Final Precept: Ish

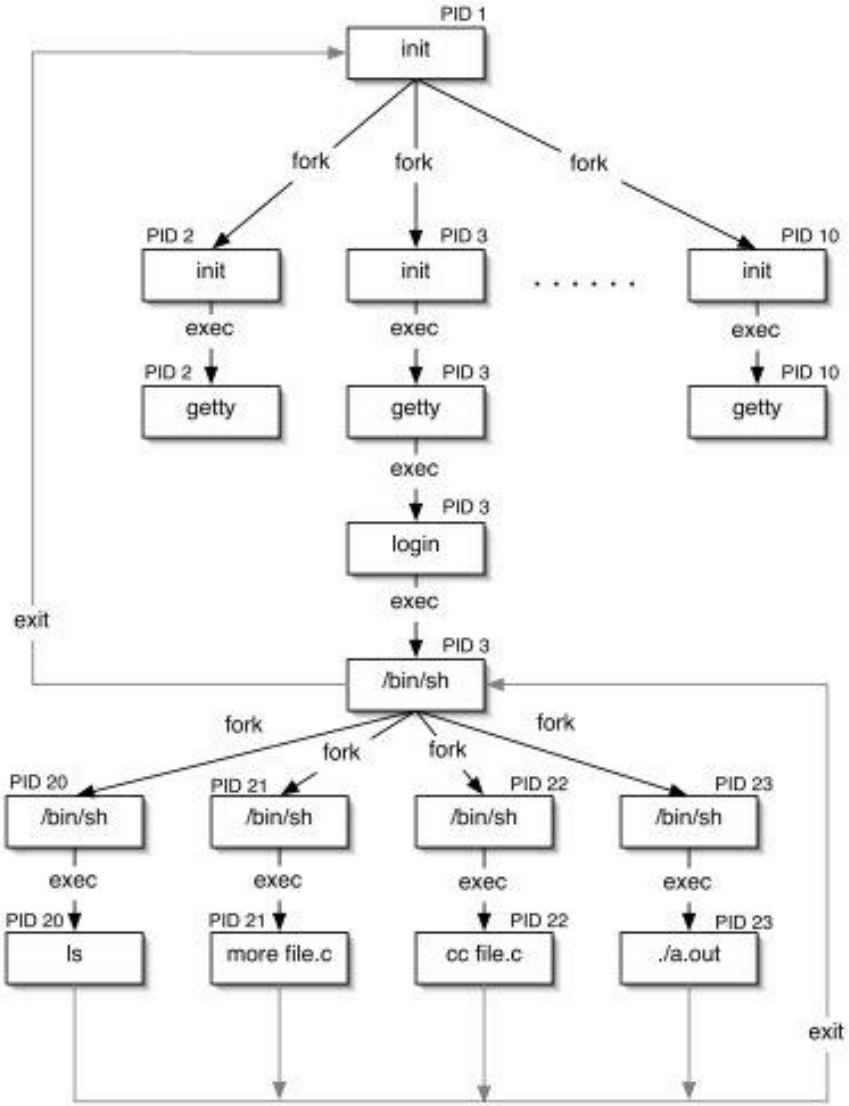
Slides Originally Prepared by:
Wonho Kim

Agenda

- Last time
  - exec(), fork()
  - wait()
- Today
  - zombie, orphan process
  - built-in commands in ish
  - I/O redirection
  - Unix signal

Process Hierarchy

- Every process is created by fork()
  - Except ‘init (pid 1)’ process
- Every process should have its parent


PID 1
init
fork
fork
fork
PID 2
init
PID 3
init
PID 10
init
exec
exec
exec
PID 2
getty
PID 3
getty
PID 10
getty
exec
PID 3
login
exec
exit
PID 3
/bin/sh
fork
fork
fork
fork
PID 20
/bin/sh
PID 21
/bin/sh
PID 22
/bin/sh
PID 23
/bin/sh
exec
exec
exec
exec
PID 20
ls
PID 21
more file.c
PID 22
cc file.c
PID 23
./a.out
exit

Zombie Process

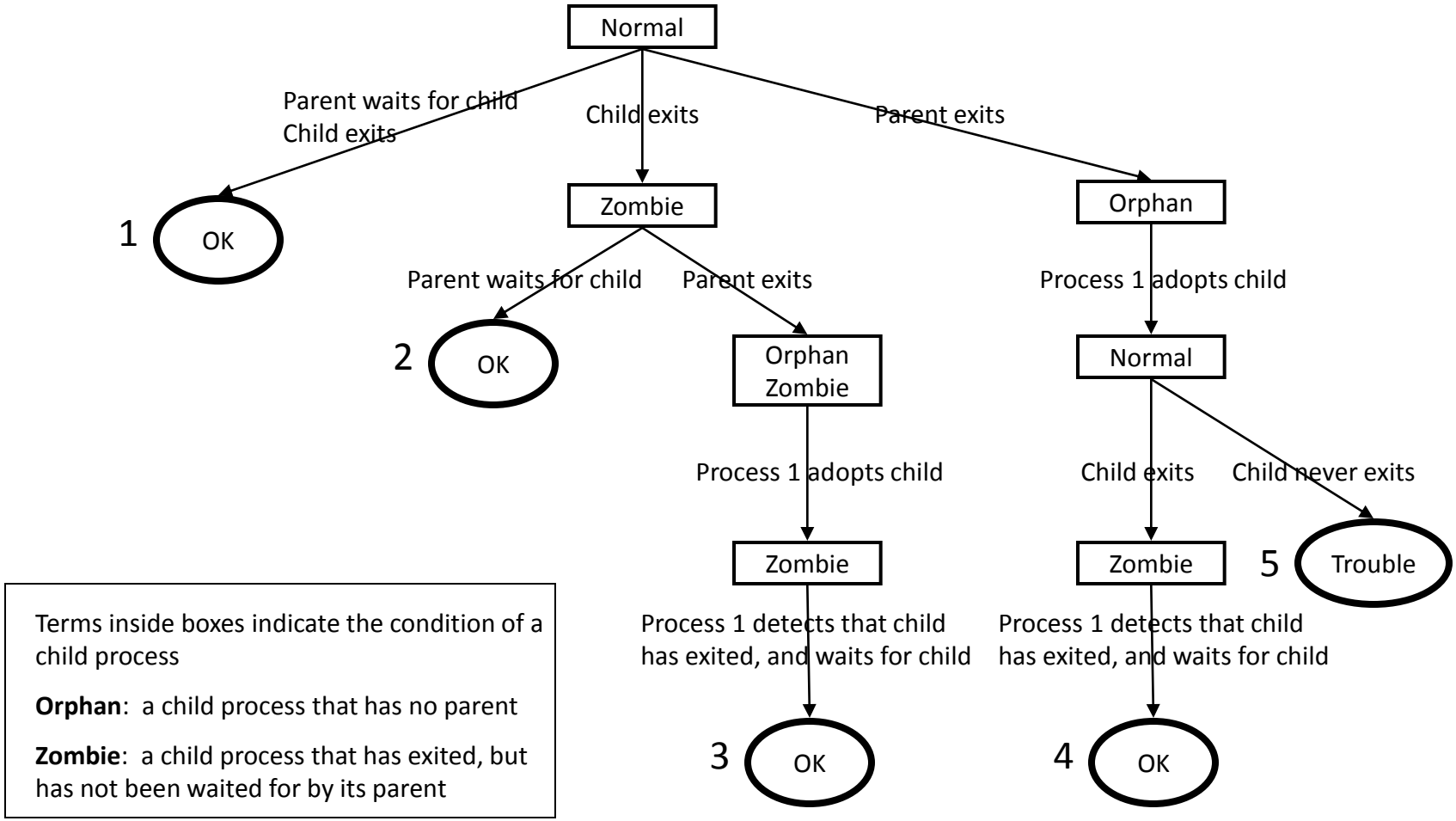
- When child process exits
  - it becomes ‘zombie’ process
- Why do we need ‘zombie’ status??
  - it has some information (e.g., exit status)
  - So, it takes some memory

Zombie Process

- When parent process calls wait()
  - The ‘zombie’ process is completely destroyed
  - Its resources are reclaimed
- We should call wait() for every fork()
  - Like free() for every malloc()

Orphan process

- A child process becomes an ‘orphan’ process
  - when its parent process exists without calling wait()
- ‘init’ process takes the orphan process, and will call wait()
  - Anyhow, it will be cleared.
  - However, it would be much better to call wait() explicitly in parent…it’s sad to make orphans.



Terms inside boxes indicate the condition of a child process

Orphan: a child process that has no parent

Zombie: a child process that has exited, but has not been waited for by its parent

Shell Built-in commands

- 4 built-in commands
  - cd
  - setenv
  - unsetenv
  - exit

cd

- Each unix process maintains its own current directory
  - Parent process and child process maintain separate current directories
- What happens if cd is handled by child process?
  - fork(), exec() for ‘cd’ command
  - The execution would change the current directory of the child process,
  - but the current directory of the parent process remain would remain the same

cd

- So, ‘cd’ should be implemented in a shell
- Use ‘chdir()’ system call

setenv & unsetenv

- The same reason
  - Each process has its own set of environment variables
- Use ‘setenv()’ & ‘unsetenv()’ system call

exit

- The same reason
- Use exit() system call

The difficulties

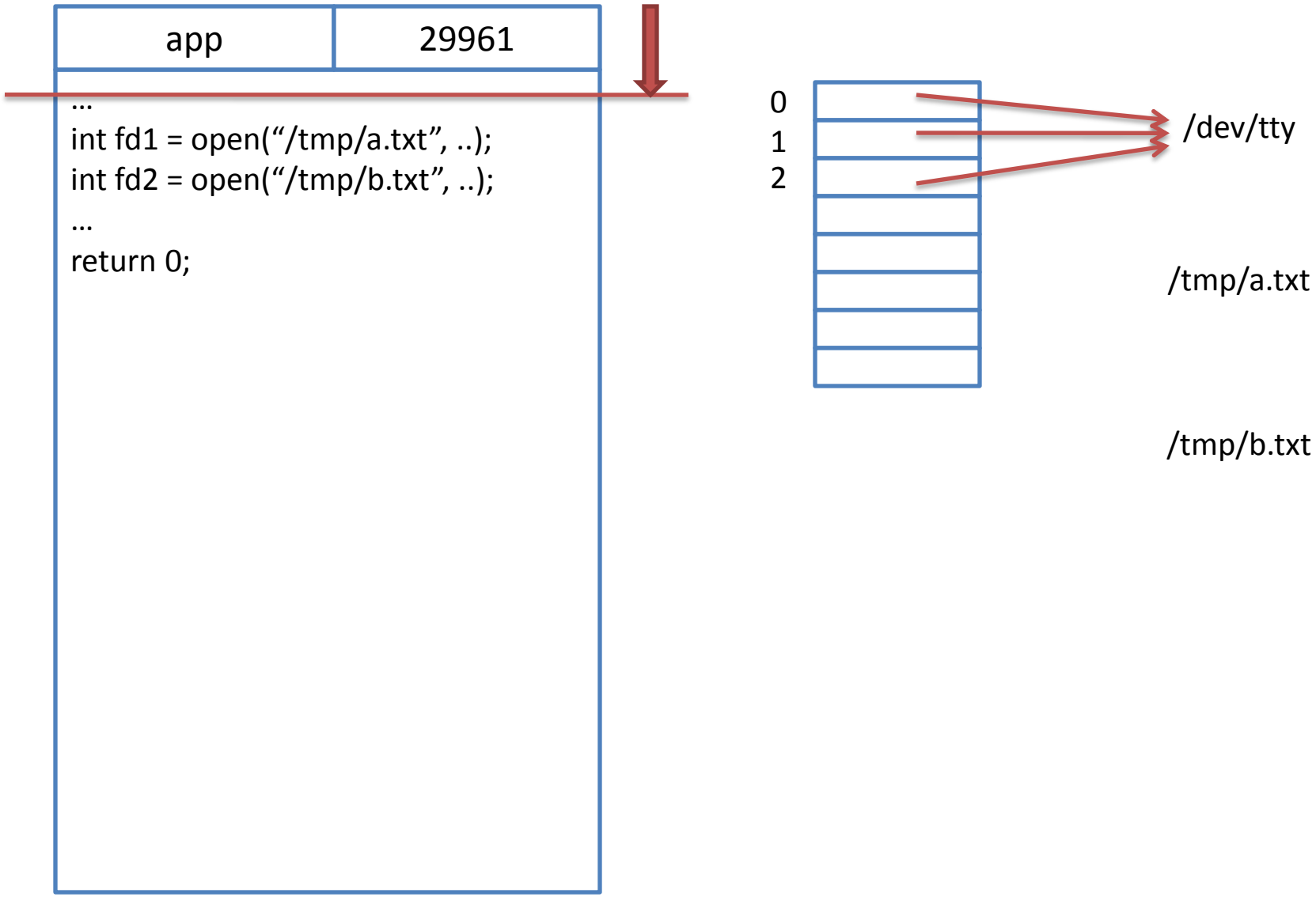
- Implementation would be straightforward
- The difficult part is handling possible user errors
  - cd dir1 dir2
  - cd nonexistingdir
  - cd (but, HOME is not set)
  - setenv XXX YYY ZZZ
  - setenv
  - unsetenv
  - unsetnv XXX YYY

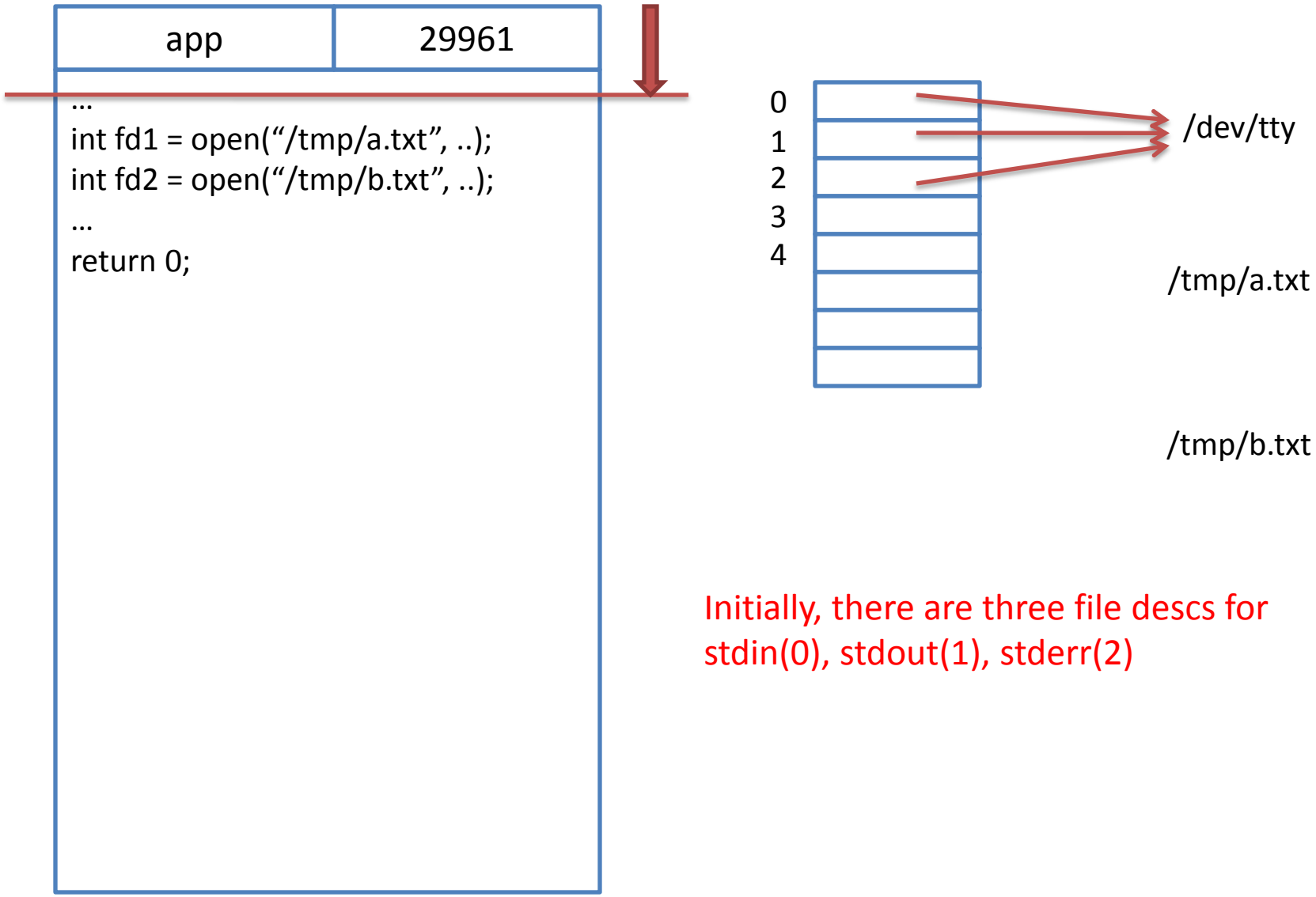
I/O redirection

- We must know
  - file descriptors
  - systems calls
    - creat()
    - open()
    - close()
    - dup()

