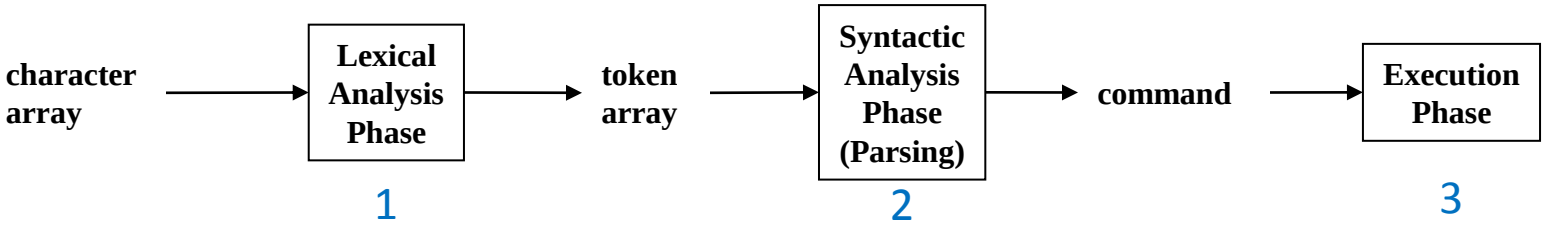
Ish: Process Management

Slides Originally Prepared by:
Wonho Kim

Agenda

- Last time
  - Assignment 6
    - Step 0, Step 1, Step 2
- Today
  - Assignment 6
    - Step3

Ish Design



```
echo one two three > myfile
```

```
echo
one
two
three
>
myfile
```

```
Command:
   Name:  echo
   Arg 0:  one
   Arg 1:  two
   Arg 2:  three
   Stdinredirect:  NULL
   Stdoutredirect:  myfile
```

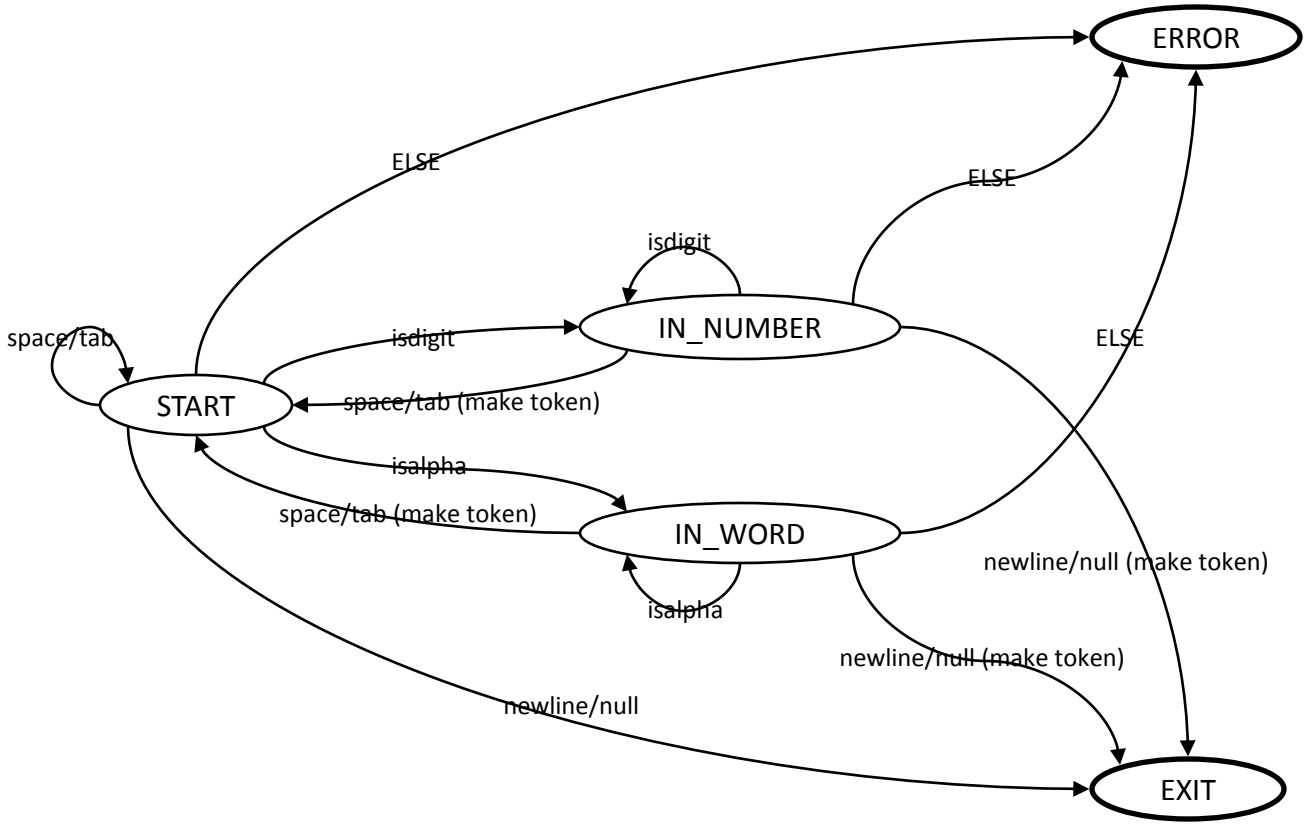
```
echo one "two three" > myfile
```

```
echo
one
two three
>
myfile
```

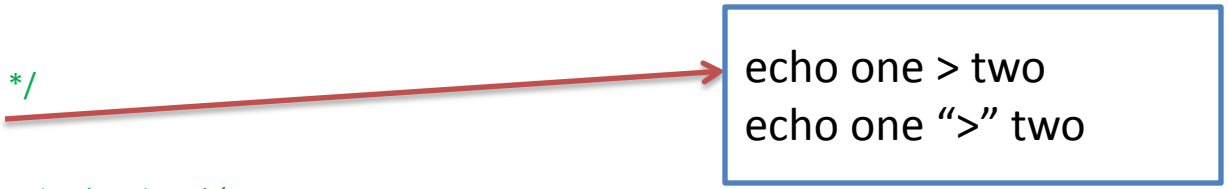
```
Command:
   Name:  echo
   Arg 0:  one
   Arg 1:  two three
   Stdinredirect:  NULL
   Stdoutredirect:  myfile
```

Ish Development Stages

- Stage 1: Lexical Analysis
  - Create a lexical analyzer
    - Input: character array
    - Output: token array
  - Recommendation #1
    - Design DFA (As with decomment program)
    - Test DFA enough before you move to the next stage
  - Recommendation #2
    - The lexical analyzer reads from a char array rather than a file
  - Use DFA implementation in your handouts


ERROR
ELSE
ELSE
isdigit
IN_NUMBER
space/tab
isdigit
ELSE
START
space/tab (make token)
isalpha
space/tab (make token)
IN_WORD
newline/null (make token)
isalpha
newline/null (make token)
newline/null
EXIT

```
/*---------------------------------------------------------------------*/
/* dfa.c                                                               */
/* Author: Bob Dondero                                                 */
/* Illustrate lexical analysis using a deterministic finite state      */
/* automaton (DFA)                                                     */
/*---------------------------------------------------------------------*/

#include "dynarray.h"
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

enum {MAX_LINE_SIZE = 1024};

enum {FALSE, TRUE};

enum TokenType {TOKEN_NUMBER, TOKEN_WORD};

enum LexState {STATE_START, STATE_IN_NUMBER, STATE_IN_WORD, STATE_ERROR, STATE_EXIT};

/* A Token is either a number or a word, expressed as a string. */
struct Token
{
  /* The type of the token. */
  enum TokenType eType;

  /* The string which is the token's value. */
  char *pcValue;
};
```



Step3: Executable Binary commands

- We should call some new system calls
  - fork()
  - exec()
  - wait()

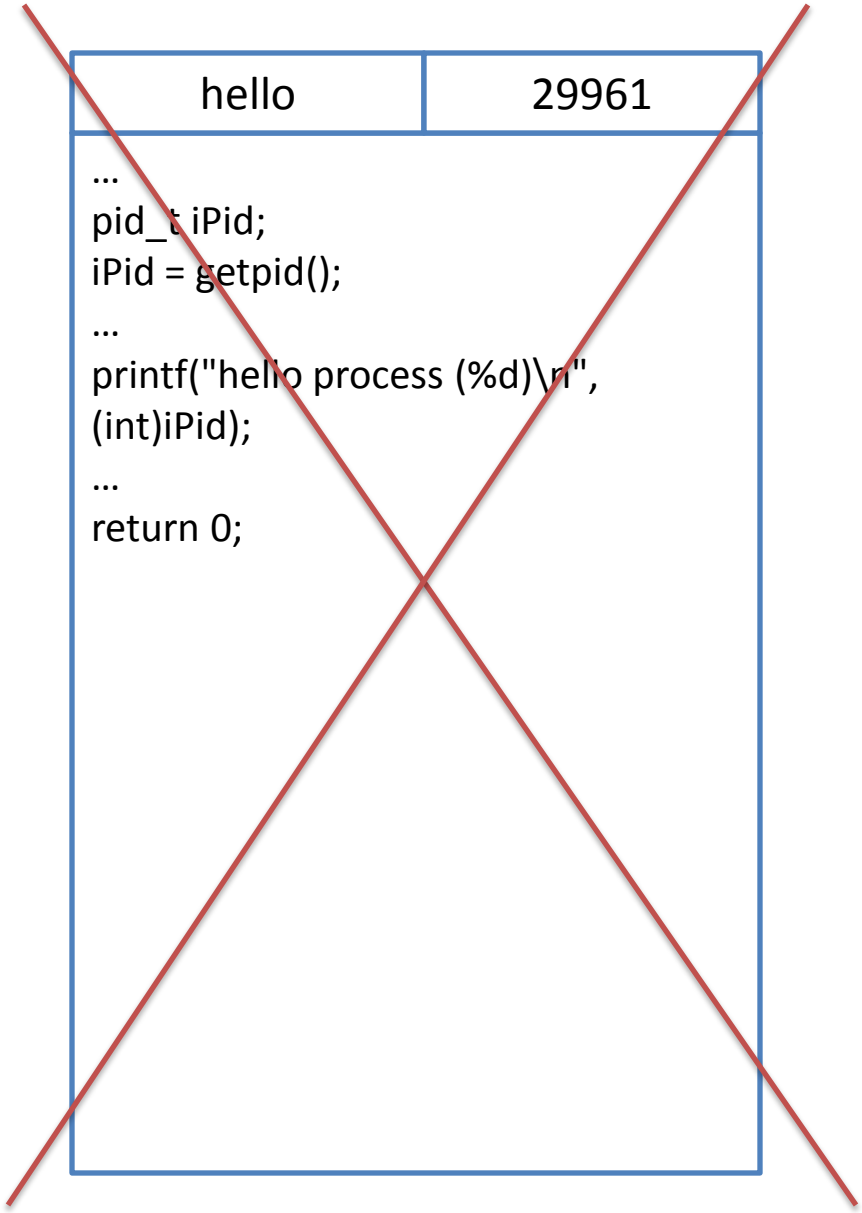
pid

- A running application has pid (Process ID).
- Let’s look at ‘hello.c’

hello	29961

```
…
pid_t iPid;
iPid = getpid();
…
printf("hello process (%d)\n",
(int)iPid);
…
return 0;
```

```
/* Sample executions:

$ hello
hello process (29961)

*/
```




```
/* Sample executions:

$ hello
hello process (29961)

*/
```

hello	29963

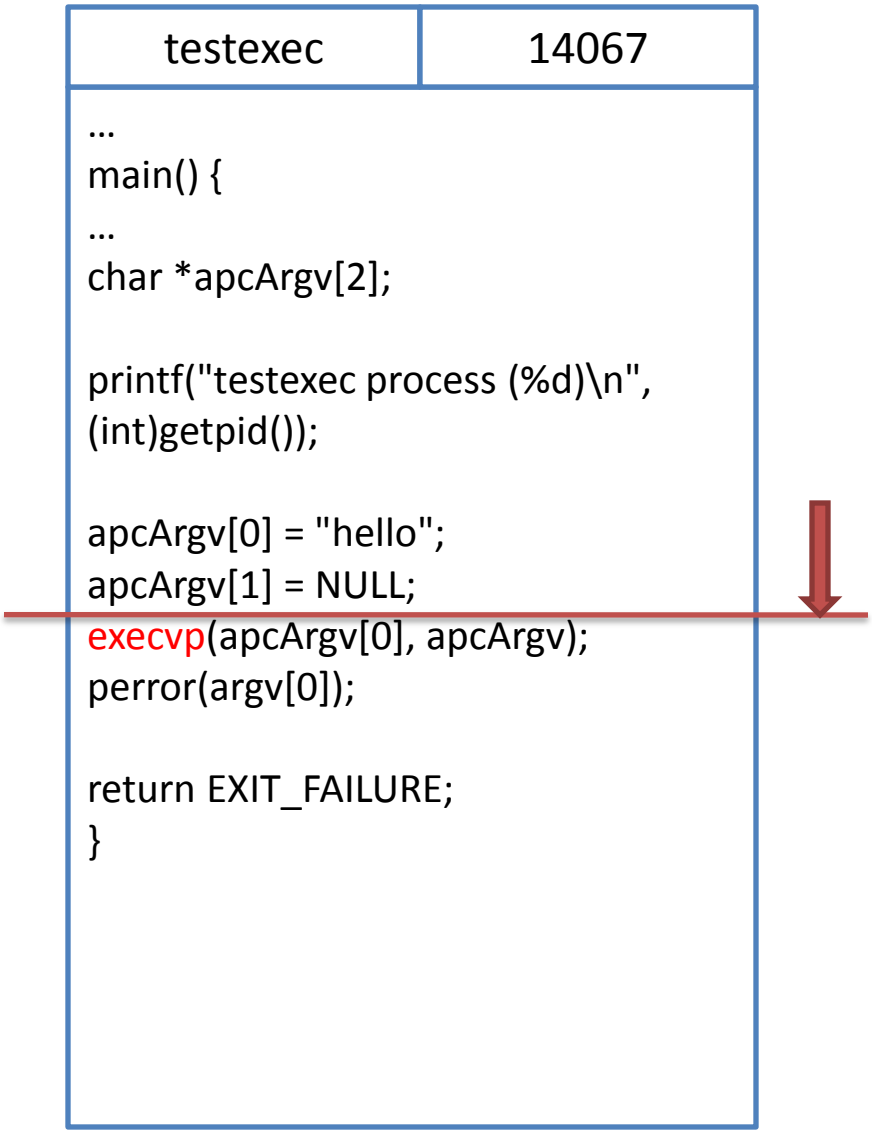
```
…
pid_t iPid;
iPid = getpid();
…
printf("hello process (%d)\n",
(int)iPid);
…
return 0;
```

```
/* Sample executions:

$ hello
hello process (29961)

$ hello
hello process (29963)

*/
```

- Let’s look at ‘testexec.c’

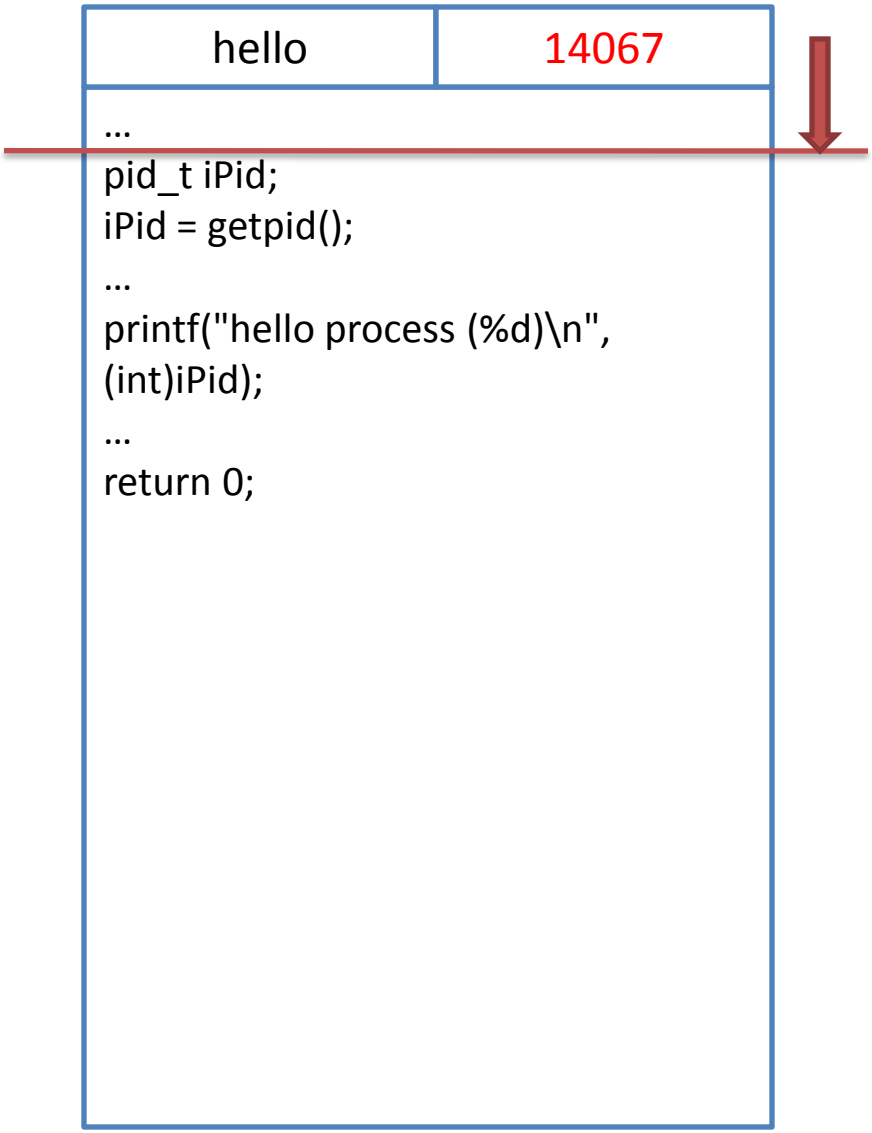
testexec	14067

```
…
main() {
…
char *apcArgv[2];

printf("testexec process (%d)\n",
(int)getpid());

apcArgv[0] = "hello";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);

return EXIT_FAILURE;
}
```



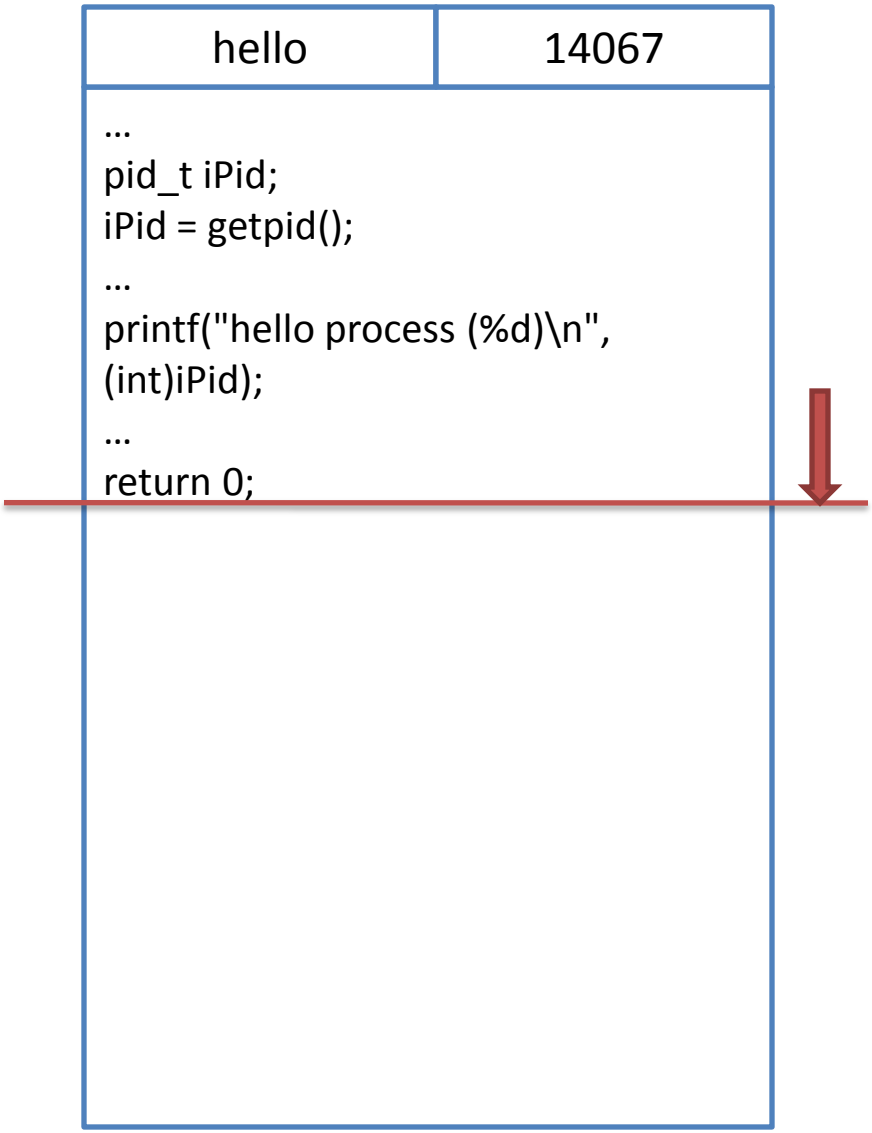
/* Sample executions:

$ testexec
testexec process (14067)

