10-UNIX
fork & wait

Process Creation

Chapter 24

The *fork()* System Call

- *fork()* creates a new process, the *child*, which is an almost exact duplicate of the calling process, the *parent*.
  - The *child* has its own process ID.
  - The *child* inherits the same stack, data and heap.
  - After the *fork(),* both processes can make changes independently.

The *fork()* System Call

Call: #include <unistd.h>

pid_t **fork** (void);

In **parent**: returns process ID of child on success, or -1 on error;

If successfully created, **child** always returns a 0.

wait System Calls

- wait
- waitpid
- waitid
- They all wait for a process to Change State

wait Calls

A state change is considered to be:

- the child was stopped by a signal
- the child was resumed by a signal
- the child terminated
  - In the case of a terminated child, performing a wait allows the system to release the resources associated with the child
  - if a wait is not performed, then the terminated child remains in a "zombie" state

wait() System Call

It suspends execution of the calling process until one of its children terminates. A simple call without much flexibility.

Call: #include <sys/wait.h>

pid_t **wait**(int *status);

Returns process ID of terminated child,

or -1 on error

waitpid() System Call (1 of 2)

- It suspends execution of the calling process until a child specified by ***pid*** argument has a **changed state**.
- Has more flexibility than ***wait()*** but also more complexity.

waitpid() function (2 of 2)

Call: #include <sys/types.h>

#include <sys/wait.h>

pid_t **waitpid**(pid_t **pid**, int *status, int opts)

The value of PID can be:

Tag	Description
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0	meaning wait for the child whose process ID is equal to the value of *pid*.

waitpid() does not block

wait vs waitpid

wait

- Can block the caller until a child process terminates
- If the caller has multiple children, **wait** returns when one terminates

waitpid

- Has an option that prevents it from blocking.
- If the caller has multiple children, **waitpid** has a number of options that control which process it waits for

Options for *waitpid()*

Zero or more of the following constants can be OR-ed.

WNOHANG

Return immediately if no child has exited.

(Demands status information immediately!)

WUNTRACED

Also return if a child has stopped. Status for traced children which have stopped is provided even if this option is not specified.

(Reports on stopped child processes as well as terminated ones!)

Return value

The process ID of the child which exited:

-1 on error;

0 if WNOHANG was used and no child was available.

The Wait Status Field

wait() and ***waitpid()*** return a pointer to the status of the 'child' which terminated:

- The child terminated by calling *_exit() (*or *exit())* specifying an integer *exit status*
- The child was terminated by the delivery of an unhandled signal
- The child was stopped by a signal, and *waitpid()* was called with the **WUNTRACED** flag
- The child was resumed by a **SIGCONT** signal, and *waitpid()* was called with the **WCONTINUED** flag.

The Wait Status Field

Status can be tested using one of three POSIX macros:

WIFEXITED(status): true if normal exit

WEXITSTATUS(status) fetches exit/return value

WIFSIGNALED(status): true if abnormally exited

WTERMSIG(status) fetches signal number

WIFSTOPPED(status): true if stopped

WSTOPSIG(status) fetches signal number

Example: waitpid (1 of 2)

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

int main(void)
 {
   pid_t pid;
   int status;

   if( (pid = fork() ) == 0 )
     { /* child */
       printf("I am a child with pid = %d\n", getpid());
       sleep(10);
       printf("Child. Child terminates\n");
       _exit(EXIT_SUCCESS);
   }
```

(2 of 2)

