Real-Time Programming TI00AA55

Lecture 8 - 18.04.2015

Jarkko.Vuori@metropolia.fi

Terminal settings

- Slides 1 – 8 handle terminal i/o
  - They are not studied in the class
  - Knowing these things can be useful, because they are applicable for example in serial ports
- Terminal can be in two different modes:
  - Canonical mode (input is processed as lines)
  - Non-canonical mode (no lines are constructed from characters)
- Remark. Line processing is not same than buffering at standard library level
  - We are now talking about i/o on the kernel level
- If terminal is in canonical mode, system call read returns only when new line is received (or when interrupted by a signal)
- Terminal is in canonical mode by default
- POSIX-defines 11 special characters that have special treatment in canonical mode (for example CR, EOF, ERASE, INTR, KILL (line erase char), START (XON), STOP (XOFF)
- The size of input queue (type ahead buffer) is MAX_INPUT
- The maximum size of canonical line is MAX_CANON
- The input queue can be flushed with the function `tcflush` (see an example later)

Structure termios

- To control terminal settings we need structure termios:

```
struct termios {
  tcflag_t  c_iflag; //input flags
  tcflag_t  c_oflag; //output flags
  tcflag_t  c_cflag; //control flags
  tcflag_t  c_lflag; //local flags
  cc_t      c_cc[NCCS]; //control
                        //characters
};
```

- The terminal settings can be get and set with functions

```
<termios.h>
int tcgetattr(int filedes,
struct termios *pterm_struct);
int tcsetattr(int filedes, int opt,
const struct termios *pterm_struct);
```

- Options for parameter opt
  - TCSANOW   Modifications takes effect immediately
  - TCSADRAIN Modifications takes effect, when all output is delivered
  - TCSAFLUSH As previous but additionally all unread input is flushed (ignored)
- Function `tcgetattr` stores the current settings in the struct termios
- Function `tcsetattr` sets the terminal settings according the contents of struct termios
  - Function `tcsetattr` returns 0 (OK), if it succeeded to set at least one flag from the structure termios!

Flags in termios structure

- One flag can be a bit or a bit field
- We need masks to get values of flags from the structure members
- Masks and possible values have defined names
- For example, the number of bits in the character can be accessed using the mask CSIZE
  - Possible values in this field are CS5, CS6, CS7 or CS8
- The flags in the members of structure termios are presented in the figures 18.3 – 18.6 in the book (Stevens & Rago)

- Example 1. Change the number of bits in a character to 8

```
struct termios term;
tcgetattr(STDIN_FILENO, &term);
term.c_cflag = term.c_cflag & ~CSIZE;
term.c_cflag = term.c_cflag | CS8;
tcsetattr(STDIN_FILENO, TCSANOW, &term);
```

- Example 2. Remove echo and canonicality

```
struct termios settings, old_settings;
tcgetattr(STDIN_FILENO, &settings);
old_settings = settings;
settings.c_lflag= settings.c_lflag &
~(ICANON|ECHO);
tcsetattr(STDIN_FILENO, TCSANOW,
&settings);
      ...
tcsetattr(STDIN_FILENO, TCSANOW,
&old_settings);
```

More ”terminal functions”

- Speed setting functions
```
speed_t cfgetispeed(const struct
termios *tp);
speed_t cfgetospeed(const struct
termios *tp);
int cfsetispeed(struct termios *tp,
speed_t speed);
int cfsetospeed (struct termios *tp,
speed_t speed);
```
  - Each speed is defined as a constant:
  - for example B2400, B9600, B19200, B38400
- All the terminal settings presented now or earlier can be done using command stty instead of system calls
  - You can display all terminal settings using command line command stty –a or stty -F/dev/ttyS0 –a for serial port S0
  - Echo and canonical mode can be removed with command stty –echo –icanon.

