Objectives

- Ideas and Skills
  - Reading and changing settings of the terminal driver
  - Modes of the terminal driver
  - Nonblocking input
  - Timeouts on user input
  - Introduction to signals: How Ctrl-C work
- System calls
  - fcntl
  - signal

Three standard file descriptors

- **Standard input** is stream going into a program. The program requests data transfers by use of the read operation. Unless redirected, input is expected from the text terminal which started the program. The file descriptor for standard input is 0 (zero); the corresponding stdio.h variable is FILE *stdin
- **Standard output** is the stream where a program writes its output data. The program requests data transfer with the write operation. Unless redirected, standard output is the text terminal which initiated the program. The file descriptor for standard output is 1 (one); the corresponding stdio.h variable is FILE *stdout
- **Standard error** is another output stream typically used by programs to output error messages or diagnostics. The file descriptor for standard error is 2; the corresponding stdio.h variable is FILE *stderr

Common concerns of program interacting with terminals

For the programs that interacting with terminals, such as vi, pico, shells, and terminal games, their common concerns include

- immediate response to keys
- limited input set
- timeout on input
- resistant to Ctrl-C.

Mode of the terminal driver–an translate program

```
/* rotate.c : map a->b, b->c, .. z->a
 *   purpose: useful for showing tty modes
 */

#include   <stdio.h>
#include   <ctype.h>

int main()
{
    int c;
    while ( ( c=getchar() ) != EOF ){
        if ( c == 'z' )
            c = 'a';
        else if (islower(c))
            c++;
        putchar(c);
    }
}
```

Mode of the terminal driver–an translate program

- Backspace will erase characters we entered
- Characters appears on the screen as we type them
- The program do not receive any input until we press Enter key
- The Ctrl-C key discards input and stops the program.

Processing in the terminal driver

- Input editing
- Convert \r to \n
- Echo
- Control character handling (Special character handling).

Another experiment with rotate

```
$stty -icanon; ./rotate
$stty icanon;

$stty -icanon -echo; ./rotate
$stty icanon echo;
```

Modes of the terminal connection

There are three modes for terminal connection

- **Canonical mode**: The driver stores incoming characters in a buffer and only sends those buffered character to the program when the driver receives the Enter key or EOF key.
- **Noncanonical mode**: When buffering, and thus editing functions, is turned off, the connection is said to be in noncanonical mode.
- **raw mode** When all processing is turned off, the driver passes input directly to the program.

Noncanonical Mode

Noncanonical mode is specified by turning off the **ICANON** flag in the **c_lflag** field of the **termios** structure. Some special characters are not working. Look at the manpage of termios for those special characters.

Noncanonical mode offers special parameters called MIN and TIME for controlling whether and how long to wait for input to be available. The MIN and TIME are stored in elements of the c_cc array, which is a member of the struct termios structure. Each element of this array has a particular role, and each element has a symbolic constant that stands for the index of that element. VMIN and VMAX are the names for the indices in the array of the MIN and TIME slots.

- Macro: int VMIN

  This is the subscript for the MIN slot in the c_cc array. Thus,

termios.c_cc[VMIN] is the value itself.

The MIN slot is only meaningful in noncanonical input mode; it specifies the minimum number of bytes that must be available in the input queue in order for read to return.

- Macro: int VTIME

  This is the subscript for the TIME slot in the c_cc array. Thus, termios.c_cc[VTIME] is the value itself.

  The TIME slot is only meaningful in noncanonical input mode; it specifies how long to wait for input before returning, in units of 0.1 seconds.

Noncanonical Mode

The MIN and TIME values interact to determine the criterion for when read should return; their precise meanings depend on which of them are nonzero. There are four possible cases:

- Both TIME and MIN are nonzero.

  In this case, TIME specifies how long to wait after each input character to see if more input arrives. After the first character received, read keeps waiting until either MIN bytes have arrived in all, or TIME elapses with no further input.

  read always blocks until the first character arrives, even if TIME elapses first. read can return more than MIN characters if more than MIN happen to be in the queue.

- Both MIN and TIME are zero.

  In this case, read always returns immediately with as many characters as are available in the queue, up to the number

requested. If no input is immediately available, read returns a value of zero.

- MIN is zero but TIME has a nonzero value.

