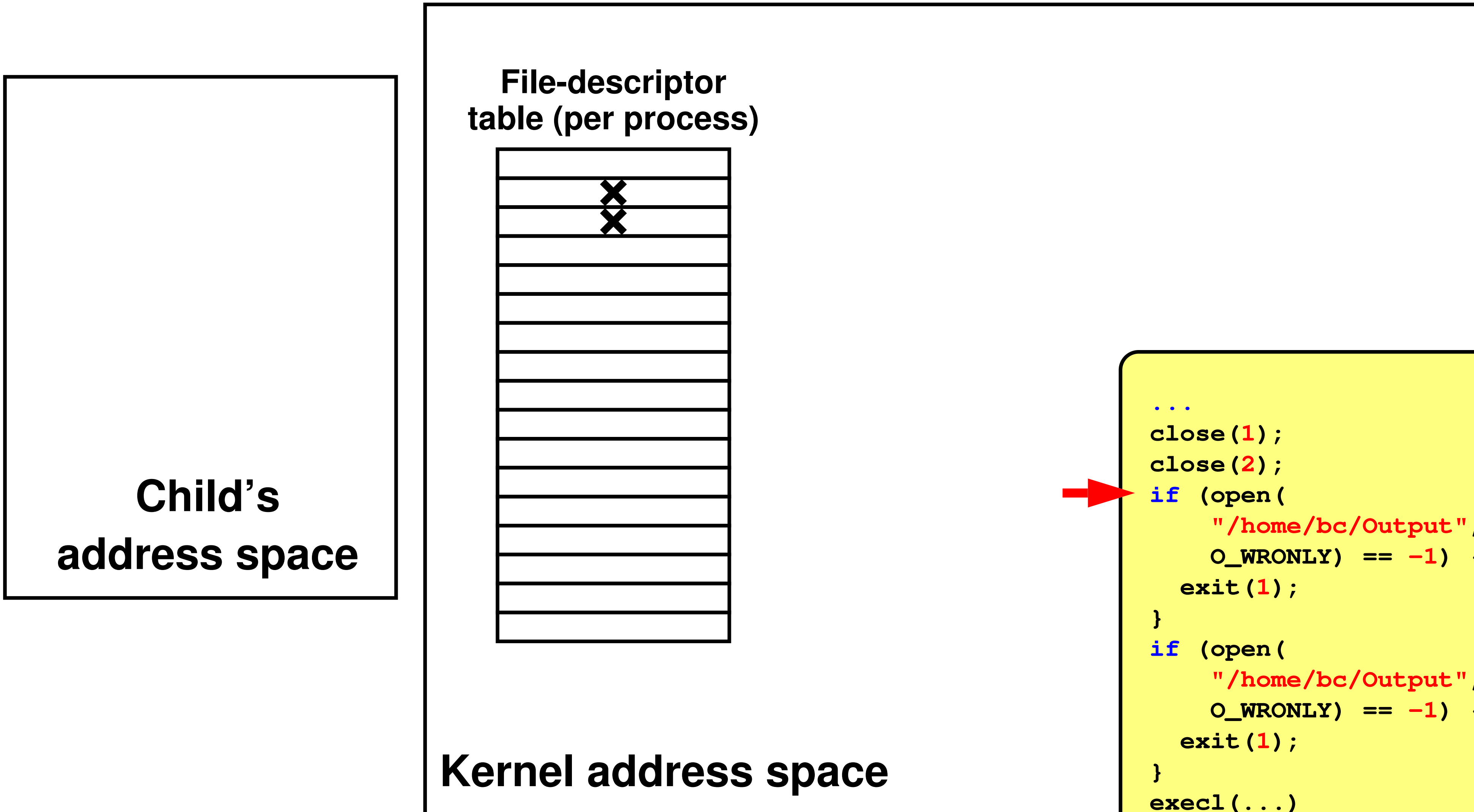
Redirecting Output ... Twice

- Every call to `open()` creates a new entry in the system file table

```
if (fork() == 0) {
  /* set up file descriptors 1 and 2 in the child
     process */
  close(1);
  close(2);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    exit(1);
  }
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    exit(1);
  }
  execl("/home/bc/bin/program", "program", 0);
  exit(1);
}
/* parent continues here */
```

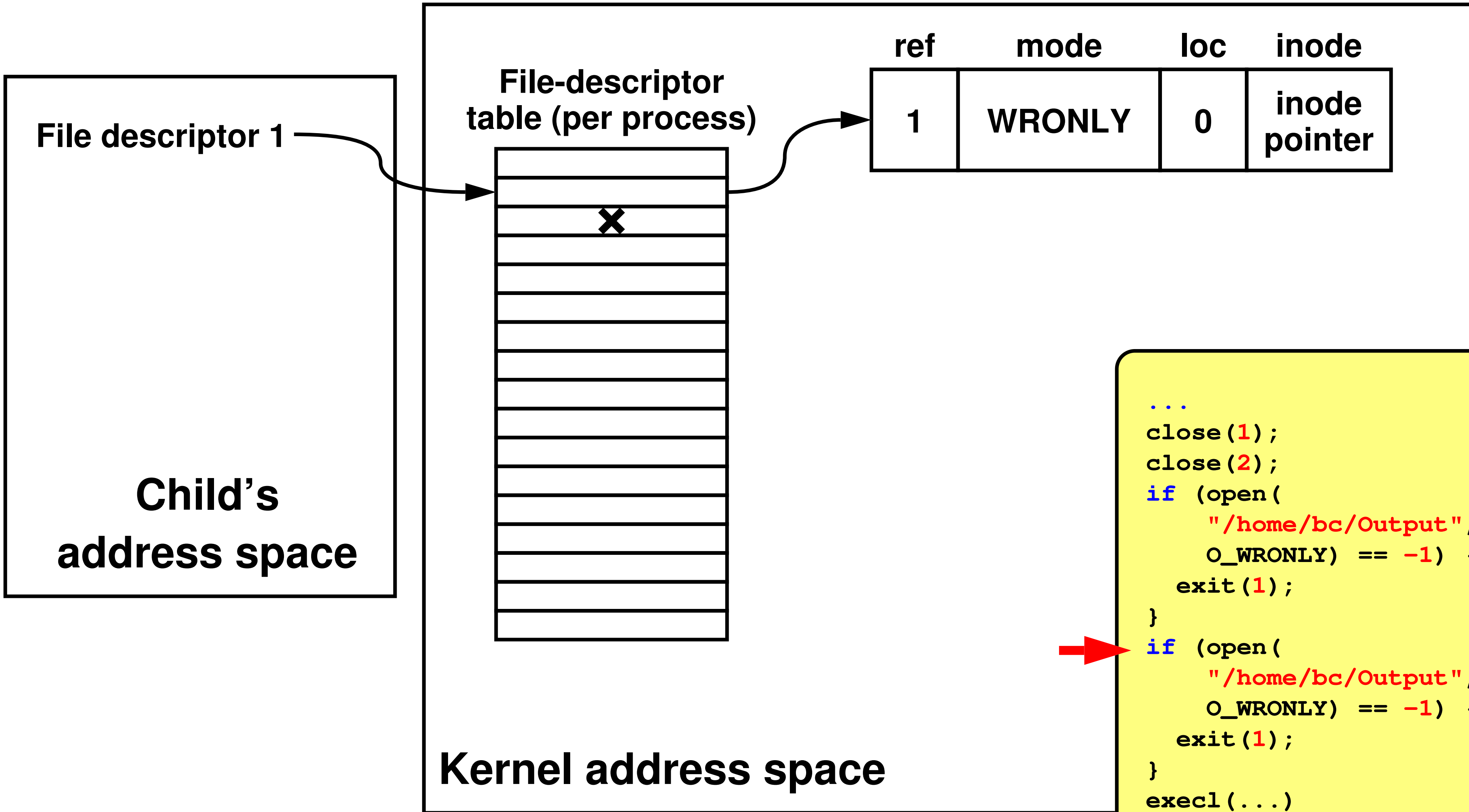
- `stdout` and `stderr` both go into the same file
  - would it cause any problem?

Redirected Output



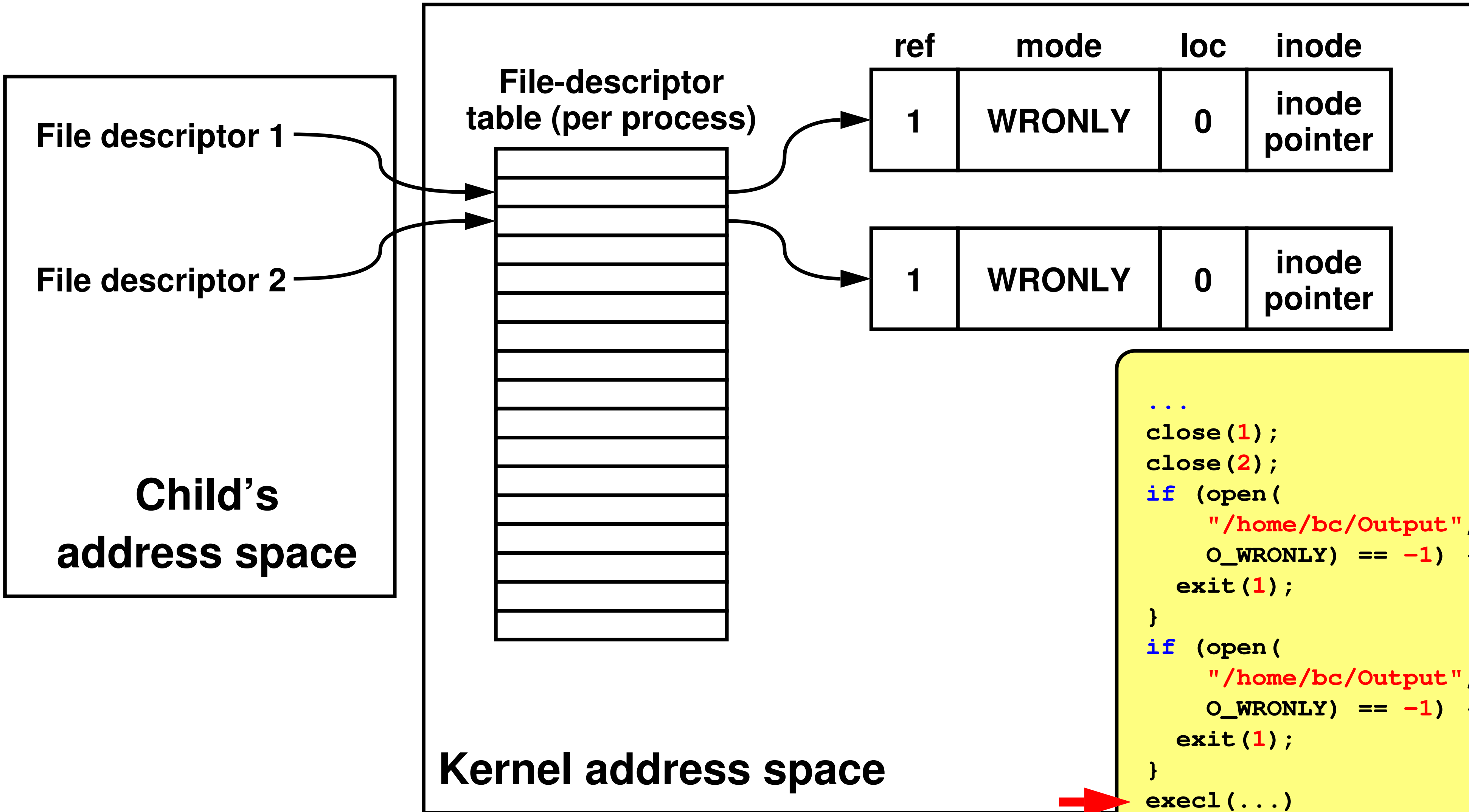

```
...
close(1);
close(2);
if (open(
      "/home/bc/Output",
      O_WRONLY) == -1) {
  exit(1);
}
if (open(
      "/home/bc/Output",
      O_WRONLY) == -1) {
  exit(1);
}
execl(...)
```