```
    else
    { /* parent */
       while (1) {
          waitpid( pid, &status, WUNTRACED );
          if( WIFSTOPPED(status) ){
              printf("Parent. Child stopped, signal(%d)\n",
                                  WSTOPSIG(status));
               continue;
          }
          else if( WIFEXITED(status) ){
              printf("Parent: Normal termination with
                       status(%d)\n",
                      WEXITSTATUS(status));
              break;
          }
          else if (WIFSIGNALED(status)){          {
              printf("Parent: Abnormal termination,
                       signal(%d)\n",
                       WTERMSIG(status));
               exit(EXIT_SUCCESS);
          }
       } /* while */
    } /* parent */   return EXIT_SUCCESS;
} /* main */
```

Process Data

Process Data

- Since a child process is a **copy** of the parent, it has copies of the parent's data.
- A change to a variable in the child will **not** change that variable in the parent.

Example (globex.c) (1 of 3)

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

int globvar = 6;
char buf[] = "stdout write\n";

int main(void)
{
    int w = 88;
    pid_t pid;
```

continued

(2 of 3)

```
write( 1, buf, sizeof(buf)-1 );
printf( "Before fork()\n" );

if( (pid = fork()) == 0 ) {
      /* child */
      globvar++;
      w++;
}
else if( pid > 0 )          /* parent */
      sleep(2);
else
      perror( "fork error" );

printf( "pid = %d, globvar = %d, w = %d\n",
                  getpid(), globvar, w );
return EXIT_SUCCESS;
} /* end main */
```

Output (3 of 3)

```
$ globex
stdout write     /* write not buffered */


  Before fork()
pid = 430, globvar = 7, w = 89  /*child chg*/
pid = 429, globvar = 6, w = 88  /* parent no
                                   chg */
------------------------------------
$ globex > temp.out
$ cat temp.out
stdout write
Before fork()
pid = 430, globvar = 7, w = 89
Before fork() /* fully buffered */
   pid = 429, globvar = 6, w = 88
```

Process File Descriptors

Process File Descriptors

A child and parent have copies of the file descriptors, but the R-W pointer is maintained by the system:

the R-W pointer is shared

This means that a **`read()`** or **`write()`** in one process will affect the other process since the R-W pointer is changed.

Example: File used across processes (1 of 5) (shfile.c)

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

void printpos(char *msg, int fd);
void fatal(char *msg);

int main(void)
{ int fd;        /* file descriptor */
  pid_t pid;
  char buf[10]; /* for file data */
    :
```

continued

Example: File used across processes (2 of 5)

```
if ((fd=open("data-file", O_RDONLY)) < 0)
{  perror("error on open");
}
   read(fd, buf, 10);  /* move R-W ptr */

   printpos( "Before fork", fd );

   if( (pid = fork()) == 0 )
   {  /* child */
         printpos( "Child before read", fd );
         read( fd, buf, 10 );
         printpos( " Child after read", fd );
   }
```

continued

Example: File used across processes (3 of 5)

```
else if( pid > 0 )
    {    /* parent */
         wait((int *)0);
         printpos( "Parent after wait", fd );
    }
else
    perror( "fork" );
}
```

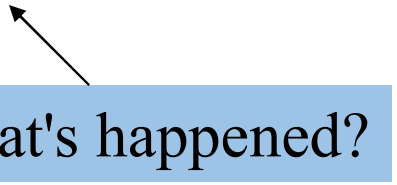
continued

Example: File used across processes (4 of 5)

```
void printpos( char *msg, int fd )
/* Print position in file */
{
   long int pos;
   if( (pos = lseek(fd, 0L, SEEK_CUR) ) < 0L )
       perror("lseek");
   printf( "%s: %ld\n", msg, pos );
}