  In this case, read waits for time TIME for input to become available; the availability of a single byte is enough to satisfy the read request and cause read to return. When it returns, it returns as many characters as are available, up to the number requested. If no input is available before the timer expires, read returns a value of zero.

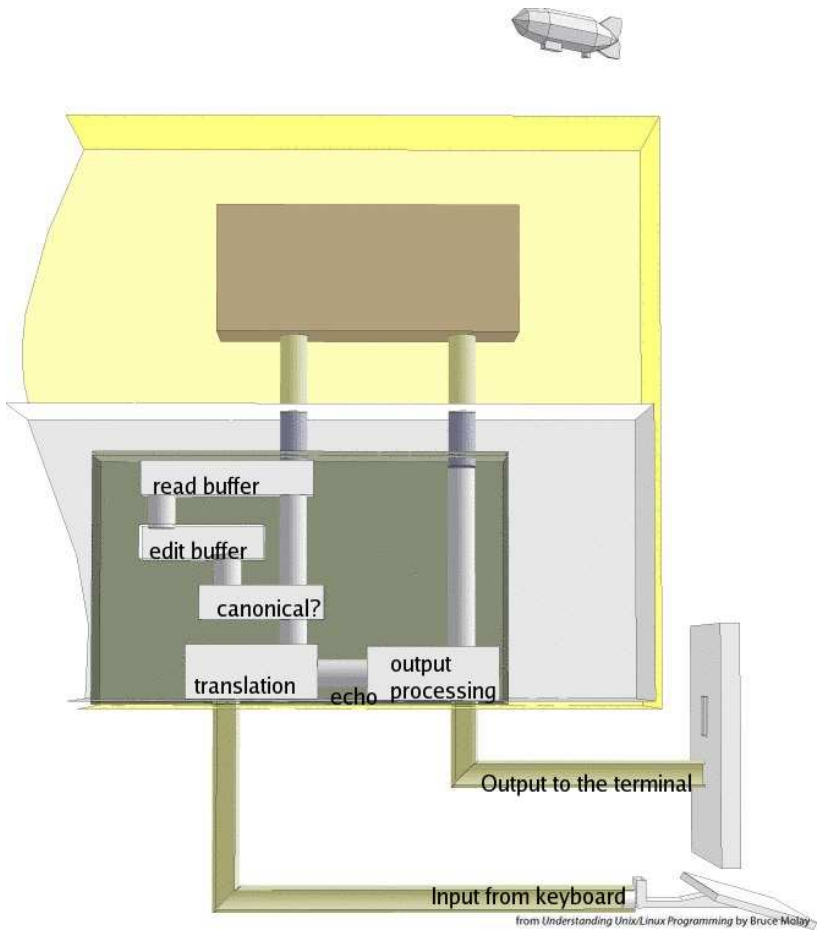
- TIME is zero but MIN has a nonzero value.

  In this case, read waits until at least MIN bytes are available in the queue. At that time, read returns as many characters as are available, up to the number requested. read can return more than MIN characters if more than MIN happen to be in the queue.

	MIN>0	MIN==0
TIME>0	A: read() returns [MIN, nbytes] before timer expires read() returns [1, MIN] if timer expires (TIME = interbyte timmer. Caller can block indefinitely)	C: read() returns [1, nbytes] before timer expires; read() returns 0 if timer expires (TIME = read timer)
TIME==0	B: read() returns [MIN, nbytes] when available. (Caller can block indefinitely.)	D: read() returns [0, nbytes] immediately.

Four cases for noncanonical input

Major components of the terminal driver



Writing a user program: play_again.c

```
#!/bin/sh
#
# atm.sh --a wrapper for two programs
#

while true
do
   echo "run a progam"        #run a program
   if ./play_again          #run our program
   then
      continue             #if "y" loop back
   fi
   break
done
```