- Line control functions
```
<termios.h>
int tcdrain(int filedes);
```
  - Function waits until all output is delivered
- `int tcflow(int filedes, int action);`
  - This function is used for flow control (XON/XOFF)
- `int tcflush(int filedes, int queue);`
  - Function flushes the output queue a well as input queue. Parameter queue can be
    - TCIFLUSH (Input queue)
    - TCOFLUSH (Output queue)
    - TCIOFLUSH (Both queues)

Other functions

- The name of controlling terminal of the process can be asked with a function ctermid:

```
<stdio.h>
char *ctermid(char *ptr);
```

- Example 1

```
char
control_terminal_id[L_ctermid];
ctermid(control_terminal_id);
printf("The name of controlling
terminal is %s\n",
control_terminal_id);
```

- Example 2.

```
printf("The name of controlling
terminal is %s\n",
ctermid(NULL));
```

- Asking the name of what ever terminal that is open in a file descriptor:

```
<unistd.h>
char *ttyname(int filedes);
```

  – Function returns a pointer to the device name or NULL- pointer if error (for example if the device type is not ”a character special file” (see transparencies in part 3))

- Example

```
int fd;
  ...
printf("Term dev name is %s\n",
ttyname(fd));
```

Timeout in non-canonical mode

- We can use timeout in read operation from terminal. This means that read function returns if
  - A certain amount of bytes are read or
  - A certain amount of time has elapsed
- Two elements in the array c_cc in the termios structure are used to set the read mode with time out
- The first value is called MIN and the second TIME
- MIN value is stored at index VMIN and TIME value at index VTIME
  - The time unit is 1/10 seconds

- Case 1. MIN > 0 and TIME > 0
  - Function read waits for the first character
  - When one character is received the timer is activated
  - Read returns when all characters are read or timer expires
- Case 2. MIN > 0 and TIME = 0
  - Read returns when all characters are read
- Case 3. MIN == 0 and TIME > 0
  - Timer is activated when read is called
  - Read returns when one character is read or timer expires. The return value is 0, if no characters were read.
- Case 4. MIN == 0 and TIME == 0
  - Read returns always immediately
  - Function returns characters that are ready in the input queue

Example: polling keyboard

```
#include <stdio.h>
#include <unistd.h>
#include <termios.h>
#include <stdlib.h>

int main(void) {
  char chr;
  int result;
  struct termios term;

  tcgetattr(STDIN_FILENO, &term);
  term.c_lflag &= ~ICANON;
  term.c_cc[VMIN]  = 0;
  term.c_cc[VTIME] = 1;  // or 0 works too
  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term)< 0)
    fprintf(stderr, "We failed to change terminal settings\n");

  // Polling the keyboard
  while (1) {
    result =read(STDIN_FILENO, &chr, 1);
    if (result ==1) {
      //Do something with input
      printf("Character %c was read\n", chr);
    }
    // Do something "more important"
    printf("We are working all the time\n");
  }
}
```

Problem when waiting for several devices

- We return back to our main problem where real-time system needs to give "immediate" responses to several inputs
  - We continue to use the same example that was a simple chat program
- The problem is that the program should read input from two (or more) separate devices
  - This is the situation in a simple bi-directional communication program between two computers
  - We need to read input from the keyboard and input from the communication line
  - The problem is that the program shall not block on either of the read operation
- We still repeat the non working solution as a rpetition

```
char chr_from_kb, chr_from_line;
int fd_kb, fd_line, fd_display;
...
while (...) {
  read(fd_kb, &chr_from_kb, 1);//must not block
  write(fd_line, &chr_from_kb, 1);
  read(fd_line, &chr_from_line, 1);//must not
block
  write(fd_display, &chr_from_line, 1);
}
```

- We have already seen how this problem can be solved in two different ways:
  - Divide the work into two different processes
    - Then you can use blocked input in both read operations
  - Instead of dividing the program to different processes we can use polling (but this is not an efficient solution)
    - This is possible using non blocking read