   /* 0L = Long constant.
     Changes the file position by zero bytes.
     Then return the new Current Position. */
```

Output (5 of 5)

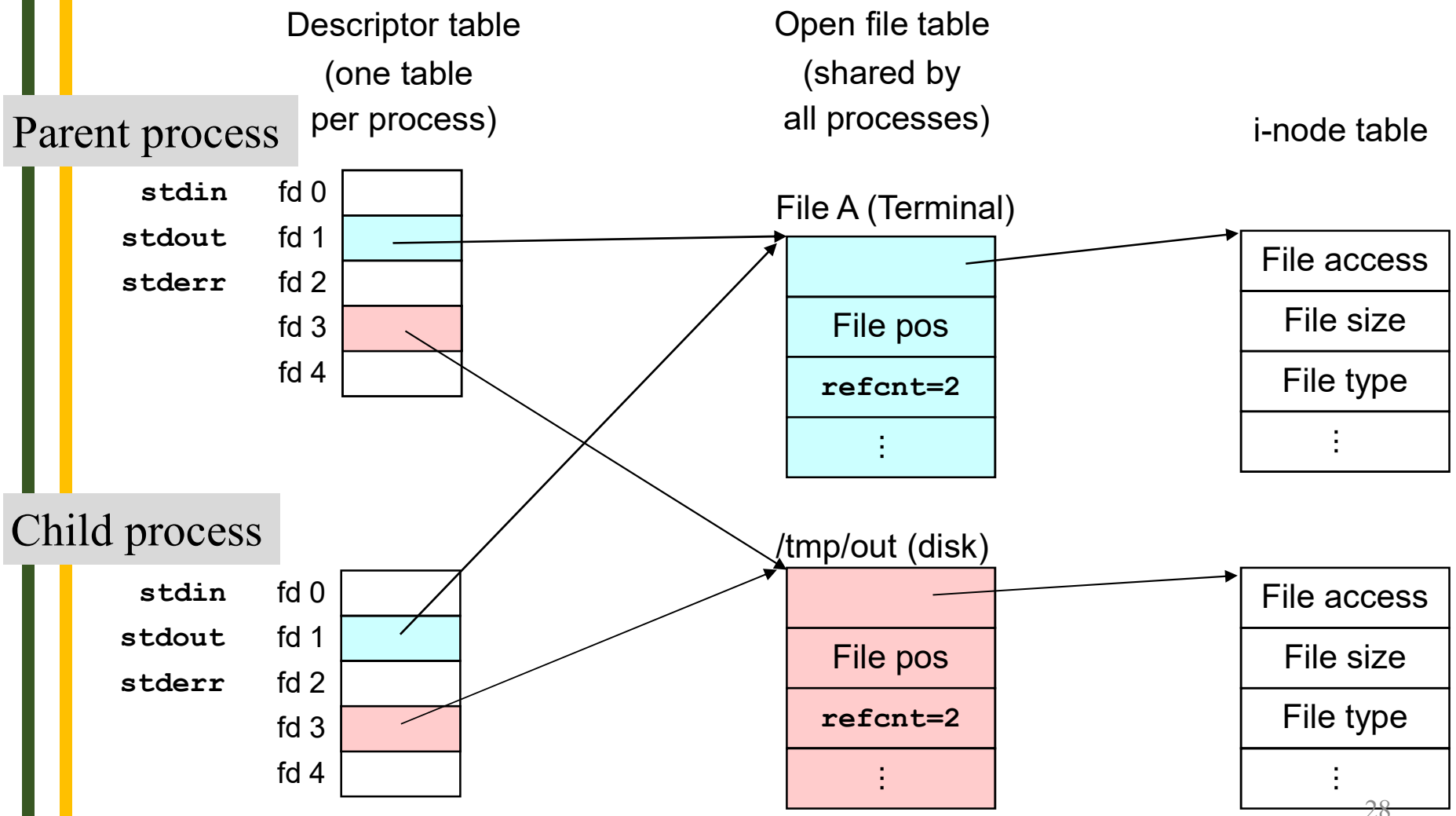
```
$ shfile

Before fork: 10
Child before read: 10
Child after read: 20
Parent after wait: 20
```



Parent and child both use the same file descriptor, so the current position of *fd* changed in both.

Child process inherits its parent's open files

After fork() call: Child's descriptor table is the same as parent's, and +1 to each refcnt



Parent-Child relationship with the file.

- The parent opens the file with the fd.
- The child inherits the open file from the parent.
- The child can close the file, but it will still be open by the parent.

Special Exit Cases

Special Exit Cases

Two special cases:

1) A child exits when its parent is not currently executing **`wait()`**
   the child becomes a ***zombie***
   `status` data about the child is stored until the parent does a **`wait()`**

2) A parent exits when 1 or more children are still running
   children are adopted by the system's initialization process **(`/etc/init`)**
   It can then monitor/kill them

10-UNIX
fork & wait

The End