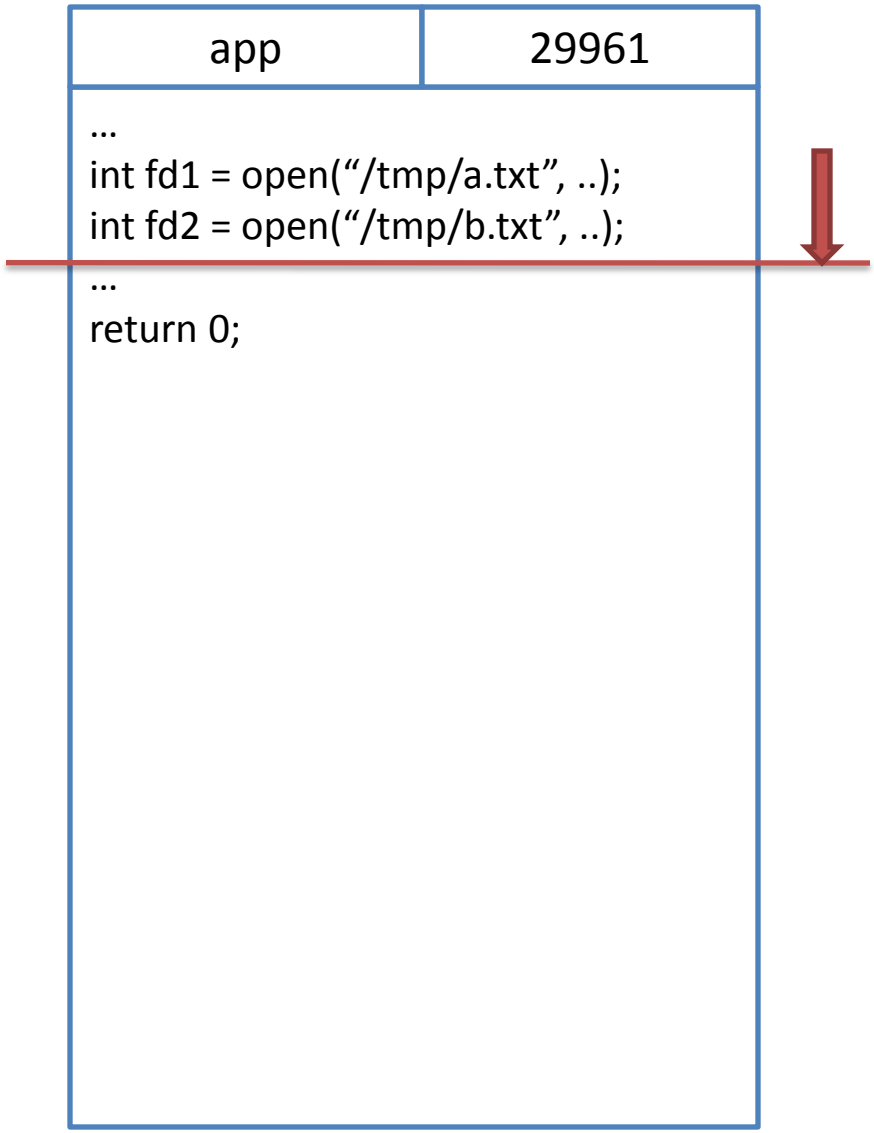
File descriptor

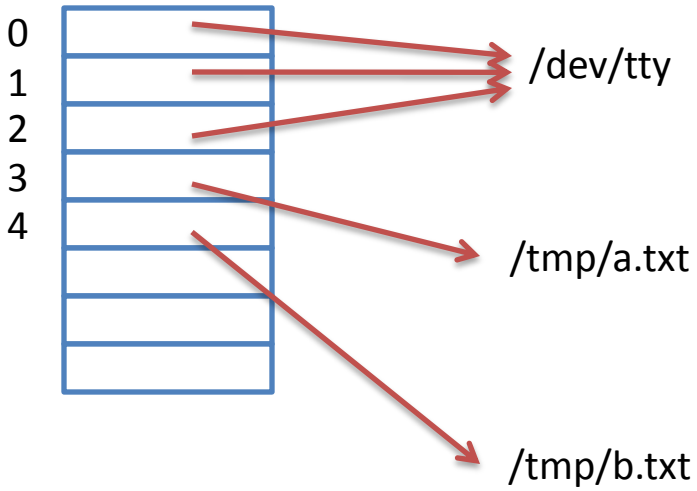
- File descriptor
  - An integer
  - Uniquely identifies an opened file
- File descriptor table
  - An array in the Kernel
  - Indices: file descriptors
  - Elements: Pointers to opened files


app
29961
…
int fd1 = open(“/tmp/a.txt”, ..);
int fd2 = open(“/tmp/b.txt”, ..);
…
return 0;
0
1
2
/dev/tty
/tmp/a.txt
/tmp/b.txt

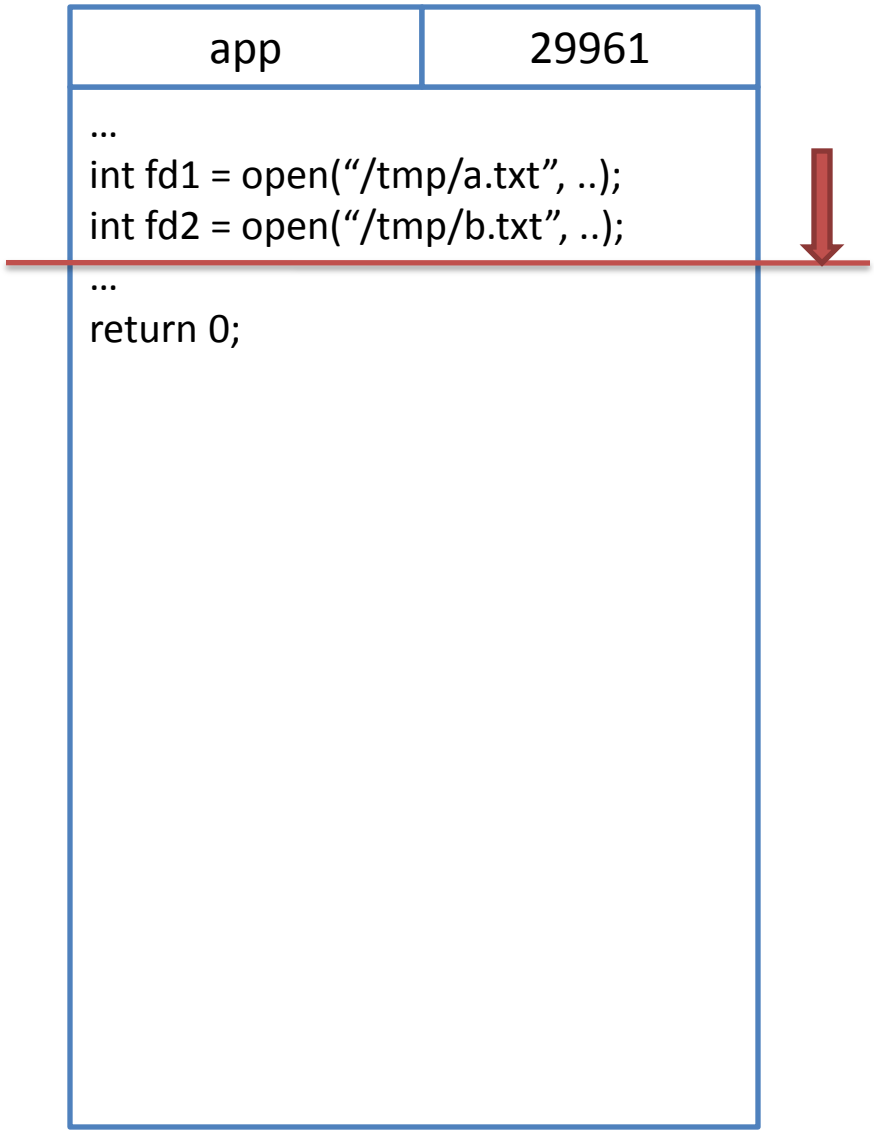


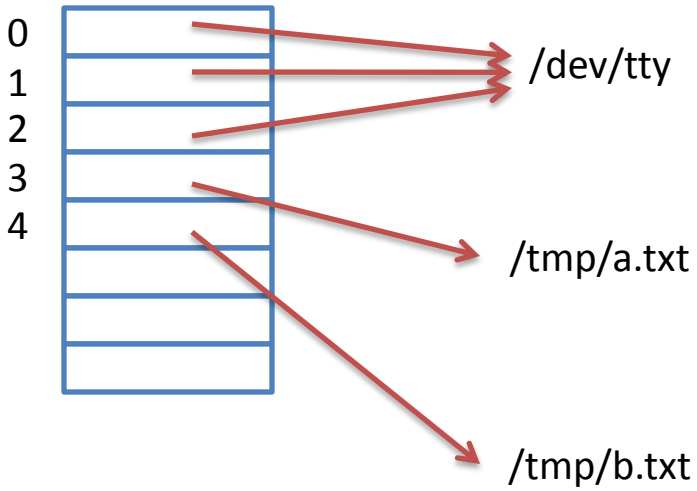
Initially, there are three file descs for stdin(0), stdout(1), stderr(2)





Initially, there are three file descs for stdin(0), stdout(1), stderr(2)

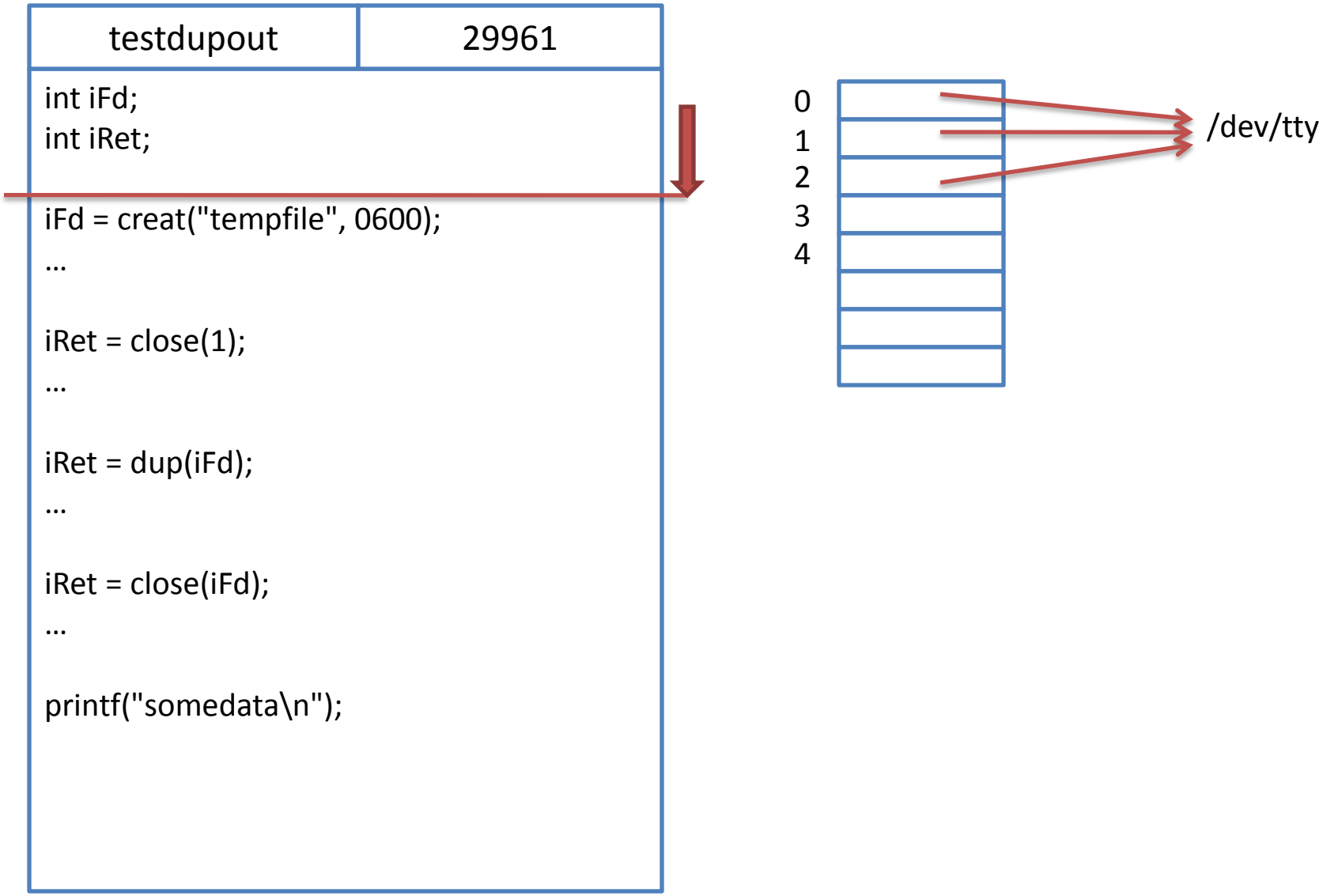


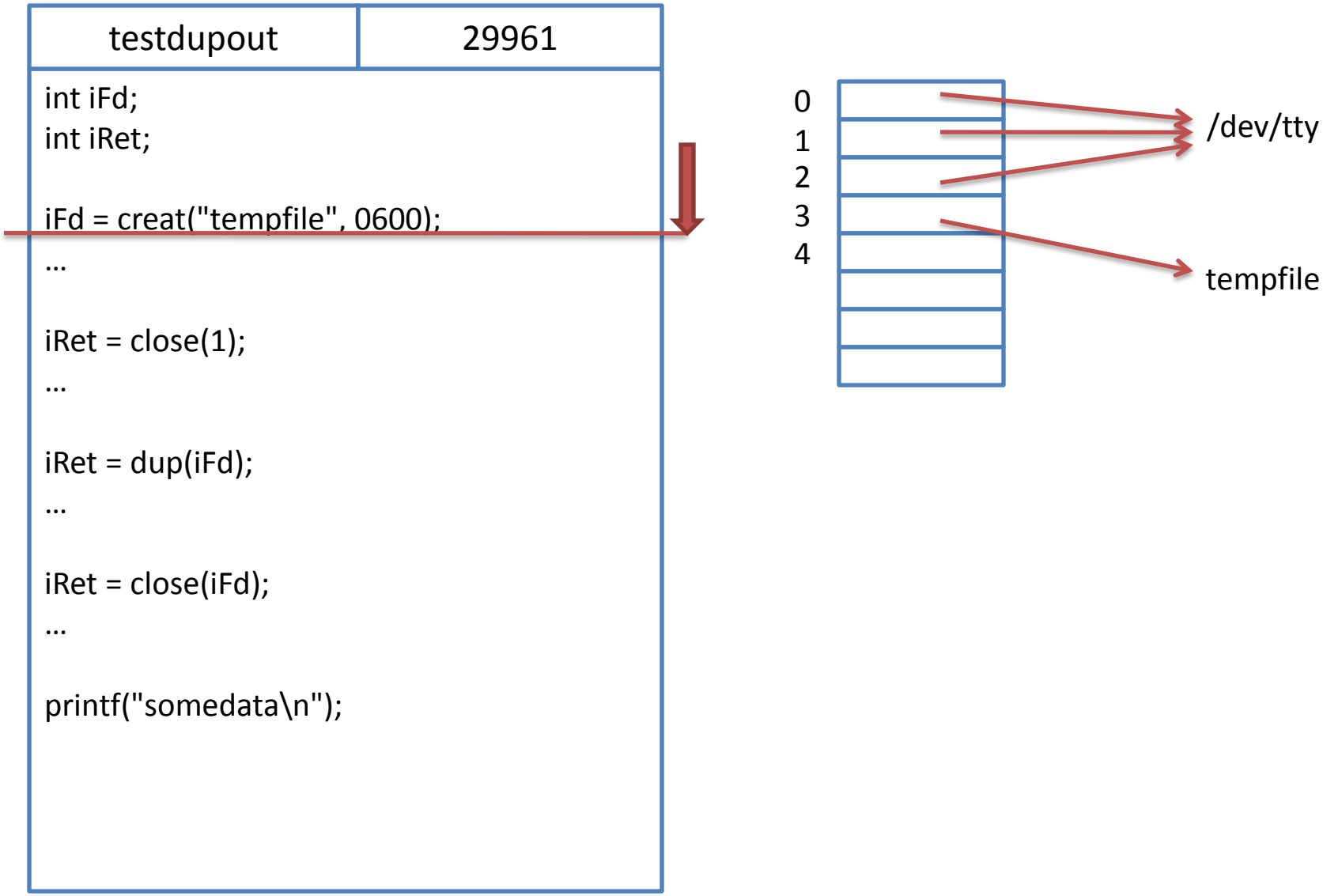


Initially, there are three file descs for stdin(0), stdout(1), stderr(2)