Redirected Output



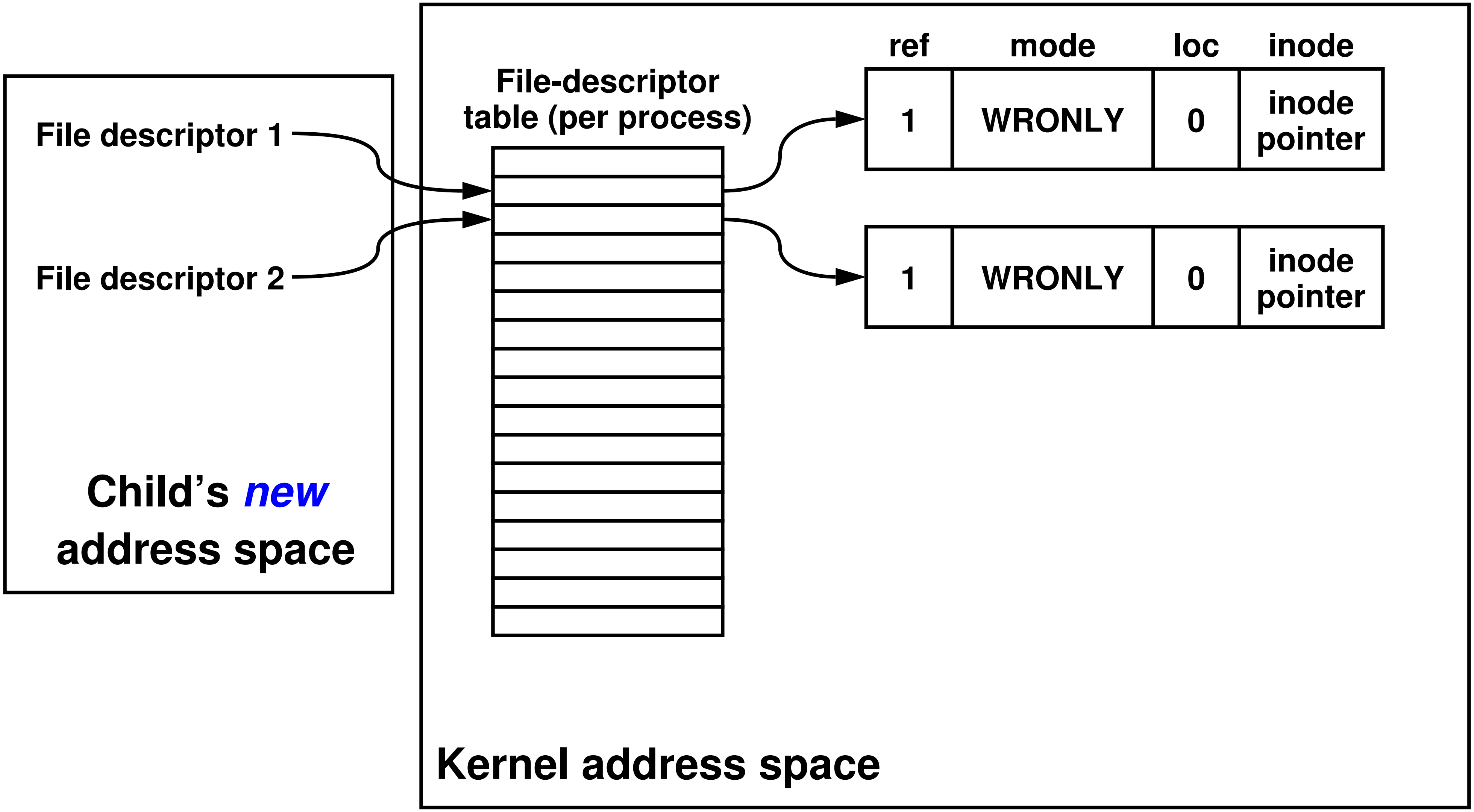

```
...
close(1);
close(2);
if (open(
      "/home/bc/Output",
      O_WRONLY) == -1) {
   exit(1);
}
if (open(
      "/home/bc/Output",
      O_WRONLY) == -1) {
   exit(1);
}
execl(...)
```

Redirected Output



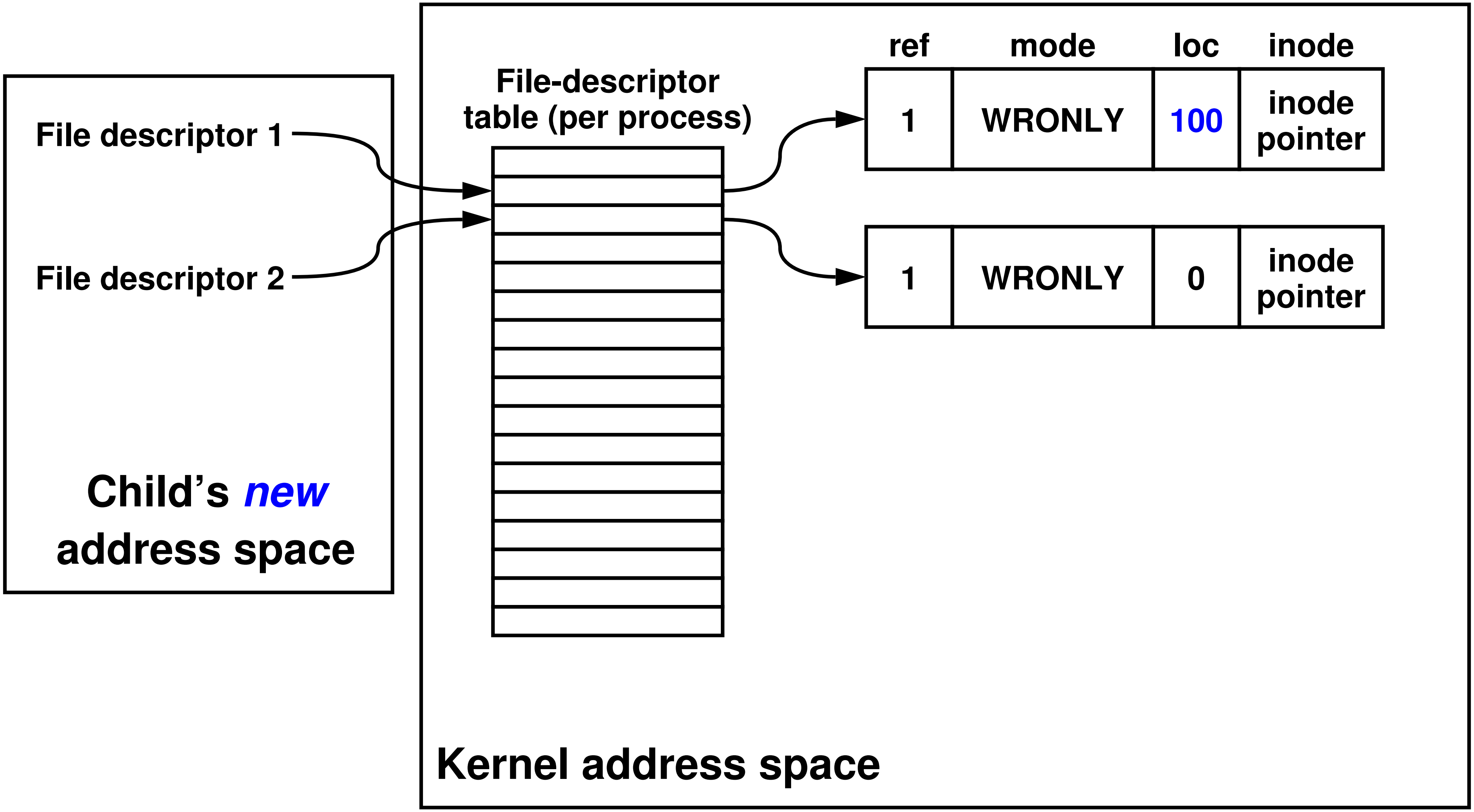

```
...
close(1);
close(2);
if (open(
     "/home/bc/Output",
     O_WRONLY) == -1) {
  exit(1);
}
if (open(
     "/home/bc/Output",
     O_WRONLY) == -1) {
  exit(1);
}
execl(...)
```