New solutions

- Other solutions are:
  1. Use a so called i/o multiplexing
  2. We can use asynchronous i/o
     - This means that we ask kernel to send a signal when input is available
  3. Use threads
- In this part we will see how to use multiplexed i/o and asynchronous i/o
  - We are going to study threads later
- We are not able to wait for one descriptor as blocked
- The basic idea of multiplexed i/o is to wait for all file descriptors at the same time as blocked
- When one (or more) of the descriptors is ready for read or write, the program returns and it is possible to do those operations (without blocking) for the descriptors that were ready
- The function select works in this way. It is presented on the next page

I/O Multiplexing

- Function `select` makes it possible to wait for multiple file descriptors simultaneously (for example descriptors fd_kb and fd_line in the previous example)
- The function `select` returns if one of the waited descriptors is ready (so that input is available or output is possible)
- When `select` has returned, it is guaranteed that the next i/o (on the descriptor that `select` tells to be ready) does not block the program
- It is possible to set timeout for `select` function too so that it returns after timeout although no descriptor is ready for i/o

- Function select

```
<sys/types.h>  <sys/time.h>
<unistd.h>
int select(int maxfdplus1, fd_set
*readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval
*timevalptr);
```

  - Function returns the amount of descriptors that are ready for i/o or 0, if timeout and -1, if error
  - Parameter maxfdplus1 indicates the value of largest descriptor plus one
  - The next parameter (readfds) indicates the descriptors that we want to wait for reading
  - Parameter writefds indicates the descriptors whose readiness for writing we want to wait for
  - Parameter exceptfds indicates the descriptors whose exceptions we want to wait for

I/O Multiplexing (cont.)

- The last parameter for function select is pointer to the structure struct timeval. The definition of structure is:

```
struct timeval {
  long tv_sec;   // seconds
  long tv_usec;  // microseconds
};
```

- Case 1. timevalptr == NULL
  - select blocks until file descriptor is ready (may be forever)
- Case 2. If seconds and microseconds are both 0 select function does not wait at all (polling)
- Case 3. If seconds or microseconds field > 0, the select waits a specified time
- Function `select` waits for all descriptors given in three parameters
  - If at least one descriptor becomes ready (input is available, output is possible or exception state is on), select returns
  - The number of ready descriptors is a return value
  - Select has modified the parameters readfds and writefds so that they indicate the descriptors that are ready for reading or writing
- If `select` has returned because of timeout, and no descriptors are ready, the return value is 0
- Function `select` returns -1 if error (for example, if signal has interrupted it, return value is -1 and errno is EINTR)

Macros for type fd_set

- Type fd_set is type of file descriptor set
  - It can be implemented as a bit set containing one bit for each descriptor
- Four macros are defined for this type to handle file descriptor set (manipulate bits in the set):
  - FD_ZERO(fd_set *fdset);
  - FD_SET(int fd, fd_set *fdset);
  - FD_CLR(int fd, fd_set *fdset);
  - FD_ISSET(int fd, fd_set *fdset);
- Remark. Recall and compare to functions:

```
int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int
signo);
```

- Macro FD_ZERO removes all descriptors from the set (clears all bits in the descriptor set)
- Macro FD_SET adds a descriptor to the set (turns on the bit corresponding the descriptor given as a parameter)
- Macro FD_CLR removes a descriptor from the set (turns off the bit corresponding the descriptor given as a parameter)
- Macro FD_ISSET can be used to find out whether the descriptor is in the descriptor set or not (whether the bit corresponding the descriptor is on or off in the bit set)