Now,
fd1 should be 3
fd2 should be 4

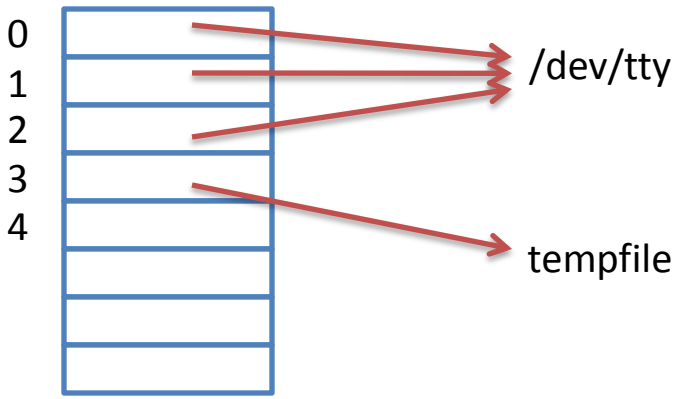
- How can we redirect stdout?
  - Let’s look at ‘testdupout.c’


testdupout
29961
int iFd;
int iRet;
iFd = creat("tempfile", 0600);
…
iRet = close(1);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
printf("somedata\n");
0
1
2
3
4
/dev/tty


testdupout
29961
int iFd;
int iRet;
iFd = creat("tempfile", 0600);
…
iRet = close(1);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
printf("somedata\n");
0
1
2
3
4
/dev/tty
tempfile

testdupout	29961

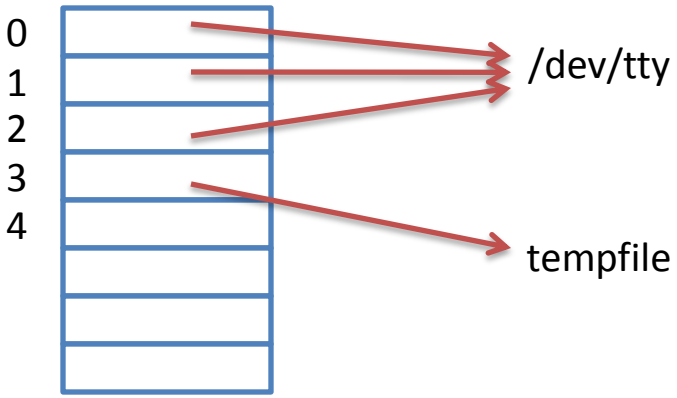
```
int iFd;
int iRet;

iFd = creat("tempfile", 0600);
…

iRet = close(1);
…

iRet = dup(iFd);
…

iRet = close(iFd);
…

printf("somedata\n");
```



create(…, 0600) is what fopen(…, “w”) calls
creates a new file
rewrites file if it exists
returns a file descriptr (e.g., 3)

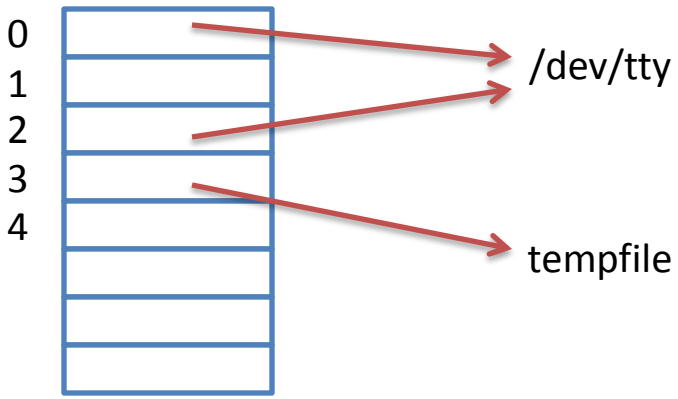
testdupout	29961

```
int iFd;
int iRet;

iFd = creat("tempfile", 0600);
…

iRet = close(1);
…

iRet = dup(iFd);
…

iRet = close(iFd);
…

printf("somedata\n");
```



testdupout	29961

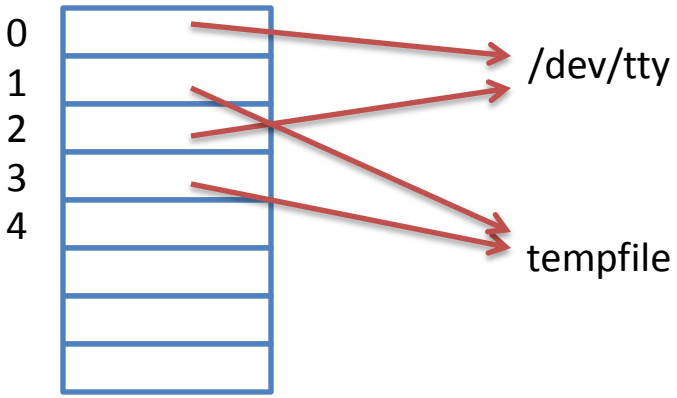
```
int iFd;
int iRet;

iFd = creat("tempfile", 0600);
…

iRet = close(1);
…

iRet = dup(iFd);
…

iRet = close(iFd);
…

printf("somedata\n");
```



close() is what fclose() calls
breaks the connection
frees the file descriptor

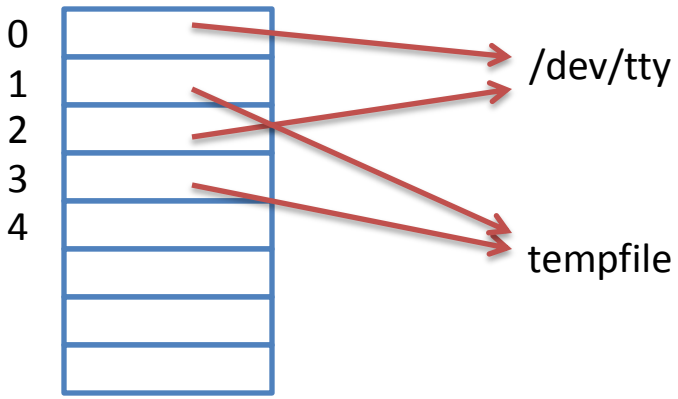
testdupout	29961

```
int iFd;
int iRet;

iFd = creat("tempfile", 0600);
…

iRet = close(1);
…

iRet = dup(iFd);
…

iRet = close(iFd);
…

printf("somedata\n");
```



testdupout	29961

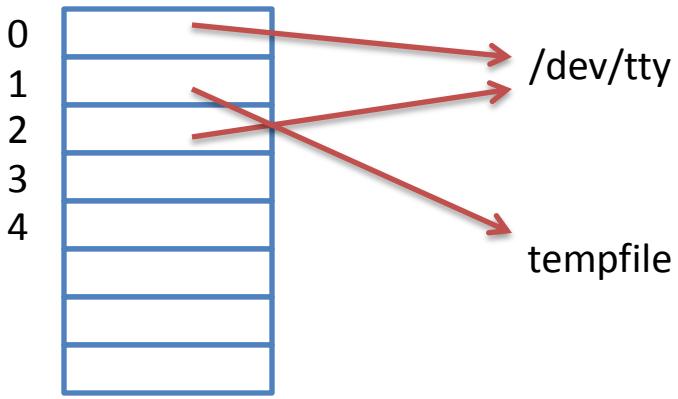
```
int iFd;
int iRet;

iFd = creat("tempfile", 0600);
…

iRet = close(1);
…

iRet = dup(iFd);
…

iRet = close(iFd);
…

printf("somedata\n");
```



dup() duplicates given file descriptor
find first unused element in the fd table
used to redirect stdin or stdout

testdupout	29961

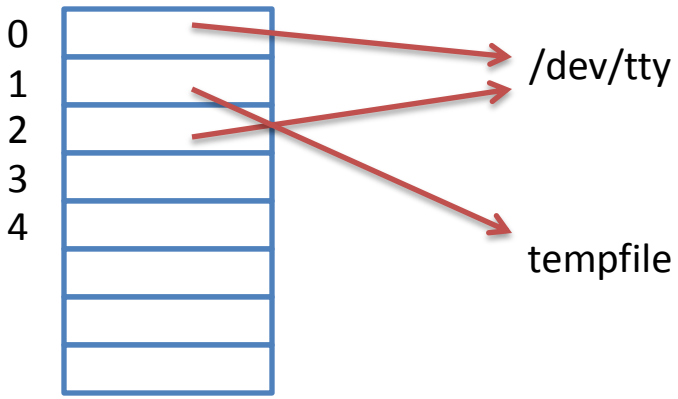
```
int iFd;
int iRet;

iFd = creat("tempfile", 0600);
…

iRet = close(1);
…

iRet = dup(iFd);
…

iRet = close(iFd);
…

printf("somedata\n");
```



Now, we have redirected stdin to temp file

testdupout	29961

```
int iFd;
int iRet;

iFd = creat("tempfile", 0600);
…

iRet = close(1);
…

iRet = dup(iFd);
…

iRet = close(iFd);
…

printf("somedata\n");
```

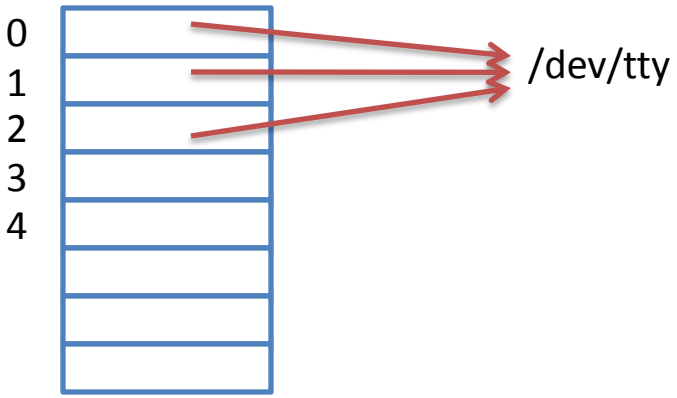


It will write “somedata” into file “tempfile”, not to screen!

- How can we redirect stdin?
  - Let’s look at ‘testdupin.c’

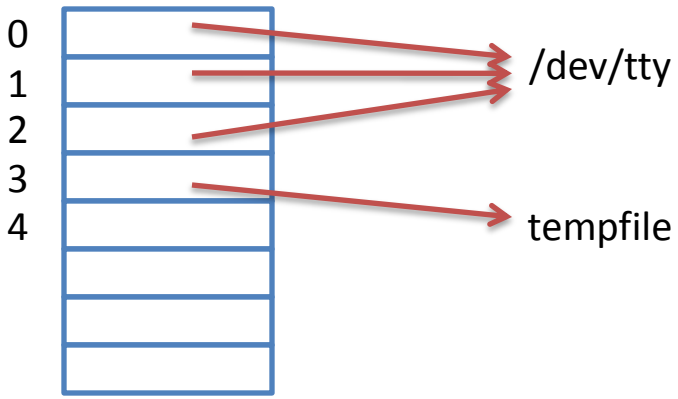
testdupin	29961

```
int iFd;
int iRet;
char pcBuffer[BUFFER_LENGTH];

iFd = open("tempfile", O_RDONLY);
…
iRet = close(0);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
scanf("%s", pcBuffer);
printf("%s\n", pcBuffer);
return 0;
```

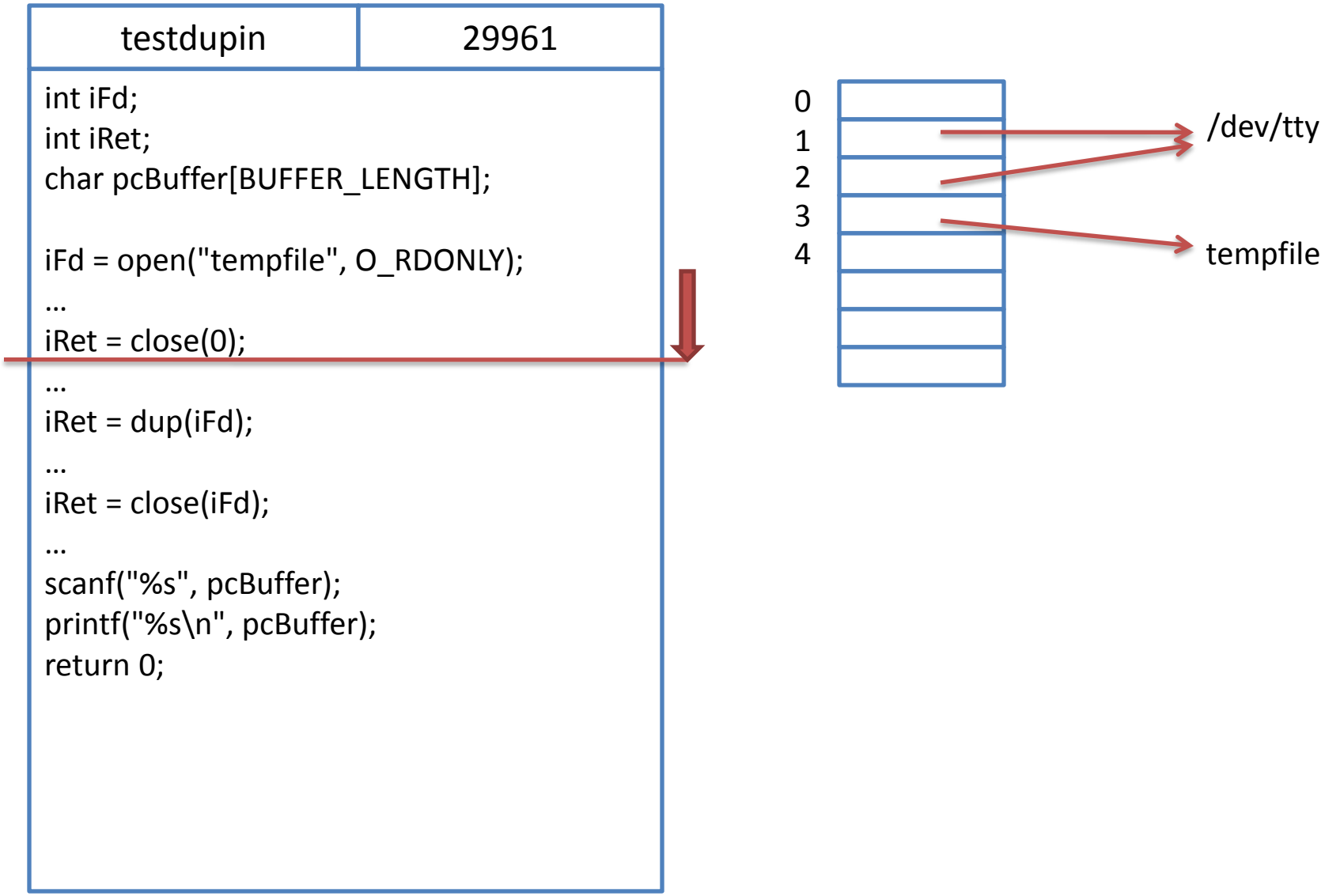


testdupin	29961

```
int iFd;
int iRet;
char pcBuffer[BUFFER_LENGTH];

iFd = open("tempfile", O_RDONLY);
...
iRet = close(0);
...
iRet = dup(iFd);
...
iRet = close(iFd);
...
scanf("%s", pcBuffer);
printf("%s\n", pcBuffer);
return 0;
```

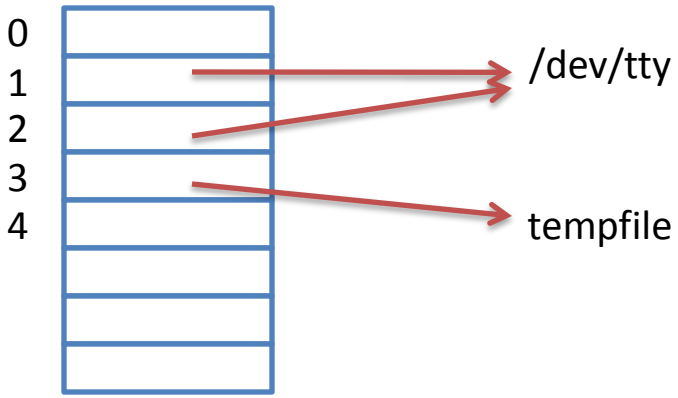


open() underlies fopen(…, “r”)
opens a file for reading or/and writing
returns a file desc (e.g., 3)


testdupin
29961
int iFd;
int iRet;
char pcBuffer[BUFFER_LENGTH];
iFd = open("tempfile", O_RDONLY);
…
iRet = close(0);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
scanf("%s", pcBuffer);
printf("%s\n", pcBuffer);
return 0;
0
1
2
3
4
/dev/tty
tempfile

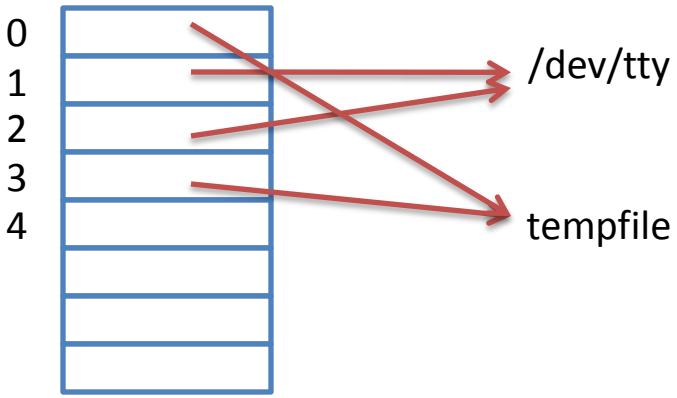
testdupin	29961

```
int iFd;
int iRet;
char pcBuffer[BUFFER_LENGTH];

iFd = open("tempfile", O_RDONLY);
…
iRet = close(0);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
scanf("%s", pcBuffer);
printf("%s\n", pcBuffer);
return 0;
```