Redirected Output



- remember, extended address space survives execs
- let's say we write 100 bytes to `stdout`

Redirected Output After Writing 100 Bytes



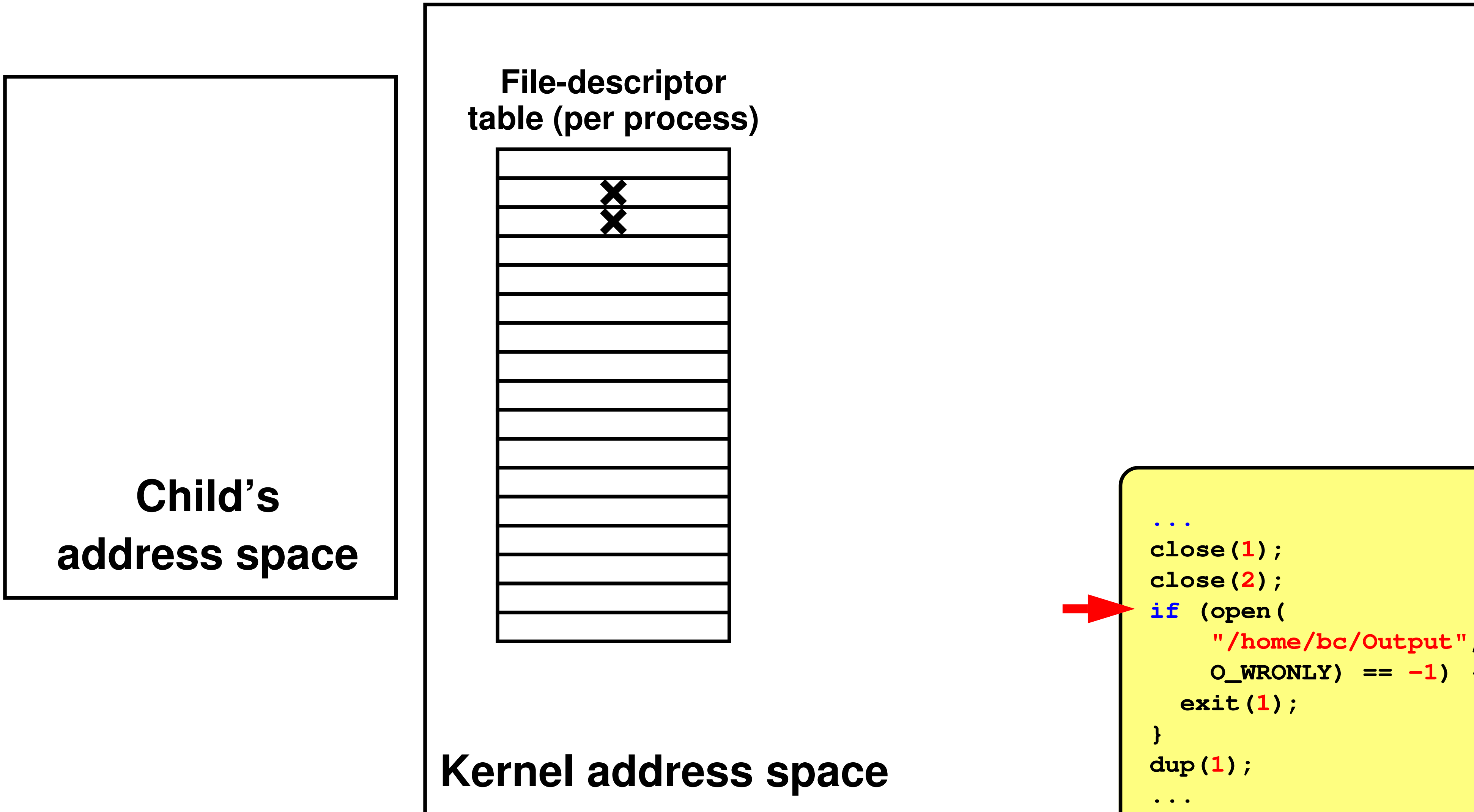
- `write()` to fd=2 will wipe out data in the first 100 bytes
  - that may not be the intent

Sharing Context Information

```
if (fork() == 0) {
  /* set up file descriptors 1 and 2 in the child
     process */
  close(1);
  close(2);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
     exit(1);
  }
  dup(1);
  execl("/home/bc/bin/program", "program", 0);
  exit(1);
}
/* parent continues here */
```

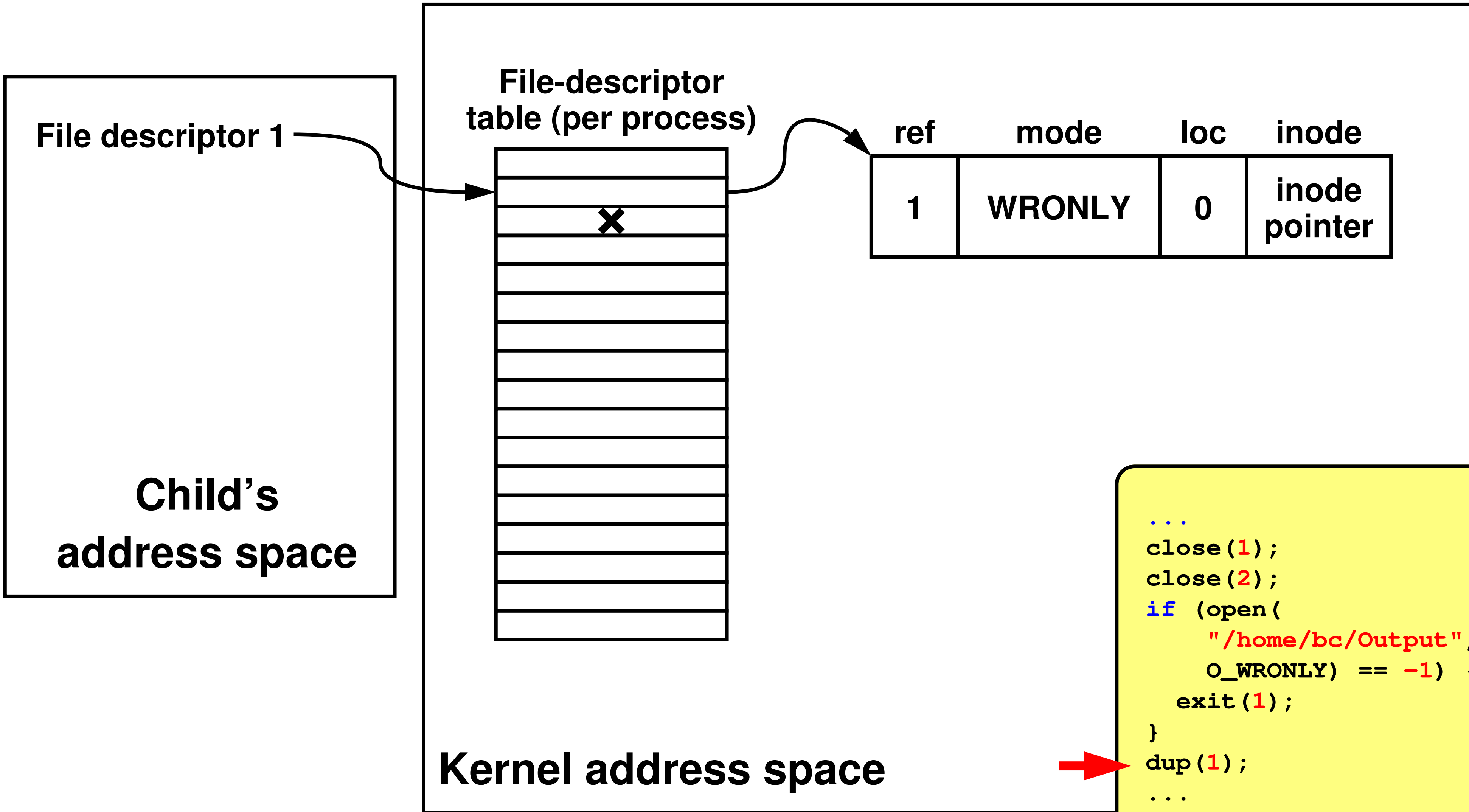
- use the `dup()` system call to *share* context information
  - if that's what you want

Redirected Output After Dup



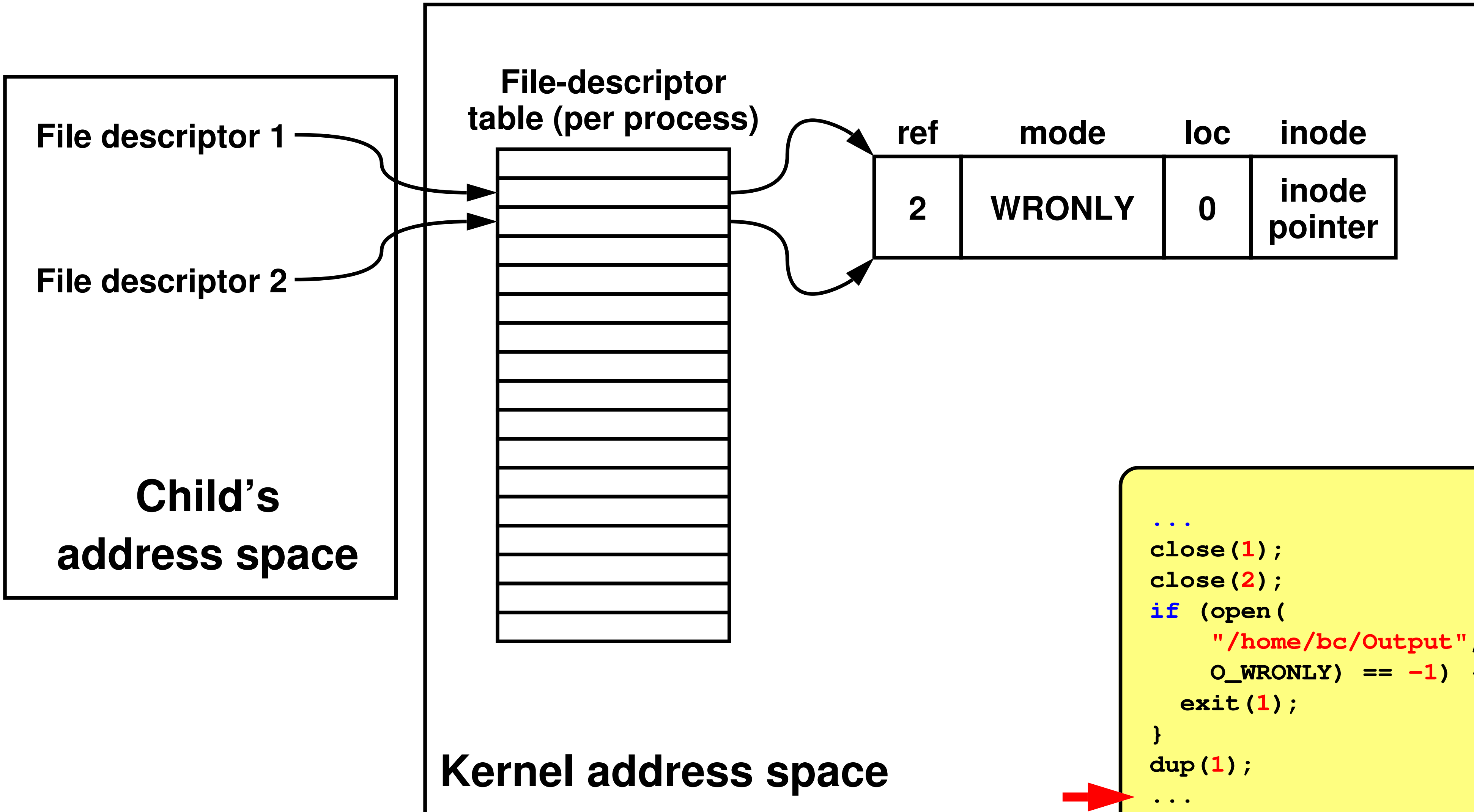

```
...
close(1);
close(2);
if (open(
      "/home/bc/Output",
      O_WRONLY) == -1) {
  exit(1);
}
dup(1);
...
```

