System and Device Programming

Non-Blocking I/O

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Non-Blocking I/O

- Standard I/O functions are time-expensive
  - Opening a file can block until some condition occurs
    - Until the unit is attached to the system
    - Writing on a FIFO when no other process has the FIFO open for reading
  - Reads and writes can block the caller forever
    - A read is blocking if data is not present
      - Example: read from stdin or a pipe
    - A write is blocking if the device is full
      - Examples: write to a disk or a pipe
    - A read/write can be blocking if record/file locking is activated

See unit u06

See section u04s06

Non-Blocking I/O

- Non-blocking I/O lets us issue an operation and not have it blocked forever
  - If the operation cannot be completed
  - The call returns **immediately** with an error noting that the operation would have blocked
- There are two ways to specify non-blocking I/O for a given descriptor

Strategy 1

```
int fd;
...
fd = open (name, O_WRONLY | ... | O_NONBLOCK );
```

- The first strategy is to define the non-blocking option when the file is opened
  - Call **open** to get the descriptor and specify the **O_NONBLOCK** flag
  - The nonblocking mode will be set for opening and all subsequent I/O operations

Strategy 2

```
#include <fcntl.h>

int fcntl (int fd, int cmd, ... /* int arg */ );
```

- The second strategy can be adopted when a descriptor is already open
  - We can call function **fcntl** to turn-on the **O_NONBLOCK** file status flag
- Function **fcntl** has a more general use, as it can change the properties of a file that is already open

File Properties

- Parameters
  - **fd** specifies the file
  - **cmd** indicates the operation
    - Function fcntl can be used for 5 different operations
  - **arg** is either an integer or a pointer (in file locking)
- The return value is
  - The file descriptor flag, in case of success
  - The value -1, in case of error

```
int fcntl (int fd, int cmd, ... /* int arg */ );
```

File Properties

- Allowed operations (through cmd)

cmd	Purpose
F_GETFL or F_SETFL	Get/set file status flags
F_DUPFD or F_DUPFD_CLOEXEC	Duplicate an existing descriptor
F_GETFD or F_SETFD	Get/set file descriptor flags
F_GETOWN or F_SETOWN	Get/set asynchronous I/O ownership
F_GETLK, F_SETLK, or F_SETLKW	Get/set record locks

```
int fcntl (int fd, int cmd, ... /* int arg */ );
```

File Properties

- Allowed operations (through cmd)

cmd	Purpose
F_GETFL or F_SETFL	Get/set file status flags

- To modify the file status flags, we must
  - Fetch the existing flag value
  - Modify it as desired
  - Set the new flag value
- Notice that we cannot directly modify them as we can turn off flag bits that were previously set

```
int fcntl (int fd, int cmd, ... /* int arg */ );
```

Example

Set file status flag (wrapper function)

- Fetch the existing flag value and modify (set) it as desired

```
#include <errno.h>
#include <fcntl.h>

void set_fnctl (int fd, int flags) {
  int val;
  if ((val = fcntl(fd, F_GETFL, 0)) < 0)
    { ... error ... }

  val |= flags;

  if (fcntl(fd, F_SETFL, val) < 0)
    { ... Error ...}
}
```

File descriptor

File flag, e.g., O_NONBLOCK

Get flags

Set the desire flag

Set new flag status for the descriptor fd

Example

Clear file status flag (wrapper function)

- Fetch the existing flag value and modify (reset, clear) it as desired

```
#include <errno.h>
#include <fcntl.h>

void clr_fnctl (int fd, int flags) {
  int val;
  if ((val = fcntl(fd, F_GETFL, 0)) < 0)
    { ... error ... }

  val &= ~flags;

  if (fcntl(fd, F_SETFL, val) < 0)
    { ... Error ...}
}
```

File descriptor

File flag, e.g., O_NONBLOCK

Get flags

Clear the desire flag

Set new flag status for the descriptor fd

Example

- Use a nonblocking write operation to avoid waiting for the output termination

```
#define N 1000000
...
char buf[N];
int ntowrite, nwrite;
char *ptr;
...
ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
fprintf(stderr, "read %d bytes\n", ntowrite);
...
set_fnctl(STDOUT_FILENO, O_NONBLOCK);
```

Read a lot of data from stdin

Read is blocking

Set stdout to nonblocking

Example

```
ptr = buf;
while (ntowrite > 0) {
  nwrite = write (STDOUT_FILENO, ptr, ntowrite);

  fprintf (stderr, "nwrite = %d\n", nwrite);
  if (nwrite > 0) {
    ptr += nwrite;
    ntowrite -= nwrite;
  }

 ... Do somenthing else useful ...

}

clr_fnctl(STDOUT_FILENO, O_NONBLOCK);
```

Try to write a lot of data to stdout

Display output status

Do some other job

Clear stdout from nonblocking

Example

If the standard output is a regular file, we expect the write to be executed once

```
➢  ./a.out < intFile.txt > outfite.txt
read 1000000 bytes
nwrite = 1000000
➢  ls -l outFile.txt
-rw-rw-r-- 1 sar 1000000 Apr 1 13:03 outFile.txt
```

If the standard output is a terminal, we expect the write to return a partial count sometimes and an error at other times

```
➢  ./a.out < intFile.txt 2> stderr.txt
➢  cat stderr.txt
read 1000000 bytes
nwrite = 999
nwrite = -1
nwrite = 1001
nwrite = -1
nwrite = 1002
...
```