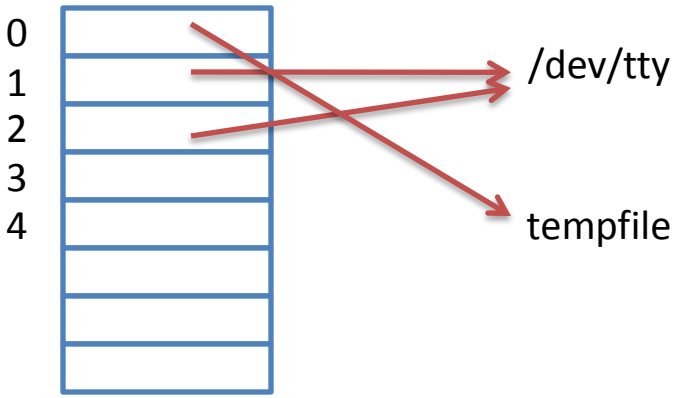
testdupin	29961

```
int iFd;
int iRet;
char pcBuffer[BUFFER_LENGTH];

iFd = open("tempfile", O_RDONLY);
…
iRet = close(0);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
scanf("%s", pcBuffer);
printf("%s\n", pcBuffer);
return 0;
```



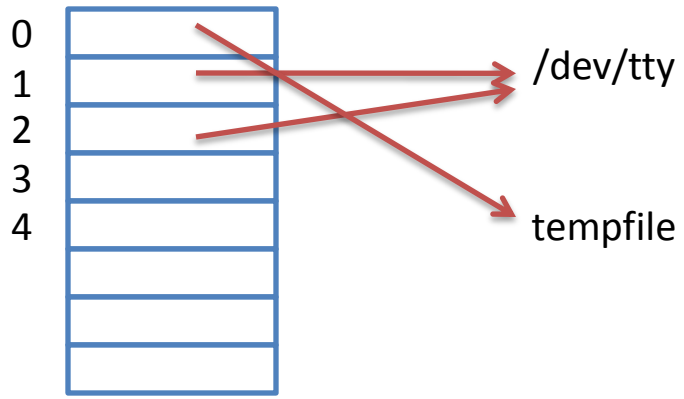
testdupin	29961

```
int iFd;
int iRet;
char pcBuffer[BUFFER_LENGTH];

iFd = open("tempfile", O_RDONLY);
…
iRet = close(0);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
scanf("%s", pcBuffer);
printf("%s\n", pcBuffer);
return 0;
```



testdupin	29961

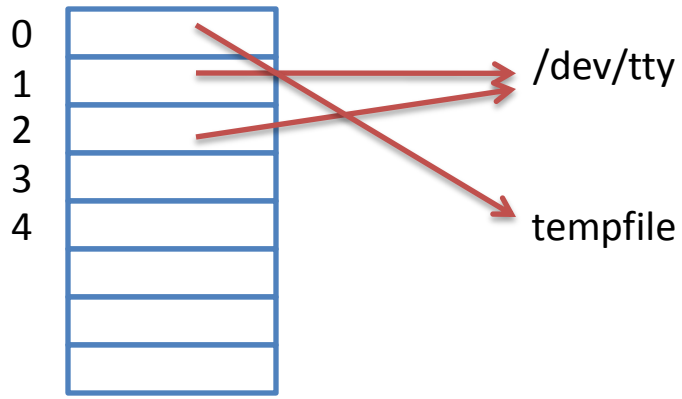
```
int iFd;
int iRet;
char pcBuffer[BUFFER_LENGTH];

iFd = open("tempfile", O_RDONLY);
…
iRet = close(0);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
scanf("%s", pcBuffer);
printf("%s\n", pcBuffer);
return 0;
```



Now, we've redirected stdin to tempfile

testdupin	29961

```
int iFd;
int iRet;
char pcBuffer[BUFFER_LENGTH];

iFd = open("tempfile", O_RDONLY);
…
iRet = close(0);
…
iRet = dup(iFd);
…
iRet = close(iFd);
…
scanf("%s", pcBuffer);
printf("%s\n", pcBuffer);
return 0;
```

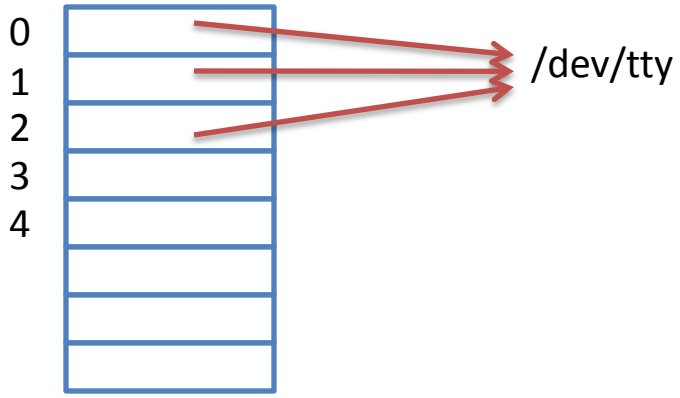


It will read a string from “tempfile”, not from keyboard

- How can we redirect stdin or stdout in child process??
- Let’s look at ‘testdupforkexec.c’

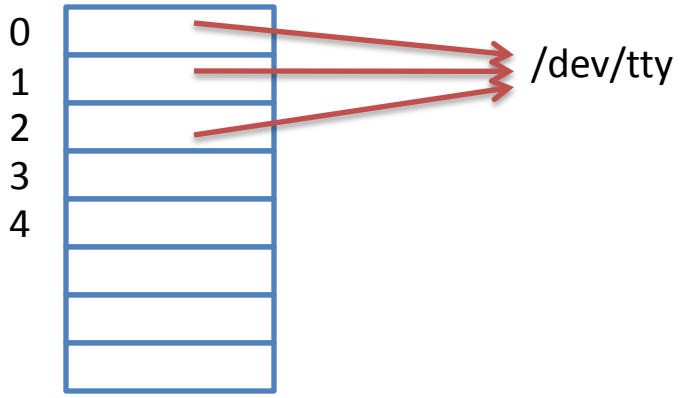
testdupforkexec	29961

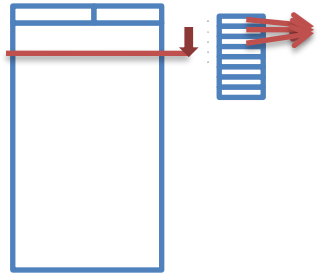
```
pid_t iPid;
 …
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  …
  iRet = close(1);
  …
  iRet = dup(iFd);
  …
  iRet = close(iFd);
  …
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  …
}
…
iPid = wait(NULL);
…
return 0;
```



testdupforkexec	29961

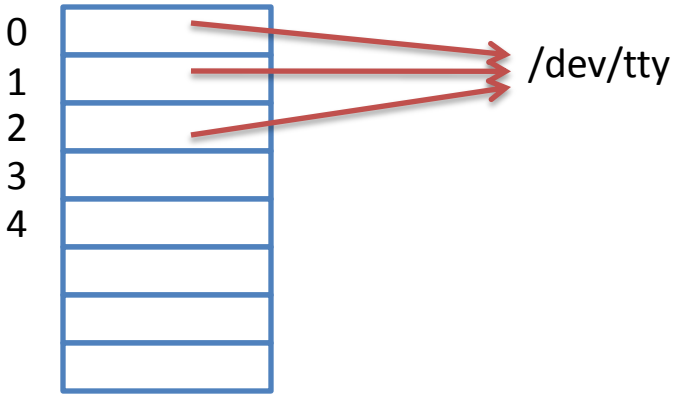
```
pid_t iPid;
 ...
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  ...
  iRet = close(1);
  ...
  iRet = dup(iFd);
  ...
  iRet = close(iFd);
  ...
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  ...
}
...
iPid = wait(NULL);
...
return 0;
```

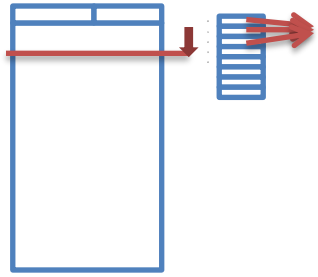


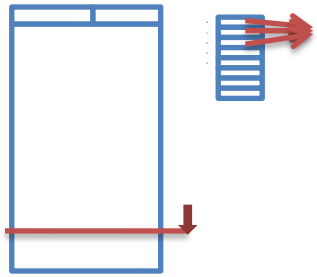


testdupforkexec	29961

```
pid_t iPid;
 ...
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  ...
  iRet = close(1);
  ...
  iRet = dup(iFd);
  ...
  iRet = close(iFd);
  ...
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  ...
}
...
iPid = wait(NULL);
...
return 0;
```

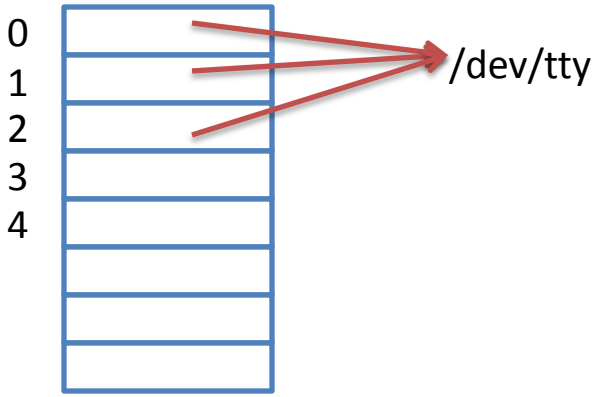


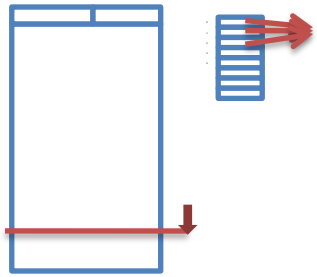




testdupforkexec	29962

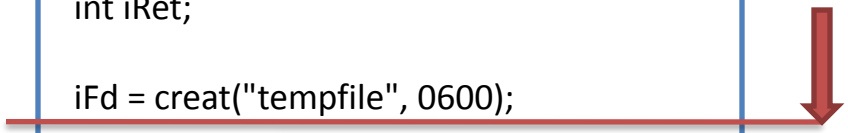
```
pid_t iPid;
 …
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  …
  iRet = close(1);
  …
  iRet = dup(iFd);
  …
  iRet = close(iFd);
  …
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  …
}
…
iPid = wait(NULL);
…
return 0;
```

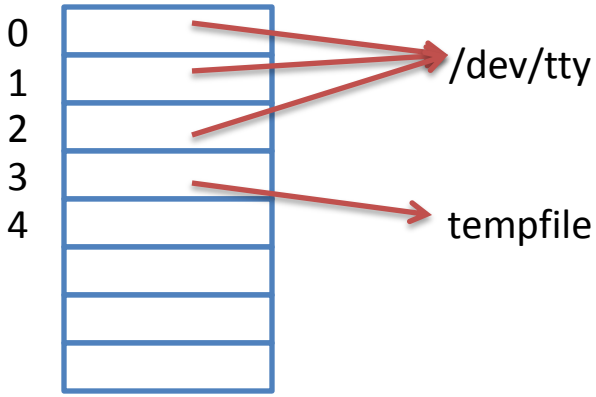


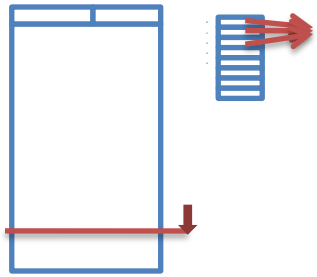


testdupforkexec	29962

```
pid_t iPid;
 …
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  …
  iRet = close(1);
  …
  iRet = dup(iFd);
  …
  iRet = close(iFd);
  …
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  …
}
…
iPid = wait(NULL);
…
return 0;
```

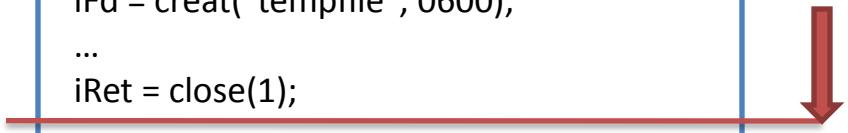


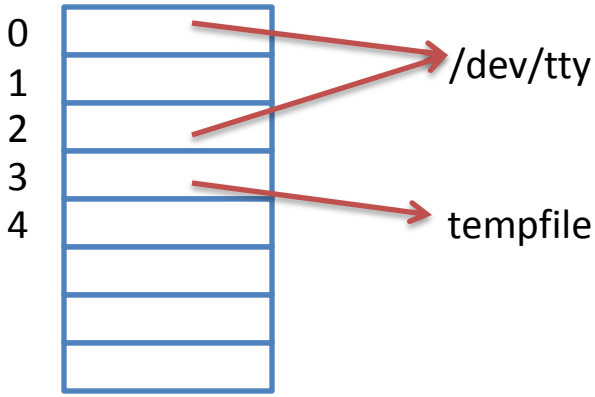


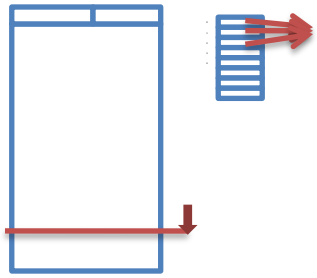


testdupforkexec | 29962

```
pid_t iPid;
 …
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  …
  iRet = close(1);
  …
  iRet = dup(iFd);
  …
  iRet = close(iFd);
  …
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  …
}
…
iPid = wait(NULL);
…
return 0;
```

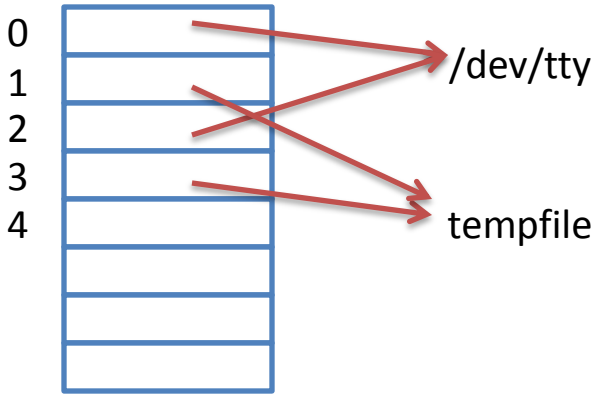


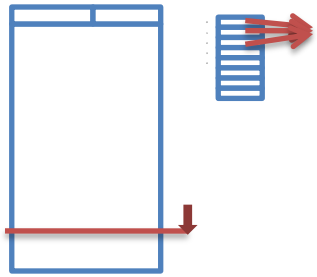




testdupforkexec | 29962

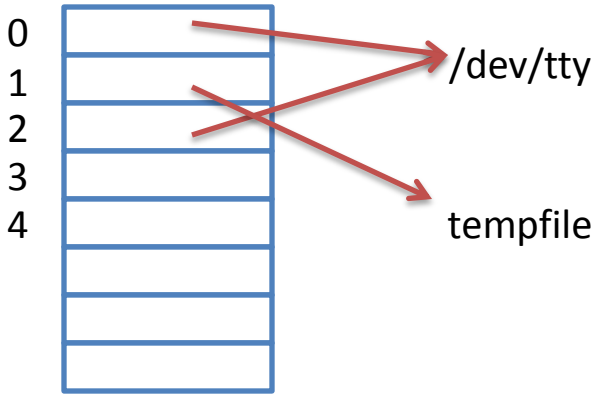
```
pid_t iPid;
 …
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  …
  iRet = close(1);
  …
  iRet = dup(iFd);
  …
  iRet = close(iFd);
  …
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  …
}
…
iPid = wait(NULL);
…
return 0;
```

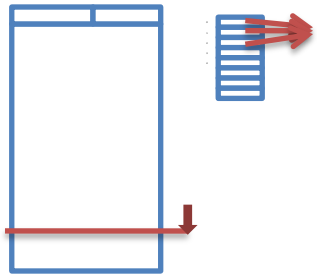




testdupforkexec	29962

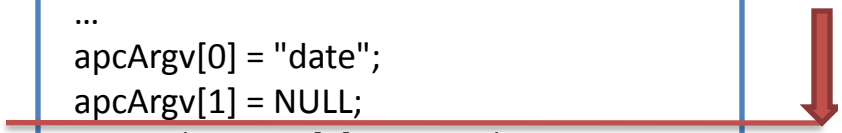
```
pid_t iPid;
 …
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  …
  iRet = close(1);
  …
  iRet = dup(iFd);
  …
  iRet = close(iFd);
  …
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  …
}
…
iPid = wait(NULL);
…
return 0;
```