The solution by multiplexing

Now we are able to solve the problem on the slide 9

```
int main(void) {
  char chr_from_kb, chr_from_line;
  int fd_kb, fd_line, fd_display;
  fd_set fdset;
  int n;

  while (...) {
    FD_ZERO(&fdset);
    FD_SET(fd_kb, &fdset);
    FD_SET(fd_line, &fdset);
    n = select(max(fd_kb, fd_line)+1, &fdset, NULL, NULL, NULL);
    if (n > 0) {
      if (FD_ISSET(fd_kb, &fdset)) {
        read(fd_kb, &chr_from_kb, 1);
        write(fd_line, &chr_from_kb, 1);
        // or do whatever necessary
      }
      if (FD_ISSET(fd_line, &fdset)) {
        read(fd_line, &chr_from_line, 1);
        write(fd_display, &chr_from_line, 1);
        // or do whatever necessary
      } // inner if
    } // outer if
  } // while
} // main
```

Basic idea of multiplexed i/o

- The basic idea of multiplexed i/o is often like in the following pseudo code:

```
while (1) {
Set all descriptors that you need to read from in the descriptor set readfds
Set all descriptors that you need to write to in the descriptor set writefds
Call function select
When function select returns
  – Find out what descriptors are in set readfds and read from them
  – Find out what descriptors are in set writefds and write to them
}
```

- Next we will move to the second method mentioned on the page 10 that is asynchronous i/o
- The basic idea of asynchronous i/o is that the operating system is asked to inform the process, when the file descriptor is ready to be read or ready to be written
- In the conventional unix way the request is done once and the kernel informs every time, when the file descriptor is ready
- In the new way the request is done separately for each i/o operation

Asynchronous i/o ("conventional way")

- First we learn the conventional UNIX way and then the new POSIX-method
- Asynchronous i/o means that the program does not wait for input or output (does not block)
  - The process is executing a program simultaneously and parallel with i/o
  - Operating system informs asynchronously the process when i/o is ready by sending a signal
  - Signal SIGIO is used for this purpose by default

- To make asynchronous i/o to work in practice we need to do the following things:
  - Install the signal handler for the signal SIGIO, where we read the input (write the output)
  - We have to tell the kernel what is the process we want the kernel to inform, when file descriptor is ready for i/o
    - This is done with the function `fcntl` (command F_SETOWN)
  - We have to set the device (file descriptor) in the asynchronous state (this means that we have to set file flag O_ASYNC)
    - This can be done with the function `fcntl` again (command F_SETFL)
- Point 3 actually means a request for the kernel that “Always when descriptor is ready, send a signal!”
- See an example on the next page

Example

```
void input_ready(int signo);
int main(void) {
  int flags;
  // install the signal handler for signal SIGIO
  signal(SIGIO, input_ready);
  // set the process that receives the signal
  fcntl(STDIN_FILENO, F_SETOWN, getpid());
  // set the device in asynchronous mode
  flags = fcntl(STDIN_FILENO, F_GETFL, 0);
  flags = flags | O_ASYNC;
  fcntl(STDIN_FILENO, F_SETFL, flags);
  while(1) {
    do_something();
    sleep(1);
  }
}
```

// to solve problem on page 8

```
read(fd_line, &chr_from_line, 1);
write(fd_display, &chr_from_line, 1)
```

```
void input_ready(int signo) {
  char input[80];
  int n;
  n = read(STDIN_FILENO, input, 80);
  if (n > 0) {
    input[n] = '\0';
    printf("\nInput was %s", input);
  }
}
```

// to solve problem on page 8

```
read(STDIN_FILENO, &chr_from_kb, 1);
write(fd_display, &chr_from_kb, 1)
```

Asynchronous i/o and terminal

- Asynchronous i/o mode can be used for all kind of files (for example sockets, pipes and terminals)
- When we are talking about asynchronous i/o of terminals, we have to separate canonical mode and non-canonical mode
- Non-canonical mode:
  - Signal is sent when there is one byte available in the input queue (type ahead buffer)
- Canonical mode:
  - Signal is sent when a canonical line is ready to read (in other words when enter is found)
    - The example on the previous page works this way
- Remark 1. The drawbacks of this conventional method are:
  - works only on input
  - it is difficult to manage with many descriptors (there is only one signal and signals are not queued)
  - the way is not standard (BSD)
- The new POSIX way (AIO) is presented after few pages