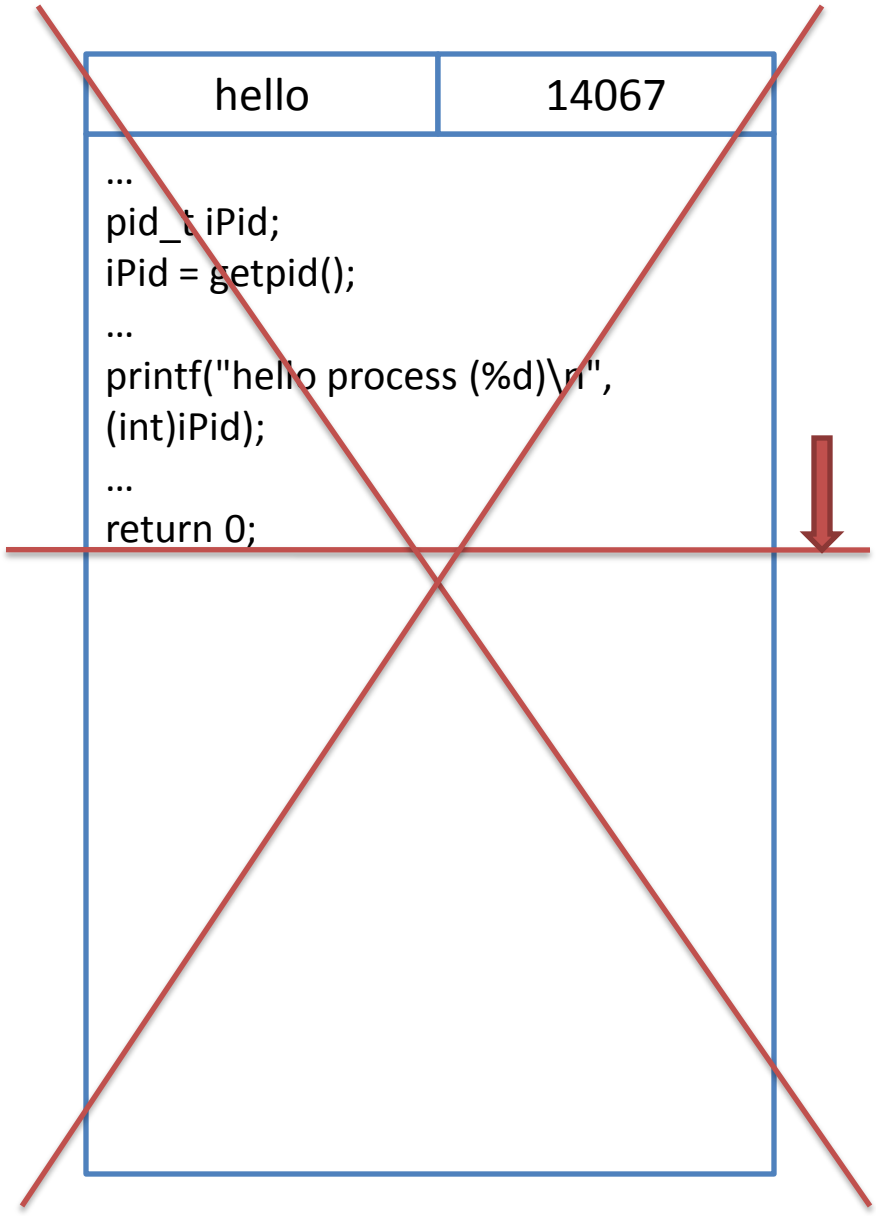


/* Sample executions:

$ testexec
testexec process (14067)




```
/* Sample executions:

$ testexec
testexec process (14067)
hello process (14067)
```

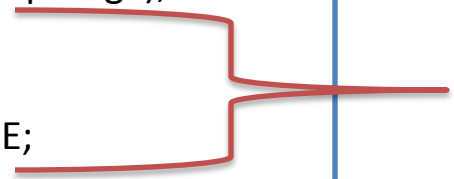


/* Sample executions:

$ testexec
testexec process (14067)
hello process (14067)

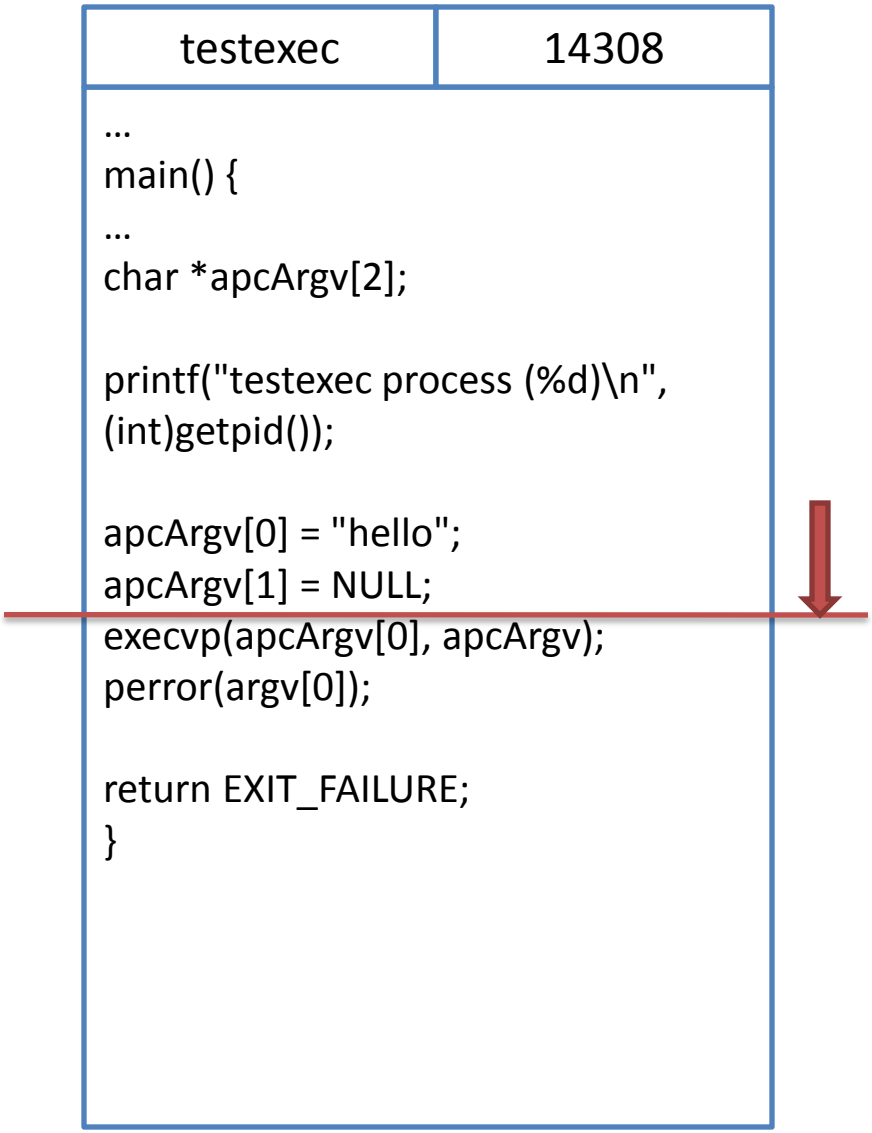
testexec	14067

```
…
main() {
…
char *apcArgv[2];

printf("testexec process (%d)\n",
(int)getpid());

apcArgv[0] = "hello";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);

return EXIT_FAILURE;
}
```



Should not be reachable
if execvp() succeeds

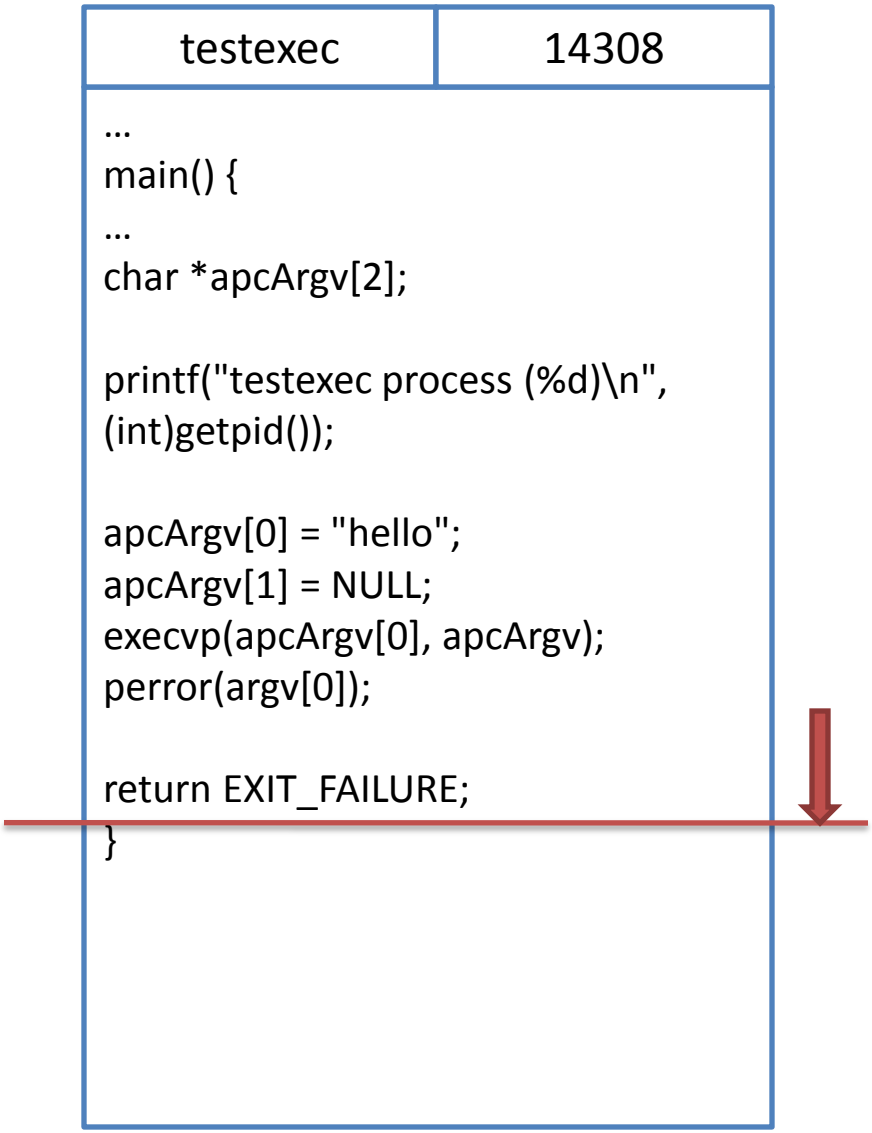
```
/* Sample executions:

$ testexec
testexec process (14067)
hello process (14067)
```



/* Sample executions:

$ mv hello nothello

$ testexec
testexec process (14308)



/* Sample executions:

$ mv hello nothello

$ testexec
testexec process (14308)
testexec: No such file or directory

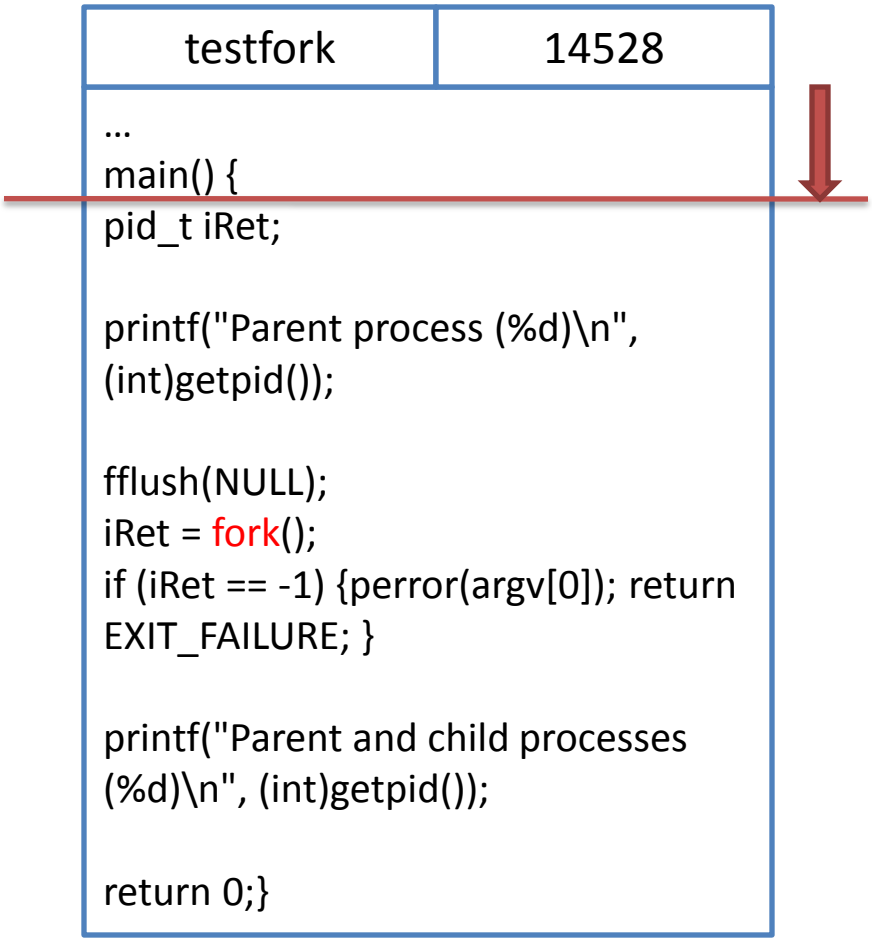
perror()

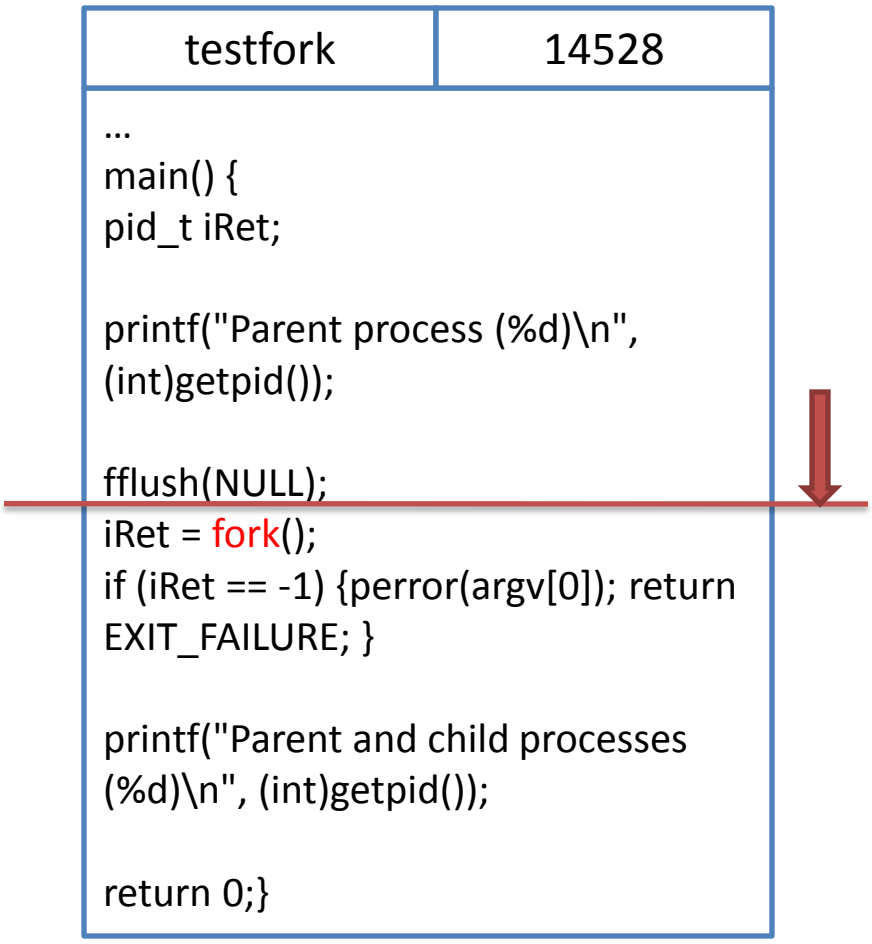
- Many library functions (including execvp())
  - return -1 when there's an error
  - (and) set a global variable 'errno' to an error code
- perror()
  - Prints readable error messages for current errno value
  - ex) perror("hello") => hello: …error message…
- Alternative
  - fprintf(stderr, "%s: %s\n", argv[0], strerror(errno));

- There are 6 exec system calls
  - see the table in your handouts
- You will find execvp() is the most appropriate for ish assignment

System Call	Will Search PATH (p)?	Accepts Vector (v)?	Accepts Environment (e)?
`int execl (const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/);`	No	No	No
`int execv (const char *path, char *const argv[]);`	No	Yes	No
`int execle(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/, char *const envp[]);`	No	No	Yes
`int execve(const char *path, char *const argv[], char *const envp[]);`	No	Yes	Yes
`int execlp(const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/);`	Yes	No	No
`int execvp(const char *file, char *const argv[]);`	Yes	Yes	No

- Let’s move to ‘fork()’
- See ‘testfork.c’


testfork
14528
…
main() {
pid_t iRet;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Parent and child processes
(%d)\n", (int)getpid());
return 0;}




```
/* Sample executions:

$ testfork
Parent process (14528)
```

testfork	14528

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

testfork	14529

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

/* Sample executions:

$ testfork
Parent process (14528)

Now, both processes will run concurrently!!

In an old machine (single CPU), CPU switches between multiple processes. (physically, serialized)

Modern machines (multi-core) can run multiple processes concurrently (real concurrency)

testfork	14528

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

testfork	14529

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

/* Sample executions:

$ testfork
Parent process (14528)

We don't know in advance which process will run first

Let's assume that child process runs first

testfork	14528

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

testfork	14529

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

```
/* Sample executions:

$ testfork
Parent process (14528)
Parent and child processes (14529)
```

testfork	14528

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

testfork	14529

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

```
/* Sample executions:

$ testfork
Parent process (14528)
Parent and child processes (14529)
```

testfork	14528

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

testfork	14529

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

```
/* Sample executions:

$ testfork
Parent process (14528)
Parent and child processes (14529)
Parent and child processes (14528)
```

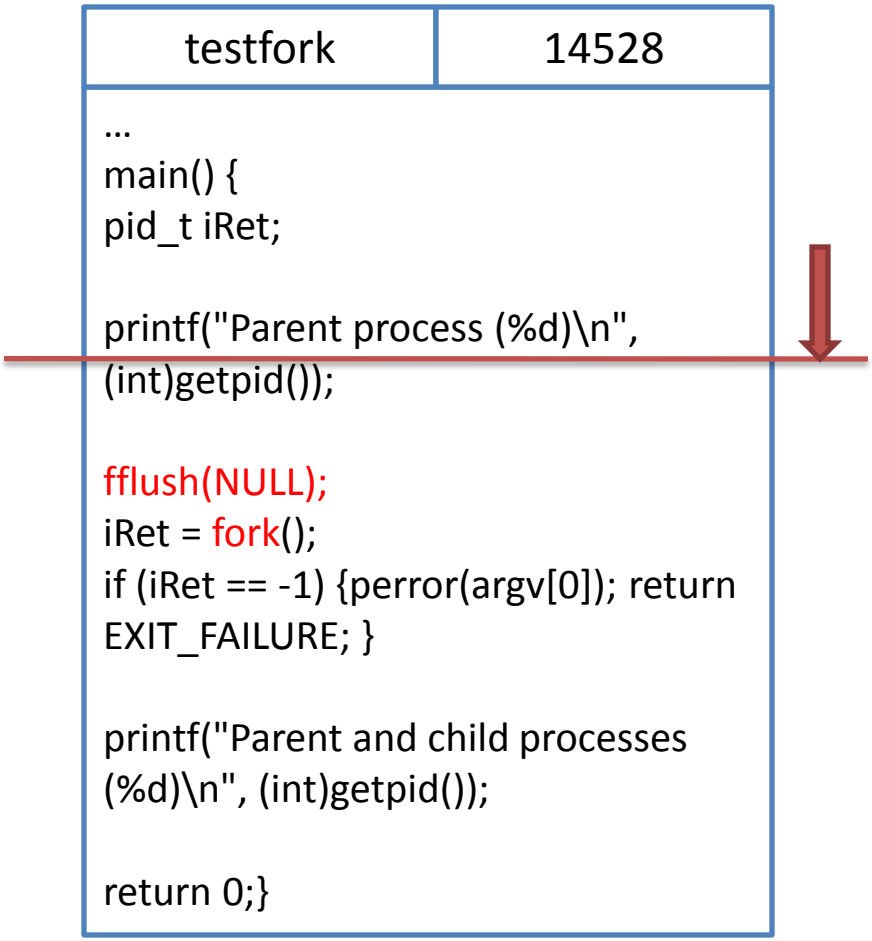
testfork	14528

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

testfork	14529

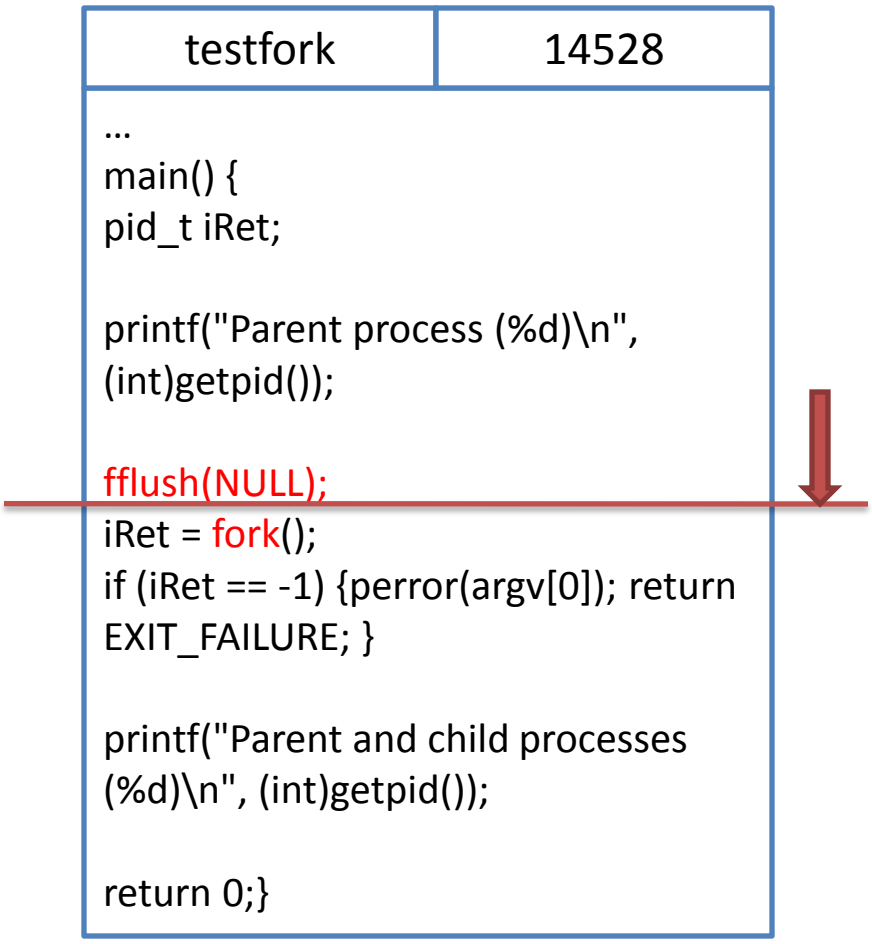
```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

```
/* Sample executions:

$ testfork
Parent process (14528)
Parent and child processes (14529)
Parent and child processes (14528)
```



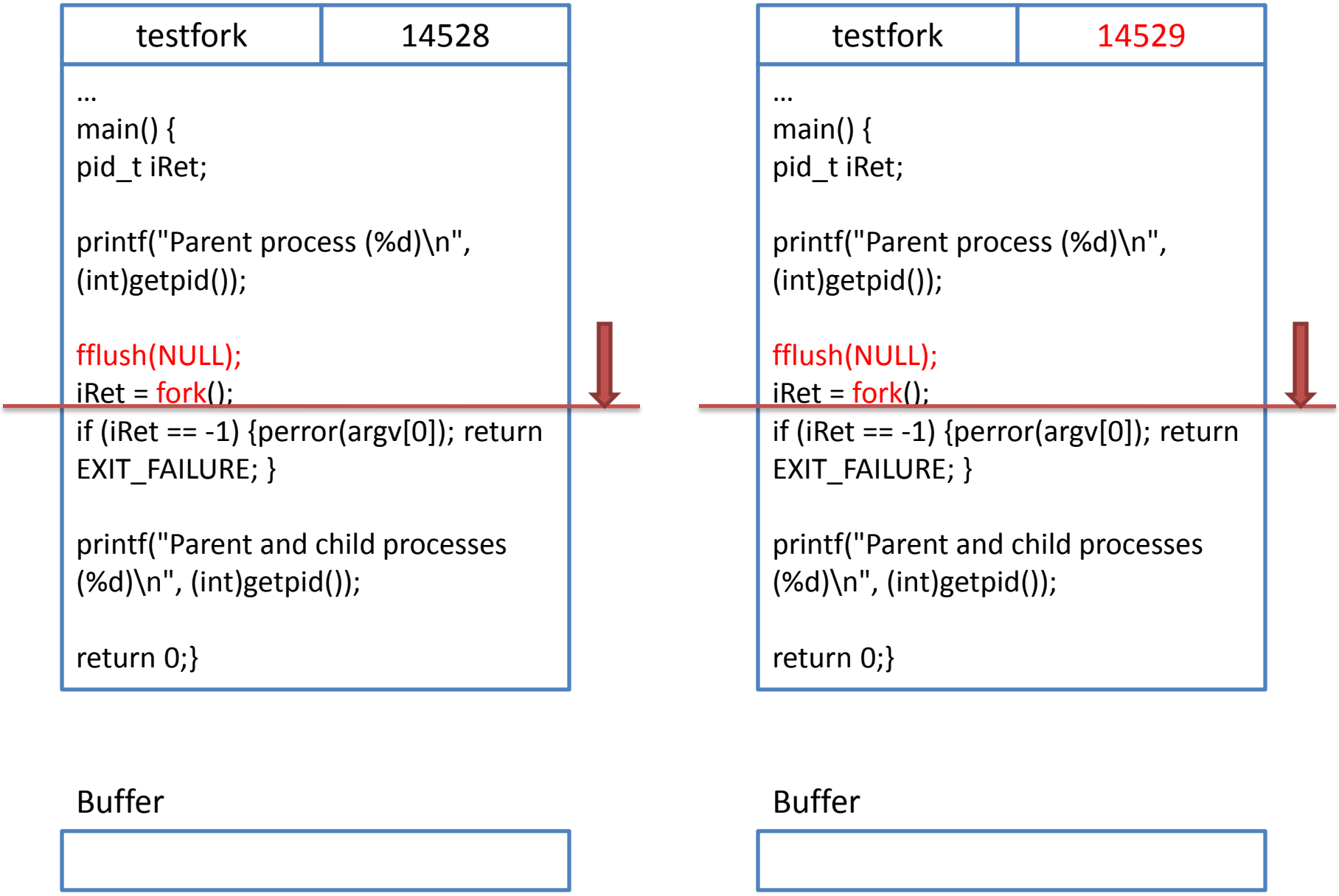
Buffer

“Parent process 14528”

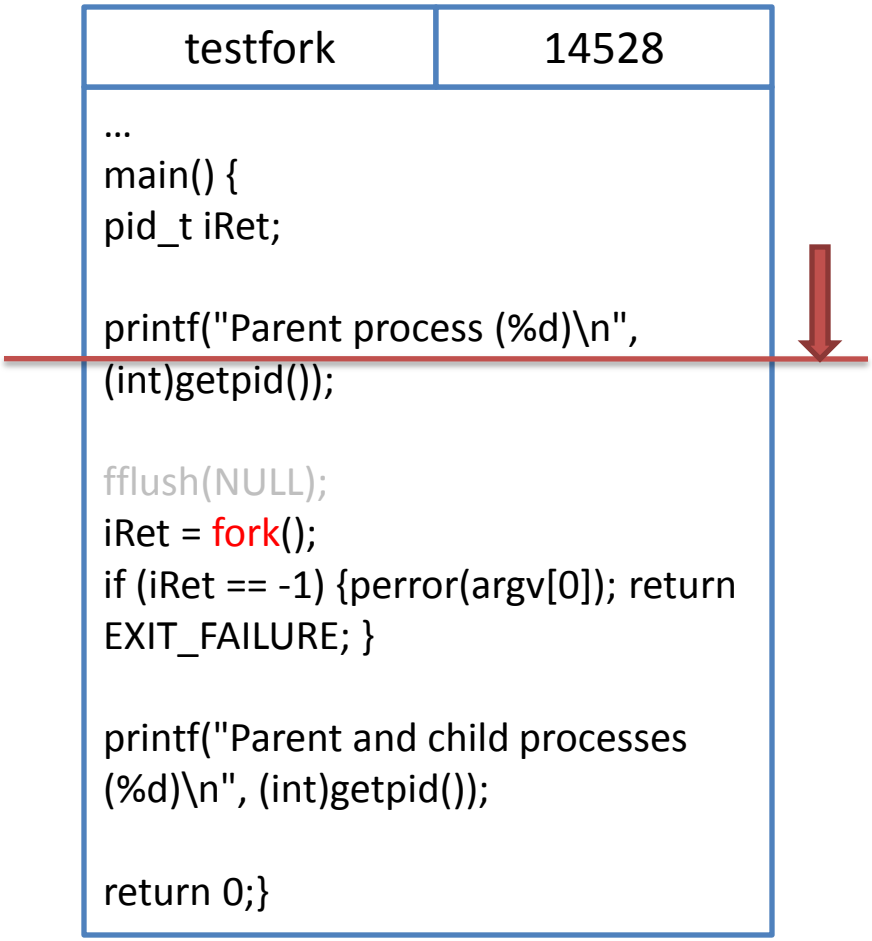


Buffer

```
/* Sample executions:

$ testfork
Parent process (14528)
```



What if we don't have fflush() ??



Buffer

“Parent process 14528”

testfork	14528

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

Buffer

“Parent process 14528”

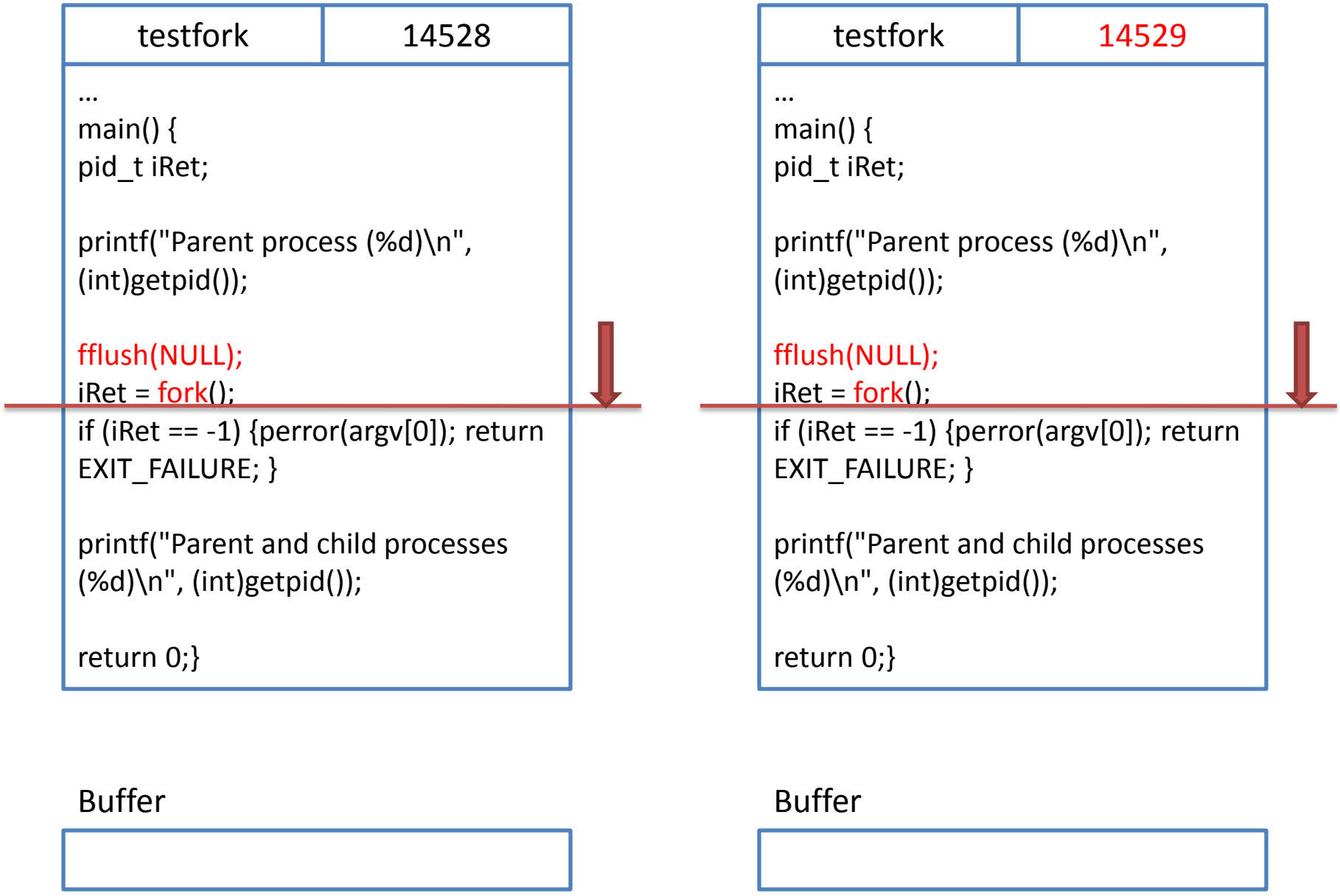
testfork	14529

```
…
main() {
pid_t iRet;

printf("Parent process (%d)\n",
(int)getpid());

fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Parent and child processes
(%d)\n", (int)getpid());

return 0;}
```

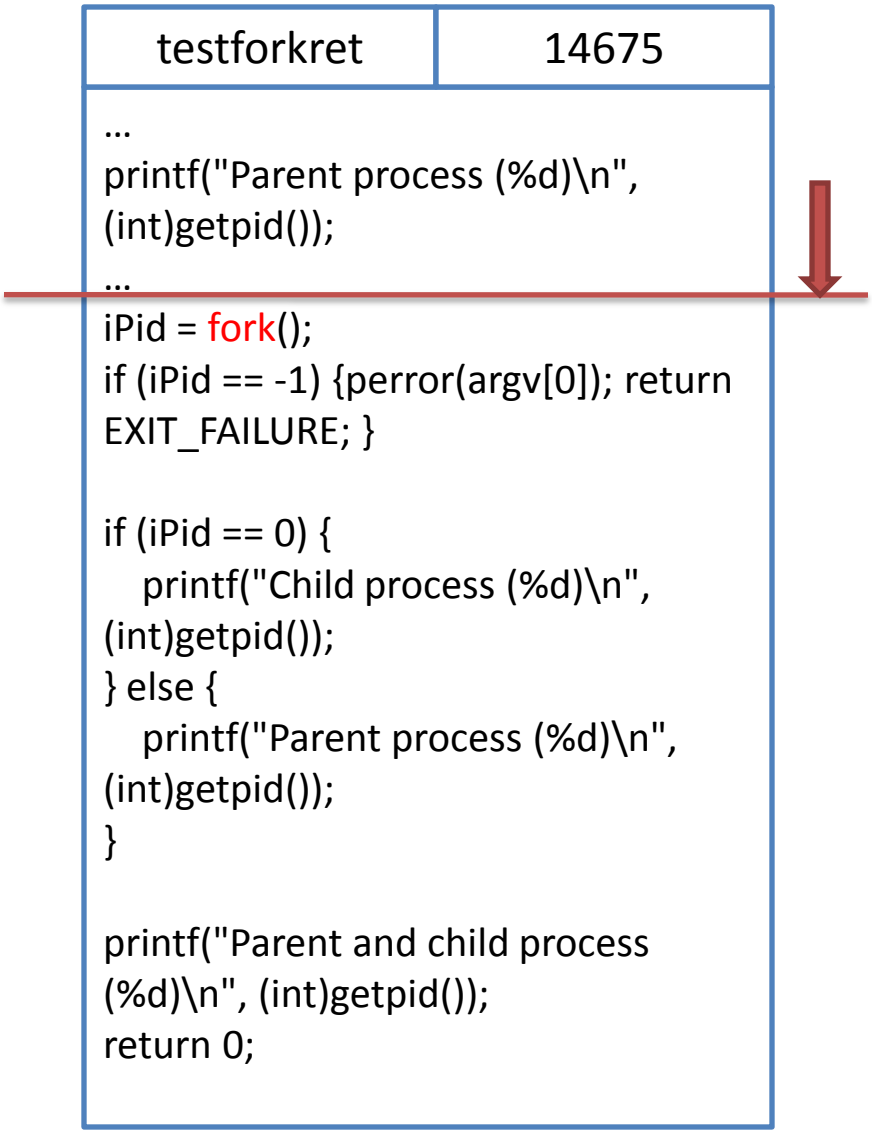
Buffer

“Parent process 14528”

- How can we know if we are in parent process or child process ??


testfork
14528
…
main() {
pid_t iRet;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Parent and child processes
(%d)\n", (int)getpid());
return 0;}
Buffer
testfork
14529
…
main() {
pid_t iRet;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iRet = fork();
if (iRet == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Parent and child processes
(%d)\n", (int)getpid());
return 0;}
Buffer

- How can we know if we are in parent process or child process ??
  - => By using fork() return value
- Let's look at 'testforkret.c'


testforkret
14675
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0) {
printf("Child process (%d)\n",
(int)getpid());
} else {
printf("Parent process (%d)\n",
(int)getpid());
}
printf("Parent and child process
(%d)\n", (int)getpid());
return 0;

testforkret	14675

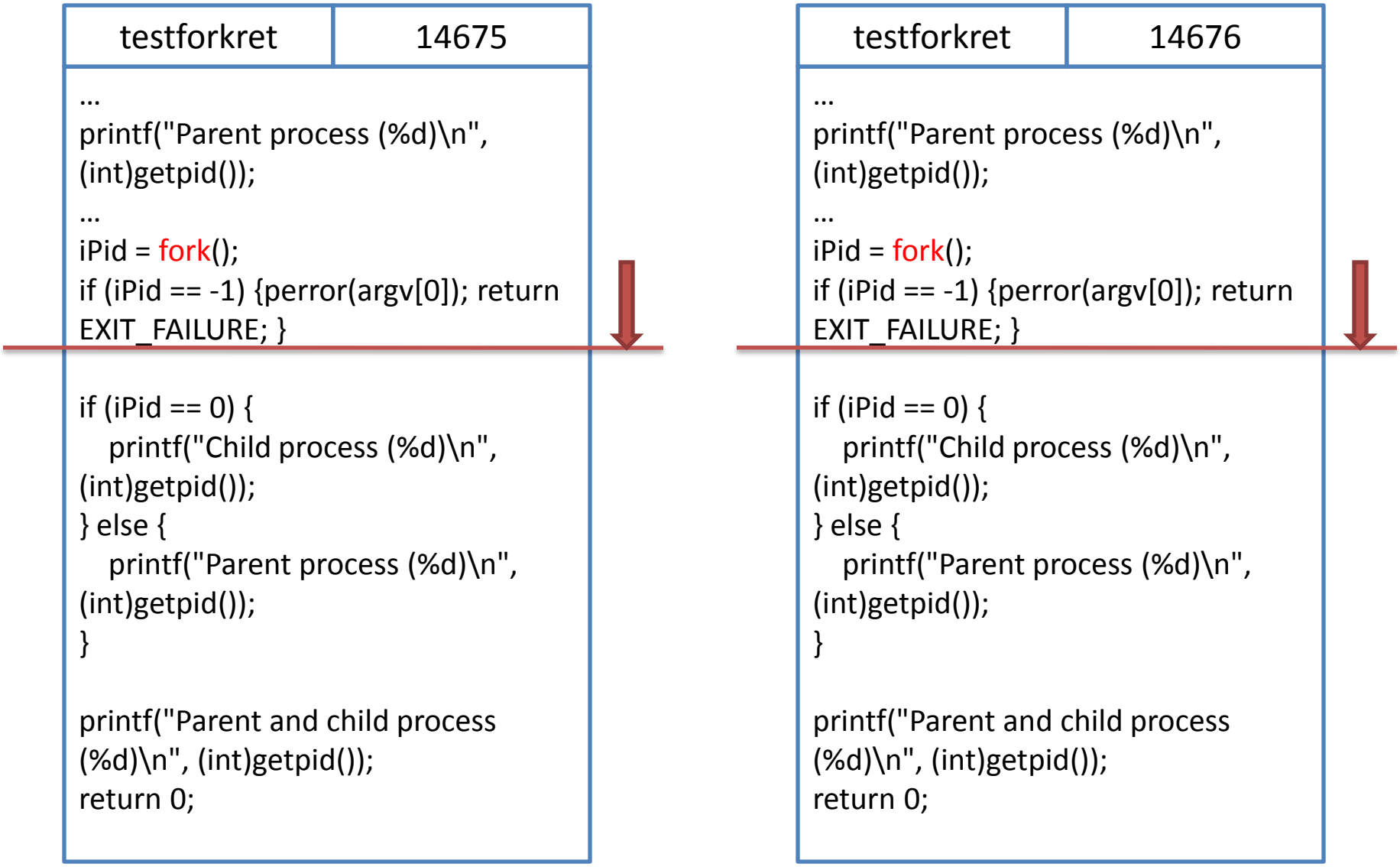
```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

testforkret	14676

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

fork() returns ‘child process pid’ in parent process

fork() returns 0 in child process



fork() returns ‘child process pid’ in parent process

fork() returns 0 in child process

testforkret	14675

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

testforkret	14676

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

fork() returns ‘child process pid’ in parent process
fork() returns 0 in child process

testforkret	14675

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

testforkret	14676

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

fork() returns ‘child process pid’ in parent process
fork() returns 0 in child process

testforkret	14675

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

testforkret	14676

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

fork() returns ‘child process pid’ in parent process
fork() returns 0 in child process

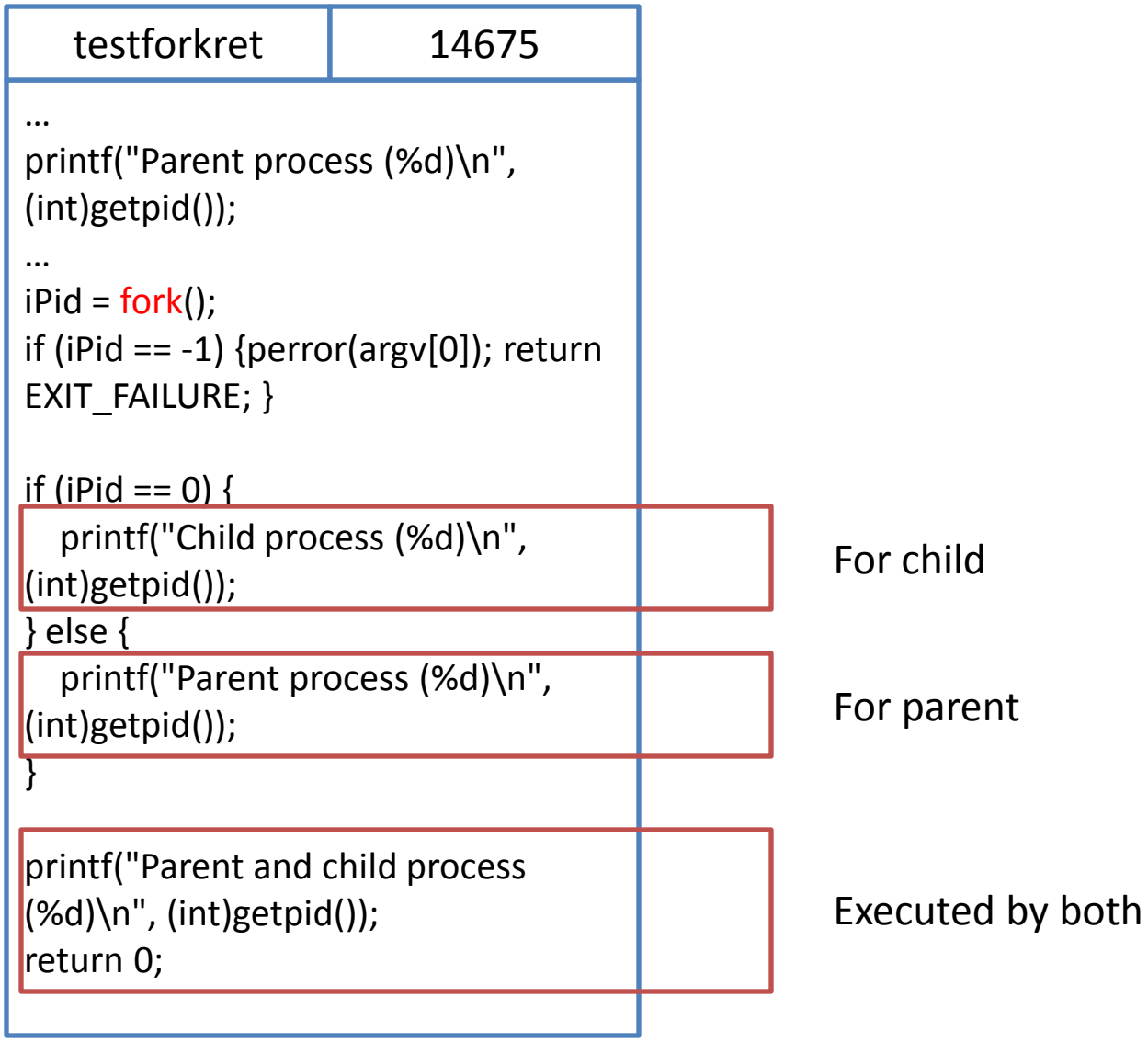
testforkret	14675

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

testforkret	14676

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

fork() returns ‘child process pid’ in parent process
fork() returns 0 in child process


testforkret
14675
…
printf("Parent process (%d)\n", (int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return EXIT_FAILURE; }
if (iPid == 0) {
printf("Child process (%d)\n", (int)getpid());
} else {
printf("Parent process (%d)\n", (int)getpid());
}
printf("Parent and child process (%d)\n", (int)getpid());
return 0;
For child
For parent
Executed by both

testforkret	14675

```
…
printf("Parent process (%d)\n",
(int)getpid());
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0) {
   printf("Child process (%d)\n",
(int)getpid());
} else {
   printf("Parent process (%d)\n",
(int)getpid());
}

printf("Parent and child process
(%d)\n", (int)getpid());
return 0;
```