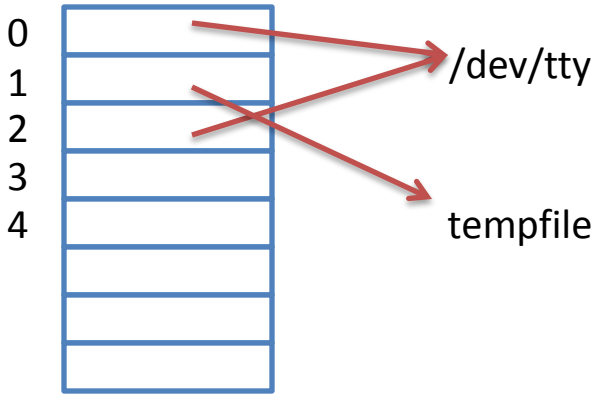


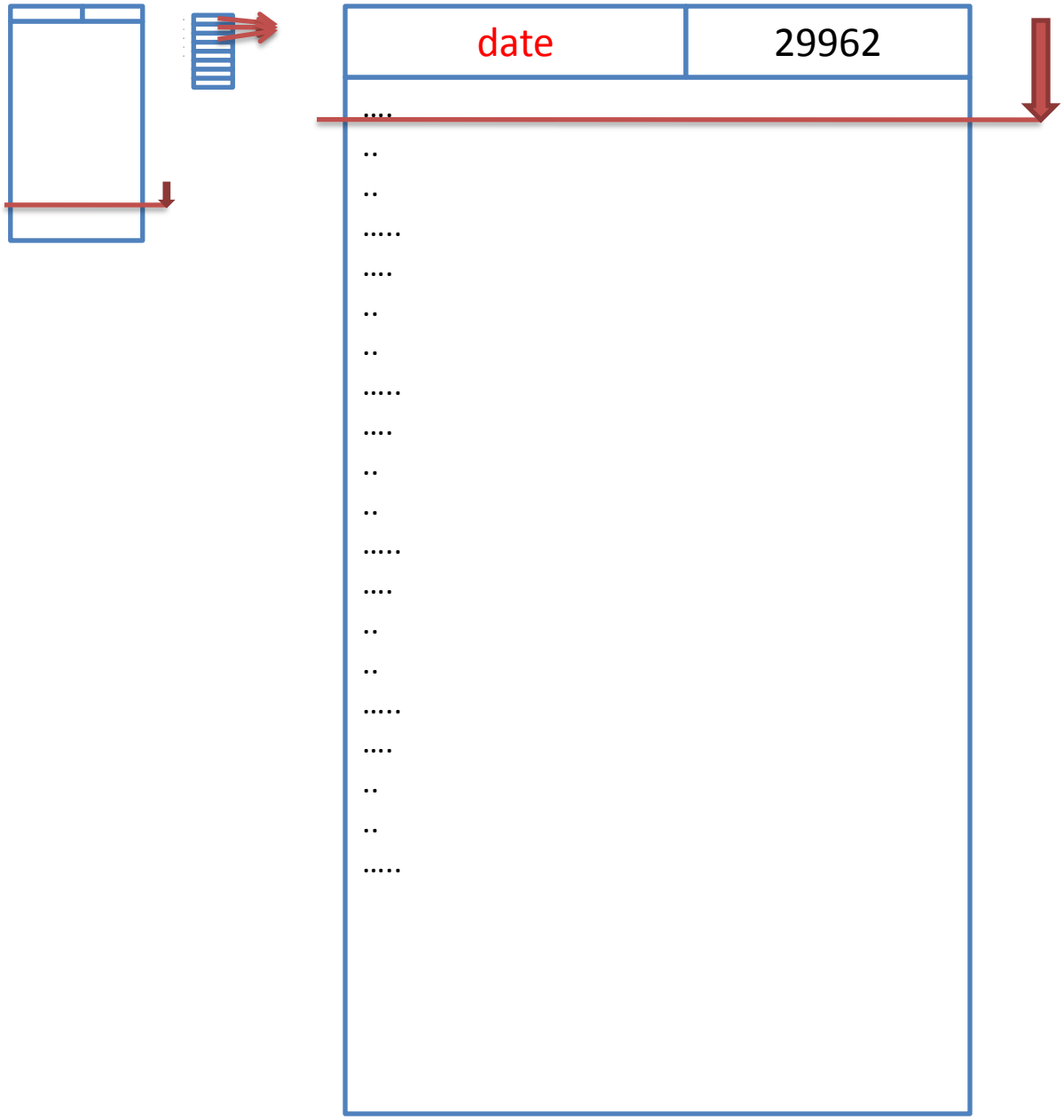


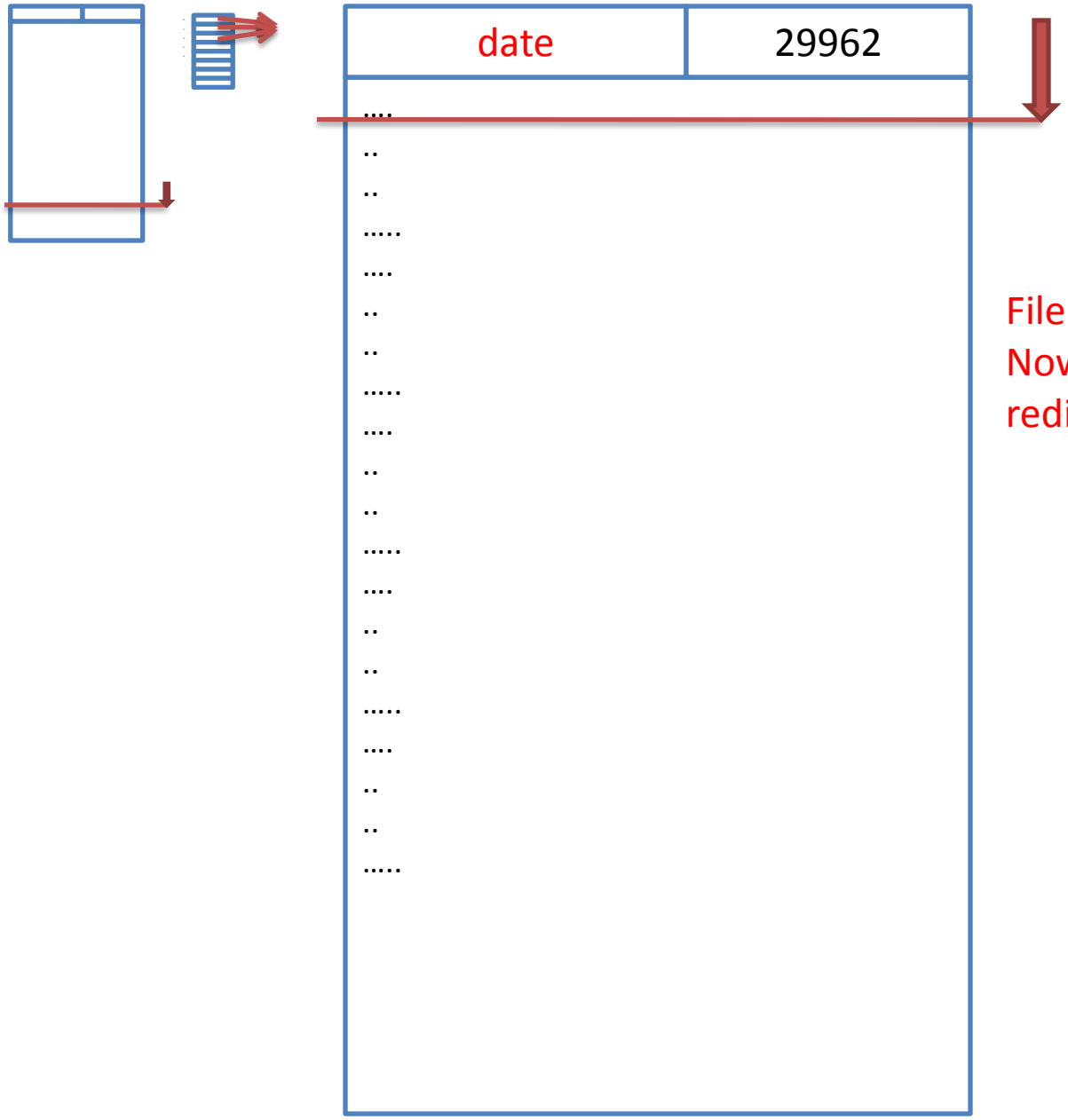
testdupforkexec	29962

```
pid_t iPid;
 …
iPid = fork();
if (iPid == 0) {
  char *apcArgv[2];
  int iFd;
  int iRet;

  iFd = creat("tempfile", 0600);
  …
  iRet = close(1);
  …
  iRet = dup(iFd);
  …
  iRet = close(iFd);
  …
  apcArgv[0] = "date";
  apcArgv[1] = NULL;
  execvp(apcArgv[0], apcArgv);
  …
}
…
iPid = wait(NULL);
…
return 0;
```

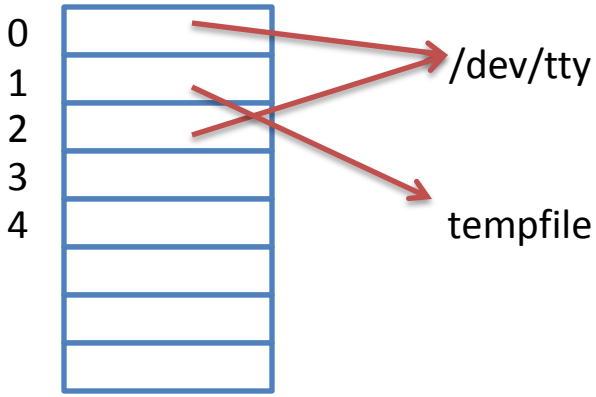





date
29962
0
1
2
3
4
/dev/tty
tempfile



File descriptor table survives!!
Now, the stdout of ‘date’ has been redirected to “tempfile”



Process Control

- How can we quit a runaway process? (e.g., infloop)
  - Type Ctrl+C

Signal

- Unix sends a signal to your process
  - Ctrl+C => 2/SIGINT (Run `kill –l` on bash for more info)
- A signal handler for SIGINT will be called
  - A signal handler is a function
- Or default signal handler will be called
  - Default signal handler for SIGINT will exit the process

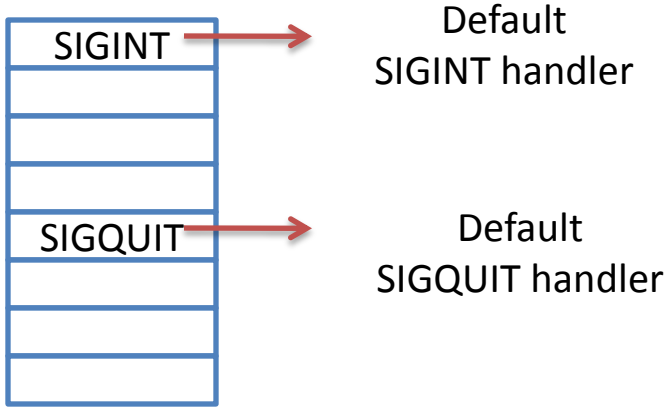
Signal

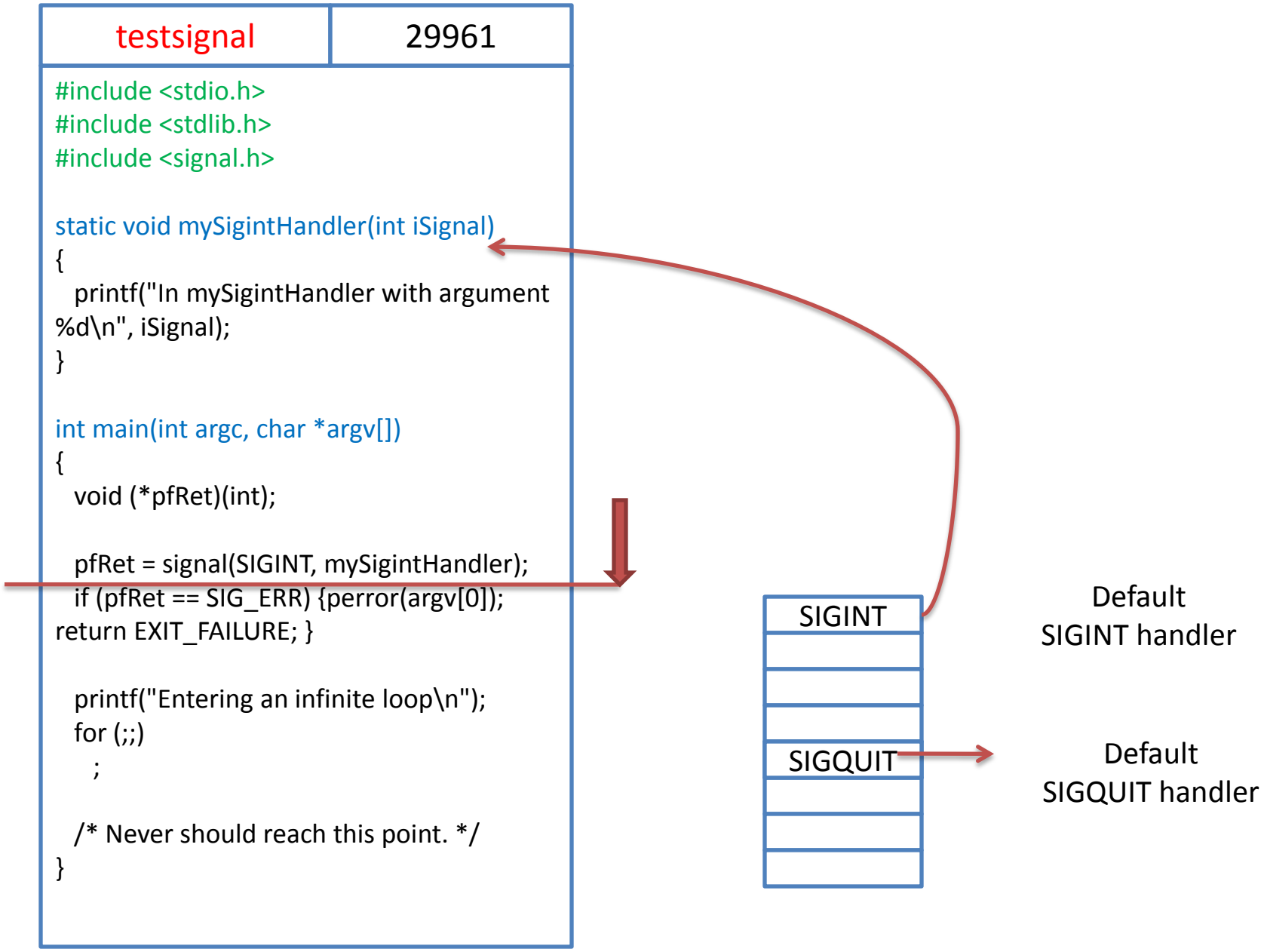
- You can use other signals to kill the process
  - Ctrl+\ => 3/SIGQUIT
- You can use SIGQUIT when the process overrides SIGINT

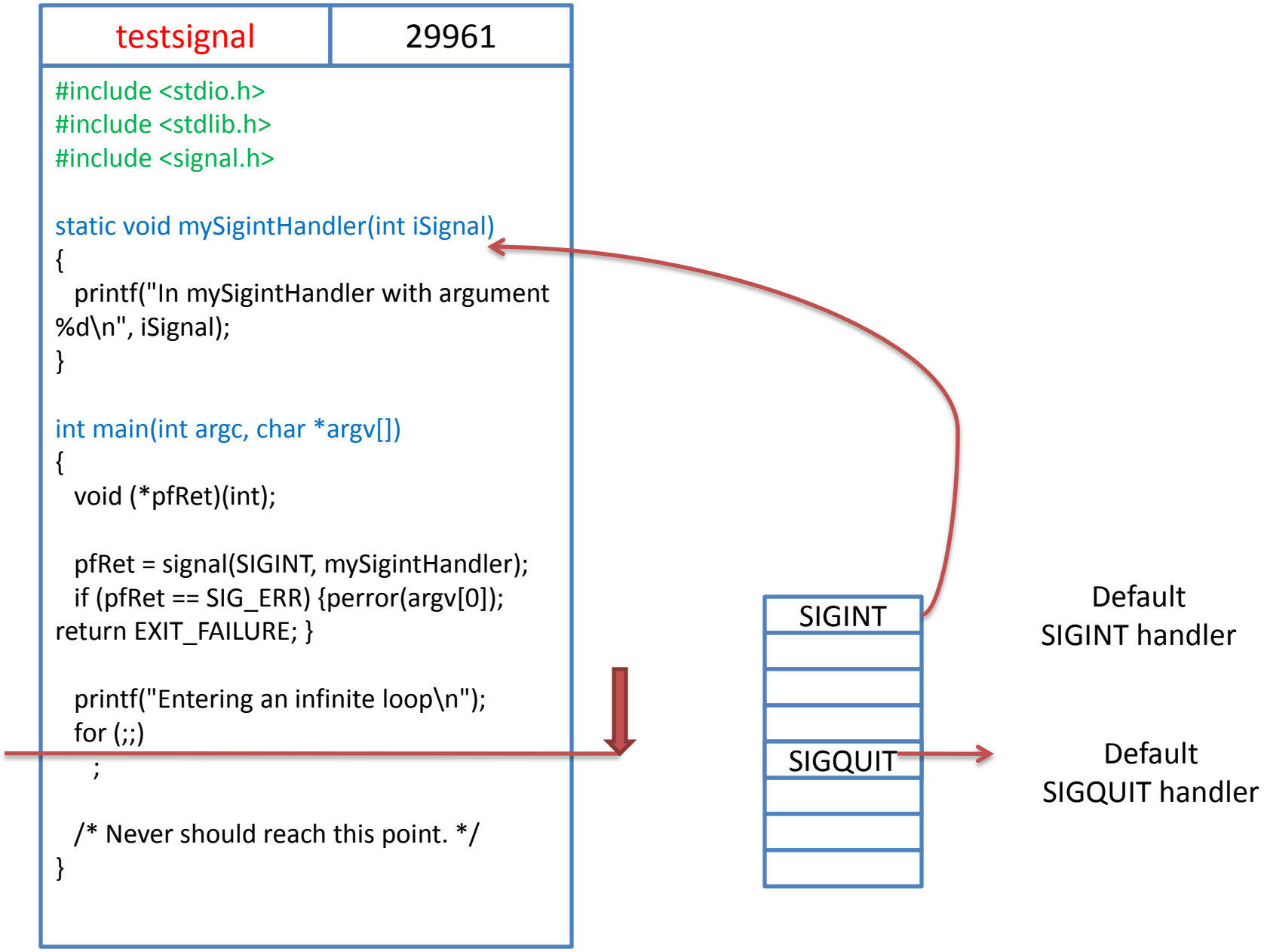
- Why do we need to care about signals??
  - Shell should not exit when user types Ctrl+C
- Let’s look at ‘testsignal.c’