Redirected Output After Dup



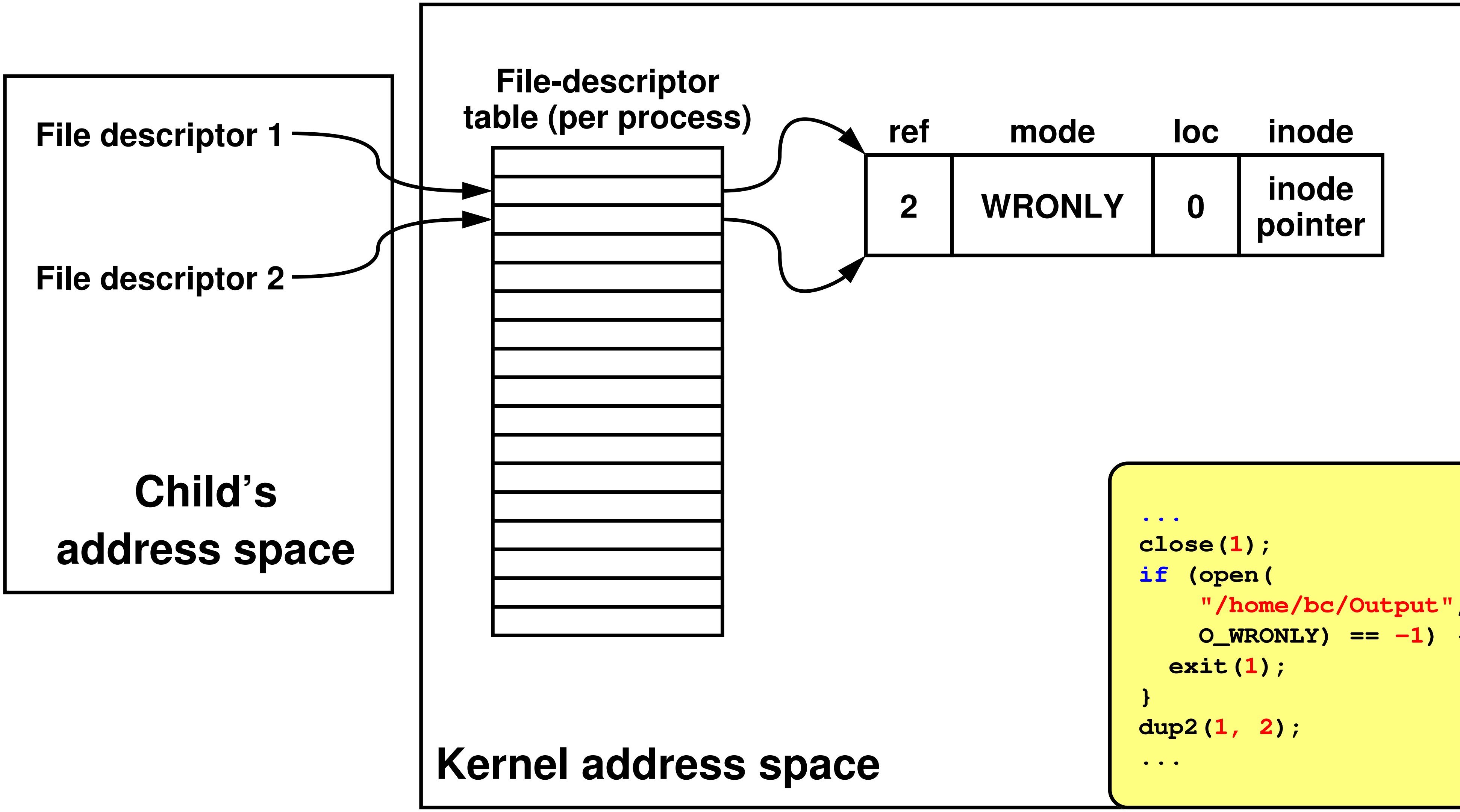

```
...
close(1);
close(2);
if (open(
    "/home/bc/Output",
    O_WRONLY) == -1) {
  exit(1);
}
dup(1);
...
```

Redirected Output After Dup




```
...
close(1);
close(2);
if (open(
    "/home/bc/Output",
    O_WRONLY) == -1) {
  exit(1);
}
dup(1);
...
```

Redirected Output After Dup




```
...
close(1);
if (open(
      "/home/bc/Output",
      O_WRONLY) == -1) {
   exit(1);
}
dup2(1, 2);
...
```

➡ There is also a `dup2()` system call

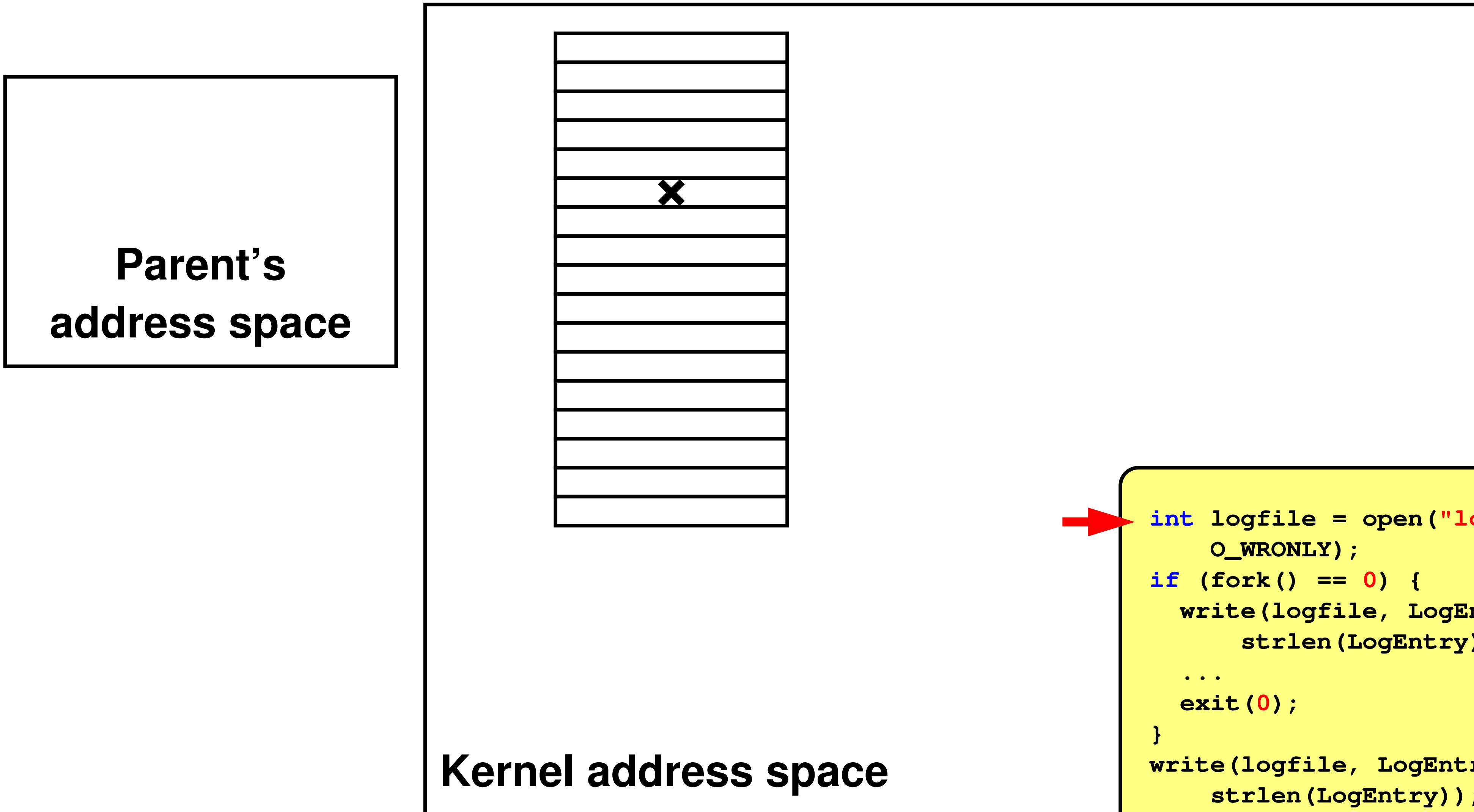
Fork and File Descriptors

- It would be useful to be able to share file context information with a child process
  - when `fork()` is called, the child process gets a *copy* of the parent's file descriptor table