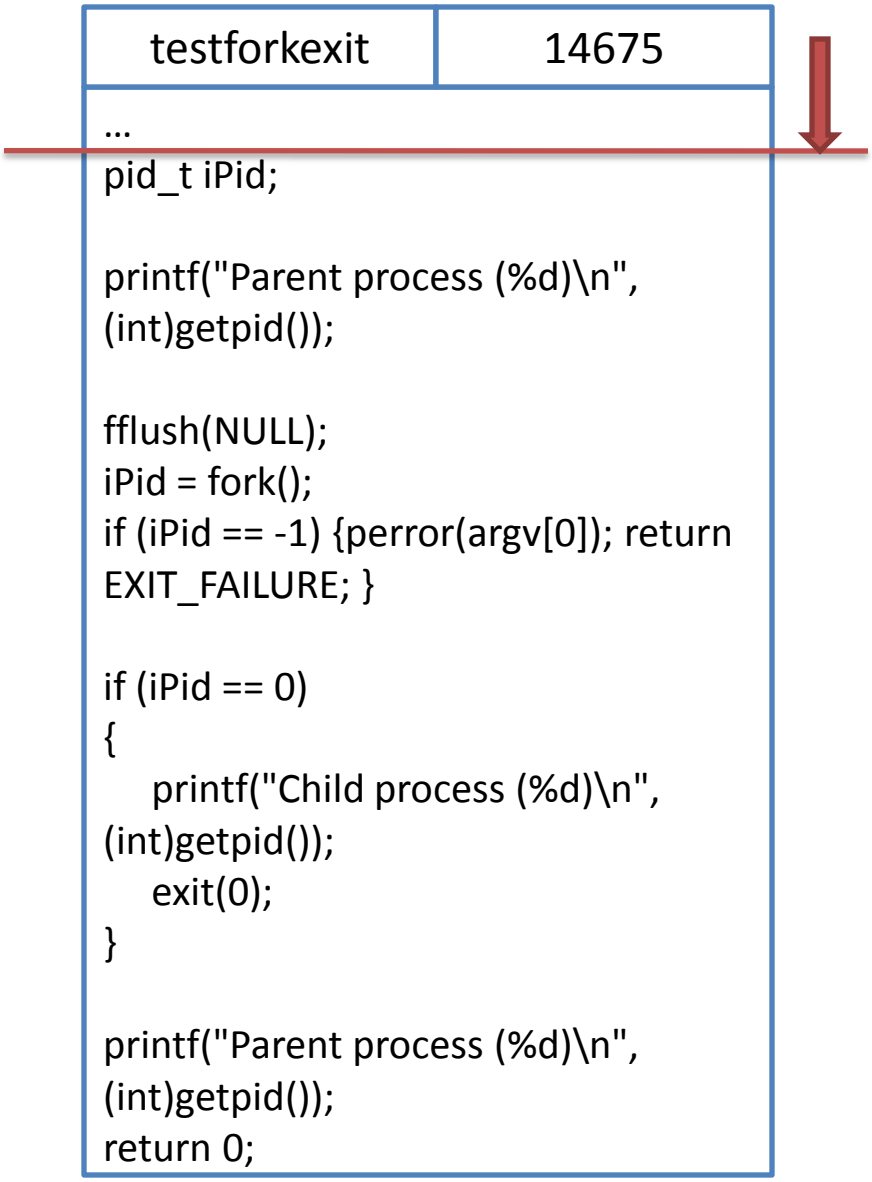
For child

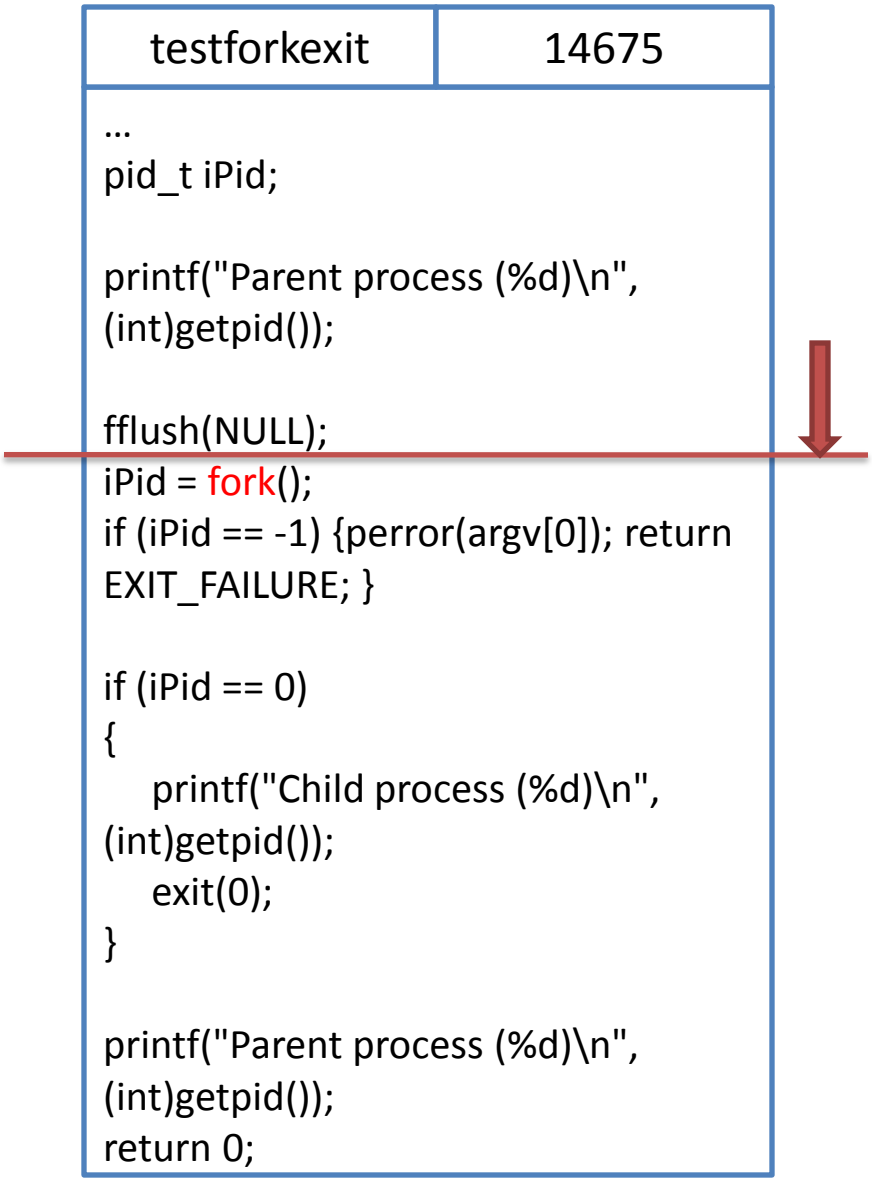
For parent

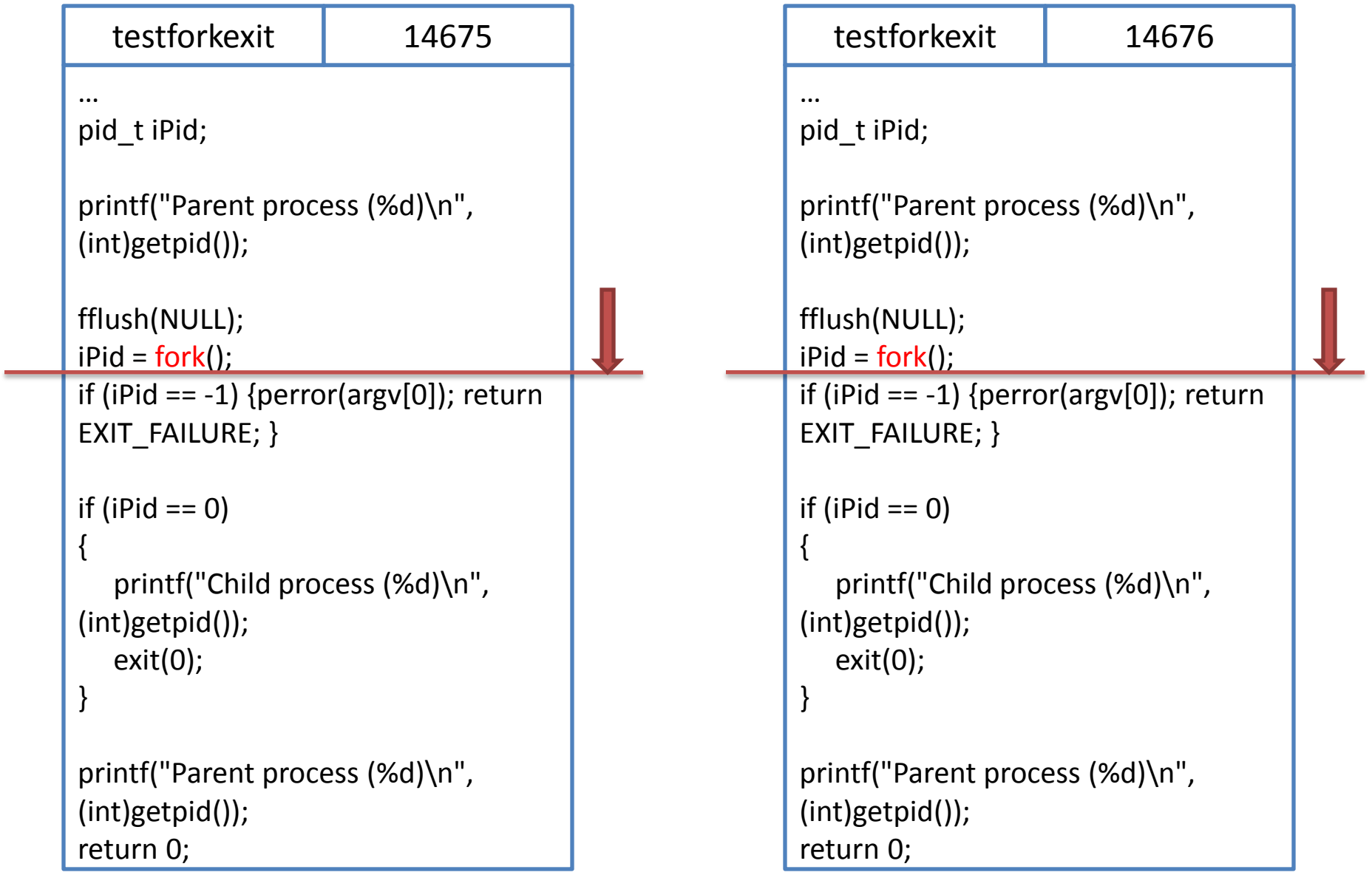
Executed by both

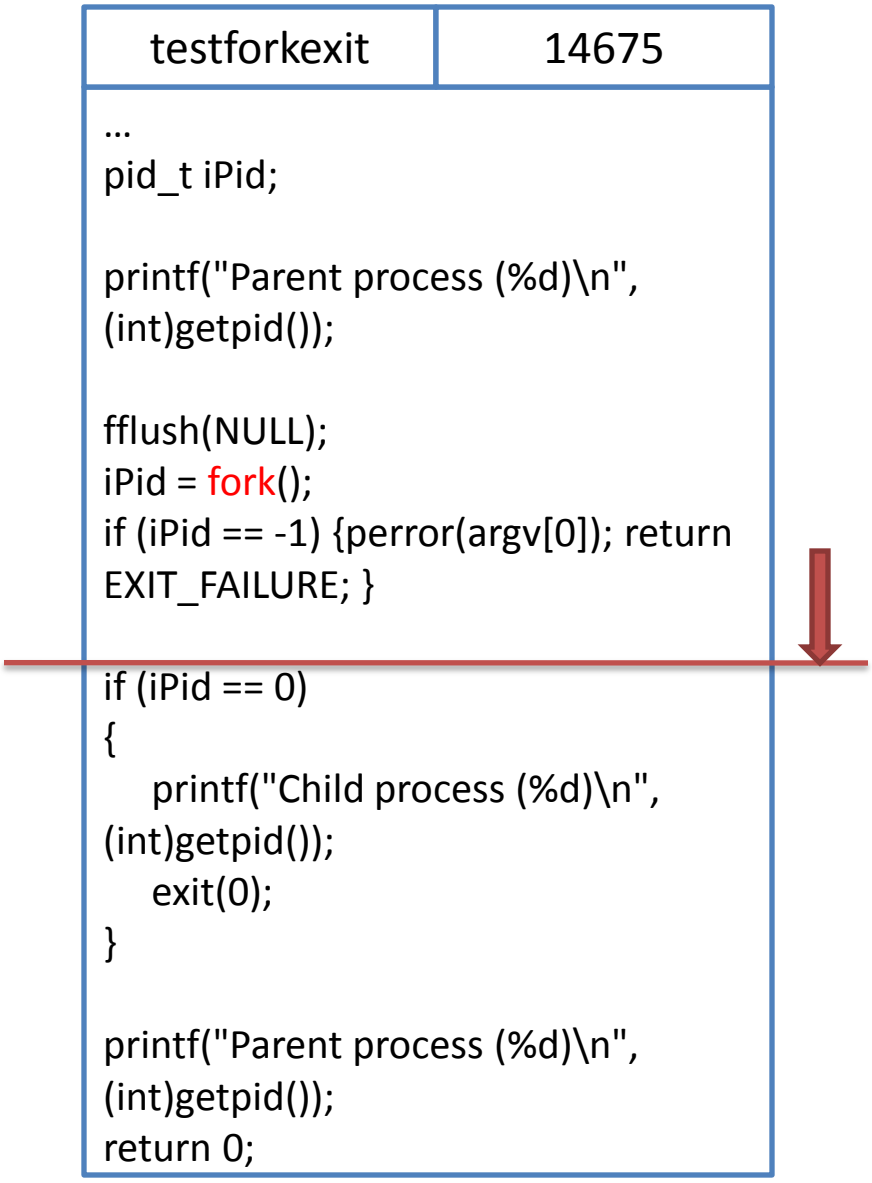
How can we isolate the code for child into the if-stmt ??

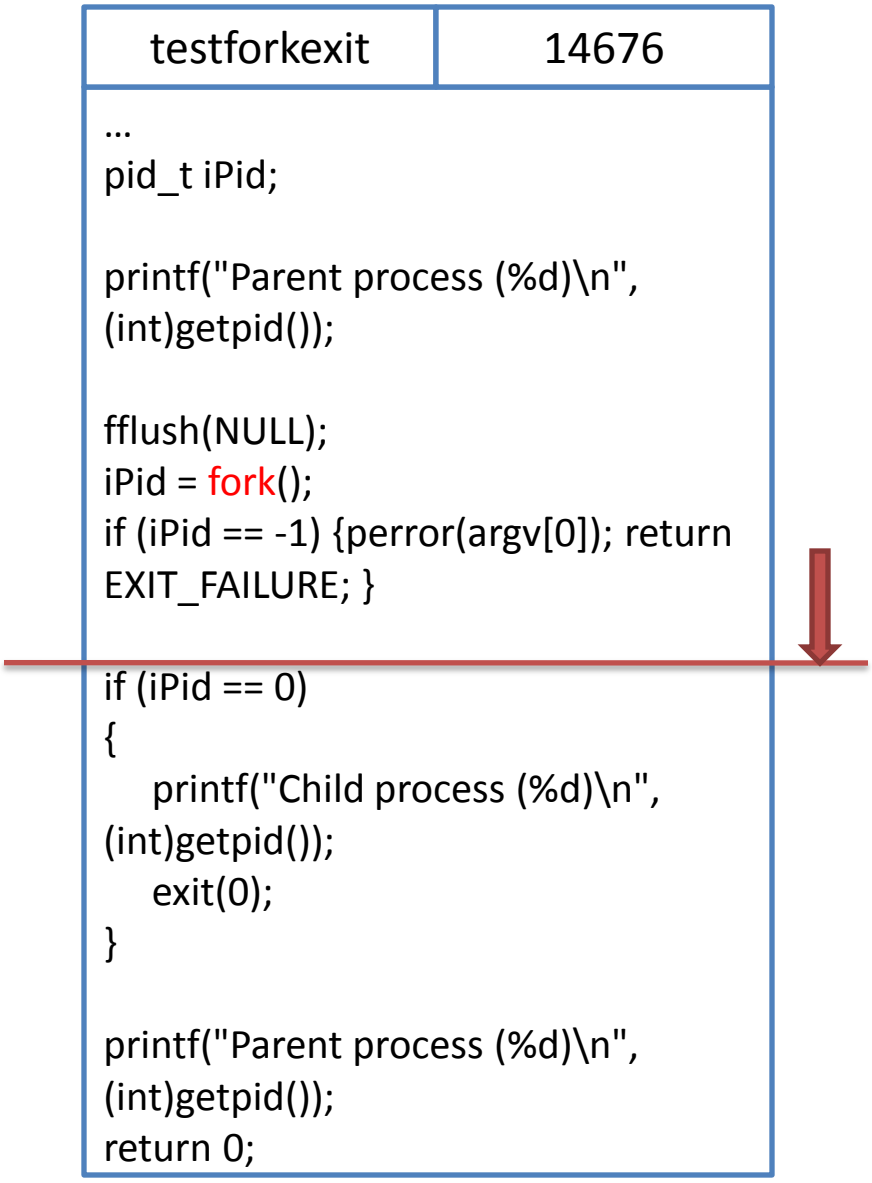
- Let’s look at ‘testfork exit’

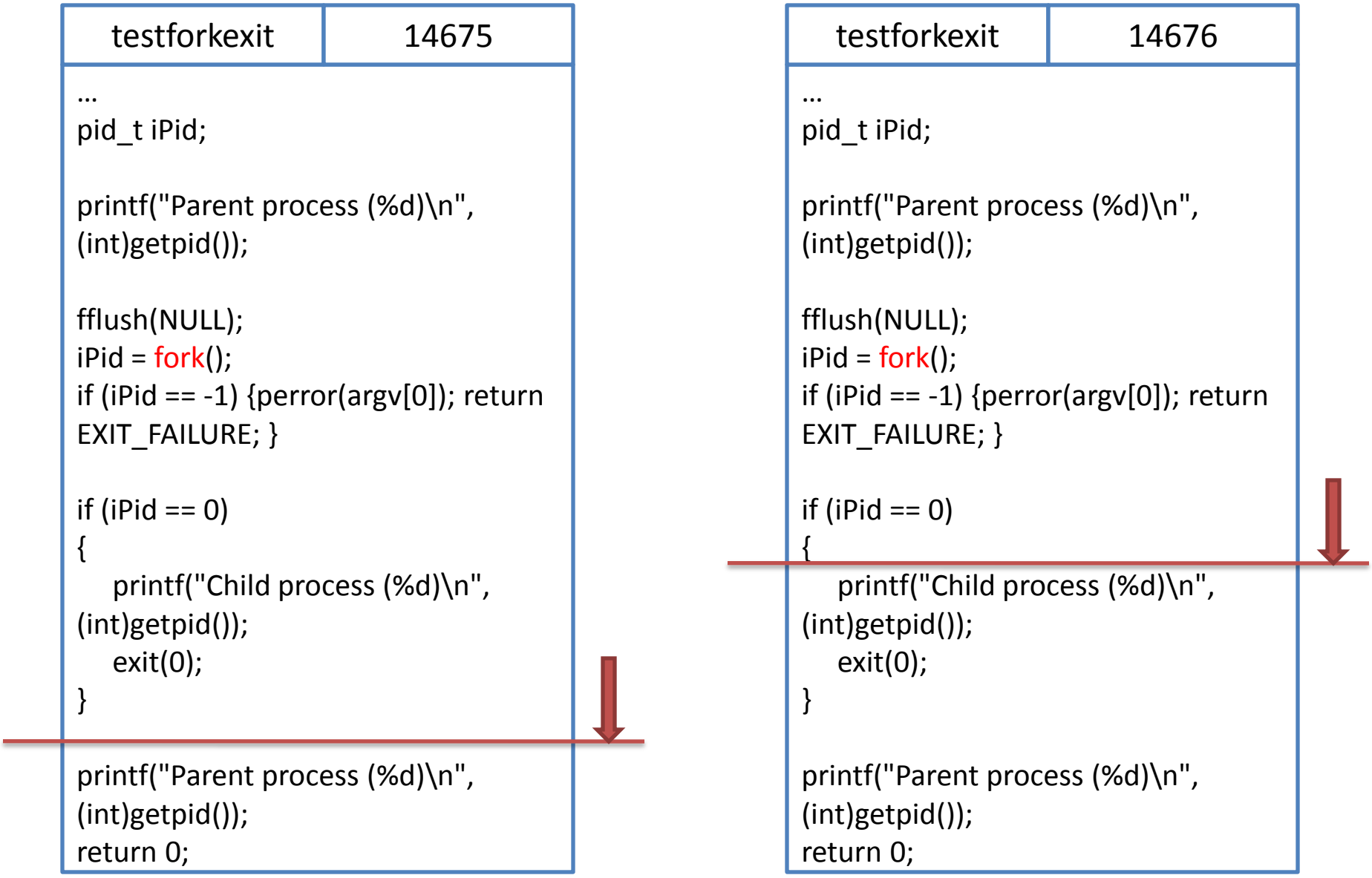

testforkexit
14675
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;