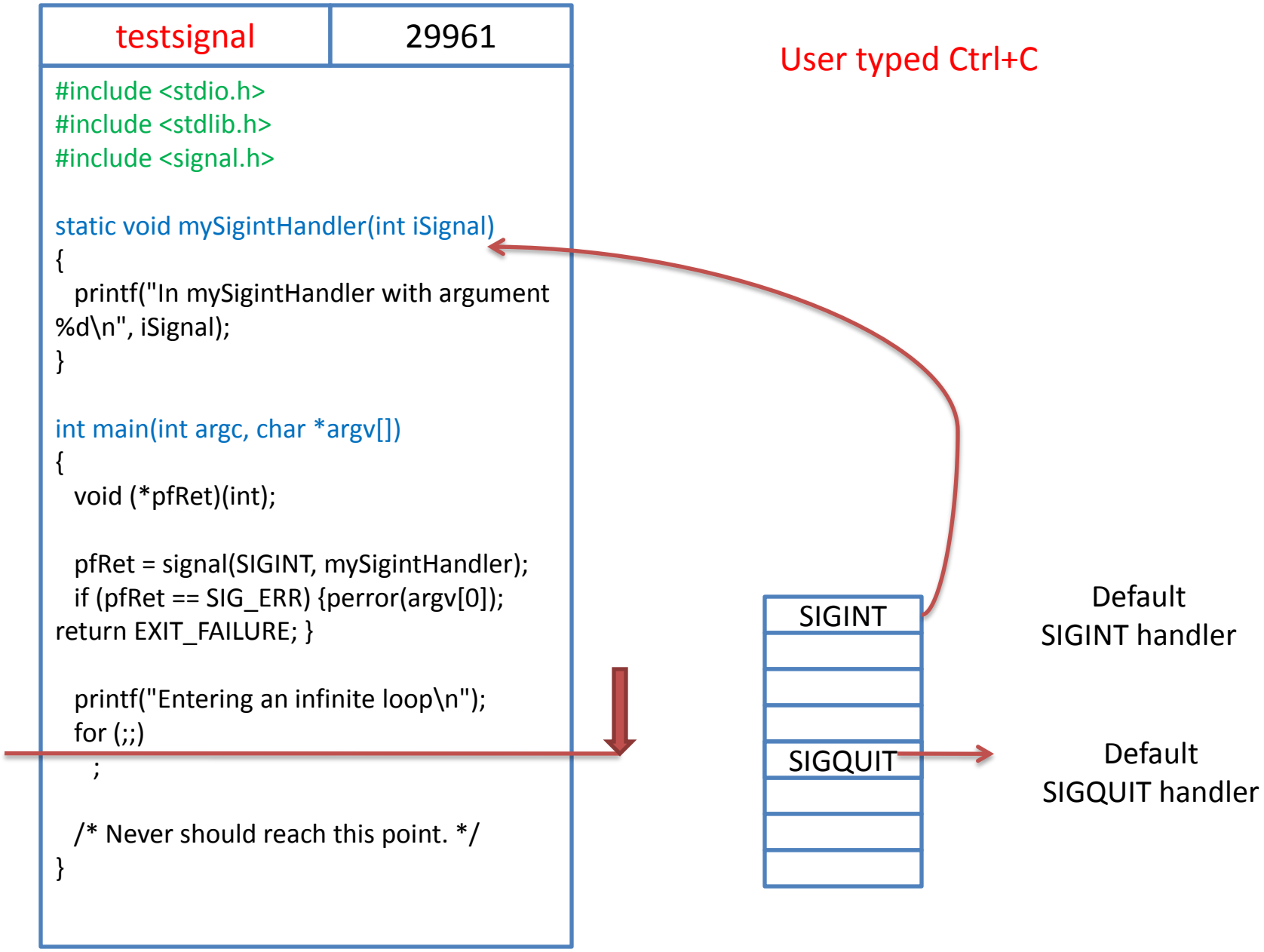
testsignal	29961

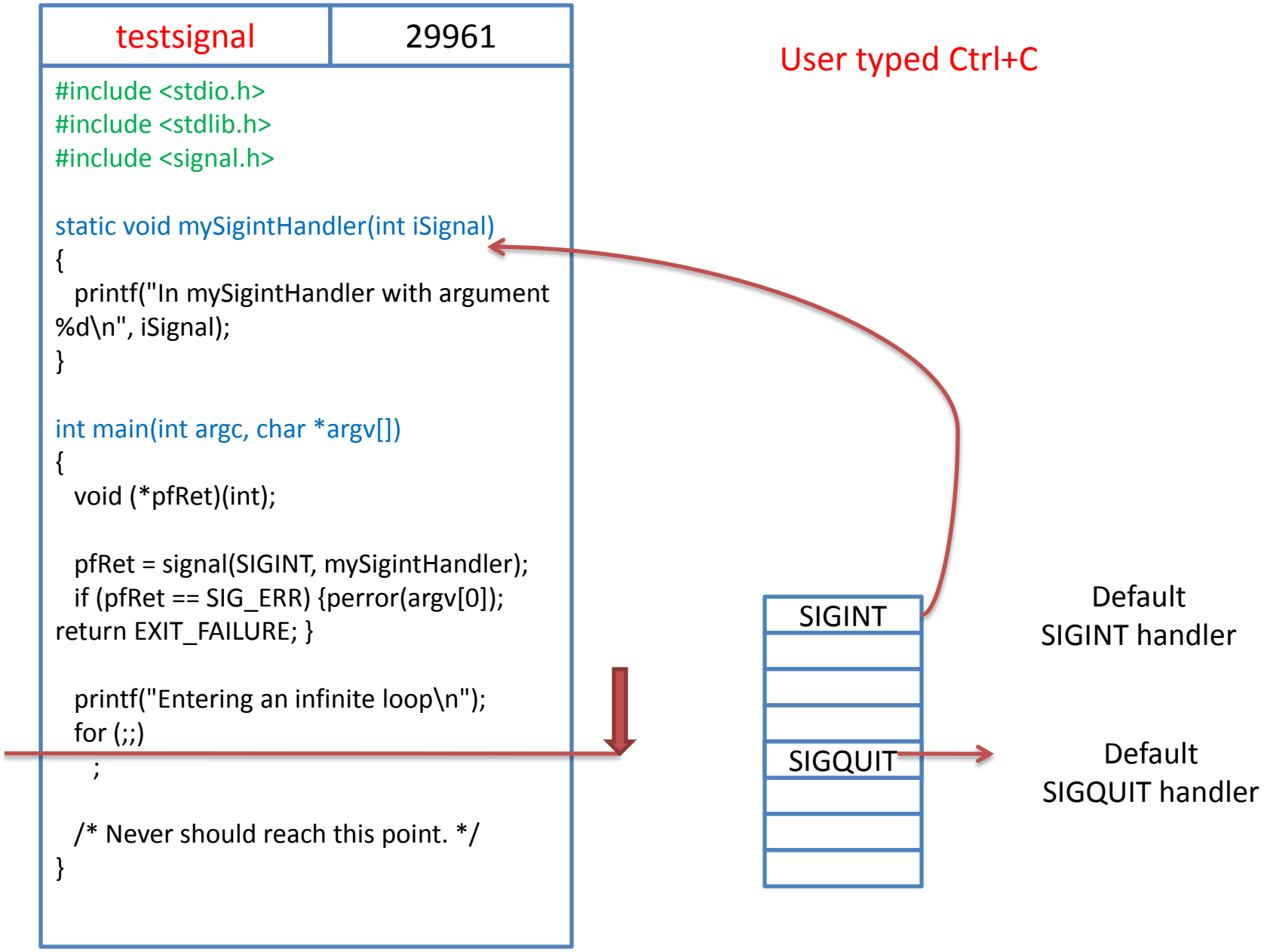
```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

static void mySigintHandler(int iSignal)
{
  printf("In mySigintHandler with argument
%d\n", iSignal);
}

int main(int argc, char *argv[])
{
  void (*pfRet)(int);

  pfRet = signal(SIGINT, mySigintHandler);
  if (pfRet == SIG_ERR) {perror(argv[0]);
return EXIT_FAILURE; }

  printf("Entering an infinite loop\n");
  for (;;)
    ;

  /* Never should reach this point. */
}
```

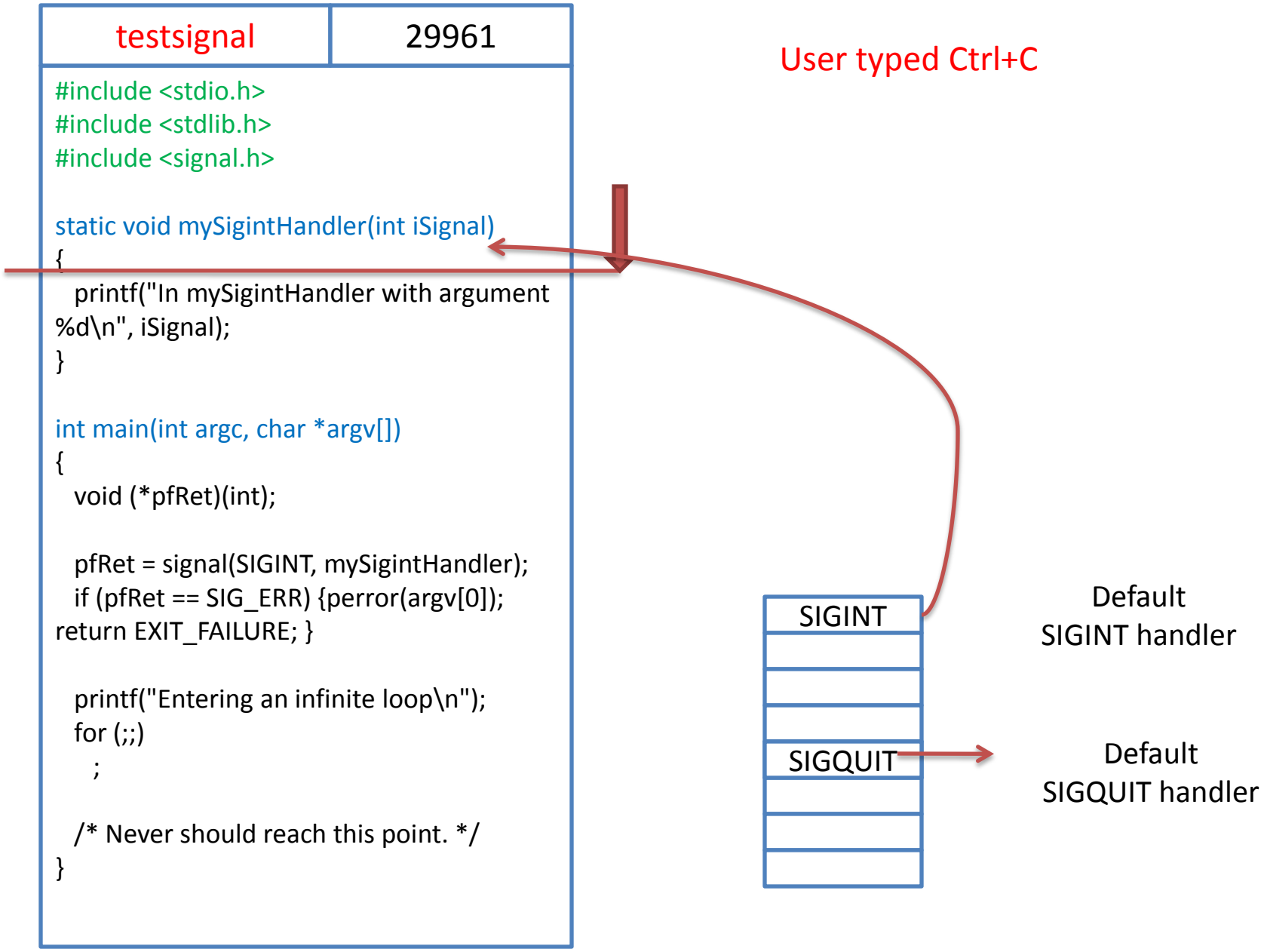


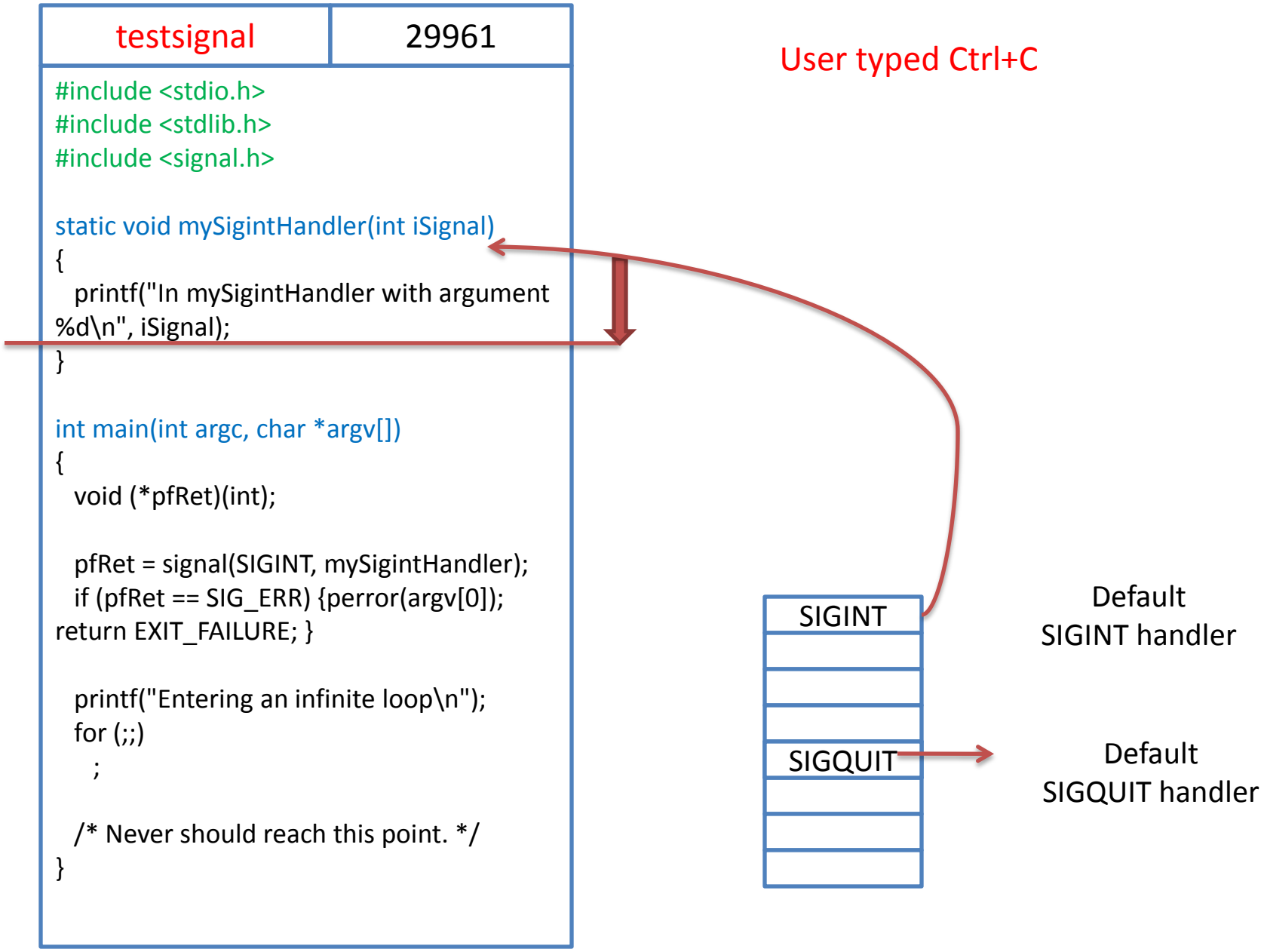

testsignal
29961
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void mySigintHandler(int iSignal)
{
printf("In mySigintHandler with argument %d\n", iSignal);
}
int main(int argc, char *argv[])
{
void (*pfRet)(int);
pfRet = signal(SIGINT, mySigintHandler);
if (pfRet == SIG_ERR) {perror(argv[0]); return EXIT_FAILURE; }
printf("Entering an infinite loop\n");
for (;;)
;
/* Never should reach this point. */
}
SIGINT
SIGQUIT
Default SIGINT handler
Default SIGQUIT handler