POSIX asynchronous i/o

- POSIX defines a new method for asynchronous i/o in POSIX:AIO extension
- The basic idea of asynchronous i/o is:
  1. The process initiates one read or write operation using functions `aio_read` or `aio_write`
     - They do not block
  2. The process continues executing the program and i/o happens parallel with the program execution
  3. The process is informed about the completion with the signal, or it can ask whenever the i/o is ready
- POSIX defines five new functions for asynchronous i/o:
  - `aio_read` initiates asynchronous read
  - `aio_write` initiates asynchronous write
  - `aio_error` returns error status
  - `aio_return` returns the number of bytes read or written
  - `io_cancel` cancels i/o in progress
- Functions `aio_read` and `aio_write` take a single parameter that is a pointer to the structure struct aiocb

```
<aio.h>
int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
```

- Remark! To use these functions you need to include <aio.h> and link the real-time library using switch -lrt in the gcc command line

struct aiocb

- The definition of the structure aiocb is:

```
struct aiocb {
  int             aio_fildes;  // file descriptor
  volatile void  *aio_buf;     // buffer address
  size_t          aio_nbytes;  // number of bytes
  off_t           aio_offset;  // file offset
  int             aio_reqprio; // request priority
  struct sigevent aio_sigevent;// signal # and value
  int             aio_lio_opcode;// listio operation
                             //(multiple i/o requests)
};
```

- The first three members are like parameters of `read` and `write` functions

- Field aio_sigevent (struct) specifies how the process is notified about the completion
  1. If the member sigev_notify of the structure aio_sigevent is SIGEV_NONE, no signal is generated
  2. If the member sigev_notify of the structure aio_sigevent is SIGEV_SIGNAL, kernel generates signal indicated by the member sigev_signo
- The definition of struct sigevent:

```
struct sigevent {
  int sigev_notify;       //notification type
  int sigev_signo;        //signal number
  union sigval sigev_value;//signal value
};
```

union sigval

- The definition of the union sigval is as follows:

```
union sigval {
  int   sival_int;
  void *sival_ptr;
};
```

- The third member sigev_value in the structure sigevent is the parameter value that is sent with the signal
  - This is used with the real-time signals
  - The handler functions of real-time signals have an additional parameter (in addition to the signal number)
  - This parameter can be used to pass extra information to the signal handler function
  - Because this parameter is union, it is possible to pass either integer (int) or pointer to any kind of data type (void*)
- We will study real-time signals a bit later

Functions aio_return and aio_error

- The progress of i/o can be monitored with `aio_error` function
  ```
  int aio_error(const struct aiocb
  *aiocbp);
  ```
- Function `aio_error` returns 0, when i/o is completed successfully. It returns EINPROGRESS if i/o-operation is not completed yet. If the operation failed, the function returns error code that corresponds the errno variable after read or write
- When operation is completed successfully function `aio_return` can be used to retrieve the number of bytes read or written
  ```
  <aio.h>
  ssize_t aio_return( struct aiocb
  *aiocbp);
  ```

- See an example on the following page that demonstrates
  1. how to start asynchronous i/o,
  2. how to do something parallel with i/o and
  3. how to wait for the completion of the i/o when we cannot continue anymore without the result from the i/o
  - This first example uses polling when waiting for the completion. This is not a good solution. The main purpose of this example is to demonstrate the aio-functions mentions above
- See another example that shows how to use signals to notify the process about the completion of the i/o from the link asynch_io_new_1.c