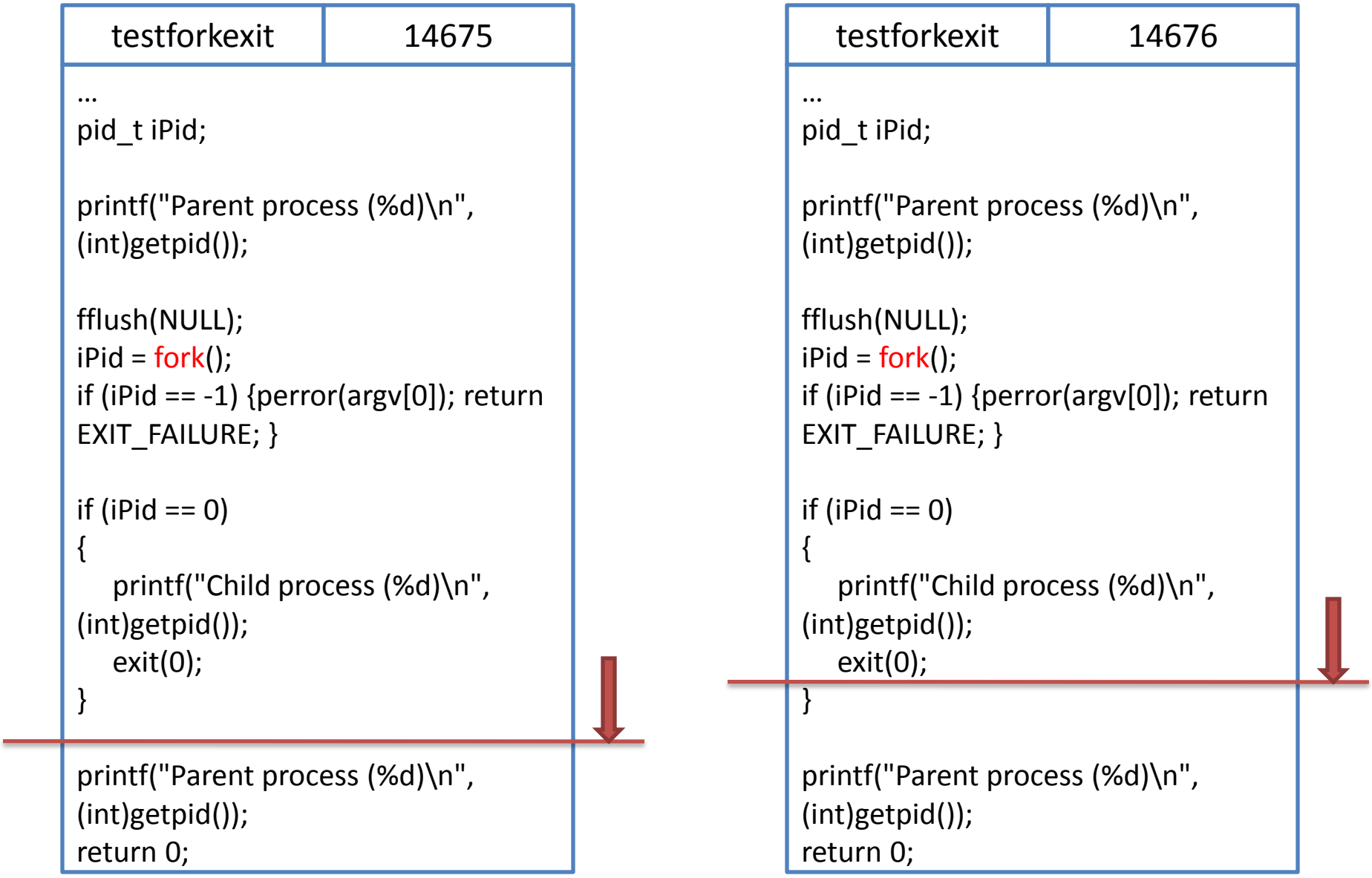

testforkexit
14675
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;

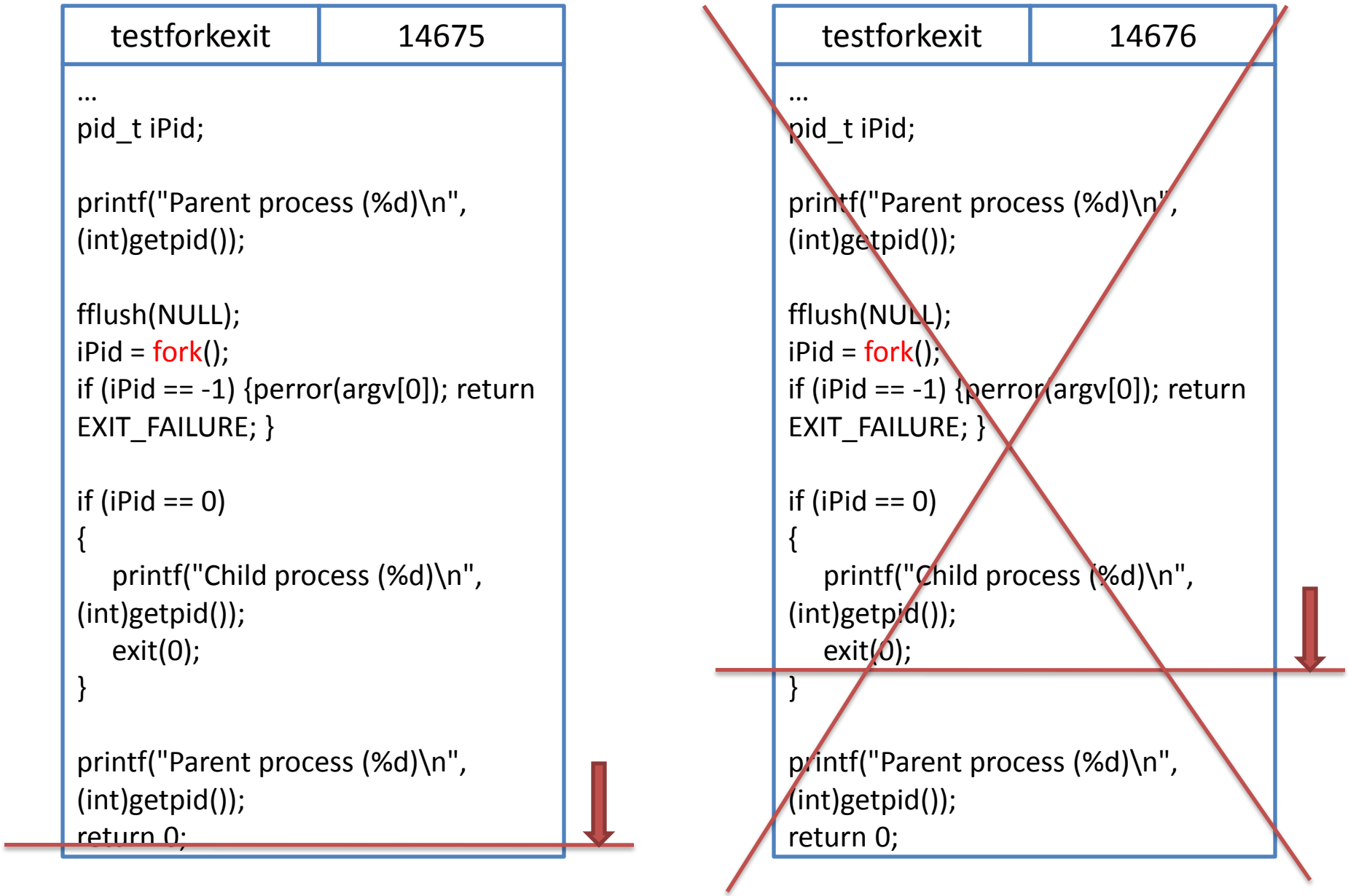

testforkexit
14675
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;
testforkexit
14676
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;


testforkexit
14675
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;


testforkexit
14676
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;


testforkexit
14675
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;
testforkexit
14676
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;


testforkexit
14675
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;
testforkexit
14676
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;


testforkexit
14675
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;
testforkexit
14676
…
pid_t iPid;
printf("Parent process (%d)\n",
(int)getpid());
fflush(NULL);
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
if (iPid == 0)
{
printf("Child process (%d)\n",
(int)getpid());
exit(0);
}
printf("Parent process (%d)\n",
(int)getpid());
return 0;

- Let’s look at ‘testforkloop.c’

testforkloop	14675

```
pid_t iPid;
int i = 0;
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0)
{
    for (i = 0; i < 1000; i++)
        printf("Child process (%d):
%d\n", (int)getpid(), i);

  exit(0);
}

for (i = 0; i < 1000; i++)
     printf("Parent process (%d):
%d\n", (int)getpid(), i);

return 0;
```

Let’s assume that we have a single CPU.
The CPU will switch back and forth between two processes

testforkloop	14675

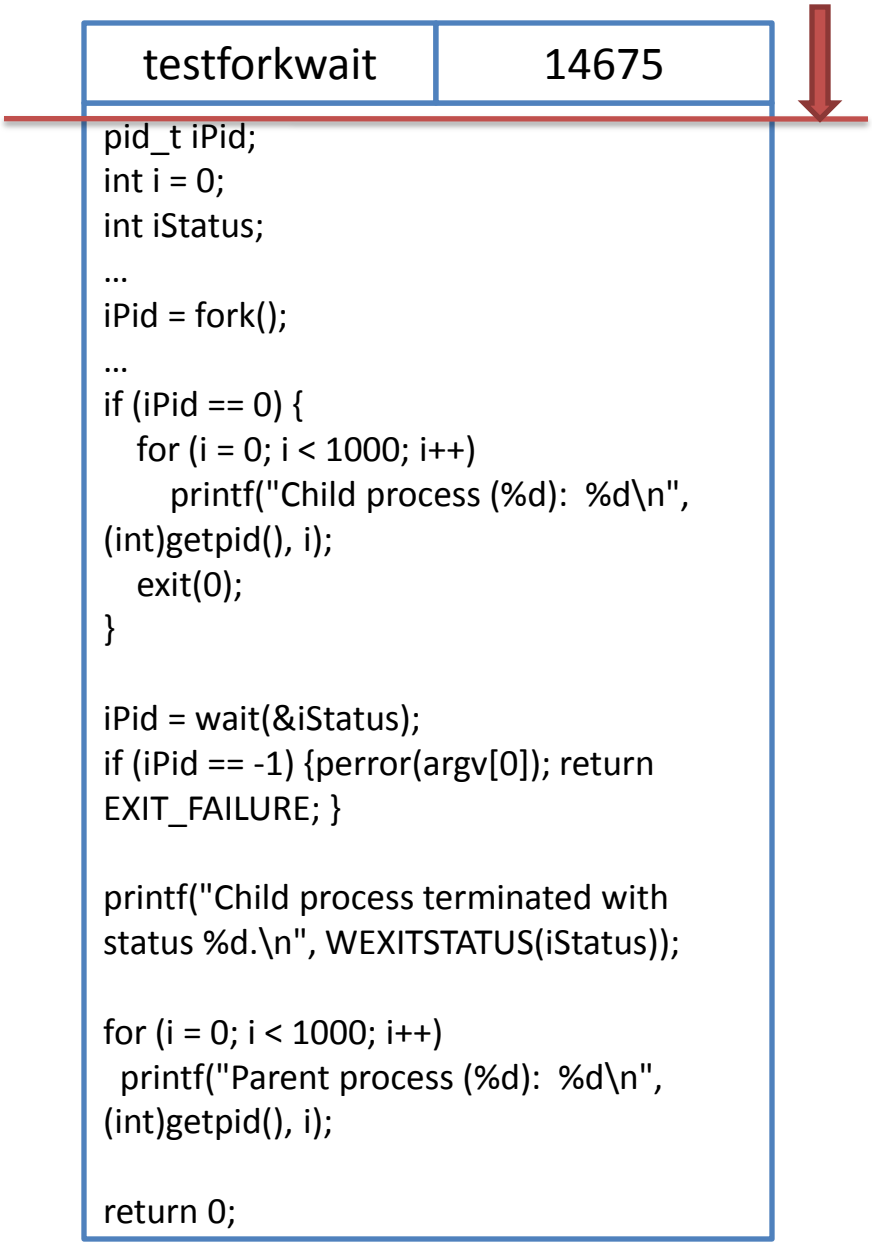
```
pid_t iPid;
int i = 0;
…
iPid = fork();
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

if (iPid == 0)
{
    for (i = 0; i < 1000; i++)
        printf("Child process (%d):
%d\n", (int)getpid(), i);

  exit(0);
}

for (i = 0; i < 1000; i++)
     printf("Parent process (%d):
%d\n", (int)getpid(), i);

return 0;
```

```
/* Sample execution:

$ testforkloop
Parent process (27915)
Child process (27916):  0
Child process (27916):  1
Child process (27916):  2
...
Child process (27916):  486
Child process (27916):  487
Child process (27916):  488
Parent process (27915):  0
Parent process (27915):  1
Parent process (27915):  2
...
Parent process (27915):  997
Parent process (27915):  998
Parent process (27915):  999
Child process (27916):  489
Child process (27916):  490
Child process (27916):  491
...
Child process (27916):  997
Child process (27916):  998
Child process (27916):  999
```

Let’s assume that we have a single CPU.
The CPU will switch back and forth between two processes

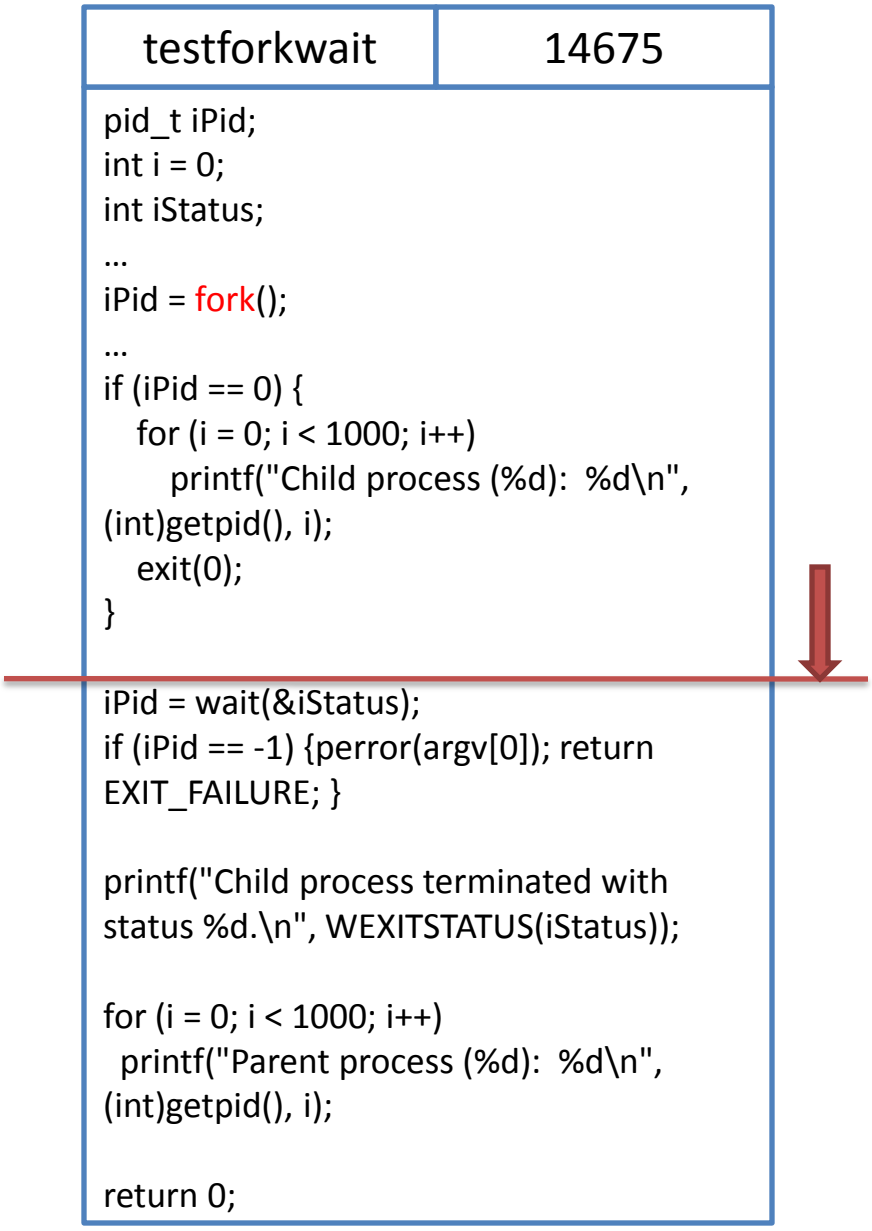
```
*/
```

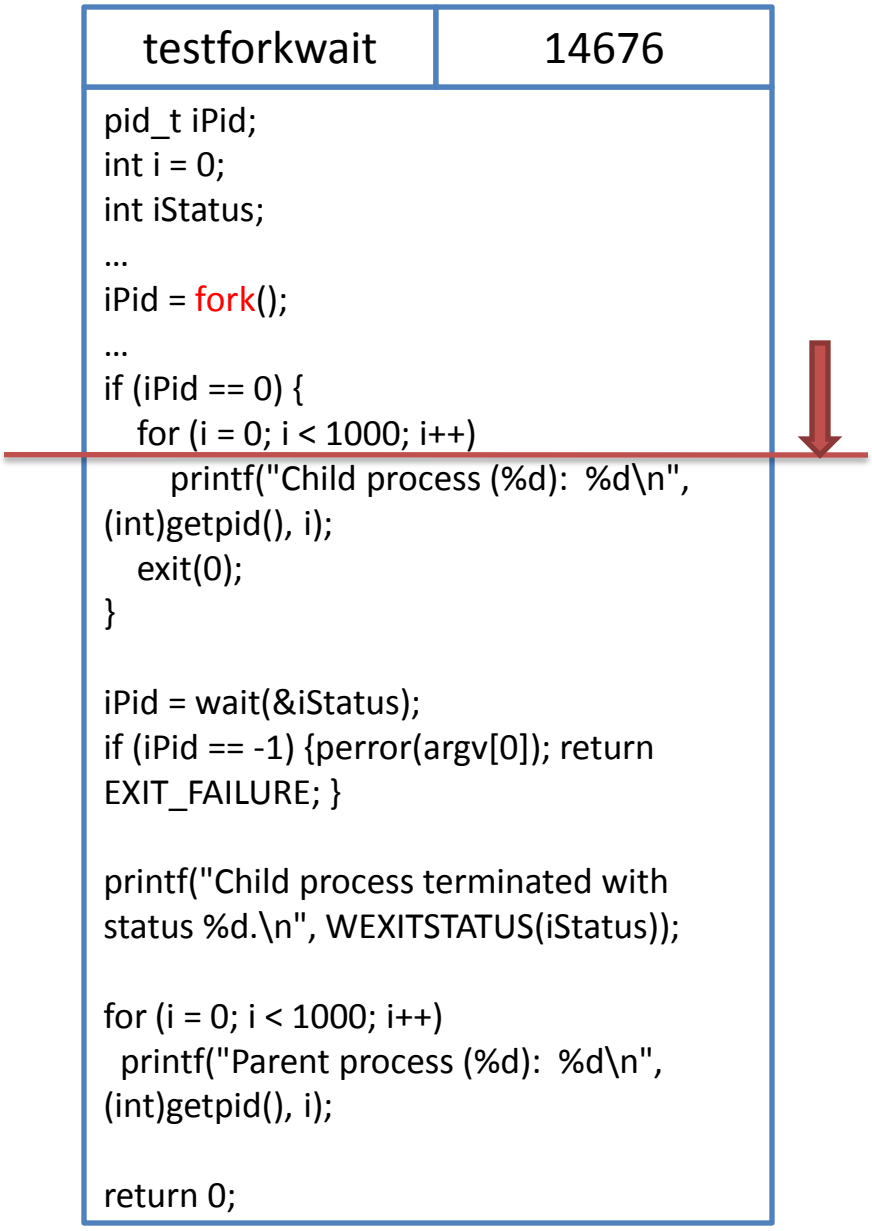
- Either of the following scenarios is possible
  - Parent process finishes first
  - Child process finishes first
- Can we specify the order of the execution?
  - Parent calls wait()
- Let's look at 'testforkwait.c'


testforkwait
14675
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;

testforkwait	14675

```
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
   for (i = 0; i < 1000; i++)
      printf("Child process (%d):  %d\n",
(int)getpid(), i);
   exit(0);
}

iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));

for (i = 0; i < 1000; i++)
  printf("Parent process (%d):  %d\n",
(int)getpid(), i);

return 0;
```


testforkwait
14675
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;

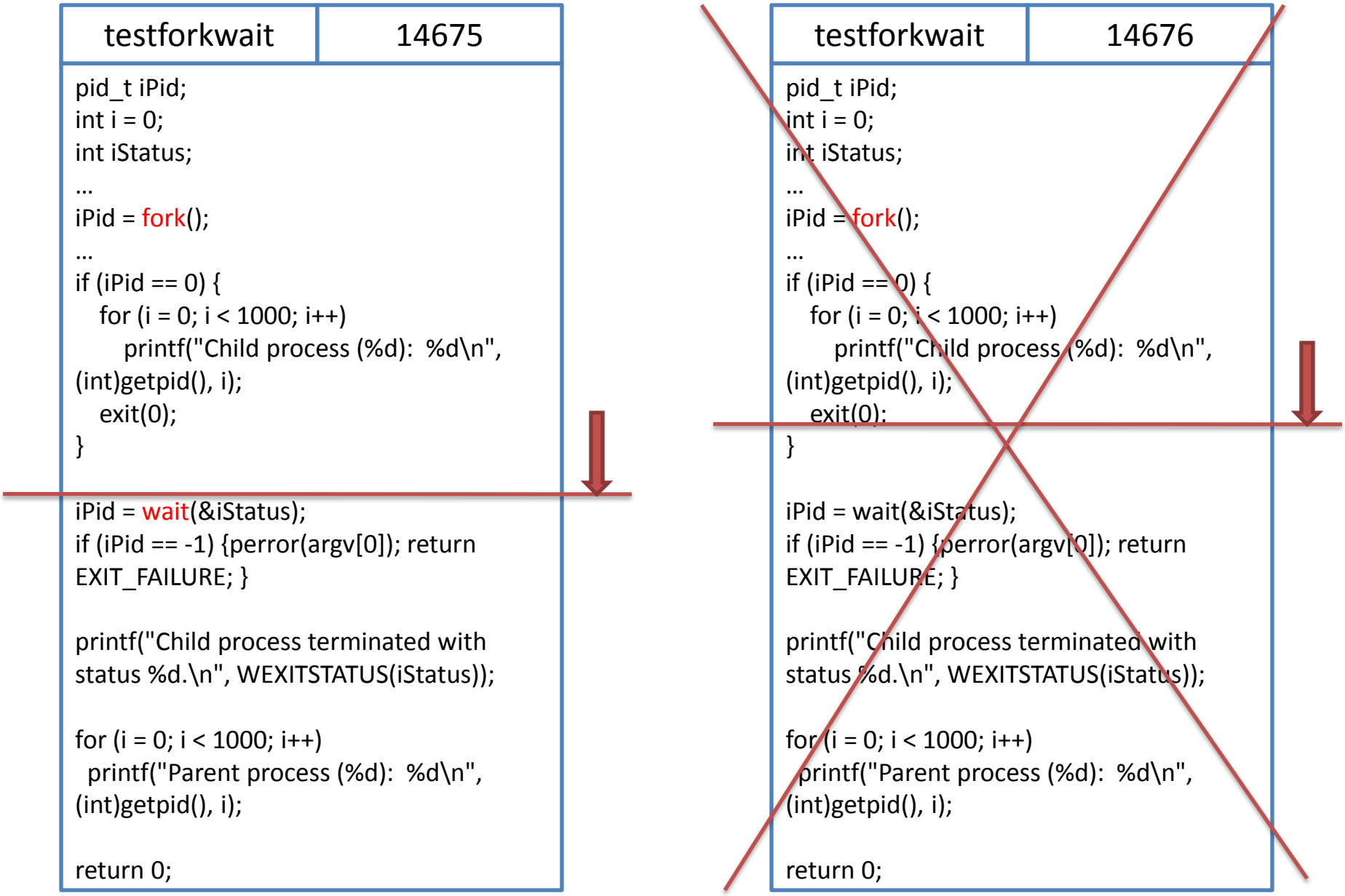

testforkwait
14676
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;