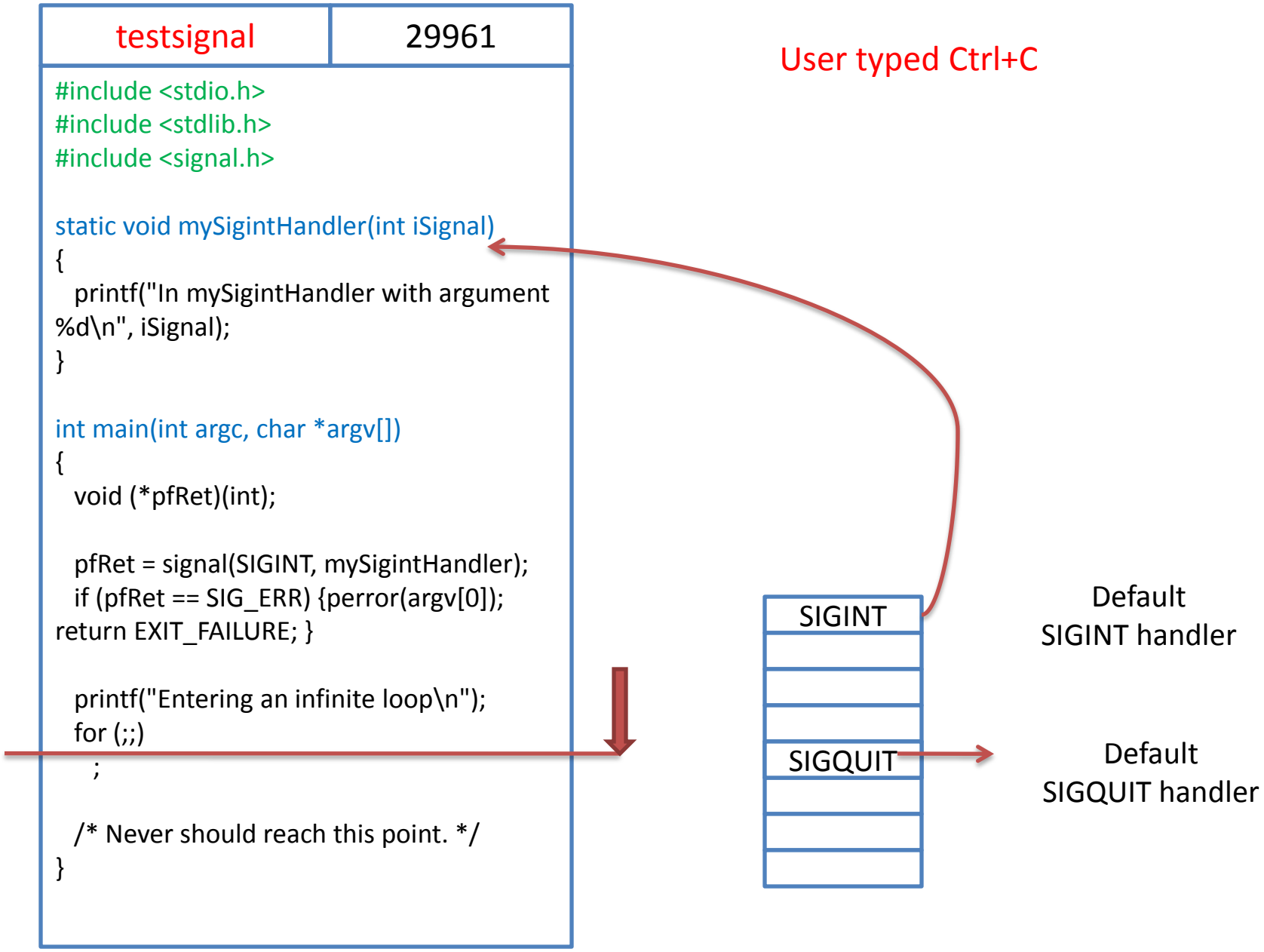

testsignal
29961
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void mySigintHandler(int iSignal)
{
printf("In mySigintHandler with argument %d\n", iSignal);
}
int main(int argc, char *argv[])
{
void (*pfRet)(int);
pfRet = signal(SIGINT, mySigintHandler);
if (pfRet == SIG_ERR) {perror(argv[0]); return EXIT_FAILURE; }
printf("Entering an infinite loop\n");
for (;;)
;
/* Never should reach this point. */
}
SIGINT
SIGQUIT
Default SIGINT handler
Default SIGQUIT handler

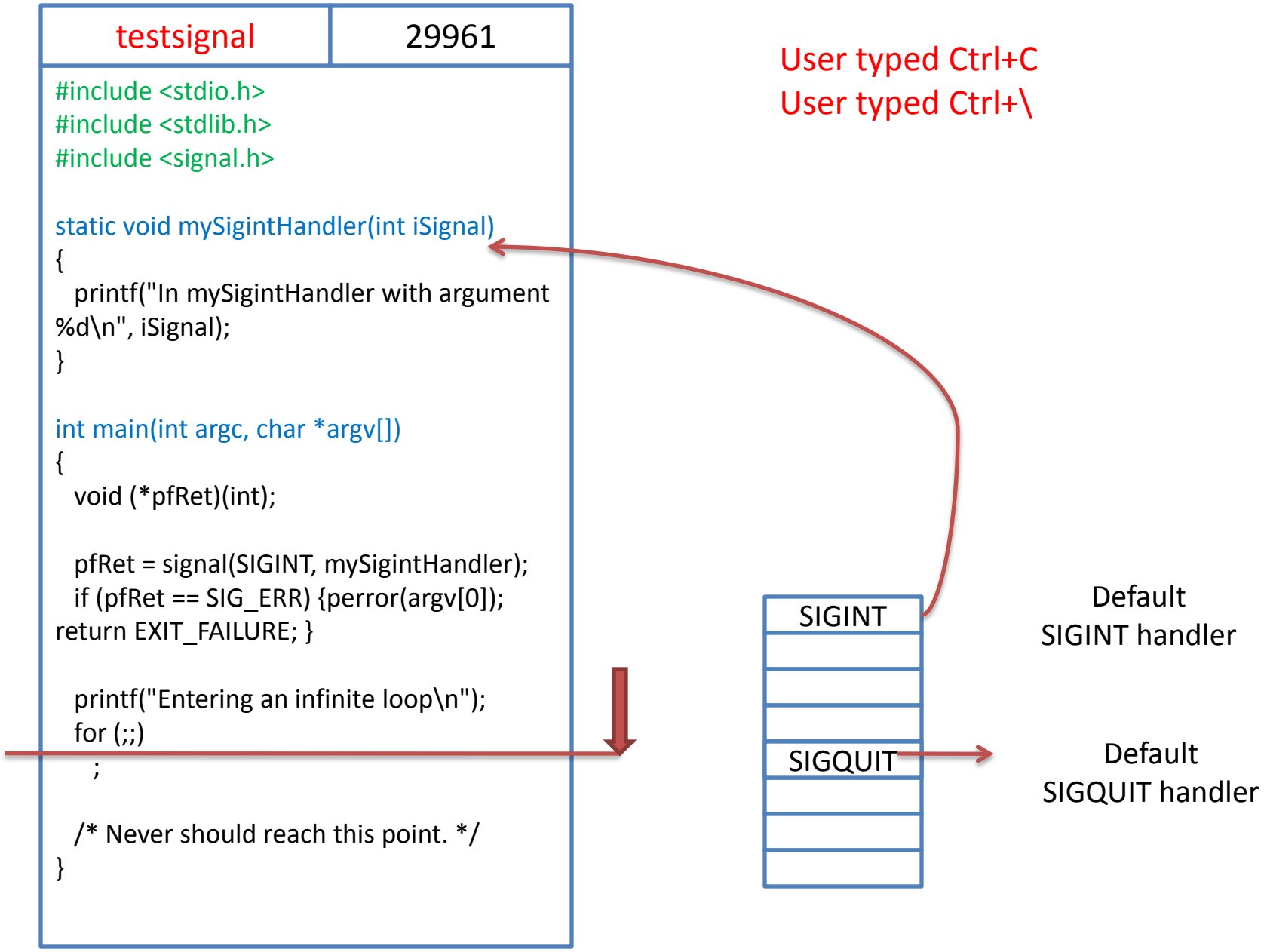

testsignal
29961
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void mySigintHandler(int iSignal)
{
printf("In mySigintHandler with argument %d\n", iSignal);
}
int main(int argc, char *argv[])
{
void (*pfRet)(int);
pfRet = signal(SIGINT, mySigintHandler);
if (pfRet == SIG_ERR) {perror(argv[0]); return EXIT_FAILURE; }
printf("Entering an infinite loop\n");
for (;;)
;
/* Never should reach this point. */
}
User typed Ctrl+C
SIGINT
SIGQUIT
Default SIGINT handler
Default SIGQUIT handler


testsignal
29961
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void mySigintHandler(int iSignal)
{
printf("In mySigintHandler with argument %d\n", iSignal);
}
int main(int argc, char *argv[])
{
void (*pfRet)(int);
pfRet = signal(SIGINT, mySigintHandler);
if (pfRet == SIG_ERR) {perror(argv[0]); return EXIT_FAILURE; }
printf("Entering an infinite loop\n");
for (;;)
;
/* Never should reach this point. */
}
User typed Ctrl+C
SIGINT
SIGQUIT
Default SIGINT handler
Default SIGQUIT handler


testsignal
29961
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void mySigintHandler(int iSignal)
{
printf("In mySigintHandler with argument %d\n", iSignal);
}
int main(int argc, char *argv[])
{
void (*pfRet)(int);
pfRet = signal(SIGINT, mySigintHandler);
if (pfRet == SIG_ERR) {perror(argv[0]); return EXIT_FAILURE; }
printf("Entering an infinite loop\n");
for (;;)
;
/* Never should reach this point. */
}
User typed Ctrl+C
SIGINT
SIGQUIT
Default SIGINT handler
Default SIGQUIT handler


testsignal
29961
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void mySigintHandler(int iSignal)
{
printf("In mySigintHandler with argument %d\n", iSignal);
}
int main(int argc, char *argv[])
{
void (*pfRet)(int);
pfRet = signal(SIGINT, mySigintHandler);
if (pfRet == SIG_ERR) {perror(argv[0]); return EXIT_FAILURE; }
printf("Entering an infinite loop\n");
for (;;)
;
/* Never should reach this point. */
}
User typed Ctrl+C
SIGINT
SIGQUIT
Default SIGINT handler
Default SIGQUIT handler

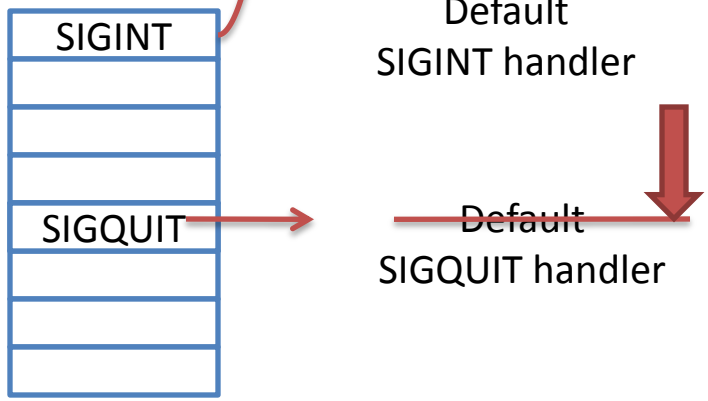

testsignal
29961
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void mySigintHandler(int iSignal)
{
printf("In mySigintHandler with argument %d\n", iSignal);
}
int main(int argc, char *argv[])
{
void (*pfRet)(int);
pfRet = signal(SIGINT, mySigintHandler);
if (pfRet == SIG_ERR) {perror(argv[0]); return EXIT_FAILURE; }
printf("Entering an infinite loop\n");
for (;;)
;
/* Never should reach this point. */
}
User typed Ctrl+C
SIGINT
SIGQUIT
Default SIGINT handler
Default SIGQUIT handler


testsignal
29961
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void mySigintHandler(int iSignal)
{
printf("In mySigintHandler with argument %d\n", iSignal);
}
int main(int argc, char *argv[])
{
void (*pfRet)(int);
pfRet = signal(SIGINT, mySigintHandler);
if (pfRet == SIG_ERR) {perror(argv[0]); return EXIT_FAILURE; }
printf("Entering an infinite loop\n");
for (;;)
;
/* Never should reach this point. */
}
User typed Ctrl+C
User typed Ctrl+\
SIGINT
SIGQUIT
Default SIGINT handler
Default SIGQUIT handler

testsignal	29961

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

static void mySigintHandler(int iSignal)
{
  printf("In mySigintHandler with argument
%d\n", iSignal);
}

int main(int argc, char *argv[])
{
  void (*pfRet)(int);

  pfRet = signal(SIGINT, mySigintHandler);
  if (pfRet == SIG_ERR) {perror(argv[0]);
return EXIT_FAILURE; }

  printf("Entering an infinite loop\n");
  for (;;)
    ;

  /* Never should reach this point. */
}
```

User typed Ctrl+C
User typed Ctrl+\

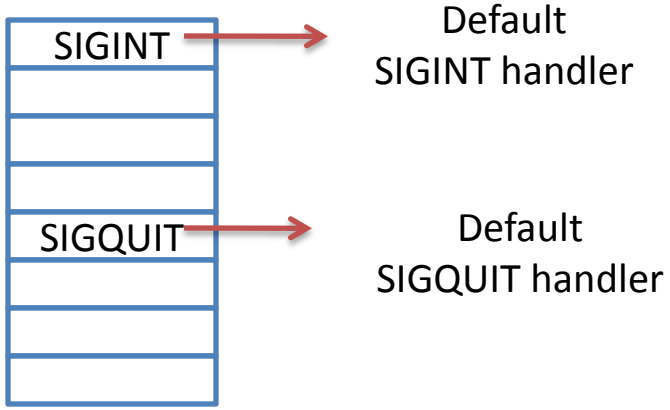


- We can simply ignore a signal
  - By using some predefined handlers
  - SIG_IGN
  - SIG_DFL

testsingalignore	29961

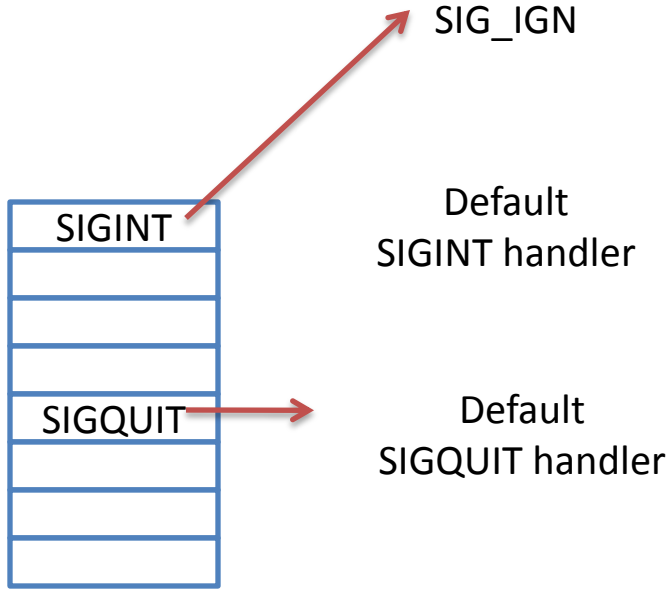
```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int main(int argc, char *argv[])
{
  void (*pfRet)(int);

  pfRet = signal(SIGINT, SIG_IGN);
  if (pfRet == SIG_ERR) {perror(argv[0]);
return EXIT_FAILURE; }

  for (;;)
    ;