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
  /* child process computes something, then does: */
  write(logfile, LogEntry, strlen(LogEntry));
  ...
  exit(0);
}
/* parent process computes something, then does: */
write(logfile, LogEntry, strlen(LogEntry));
...
```

  - remember, extended address space survives execs
    - also `fork()`

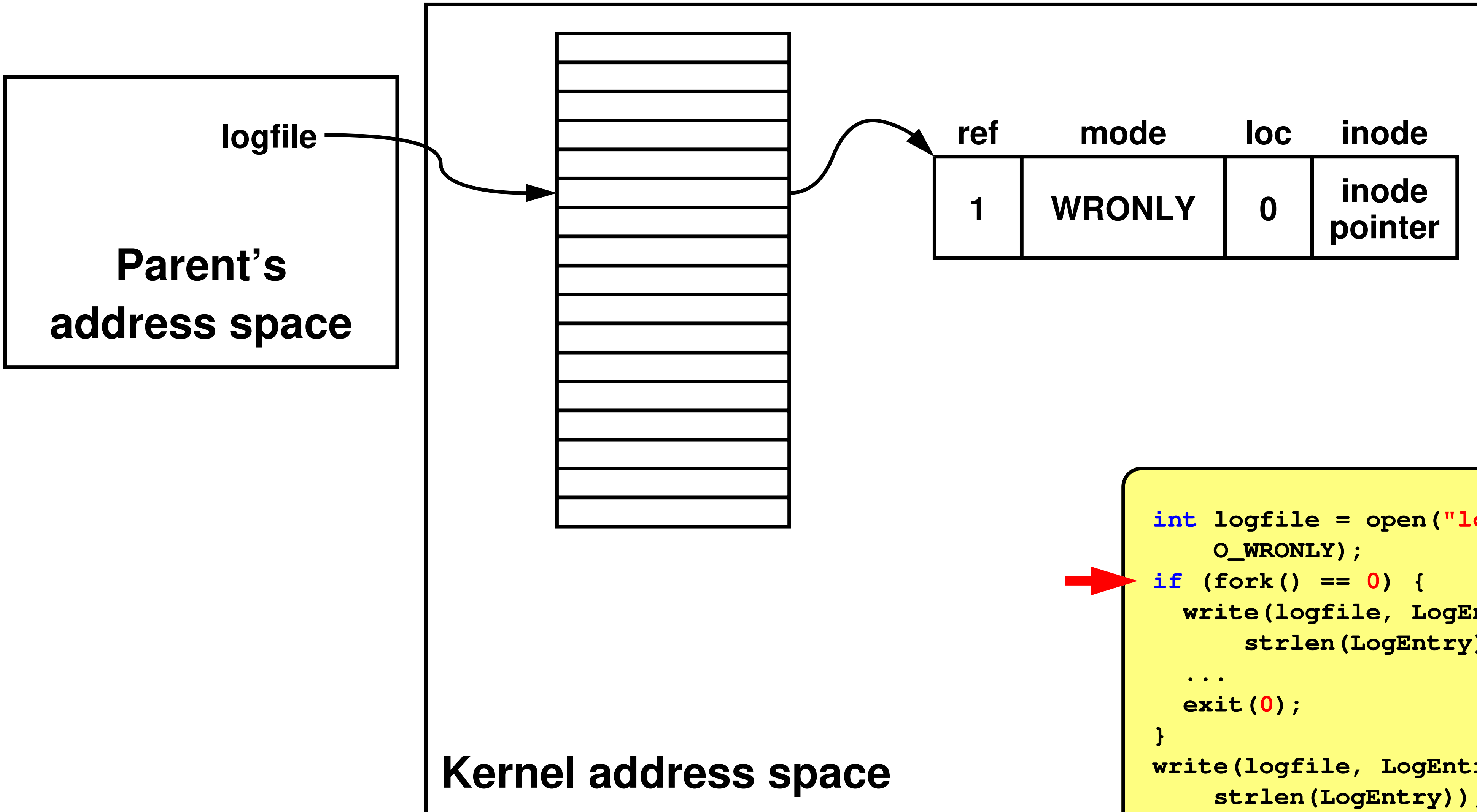
File Descriptors After Fork




```
int logfile = open("log",
    O_WRONLY);
if (fork() == 0) {
  write(logfile, LogEntry,
      strlen(LogEntry));
  ...
  exit(0);
}
write(logfile, LogEntry,
    strlen(LogEntry));
```

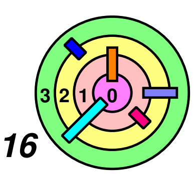
- parent and child processes get *separate* file descriptor table but *share* extended address space

File Descriptors After Fork

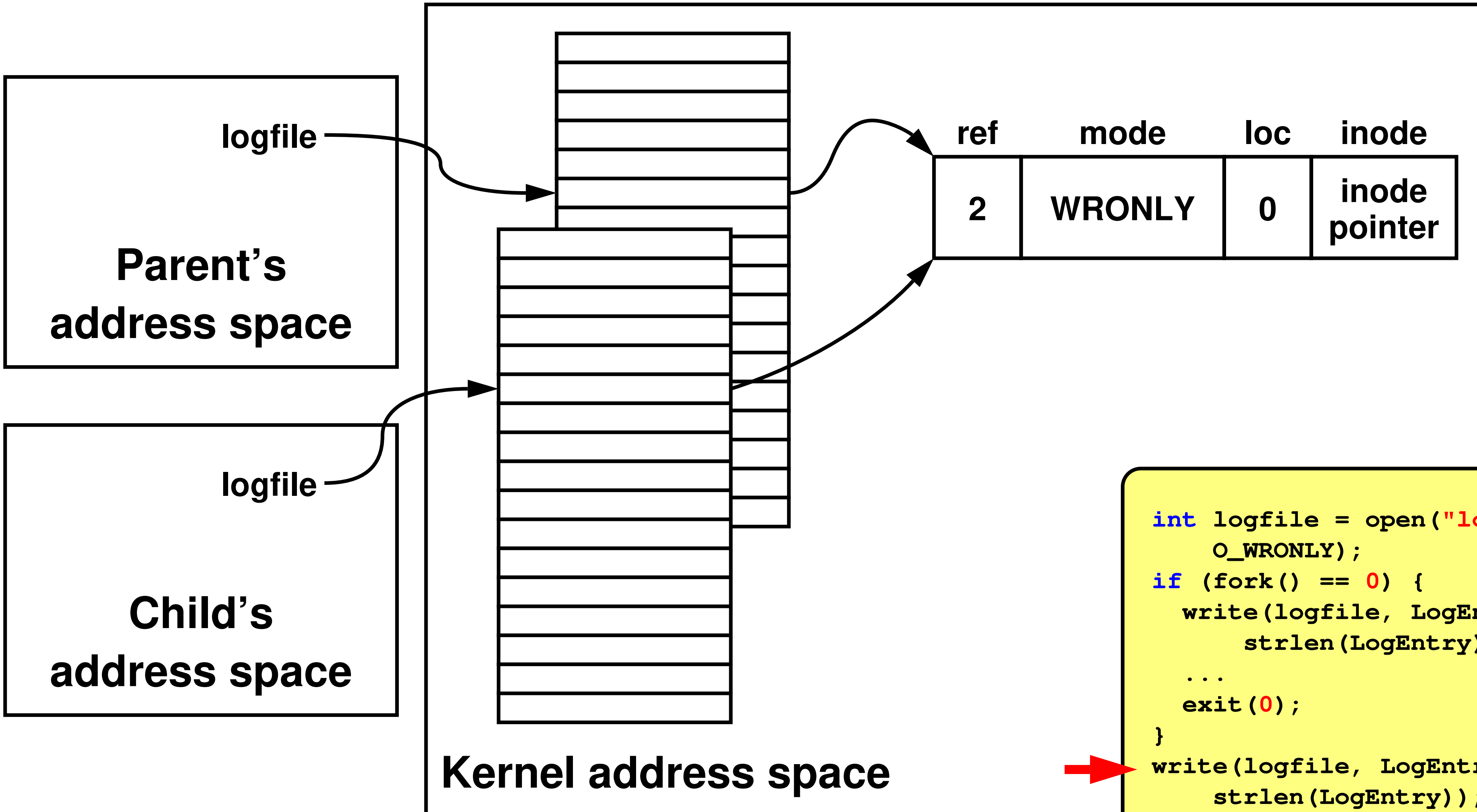