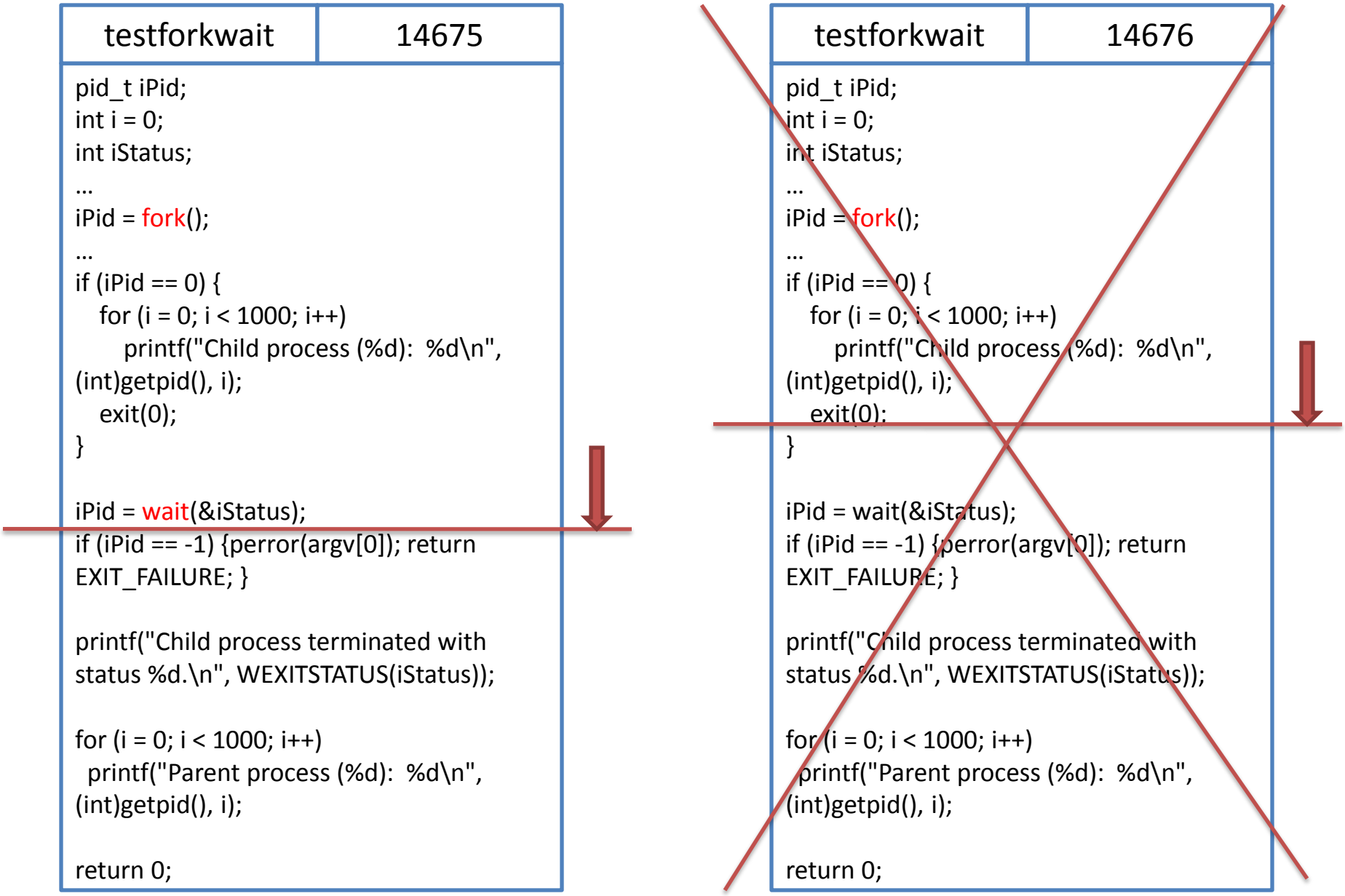
testforkwait	14675

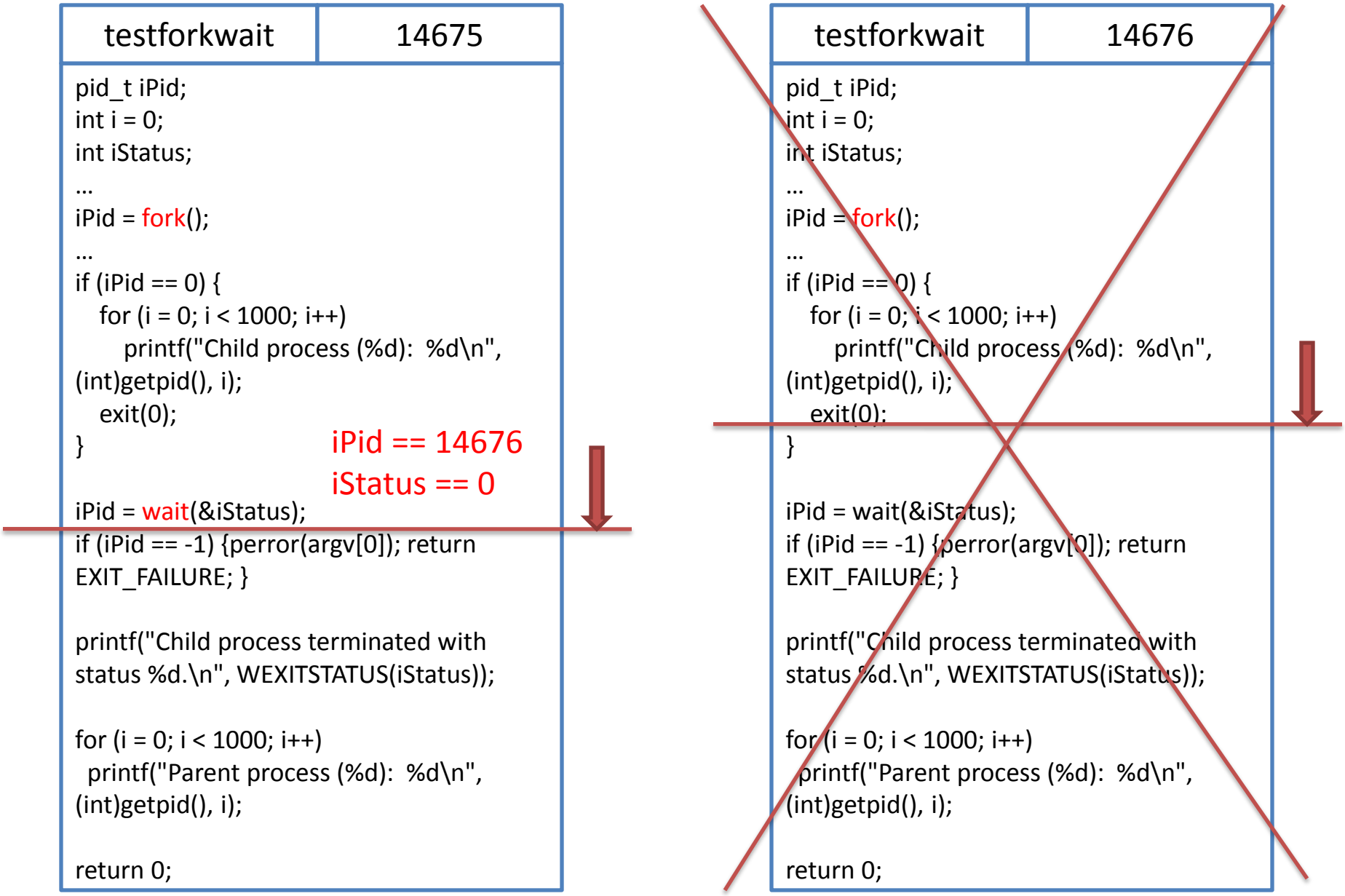
```
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
   for (i = 0; i < 1000; i++)
      printf("Child process (%d):  %d\n",
(int)getpid(), i);
   exit(0);
}

iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));

for (i = 0; i < 1000; i++)
  printf("Parent process (%d):  %d\n",
(int)getpid(), i);

return 0;
```

testforkwait	14676

```
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
   for (i = 0; i < 1000; i++)
      printf("Child process (%d):  %d\n",
(int)getpid(), i);
   exit(0);
}

iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));

for (i = 0; i < 1000; i++)
  printf("Parent process (%d):  %d\n",
(int)getpid(), i);

return 0;
```


testforkwait
14675
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;
testforkwait
14676
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;


testforkwait
14675
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;
testforkwait
14676
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;

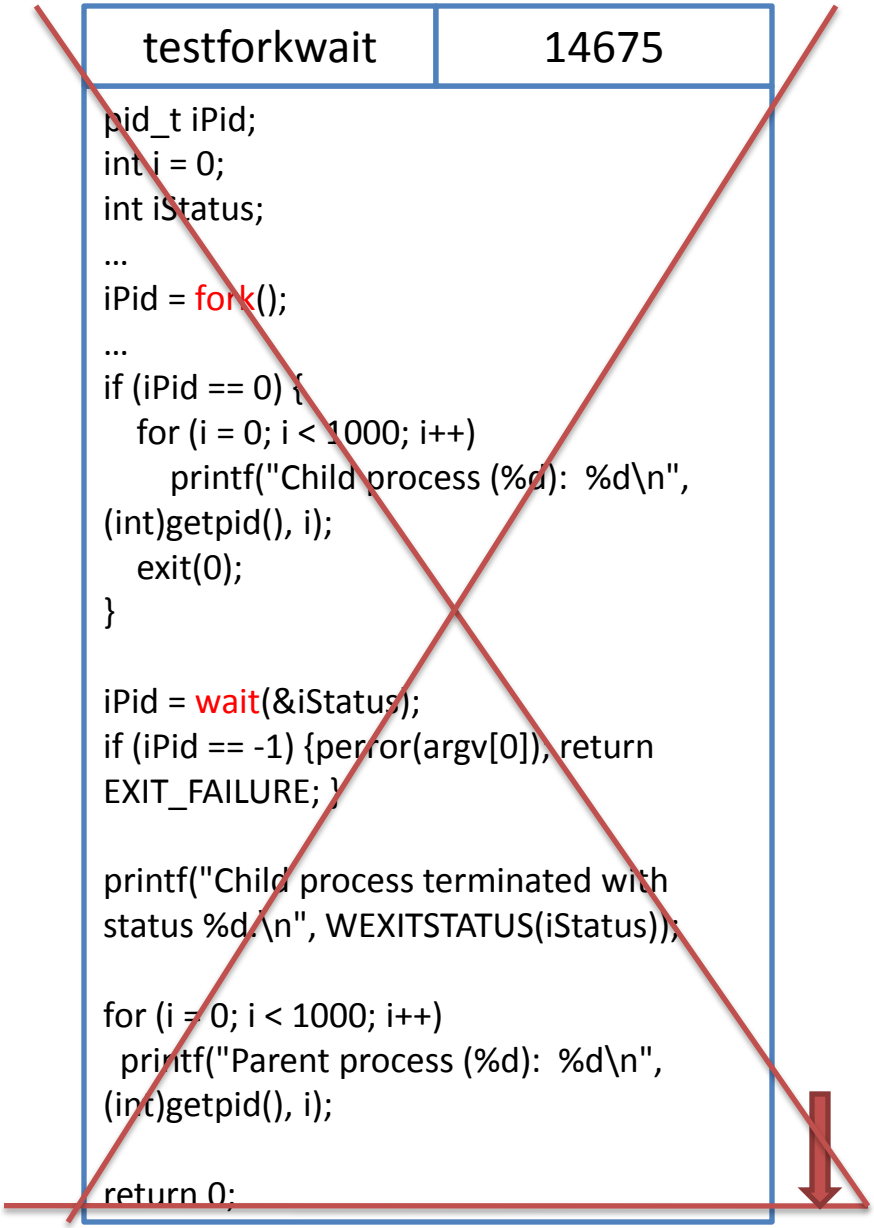

testforkwait
14675
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid == 14676
iStatus == 0
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;
testforkwait
14676
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;

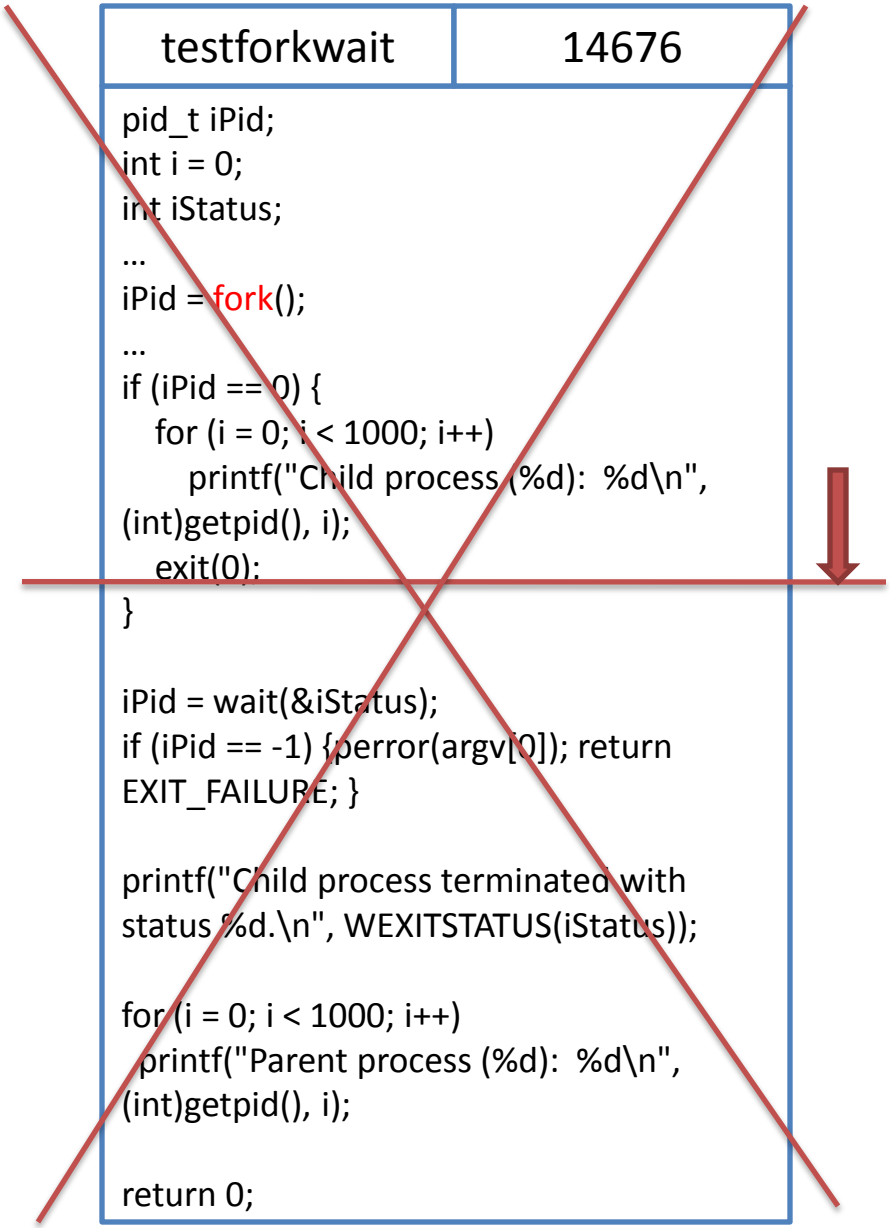
testforkwait	14675

```
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
   for (i = 0; i < 1000; i++)
      printf("Child process (%d):  %d\n",
(int)getpid(), i);
   exit(0);
}

iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));

for (i = 0; i < 1000; i++)
  printf("Parent process (%d):  %d\n",
(int)getpid(), i);

return 0;
```

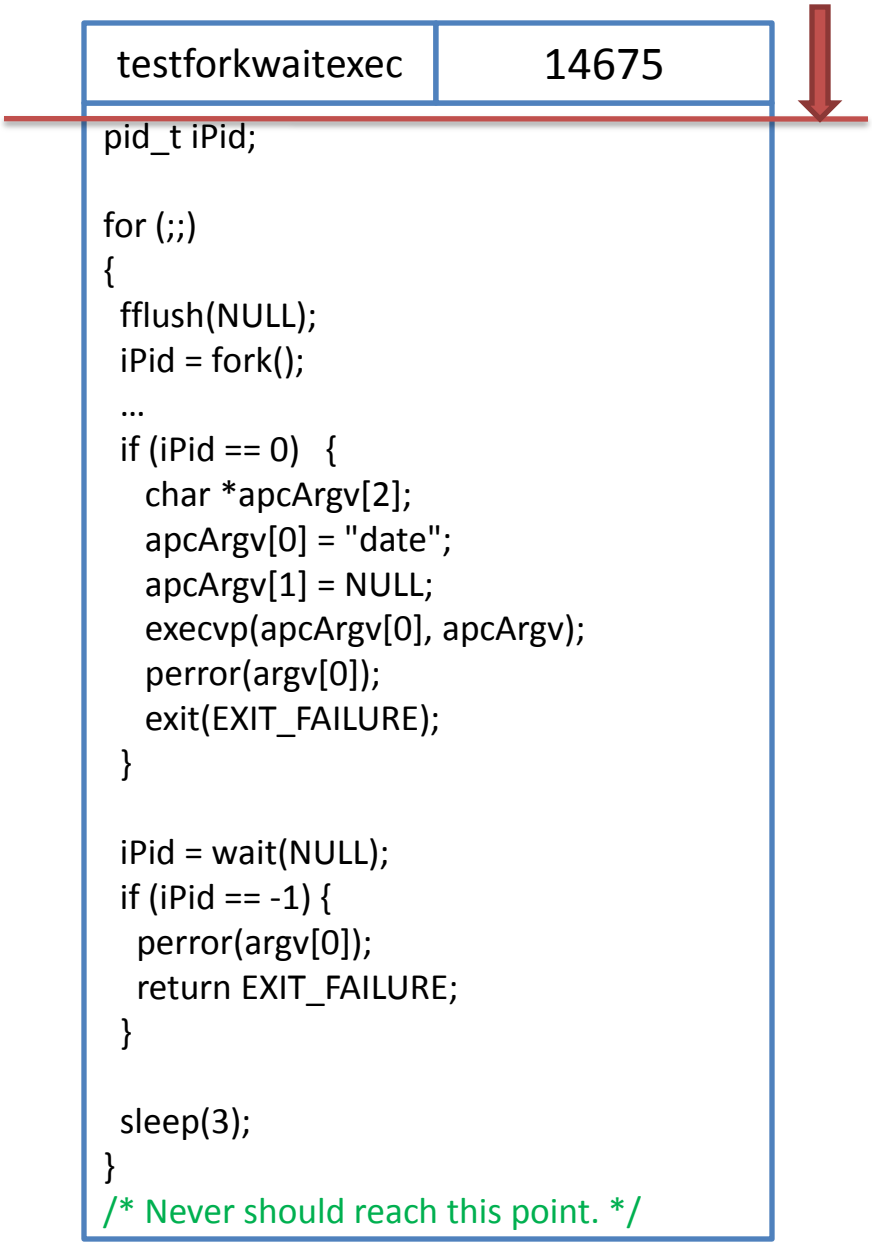
testforkwait	14676

```
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
   for (i = 0; i < 1000; i++)
      printf("Child process (%d):  %d\n",
(int)getpid(), i);
   exit(0);
}

iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }

printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));

for (i = 0; i < 1000; i++)
  printf("Parent process (%d):  %d\n",
(int)getpid(), i);

return 0;
```


testforkwait
14675
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;