Example program

```
int main(void) {
    char chr;
    struct aiocb aiocb;
    int r, n;

    // fill i/o control block
    aiocb.aio_fildes = STDIN_FILENO;
    aiocb.aio_buf = &chr;
    aiocb.aio_nbytes = 1;
    aiocb.aio_offset = 0;
    aiocb.aio_reqprio = 0;
    aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
    aiocb.aio_lio_opcode = 0;
    // initiate i/o
    aio_read(&aiocb);
    // do something parallel with the i/o
    sleep(2);  // demonstrates doing something
    // wait for the completion of the i/o
    while (aio_error(&aiocb) == EINPROGRESS);
    // test for error
    if ((r = aio_error(&aiocb)) != 0) {
       printf("%s\n", strerror(r));
       exit(0);
    }
    else // i/o was successful
       n = aio_return(&aiocb);
    if (n > 0) // we really read something
       printf("i/o completed. The result was %c (code %d)\n", chr, chr);
    else // n == 0 (end of file)
       printf("End of file received\n");
    return 0;
}
```

The main purpose of this example is to illustrate the usage of functions aio_error and aio_return

A bit more about non-blocked i/o 1

- Let's further compare asynchronous i/o and non-blocked i/o
- Both have common that i/o-instruction does not block but returns immediately. The difference is that
  1. In non-blocked i/o the i/o is completed or not, when the instruction returns
     - If the i/o is not completed, the instruction is in a way "forgotten"
  2. When instruction returns in asynchronous i/o, there is no knowledge how or when the i/o will be finished
- In non-asynchronous i/o also the **output operation can block** if the descriptor is not set on non-blocked mode
- Remember that we can set the descriptor to the non-blocking mode using the flag O_NONBLOCK
  - When file is opened (function open)
  - Later, using function `fcntl` (command F_SETFL)
- Then the functions `read` and `write` would return immediately with the value -1 and errno set to EAGAIN (POSIX.1) if they otherwise would block
- The i/o-operation would block for example if
  - data is not available in input (terminal, pipe, socket,…)
  - output cannot be done immediately (there is no room in the pipe, internal buffers are full for a slow device)

A bit more about non-blocked i/o 2

- Example:
  - If we want to output a big file (e.g. 1 000 000 bytes) to the display, the system call `write` waits until all bytes are written, if blocked i/o is used
  - In non-blocked mode, write moves as many bytes as it can to the system buffers and then returns, returning the number of bytes written (errno is 0)
    - If the process tries immediately again to write unwritten bytes, it probably returns with error return value -1 and with errno EAGAIN
- See the example program on the following page

- The program was run in certain environment. The results are:
  1. The first call of write function wrote 544635 bytes to the disk
  2. Then the program tried to do write operation several times and returned every time with the error EAGAIN
  3. After some failed trials the write operation succeeded so that 4095 bytes were written to the disk (even we had asked to write all unwritten bytes which is much more than 4095)
  4. Next the things mentioned in point 2 and 3 happened again
     - This continued until at the final step the program wrote 820 bytes, meaning that all bytes had been written

Example program

```
#define SIZE 1000000   // includes are missing because of ”lack of space”
char buffer[SIZE];
int main(void){
  char *p;
  int bytes_left, n, i, flags;
  FILE *logf;

  logf = fopen("log.txt", "w");
  // initialize array
  for (i = 0; i < SIZE; i++ )
    buffer[i] = 'A' + i % ('Z' - 'A' + 1);
    // set terminal to non-block mode
    flags = fcntl(STDOUT_FILENO, F_GETFL, 0);
    flags |=O_NONBLOCK;
    flags = fcntl(STDOUT_FILENO, F_SETFL, flags);
    // send big buffer to the display
    p = buffer;
    bytes_left = SIZE;
    while (bytes_left > 0) {
      n = write(STDOUT_FILENO, p , bytes_left);
      if (n >= 0) {
        bytes_left -= n;
        p          += n;
        fprintf(logf, "Bytes sent %d %c%c", n, 13, 10);
      }
      if (n < 0) {
        fprintf(logf, "Error, errno is %d %c%c", errno, 13, 10 );
        sleep(1);
      }
  }
  fclose(logf);
}
```