ISA 563: Fundamentals of Systems Programming

Advanced IO

April 9, 2012

Non-blocking IO

- Data processing can be much faster than data access
  - Waiting for IO to finish can be time consuming, and may not even finish
- Reads from files can block the caller forever if data is not present:
  - Reading from a FIFO with no input
- Distributed processing complicates things further:
  - Remote host could be down
  - Remote host could be busy
  - ...

Non-blocking IO (cont'd)

- In blocking IO, a request is blocked until it is satisfied.
- In non-blocking IO, a request returns immediately with apprioate return code.
- Ways to open a file for non-blocking IO:
  - pass O_NONBLOCK flag when opening file
  - call fcntl on an open file descriptor to turn on O_NONBLOCK status flag

Demo

non_blocking.c

Record Locking

- What happens when two people edit the same file at the same time?
  - File content will depend on the last process that write to that file (on most Unix/Linux systems)
- Need a way to get exclusive access:
  - Use some kind of inter-process locking mechanism, e.g., semaphores
  - Lock down the file altogether using flock (2)
- File-level locking can be inefficient
  - Two processes could be using two different portions of a file at the same time

Record Locking (cont'd)

- Record locking allows multiple processes to lock different regions of a file at the same time
- The kernel itself has no notion of “records” in a file:
  - Kernel only recognizes byte-ranges within a file
- POSIX chose to standardize on fcntl() for record locking (or more precisely byte-range locking)

flock (2)

- flock (2) applies or removes an advisory lock on an open file.
  - A file cannot be locked if any one of the locks is exclusive

```
#include <sys/file.h>

int flock(int fd, int operation);

Operation is one of:

  LOCK_SH: place a shared lock
  LOCK_EX: place an exclusive lock
  LOCK_UN: remove an existing lock held by process
```

fcntl(2)

```
#include <fcntl.h>
int fcntl(int filedes, int cmd, ... /* struct flock *flp */)
```

```
cmd is on of F_GETLK, F_SETLK, or F_SETLKW
```

```
struct flock {
     short l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
     off_t l_start;  /* offset in bytes, relative to l_whence */
     short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
     off_t l_len;    /* length, in bytes; 0 means lock to EOF */
     pid_t l_pid;    /* returned with F_GETLK */
   };
```

fnctl (2) permission rules

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

(Image courtesy of Advanced Programming in the Unix Environment)