  /* Never should reach this point. */
}
```

SIG_IGN



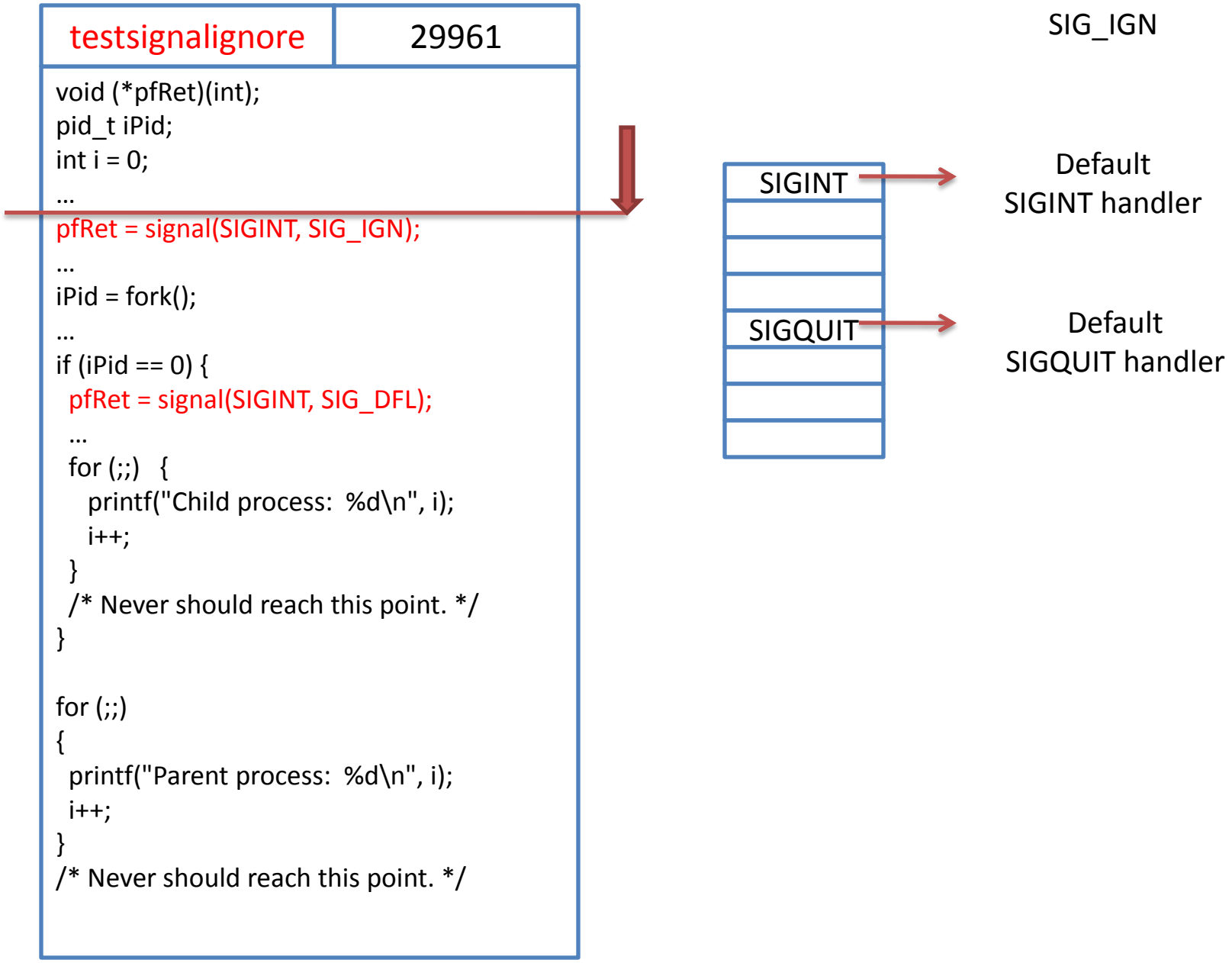
testsingalignore | 29961

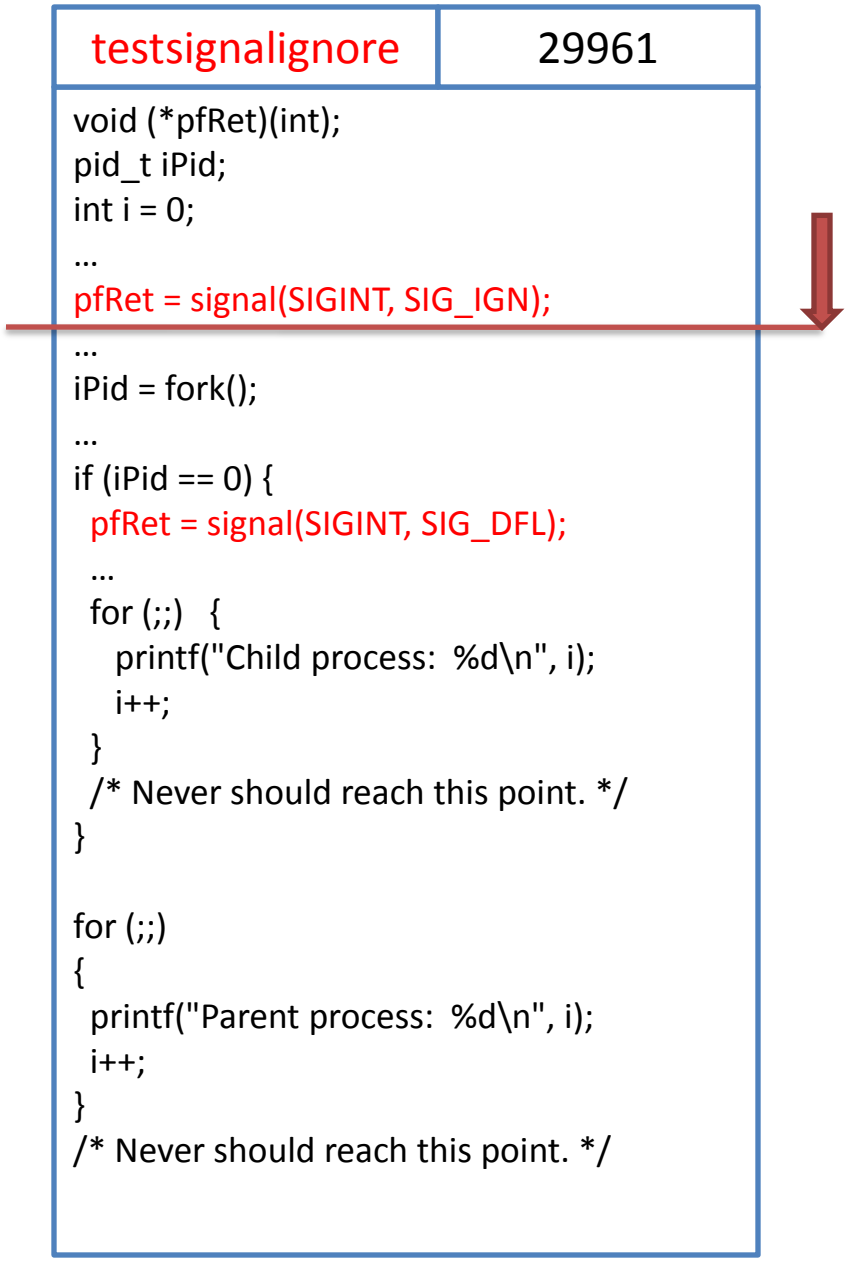
```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int main(int argc, char *argv[])
{
  void (*pfRet)(int);

  pfRet = signal(SIGINT, SIG_IGN);
  if (pfRet == SIG_ERR) {perror(argv[0]);
return EXIT_FAILURE; }

  for (;;)
    ;

  /* Never should reach this point. */
}
```

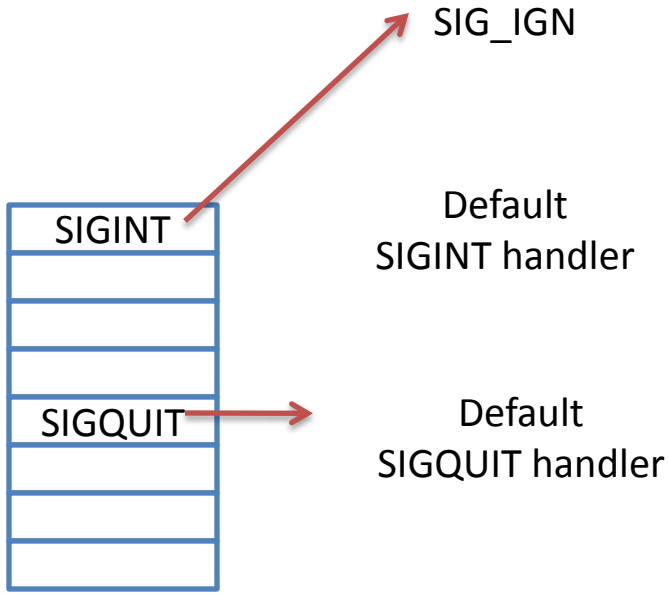


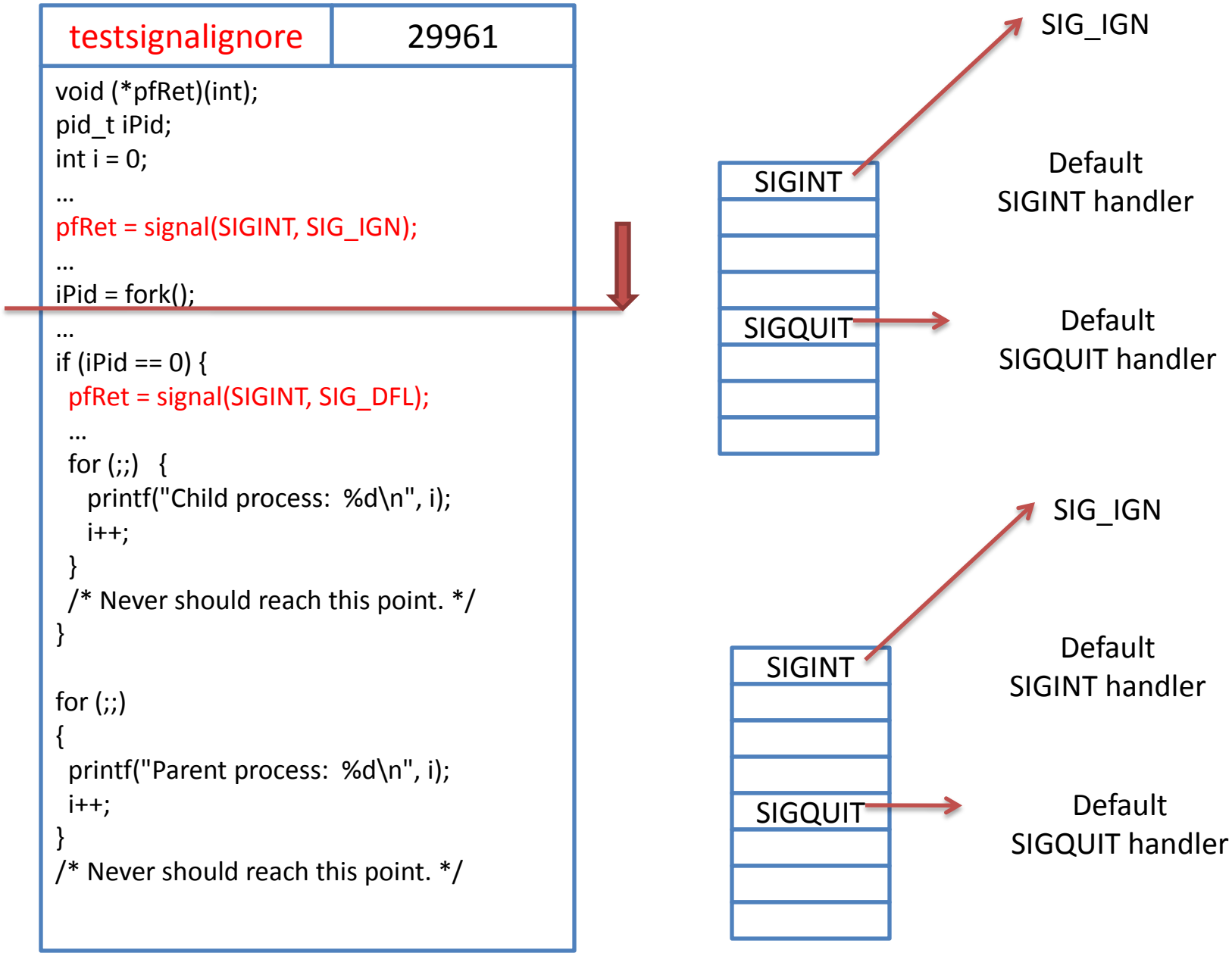
- For ish,
  - Parent should not exit for Ctrl+C
  - Child should exit for Ctrl+C
- When you type Ctrl+C, Unix sends SIGINT signal to both parent and child
- When you type Ctrl+C
  - SIGINT will kill child => Good
  - SIGINT will kill parent => Bad !!

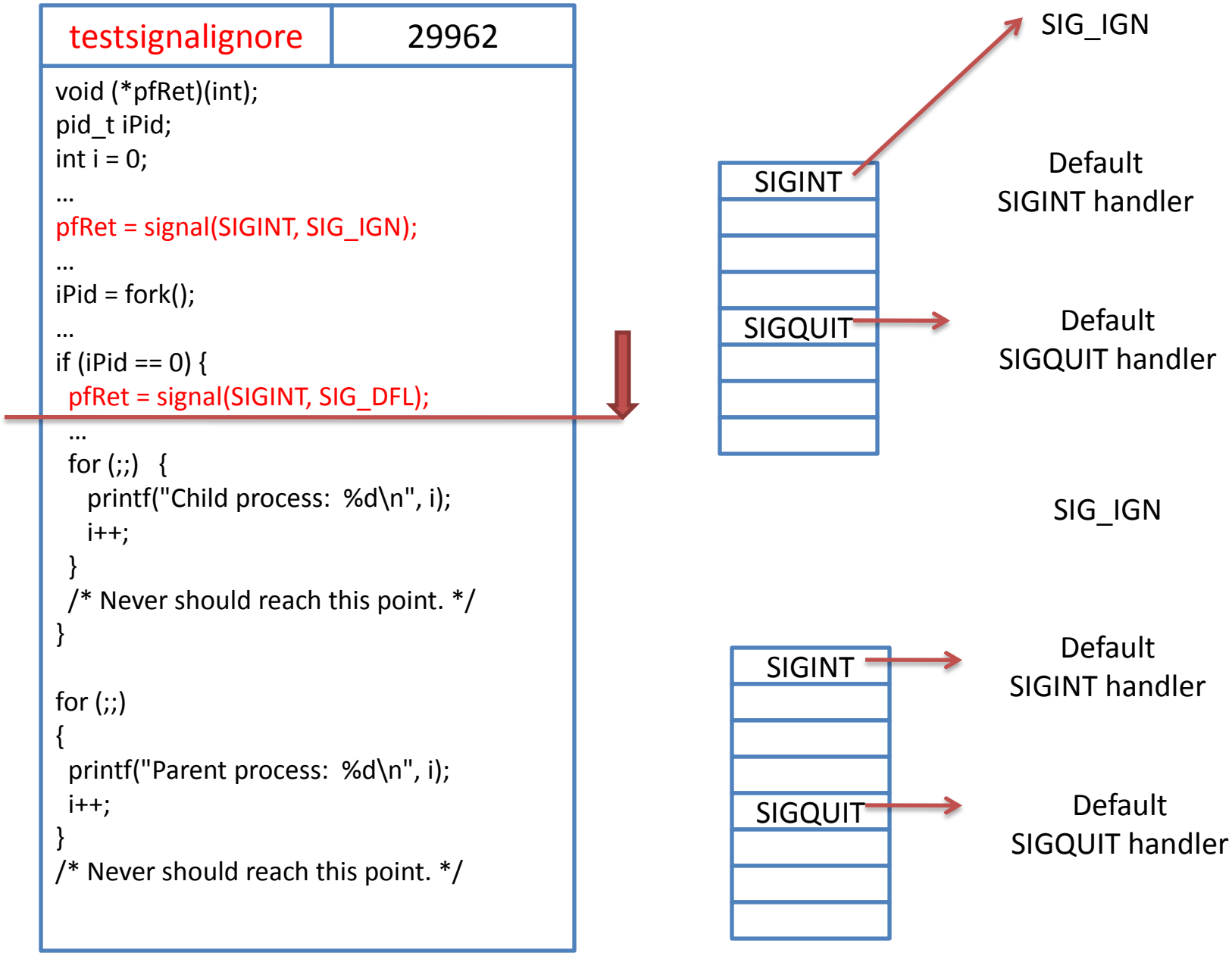
- Solution
  - Ignore Ctrl+C in parent
  - But not in a child
- Let’s look at ‘testsignalignore.c’


testsignalignore
29961
void (*pfRet)(int);
pid_t iPid;
int i = 0;
…
pfRet = signal(SIGINT, SIG_IGN);
…
iPid = fork();
…
if (iPid == 0) {
pfRet = signal(SIGINT, SIG_DFL);
…
for (;;) {
printf("Child process: %d\n", i);
i++;
}
/* Never should reach this point. */
}
for (;;)
{
printf("Parent process: %d\n", i);
i++;
}
/* Never should reach this point. */
SIG_IGN
SIGINT
SIGQUIT
Default
SIGINT handler
Default
SIGQUIT handler


testsignalignore
29961
void (*pfRet)(int);
pid_t iPid;
int i = 0;
…
pfRet = signal(SIGINT, SIG_IGN);
…
iPid = fork();
…
if (iPid == 0) {
pfRet = signal(SIGINT, SIG_DFL);
…
for (;;) {
printf("Child process: %d\n", i);
i++;
}
/* Never should reach this point. */
}
for (;;)
{
printf("Parent process: %d\n", i);
i++;
}
/* Never should reach this point. */


SIG_IGN
SIGINT
Default
SIGINT handler
SIGQUIT
Default
SIGQUIT handler


testsignalignore
29961
void (*pfRet)(int);
pid_t iPid;
int i = 0;
…
pfRet = signal(SIGINT, SIG_IGN);
…
iPid = fork();
…
if (iPid == 0) {
pfRet = signal(SIGINT, SIG_DFL);
…
for (;;) {
printf("Child process: %d\n", i);
i++;
}
/* Never should reach this point. */
}
for (;;)
{
printf("Parent process: %d\n", i);
i++;
}
/* Never should reach this point. */
SIGINT
SIGQUIT
SIG_IGN
Default
SIGINT handler
Default
SIGQUIT handler
SIGINT
SIGQUIT
SIG_IGN
Default
SIGINT handler
Default
SIGQUIT handler


testsignalignore
29962
void (*pfRet)(int);
pid_t iPid;
int i = 0;
…
pfRet = signal(SIGINT, SIG_IGN);
…
iPid = fork();
…
if (iPid == 0) {
pfRet = signal(SIGINT, SIG_DFL);
…
for (;;) {
printf("Child process: %d\n", i);
i++;
}
/* Never should reach this point. */
}
for (;;)
{
printf("Parent process: %d\n", i);
i++;
}
/* Never should reach this point. */
SIG_IGN
SIGINT
SIGQUIT
Default
SIGINT handler
Default
SIGQUIT handler
SIG_IGN
SIGINT
SIGQUIT
Default
SIGINT handler
Default
SIGQUIT handler

Final Comments

- Please refer to man pages, *e.g.*

  $ man signal
- If you are not sure how ish should behave in a certain scenario, check with sampleish
- Your program should be MODULAR
  - easier to grade
  - easier to divide work between group members
  - will carry small weightage during grading
- I will not be able to answer questions after 21st December
- Please mention your group ID in README
  - we will grade your work based on group id
- Good Luck for both your assignment and final exams ☺