```
int logfile = open("log",
    O_WRONLY);
if (fork() == 0) {
  write(logfile, LogEntry,
      strlen(LogEntry));
  ...
  exit(0);
}
write(logfile, LogEntry,
    strlen(LogEntry));
```

- parent and child processes get *separate* file descriptor table but *share* extended address space



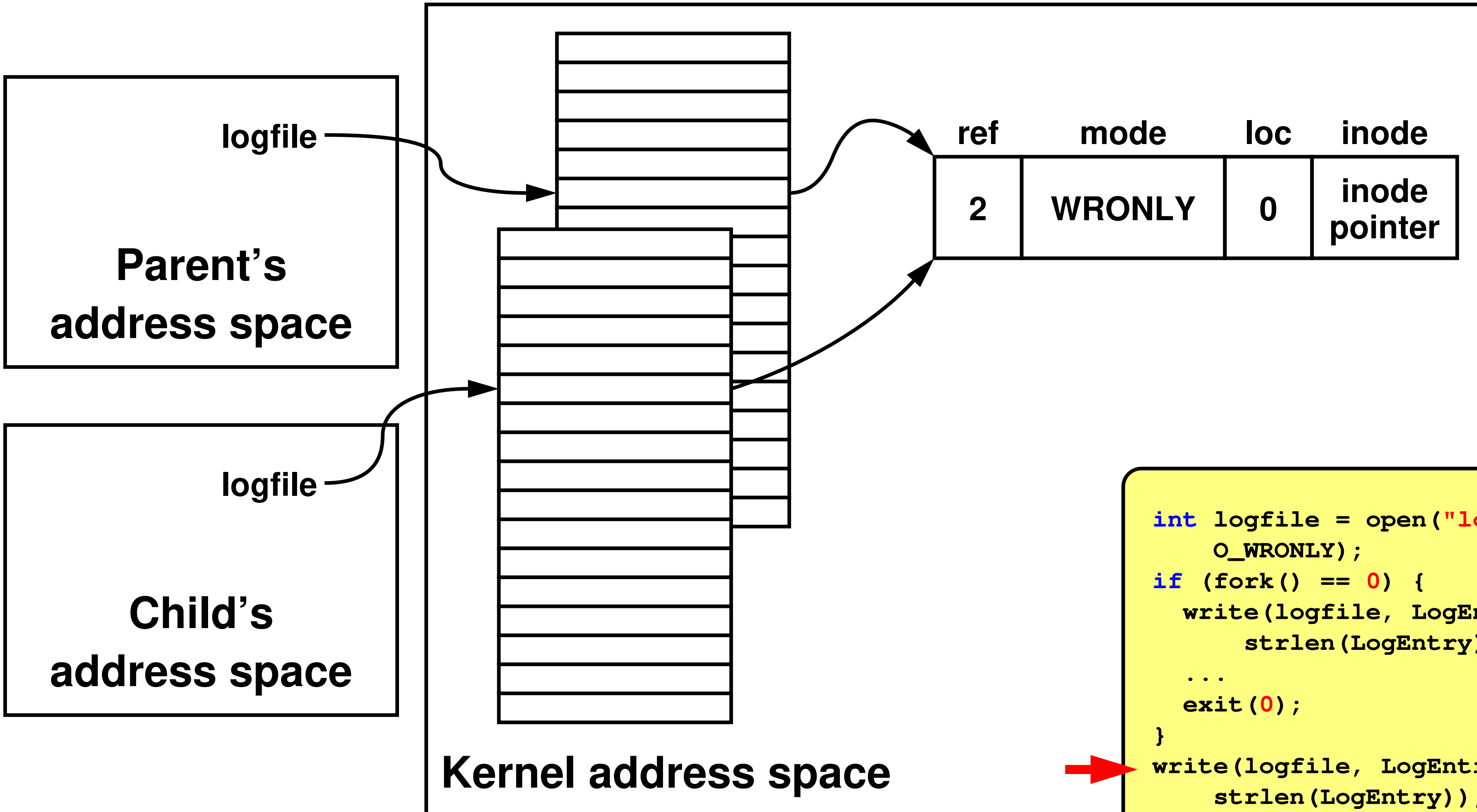
File Descriptors After Fork




```
int logfile = open("log",
    O_WRONLY);
if (fork() == 0) {
  write(logfile, LogEntry,
      strlen(LogEntry));
  ...
  exit(0);
}
write(logfile, LogEntry,
    strlen(LogEntry));
```

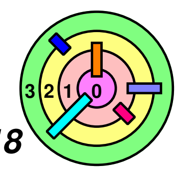
- parent and child processes get *separate* file descriptor table but *share* extended address space *indirectly*

File Descriptors After Fork



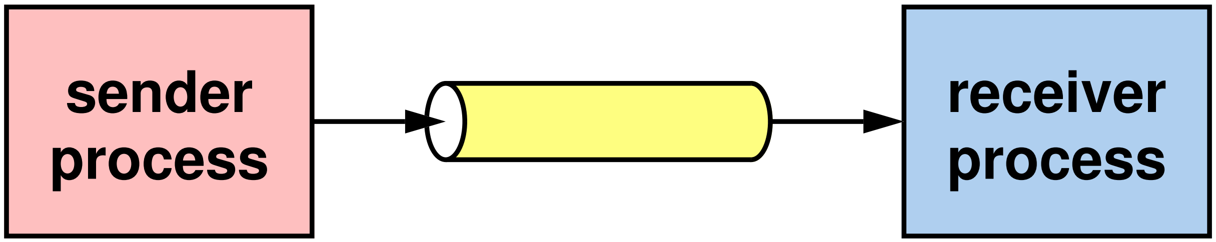

```
int logfile = open("log",
    O_WRONLY);
if (fork() == 0) {
  write(logfile, LogEntry,
      strlen(LogEntry));
  ...
  exit(0);
}
write(logfile, LogEntry,
    strlen(LogEntry));
```

- parent and child processes can communicate using such a shared file descriptor, although *difficult to synchronize*



Pipes

- A pipe is a means for one process to send data to another directly, as if it were writing to a file



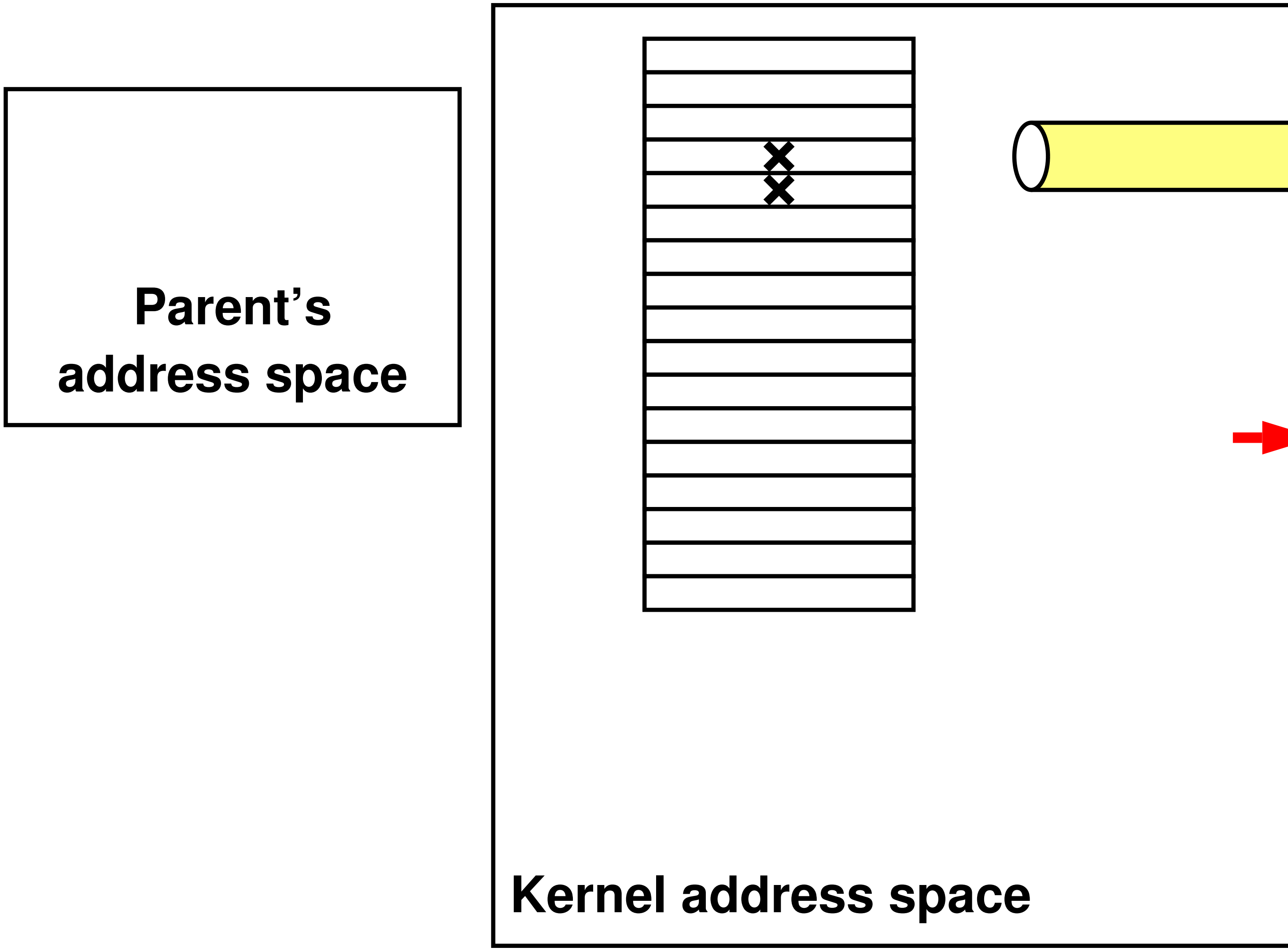
  - the sending process behaves as if it has a file descriptor to a file that has been opened for writing
  - the receiving process behaves as if it has a file descriptor to a file that has been opened for reading

- The `pipe()` system call creates a pipe object in the kernel and returns (via an output parameter) the two file descriptors that refer to the pipe
  - one, set for write-only, refers to the input side
  - the other, set for read-only, refers to the output side
  - a pipe has no name, cannot be passed to another process

Pipes