Source code: play_again0.c

```
/* play_again0.c
 *purpose: ask if user wants another transaction
 *method: ask a question, wait for yes/no answer
 *returns: 0=>yes, 1=>no
 * better: eliminate need to press return
 */
#include        <stdio.h>
#include        <termios.h>

#define QUESTION        "Do you want another transaction"

int get_response( char * );

int main()
{
        int  response;

        response = get_response(QUESTION);       /* get some answer */
        return response;
}
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
```

```
 *  method: use getchar and ignore non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        printf("%s (y/n)?", question);
        while(1){
                switch( getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                }
        }
}
```

play_again1.c, immediate response

```
/* play_again1.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into char-by-char mode, read char, return result
 *      returns: 0=>yes, 1=>no
 *       better: do no echo inappropriate input
 */
#include        <stdio.h>
#include        <termios.h>

#define QUESTION        "Do you want another transaction"

main()
{
        int response;

        tty_mode(0);                            /* save tty mode */
        set_crmode();                           /* set chr-by-chr mode */
        response = get_response(QUESTION);      /* get some answer */
        tty_mode(1);                            /* restore tty mode */
        return response;
}
int get_response(char *question)
/*
```

```
 * purpose: ask a question and wait for a y/n answer
 *  method: use getchar and complain about non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        int input;
        printf("%s (y/n)?", question);
        while(1){
                switch( input = getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                        default:
                                printf("\ncannot understand %c, ", input);
                                printf("Please type y or no\n");
                }
        }
}

set_crmode()
/*
 * purpose: put file descriptor 0 (i.e. stdin) into chr-by-chr mode
 *  method: use bits in termios
```

```
 */
{
        struct termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting */
        ttystate.c_lflag        &= ~ICANON;     /* no buffering */
        ttystate.c_cc[VMIN]     =  1;           /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings */
}

/* how == 0 => save current mode,  how == 1 => restore mode */
tty_mode(int how)
{
        static struct termios original_mode;
        if ( how == 0 )
                tcgetattr(0, &original_mode);
        else
                return tcsetattr(0, TCSANOW, &original_mode);
}
```

play_again2.c, ignore illegal keys

```
#include <termios.h>

#define QUESTION "Do you want another transaction"

main()
{
        int response;

        tty_mode(0);                            /* save mode */
        set_cr_noecho_mode();                   /* set -icanon, -echo */
        response = get_response(QUESTION);      /* get some answer */
        tty_mode(1);                            /* restore tty state */
        return response;
}

int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 *  method: use getchar and ignore non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        printf("%s (y/n)?", question);
```

```
        while(1){
                switch( getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                }
        }
}
set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting */
        ttystate.c_lflag        &= ~ICANON;     /* no buffering */
        ttystate.c_lflag        &= ~ECHO;       /* no echo either */
        ttystate.c_cc[VMIN]     =  1;           /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings */
}
```

```
/* how == 0 => save current mode,  how == 1 => restore mode */
tty_mode(int how)
{
        static struct termios original_mode;
        if ( how == 0 )
                tcgetattr(0, &original_mode);
        else
                return tcsetattr(0, TCSANOW, &original_mode);
}
```

play_again3.c, Nonblocking input

```
/* play_again3.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into chr-by-chr, no-echo mode
 *            set tty into no-delay mode
 *            read char, return result
 *      returns: 0=>yes, 1=>no, 2=>timeout
 *       better: reset terminal mode on Interrupt
 */
#include        <stdio.h>
#include        <termios.h>
#include        <fcntl.h>
#include        <string.h>

#define ASK         "Do you want another transaction"
#define TRIES       3                                   /* max tries */
#define SLEEPTIME   2                                      /* time per try */
#define BEEP        putchar('\a')                          /* alert user */

main()
{
        int  response;

        tty_mode(0);                            /* save current mode */
```