Demo: File locking vs. Record locking

flock.c
rlock.c

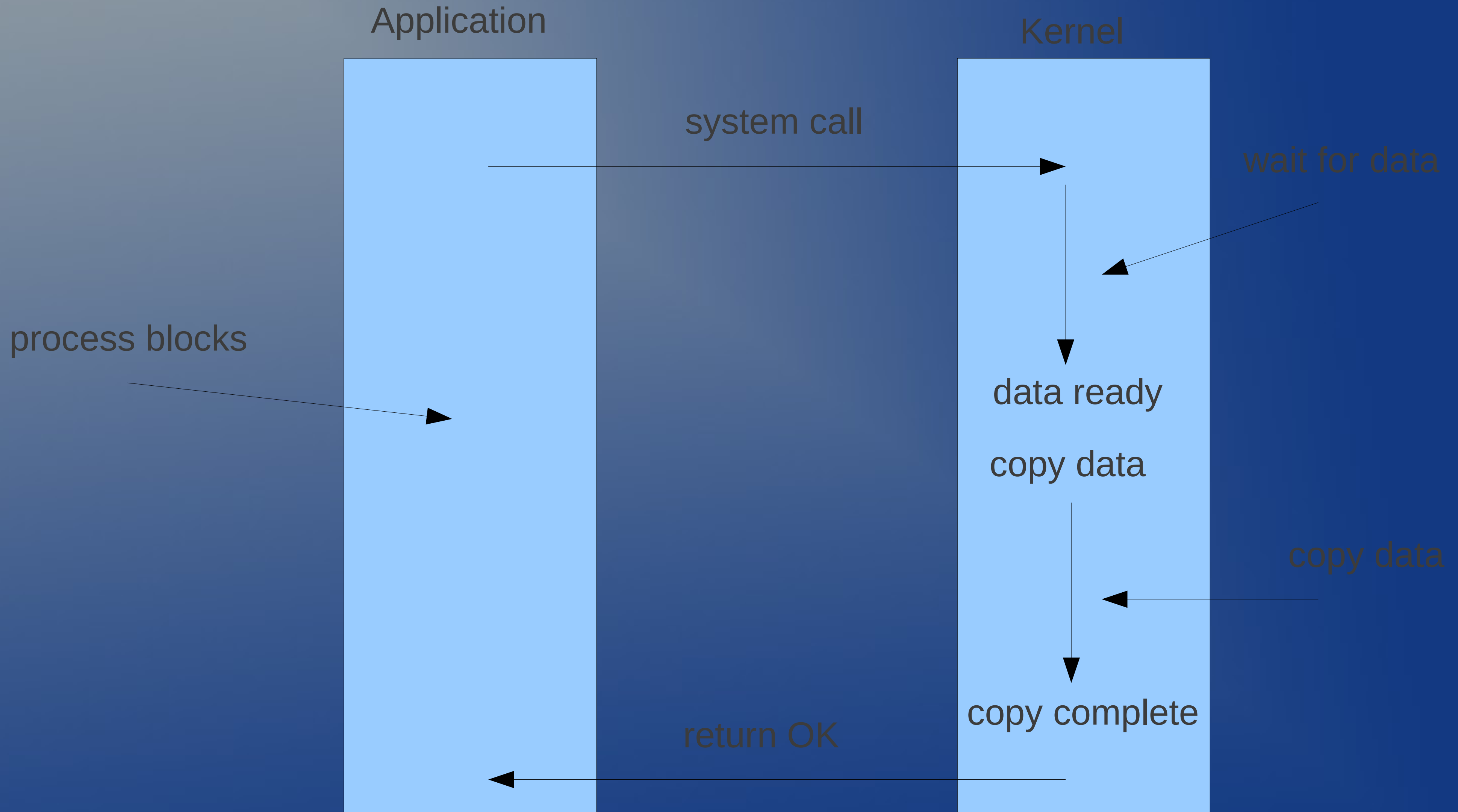
Multiple IO Scenarios

- A process can be handling multiple file descriptors concurrently:
  - A server handling multiple clients concurrently
  - A server that handles multiple services
  - A server using both TCP and UDP
  - A client handling both user input and network IO at the same time

I/O Models

- I/O models available under Unix/Linux systems:
    - blocking I/O
    - nonblocking I/O
    - I/O multiplexing
    - signal driven I/O
    - asynchronous I/O

Blocking I/O



Solutions for Handling Multiple IO Concurrently

- Spawn a new process [using fork()]
  - process overhead, signal handling
- Use multiple threads
  - has to handle synchronization
- Use non-blocking IO
  - has to keep polling, wastes CPU
- Use asynchronous IO
  - cannot descriminate between multiple file descriptors using AIO
- IO Multiplexing
  - complicates code somewhat

IO Multiplexing

- A *single* process handles multiple file descriptors
  - If no file descriptor is ready, process blocks
  - If any file descriptor is ready, process handles that file descriptor
    - Process still waits for kernel to copy data to user space
- Process can choose from following options:
  - indefinite blocking
  - immediate return
  - return if nothing is ready until a timeout happens

IO Multiplexing Functions

```
#include <sys/select.h>
int select(int maxfdp1, fd_set *readfds, fd_set *writefds,
fd_set *exceptset, const struct timeval *timeout);
```

```
#include <sys/select.h>
int pselect(int maxfdp1, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, const struct timespec *restrict tsptr,
const sigset_t *restrict sigmask);
```

```
#include <poll.h>
int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

select() system call

- select() allows a program to monitor multiple file descriptors.
- select() blocks until one the following occurs:
  - any of the descriptors in the readset becomes ready for reading
  - any of the descriptors in the writeset becomes ready for writing
  - any of the descriptors in exceptset has an exception condition pending
  - a timeout value was set and is expired

Macros Supporting select()

- Main data type: `fd_set`
  - `fd_set fds;`
- Macros:
  - `FD_ZERO(&fds) // initialize set, all bits off`
  - `FD_SET(7, &fds) // turn on bit for fd 7`
  - `FD_ISSET(3, &fds) // check if fd 3 is turned on`
  - `FD_ZERO(5, &fds) // turn off bit for fd 5`

Timeouts for select()

- Time struct data structure:

```
struct timeval {
   long tv_sec;
   long tv_usec;
}
```

- Three possibilities based on time value:
  - Wait forever: happens if a NULL pointer is passed
  - Wait up to fixed amount of time: struct specifies a non-zero time
  - Do not wait at all: struct specifices a zero value

Demo

stdin_timeout.c

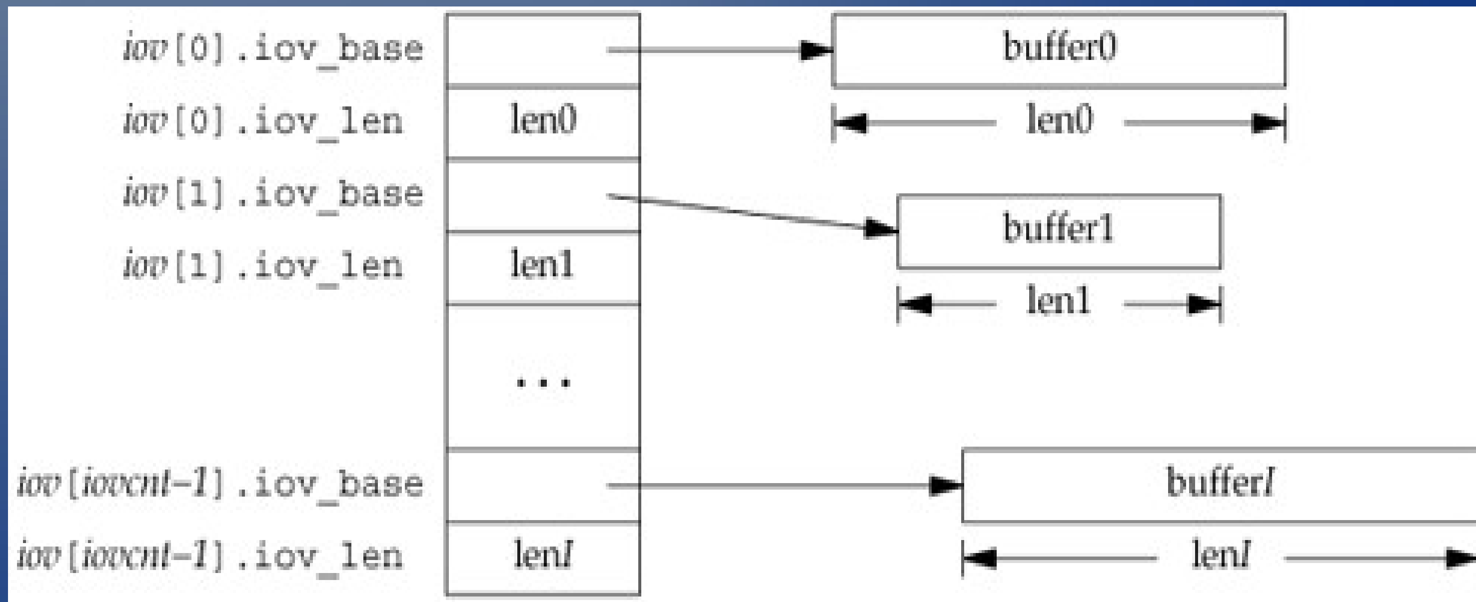
Demo

multiplexed_echo_server.c

readv() and writev() functions

```
ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);
ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);

struct iovec {
     void   *iov_base;
     size_t  iov_len;
   };
```



(Image courtesy of Advanced Programming in the Unix Environment)

Memory-mapped IO