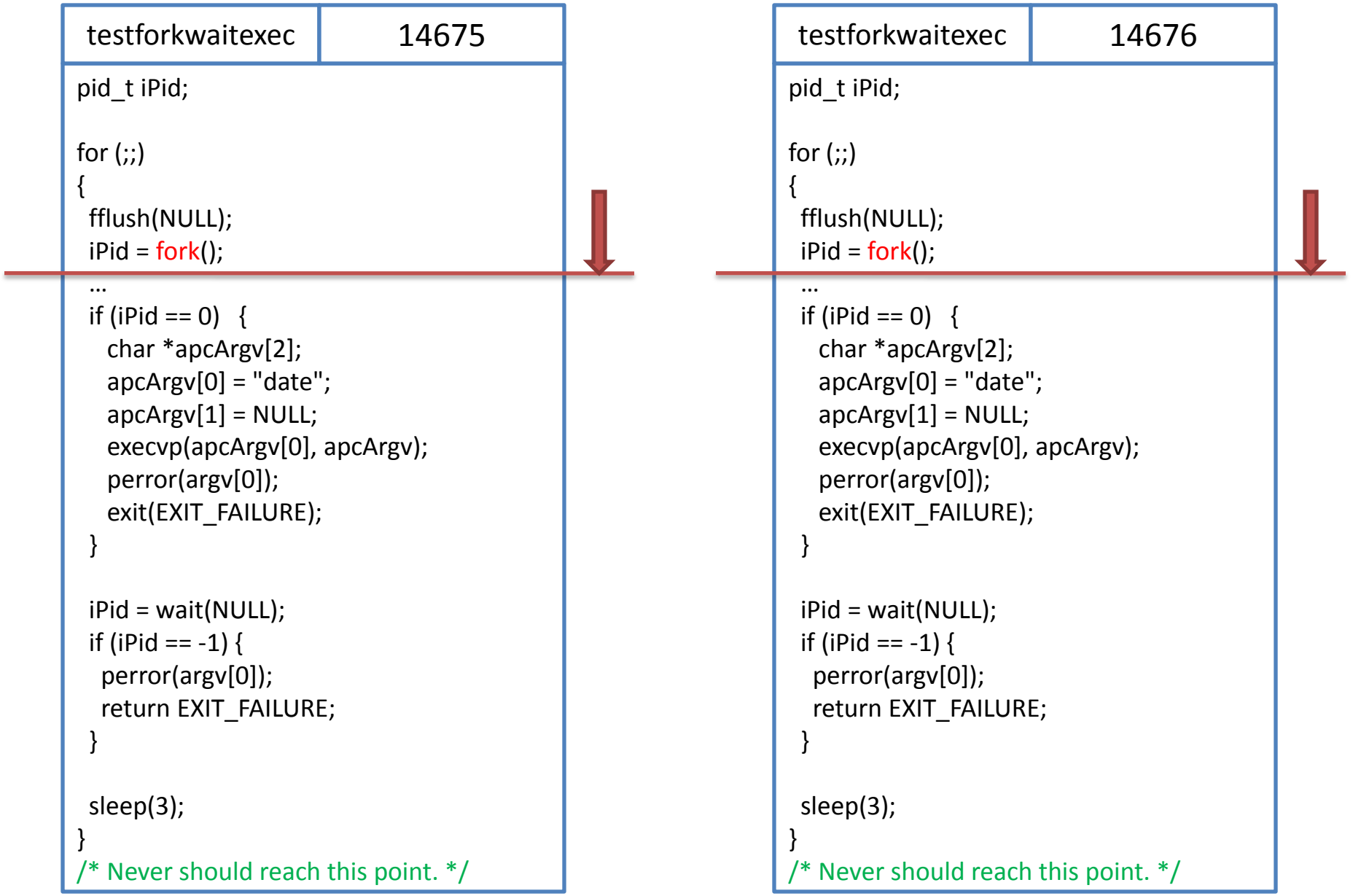

testforkwait
14676
pid_t iPid;
int i = 0;
int iStatus;
…
iPid = fork();
…
if (iPid == 0) {
for (i = 0; i < 1000; i++)
printf("Child process (%d): %d\n",
(int)getpid(), i);
exit(0);
}
iPid = wait(&iStatus);
if (iPid == -1) {perror(argv[0]); return
EXIT_FAILURE; }
printf("Child process terminated with
status %d.\n", WEXITSTATUS(iStatus));
for (i = 0; i < 1000; i++)
printf("Parent process (%d): %d\n",
(int)getpid(), i);
return 0;

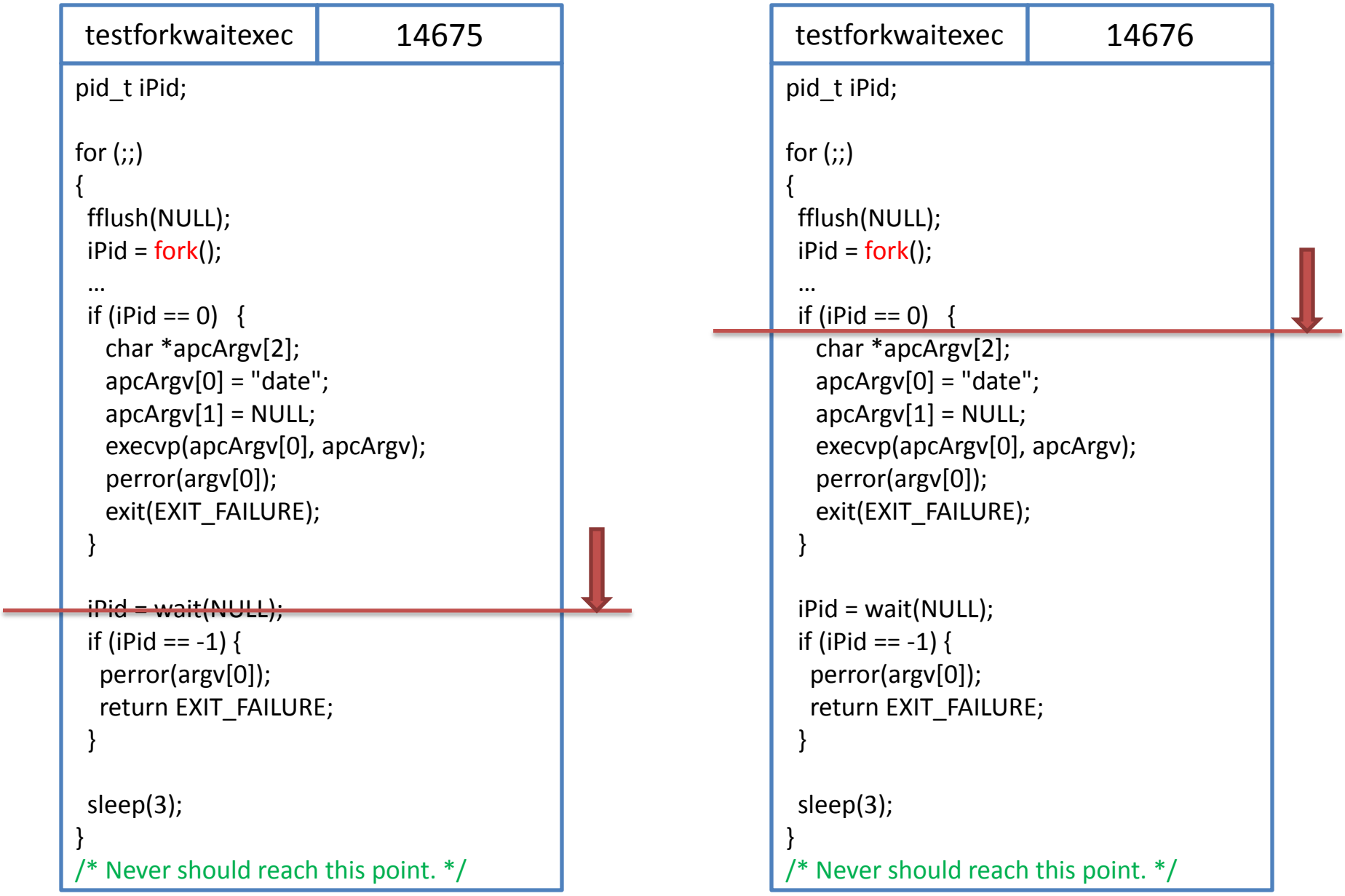
- Let’s look at ‘textforkexecwait.c’

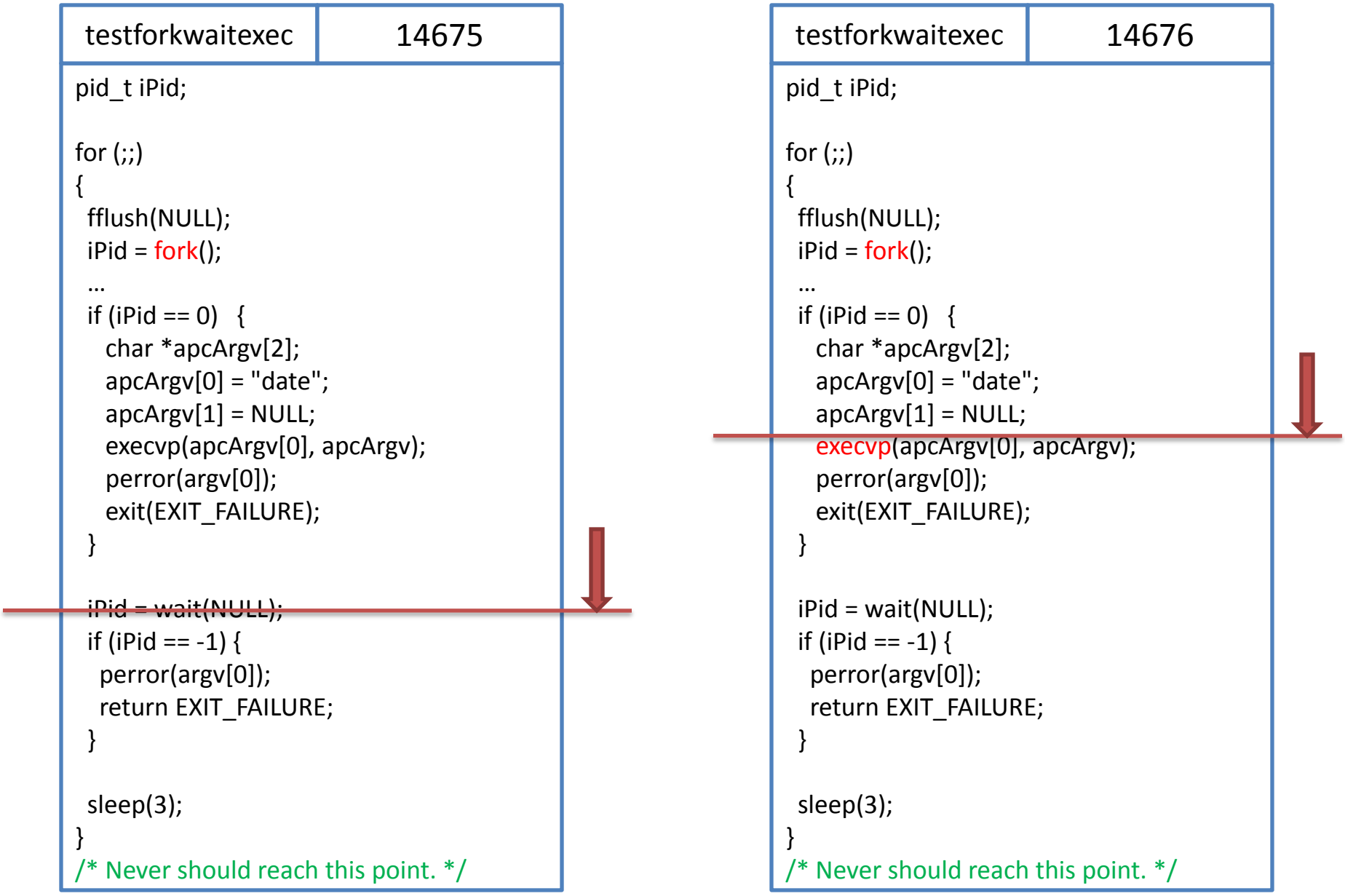

testforkwaitexec
14675
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
…
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */

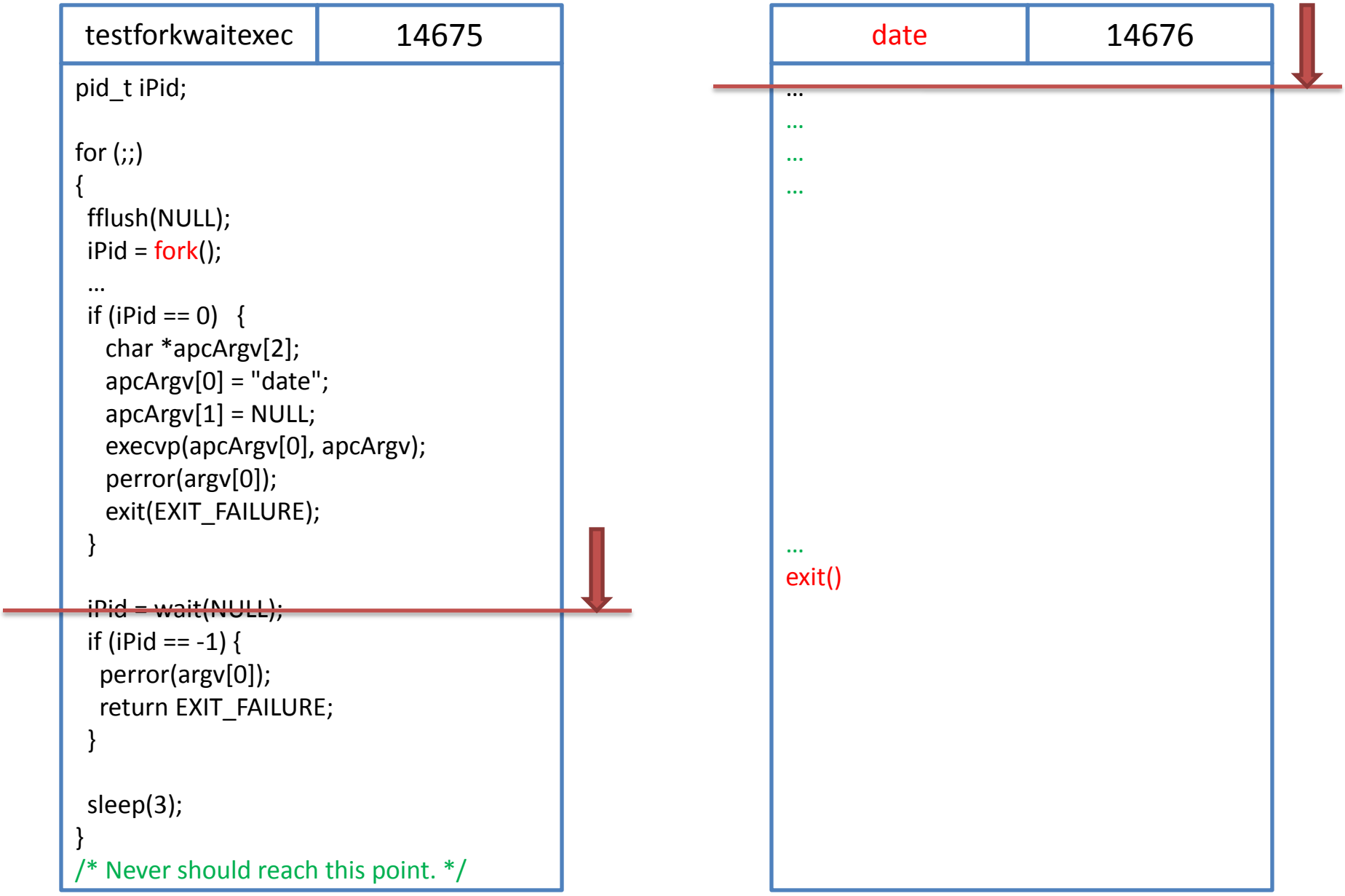
testforkwaitexec	14675

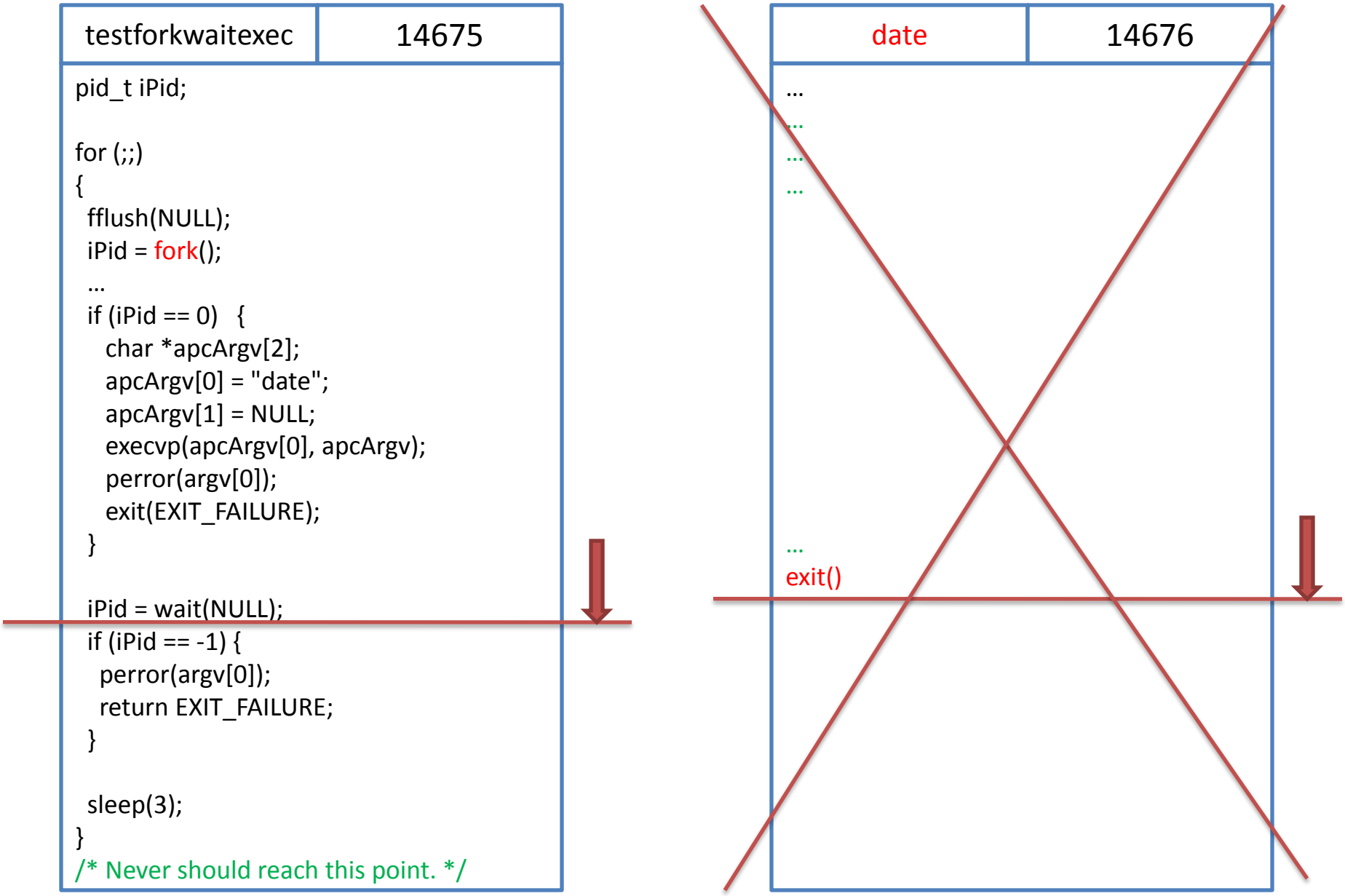
```
pid_t iPid;

for (;;)
{
  fflush(NULL);
  iPid = fork();
  …
  if (iPid == 0)   {
     char *apcArgv[2];
     apcArgv[0] = "date";
     apcArgv[1] = NULL;
     execvp(apcArgv[0], apcArgv);
     perror(argv[0]);
     exit(EXIT_FAILURE);
  }

  iPid = wait(NULL);
  if (iPid == -1) {
    perror(argv[0]);
    return EXIT_FAILURE;
  }

  sleep(3);
}
/* Never should reach this point. */
```


testforkwaitexec
14675
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
...
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */
testforkwaitexec
14676
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
...
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */


testforkwaitexec
14675
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
…
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */
testforkwaitexec
14676
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
…
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */


testforkwaitexec
14675
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
…
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */
testforkwaitexec
14676
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
…
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */


testforkwaitexec
14675
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
…
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */
date
14676
…
…
…
…
…
exit()


testforkwaitexec
14675
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
…
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */
date
14676
…
…
…
…
…
exit()

testforkwaitexec	14675

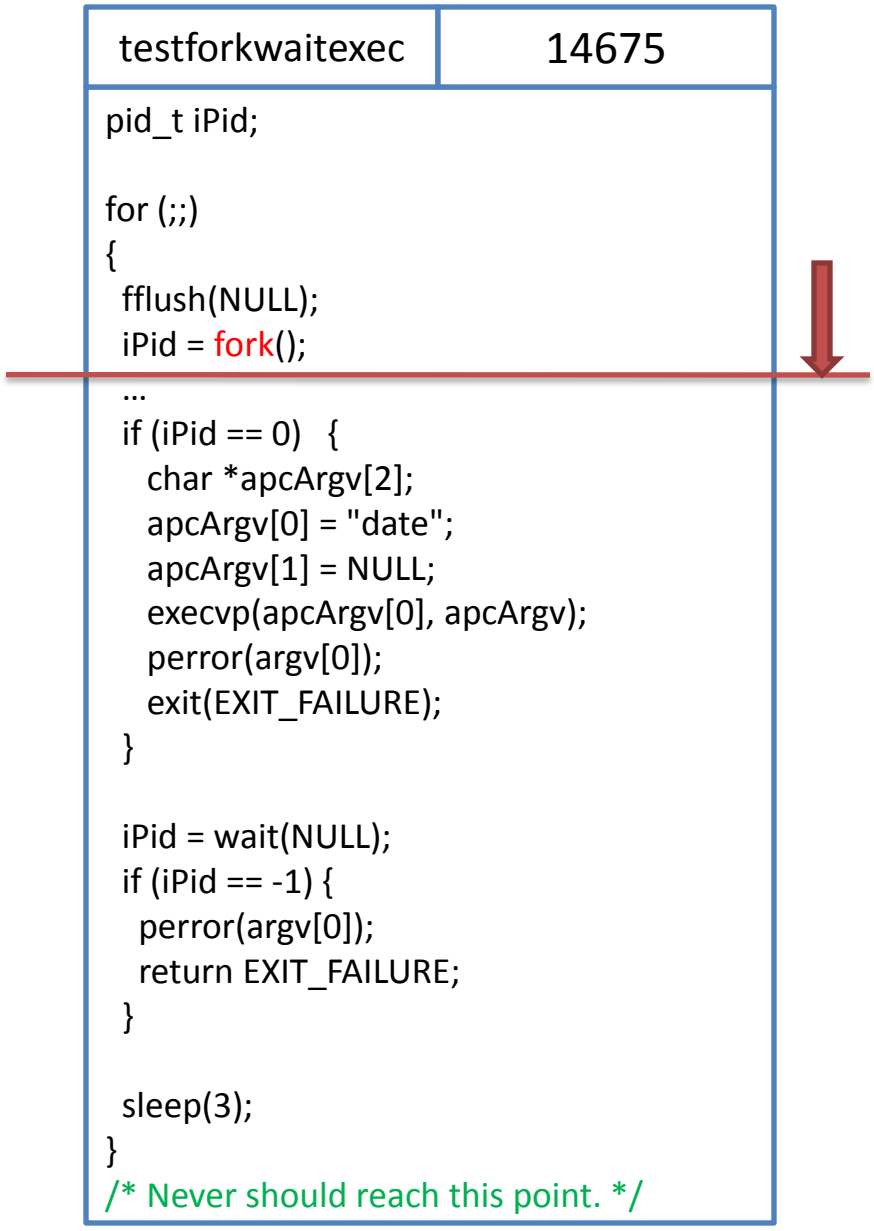
```
pid_t iPid;

for (;;)
{
 fflush(NULL);
 iPid = fork();
 …
 if (iPid == 0)   {
    char *apcArgv[2];
    apcArgv[0] = "date";
    apcArgv[1] = NULL;
    execvp(apcArgv[0], apcArgv);
    perror(argv[0]);
    exit(EXIT_FAILURE);
 }

 iPid = wait(NULL);
 if (iPid == -1) {
   perror(argv[0]);
   return EXIT_FAILURE;
 }

 sleep(3);
}
/* Never should reach this point. */
```