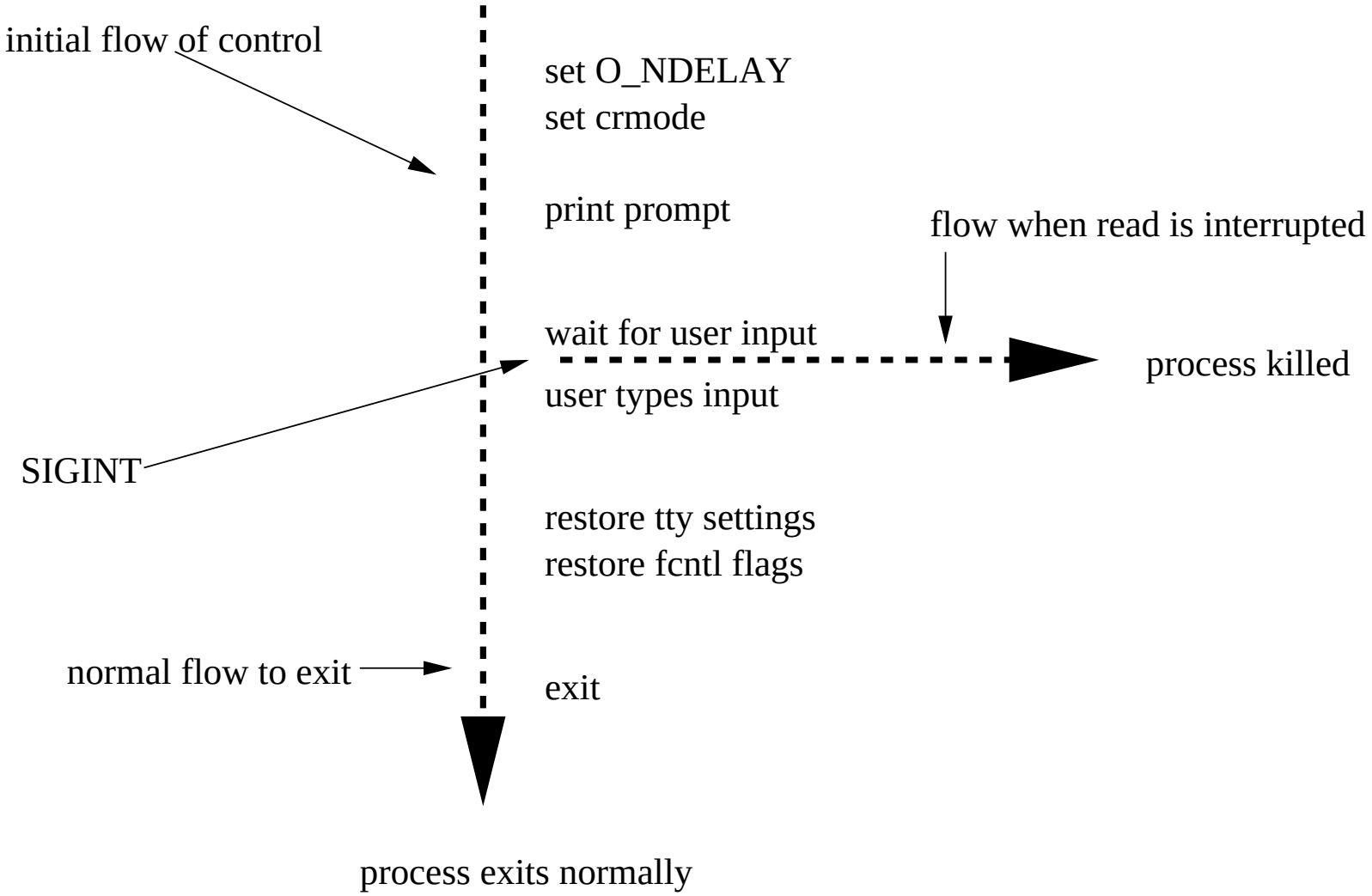
```
        set_cr_noecho_mode();                   /* set -icanon, -echo */
        set_nodelay_mode();                     /* noinput => EOF */
        response = get_response(ASK, TRIES);    /* get some answer */
        tty_mode(1);                            /* restore orig mode */
        return response;
}
get_response( char *question , int maxtries)
/*
 * purpose: ask a question and wait for a y/n answer or maxtries
 *  method: use getchar and complain about non-y/n input
 * returns: 0=>yes, 1=>no, 2=>timeout
 */
{
        int input;

        printf("%s (y/n)?", question);                  /* ask          */
        fflush(stdout);                                 /* force output */
        while ( 1 ){
                sleep(SLEEPTIME);                       /* wait a bit */
                input = tolower(get_ok_char());         /* get next chr */
                if ( input == 'y' )
                        return 0;
                if ( input == 'n' )
                        return 1;
```

```
                if ( maxtries-- == 0 )                  /* outatime? */
                        return 2;                       /* sayso */
                BEEP;
        }
}
/*
 *  skip over non-legal chars and return y,Y,n,N or EOF
 */
get_ok_char()
{
        int c;
        while( ( c = getchar() ) != EOF && strchr("yYnN",c) == NULL );
        return c;
}

set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting */
```

```
        ttystate.c_lflag    &= ~ICANON;           /* no buffering */
        ttystate.c_lflag    &= ~ECHO;             /* no echo either */
        ttystate.c_cc[VMIN] =  1;                 /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);       /* install settings */
}

set_nodelay_mode()
/*
 * purpose: put file descriptor 0 into no-delay mode
 *  method: use fcntl to set bits
 *   notes: tcsetattr() will do something similar, but it is complicated
 */
{
        int termflags;

        termflags = fcntl(0, F_GETFL);          /* read curr. settings */
        termflags |= O_NDELAY;                  /* flip on nodelay bit */
        fcntl(0, F_SETFL, termflags);           /* and install 'em */
}

/* how == 0 => save current mode,  how == 1 => restore mode */
/* this version handles termios and fcntl flags             */

tty_mode(int how)
```

```
{
        static struct termios original_mode;
        static int            original_flags;
        if ( how == 0 ){
                tcgetattr(0, &original_mode);
                original_flags = fcntl(0, F_GETFL);
        }
        else {
                tcsetattr(0, TCSANOW, &original_mode);
                fcntl( 0, F_SETFL, original_flags);
        }
}
```

