exit handlers

- by using the **atexit()** function

```
#include <stdlib.h>

int atexit(void (*func) (void));   Returns 0 on success, or nonzero on error
```

- adds *func* to a list of functions that are called when the process terminates
  - *func* should be defined to take no arguments and return no value
  - possible to register multiple exit handlers; these functions are called in reverse order of registration
- suffers limitations
  - when called, an exit handler doesn’t know what status was passed to *exit()*
  - can’t specify an argument to the exit handler when it is called

Registering exit handlers

- by using the **on_exit()** function

```
#include <stdlib.h>

int on_exit(void (*func) (int, void *), void *arg);
                    Returns 0 on success, or nonzero on error
```

- *func* is a pointer to a function of the following type:

```
void func (int status, void *arg) {
        /* perform cleanup actions */
}
```

- As with **atexit()**, multiple exit handlers can be registered with **on_exit()**
  - are called in reverse order of their registration

procexec/exit_handlers.c

```
#define _BSD_SOURCE     /* Get on_exit() declaration from <stdlib.h> */
#include <stdlib.h>
#include "tlpi_hdr.h"

static void
atexitFunc1(void)
{
    printf("atexit function 1 called\n");
}

static void
atexitFunc2(void)
{
    printf("atexit function 2 called\n");
}

static void
onexitFunc(int exitStatus, void *arg)
{
    printf("on_exit function called: status=%d, arg=%ld\n",
            exitStatus, (long) arg);
}
```

```c
int
main(int argc, char *argv[])
{
    if (on_exit(onexitFunc, (void *) 10) != 0)
        fatal("on_exit 1");
    if (atexit(atexitFunc1) != 0)
        fatal("atexit 1");
    if (atexit(atexitFunc2) != 0)
        fatal("atexit 2");
    if (on_exit(onexitFunc, (void *) 20) != 0)
        fatal("on_exit 2");

    exit(2);
}
```

```
$ ./exit_handlers
on_exit function called: status=2, arg=20
atexit function 2 called
atexit function 1 called
on_exit function called: status=2, arg=10
```

Interaction Between **fork()**, ***stdio*** buffer, **_exit()**

procexec/fork_stdio_buf.c

```
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    printf("Hello world\n");
    write(STDOUT_FILENO, "Ciao\n", 5);

    if (fork() == -1)
        errExit("fork");

    /* Both child and parent continue execution here */

    exit(EXIT_SUCCESS);
}
```

Interaction Between **fork()**, ***stdio*** buffer, **_exit()**

```
$ ./fork_stdio_buf
Hello world
Ciao

$ ./fork_stdio_buf > a
$ cat a
Ciao
Hello world
Hello world
```

- Standard output to a terminal: line-buffered
  - newline-terminated string appears immediately
- Standard output to file: block-buffered
  - The string written by *printf()* is still in the parent's *stdio* buffer at the time of the *fork()*, and this is duplicated in the child
  - When the parent and child later call exit(), they both flush *stdio* buffer, resulting in duplicate output