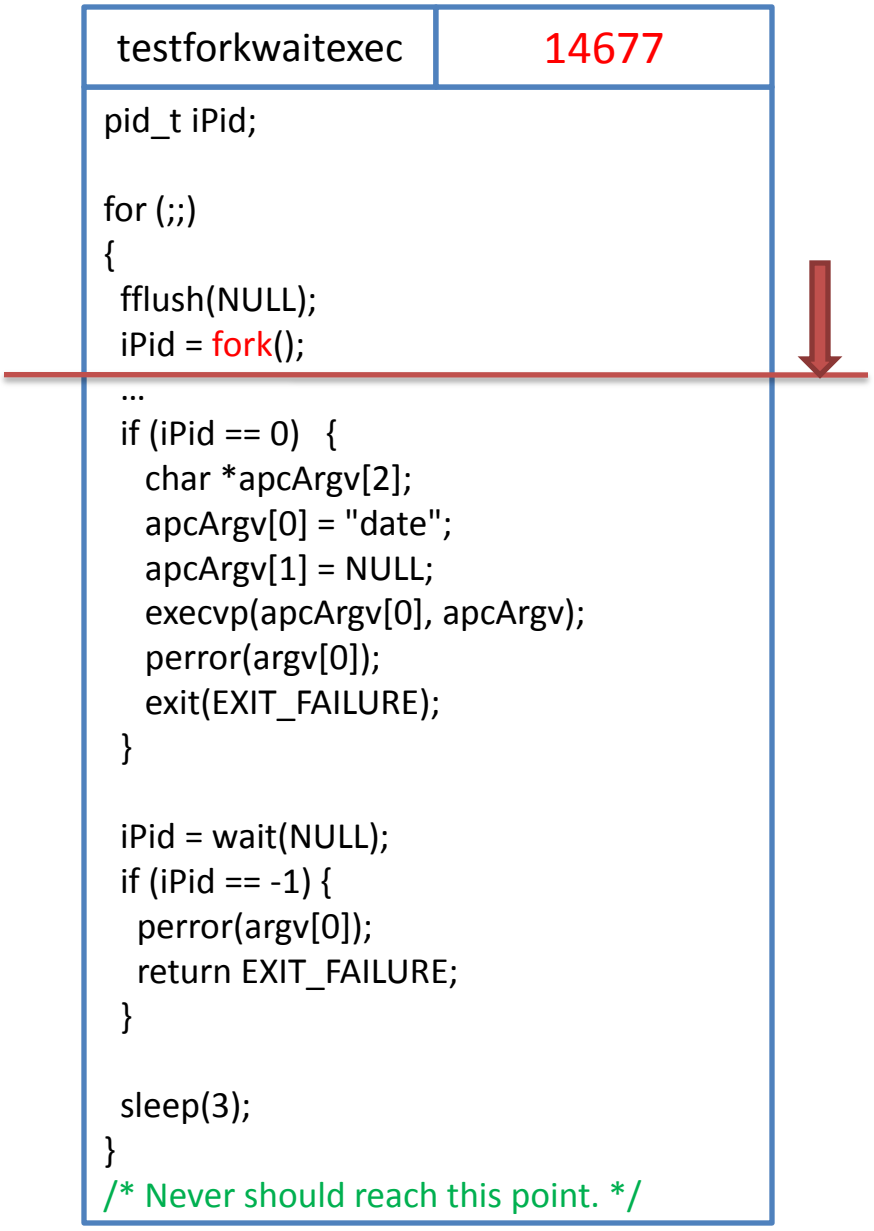
testforkwaitexec	14675

```
pid_t iPid;

for (;;)
{
  fflush(NULL);
  iPid = fork();
  …
  if (iPid == 0)   {
     char *apcArgv[2];
     apcArgv[0] = "date";
     apcArgv[1] = NULL;
     execvp(apcArgv[0], apcArgv);
     perror(argv[0]);
     exit(EXIT_FAILURE);
  }

  iPid = wait(NULL);
  if (iPid == -1) {
    perror(argv[0]);
    return EXIT_FAILURE;
  }

  sleep(3);
}
/* Never should reach this point. */
```

testforkwaitexec	14675

```
pid_t iPid;

for (;;)
{
  fflush(NULL);
  iPid = fork();
  …
  if (iPid == 0)   {
     char *apcArgv[2];
     apcArgv[0] = "date";
     apcArgv[1] = NULL;
     execvp(apcArgv[0], apcArgv);
     perror(argv[0]);
     exit(EXIT_FAILURE);
  }

  iPid = wait(NULL);
  if (iPid == -1) {
    perror(argv[0]);
    return EXIT_FAILURE;
  }

  sleep(3);
}
/* Never should reach this point. */
```

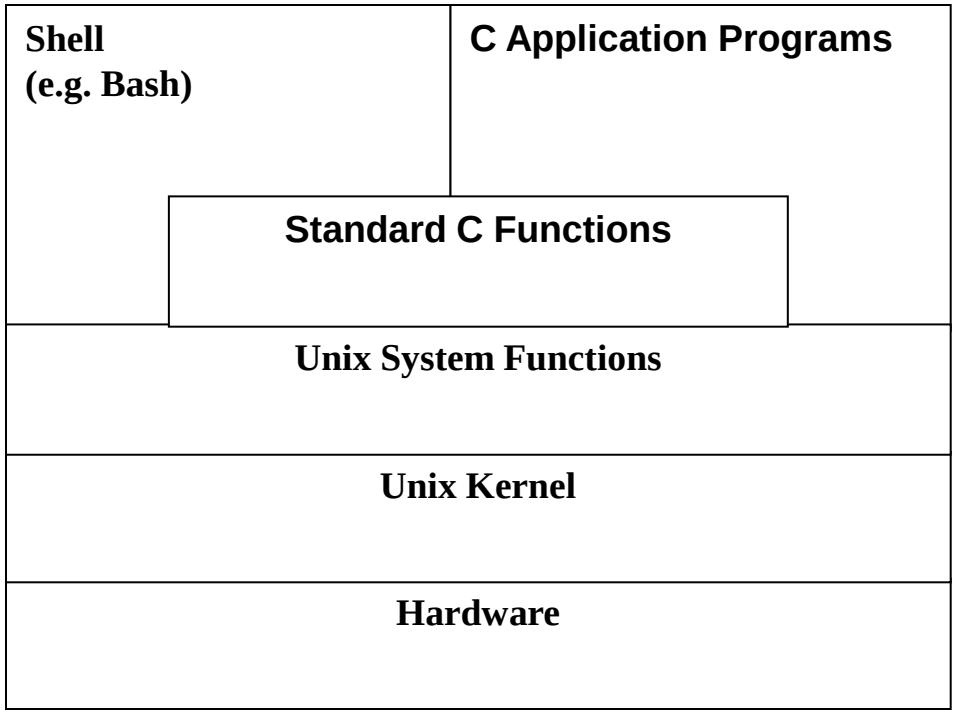

testforkwaitexec
14675
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
...
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */

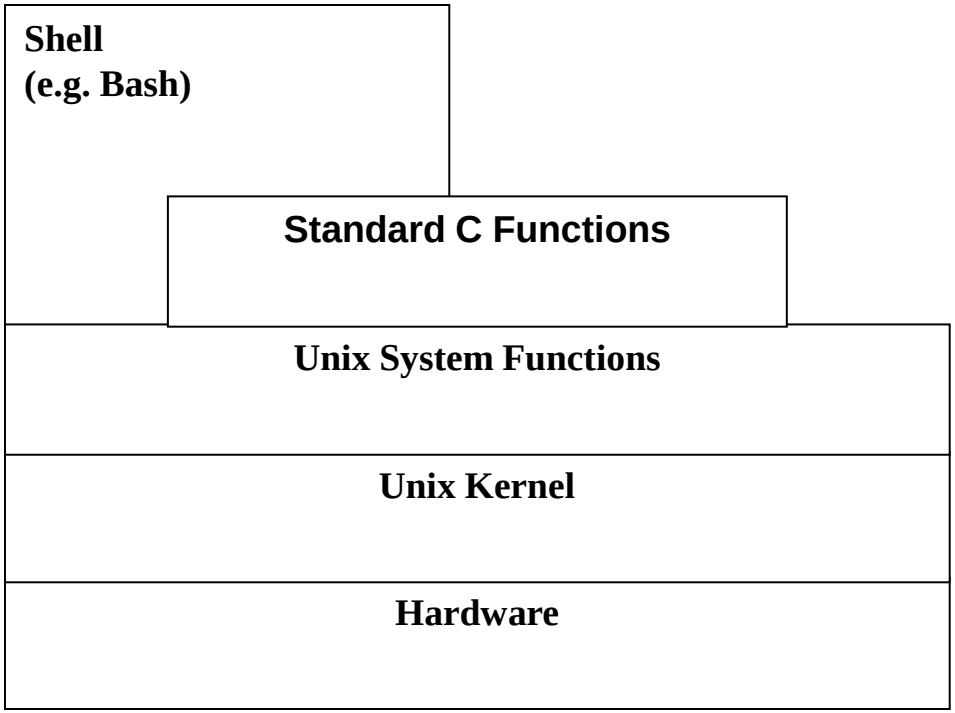

testforkwaitexec
14677
pid_t iPid;
for (;;)
{
fflush(NULL);
iPid = fork();
...
if (iPid == 0) {
char *apcArgv[2];
apcArgv[0] = "date";
apcArgv[1] = NULL;
execvp(apcArgv[0], apcArgv);
perror(argv[0]);
exit(EXIT_FAILURE);
}
iPid = wait(NULL);
if (iPid == -1) {
perror(argv[0]);
return EXIT_FAILURE;
}
sleep(3);
}
/* Never should reach this point. */

testforkwaitexec	14675

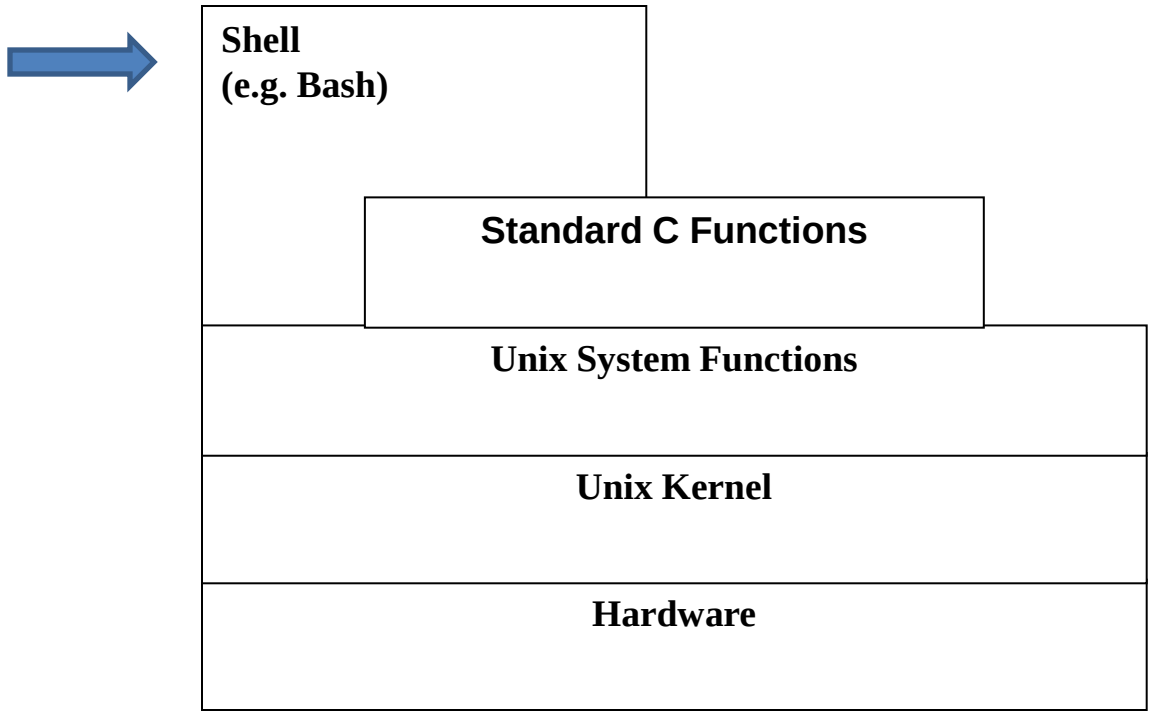
```
pid_t iPid;

for (;;)
{
 fflush(NULL);
 iPid = fork();
 …
 if (iPid == 0)   {
    char *apcArgv[2];
    apcArgv[0] = "date";
    apcArgv[1] = NULL;
    execvp(apcArgv[0], apcArgv);
    perror(argv[0]);
    exit(EXIT_FAILURE);
 }

 iPid = wait(NULL);
 if (iPid == -1) {
   perror(argv[0]);
   return EXIT_FAILURE;
 }

 sleep(3);
}
/* Never should reach this point. */
```

```
$ testforkexecwait
Sat Dec  8 16:18:13 EST 2007
Sat Dec  8 16:18:16 EST 2007
Sat Dec  8 16:18:19 EST 2007
Sat Dec  8 16:18:22 EST 2007
Sat Dec  8 16:18:25 EST 2007
Ctrl-C
```


Shell
(e.g. Bash)
C Application Programs
Standard C Functions
Unix System Functions
Unix Kernel
Hardware

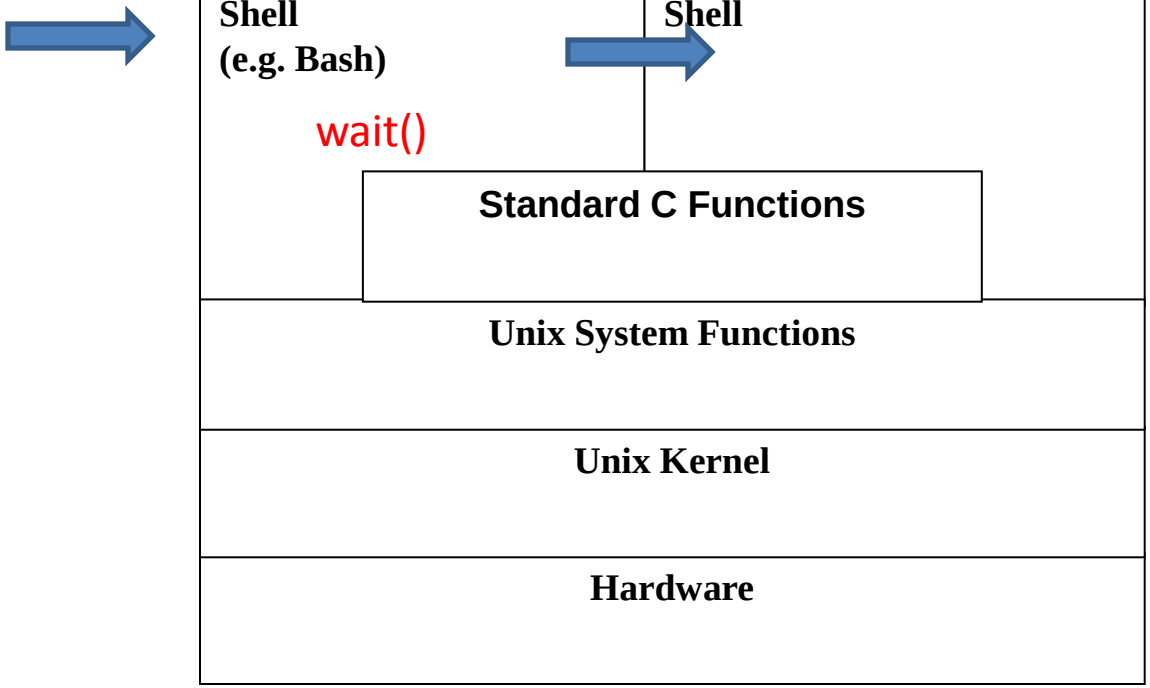

Shell
(e.g. Bash)
Standard C Functions
Unix System Functions
Unix Kernel
Hardware

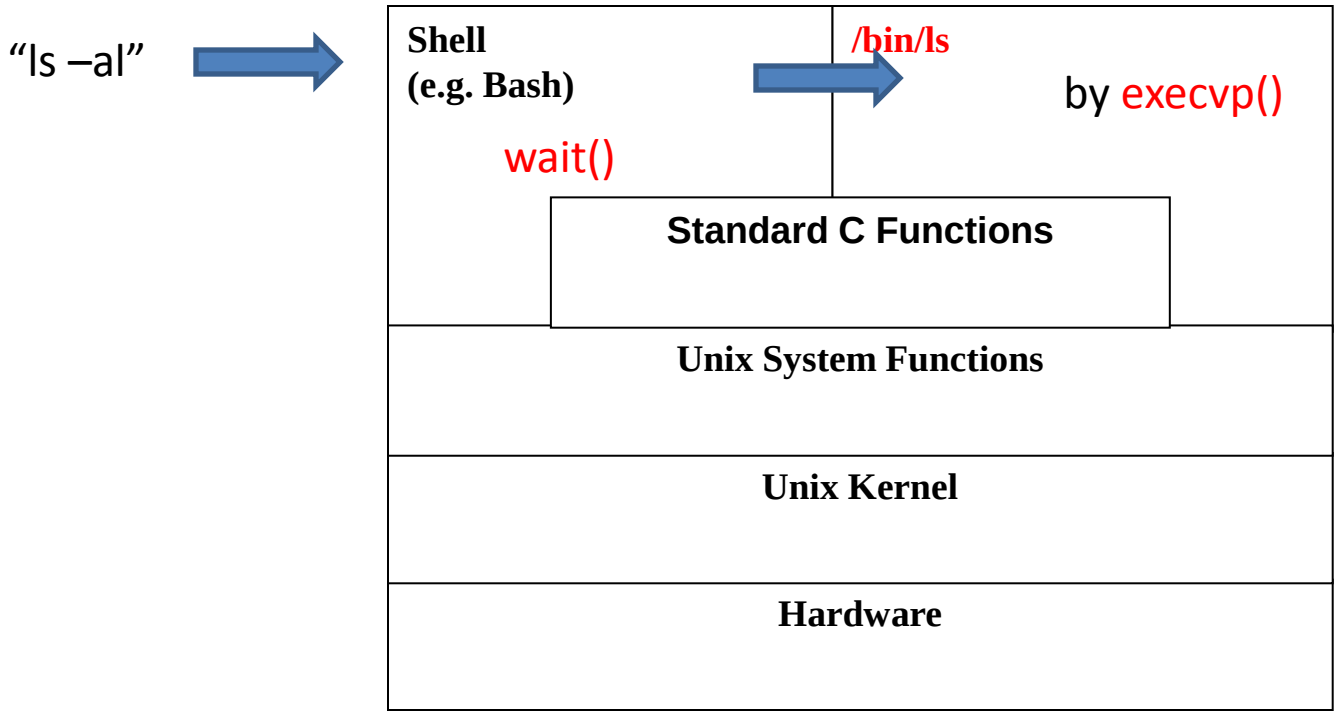
“ls –al”


Shell
(e.g. Bash)
Standard C Functions
Unix System Functions
Unix Kernel
Hardware

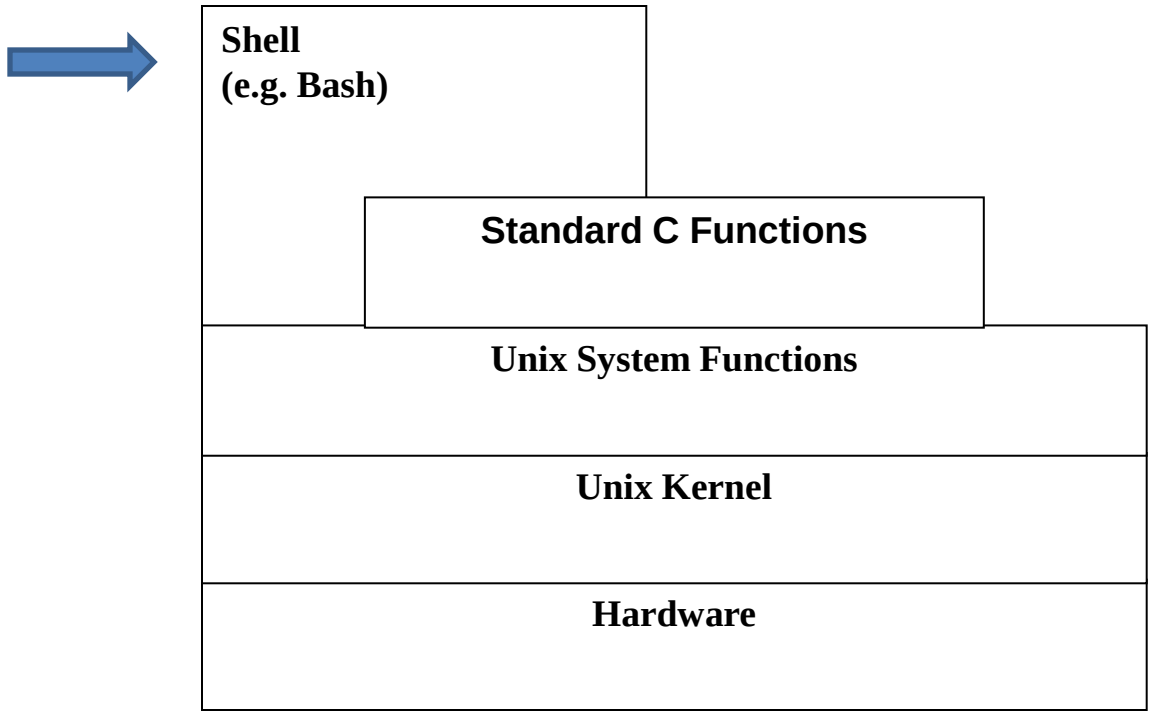
by fork()

“ls –al”


Shell
(e.g. Bash)
wait()
Shell
Standard C Functions
Unix System Functions
Unix Kernel
Hardware


"ls –al"
Shell
(e.g. Bash)
wait()
/bin/ls
by execvp()
Standard C Functions
Unix System Functions
Unix Kernel
Hardware

“ls –al”


Shell
(e.g. Bash)
Standard C Functions
Unix System Functions
Unix Kernel
Hardware