A problem with play_again3



Signal handling– How does Ctrl-C work

- User presses Ctrl-C
- driver receives char
- char matches VINTR and ISIG is on
- driver calls signal system
- signal system sends SIGINT to process
- process dies

Signal handling– Sources of signals

- Users: special characters entered by the user
- Kernel: something is wrong or notify some events, segmentation error, divid-by-zero, illegal machine language command
- process: communicate with other processes using **kill** system call

For a list of the signals and the default action of each signal, try

```
man signal.h
```

signal.h

In Linux, /usr/include/signal.h defines 31 signals. Some of them are listed below

```
/* Signals.  */
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGILL          4       /* Illegal instruction (ANSI).  */
#define SIGTRAP         5       /* Trace trap (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGIOT          6       /* IOT trap (4.2 BSD).  */
#define SIGBUS          7       /* BUS error (4.2 BSD).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
#define SIGPIPE         13      /* Broken pipe (POSIX).  */
#define SIGALRM         14      /* Alarm clock (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
```

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Signal handling– three choices when receiving a signal

- accept the default action.

  The manpage of signal.h lists the default action for each signal
- ignore the signal

  ```
  signal(SIINT, SIG_IGN);
  ```
- specify a signal handler by

  ```
  signal(signum, functionname);
  ```

Catching a signal: A short sample program

```
/* sigdemo1.c - shows how a signal handler works.
 *            - run this and press Ctrl-C a few times
 */

#include        <stdio.h>
#include        <signal.h>

main()
{
        void    f(int);                 /* declare the handler */
        int     i;

        signal( SIGINT, f );            /* install the handler */
        for(i=0; i<5; i++ ){            /* do something else */
                printf("hello\n");
                sleep(1);
        }
}

void f(int signum)                      /* this function is called */
{
        printf("OUCH!\n");
}
```

Ignore a signal: A short sample program

```
/* sigdemo2.c - shows how to ignore a signal
 *            - press Ctrl-\ to kill this one
 */
#include        <stdio.h>
#include        <signal.h>
main()
{
        signal( SIGINT, SIG_IGN );

        printf("you can't stop me!\n");
        while( 1 ){
                sleep(1);
                printf("haha\n");
        }
}
```

The following two signals can't be blocked or ignored.

SIGKILL	Kill a process
SIGSTOP	Stop executing

Partial code of play_again4.c

```
main()
{
        int     response;
        void    ctrl_c_handler(int);

        tty_mode(0);                            /* save current mode */
        set_cr_noecho_mode();                   /* set -icanon, -echo */
        set_nodelay_mode();                     /* noinput => EOF */
        signal( SIGINT, ctrl_c_handler );       /* handle INT */
        signal( SIGQUIT, SIG_IGN );             /* ignore QUIT signals */
        response = get_response(ASK, TRIES);    /* get some answer */
        tty_mode(1);                            /* reset orig mode */
        return response;
}
void ctrl_c_handler(int signum)
/*
 * purpose: called if SIGINT is detected
 *  action: reset tty and scram
 */
{
        tty_mode(1);
        exit(2);
}
```

Some links about terminal control and programming

Terminal I/O FAQ

`http://www.erlenstar.demon.co.uk/unix/faq_4.html`

The GNU C Library

`http://www.gnu.org/software/libc/manual/html_node/`