```
int p[2]; // array to hold pipe's file descriptors
pipe(p);  // creates a pipe, assume no errors
  // p[0] refers to the read/output end of the pipe
  // p[1] refers to the write/input end of the pipe
if (fork() == 0) {
  char buf[80];
  close(p[1]); // not needed by the child
  while (read(p[0], buf, 80) > 0) {
    // use data obtained from parent
    ...
  }
  exit(0); // child done
} else {
  char buf[80];
  close(p[0]); // not needed by the parent
  for (;;) {
    // prepare data for child
    ...
    write(p[1], buf, 80);
  }
}
```

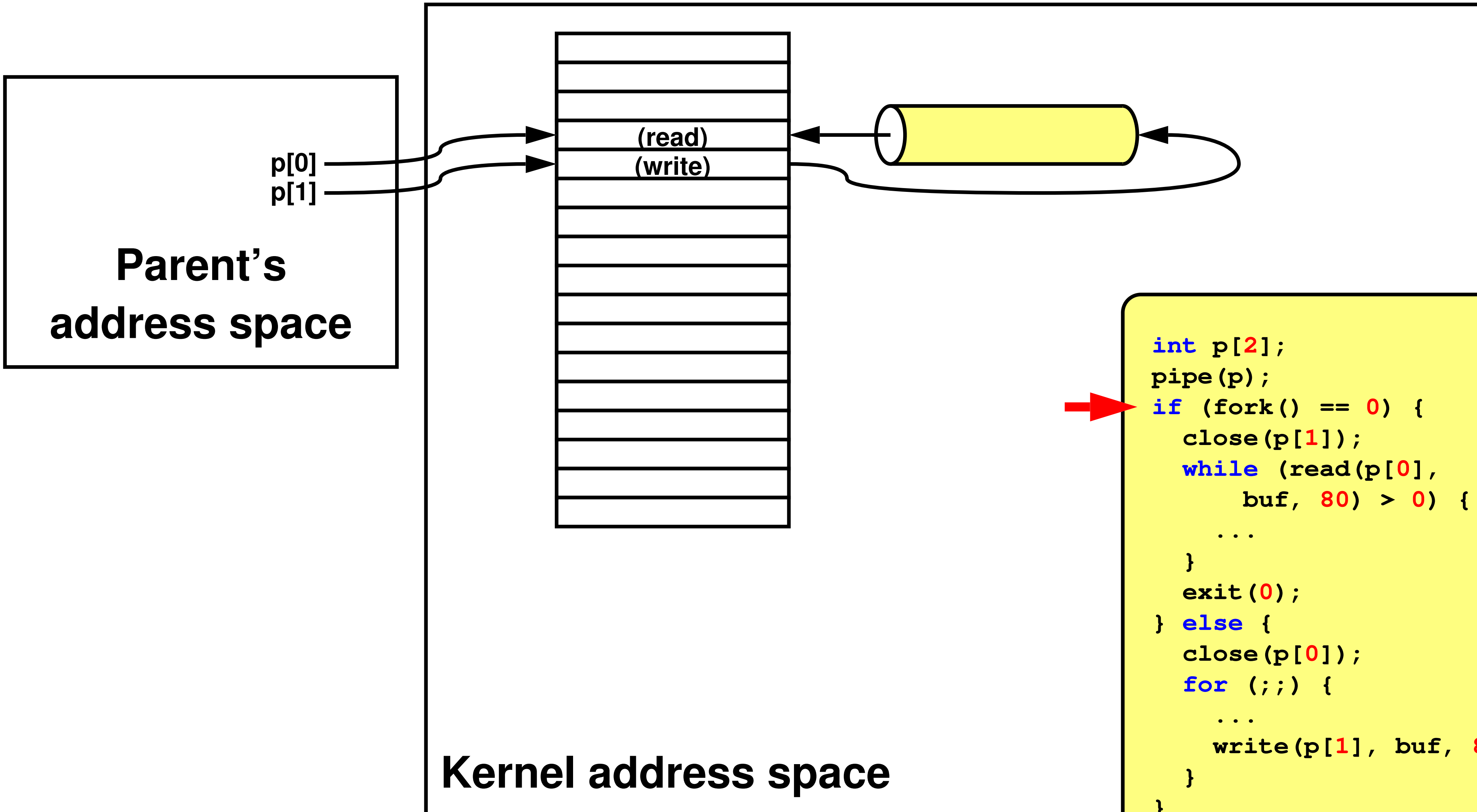
Pipes




```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

- parent creates a pipe object in the kernel

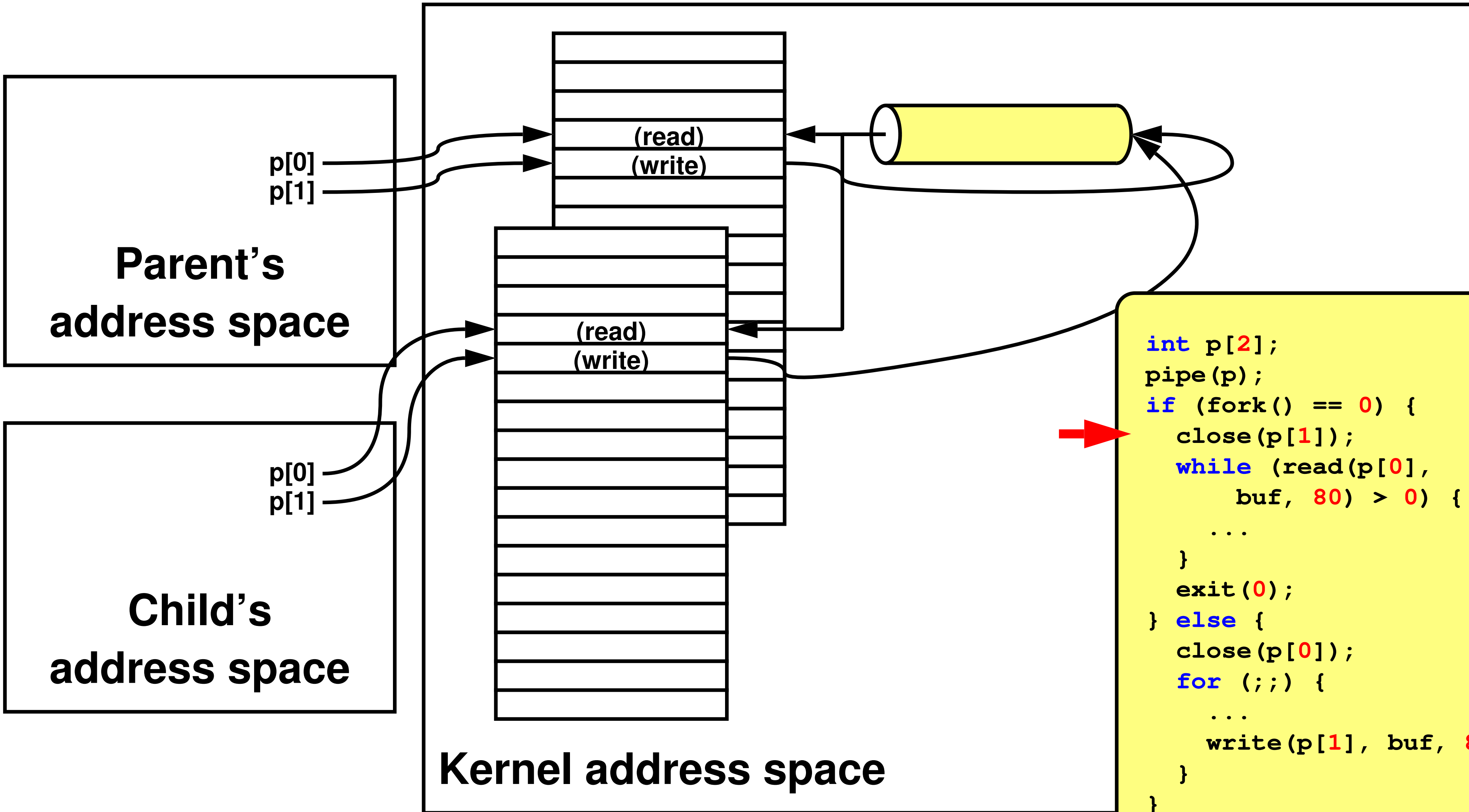
Pipes




```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

- parent creates a pipe object in the kernel

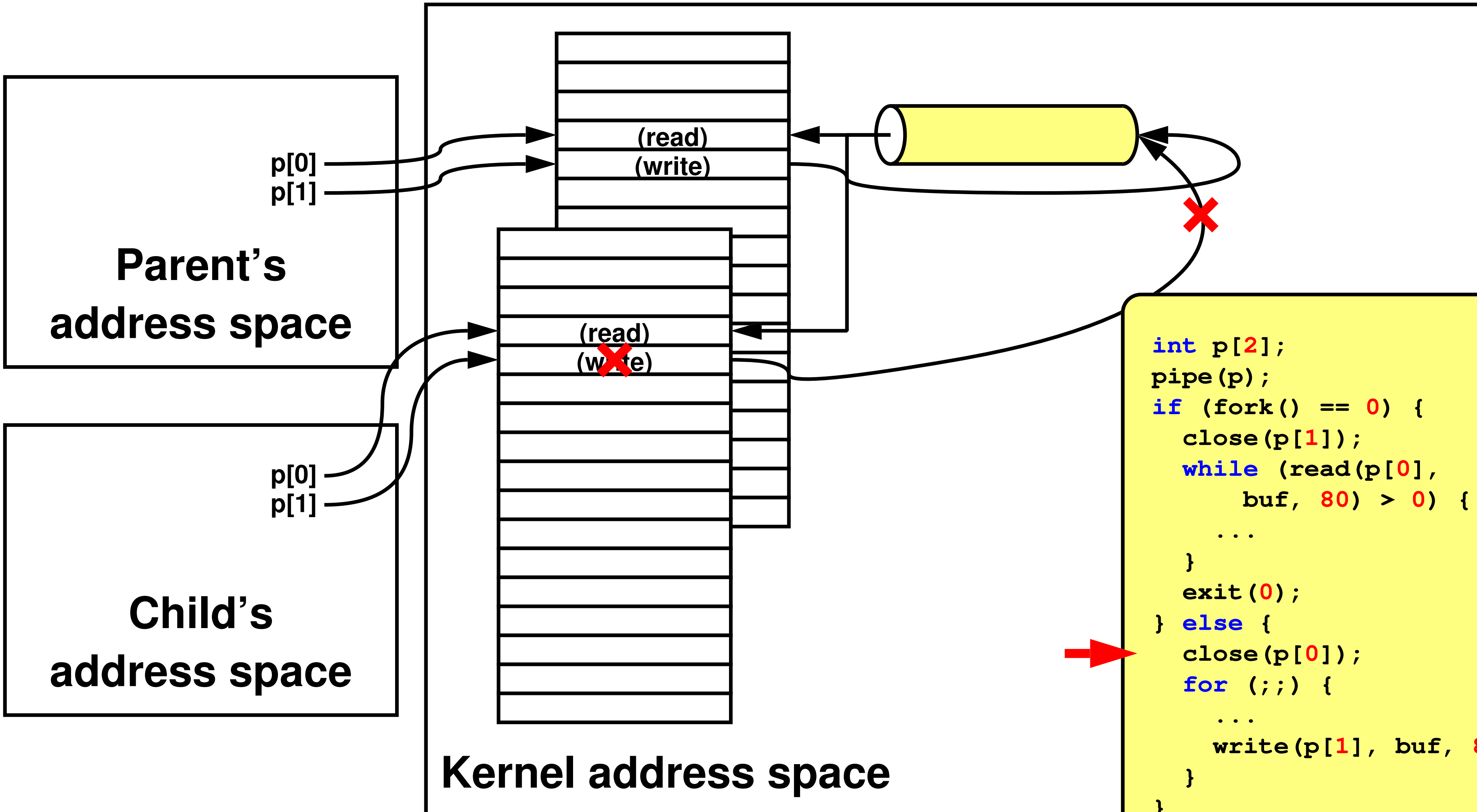
Pipes




```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

- parent and child processes get *separate* file descriptor tables but *share* extended address space

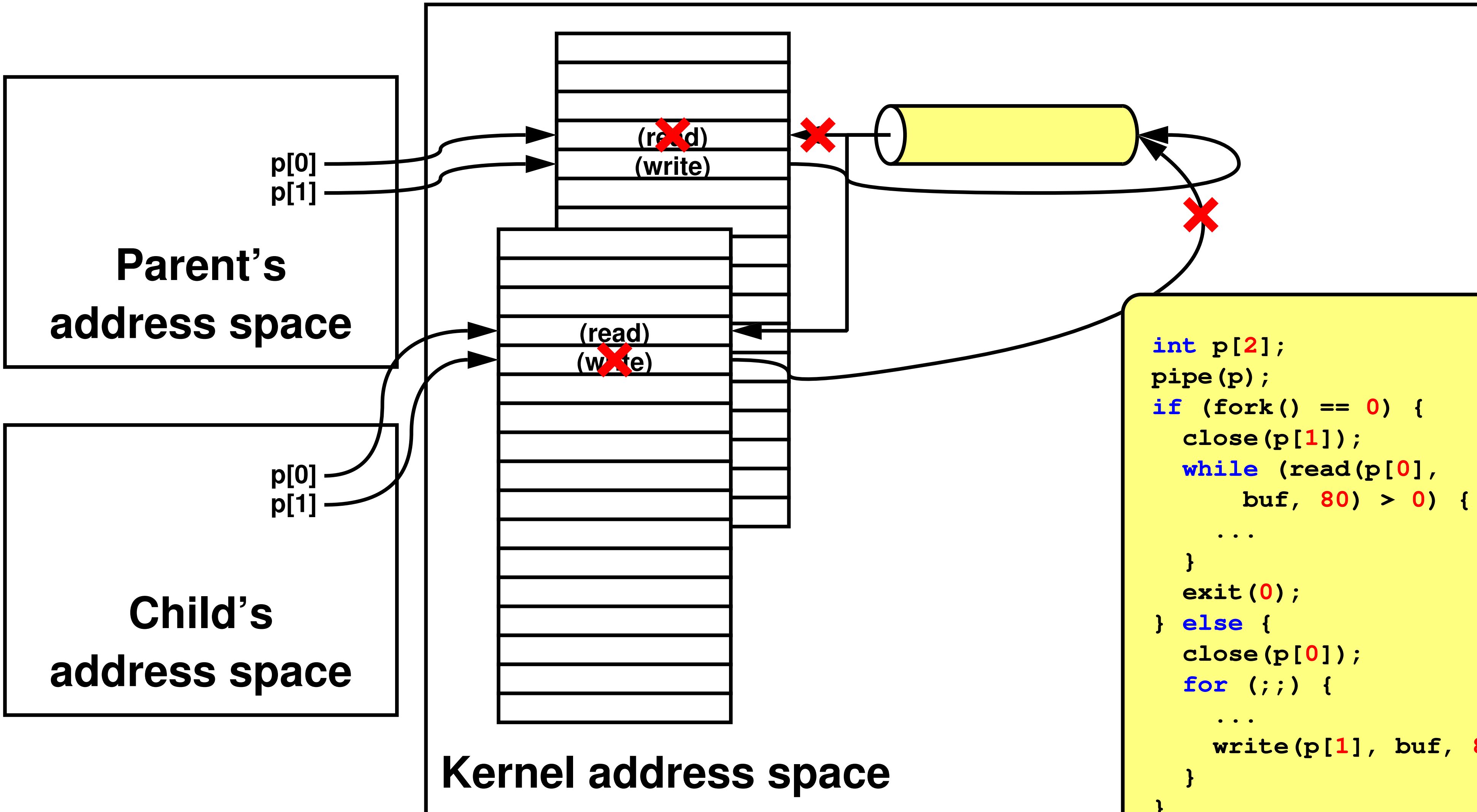
Pipes




```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
        buf, 80) > 0) {
     ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
     ...
    write(p[1], buf, 80);
  }
}
```

- child closes the write-end of the pipe

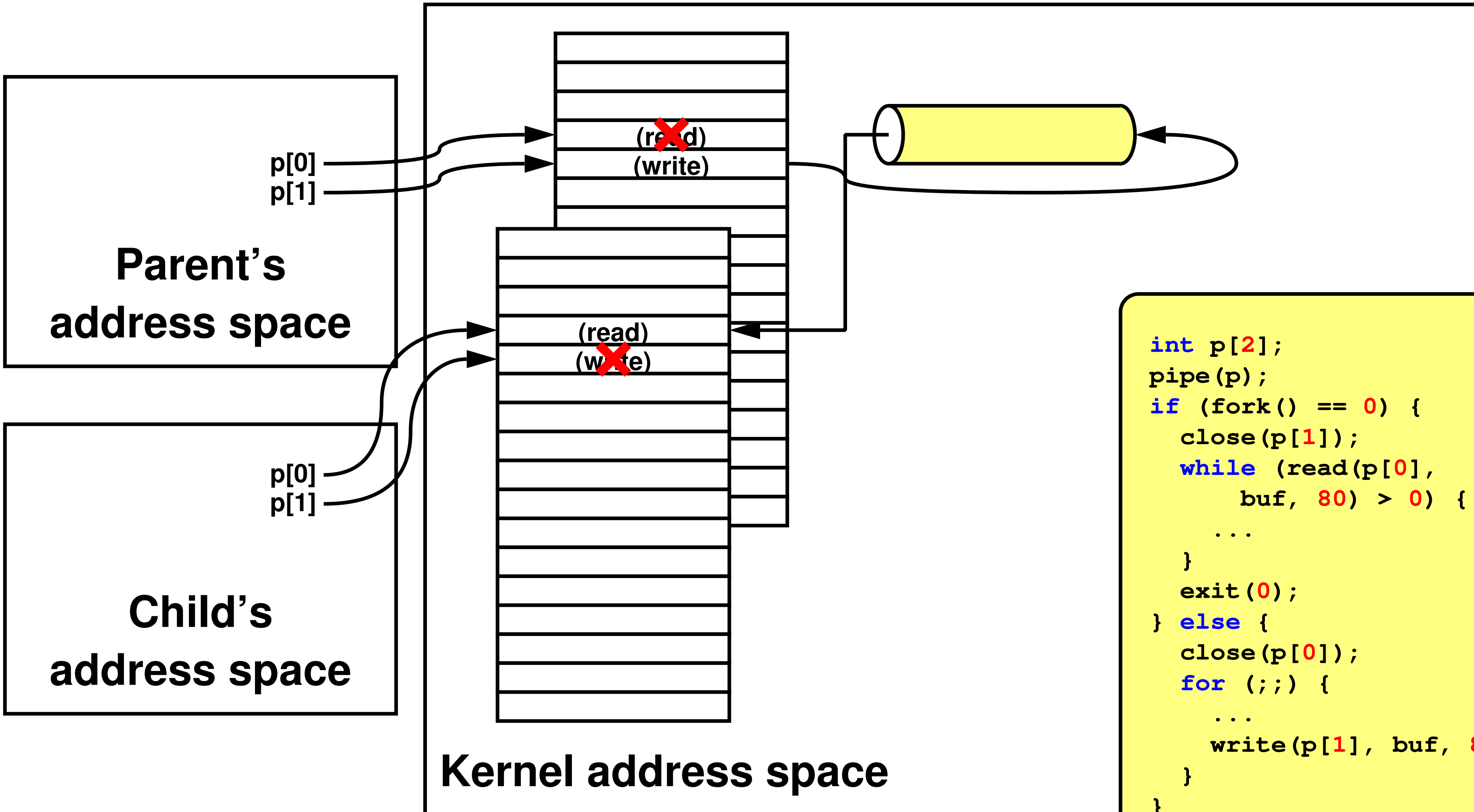
Pipes




```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

- child closes the write-end of the pipe
- parent closes the read-end of the pipe

Pipes




```
int p[2];
pipe(p);
if (fork() == 0) {
  close(p[1]);
  while (read(p[0],
      buf, 80) > 0) {
    ...
  }
  exit(0);
} else {
  close(p[0]);
  for (;;) {
    ...
    write(p[1], buf, 80);
  }
}
```

- child closes the write-end of the pipe
- parent closes the read-end of the pipe

Command Shell

- Now you know enough to write a command shell
  - execute a command
  - redirect I/O
  - pipe the output of one program to another

    ```
    cat f0 | ./warmup1 sort
    ```

    - the shell needs to create a pipe
    - create two child processes
    - in the first child
      - have `stdout` go to the write-end of the pipe
      - close the read-end of the pipe
      - exec "`cat f0`"
    - in the 2nd child
      - have `stdin` come from the read-end of the pipe
      - close the write-end of the pipe
      - exec "`./warmup1 sort`"
  - run a program in the background

    ```
    primes 1000000 > primes.out &
    ```

Random Access In Sequential I/O

```
fd = open("textfile", O_RDONLY);
// go to last char in file
fptr = lseek(fd, (off_t)(-1), SEEK_END);
while (fptr != -1) {
  read(fd, buf, 1);
  write(1, buf, 1);
  fptr = lseek(fd, (off_t)(-2), SEEK_CUR);
}
```

- "man lseek" gives

  `off_t lseek(int fd, off_t offset, int whence);`
- `whence` can be SEEK_SET, SEEK_CUR, SEEK_END
- if succeeds, returns cursor position (always measured from the beginning of the file)
  - otherwise, returns (-1)
  - `errno` is set to indicate the error
- `read(fd,buf,1)` advances the cursor position by 1, so we need to move the cursor position back 2 positions